

Windows PowerShell



NAJLEPSZE PRAKTYKI

Ed Wilson

Helion 

Tytuł oryginału: Windows PowerShell Best Practices

Tłumaczenie: Łukasz Piwko

ISBN: 978-83-283-0481-9

Authorized translation from the English language edition: WINDOWS POWERSHELL BEST PRACTICES; ISBN 0735666490; by Ed Wilson; published by Microsoft Press, a division of Microsoft Corporation, Inc.

Copyright © 2013 by Ed Wilson.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc., or Microsoft Press.

Polish language edition published by HELION S.A., under license and with the permission of Pearson Education, Inc. Copyright © 2015.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiegokolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz Wydawnictwo HELION dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz Wydawnictwo HELION nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Wydawnictwo HELION

ul. Kościuszki 1c, 44-100 GLIWICE

tel. 32 231 22 19, 32 230 98 63

e-mail: helion@helion.pl

WWW: <http://helion.pl> (księgarnia internetowa, katalog książek)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:

<ftp://ftp.helion.pl/przyklady/winpsp.zip>

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres

http://helion.pl/user/opinie/winpsp_ebook

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

- [Poleć książkę na Facebook.com](#)
- [Kup w wersji papierowej](#)
- [Oceń książkę](#)

- [Księgarnia internetowa](#)
- [Lubię to! » Nasza społeczność](#)

Książkę tę dedykuję Teresie. Dzięki niej codziennie czuję, że mógłbym przenosić góry.
— Ed Wilson

Spis treści

| | |
|--------------------|----|
| Przedmowa | 15 |
| Wprowadzenie | 17 |

Część I Podstawowe wiadomości o konsoli Windows PowerShell21

| | |
|---|-----------|
| Rozdział 1. Przegląd możliwości konsoli Windows PowerShell | 23 |
| Czym jest konsola Windows PowerShell | 23 |
| Instalowanie konsoli Windows PowerShell | 25 |
| Wdrażanie konsoli Windows PowerShell | 26 |
| Polecenia cmdlet | 27 |
| Używanie narzędzi wiersza poleceń | 28 |
| Kwestie bezpieczeństwa dotyczące konsoli Windows PowerShell | 29 |
| Kontrolowanie wykonywania poleceń cmdlet Windows PowerShell | 30 |
| Zatwierdzanie poleceń | 30 |
| Zawieszanie potwierdzenia wykonania poleceń cmdlet | 31 |
| Praca z konsolą Windows PowerShell | 31 |
| Włączanie konsoli Windows PowerShell | 32 |
| Konfigurowanie konsoli Windows PowerShell | 33 |
| Przekazywanie opcji do poleceń cmdlet | 34 |
| Korzystanie z opcji pomocy | 35 |
| Zdobywanie informacji w pomocy | 38 |
| Dodatkowe źródła informacji | 43 |
| Rozdział 2. Polecenia CIM | 45 |
| Przeglądanie klas usługi WMI za pomocą poleceń CIM | 45 |
| Sposób użycia parametru classname | 45 |
| Znajdowanie metod klas WMI | 47 |
| Filtrowanie klas według kwalifikatora | 48 |

| | |
|--|----|
| Wyszukiwanie egzemplarzy klas WMI | 49 |
| Zmniejszanie liczby zwróconych własności i egzemplarzy | 50 |
| Usuwanie niepotrzebnych informacji | 51 |
| Praca z klasami Association | 52 |
| Dodatkowe źródła informacji | 57 |

Część II Planowanie skryptów 59

Rozdział 3. Moduł Active Directory 61

| | |
|---|----|
| Podstawowe wiadomości o module Active Directory | 61 |
| Instalowanie modułu Active Directory | 62 |
| Rozpoczynanie pracy z modułem Active Directory | 63 |
| Zastosowanie modułu Active Directory | 63 |
| Wyszukiwanie posiadaczy roli FSMO | 65 |
| Dokumentowanie Active Directory | 70 |
| Zmienianie nazw lokalizacji usługi Active Directory | 73 |
| Zarządzanie użytkownikami | 74 |
| Tworzenie użytkownika | 77 |
| Znajdowanie kont użytkowników i ich odblokowywanie | 78 |
| Znajdowanie wyłączonych użytkowników | 80 |
| Znajdowanie nieużywanych kont użytkowników | 81 |
| Dodatkowe źródła informacji | 85 |

Rozdział 4. Znajdowanie możliwości zastosowania skryptów 87

| | |
|--|-----|
| Automatyzowanie rutynowych zadań | 87 |
| Interfejs automatyzacji | 88 |
| Odczytywanie rejestru za pomocą metody RegRead | 91 |
| Odczytywanie rejestru za pomocą WMI | 91 |
| Odczytywanie rejestru za pomocą klas platformy .NET | 92 |
| Macierzyste techniki Windows PowerShell do wykonywania niektórych zadań | 93 |
| Wymagania strukturalne | 96 |
| Wymagania dotyczące bezpieczeństwa | 96 |
| Wykrywanie bieżącego użytkownika | 97 |
| Wykrywanie roli użytkownika | 107 |
| Wymagania dotyczące wersji platformy .NET | 111 |
| Wymagania dotyczące systemu operacyjnego | 113 |
| Wymagania dotyczące aplikacji | 117 |
| Wymagania dotyczące modułów | 118 |
| Dodatkowe źródła informacji | 119 |

| | |
|---|----------------|
| Rozdział 5. Konfigurowanie środowiska skryptowego | 121 |
| Konfigurowanie profilu | 121 |
| Tworzenie aliasów | 122 |
| Tworzenie funkcji | 125 |
| Przesłanie istniejących poleceń | 126 |
| Przekazywanie wielu parametrów | 129 |
| Tworzenie zmiennych | 134 |
| Tworzenie dysków PowerShell | 141 |
| Włączanie obsługi skryptów | 146 |
| Tworzenie profilu | 148 |
| Wybór odpowiedniego profilu | 148 |
| Tworzenie innych profili | 150 |
| Używanie funkcji z innych skryptów | 152 |
| Tworzenie biblioteki funkcji | 153 |
| Dołączanie pliku | 154 |
| Dodatkowe źródła informacji | 156 |
| Rozdział 6. Unikanie pułapek podczas pisania skryptów | 157 |
| Brak obsługi poleceń | 157 |
| Skomplikowane konstruktory | 159 |
| Kwestie dotyczące zgodności wersji | 160 |
| Sprawdzanie wersji systemu operacyjnego | 165 |
| Brak obsługi WMI | 167 |
| Praca z obiektami i przestrzeniami nazw | 167 |
| Pobieranie listy dostawców WMI | 172 |
| Praca z klasami WMI | 173 |
| Zmianie ustawień | 176 |
| Modyfikowanie wartości przez rejestr | 178 |
| Brak obsługi platformy .NET | 182 |
| Używanie statycznych metod i własności | 182 |
| Zależność od wersji | 185 |
| Brak obsługi COM | 185 |
| Brak obsługi aplikacji zewnętrznych | 191 |
| Dodatkowe źródła informacji | 195 |
| Rozdział 7. Śledzenie możliwości zastosowania skryptów | 197 |
| Ewaluacja potrzeby napisania skryptu | 197 |
| Odczytywanie pliku tekstowego | 198 |
| Eksportowanie historii poleceń | 204 |
| Polecenia promienne | 205 |
| Wysyłanie zapytań do Active Directory | 208 |
| Po prostu używaj wiersza poleceń | 214 |

| | |
|--|-----|
| Obliczanie korzyści z użycia skryptu | 217 |
| Powtarzalność | 218 |
| Możliwość opisanie w dokumentacji | 222 |
| Zdolność do adaptacji | 223 |
| Współpraca nad skryptami | 227 |
| Dodatkowe źródła informacji | 227 |

Część III Projektowanie skryptów 229

Rozdział 8. Projektowanie skryptów 231

| | |
|---|-----|
| Podstawowe wiadomości o funkcjach | 231 |
| Definiowanie funkcji w celu ułatwienia wielokrotnego wykorzystania kodu | 240 |
| Definiowanie funkcji z dwoma parametrami | 244 |
| Ograniczenia typu | 249 |
| Funkcje przyjmujące więcej niż dwa parametry | 252 |
| Definiowanie logiki biznesowej w funkcjach | 254 |
| Definiowanie funkcji w celu ułatwienia modyfikacji skryptów | 256 |
| Podstawowe wiadomości o filtrach | 263 |
| Dodatkowe źródła informacji | 268 |

Rozdział 9. Projektowanie pomocy do skryptów 271

| | |
|---|-----|
| Dodawanie dokumentacji w komentarzach jednowierszowych | 271 |
| Praca z folderami tymczasowymi | 277 |
| Używanie wielowierszowych znaczników komentarzowych w Windows PowerShell 4.0 | 279 |
| Tworzenie komentarzy wielowierszowych za pomocą znaczników komentarzowych | 279 |
| Tworzenie jednowierszowych komentarzy przy użyciu znaczników komentarzowych | 280 |
| Używanie pomocy komentarzowej | 281 |
| 13 zasad pisania efektywnych komentarzy | 286 |
| Aktualizuj dokumentację razem ze skryptem | 287 |
| Dodawaj komentarze podczas pisania kodu | 288 |
| Pisz z myślą o użytkownikach z różnych krajów | 288 |
| Zamieszczaj informacje w nagłówku | 290 |
| Podaj listę warunków używania | 291 |
| Opisuj niedoskonałości | 292 |
| Nie dodawaj zbędnych informacji | 293 |
| Uzasadniaj powody napisania kodu | 293 |
| Zastosowanie komentarzy jednowierszowych | 294 |
| Unikaj komentarzy na końcu wiersza | 295 |
| Opisuj struktury zagnieżdżone | 295 |

| | |
|--|-----|
| Używaj standardowego zestawu słów kluczowych | 296 |
| Opisuj wszelkie dziwne fragmenty kodu | 297 |
| Dodatkowe źródła informacji | 300 |

Rozdział 10. Projektowanie modułów 301

| | |
|--|-----|
| Podstawowe wiadomości o modułach | 301 |
| Znajdowanie i ładowanie modułów | 302 |
| Wyświetlanie listy dostępnych modułów | 302 |
| Ładowanie modułów | 305 |
| Instalowanie modułów | 308 |
| Tworzenie folderu na moduły | 309 |
| Sposób użycia zmiennej \$modulePath | 311 |
| Tworzenie dysku modułu | 313 |
| Sprawdzanie zależności modułów | 314 |
| Używanie modułów pochodzących z udziałów | 318 |
| Tworzenie modułu | 319 |
| Dodatkowe źródła informacji | 325 |

Rozdział 11. Obsługa wejścia i wyjścia 327

| | |
|---|-----|
| Wybór najlepszej metody pobierania danych | 328 |
| Wczytywanie danych z wiersza poleceń | 328 |
| Sposób użycia instrukcji Param | 335 |
| Pobieranie haseł na wejściu | 348 |
| Pobieranie łańcuchów połączenia | 356 |
| Monitowanie o informacje | 357 |
| Wybór najlepszej metody zwracania danych | 358 |
| Wysyłanie informacji na ekran | 360 |
| Wysyłanie wyników do pliku | 366 |
| Wysyłanie danych równocześnie na ekran i do pliku | 367 |
| Wysyłanie wyników na adres e-mail | 371 |
| Wysyłanie wyników z funkcji | 371 |
| Dodatkowe źródła informacji | 377 |

Rozdział 12. Obsługa błędów 379

| | |
|--|-----|
| Obsługa brakujących parametrów | 380 |
| Przypisywanie parametrowi wartości domyślnej | 380 |
| Tworzenie parametrów obowiązkowych | 381 |
| Ograniczanie możliwości wyboru | 382 |
| Ograniczanie liczby opcji za pomocą metody PromptForChoice | 382 |
| Znajdowanie dostępnych komputerów za pomocą polecenia ping | 384 |
| Sprawdzanie zawartości tablicy za pomocą operatora -contains | 385 |
| Testowanie własności za pomocą operatora -contains | 387 |

| | |
|--|-----|
| Postępowanie w przypadku braku uprawnień | 389 |
| Podejmowanie nieudanych prób | 391 |
| Sprawdzanie, czy skrypt posiada potrzebne uprawnienia, i eleganckie kończenie pracy | 393 |
| Sposób użycia instrukcji #Requires | 393 |
| Postępowanie w przypadku braku dostawców WMI | 396 |
| Niepoprawne typy danych | 403 |
| Błędy zakresu | 408 |
| Sposób użycia funkcji sprawdzającej wartości graniczne | 408 |
| Ograniczanie dopuszczalnych wartości parametru | 409 |
| Dodatkowe źródła informacji | 410 |

Rozdział 13. Testowanie skryptów 411

| | |
|---|-----|
| Techniki testowania podstawowej składni | 411 |
| Szukanie błędów | 415 |
| Uruchamianie skryptu | 417 |
| Dokumentowanie pracy | 419 |
| Testowanie wydajności skryptów | 421 |
| Zapisywanie i przeglądanie danych | 421 |
| Użycie potoku Windows PowerShell | 423 |
| Szacowanie wydajności różnych wersji skryptu | 426 |
| Sposób użycia parametrów standardowych | 434 |
| Sposób użycia parametru debug | 434 |
| Sposób użycia parametru Verbose | 436 |
| Sposób użycia parametru whatif | 437 |
| Tworzenie dzienników za pomocą funkcji Start-Transcript | 441 |
| Zaawansowane techniki testowania skryptów | 443 |
| Dodatkowe źródła informacji | 445 |

Rozdział 14. Dokumentowanie skryptów 447

| | |
|--|-----|
| Pobieranie dokumentacji z pomocy | 447 |
| Pobieranie dokumentacji z komentarzy | 452 |
| Sposób użycia parsera AST | 455 |
| Dodatkowe źródła informacji | 457 |

Część IV Wdrażanie skryptu 459

Rozdział 15. Ustawianie zasad wykonywania skryptów 461

| | |
|--|-----|
| Wybór zasady wykonywania dla skryptu | 461 |
| Przeznaczenie zasad wykonywania skryptów | 462 |
| Różne zasady wykonywania skryptów | 462 |
| Co to jest strefa internetowa | 463 |

| | |
|---|------------|
| Wdrażanie zasady wykonywania skryptu | 465 |
| Modyfikowanie rejestru | 465 |
| Użycie polecenia Set-ExecutionPolicy | 466 |
| Wdrażanie zasady wykonywania skryptów za pomocą zasady grupowej | 469 |
| Podpisywanie kodu | 472 |
| Dodatkowe źródła informacji | 474 |
| Rozdział 16. Uruchamianie skryptów | 475 |
| Skrypty logowania | 475 |
| Co uwzględnić w skryptach logowania | 476 |
| Metody wywoływania skryptów logowania | 480 |
| Folder skryptów | 482 |
| Wdrażanie lokalne | 482 |
| Lokalne wdrażanie pakietu MSI | 482 |
| Samodzielne skrypty | 483 |
| Diagnostyka | 483 |
| Raportowanie i kontrolowanie | 483 |
| Skrypty pomocy technicznej | 484 |
| Unikaj edytowania | 484 |
| Dostarcz dobrej jakości pomoc | 484 |
| Dodatkowe źródła informacji | 487 |
| Rozdział 17. Kontrola wersji skryptów | 489 |
| Dlaczego warto stosować kontrolę wersji | 489 |
| Unikanie wprowadzania błędów | 490 |
| Precyzyjne usuwanie usterek | 491 |
| Śledzenie zmian | 491 |
| Lista skryptów | 491 |
| Zachowanie zgodności z innymi skryptami | 491 |
| Wewnętrzny numer wersji w komentarzach | 493 |
| Programy do kontroli wersji | 496 |
| Dodatkowe źródła informacji | 497 |
| Rozdział 18. Rejestrowanie wyników | 499 |
| Zapisywanie danych w pliku tekstowym | 499 |
| Projektowanie metody rejestrowania wyników skryptu | 500 |
| Miejsce przechowywania tekstu | 509 |
| Przechowywanie dzienników w lokalizacjach sieciowych | 513 |
| Zapisywanie danych w dzienniku zdarzeń | 517 |
| Sposób użycia dziennika aplikacji | 519 |
| Tworzenie własnego dziennika zdarzeń | 519 |
| Zapisywanie danych w rejestrze | 520 |
| Dodatkowe źródła informacji | 522 |

| | |
|---|----------------|
| Rozdział 19. Rozwiązywanie problemów ze skryptami | 523 |
| Podstawy debugowania w Windows PowerShell | 523 |
| Błędy składniowe | 524 |
| Błędy wykonawcze | 524 |
| Błędy logiczne | 527 |
| Sposób użycia polecenia Set-PSDebug | 530 |
| Śledzenie wykonywania skryptu | 531 |
| Wykonywanie skryptu krok po kroku | 535 |
| Włączanie trybu ścisłego | 542 |
| Debugowanie skryptów | 545 |
| Ustawianie punktów wstrzymania | 547 |
| Reagowanie na punkty wstrzymania | 555 |
| Wyświetlanie listy punktów wstrzymania | 556 |
| Włączanie i wyłączanie punktów wstrzymania | 558 |
| Usuwanie punktów wstrzymania | 559 |
| Dodatkowe źródła informacji | 561 |
| Rozdział 20. Praca w środowisku Windows PowerShell ISE | 563 |
| Uruchamianie środowiska Windows PowerShell ISE | 563 |
| Zawartość okna Windows PowerShell ISE | 563 |
| Praca w okienku skryptu | 565 |
| Rozwijanie nazw za pomocą klawisza Tab i funkcja IntelliSense | 567 |
| Sposób użycia fragmentów kodu w środowisku Windows PowerShell ISE | 569 |
| Tworzenie skryptów przy użyciu gotowych fragmentów kodu | 569 |
| Tworzenie nowych gotowych fragmentów kodu w Windows PowerShell ISE | 569 |
| Usuwanie fragmentów kodu zdefiniowanych przez użytkownika | 570 |
| Dodatkowe źródła informacji | 571 |
| Rozdział 21. Narzędzia do pracy zdalnej i zadania Windows PowerShell | 573 |
| Narzędzia do pracy zdalnej Windows PowerShell | 573 |
| Klasyczne sposoby pracy zdalnej | 573 |
| Zdalne zarządzanie systemem Windows | 583 |
| Zadania Windows PowerShell | 590 |
| Dodatkowe źródła informacji | 596 |
| Rozdział 22. Przepływy pracy w Windows PowerShell | 597 |
| Do czego służą przepływy pracy w Windows PowerShell | 597 |
| Wymagania przepływów pracy | 598 |
| Prosty przepływ pracy | 598 |
| Równoległe wykonywanie poleceń | 599 |
| Aktywności przepływów pracy | 602 |
| Polecenia Windows PowerShell jako aktywności | 602 |
| Niedozwolone polecenia rdzenne | 604 |

| | |
|---|-----|
| Nieautomatyczne aktywności z poleceń | 604 |
| Aktywności równoległe | 605 |
| Ustalanie punktów kontrolnych dla przepływów pracy w Windows PowerShell | 606 |
| Czym są punkty kontrolne | 606 |
| Tworzenie punktów kontrolnych | 606 |
| Dodawanie punktów kontrolnych | 607 |
| Dodawanie aktywności sekwencyjnej do przepływu pracy | 609 |
| Dodatkowe źródła informacji | 611 |

Rozdział 23. Usługa konfiguracji żądanego stanu programu PowerShell 613

| | |
|--|-----|
| Podstawowe informacje o Usłudze konfiguracji żądanego stanu programu PowerShell | 613 |
| Proces DSC | 614 |
| Parametry konfiguracji | 617 |
| Ustawianie zależności | 618 |
| Dane konfiguracji | 619 |
| Kontrolowanie dryfu konfiguracji | 623 |
| Dodatkowe źródła informacji | 625 |

| | |
|------------------------|------------|
| O autorze | 627 |
|------------------------|------------|

| | |
|------------------------|------------|
| Skorowidz | 629 |
|------------------------|------------|

Przedmowa

W kwietniu 2003 r. Jeffrey Snover z firmy Microsoft pozwolił mi podejrzeć wczesną wersję konsoli PowerShell, która wówczas była w fazie beta i nazywała się Monad. Muszę przyznać, że zakochałem się w tym narzędziu od pierwszego wejrzenia, ale przez kolejnych pięć lat byłem zbyt zajęty, aby bliżej się nim zainteresować. Wkrótce zdałem sobie sprawę, że chyba coś mi umknęło. „Obiekty w potoku”? Czy to coś w rodzaju węży w samolocie? „Tablice mieszające? Poproszę jedną ze smażonym jajkiem”.

Okazało się, że mam dużo do nadrobienia, i prawie przegrzałem Google, szukając różnych informacji na temat PowerShell. Przemierzając zasoby internetu, odkryłem też, że prawie wszystkie drogi prowadzą w jedno miejsce: na stronę blogu *Hey, Scripting Guy!*. Artykuły na tym blogu pojawiały się codziennie i bardzo się zdziwiłem, gdy dowiedziałem się, że prawie wszystkie są pisane przez jednego autora: Eda Wilsona. Później poznałem go osobiście i uwierz mi, ten gość jest jeszcze bardziej zabawny w rzeczywistości, niż się wydaje, sądząc po jego tekstach. Tym mniej więcej sposobem znalazłem się w tej książce.

Jeśli jesteś administratorem systemu Windows, to znajomość konsoli Windows PowerShell jest dla Ciebie podstawą (podobnie jest, jeśli dopiero zamierzasz zostać takim administratorem). Ale to nie znaczy, że nauka obsługi tego narzędzia jest zawsze łatwa. W jej trakcie nieraz pewnie pomyślisz, że przydałby się jakiś poradnik z gotowymi rozwiązaniami. Ed Wilson właśnie go napisał i w rękę trzymasz najnowsze wydanie. Przestudiuj go dokładnie i wypróbuj wszystkie przykłady, a wkrótce sam będziesz automatyzował i skryptował wszystko jak szalony. Dobrej zabawy!

— Mark Minasi, autor książek z serii *Mastering Windows Server*

PS. Gdybyś nie wiedział, obiekty w potoku są o wiele lepsze od węży w samolocie. Naprawdę.

Wprowadzenie

Witaj w książce Windows PowerShell. Najlepsze praktyki, która powstała przy współpracy z zespołem ds. produktu Windows PowerShell firmy Microsoft. Znajdziesz w niej szczegółowe informacje na temat tego narzędzia oraz poznasz najlepsze techniki jego obsługi poparte praktycznymi doświadczeniami wyniesionymi z pracy z tym produktem w różnych środowiskach. Na marginesie w wielu rozdziałach znajdziesz też notatki opisujące spostrzeżenia różnych specjalistów, administratorów przedsiębiorstw i posiadaczy tytułu Windows PowerShell Most Valuable Professional (MVP).

Znaczna część tej książki dotyczy narzędzia Windows PowerShell 4.0 dostępnego w systemach Windows 8.1 i Windows Server 2012 R2. Ponieważ w Windows PowerShell 4.0 wprowadzono usługę konfiguracji żądanego stanu programu (ang. *Desired State Configuration* — DSC), przykłady przedstawione w rozdziale 23. muszą być uruchamiane w konsoli Windows PowerShell 4.0. Natomiast prawie wszystkie pozostałe przykłady można uruchamiać także w konsoli Windows PowerShell 3.0 (w systemach Windows 8 i Windows Server 2012). Ponadto wiele przykładów działa także w konsoli Windows PowerShell 2.0 w każdej wersji systemu Windows, w której da się ją zainstalować.

Adresaci książki

Książka *Windows PowerShell. Najlepsze rozwiązania* jest przeznaczona dla każdego, kto musi projektować, implementować oraz obsługiwać produkty dla przedsiębiorstw, takie jak np. Active Directory Domain Services, System Center, Exchange oraz SharePoint. Ponadto pozycja ta przyda się nauczycielom uczącym obsługi PowerShell, a nawet podczas kursów MCSE. Powinni się w nią zaopatrzyć także ci, którzy po prostu chcą zautomatyzować działanie swoich komputerów.

Podział książki

Książka ta jest podzielona na cztery części:

- Część I: „Podstawowe wiadomości o konsoli Windows PowerShell”.
- Część II: „Planowanie skryptów”.

- Część III: „Projektowanie skryptów”.
- Część IV: „Wdrażanie skryptu”.

Pierwsza część zawiera tylko dwa rozdziały, w których znajduje się opis podstawowych funkcji konsoli Windows PowerShell. Jest to część mająca za zadanie wyrównanie poziomu wiedzy różnych czytelników i powinna zostać szczególnie uważnie przestudiowana przez osoby dopiero uczące się obsługi Windows PowerShell.

W drugiej części opisano, kiedy warto napisać skrypt, środowisko skryptowe oraz jak uniknąć różnych pułapek. Część ta również jest idealna dla osób uczących się obsługi konsoli Windows PowerShell, ale skorzystają z niej także administratorzy, którzy znają już podstawy, ale szukają informacji na temat pisania nowych skryptów.

Część trzecia jest poświęcona projektowaniu skryptów. Dowiesz się, jak zaplanować wejście i wyjście skryptów oraz jak pisać do nich dokumentację. Ta bardziej zaawansowana część jest przeznaczona dla doświadczonych użytkowników i programistów piszących skrypty dla innych.

Tematem ostatniej części jest wdrażanie skryptów, a więc dowiesz się, jakie są sposoby uruchamiania skryptów, poznasz techniki wersji, pracy zdalnej, a także dowiesz się, czym jest przepływ pracy oraz konfiguracja żadanego stanu programu. Część ta jest przeznaczona dla administratorów przedsiębiorstw zaangażowanych w prace nad rozwojem i eksploatacją oprogramowania.

Wymagania systemowe

Przykłady przedstawione w tej książce najlepiej uruchamiać w poniższej konfiguracji sprzętowo-programowej dla oprogramowania Exchange 2010:

- Windows Server 2008 lub Windows Server 2008 R2.
- 1 GB pamięci RAM.
- Procesor o architekturze x64 firmy Intel lub AMD obsługujący operacje 64-bitowe.
- 1,2 GB wolnego miejsca na dysku.
- Monitor o rozdzielczości nie mniejszej niż 800×600.

Poniżej znajduje się opis minimalnych wymagań systemowych, które muszą być spełnione, aby można było uruchomić przykłady dołączone do tej książki:

- System Windows XP z najnowszym dodatkiem Service Pack i najnowszymi aktualizacjami z serwisu Microsoft Update.
- Monitor o rozdzielczości nie mniejszej niż 1024×768.
- Napęd CD-ROM.
- Mysz firmy Microsoft lub inne zgodne urządzenie wskazujące.

Podziękowania

Książki tak bogate w treść nie powstają bez pomocy wielu osób. Przede wszystkim dziękuję mojej żonie, Teresie Wilson, znanej też pod pseudonimem Scripting Wife. Nie dość, że koordynowała formalności związane z nabywaniem notatek zamieszczanych na marginesie, to jeszcze dodatkowo przeczytała całą książkę przynajmniej trzy razy. Mój redaktor merytoryczny Microsoft PFE Brian Wilhite znalazł wiele błędów, przez które wyszedłbym na głupka. Ponadto udzielił mi wielu wskazówek dotyczących nie tylko kwestii stricte językowych, ale również dotyczących jakości kodu źródłowego. Brian to jest gość. Mam też szczęście, że społeczność użytkowników konsoli Windows PowerShell jest bardzo aktywna, dzięki czemu szybko otrzymałem odzew na apel o dostarczenie materiałów do notatek na marginesie. Ich wysoka jakość i różnicowanie tematyczne sprawiają, że przyjemnie się je czyta i podnoszą jakość książki. Jeśli spotkasz autora którejś z tych notatek, powiedz mu „cześć”. W końcu chcę podziękować Jefffeyowi Snoverowi, Kenowi Hansenowi i wszystkim członkom zespołu Windows PowerShell. Zrobili doskonały produkt, który z roku na rok staje się jeszcze lepszy. Windows PowerShell rządzi!

Przykłady kodu

Pliki źródłowe z przykładami kodu opisanego w tej książce znajdują się na serwerze FTP wydawnictwa Helion, pod adresem <ftp://ftp.helion.pl/przyklady/winpsp.zip>.

CZĘŚĆ I

Podstawowe wiadomości o konsoli Windows PowerShell

Rozdział 1. Przegląd możliwości konsoli Windows PowerShell

Rozdział 2. Polecenia CIM

Rozdział 1

Przegląd możliwości konsoli Windows PowerShell

- Czym jest konsola Windows PowerShell
- Instalowanie konsoli Windows PowerShell
- Wdrażanie konsoli Windows PowerShell
- Używanie narzędzi wiersza poleceń
- Kwestie bezpieczeństwa dotyczące konsoli Windows PowerShell
- Praca z konsolą Windows PowerShell
- Przekazywanie opcji do poleceń cmdlet
- Korzystanie z opcji pomocy
- Dodatkowe źródła informacji

Czym jest konsola Windows PowerShell

Największą przeszkodą utrudniającą administratorom sieci Windows przejście na konsolę Windows PowerShell 4.0 jest brak wiedzy, co to właściwie jest. W pewnym sensie jest to następca sędziwego wiersza poleceń. W systemie Windows Server 2012 R2 można nawet ustawić konsolę Windows PowerShell jako podstawowy interfejs zamiast wiersza poleceń. Jak widać w poniższym przykładzie, w konsoli Windows PowerShell można zmienić katalog roboczy za pomocą polecenia `cd` oraz wyświetlić listę katalogów za pomocą polecenia `dir`, dokładnie tak, jak robi się to w wierszu poleceń.

Windows PowerShell

Copyright (C) 2013 Microsoft Corporation. All rights reserved.

PS C:\Users\ed.IAMMRED> cd c:\

PS C:\> dir

Directory: C:\

| Mode | LastWriteTime | Length | Name |
|-------|-------------------|--------|-----------|
| ---- | ----- | ----- | ---- |
| d---- | 9/4/2013 12:06 PM | | DCS |
| d---- | 9/8/2013 8:32 PM | | DemoUser |
| d---- | 9/8/2013 6:52 PM | | fso |
| d---- | 9/8/2013 9:15 PM | | myprocess |

| | | | |
|-------|-----------|----------|---------------------|
| d---- | 8/22/2013 | 11:22 AM | PerfLogs |
| d-r-- | 8/22/2013 | 3:11 PM | Program Files |
| d-r-- | 8/27/2013 | 8:19 PM | Program Files (x86) |
| d---- | 9/8/2013 | 10:12 PM | ScriptFolder |
| d---- | 9/8/2013 | 7:22 PM | server1Config |
| d---- | 9/8/2013 | 10:46 PM | ServerConfig |
| d---- | 9/8/2013 | 9:22 PM | StartBits |
| d-r-- | 8/27/2013 | 8:06 PM | Users |
| d---- | 8/27/2013 | 7:52 PM | Windows |

PS C:\>

W razie potrzeby można też łączyć „tradycyjne” polecenie wiersza poleceń z innymi narzędziami, takimi jak np. `fsutil`. Spójrz na poniższy przykład:

PS C:\> md c:\test

Directory: C:\

| Mode | LastWriteTime | Length | Name |
|-------|------------------|--------|------|
| ---- | ----- | ----- | ---- |
| d---- | 9/9/2013 3:31 PM | | test |

PS C:\> fsutil file createnew c:\test\mynewfile.txt 1000
File c:\test\mynewfile.txt is created
PS C:\> cd test
PS C:\test> dir

Directory: C:\test

| Mode | LastWriteTime | Length | Name |
|------|------------------|--------|---------------|
| ---- | ----- | ----- | ---- |
| a--- | 9/9/2013 3:31 PM | 1000 | mynewfile.txt |

PS C:\test>

Użyliśmy konsoli Windows PowerShell w sposób interaktywny. Jest to podstawowy sposób obsługi tego narzędzia i polega na uruchomieniu jego okna oraz wpisywaniu poleceń. Polecenia można wpisywać pojedynczo lub grupować, jak w plikach wsadowych. Grupowaniem zajmujemy się później, bo żeby zrozumieć te techniki, trzeba trochę więcej wiedzieć.

Zapiski praktyka

Jason Helmick, starszy specjalista ds. technologicznych
Concentrated Technology

To niesamowite, że w ciągu zaledwie paru lat liczba poleceń konsoli Windows PowerShell wzrosła z kilkuset do kilku tysięcy i za ich pomocą można obsługiwać rozmaite produkty firmy Microsoft. To oznacza, że prawie na pewno jest w niej coś, o czym nie wiesz, a co warto byłoby wiedzieć. Doskonałym i nie do przecenienia narzędziem do poznawania poleceń jest komenda `Get-Help`, chociaż nie jest to jedyny sposób zdobywania informacji.

Spółeczność użytkowników konsoli Windows PowerShell jest bardzo prężna i aktywna. Należą do niej posiadacze tytułów MVP, różni guru oraz maniacy. Wszyscy piszą blogi, tweetują oraz udzielają się na forach, na których szukają informacji, dyskutują i informują innych o praktycznych sposobach rozwiązania problemów. Zaangażowanie w tej społeczności jest doskonałym sposobem na poszerzenie wiedzy na temat konsoli Windows PowerShell, ale oczywiście też nie jedynym.

Najwięcej uczę się podczas pracy z innymi administratorami, którzy również używają tej konsoli. Wielokrotnie widziałem, jak inni wykonują te same czynności co ja w odmienny sposób. Często używali do tego technik lub poleceń, o których nie miałem pojęcia. Jako przykład opiszę sytuację, która przydarzyła mi się niedawno.

Przygotowywałem maszyny wirtualne do prezentacji dla początkujących użytkowników konsoli Windows PowerShell. Było to dla mnie wyjątkowe wydarzenie i za wszelką cenę nie chciałem niczego zepsuć. Podczas ładowania maszyn wirtualnych musiałem sprawdzić parę adresów IP, aby umożliwić zdalne połączenie się z nimi przez drugiego prowadzącego. Jest to łatwe zadanie, wymagające jedynie użycia macierzystego narzędzia wiersza poleceń *IPConfig.exe* (które oczywiście działa jak marzenie także w konsoli Windows PowerShell). W pewnym momencie powiedziałem do kolegi coś w rodzaju „Czekaj, tylko sprawdzę zewnętrzny IP; zaraz włączę IPConfig i...”.

On na to: „Nie używasz *gip*?”. Byłem zaskoczony, bo nigdy nie słyszałem o czymś takim jak *gip*. Gdy zobaczył, że jestem zdezorientowany, uśmiechnął się i powiedział: „Codziennie uczę się czegoś nowego o Windows PowerShell — spróbuj tego”. Poszedłem za jego radą. Okazało się, że *gip* to skrócona nazwa polecenia *Get-NetIPConfiguration*, które zwraca lepiej sformatowany i wygodniejszy do czytania wynik niż narzędzie *IPConfig.exe*. A ponieważ jest to polecenie konsoli Windows PowerShell, zwraca obiekty, z którymi można zrobić wiele niesamowitych rzeczy. Wtedy nauczyłem się czegoś nowego, znacznie lepszego rozwiązania pewnego problemu niż to, które znałem wcześniej. A było to możliwe dlatego, że współpracowałem z innym użytkownikiem konsoli Windows PowerShell. Morał tej historii jest prosty: współpracuj z innymi administratorami, którzy też używają Windows PowerShell — najlepiej w tym samym pomieszczeniu. Dzięki temu będziecie uczyć się od siebie nawzajem.

A kim był ten człowiek, który nauczył mnie czegoś nowego? Był to wynalazca konsoli Windows PowerShell, Distinguished Engineer, Jeffrey Snover. Skoro nawet on może nauczyć się czegoś nowego o tym narzędziu, to ja i Ty z pewnością też. Współpracuj ze znajomymi i wymieniaj doświadczenia.

Instalowanie konsoli Windows PowerShell

Konsola Windows PowerShell 4.0 jest standardowo dostępna w systemach Windows 8.1 i Windows Server 2012 R2. Z portalu Microsoft Download Center (*Microsoft.Com/Downloads*) można pobrać pakiet Windows Management Framework 4.0 zawierający aktualne wersje narzędzi WinRM, WMI oraz Windows PowerShell. Pakiet ten można zainstalować w systemach Windows 7 i Windows

Server 2008 R2 — oba muszą mieć zainstalowany przynajmniej dodatek Service Pack 1 i platformę .NET Framework 4.5. Ponadto pakiet ten można też zainstalować w systemach Windows 8 i Windows Server 2012.

Aby bezproblemowo przeprowadzić instalację, można użyć skryptu sprawdzającego system operacyjny, numer zainstalowanego dodatku serwisowego oraz wersję platformy .NET. Poniżej znajduje się kod takiego skryptu o nazwie *Get-PowerShellRequirements.ps1*:

Get-PowerShellrequirements.ps1

```
Param([string[]]$computer = @($env:computername, "LocalHost"))
foreach ($c in $computer)
{
    $o = Get-WmiObject win32_operatingsystem -cn $c
    switch ($o.version)
    {
        {$o.version -gt 6.2} {"$c to Windows 8 lub nowszy"; break}
        {$o.version -gt 6.1}
        {
            If($o.ServicePackMajorVersion -gt 0){$sp = $true}
            If(Get-WmiObject Win32_Product -cn $c |
                where { $_.name -match '.NET Framework 4.5'}) {$net = $true }
            If($sp -AND $net) { "$c Spełnia wymagania PowerShell 3" ; break}
            ElseIf (!$sp) {"$c wymaga zainstalowania dodatku Service Pack"; break}
            ElseIf (!$net) {"$c wymaga aktualizacji platformy .NET Framework" ; break}
            {$o.version -lt 6.1} {"$c nie spełnia wymagań PowerShell 3.0"; break}
            Default {"Nie wiadomo, czy $c spełnia wymagania PowerShell 3.0"}
        }
    }
}
```

Wdrażanie konsoli Windows PowerShell

Po pobraniu konsoli Windows PowerShell ze strony <http://www.Microsoft.com/downloads> można ją wdrożyć w swoim przedsiębiorstwie przy użyciu jednej z kilku standardowych metod. Oto niektóre z najczęściej stosowanych sposobów:

1. Utwórz pakiet Microsoft Systems Center Configuration Manager i ogłoś go w odpowiedniej jednostce organizacyjnej lub kolekcji.
2. Utwórz obiekt zasad grupy w usługach domenowych Active Directory i połącz go z odpowiednią jednostką organizacyjną.
3. Zatwierdź aktualizację w usłudze Software Update Services.

UWAGA

Aby włączyć narzędzie wiersza poleceń konsoli Windows PowerShell, wykonaj polecenie *Start/Uruchom/PowerShell*.

Polecenia cmdlet

Oprócz zwykłych programów i poleceń interpretera *CMD.exe* w konsoli Windows PowerShell można też używać jej macierzystych poleceń cmdlet (czyt. jak ang. *commandlet*). Polecenia cmdlet może tworzyć każdy, kto chce. Programiści konsoli Windows PowerShell utworzyli podstawowy zestaw tych poleceń, ale oprócz tego do Windows 8 dodano jeszcze setki poleceń utworzonych przez inne zespoły programistyczne firmy Microsoft. Polecenia te są jak pliki wykonywalne, tylko wykorzystują infrastrukturę konsoli Windows PowerShell, dzięki czemu bardzo łatwo się je tworzy. Nie są to skrypty, tylko skompilowane programy, ponieważ do ich budowy używa się usług specjalnej przestrzeni nazw platformy .NET. Konsola Windows PowerShell 4.0 w systemie Windows 8.1 ma około tysiąca poleceń cmdlet. Jako że często dodawane są nowe funkcje i role, podobnie sprawa wygląda z poleceniami. Ich zadaniem jest ułatwiać prace administratorów sieci i konsultantów bez konieczności nauki nowego języka skryptowego. Jedną z największych zalet konsoli Windows PowerShell jest to, że wszystkie nazwy poleceń cmdlet są zbudowane według jednego wzoru Czasownik-Rzeczownik, np. *Get-Hello* (Pobierz-Pomoc), *Get-EventLog* (Pobierz-DziennikZdarzeń), *Get-Process* (Pobierz-Proces). Polecenia, których nazwy zaczynają się od czasownika *Get*, wyświetlają informacje o tym, co jest wymienione po prawej stronie łącznika. Są też polecenia zaczynające się od czasownika *Set* (ustawiające), np. za pomocą polecenia *Set-Service* można zmienić tryb uruchamiania usługi. Nazwy wszystkich poleceń cmdlet zaczynają się od standardowego angielskiego czasownika. Za pomocą polecenia *Get-Verb* można nawet wyświetlić listę wszystkich zatwierdzonych czasowników. W Windows PowerShell 4.0 jest ich prawie 100.

Zapiski praktyka

David Moravec, MVP Microsoft PowerShell

Mainstream Technologies

Jedną z najbardziej przydatnych nowości w Windows PowerShell 4.0 jest polecenie cmdlet *Get-FileHash* służące do liczenia skrótów plików. Kiedyś trzeba było do tego używać klasy *System.Security.Cryptography.HashAlgorithm*. Gdy trzeba było to zrobić lokalnie, nie było problemu, ale jeśli udostępniało się skrypty także innym programistom, trzeba było dodatkowo dostarczyć swoją funkcję użytą do tworzenia skrótów. Teraz już nie trzeba tego robić.

Z mojego doświadczenia wynika, że do tworzenia skrótów najczęściej używa się algorytmu MD5. Jest szybki i łatwy w użyciu oraz obsługuje go każde narzędzie. Jeśli wykonasz polecenie *Get-FileHash* z domyślnymi ustawieniami, otrzymasz następujący wynik:

```
PS C:\Users\Makovec> Get-FileHash .\myFile.exe | fl *
Path : C:\Users\Makovec\myFile.exe
Type : System.Security.Cryptography.SHA256Managed
Hash : p/a6HFn9QkCFQWiaQMo8hVILmCHCPMuaNrRn2DKJKVM=
```

Jak widać, została użyta metoda SHA256. Ale można ją zmienić na MD5 za pomocą parametru *Algorithm*:

```
PS C:\Users\Makovec> Get-FileHash .\myFile.exe -Algorithm MD5 | fl *
Path : C:\Users\Makovec\myFile.exe
Type : System.Security.Cryptography.MD5CryptoServiceProvider
Hash : L1uabH1YgDx/WSR4e2SIgw==
```

Parametr `Algorithm` przyjmuje następujące wartości: SHA1, SHA256, SHA384, SHA512, MACTripleDES, MD5 oraz RIPEMD160. Niestety polecenie `Get-FileHash` nie przyjmuje na wejściu potoków, więc aby sprawdzić kilka plików, należy użyć następującej metody:

```
PS C:\Users\Makovec> dir myfile* |% { Get-FileHash -FilePath $_.FullName
} | ft Path, Hash -auto
Path                                     Hash
----
C:\Users\Makovec\myFile.exe             pa6HFn9QkCFQWiaQM08hVILmCHCPMuanRn2DKJKVM=
C:\Users\Makovec\myFile1.txt            hvEVE3TDmfnYS9Hr0weNDTt2YJjXNfPIjKIn0KNYp8g=
C:\Users\Makovec\myFile10.txt           MD01qpQP8CWfY9RfRhJRFxf6tBRUU18QhUBsEBZzTg0=
C:\Users\Makovec\myFile2.txt            PrLYwFUSFV6ffc+pOPk5voQW1D0jPeKDY3071VFFCQ=
C:\Users\Makovec\myFile3.txt            VF1Q01uLMVJUWHJCoyQDf6+KCLu9BU5mokUpDhUH5hY=
C:\Users\Makovec\myFile4.txt            9ipYmXXKSrPapxgGZII5HKt6iz8gmuQnSky8DJXe0=
C:\Users\Makovec\myFile5.txt            Pt95mm1rE1r0N7zPmkZ8ntffRmmbN6q22bnI1gzJaJk=
C:\Users\Makovec\myFile6.txt            dJXh7cLzB2hf87DtJCRrTAjDLhXJo1opRBQYNGt7CPc=
C:\Users\Makovec\myFile7.txt            00AHHQebMbTxQv1QEKYkd63bF8J8jqHHH0zgA4rFGA=
C:\Users\Makovec\myFile8.txt            EEXKqgV/KXSesD6x8HVMfjZTN4DzyjCEWjRuM5R7dI=
C:\Users\Makovec\myFile9.txt            jdh1fHvSJ5RJSZ62M0c+J5ujM3fMzzWXWdndZ8V0L4s=
```

Jeśli chcesz, aby algorytm MD5 był domyślnym ustawieniem, możesz go ustawić za pomocą `$PSDefaultParameterValues`, tylko pamiętaj, że taka modyfikacja spowoduje, że na swoim komputerze będziesz otrzymywać wyniki różniące się od wyników na innych komputerach. Ale jeśli chcesz tylko szybko coś sprawdzić w swojej maszynie, to nie powinno być z tym żadnego problemu. Ja dodałem do swojego profilu poniższe dwie linijki kodu:

```
Set-Alias -Name md5 -Value Get-FileHash
$PSDefaultParameterValues = @{'Get-FileHash:Algorithm'='MD5'}
```

I mimo tego poniższe dwa polecenia i tak zwracają taki sam wynik:

```
PS C:\Users\Makovec> Get-FileHash .\myFile.exe -Algorithm MD5
PS C:\Users\Makovec> md5 .\myFile.exe

Path                                     Type
Hash
----
C:\Users\Makovec\myFile.exe             System.Security.Cryptography.MD5Crypt...
L1uabH1YgDx\WSR4e2SIgw==
```

Używanie narzędzi wiersza poleceń

Jak napisałem wcześniej, w konsoli Windows PowerShell można używać także narzędzi wiersza poleceń. Zaletą wspomnianego rozwiązania jest to, że konsola Windows PowerShell, w odróżnieniu od zwykłego wiersza poleceń, obsługuje przetwarzanie potokowe i formatuje dane. Ponadto jeśli masz pliki wsadowe albo polecenia wiersza poleceń wykorzystujące istniejące narzędzia wiersza poleceń, to możesz je w łatwy sposób zmodyfikować tak, aby działały w środowisku Windows PowerShell.

Aby wykonać polecenie `ipconfig`, wykonaj następujące czynności:

1. Uruchom konsolę Windows PowerShell, wpisując PowerShell na stronie startowej systemu Windows. Pojawi się okno dialogowe Windows PowerShell, domyślnie ustawione na katalog główny folderu *Dokumenty*.

2. Wpisz polecenie `ipconfig /all`:

```
PS C:\> ipconfig /all
```

3. Przekaż potokowo wynik tego polecenia do pliku tekstowego, jak pokazano w poniższym przykładzie:

```
PS C:\> ipconfig /all >ipconfig.txt
```

4. Otwórz utworzony plik tekstowy w Notatniku za pomocą poniższego polecenia:

```
PS C:\> notepad ipconfig.txt
```

Możliwość wykonywania pojedynczych poleceń w konsoli Windows PowerShell jest bardzo przydatna, ale czasami potrzebne są dodatkowe informacje diagnostyczne albo szczegóły konfiguracyjne i aby je uzyskać, trzeba wykonać więcej niż jedno polecenie. Dopiero w takich przypadkach można w pełni docenić zalety konsoli Windows PowerShell. Kiedyś trzeba by było napisać plik wsadowy albo wpisać wszystkie polecenia ręcznie.

Poniżej znajduje się zawartość przykładowego pliku wsadowego *TroubleShoot.bat*.

troubleShoot.bat

```
ipconfig /all >C:\tshoot.txt  
route print >>C:\tshoot.txt  
hostname >>C:\tshoot.txt  
net statistics workstation >>C:\tshoot.txt
```

Oczywiście podczas ręcznego wpisywania poleceń po każdym poleceniu trzeba odczekać, aż zwróci wynik; dopiero wtedy można wpisać następne. Zanim jedno polecenie skończy swe działanie, można jednak zapomnieć, które miało być następne. W Windows PowerShell wyeliminowano ten problem. Można wpisać kilka poleceń w jednej linii i odejść od komputera na czas wykonywania zleconych zadań. Nie trzeba w tym celu pisać żadnych plików wsadowych.

WSKAZÓWKA

W konsoli Windows PowerShell można wpisywać po kilka poleceń w jednej linii. Polecenia należy oddzielać od siebie średnikami.

Kwestie bezpieczeństwa dotyczące konsoli Windows PowerShell

Jak każdego wszechstronnego narzędzia, tak i konsoli Windows PowerShell muszą dotyczyć pewne kwestie związane z bezpieczeństwem. Chociaż należy zaznaczyć, że bezpieczeństwo było jednym z podstawowych celów programistów tego narzędzia.

Domyślnie konsola Windows PowerShell otwiera się w folderze *Dokumenty*, dzięki czemu od razu znajdujesz się w katalogu, w którym masz uprawnienia do wykonywania pewnych czynności. Jest to znacznie bezpieczniejsze rozwiązanie niż przechodzenie od razu do katalogu głównego napędu lub systemu.

Gdy chce się zmienić katalog, nie można automatycznie przejść w górę do następnego poziomu. Należy wpisać dokładną nazwę folderu docelowego (choć można przejść o poziom wyżej za pomocą polecenia `cd ..`).

Domyślnie możliwość wykonywania skryptów jest wyłączona, ale można to łatwo zmienić przy użyciu zasad grupy. Ponadto funkcję tę można ustawiać dla pojedynczych użytkowników i sesji.

Kontrolowanie wykonywania poleceń cmdlet Windows PowerShell

Czy zdarzyło Ci się kiedykolwiek uruchomić wiersz poleceń tylko po to, by dowiedzieć się, do czego służy jakieś polecenie? A gdyby tym poleceniem było `Format C:\`? Czy na pewno chcesz sformatować dysk C? W tym podrozdziale przedstawiam parę argumentów, które można przekazywać do poleceń cmdlet, aby kontrolować sposób ich wykonywania. Nie są one wprowadzane obsługiwane przez wszystkie polecenia cmdlet, ale przez większość standardowo dostarczanych z konsolą. Do kontrolowania sposobu wykonywania poleceń cmdlet służą trzy argumenty: `-whatif`, `-confirm` oraz `suspend`. Tak naprawdę `suspend` nie jest argumentem polecenia, a raczej czynnością, którą można podjąć przy potwierdzeniu.

UWAGA

Parametr `-whatif` należy wpisać po nazwie polecenia cmdlet. Działa on tylko dla poleceń zmieniających stan systemu, a więc nie można go używać z takimi poleceniami jak `Get-Process`, które tylko wyświetlają informacje.

Polecenia cmdlet konsoli Windows PowerShell zmieniające stan systemu (np. `Set-Service`) mogą działać w trybie prototypowym włączanym za pomocą parametru `-whatif`. To, czy zostanie on zaimplementowany, zależy wyłącznie od twórcy polecenia, ale programiści konsoli Windows PowerShell zalecają implementowanie tego parametru we wszystkich poleceniach zmieniających stan systemu. Poniżej przedstawiony jest przykład użycia parametru `-whatif` w praktyce.

```
PS C:\> Set-Service -Name bits -StartupType 'manual' -WhatIf
What if: Performing the operation "Set-Service" on target "Usługa inteligentnego transferu w tle (bits)".
```

Zatwierdzanie poleceń

Jak pokazałem w poprzednim podrozdziale, za pomocą parametru `-whatif` można sprawdzić, co się stanie, gdy wykonamy pewne polecenia cmdlet. Ale jeśli chcesz zostać spytany jeszcze raz, czy na pewno masz zamiar wykonać dane polecenie, możesz użyć argumentu `-Confirm`, jak pokazano w poniższym przykładzie:

```
PS C:\> Get-Process -Name notepad | Stop-Process -Confirm

Confirm
Are you sure you want to perform this action?
Performing operation "Stop-Process" on Target "notepad (4148)".
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help
(default is "Y"):y
```

Zawieszanie potwierdzenia wykonania poleceń cmdlet

Możliwość wyświetlania prośby o potwierdzenie jest bardzo przydatna i może być niezbędna do utrzymania wysokiego poziomu niezawodności działania systemu. Czasami się zdarza, że po wpisaniu jakiegoś długiego polecenia przypomnimy sobie, że najpierw mieliśmy zrobić coś innego. W takim przypadku można zawiesić wykonywanie tego polecenia. Najlepsze jest to, że nadal możemy korzystać z konsoli i wykonywać inne polecenia. W przedstawionym poniżej przykładzie uruchomionych zostaje kilka egzemplarzy programu *Notatnik*. W poleceniu wyłączającym ten proces użyto parametru `-confirm`. Następuje zamknięcie pierwszego egzemplarza i zawieszenie polecenia. Wtedy mamy okazję użyć polecenia `Get-Process`, aby zdobyć informacje o pozostałych uruchomionych procesach.

```
PS C:\> 1..5 | % notepad
PS C:\> 1..5 | % {notepad}
PS C:\> Get-Process -Name notepad | Stop-Process -Confirm

Confirm
Are you sure you want to perform this action?
Performing operation "Stop-Process" on Target "notepad (3552)".
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help
(default is "Y"):y

Confirm
Are you sure you want to perform this action?
Performing operation "Stop-Process" on Target "notepad (5404)".
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help
(default is "Y"):s
PS C:\>> get-process notepad
```

| Handles | NPM(K) | PM(K) | WS(K) | VM(M) | CPU(s) | Id | ProcessName |
|---------|--------|-------|-------|-------|--------|------|-------------|
| 81 | 9 | 1688 | 11328 | 98 | 0.03 | 5404 | notepad |
| 81 | 9 | 1680 | 11480 | 98 | 0.06 | 6344 | notepad |
| 81 | 9 | 1676 | 11364 | 98 | 0.05 | 6868 | notepad |
| 81 | 9 | 1680 | 11312 | 98 | 0.00 | 7092 | notepad |

```
PS C:\>> exit
```

```
Confirm
Are you sure you want to perform this action?
Performing operation "Stop-Process" on Target "notepad (5404)".
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help
(default is "Y"):a
PS C:\>
```

Praca z konsolą Windows PowerShell

Konsoli Windows PowerShell można używać zamiast wiersza poleceń. Zawiera ona wiele poleceń cmdlet, za pomocą których można wykonać wiele różnych czynności. Poleceń tych można używać zarówno osobno, jak i grupowo.

Zapiski praktyka

Bill Mell, menedżer infrastruktury MCSE

DAV

Używam konsoli Windows PowerShell od ponad pięciu lat, a zacząłem jej używać po przeczytaniu książki *Windows PowerShell* Eda Wilsona. To właśnie dzięki tej lekturze zrozumiałem, jakie dokładnie możliwości daje to narzędzie. Konsola Windows PowerShell bardzo ułatwia i przyspiesza wykonywanie żmudnych oraz skomplikowanych zadań. Używamy jej na przykład do pobierania szczegółowych informacji, takich jak znaczniki usług czy numery seryjne serwerów w naszym środowisku, których mamy już ponad 200, i liczba ta ciągle rośnie. Zdobycie tych wszystkich informacji ręcznie wymagałoby kilku dni, a może nawet całego tygodnia. Natomiast przy użyciu konsoli Windows PowerShell uwijamy się w ciągu kilku minut. Innym bardzo przydatnym dodatkiem są polecenia cmdlet Active Directory. Dzięki nim zautomatyzowałem żmudne zadania, jak tworzenie dużych ilości użytkowników i grup. To, co kiedyś zajmowało godziny i całe dni, teraz zajmuje zaledwie minuty. Ponadto wydaje mi się, że coraz więcej firm zaczyna dostrzegać niewątpliwe zalety tego narzędzia. Dwie z nich, które od razu przychodzą mi do głowy, to Dell i VMWare. Wtyczka VMWare umożliwia pobieranie informacji o mapowaniach woluminów RDM i gościach, z którymi są związane. Ręczne pobieranie tych informacji byłoby bardzo czasochłonne. Krótko mówiąc: konsola Windows PowerShell pozwala oszczędzić mnóstwo czasu i dzięki niej wykonuję dwa razy więcej pracy w o połowę krótszym czasie. Nie wyobrażam sobie już pracy bez tego narzędzia.

Włączanie konsoli Windows PowerShell

Konsola Windows PowerShell jest gotowa do użytku od razu po instalacji. Ale włączanie jej poprzez naciśnięcie klawisza *Windows* i litery *R* w celu uruchomienia najpierw wiersza poleceń albo klikanie za pomocą myszy opcji *Start/Uruchom/Windows PowerShell* nie jest zbyt wygodne. (W systemie Windows 8 jest trochę lepiej, bo wystarczy wpisać *PowerShell* na ekranie startowym). W systemie Windows 8.1 przypinam Windows PowerShell i Windows PowerShell ISE do ekranu startowego i do paska zadań. W systemie Windows Server 2012 R2 w trybie podstawowym w ogóle zastąpiłem wiersz poleceń konsolą Windows PowerShell. Dla mnie jest to idealne rozwiązanie. Co więcej, uznałem, że jest to tak przydatne, że napisałem nawet specjalny skrypt. Można go wywołać przez skrypt logowania, aby automatycznie utworzył skrót na pulpicie. W systemie Windows 8.1 skrypt ten dodaje zarówno Windows PowerShell, jak i Windows PowerShell ISE do ekranu startowego i paska zadań. W systemie Windows 7 dodaje konsolę Windows PowerShell i Windows PowerShell ISE do paska zadań i menu Start. Skrypt ten działa tylko w języku angielskim, ale można go przystosować do innych języków, zmieniając wartości zmiennych *\$pinToStart* i *\$pinToTaskBar*.

UWAGA

Zasady posługiwania się skryptami Windows PowerShell zostały opisane w rozdziale 16. Znajdziesz w nim informacje, jak działa poniższy skrypt i jak go uruchomić.

Poniższy skrypt nazywa się *PinToStartAndTaskBar.ps1*.

PintoStartAndTaskBar.ps1

```
$pinToStart = "Przypnij do menu Start"
$pinToTaskBar = "Przypnij do paska zadań"
$file = @((Join-Path -Path $PSHOME -childpath "PowerShell.exe"),
           (Join-Path -Path $PSHOME -childpath "powershell_ise.exe") )
Foreach($f in $file)
{
    $path = Split-Path $f
    $shell=New-Object -com "Shell.Application"
    $folder=$shell.Namespace($path)
    $item = $folder.parsename((Split-Path $f -leaf))
    $verbs = $item.verbs()
    foreach($v in $verbs)
    {
        if($v.Name.Replace("&", "") -match $pinToStart){$v.DoIt()}
    }
    foreach($v in $verbs)
    {
        if($v.Name.Replace("&", "") -match $pinToTaskBar){$v.DoIt()} }
}
```

Konfigurowanie konsoli Windows PowerShell

W konsoli Windows PowerShell można skonfigurować wiele elementów, które można zapisać w pliku PSConsole. Do eksportowania pliku konfiguracyjnego konsoli służy polecenie cmdlet `Export-Console`:

```
PS C:\> Export-Console myconsole
```

Polecenie to spowoduje zapisanie w bieżącym katalogu pliku PSConsole o rozszerzeniu *psc1*. Plik ten ma format XML i dla standardowej konsoli jego zawartość wygląda tak:

```
<?xml version="1.0" encoding="utf-8"?>
<PSConsoleFile ConsoleSchemaVersion="1.0">
  <PSVersion>3.0</PSVersion>
  <PSSnapIns />
</PSConsoleFile>
```

Zapiski praktyka

Jeff Truman, inżynier platformy

Serve American Express

Zaciągnięto mnie na spotkanie prawie bez ostrzeżenia ze względu na to, że w firmie cały czas zachwalałem zalety pisania skryptów. Skrypt, o którego napisanie mnie poproszony, był całkiem ciekawy. Trzeba było pobrać najnowsze kopie zapasowe urządzeń sieciowych i spakować je w formacie ZIP dla partnera. W firmie do zarządzania konfiguracjami sieciowymi używamy narzędzia SolarWinds Network Configuration Manager. Program ten wykonuje co noc kopie zapasowe bieżących konfiguracji każdego urządzenia sieciowego i zapisuje je w folderze na serwerze centralnym.

Nie wydaje się to trudne, więc sprawdźmy, jakie konkretnie są wymagania. Loguję się do serwera. Uruchamiam ISE i zaczynam przekopywać katalogi, aby dotrzeć do folderu *Data Drive\Solarwinds\NCM\Backups*. W porządku, to wydaje się sensowne. Co teraz widzę? Ponad 100 folderów z nazwami urządzeń sieciowych. Otwieram pierwszy i widzę kolejnych ponad 25 folderów o nazwach utworzonych ze znaczników czasu. Otwieram jeden z tych folderów i widzę potrzebną mi konfigurację. Jest jeden haczyk: codzienna kopia zapasowa jest tworzona tylko dla niektórych urządzeń. Nie wiem, jaki będzie ostatni znacznik czasu w każdym z folderów. Zwykłego śmiertelnika zadanie to by przytłoczyło, ale znawcę konsoli PowerShell jedynie rozzmieszy. Poniżej znajduje się odpowiedni kod:

```
$Path = "D:\Program Files (x86)\SolarWinds\Orion\NCM\Config-Archive\"
$Folders = Get-ChildItem $Path
$FullFiles
Foreach($Folder in $Folders)
{
$FF = Get-ChildItem -Path "$Path$Folder" | Sort | Select-object -First 1
$File = Get-ChildItem -Path "$Path$Folder\$FF" | Where-object {$_.Name
-like "**Running*"}
$FullFiles += $File
}
```

Magia powyższego skryptu kryje się w pętli. Oczywiście większość tego kodu mógłbym wykonać w jednej linii przy użyciu potoku, ale chciałem pokazać nowicuszom, jak się tworzy i uruchamia skrypty.

```
$FF = Get-ChildItem -Path "$Path$Folder" | Sort | Select-object -First 1
- This gets the Child Items of the full path, sorts them and then select the newest timestamp.

$File = Get-ChildItem -Path "$Path$Folder\$FF" | Where-object {$_.Name -like "**Running*"} -
now get the Running config within this folder.
```

Przekazywanie opcji do poleceń cmdlet

Jedną z zalet konsoli Windows PowerShell jest standardowa składnia poleceń cmdlet. To bardzo ułatwia naukę obsługi samego narzędzia i jego języka. W tabeli 1.1 przedstawiono listę najczęściej używanych parametrów. Nie wszystkie polecenia je obsługują, ale jeśli tak, to zawsze w ten sam sposób, ponieważ parametry podlegają interpretacji przez sam mechanizm Windows PowerShell.

TABELA 1.1. Najczęściej spotykane parametry

| Parametr | Opis |
|--------------|---|
| -whatif | Powoduje, że polecenie nie zostanie wykonane, tylko zostanie wyświetlona informacja, co by się stało, gdyby wykonano to polecenie |
| -confirm | Powoduje, że przed wykonaniem polecenia konsola poprosi o potwierdzenie |
| -verbose | Powoduje dostarczenie większej ilości szczegółów niż normalnie |
| -debug | Powoduje wyświetlenie informacji diagnostycznych |
| -ErrorAction | Powoduje wykonanie pewnej czynności, gdy wystąpi błąd. Możliwe czynności to: continue, stop, silentlyContinue oraz inquire |

TABELA 1.1. Najczęściej spotykane parametry — ciąg dalszy

| Parametr | Opis |
|----------------|---|
| -ErrorVariable | Powoduje zapisanie informacji o błędzie w określonej zmiennej, oprócz standardowej zmiennej \$error |
| -Outvariable | Wskazuje zmienną do zapisania informacji wyjściowych |
| -OutBuffer | Powoduje przetrzymanie określonej liczby obiektów przed wywołaniem następnego polecenia w potoku |

UWAGA

Aby wyświetlić informacje pomocnicze na temat wybranego polecenia cmdlet, należy użyć polecenia `Get-Help`. Na przykład: aby przeczytać pomoc dotyczącą polecenia `Get-Process`, należy napisać `Get-Help Get-Process`.

Korzystanie z opcji pomocy

Jednym z pierwszych poleceń, jakie należy wykonać po uruchomieniu konsoli Windows PowerShell, jest `Update-Help`. Jest to konieczne, ponieważ konsola Windows PowerShell standardowo nie zawiera plików **pomocy**. To oczywiście nie znaczy, że nie ma w niej w ogóle samego mechanizmu pomocy, tylko że aby wyświetlić jakiekolwiek inne informacje niż podstawy składni, należy pobrać dodatkowe dane.

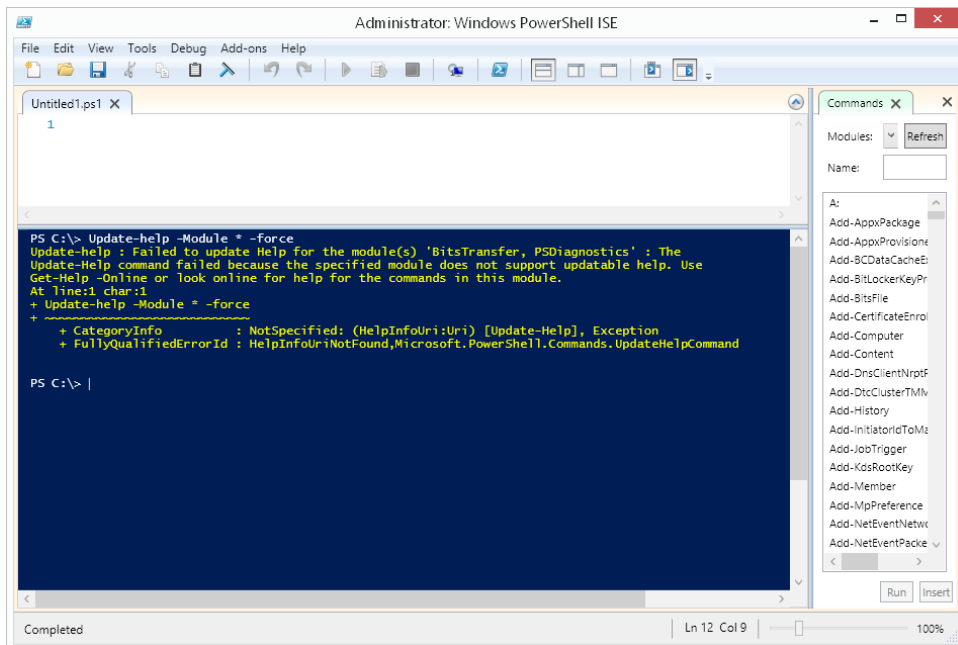
Domyślna instalacja konsoli Windows PowerShell 4.0 zawiera wiele modułów, które mogą się różnić w zależności od systemu operacyjnego i wybranych ról. Na przykład konsola Windows PowerShell 4.0 zainstalowana w systemie operacyjnym Windows 7 zawiera o wiele mniej modułów i poleceń cmdlet niż podobna instalacja w systemie Windows 8.1. Nie oznacza to jednak, że panuje chaos, ponieważ podstawowy zestaw poleceń jest taki sam w każdej instalacji. Różnice biorą się stąd, że różne dodatkowe funkcje systemowe i role powodują zainstalowanie różnych modułów i poleceń cmdlet. (Szczegółowe informacje na temat tego, które role powodują zainstalowanie różnych modułów, można znaleźć w portalu technet.microsoft.com — w tematach dotyczących ról i funkcji).

Aktualizowanie informacji pomocy

Modułowa budowa konsoli Windows PowerShell sprawia, że przy aktualizowaniu **pomocy** należy uwzględnić pewne dodatkowe uwarunkowania. Proste wykonanie polecenia `Update-Help` nie spowoduje zaktualizowania wszystkich modułów załadowanych w danym systemie. Tak naprawdę to niektóre moduły mogą nawet w ogóle nie obsługiwać funkcji aktualizacji **pomocy** i przy próbie aktualizacji zwracają błąd. Najprostszym sposobem na zaktualizowanie wszystkich możliwych składników pomocy jest użycie parametrów `module` i `force`. Poniżej znajduje się polecenie aktualizujące **pomoc** wszystkich zainstalowanych modułów (które obsługują funkcję aktualizacji **pomocy**):

```
Update-Help -Module * -Force
```

Na rysunku 1.1 pokazano wynik wykonania tego polecenia w typowym kliencie w systemie Windows 8.



RYSUNEK 1.1. Przy próbie zaktualizowania plików pomocy modułów nieobsługujących aktualizacji pomocy pojawiają się błędy

Jednym ze sposobów na zaktualizowanie **pomocy** bez otrzymywania powiadomień o błędach jest stłumienie błędów w poleceniu `Update-Help`, jak pokazano poniżej:

```
Update-Help -Module * -Force -ea 0
```

Wadą tego rozwiązania jest to, że nie wiadomo, czy **pomoc** została zaktualizowana dla wszystkich interesujących nas modułów. Lepszym rozwiązaniem jest ukrycie błędów podczas procesu aktualizacji i wyświetlenie ich po jego zakończeniu. Zaletą tego podejścia jest to, że błędy zostaną wyświetlone w bardziej czytelny sposób. Opisywaną techniką zaimplementowałem w skrypcie `UpdateHelpTrackErrors.ps1`. Skrypt ten zaczyna pracę od skasowania zawartości stosu błędów za pomocą metody `clear`. Następnie wywołuje polecenie `Update-Help` z parametrami `module` i `force`. Ponadto używa parametru `ErrorAction` (`ea` to jego alias) o wartości 0, która zakazuje wyświetlania informacji o błędach podczas działania polecenia. Na koniec pętla `For` wyświetla po kolei wszystkie błędy. Zawartość skryptu `UpdateHelpTrackErrors.ps1` przedstawiono poniżej.

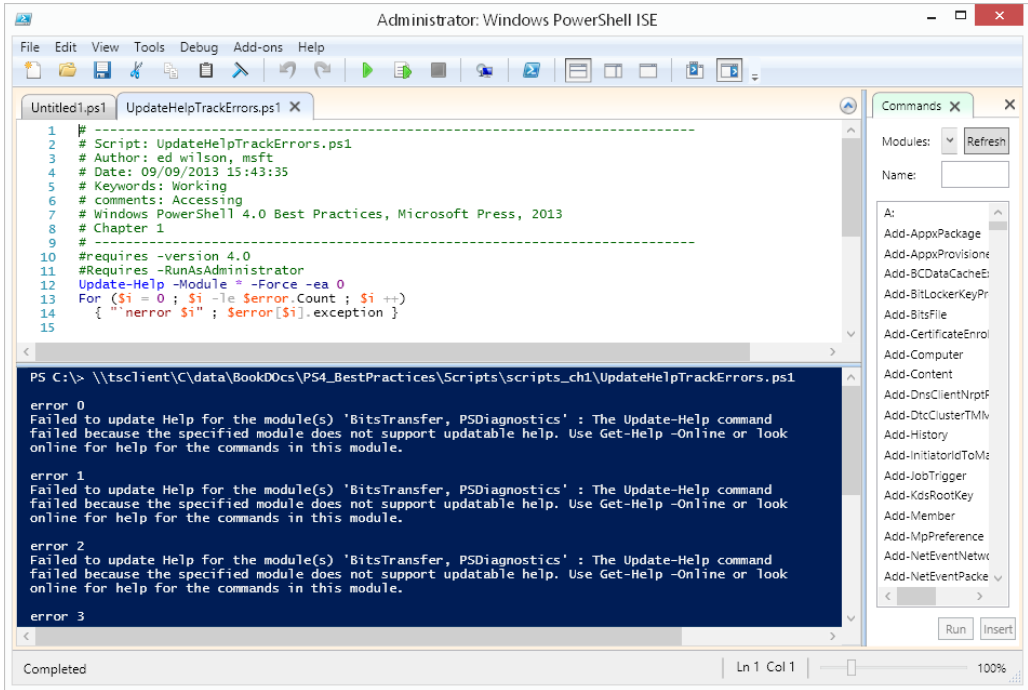
UWAGA

Informacje na temat pisania skryptów Windows PowerShell znajdują się w rozdziale 8.

UpdateHelpTrackErrors.ps1

```
#requires -version 4.0
#Requires -RunAsAdministrator
Update-Help -Module * -Force -ea 0
For ($i = 0 ; $i -le $error.Count ; $i ++ )
{ "'nerror $i" ; $error[$i].exception }
```

Po uruchomieniu skryptu *UpdateHelpTrackErrors.ps1* w konsoli pojawia się pasek postępu wskazujący stopień aktualizacji plików pomocy. Po zakończeniu procesu następuje wyświetlenie błędów. Na rysunku 1.2 widać sam skrypt i wyświetlone informacje o błędach.



RYSunek 1.2. Przejrzyste informacje o błędach wygenerowane przez skrypt *UpdatehelptrackErrors.ps1*

Zapiski praktyka

Jan Egil Ring, MVP Microsoft PowerShell, wiodący architekt
Crayon, Norwegia

Dobrym zwyczajem dotyczącym dającego się aktualizować systemu pomocy, który został wprowadzony w Windows PowerShell 3.0 lub 4.0, jest ciągle utrzymywanie aktualnych plików pomocy.

Jeśli trzeba raz zaktualizować pomoc, to wystarczy wykonać polecenie *Update-Help* w sesji Windows PowerShell ze zwiększonymi uprawnieniami. Ale pliki pomocy są aktualizowane regularnie i firma Microsoft utworzyła nawet kanał RSS, w którym informuje o nowych wydaniach: <http://sxp.microsoft.com/feeds/msdn/PowerShellHelpVersions>.

Podobnie jak w przypadku wszystkich innych zadań wykonywanych w konsoli Windows PowerShell kluczem jest automatyzacja. Do zaktualizowania własnego komputera można po prostu użyć modułu *PSScheduledJob* wprowadzonego w Windows PowerShell 3.0 i utworzyć zadanie wywołujące polecenie *Update-Help* powiedzmy raz w tygodniu i bez naszego udziału.

Poniżej znajduje się przykład takiego zadania:

```
Register-ScheduledJob -Name Update-Help -ScriptBlock {Update-Help -Module *}
-Trigger (New-JobTrigger -DaysOfWeek Monday -Weekly -At 8AM) -ScheduledJobOption
(New-ScheduledJobOption -RequireNetwork)
```

Mam listę rzeczy, które ustawiam, gdy instaluję świeży system albo konfiguruję nowy komputer dla siebie, i to zadanie się na niej znajduje.

W środowisku korporacyjnym należy uwzględnić więcej czynników. Na przykład istnieje ustawienie zasady grupy o nazwie „Set the default source path for Update-Help”, o którym można przeczytać na stronie <http://go.microsoft.com/fwlink/?LinkId=251696>. Ustawienie to pozwala na pobranie zaktualizowanych plików pomocy z jednego komputera, podczas gdy komputery przyłączone do domeny pobierają pliki pomocy z wewnętrznej ścieżki UNC określonej przez ustawienie zasady grupy lub za pomocą parametru `-SourcePath` polecenia `Update-Help`.

Jednym z powodów zastosowania takiego rozwiązania może być to, że niektóre komputery w sieci mogą nie mieć dostępu do internetu, przez co muszą pobierać pliki pomocy z miejsca w sieci wewnętrznej.

Na komputerze używanym do zarządzania pobieraniem plików pomocy można, posługując się poleceniem `Save-Help`, zapisać pliki pomocy na wewnętrznej ścieżce UNC. W Windows PowerShell 4.0 polecenie to rozszerzono o możliwość pobierania także plików pomocy dla modułów, których nie ma w komputerze, na którym wykonano to polecenie. Do tego celu służy parametr `-Module`, który przyjmuje jako wartość symbole wieloznaczne. Poniższe polecenie pobiera pliki pomocy dla wszystkich dostępnych modułów:

```
Save-Help -Module * -DestinationPath \\server\share
```

Ktoś może powiedzieć, że na serwerach nie trzeba aktualizować plików pomocy, ponieważ administratorzy powinni nimi zarządzać zdalnie i używać plików pomocy dostępnych w ich własnych komputerach. Ale wiele firm ma centralne serwery zarządzania, których używają pracownicy biura obsługi klienta i administratorzy. Serwery te, podobnie jak komputery klienckie z działu informatycznego, powinny być aktualizowane regularnie. Jednym ze sposobów robienia tego jest utworzenie przy użyciu preferencji zasad grupy zaplanowanego zadania, które będzie wywoływało polecenie `Update-Help` zgodnie z wyznaczonym harmonogramem. Poniżej znajduje się przykład ścieżki i argumentów, których można użyć do utworzenia takiego zaplanowanego zadania:

```
Path: C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe
Arguments: "& {if (($PSVersionTable.PSVersion).Major -ge 3)
{Update-Help -SourcePath '\\server\share\PowerShell\Help-files'}}"
```

Zdobywanie informacji w pomocy

Informacje na temat sposobu obsługi konsoli Windows PowerShell są bardzo łatwo dostępne i do ich znalezienia można użyć właśnie samej konsoli. Ważną rolę w zdobywaniu tych informacji odgrywa też **pomoc** internetowa. Do systemu **pomocy** Windows PowerShell można się dostać na kilka sposobów. Aby nauczyć się obsługi konsoli Windows PowerShell, użyj polecenia cmdlet `Get-Help` w następujący sposób:

```
Get-Help Get-Help
```

Polecenie to powoduje wydrukowanie pomocy na temat polecenia `Get-Help`. Wynik jego działania jest następujący:

NAME

Get-Help

SYNOPSIS

Displays information about Windows PowerShell commands and concepts.

SYNTAX

```
Get-Help [[-Name] <String>] [-Category <String>] [-Component <String>] [-Full
[<SwitchParameter>]] [-Functionality <String>] [-Path <String>] [-Role <String>]
[<CommonParameters>]
```

```
Get-Help [[-Name] <String>] [-Category <String>] [-Component <String>] [-Functionality
<String>] [-Path <String>] [-Role <String>] -Detailed [<SwitchParameter>] [<CommonParameters>]
```

```
Get-Help [[-Name] <String>] [-Category <String>] [-Component <String>] [-Functionality
<String>] [-Path <String>] [-Role <String>] -Examples [<SwitchParameter>]
[<CommonParameters>]
```

```
Get-Help [[-Name] <String>] [-Category <String>] [-Component <String>] [-Functionality
<String>] [-Path <String>] [-Role <String>] -Online [<SwitchParameter>] [<CommonParameters>]
```

```
Get-Help [[-Name] <String>] [-Category <String>] [-Component <String>] [-Functionality
<String>] [-Path <String>] [-Role <String>] -Parameter <String> [<CommonParameters>]
```

```
Get-Help [[-Name] <String>] [-Category <String>] [-Component <String>] [-Functionality
<String>] [-Path <String>] [-Role <String>] -ShowWindow [<SwitchParameter>]
[<CommonParameters>]
```

DESCRIPTION

The **Get-Help** cmdlet displays information about Windows PowerShell concepts and commands, including cmdlets, providers, functions, aliases and scripts.

Get-Help gets the **help** content that it displays from **help** files on your computer. Without the **help** files, **Get-Help** displays only basic information about commands. Some Windows PowerShell modules come with **help** files. However, beginning in Windows PowerShell 3.0, the modules that come with Windows PowerShell do not include **help** files. To download or update the **help** files for a module in Windows PowerShell 3.0, use the **Update-Help** cmdlet. You can also view the **help** topics for Windows PowerShell online in the TechNet Library at <http://go.microsoft.com/fwlink/?LinkID=107116><http://go.microsoft.com/fwlink/?LinkID=107116>.

To get **help** for a Windows PowerShell command, type "**Get-Help**" followed by the command name. To get a list of all **help** topics on your system, type "**Get-Help***".

Conceptual **help** topics in Windows PowerShell begin with "about_", such as "about_Comparison_Operators". To see all "about_" topics, type "**Get-Help** about_*". To see a particular topic, type "**Get-Help** about_<topic-name>", such as "**Get-Help** about_Comparison_Operators".

You can display the entire **help** topic or use the parameters of the **Get-Help** cmdlet to get selected parts of the topic, such as the syntax, parameters, or examples. You can also use the **Online** parameter to display an online version of a **help** topic for a command in your Internet browser.

If you type "**Get-Help**" followed by the exact name of a **help** topic, or by a word unique to a **help** topic, **Get-Help** displays the topic contents. If you enter a word or word pattern that

appears in several **help** topic titles, **Get-Help** displays a list of the matching titles. If you enter a word that does not appear in any **help** topic titles, **Get-Help** displays a list of topics that include that word in their contents.

In addition to "**Get-Help**", you can also type "**help**" or "**man**", which displays one screen of text at a time, or "<cmdlet-name> -?", which is identical to **Get-Help** but works only for cmdlets.

For information about the symbols that **Get-Help** displays in the command syntax diagram, see [about_Command_Syntax](http://go.microsoft.com/fwlink/?LinkID=113215)<http://go.microsoft.com/fwlink/?LinkID=113215>. For information about parameter attributes, such as Required and Position, see [about_Parameters](http://go.microsoft.com/fwlink/?LinkID=113243)<http://go.microsoft.com/fwlink/?LinkID=113243>.

RELATED LINKS

Online Version: <http://go.microsoft.com/fwlink/?LinkID=113316>

Get-Command

Get-Member

Get-PSDrive

[about_Command_Syntax](#)

[about_Comment_Based_Help](#)

[about_Parameters](#)

REMARKS

To see the examples, type: "Get-Help Get-Help-examples".

For more information, type: "Get-Help Get-Help-detailed".

For technical information, type: "Get-Help Get-Help-full".

For online help, type: "Get-Help Get-Help-online"

Zapiski praktyka

**Sean Kearney, MVP Microsoft PowerShell, starszy architekt rozwiązań
Cistel Technology Inc.**

Ledwie pamiętam, jak wyglądało moje życie, zanim zacząłem używać konsoli Windows PowerShell, ponieważ dopiero wtedy moje życie stało się szczęśliwe.

Już przy pierwszym zetknięciu z tą technologią dowiedziałem się, że jest nie tylko niezwykle łatwa w obsłudze, ale również bardzo przydatna, bo użyłem jej do usuwania plików według dat i godzin utworzenia. Dzięki temu narzędziu w piątki wychodzę z pracy do domu dwie godziny wcześniej, więc co piątek śpiewam pieśń dziękczynną. (Tak właśnie narodziła się w mojej głowie piosenka *Highway to PowerShell*).

Dzięki temu utworowi poznałem (za pośrednictwem wpisu na blogu, który prawie go wystraszył) Jeffreya Snovera, architekta PowerShell. Odkryłem całą społeczność obejmującą ludzi zarówno z firmy Microsoft, jak i spoza niej, których pasją było poprawianie i udoskonalanie tego niezwykle przydatnego programu. Społeczność ta prowadziła własne podcasty, tworzyła własne narzędzia i dopingowała firmę Microsoft do dalszego doskonalenia produktu. Grupa ta, w której skład wchodził także programiści z zespołu ds. Windows PowerShell w Microsoftzie, posiadacze tytułów Microsoft MVP i inni eksperci, przyczyniła się do tego, że przekroczyłem własne ograniczenia.

W mniej więcej tym samym czasie podczas nagrywania jakiegoś podcastu spotkałem Mr. Eda „Hey Scripting Guya” Wilsona. Chyba napisałem o nim jakiś krótki utwór i od tamtej pory jest moim przyjaciелеm.

Później podjąłem pracę w jednej z 15 największych firm na świecie, w której przeprowadzałem migrację użytkowników Active Directory i Exchange przy użyciu konsoli Windows PowerShell. Odkryłem, że był to najszybszy sposób na odblokowywanie kont i wyłączanie użytkowników (jako jedyny administrator w rozrastającym się dziale firmy). Ponadto konsoli tej używałem do sporządzania kwartalnych raportów dla SOX oraz codziennego zapewniania spójności tworzenia użytkowników. Za pomocą jednego skryptu Windows PowerShell proszącego o podanie nazwy użytkownika, numeru wewnętrznego i nazwy działu bez trudu zapełniałem sześć osobnych systemów i tworzyłem list dla ich menedżera z poświadczeniami i wprowadzeniem do naszego środowiska IT.

Dzięki konsoli Windows PowerShell mogłem korzystać z możliwości, jakie daje język VBScript, a jednocześnie robić to w wygodny i interaktywny sposób — jak w wierszu poleceń. W końcu dostałem narzędzie współpracujące ze starszymi systemami — nową technologię pozwalającą wnieść aplikacje konsolowe i VBScript na wyższy poziom.

Narzędzie to nic mnie nie kosztowało, a dzięki niemu zacząłem wracać do domu o przyzwoitej godzinie. Byłem zachwycony!

Teraz pracuję w firmie będącej złotym partnerem firmy Microsoft i obracam się w świecie automatyzacji i Systems Center 2012. Codziennie z przyjemnością idę do pracy. Gdy ktoś mnie spyta: „Umiesz to zrobić?”, od razu zaczynam się zastanawiać, jak to zrobić przy użyciu PowerShell, ponieważ to narzędzie umożliwia szybką i łatwą pracę. Ponadto każde rozwiązanie można łatwo powtórzyć.

Dziękuję konsoli Windows PowerShell i wszystkim, którzy nad nią pracują. Jestem Wam dożywotnie wdzięczny. Jesteście najlepsi.

Wielką zaletą systemu **pomocy** w Windows PowerShell jest to, że nie tylko wyświetla informacje o poleceniach, ale ma również trzy poziomy szczegółowości: normalny, szczegółowy oraz pełny. Ponadto można pobrać informacje dotyczące wybranych pojęć związanych z Windows PowerShell, co jest bardzo podobne do internetowej instrukcji obsługi. Aby wyświetlić listę wszystkich pojęciowych artykułów pomocy, należy użyć polecenia `Get-Help about*`, jak pokazano poniżej:

```
Get-Help about*
```

Powiedzmy, że zapomnieliśmy, jak dokładnie nazywa się potrzebne nam polecenie cmdlet, ale wiemy, że było typu `get`. Aby je znaleźć, możemy użyć symbolu wieloznacznego, np. gwiazdki, jak pokazano poniżej:

```
Get-Help get*
```

To nie jedyne zastosowanie tego symbolu wieloznacznego. Jeśli pamiętasz, że nazwa polecenia zaczyna się od słowa `get`, a po łączniku znajduje się słowo zaczynające się literą `p`, możesz napisać następujące polecenie:

```
Get-Help get-p*
```

Albo wyobraź sobie, że znasz nazwę polecenia, tylko nie możesz sobie przypomnieć jego składni. W takim przypadku możesz użyć argumentu `-examples`. Poniżej przedstawiono przykład zastosowania polecenia `Get-Help` z argumentem `-examples` do polecenia `Get-PSDrive`:

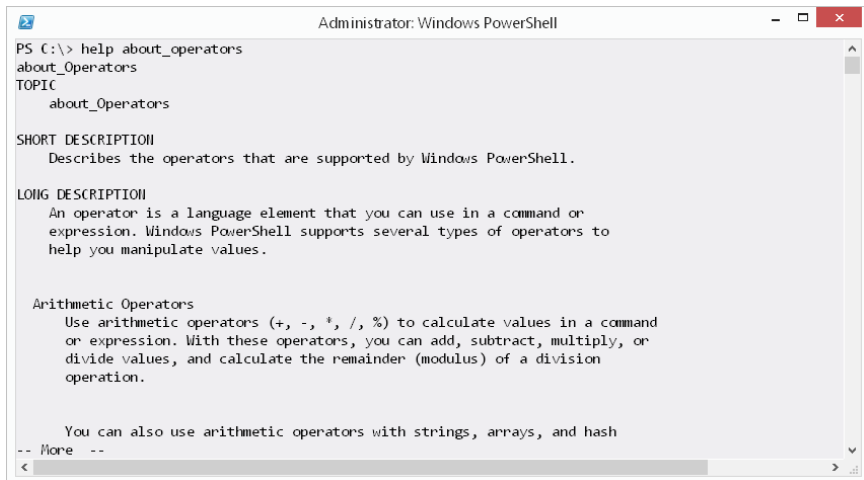
Get-Help Get-PSDrive -examples

Aby wyświetlić **pomoc** po jednej stronie, można użyć funkcji `help`. Funkcja ta przekazuje dane do polecenia `Get-Help`, a następnie wynik podaje potokowo do narzędzia *More.com*. Narzędzie to wyświetla dane w konsoli Windows PowerShell po jednej stronie. Jest to przydatne, gdy ktoś nie chce przewijać długich dokumentów.

UWAGA

W Windows PowerShell ISE stronicowanie nie działa, więc nie ma różnicy, czy użyje się polecenia `Get-Help`, czy `Help`. W konsoli tej oba te polecenia działają tak samo. Z drugiej strony, użytkownicy konsoli Windows PowerShell ISE częściej używają polecenia `Show-Command` zamiast `Get-Help`.

Sformatowany wynik jest pokazany na rysunku 1.3.



RYСУNEK 1.3. Wyświetlenie informacji na stronach przy użyciu polecenia `help`

Męczą Cię już ciągle wpisywanie polecenia `Get-Help`? W końcu to aż osiem znaków, z których jeden to łącznik. Jeśli tak, to możesz utworzyć **alias**, czyli skrót, za pomocą którego będzie można szybciej uruchamiać dane polecenie. Poniżej przedstawiono przykład, jak utworzyć alias `gh` dla polecenia `Get-Help`:

New-Alias gh Get-Help

UWAGA

Przed utworzeniem aliasu dla polecenia należy sprawdzić, czy nie ma ono już aliasu, za pomocą polecenia `Get-Alias`. Polecenie `New-Alias` służy do przypisywania poleceniom nowych alternatywnych nazw.

Zapiski praktyka

Don Jones, MVP Microsoft PowerShell

Concentrated Technologies

Kiedys miałem klienta, który był wielbicielem konsoli Windows PowerShell Web Access (PWA) i instalował ją na **wszystkich serwerach**. Tak, dobrze czytasz, **na każdym serwerze**. Skontaktował się ze mną, aby się dowiedzieć, czy da się to narzędzie zainstalować także na wszystkich komputerach klienckich. Powiedziałem: „Tak, ale zaczekaj”. PWA wymaga serwera Internet Information Services, a to z kolei oznacza, że na każdy serwer trzeba wrzucić mnóstwo kodu tylko po to, by uruchomić konsolę sieciową.

Klient ten nie wiedział, że konsola PWA jest bramą, którą instaluje się na jednym, maksymalnie paru serwerach. Administrator łączy się z nią poprzez protokół HTTPS (nie powinno się z nią pracować poprzez HTTP) i dokonuje uwierzytelniania na danym serwerze. Serwer pobiera dane poświadczające tego administratora i przy ich użyciu ustanawia zdalną sesję na dowolnym komputerze — kliencie lub serwerze — którym administrator chce zarządzać. PWA jest więc czymś w rodzaju pośrednika przepuszczającego ruch.

Gdy wyjaśniłem klientowi, w czym rzecz, szybko odinstalował konsolę PWA z większości serwerów i pozostawił ją tylko na paru z nich, którym wyznaczył zadanie bycia serwerami PWA. W ten sposób utworzył system równoważenia obciążenia round robin DNS, a dzięki posiadaniu dwóch serwerów uzyskał pewien stopień redundancji.

Dodatkowe źródła informacji

- W serwisie TechNet Script Center na stronie <http://www.microsoft.com/technet/scriptcenter> znajduje się wiele przykładowych skryptów.
- Na blogu Jana Egila Ringa na stronie <http://blog.powershell.no/2013/03/09/automatically-update-help-files-for-windows-powershell/> znajdują się szczegółowe informacje na temat wdrażania plików pomocy za pomocą zaplanowanego zadania używającego zasady grupy.
- Wszystkie skrypty opisane w tym rozdziale można pobrać z repozytorium Script Center pod adresem <http://gallery.technet.microsoft.com/scriptcenter/PowerShell-40-Best-d9e16039>.
- W centrum pobierania firmy Microsoft na stronie <http://www.microsoft.com/en-us/download/default.aspx> znajduje się do pobrania wiele materiałów dotyczących konsoli Windows PowerShell.

Rozdział 2

Polecenia CIM

- Przeglądanie klas usługi WMI za pomocą poleceń CIM
- Wyszukiwanie egzemplarzy klas WMI
- Praca z klasami skojarzeń
- Dodatkowe źródła informacji

Przeglądanie klas usługi WMI za pomocą poleceń CIM

Za pomocą poleceń cmdlet CIM można przeglądać klasy usługi WMI (ang. *Windows Management Instrumentation*) na wiele sposobów. Polecenia te bardzo dobrze działają w trybie interaktywnym. Na przykład za pomocą klawisza *Tab* można rozwijać nazwy przestrzeni nazw, co pozwala na odkrywanie takich przestrzeni, których znalezienie w inny sposób mogłoby być bardzo trudne. Za pomocą tej techniki można nawet szczegółowo badać zawartość przestrzeni nazw. Wszystkie klasy CIM obsługują rozwijanie za pomocą klawisza *Tab* parametru przestrzeni nazw, ale do badania klas WMI należy używać polecenia `Get-CimClass`.

UWAGA

Domyślna przestrzeń nazw WMI we wszystkich systemach operacyjnych od Windows NT 4.0 to `Root/Cimv2`. W związku z tym wszystkie polecenia cmdlet CIM domyślnie korzystają z przestrzeni nazw `Root/Cimv2`. Jedyne przypadki, w których należy zmienić tę domyślną przestrzeń nazw (przy użyciu parametru `namespace`), to używanie klasy WMI z innej przestrzeni nazw WMI niż domyślna.

Sposób użycia parametru `classname`

W poleceniu `Get-CimClass` w wartości parametru `classname` można używać symboli wieloznacznych, które umożliwiają szybkie znalezienie potencjalnie przydatnych klas WMI. Ponadto symboli wieloznacznych można też używać w wartości parametru `qualifiername`. Poniżej znajduje się przykład użycia polecenia `Get-CimClass` do wyszukania klas WMI związanych z komputerami.

```
PS C:\> Get-CimClass -ClassName *computer*
```

```
    Namespace: ROOT/CIMV2
```

| CimClassName | CimClassMethods | CimClassProperties |
|-------------------------------|----------------------|-------------------------|
| ----- | ----- | ----- |
| Win32_ComputerSystemEvent | {} | {SECURITY_DESCRIPTOR... |
| Win32_ComputerShutdownEvent | {} | {SECURITY_DESCRIPTOR... |
| CIM_ComputerSystem | {} | {Caption, Descripti... |
| CIM_UnitaryComputerSystem | {SetPowerState} | {Caption, Descripti... |
| Win32_ComputerSystem | {SetPowerState, R... | {Caption, Descripti... |
| Win32_ComputerSystemProcessor | {} | {GroupComponent, Pa... |
| CIM_ComputerSystemResource | {} | {GroupComponent, Pa... |
| CIM_ComputerSystemMappedIO | {} | {GroupComponent, Pa... |
| CIM_ComputerSystemDMA | {} | {GroupComponent, Pa... |
| CIM_ComputerSystemIRQ | {} | {GroupComponent, Pa... |
| CIM_ComputerSystemPackage | {} | {Antecedent, Depend... |
| Win32_ComputerSystemProduct | {} | {Caption, Descripti... |
| Win32_NTLogEventComputer | {} | {Computer, Record} |

UWAGA

Próba użycia symboli wieloznacznych w wartości parametru `classname` polecenia `Get-CimInstance` zakończy się zwróceniem przez konsolę błędu, ponieważ parametr ten nie przyjmuje symboli wieloznacznych.

Zapiski praktyka

Brian Wilhite, Premier Field Engineer (PFE)

Microsoft Corporation

W konsoli Windows PowerShell 3.0 firma Microsoft podbiła stawkę, dodając moduł `CimCmdlets`. Zawiera on wiele bardzo przydatnych funkcji i narzędzi, a do moich faworytów należy funkcja ułatwiająca znajdowanie klas WMI/CIM. Powiedzmy, że chcę sprawdzić, który dodatek serwisowy jest zainstalowany w komputerach mojego klienta. Kiedyś musiałbym przejrzeć zasoby witryny MSDN, aby znaleźć klasę o właściwościach spełniających moje kryteria. W zależności od sytuacji mogłoby mi to zająć od kilku minut do kilku godzin. Natomiast w konsoli Windows PowerShell 3.0 zadanie to jest bardzo łatwe. Wystarczy użyć polecenia `Get-CimClass` i poszukać w WMI nazw klas, metod, własności oraz kwalifikatorów za pomocą symboli wieloznacznych. W opisywanym przypadku poszukałbym nazwy własności z członem `ServicePack`:

```
Get-CimClass -PropertyName *ServicePack*
```

W normalnej sytuacji powyższe polecenie zwróci dwie klasy WMI: `Win32_OperatingSystem` i `Win32_QuickFixEngineering`. Mając je, mam dwie możliwości do wyboru. Po pierwsze: mogę odpytać klasy lokalnie, aby znaleźć własność odpowiadającą moim kryteriom, co jest świetnym rozwiązaniem, bo są tylko dwie klasy. Ale co zrobić, jeśli klas będzie dużo? W takim przypadku przetworzyłbym dane za pomocą poniższego polecenia, które należy wpisać w jednej linii:

```
Get-CimClass -PropertyName *ServicePack* |
ForEach-Object {$_ | Select-Object -Property CimClassName, '
@{L="CimClassProperties";E={$_.CimClassProperties.Name -like
"*ServicePack*"}}'}
```

Polecenie to zwróci własność CimClassName odpowiadającą zadanyim kryteriom CimClassProperties. Dzięki tej technice szybciej przeszukuję własności WMI, ponieważ nie muszę szukać w portalu MSDN nazw konkretnych klas, metod, własności ani kwalifikatorów.

Znajdowanie metod klas WMI

Aby znaleźć wszystkie klasy WMI odnoszące się do procesów, zawierające metodę o nazwie zaczynającej się od słowa term*, można użyć polecenia podobnego do poniższego:

```
PS C:\> Get-CimClass -ClassName *process* -MethodName term*
NameSpace: ROOT/cimv2
```

| CimClassName | CimClassMethods | CimClassProperties |
|---------------|----------------------|------------------------|
| Win32_Process | {Create, Terminat... | {Caption, Description, |
| Instal... | | |

Aby znaleźć wszystkie klasy WMI odnoszące się do procesów, które udostępniają jakiegokolwiek metody, można użyć poniższego polecenia.

```
PS C:\> Get-CimClass -ClassName *process* -MethodName *
NameSpace: ROOT/cimv2
```

| CimClassName | CimClassMethods | CimClassProperties |
|-----------------|----------------------|-----------------------------------|
| CIM_Processor | {SetPowerState, R... | {Caption, Description, Instal...} |
| Win32_Processor | {SetPowerState, R... | {Caption, Description, Instal...} |
| Win32_Process | {Create, Terminat... | {Caption, Description, Instal...} |

Aby znaleźć jakąkolwiek klasę WMI w przestrzeni nazw root/cimv2 udostępniającą metodę o nazwie create, należy użyć poniższego polecenia:

```
PS C:\> Get-CimClass -ClassName * -MethodName create
NameSpace: ROOT/cimv2
```

| CimClassName | CimClassMethods | CimClassProperties |
|-----------------------|----------------------|-----------------------------------|
| Win32_Process | {Create, Terminat... | {Caption, Description, Instal... |
| Win32_ScheduledJob | {Create, Delete} | {Caption, Description, Instal... |
| Win32_DfsNode | {Create} | {Caption, Description, Instal... |
| Win32_BaseService | {StartService, St... | {Caption, Description, Instal... |
| Win32_SystemDriver | {StartService, St... | {Caption, Description, Instal... |
| Win32_Service | {StartService, St... | {Caption, Description, Instal... |
| Win32_TerminalService | {StartService, St... | {Caption, Description, Instal... |
| Win32_Share | {Create, SetShare... | {Caption, Description, Instal... |
| Win32_ClusterShare | {Create, SetShare... | {Caption, Description, Instal... |
| Win32_ShadowCopy | {Create, Revert} | {Caption, Description, Instal... |
| Win32_ShadowStorage | {Create} | {AllocatedSpace, DiffVolume, ...} |

Filtrowanie klas według kwalifikatora

Aby znaleźć klasy WMI posiadające określony kwalifikator, należy użyć parametru `qualifier`. Na przykład poniższe polecenie znajduje klasy WMI związane z komputerami i mające kwalifikator `WMI.supportsupdate`:

```
PS C:\> Get-CimClass -ClassName *computer* -QualifierName *update
```

```
    Namespace: ROOT/cimv2
```

| CimClassName | CimClassMethods | CimClassProperties |
|----------------------|----------------------|----------------------------------|
| ----- | ----- | ----- |
| Win32_ComputerSystem | {SetPowerState, R... | {Caption, Description, Instal... |

Parametry można łączyć, aby tworzyć zaawansowane zapytania, które gdyby nie było poleceń `cmdlet` CIM, miałyby postać skomplikowanych skryptów. Na przykład poniższe polecenie znajduje wszystkie klasy WMI w przestrzeni nazw `root/cimv2`, które mają kwalifikator `singleton` i udostępniają jakąś metodę:

```
PS C:\> Get-CimClass -ClassName * -QualifierName singleton -MethodName *
```

```
    Namespace: ROOT/cimv2
```

| CimClassName | CimClassMethods | CimClassProperties |
|-------------------------|------------------------|----------------------------------|
| ----- | ----- | ----- |
| __SystemSecurity | {GetSD, GetSecuri... } | |
| Win32_OperatingSystem | {Reboot, Shutdown... } | {Caption, Description, Instal... |
| Win32_OfflineFilesCache | {Enable, RenameIt... } | {Active, Enabled, Location} |

Ważnym kwalifikatorem jest `deprecated`. Oznaczonych nim klas nie powinno się używać, ponieważ są wycofywane z użycia. Za pomocą polecenia `Get-CimClass` znalezienie tych klas jest bardzo łatwe, jak widać w poniższym przykładzie:

```
PS C:\> Get-CimClass * -QualifierName deprecated
```

```
    Namespace: ROOT/cimv2
```

| CimClassName | CimClassMethods | CimClassProperties |
|-------------------------------------|----------------------|----------------------------------|
| ----- | ----- | ----- |
| Win32_PageFile | {TakeOwnership, C... | {Caption, Description, Instal... |
| Win32_AllocatedResource | {} | {Antecedent, Dependent} |
| Win32_DisplayConfiguration | {} | {Caption, Description, Settin... |
| Win32_DisplayControllerConfigura... | {} | {Caption, Description, Settin... |
| Win32_VideoConfiguration | {} | {Caption, Description, Settin... |

Przy użyciu tej techniki można łatwo znaleźć klasy skojarzeń. (Więcej informacji na temat pracy z klasami skojarzeń WMI znajduje się w podrozdziale „Praca z klasami Association”). Poniższe polecenie znajduje wszystkie klasy WMI należące do przestrzeni nazw `root/cimv2`, które są związane z sesjami. Ponadto szuka kwalifikatora `association`. Na szczęście do wyszukiwania nazw kwalifikatorów można używać symboli wieloznacznych, dzięki czemu w poleceniu tym zamiast `association` napisałem `assoc*`:


```
PS C:\> Get-CimClass -ClassName *session* -QualifierName assoc*
```

```
Namespace: ROOT/cimv2
```

| CimClassName | CimClassMethods | CimClassProperties |
|------------------------------|-----------------|-------------------------|
| ----- | ----- | ----- |
| Win32_SubSession | {} | {Antecedent, Dependent} |
| Win32_SessionConnection | {} | {Antecedent, Dependent} |
| Win32_LogonSessionMappedDisk | {} | {Antecedent, Dependent} |
| Win32_SessionResource | {} | {Antecedent, Dependent} |
| Win32_SessionProcess | {} | {Antecedent, Dependent} |

Jednym z kwalifikatorów, który koniecznie trzeba sprawdzać, jest `dynamic`, ponieważ nie jest obsługiwany w zapytaniach o klasy **abstrakcyjne**. W związku z tym podczas szukania klas WMI należy przepuścić listę wyników przez filtr dotyczący tego kwalifikatora. W poniższym przykładzie znaleziono trzy klasy, które odnoszą się w jakiś sposób do czasu:

```
PS C:\> Get-CimClass -ClassName *time
```

```
Namespace: ROOT/cimv2
```

| CimClassName | CimClassMethods | CimClassProperties |
|-------------------|-----------------|----------------------------------|
| ----- | ----- | ----- |
| Win32_CurrentTime | {} | {Day, DayOfWeek, Hour, Millis... |
| Win32_LocalTime | {} | {Day, DayOfWeek, Hour, Millis... |
| Win32_UTCTime | {} | {Day, DayOfWeek, Hour, Millis... |

Dzięki dodaniu zapytania o kwalifikator znajduje się odpowiednie klasy WMI. Jedna jest abstrakcyjna, a dwie pozostałe są dynamiczne, więc mogą być przydatne. Poniżej znajdują się dwa polecenia. W pierwszym użyto zapytania o kwalifikator `dynamic`, a w drugim o kwalifikator `abstract`:

```
PS C:\> Get-CimClass -ClassName *time -QualifierName dynamic
```

```
Namespace: ROOT/cimv2
```

| CimClassName | CimClassMethods | CimClassProperties |
|-----------------|-----------------|----------------------------------|
| ----- | ----- | ----- |
| Win32_LocalTime | {} | {Day, DayOfWeek, Hour, Millis... |
| Win32_UTCTime | {} | {Day, DayOfWeek, Hour, Millis... |

```
PS C:\> Get-CimClass -ClassName *time -QualifierName abstract
```

```
Namespace: ROOT/cimv2
```

| CimClassName | CimClassMethods | CimClassProperties |
|-------------------|-----------------|----------------------------------|
| ----- | ----- | ----- |
| Win32_CurrentTime | {} | {Day, DayOfWeek, Hour, Millis... |

Wyszukiwanie egzemplarzy klas WMI

Do wysyłania zapytań dotyczących danych WMI służy polecenie cmdlet `Get-CimInstance`. Najprostszym sposobem jego użycia jest zapytanie o wszystkie własności i wszystkie egzemplarze wybranej klasy WMI na lokalnym komputerze. Jest to bardzo łatwe do wykonania. Poniższe polecenie zwraca informacje dotyczące systemu BIOS lokalnego komputera:

```
PS C:\> Get-CimInstance win32_bios

SMBIOSBIOSVersion : 1102
Manufacturer      : American Megatrends Inc.
Name              : BIOS Date: 12/09/10 20:41:43 Ver: 08.00.15
SerialNumber      : System Serial Number
Version           : ACRSYS - 20101209
```

Polecenie `Get-CimInstance` zwraca cały obiekt WMI, ale honoruje pliki w formacie XML, których konsola Windows PowerShell używa do określania, jakie własności dla danej klasy mają być wyświetlane domyślnie. Poniższe polecenie wyświetla własności dostępne w klasie WMI `Win32_Bios`:

```
PS C:\> $b = Get-CimInstance win32_bios
PS C:\> $b.CimClass.CimClassProperties | fw name -Column 3
```

| | | |
|--------------------|----------------------|-----------------------|
| Caption | Description | InstallDate |
| Name | Status | BuildNumber |
| CodeSet | IdentificationCode | LanguageEdition |
| Manufacturer | OtherTargetOS | SerialNumber |
| SoftwareElementID | SoftwareElementState | TargetOperatingSystem |
| Version | PrimaryBIOS | BiosCharacteristics |
| BIOSVersion | CurrentLanguage | InstallableLanguages |
| ListOfLanguages | ReleaseDate | SMBIOSBIOSVersion |
| SMBIOSMajorVersion | SMBIOSMinorVersion | SMBIOSPresent |

Zmniejszanie liczby zwróconych własności i egzemplarzy

Aby ograniczyć ilość danych zwracanych przez zdalne połączenie, należy zredukować liczbę własności i egzemplarzy. Aby zmniejszyć liczbę własności, należy użyć parametru `property`. Aby zmniejszyć liczbę zwracanych egzemplarzy, należy użyć parametru `filter`. W poniższym przykładzie użyto polecenia `gcim`, które jest aliasem polecenia `Get-CimInstance`. Ponadto skrócono też nazwy parametrów `classname` i `filter`. Jak widać, polecenie to zwraca tylko dane nazwy i stanu usługi bits. Podczas gdy domyślnie zwracane są wszystkie nazwy własności i własności systemowe, tylko dla dwóch wybranych własności zostały zwrócone dane.

```
PS C:\> gcim -clas win32_service -Property name, state -Fil "name = 'bits'"

Name           : BITS
Status         :
ExitCode       :
DesktopInteract :
ErrorControl   :
PathName       :
ServiceType    :
StartMode      :
Caption        :
Description    :
InstallDate    :
CreationClassName :
Started       :
SystemCreationClassName :
SystemName     :
AcceptPause    :
```

```

AcceptStop           :
DisplayName           :
ServiceSpecificExitCode :
StartName             :
State                 : Running
TagId                 :
CheckPoint            :
ProcessId             :
WaitHint              :
PSComputerName        :
CimClass               : root/cimv2:Win32_Service
CimInstanceProperties : {Caption, Description, InstallDate, Name...}
CimSystemProperties    : Microsoft.Management.Infrastructure.CimSystemProperties

```

Usuwanie niepotrzebnych informacji

Aby uzyskać bardziej czytelny wynik, wybrane dane można wysłać do polecenia Format-Table. Ponadto można użyć aliasu ft, aby nie musieć wpisywać tej długiej nazwy w całości.

```
PS C:\> gcim -clas win32_service -Property name, state -Fil "name = 'bits'" | ft name, state
```

```

name                state
----                -
BITS                Running

```

Należy wybrać te własności, które zostały wybrane w parametrze property, bo inaczej nic nie zostanie wyświetlone. W poniższym przykładzie wybrano własność status. Klasa WMI Win32_Service zawiera własność status, ale nie została ona wybrana podczas wybierania własności.

```
PS C:\> gcim -clas win32_service -Property name, state -Fil "name = 'bits'" | ft name, state, status
```

```

name                state        status
----                -
BITS                Running

```

Polecenie Get-CimInstance nie przyjmuje parametrów zawierających symbole wieloznaczne w nazwach własności (podobnie jak Get-WmiObject). Jednym ze sposobów na ułatwienie sobie pracy jest zapisanie wyboru własności w zmiennej. Dzięki temu tych samych nazw własności można używać zarówno w poleceniu Get-CimInstance, jak i Format-Table (a także Format-List, Select-Object i wielu innych), bez konieczności wpisywania ich za każdym razem od nowa. Zastosowanie tej metody przedstawiono w poniższym przykładzie:

```

PS C:\> $property = "name","state","startmode","startname"
PS C:\> gcim -clas win32_service -Pro $property -fil "name = 'bits'" | ft $property -A

```

```

name state startmode startname
---- -
BITS Running Manual LocalSystem

```

Praca z klasami Association

Kiedyś, w czasach gdy używało się jeszcze skryptów w języku VBScript, praca z klasami skojarzeń była bardzo skomplikowana. Była to bardzo niekomfortowa sytuacja, ponieważ klasy skojarzeń WMI są niezwykle przydatne. Wcześniejsze wersje konsoli Windows PowerShell ułatwiały pracę z nimi głównie dzięki temu, że ogólnie ułatwiały pracę z danymi WMI. Jednak mimo to wykorzystanie konsoli Windows PowerShell do tych celów było zaawansowaną umiejętnością. Na szczęście w Windows PowerShell 3.0 wprowadzono obsługę klas CIM, a wraz z nią pojawiło się polecenie `Get-CimAssociatedInstance`.

Pierwszą czynnością powinno być znalezienie egzemplarza klasy CIM i zapisanie go w zmiennej. W przykładzie przedstawionym poniżej pobierane są egzemplarze klasy `Win32_LogonSession`, które zostają zapisane w zmiennej `$logon`. Następnie za pomocą polecenia `Get-CimAssociatedInstance` pobieramy skojarzone z tą klasą egzemplarze. Aby dowiedzieć się, jakiego typu obiekty zostaną zwrócone przez polecenie, wynik potokowo przekazaliśmy do polecenia `Get-Member`. Jak widać, zwrócone zostały dwie klasy WMI: klasa `Win32_UserAccount` wraz z wszystkimi procesami odnoszącymi się do danego konta użytkownika w formie egzemplarzy klasy `Win32_Process`.

```
PS C:\> $logon = Get-CimInstance win32_logonsession
PS C:\> Get-CimAssociatedInstance $logon | Get-Member
```

TypeName: Microsoft.Management.Infrastructure.CimInstance#root/cimv2/Win32_UserAccount

| Name | MemberType | Definition |
|---------------------------|-------------|--|
| ----- | ----- | ----- |
| Clone | Method | System.Object ICloneable.Clone() |
| Dispose | Method | void Dispose(), void IDisposable.Dispose() |
| Equals | Method | bool Equals(System.Object obj) |
| GetCimSessionComputerName | Method | string GetCimSessionComputerName() |
| GetCimSessionInstanceId | Method | guid GetCimSessionInstanceId() |
| GetHashCode | Method | int GetHashCode() |
| GetObjectData | Method | void GetObjectData(System.Runtime.Serialization... |
| GetType | Method | type GetType() |
| ToString | Method | string ToString() |
| AccountType | Property | uint32 AccountType {get;} |
| Caption | Property | string Caption {get;} |
| Description | Property | string Description {get;} |
| Disabled | Property | bool Disabled {get;set;} |
| Domain | Property | string Domain {get;} |
| FullName | Property | string FullName {get;set;} |
| InstallDate | Property | CimInstance#DateTime InstallDate {get;} |
| LocalAccount | Property | bool LocalAccount {get;set;} |
| Lockout | Property | bool Lockout {get;set;} |
| Name | Property | string Name {get;} |
| PasswordChangeable | Property | bool PasswordChangeable {get;set;} |
| PasswordExpires | Property | bool PasswordExpires {get;set;} |
| PasswordRequired | Property | bool PasswordRequired {get;set;} |
| PSComputerName | Property | string PSComputerName {get;} |
| SID | Property | string SID {get;} |
| SIDType | Property | byte SIDType {get;} |
| Status | Property | string Status {get;} |
| PSStatus | PropertySet | PSStatus {Status, Caption, PasswordExpires} |

TypeName: Microsoft.Management.Infrastructure.CimInstance#root/cimv2/Win32_Process

| Name | MemberType | Definition |
|----------------------------|---------------|---|
| ----- | ----- | ----- |
| Handles | AliasProperty | Handles = Handlecount |
| ProcessName | AliasProperty | ProcessName = Name |
| VM | AliasProperty | VM = VirtualSize |
| WS | AliasProperty | WS = WorkingSetSize |
| Clone | Method | System.Object ICloneable.Clone() |
| Dispose | Method | void Dispose(), void IDisposable.Dispose() |
| Equals | Method | bool Equals(System.Object obj) |
| GetCimSessionComputerName | Method | string GetCimSessionComputerName() |
| GetCimSessionInstanceId | Method | guid GetCimSessionInstanceId() |
| GetHashCode | Method | int GetHashCode() |
| GetObjectData | Method | void GetObjectData(System.Runtime. |
| Serialization... | | |
| GetType | Method | type GetType() |
| ToString | Method | string ToString() |
| Caption | Property | string Caption {get;} |
| CommandLine | Property | string CommandLine {get;} |
| CreationClassName | Property | string CreationClassName {get;} |
| CreationDate | Property | CimInstance#DateTime CreationDate {get;} |
| CSCreationClassName | Property | string CSCreationClassName {get;} |
| CSName | Property | string CSName {get;} |
| Description | Property | string Description {get;} |
| ExecutablePath | Property | string ExecutablePath {get;} |
| ExecutionState | Property | uint16 ExecutionState {get;} |
| Handle | Property | string Handle {get;} |
| HandleCount | Property | uint32 HandleCount {get;} |
| InstallDate | Property | CimInstance#DateTime InstallDate {get;} |
| KernelModeTime | Property | uint64 KernelModeTime {get;} |
| MaximumWorkingSetSize | Property | uint32 MaximumWorkingSetSize {get;} |
| MinimumWorkingSetSize | Property | uint32 MinimumWorkingSetSize {get;} |
| Name | Property | string Name {get;} |
| OSCreationClassName | Property | string OSCreationClassName {get;} |
| OSName | Property | string OSName {get;} |
| OtherOperationCount | Property | uint64 OtherOperationCount {get;} |
| OtherTransferCount | Property | uint64 OtherTransferCount {get;} |
| PageFaults | Property | uint32 PageFaults {get;} |
| PageFileUsage | Property | uint32 PageFileUsage {get;} |
| ParentProcessId | Property | uint32 ParentProcessId {get;} |
| PeakPageFileUsage | Property | uint32 PeakPageFileUsage {get;} |
| PeakVirtualSize | Property | uint64 PeakVirtualSize {get;} |
| PeakWorkingSetSize | Property | uint32 PeakWorkingSetSize {get;} |
| Priority | Property | uint32 Priority {get;} |
| PrivatePageCount | Property | uint64 PrivatePageCount {get;} |
| ProcessId | Property | uint32 ProcessId {get;} |
| PSComputerName | Property | string PSComputerName {get;} |
| QuotaNonPagedPoolUsage | Property | uint32 QuotaNonPagedPoolUsage {get;} |
| QuotaPagedPoolUsage | Property | uint32 QuotaPagedPoolUsage {get;} |
| QuotaPeakNonPagedPoolUsage | Property | uint32 QuotaPeakNonPagedPoolUsage {get;} |
| QuotaPeakPagedPoolUsage | Property | uint32 QuotaPeakPagedPoolUsage {get;} |
| ReadOperationCount | Property | uint64 ReadOperationCount {get;} |
| ReadTransferCount | Property | uint64 ReadTransferCount {get;} |
| SessionId | Property | uint32 SessionId {get;} |
| Status | Property | string Status {get;} |
| TerminationDate | Property | CimInstance#DateTime TerminationDate {get;} |
| ThreadCount | Property | uint32 ThreadCount {get;} |
| UserModeTime | Property | uint64 UserModeTime {get;} |
| VirtualSize | Property | uint64 VirtualSize {get;} |
| WindowsVersion | Property | string WindowsVersion {get;} |

| | | |
|---------------------|----------------|---|
| WorkingSetSize | Property | uint64 WorkingSetSize {get;} |
| WriteOperationCount | Property | uint64 WriteOperationCount {get;} |
| WriteTransferCount | Property | uint64 WriteTransferCount {get;} |
| Path | ScriptProperty | System.Object Path {get=\$this.ExecutablePath;} |

Gdy polecenie wykonamy bez przekazania wyniku do polecenia Get-Member, najpierw zwrócony zostanie egzemplarz klasy WMI Win32_UserAccount. W wyniku widać nazwę użytkownika, typ konta, identyfikator SID, domenę oraz podpis konta użytkownika. Natomiast w danych zwróconych przez polecenie Get-Member znajduje się o wiele więcej informacji, ale wynik domyślny jest taki, a nie inny. W danych dotyczących konta użytkownika domyślnie zwrócone zostały następujące informacje: identyfikator procesu, nazwa oraz trochę danych dotyczących wydajności w odniesieniu do procesów skojarzonych z kontem użytkownika.

```
PS C:\> $logon = Get-CimInstance win32_logonsession
PS C:\> Get-CimAssociatedInstance $logon
```

| Name | Caption | AccountType | SID | Domain |
|----------------|------------------|-------------|-------------------|---------|
| ---- | ----- | ----- | --- | ----- |
| ed | IAMMRED\ed | 512 | S-1-5-21-14579... | IAMMRED |
| ProcessId | : 2780 | | | |
| Name | : taskhostex.exe | | | |
| HandleCount | : 215 | | | |
| WorkingSetSize | : 8200192 | | | |
| VirtualSize | : 242356224 | | | |
| ProcessId | : 2804 | | | |
| Name | : rdpclip.exe | | | |
| HandleCount | : 225 | | | |
| WorkingSetSize | : 8175616 | | | |
| VirtualSize | : 89419776 | | | |
| ProcessId | : 2352 | | | |
| Name | : explorer.exe | | | |
| HandleCount | : 1078 | | | |
| WorkingSetSize | : 65847296 | | | |
| VirtualSize | : 386928640 | | | |
| ProcessId | : 984 | | | |
| Name | : powershell.exe | | | |
| HandleCount | : 577 | | | |
| WorkingSetSize | : 94527488 | | | |
| VirtualSize | : 690466816 | | | |
| ProcessId | : 296 | | | |
| Name | : conhost.exe | | | |
| HandleCount | : 54 | | | |
| WorkingSetSize | : 7204864 | | | |
| VirtualSize | : 62164992 | | | |

Jeśli nie chcesz otrzymać obu klas z zapytania skojarzeniowego, możesz wymienić nazwę klasy. W tym celu użyj parametru resultclassname polecenia Get-CimAssociatedInstance. W poniższym przykładzie zapytanie zwraca tylko klasę WMI Win32_UserAccount.

```
PS C:\> $logon = Get-CimInstance win32_logonsession
PS C:\> Get-CimAssociatedInstance $logon -ResultClassName win32_useraccount
```

| Name | Caption | AccountType | SID | Domain |
|------|------------|-------------|-------------------|---------|
| ---- | ----- | ----- | --- | ----- |
| ed | IAMMRED\ed | 512 | S-1-5-21-14579... | IAMMRED |

Na wejściu polecenia `Get-CimAssociatedInstance` do parametru `inputobject` można podawać tylko obiekty będące pojedynczymi egzemplarzami. Jeśli zostanie przekazany obiekt zawierający więcej niż jeden egzemplarz klasy, zostanie zgłoszony błąd. Pokazano to w poniższym przykładzie, w którym przekazano do parametru `inputobject` więcej niż jeden dysk.

```
PS C:\> $disk = Get-CimInstance win32_logicaldisk
PS C:\> Get-CimAssociatedInstance $disk
Get-CimAssociatedInstance : Cannot convert 'System.Object[]' to the type
'Microsoft.Management.Infrastructure.CimInstance' required by parameter 'InputObject'. Określona
metoda nie jest obsługiwana.
At line:1 char:27
+ Get-CimAssociatedInstance $disk
+ ~~~~~
+ CategoryInfo          : InvalidArgument: (:) [Get-CimAssociatedInstance],
ParameterBindingException
+ FullyQualifiedErrorId : CannotConvertArgument,Microsoft.Management.Infrastructure.
CimCmdlets.GetCimAssociatedInstanceCommand
```

Błędowi temu można zaradzić na dwa sposoby. Pierwszy i łatwiejszy polega na użyciu indeksowania tablicowego, jak pokazano poniżej:

```
PS C:\> $disk = Get-CimInstance win32_logicaldisk
PS C:\> Get-CimAssociatedInstance $disk[0]
```

| Name | PrimaryOwner Name | Domain | TotalPhysical Memory | Model | Manufacturer |
|--------|----------------------|-------------|-------------------------|---------------|---------------|
| ---- | ----- | ----- | ----- | ---- | ----- |
| W8C504 | ed | iammred.net | 2147012608 | Virtual Ma... | Microsoft ... |

```
PS C:\> Get-CimAssociatedInstance $disk[1]
```

| Name | Hidden | Archive | Writeable | LastModified |
|------|--------|---------|-----------|--------------|
| ---- | ----- | ----- | ----- | ----- |

```
c:\

NumberOfBlocks : 265613312
BootPartition   : False
Name            : Disk #0, Partition #1
PrimaryPartition : True
Size            : 135994015744
Index           : 1

Domain          : iammred.net
Manufacturer    : Microsoft Corporation
Model           : Virtual Machine
Name            : W8C504
PrimaryOwnerName : ed
TotalPhysicalMemory : 2147012608
```

Użycie indeksów tablicowych jest możliwym rozwiązaniem, gdy parametr `inputobject` zawiera tablicę. Ale wyniki nie zawsze muszą być takie same. Dlatego lepszym wyjściem jest sprawienie, aby nigdy nie było tablicy. W tym celu należy za pomocą parametru `filter` zmniejszyć liczbę zwracanych egzemplarzy klasy. W poniższym przykładzie filtr zwraca liczbę egzemplarzy WMI napędu C:

```
PS C:\> $disk = Get-CimInstance win32_logicaldisk -Filter "name = 'c:'"
PS C:\> Get-CimAssociatedInstance $disk

Name                Hidden          Archive          Writeable         LastModified
----                -
c:\

NumberOfBlocks      : 265613312
BootPartition       : False
Name                : Disk #0, Partition #1
PrimaryPartition    : True
Size                : 135994015744
Index               : 1

Domain              : iammred.net
Manufacturer        : Microsoft Corporation
Model               : Virtual Machine
Name                : W8C504
PrimaryOwnerName    : ed
TotalPhysicalMemory : 2147012608
```

Prostym sposobem na podejrzenie obiektów zwróconych przez polecenie `Get-CimAssociatedInstance` jest przekazanie ich potokowo do polecenia `Get-Member` i wybranie własności `typename`. Jako że zwróconych może zostać kilka egzemplarzy obiektu, przez co wynik będzie mało czytelny, należy użyć parametru `unique`. Poniżej znajduje się to polecenie:

```
PS C:\> Get-CimAssociatedInstance $disk | gm | select typename -Unique

TypeName
-----
Microsoft.Management.Infrastructure.CimInstance#root/cimv2/Win32_Directory
Microsoft.Management.Infrastructure.CimInstance#root/cimv2/Win32_DiskPartition
Microsoft.Management.Infrastructure.CimInstance#root/cimv2/Win32_ComputerSystem
```

Mając te informacje, można z łatwością zbadać zwrócone klasy skojarzeń:

```
PS C:\> Get-CimAssociatedInstance $disk -ResultClassName win32_directory

Name                Hidden          Archive          Writeable         LastModified
----                -
c:\

PS C:\> Get-CimAssociatedInstance $disk -ResultClassName win32_diskpartition

Name                NumberOfBlocks  BootPartition  PrimaryPartition  Size          Index
----                -
Disk #0, Part...    265613312      False          True              135994015744  1

PS C:\> Get-CimAssociatedInstance $disk -ResultClassName win32_computersystem

Name                PrimaryOwner  Domain          TotalPhysical  Model          Manufacturer
----                -
W8C504              ed            iammred.net     2147012608     Virtual Ma...  Microsoft ...
```

Pamiętaj, że zwracana jest cała klasa WMI, którą można dalej badać. Najłatwiejszym sposobem na zrobienie tego jest zapisanie wyniku w zmiennej i przejrzanie danych. Po znalezieniu interesujących informacji można wyświetlić ładnie sformatowaną tabelę, jak pokazano poniżej:


```
PS C:\> $dp = Get-CimAssociatedInstance $disk -ResultClassName win32_diskpartition
PS C:\> $dp | FT deviceID, BlockSize, NumberOfBLinks, Size, StartingOffset -AutoSize
```

| deviceID | BlockSize | NumberOfBLinks | Size | StartingOffset |
|-----------------------|-----------|----------------|--------------|----------------|
| ----- | ----- | ----- | --- | ----- |
| Disk #0, Partition #1 | 512 | | 135994015744 | 368050176 |

Dodatkowe źródła informacji

- W centrum skryptowym w portalu TechNet na stronie <http://www.microsoft.com/technet/scriptcenter> znajduje się wiele przykładowych skryptów.
- Wszystkie skrypty opisane w tym rozdziale można pobrać z repozytorium Script Center pod adresem <http://gallery.technet.microsoft.com/scriptcenter/PowerShell-40-Best-d9e16039>.

CZĘŚĆ II

Planowanie skryptów

Rozdział 3. Moduł Active Directory

Rozdział 4. Znajdowanie możliwości zastosowania skryptów

Rozdział 5. Konfigurowanie środowiska skryptowego

Rozdział 6. Unikanie pułapek podczas pisania skryptów

Rozdział 7. Śledzenie możliwości zastosowania skryptów

Rozdział 3

Moduł Active Directory

- Podstawowe wiadomości o module Active Directory
- Zastosowania modułu Active Directory
- Dodatkowe źródła informacji

Podstawowe wiadomości o module Active Directory

Polecenia cmdlet Windows PowerShell do obsługi usług Active Directory Domain Services (AD DS) po raz pierwszy firma Microsoft udostępniła w systemie Windows Server 2008 R2. Można też pobrać i zainstalować usługę Active Directory Management Gateway Service (ADMGS) dostarczającą interfejs sieciowy do domen Active Directory lub Active Directory Lightweight Directory Services działających na tym samym komputerze co ADMGS. Usługa ADMGS może być uruchamiana w systemie Windows Server 2003 z dodatkiem Service Pack 2 i Windows Server 2008. W systemie Windows Server 2008 R2 i nowszych usługa ADMGS jest instalowana jako rola i nie trzeba jej dodatkowo pobierać. Jeśli w domenie znajduje się jeden kontroler domeny działający na serwerze Windows Server 2008 R2 lub nowszym, to można używać nowych poleceń cmdlet do zarządzania instalacją AD DS. Instalacja ADMGS w systemach Windows Server 2003 i Windows Server 2008 nie umożliwia załadowania modułu Active Directory, ale sprawia, że można używać go z innej maszyny do zarządzania tymi serwerami.

Wiedza tajemna

Ashley McGlone, Senior Premier Field Engineer
Microsoft Corporation

Niektórzy z nas używają konsoli Windows PowerShell od jej pierwszych wersji przygotowanych do wydania w 1999 roku. Inni z kolei zaczęli z niej korzystać dopiero niedawno. Ale wszystkich nas łączy jedno. Musimy automatyzować zadania dotyczące setek lub tysięcy użytkowników, komputerów, grup itd.

Przez lata do naszych podstawowych narzędzi należał język VBScript z ADSI lub narzędzia wiersza poleceń w rodzaju CSVDE i DSQUERY. (Niektórzy z nas do pracy z katalogami używali nawet WMI lub ADODB). Wszystkie te techniki dobrze nam służyły przez wiele lat.

Ale jesienią 2009 roku dokonał się zwrot. W systemie Windows Server 2008 R2 i narzędziach RSAT dla Windows 7 wprowadzono moduł Active Directory dla konsoli Windows PowerShell. Łał! To, do czego kiedyś trzeba było 20 wierszy kodu VBScript, dziś zapewni jedna linijka kodu Windows PowerShell.

Oto parę przykładów jednolinijkowych poleceń AD PowerShell:

```
Arkusze kalkulacyjny z informacjami o przeterminowanych kontach z 30 ostatnich dni:
Search-ADAccount -AccountInactive -TimeSpan 30 | Export-CSV .\Stale_Accts.csv
```

```
Prośba do pomocy technicznej o zresetowanie hasła użytkownika:
Set-ADAccountPassword (Read-Host 'Username') -Reset
```

```
Lista docelowa kontrolerów domeny wykazu globalnego:
(Get-ADForest).GlobalCatalogs
```

W swojej własnej pracy także napisałem parę większych skryptów, np.:

- Czyszczenie historii identyfikatorów SID Active Directory i migracje ACL serwera plików.
- Reorganizacja DNS i migracja do stref zintegrowanych z usługą Active Directory.
- Delegacja raportowania zabezpieczeń przez jednostki organizacyjne i obiekty GPO.

Na kontrolerze domeny jest to możliwe dzięki usłudze Active Directory Web Service (AWDS). Usługa ta nasłuchuje na porcie 9389 i reaguje na polecenia cmdlet konsoli Windows PowerShell. Nieważne, czy chcesz tylko wykonać proste jednowierszowe polecenie, czy zautomatyzować proces dla tysiąca kont, usługa ta ułatwia odczyt i zapis danych katalogowych.

W każdej kolejnej wersji systemu Windows Server moduł Active Directory (i nowe moduły uzupełniające) jest rozszerzany o nowe funkcje. W najnowszych wersjach dodano funkcje zastępujące znane i lubiane narzędzia, takie jak DCPROMO i REPADMIN. Ponadto moduł Group Policy zawiera jeszcze więcej narzędzi do automatyzacji zarządzania stacjami roboczymi przez Active Directory.

Moduł Active Directory dla Windows PowerShell nie jest już nowością. Jest to dojrzały produkt potrzebny każdemu administratorowi, któremu zależy na jak najsprawniejszym wykonywaniu pracy. Już dziś możesz zacząć używać tego modułu, wykonując jedno proste polecenie: `Import-Module ActiveDirectory`.

Instalowanie modułu Active Directory

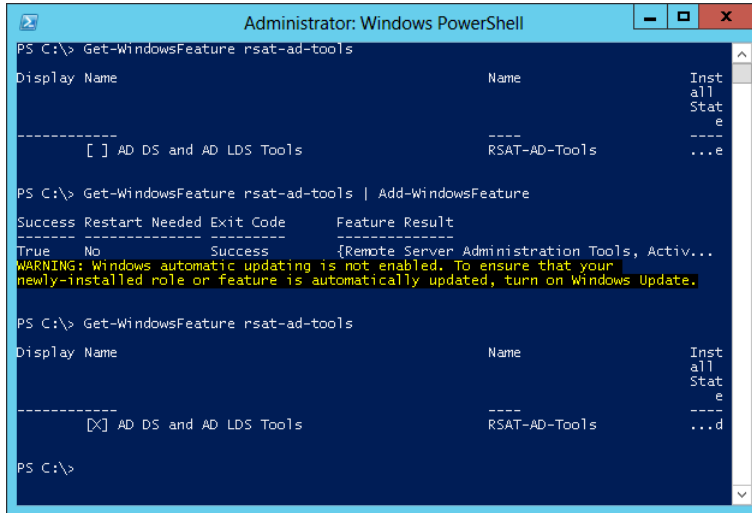
Moduł Active Directory jest dostępny od systemu Windows 7 po stronie klienckiej i od systemu Windows 2008 R2 na serwerach. Aby móc używać poleceń w systemach dla komputerów osobistych, należy pobrać i zainstalować narzędzia Remote Server Administration Tools (RSAT).

Do instalacji modułu Active Directory w systemie Windows Server 2012 lub Windows Server 2012 R2 można użyć polecenia `Add-WindowsFeature`, ponieważ moduł ten jest dostępny bezpośrednio jako opcjonalna funkcja systemowa. W związku z tym do instalacji w systemie serwerowym nie jest potrzebne pobieranie narzędzi RSAT. Aby zainstalować narzędzia RSAT

dla Active Directory, najpierw należy za pomocą polecenia cmdlet `Get-WindowsFeature` pobrać `rsat-ad-tools`, a następnie przekazać to do polecenia cmdlet `Add-WindowsFeature`. Poniżej pokazano, jak to zrobić:

`Get-WindowsFeature rsat-ad-tools | Add-WindowsFeature`

Wynik wykonania powyższego polecenia pokazano na rysunku 3.1.



RYSUNEK 3.1. Instalacja narzędzi RSAT daje dostęp do modułu Active Directory

Rozpoczynanie pracy z modułem Active Directory

Po zainstalowaniu narzędzi RSAT należy sprawdzić, czy moduł Active Directory jest dostępny i czy poprawnie się ładuje. W tym celu należy wykonać polecenie `Get-Module` z przełącznikiem `ListAvailable`. Spowoduje to wyświetlenie listy modułów, na której powinna znajdować się też pozycja `ActiveDirectory`. Poniżej znajduje się opisywane polecenie:

`Get-Module -ListAvailable ActiveDirectory`

Po załadowaniu modułu `ActiveDirectory` za pomocą polecenia `Get-Command` z parametrem `module` można wyświetlić listę wszystkich jego poleceń cmdlet. Jako że konsola Windows PowerShell 4.0 automatycznie ładuje moduły, nie trzeba ich importować własnoręcznie za pomocą polecenia `Import-Module`. Poniżej znajduje się opisywane polecenie:

`Get-Command -Module ActiveDirectory`

Zastosowanie modułu Active Directory

Nie zawsze trzeba ładować moduł Active Directory (ani tak naprawę jakiegokolwiek innego modułu), ponieważ konsole Windows PowerShell 3.0 i Windows PowerShell 4.0 automatycznie ładują moduł zawierający użyte polecenie cmdlet. Miejsce, w którym konsola szuka dostępnych modułów, jest określone za pomocą ścieżki ustawionej w zmiennej środowiskowej `PSModulePath`.

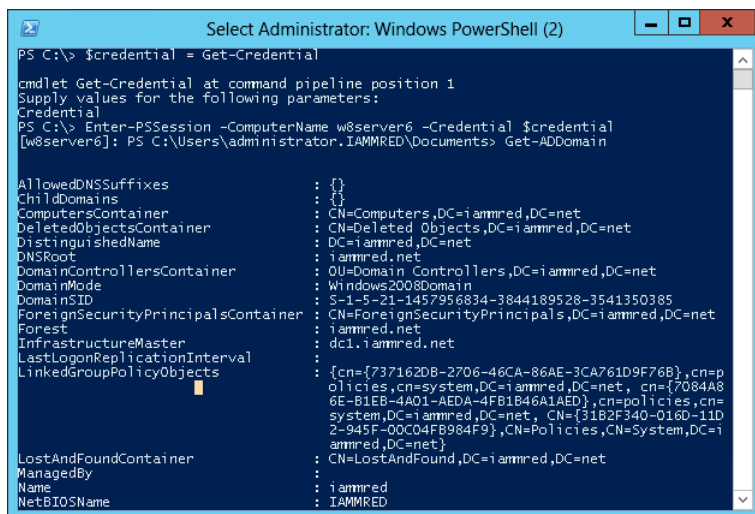
Aby wyświetlić wartość tej zmiennej, należy w konsoli wpisać przed nią `$env`. Poniższe polecenie pobiera domyślne lokalizacje modułów i wyświetla ścieżki:

```
PS C:\> $env:PSModulePath
C:\Users\ed.IAMMRED\Documents\WindowsPowerShell\Modules;C:\Program
Files\WindowsPowerShell\Modules;C:\Windows\system32\WindowsPowerShell\v1.0\Modules\
```

Jeśli nie chcesz instalować modułu Active Directory w swoim klienckim systemie operacyjnym, możesz tylko dodać funkcję `rsat-ad-tool`s przynajmniej do jednego serwera. Potem za pomocą konsoli Windows PowerShell połączysz się z tym serwerem ze swojej stacji roboczej. W trakcie sesji zdalnej, jeśli serwer zdalny jest systemem Windows 8, wystarczy wywołać dowolne z poleceń cmdlet Active Directory. Spowoduje to automatyczne załadowanie odpowiedniego modułu i zwrócenie odpowiednich informacji. Opisaną technikę ilustruje poniższe polecenie:

```
$credential = get-credential
Enter-PSsession -ComputerName w8Server6 -Credential $credential Get-ADDomain
```

Zastosowanie techniki polegającej na zdalnym połączeniu się za pomocą konsoli Windows PowerShell 4.0 z serwerem zawierającym moduł Active Directory i automatycznym załadowaniu tego modułu poprzez użycie należącego do niego polecenia cmdlet pokazano na rysunku 3.2.



RYСУNEK 3.2. Wykorzystanie funkcji pracy zdalnej konsoli Windows PowerShell 4.0 do pobrania informacji dotyczących Active Directory bez uprzedniego załadowania odpowiedniego modułu

Zapiski praktyka

Brian Wilhite, Premier Field Engineer (PFE)
Microsoft Corporation

Jak na administratora systemu Windows przyszło, prawdopodobnie przynajmniej raz w tygodniu, a może nawet codziennie korzystasz z usług Active Directory. Dzięki konsoli Windows PowerShell praca z tym narzędziem stała się znacznie łatwiejsza. Sprawiała ona, że zapominałem już,

jak skomplikowany potrafi być kod ADSI. Gdy instaluję świeżą kopię systemu Windows, to po dostosowaniu profilu pierwszą z moich czynności jest pobranie i zainstalowanie narzędzi RSAT (ang. *Remote Server Administration Tools*), aby od razu dać sobie dostęp do modułu ActiveDirectory w Windows PowerShell. Czasami kierownik prosi mnie o wysłanie zapytania do Active Directory, sprawdzającego, które komputery obsługują delegację, aby sprawdzić zgodność i ewentualnie wykonać w nich jakieś zadania. Pytania te oczywiście wysyłam za pomocą konsoli Windows PowerShell.

Po pierwsze, trzeba wiedzieć, jakich atrybutów Active Directory się szuka. W moim przypadku są to obiekty komputerowe z atrybutem `msDS-AllowedToDelegateTo` ustawionym na jakąkolwiek wartość lub atrybutem `TrustedForDelegation` o wartości `true`. W module Active Directory znajduje się polecenie `cmdlet`, przy użyciu którego bez trudu mogę wykonać takie zapytania. Zastanów się nad następującym przykładem:

```
Get-ADComputer '
-Filter {msDS-AllowedToDelegateTo -like "*" -or TrustedForDelegation -eq "True"} '
-Properties TrustedForDelegation, msDS-AllowedToDelegateTo |
Select Name, TrustedForDelegation, msDS-AllowedToDelegateTo
```

Polecenie to zwróci wszystkie obiekty komputerowe, których można użyć w celu delegacji dowolnej usługi lub wybranych usług. Teraz założmy, że chcemy sprawdzić, jakie aktualizacje do systemu Windows zostały w nich zainstalowane. Aby to zrobić, można wykonać poniższe polecenie (przy założeniu, że na komputerach docelowych włączone są funkcje pracy zdalnej konsoli Windows PowerShell). W poleceniu tym przekazujemy potokowo wynik do polecenia `Invoke-Command`, a następnie wykonujemy `Get-HotFix` na komputerze docelowym i zapisujemy wynik w zmiennej:

```
$Results = Get-ADComputer '
-Filter {msDS-AllowedToDelegateTo -like "*" -or TrustedForDelegation -eq "True"} '
-Properties TrustedForDelegation, msDS-AllowedToDelegateTo |
Select Name, TrustedForDelegation, msDS-AllowedToDelegateTo |
ForEach-Object {Invoke-Command -Command {Get-HotFix} -ComputerName $_.Name}
```

Po wykonaniu tego polecenia, co w zależności od liczby komputerów może zająć kilka minut, otrzymasz zgrabny raport. W razie potrzeby możesz nawet wysłać zawartość zmiennej do pliku CSV:

```
$Results | Export-Csv -Path C:\Temp\DelegationPatchReport.csv
```

Konsola Windows PowerShell w połączeniu z modułem ActiveDirectory ułatwia życie każdemu administratorowi, bez względu na rodzaj wykonywanych przez niego zadań.

Wyszukiwanie posiadaczy roli FSMO

Aby wyszukać informacje o kontrolerach domeny i rolach FSMO, nie trzeba pisać skryptów Windows PowerShell. Wystarczy użyć bezpośrednio konsoli Windows PowerShell lub Windows PowerShell ISE i poleceń `cmdlet` z modułu Active Directory. Pierwszą czynnością oczywiście powinno być załadowanie modułu ActiveDirectory do bieżącej sesji Windows PowerShell. Wprawdzie można dodać polecenie `import-module` do profilu Windows PowerShell, ale ładowanie za każdym razem wielu tylko okazjonalnie używanych modułów nie jest najlepszym pomysłem. W razie potrzeby można załadować wszystkie moduły naraz, przekazując wynik polecenia `Get-Module -ListAvailable` do polecenia `Import-Module`, jak pokazano poniżej:

```
PS C:\> Get-Module -ListAvailable | Import-Module
PS C:\> Get-Module
ModuleType Name                                     ExportedCommands
-----
Script     BasicFunctions                                {Get-ComputerInfo, Get-OptimalSize}
Script     ConversionModuleV6                            {ConvertTo-Feet, ConvertTo-Miles, ConvertTo-...
Script     PowerShellPack                                {New-ByteAnimationUsingKeyFrames, New-TiffBi...
Script     PSCodeGen                                     {New-Enum, New-ScriptCmdlet, New-PInvoke}
Script     PSImageTools                                  {Add-CropFilter, Add-RotateFlipFilter, Add-0...
Script     PSRss                                          {Read-Article, New-Feed, Remove-Article, Rem...
Script     PSSystemTools                                {Test-32Bit, Get-USB, Get-OSVersion, Get-Mul...
Script     PSUserTools                                  {Start-ProcessAsAdministrator, Get-CurrentUs...
Script     TaskScheduler                                {Remove-Task, Get-ScheduledTask, Stop-Task, ...
Script     WPK                                            {Get-DependencyProperty, New-ModelVisual3D, ...
Manifest   ActiveDirectory                              {Set-ADOrganizationalUnit, Get-ADDomainContr...
Manifest   AppLocker                                    {Get-AppLockerPolicy, Get-AppLockerFileInfor...
Manifest   BitsTransfer                                 {Start-BitsTransfer, Remove-BitsTransfer, Re...
Manifest   FailoverClusters                            {Set-ClusterParameter, Get-ClusterParameter,...
Manifest   GroupPolicy                                 {Get-GPStarterGPO, Get-GPOReport, Set-GPIInhe...
Manifest   NetworkLoadBalancingCl...                    {Stop-NlbClusterNode, Remove-NlbClusterVip, ...
Script     PSDiagnostics                                {Enable-PSTrace, Enable-WSManTrace, Start-Tr...
Manifest   TroubleshootingPack                          {Get-TroubleshootingPack, Invoke-Troubleshoo...
```

PS C:\>

Po załadowaniu modułu Active Directory należy wykonać polecenie `Get-Command`, aby wyświetlić polecenia eksportowane przez ten moduł, jak pokazano poniżej:

```
PS C:\> Get-Module -ListAvailable
ModuleType Name                                     ExportedCommands
-----
Script     BasicFunctions                                {}
Script     ConversionModuleV6                            {}
Script     DotNet                                         {}
Manifest   FileSystem                                    {}
Manifest   IsePack                                       {}
Manifest   PowerShellPack                                {}
Manifest   PSCodeGen                                     {}
Manifest   PSImageTools                                  {}
Manifest   PSRSS                                         {}
Manifest   PSSystemTools                                {}
Manifest   PSUserTools                                  {}

Manifest   TaskScheduler                                {}
Manifest   WPK                                            {}
Manifest   ActiveDirectory                              {}
Manifest   AppLocker                                    {}
Manifest   BitsTransfer                                 {}
Manifest   FailoverClusters                            {}
Manifest   GroupPolicy                                 {}
Manifest   NetworkLoadBalancingCl...                    {}
Manifest   PSDiagnostics                                {}
Manifest   TroubleshootingPack                          {}

PS C:\> Import-Module active*
PS C:\> Get-Command -Module active*
```

| CommandType | Name | Definition |
|-------------|------|------------|
| ----- | ---- | ----- |

```

Cmdlet      Add-ADComputerServiceAccount      Add-ADComputerServiceAccount [...
Cmdlet      Add-ADDomainControllerPasswordR... Add-ADDomainControllerPassword...
Cmdlet      Add-ADFineGrainedPasswordPolicy... Add-ADFineGrainedPasswordPolic...
Cmdlet      Add-ADGroupMember              Add-ADGroupMember [-Identity] ...
Cmdlet      Add-ADPrincipalGroupMembership  Add-ADPrincipalGroupMembership...
Cmdlet      Clear-ADAccountExpiration        Clear-ADAccountExpiration [-Id...
Cmdlet      Disable-ADAccount                Disable-ADAccount [-Identity] ...
Cmdlet      Disable-ADOptionalFeature        Disable-ADOptionalFeature [-Id...
Cmdlet      Enable-ADAccount                  Enable-ADAccount [-Identity] <...
Cmdlet      Enable-ADOptionalFeature          Enable-ADOptionalFeature [-Ide...
Cmdlet      Get-ADAccountAuthorizationGroup   Get-ADAccountAuthorizationGrou...
Cmdlet      Get-ADAccountResultantPasswordR... Get-ADAccountResultantPassword...
Cmdlet      Get-ADComputer                    Get-ADComputer -Filter <String...
<output truncated>

```

Aby znaleźć jeden kontroler domeny, jeśli nie ma się pewności co do niego w swojej witrynie, można użyć przełącznika `discover` polecenia `Get-ADDomainController`. Należy tylko pamiętać, że parametr `discover` może zwrócić informacje z pamięci podręcznej. Jeśli chcesz mieć pewność, że polecenie zostanie wykonane całkiem od nowa, użyj dodatkowo przełącznika `forceDiscover`. Opisane techniki zilustrowano w poniższych przykładach:

```
PS C:\> Get-ADDomainController -Discover
```

```

Domain      : NWTraders.Com
Forest      : NWTraders.Com
HostName    : {HyperV.NWTraders.Com}
IPv4Address : 192.168.1.100
IPv6Address :
Name        : HYPERV
Site        : NewBerlinSite

```

```
PS C:\> Get-ADDomainController -Discover -ForceDiscover
```

```

Domain      : NWTraders.Com
Forest      : NWTraders.Com
HostName    : {HyperV.NWTraders.Com}
IPv4Address : 192.168.1.100
IPv6Address :
Name        : HYPERV
Site        : NewBerlinSite

```

```
PS C:\>
```

Polecenie `Get-ADDomainController` zwraca bardzo niewielką ilość danych. Aby wyświetlić więcej informacji z odkrytego kontrolera domeny, należy się z nim połączyć za pomocą parametru `identity`. Wartością tego parametru może być adres IP, identyfikator GUID, nazwa hosta, a nawet nazwa NetBIOS. Spójrz na poniższy przykład:

```
PS C:\> Get-ADDomainController -Identity hyperv
```

```

ComputerObjectDN      : CN=HYPERV,OU=Domain Controllers,DC=NWTraders,DC=Com
DefaultPartition      : DC=NWTraders,DC=Com
Domain                 : NWTraders.Com
Enabled                : True
Forest                 : NWTraders.Com
HostName               : HyperV.NWTraders.Com
InvocationId           : 6835f51f-2c77-463f-8775-b3404f2748b2

```

```

IPv4Address      : 192.168.1.100
IPv6Address      :
IsGlobalCatalog  : True
IsReadOnly       : False
LdapPort         : 389
Name             : HYPERV
NTDSSettingsObjectDN : CN=NTDS Settings,CN=HYPERV,CN=Servers,CN=NewBerlinSite,
                  CN=Sites,CN=Configuration,DC=NWTraders,DC=Com
OperatingSystem   : Windows Server 2008 R2 Standard
OperatingSystemHotfix :
OperatingSystemServicePack :
OperatingSystemVersion : 6.1 (7600)
OperationMasterRoles : {SchemaMaster, DomainNamingMaster}
Partitions        : {DC=ForestDnsZones,DC=NWTraders,DC=Com, DC=DomainDnsZones,DC=NWTraders,DC=Com, CN=Schema,CN=Configuration,DC=NWTraders,DC=Com, CN=Configuration,DC=NWTraders,DC=Com...}
ServerObjectDN    : CN=HYPERV,CN=Servers,CN=NewBerlinSite,CN=Sites,CN=Configuration,DC=NWTraders,DC=Com
ServerObjectGuid   : ab5e2830-a4d6-47f8-b2b4-25757153653c
Site              : NewBerlinSite
SslPort           : 636
PS C:\>

```

Jak widać w wyniku powyższego polecenia, serwer o nazwie Hyperv jest serwerem wykazu globalnego. Ponadto posiada role operacji FSMO SchemaMaster i DomainNamingMaster. Działa w systemie Windows Server 2008 R2, co stanowi dowód na to, że polecenie to działa także ze starszymi wersjami systemu operacyjnego. Polecenie `Get-ADDomainController` przyjmuje parametr `filter`, za pomocą którego można wykonać wyszukiwanie i pobieranie danych. W parametrze tym używa się specjalnej składni, której opis znajduje się w internetowych plikach pomocy tematycznej. Niestety nie obsługuje składni LDAP.

Na szczęście nie trzeba uczyć się specjalnej składni filtrów, ponieważ polecenie `Get-ADObject` przyjmuje filtry w składni będącej dialektem LDAP. Wynik tego polecenia można natomiast przekazać do polecenia `Get-ADDomainController`, jak pokazano poniżej:

```

PS C:\> Get-ADObject -LDAPFilter "(objectclass=computer)" -searchbase "ou=domain controllers,dc=nwtraders,dc=com" | Get-ADDomainController

ComputerObjectDN      : CN=HYPERV,OU=Domain Controllers,DC=NWTraders,DC=Com
DefaultPartition      : DC=NWTraders,DC=Com
Domain                : NWTraders.Com
Enabled               : True
Forest               : NWTraders.Com
HostName              : HyperV.NWTraders.Com
InvocationId          : 6835f51f-2c77-463f-8775-b3404f2748b2
IPv4Address           : 192.168.1.100
IPv6Address           :
IsGlobalCatalog       : True
IsReadOnly            : False
LdapPort              : 389
Name                 : HYPERV
NTDSSettingsObjectDN  : CN=NTDS Settings,CN=HYPERV,CN=Servers,CN=NewBerlinSite,
                  CN=Sites,CN=Configuration,DC=NWTraders,DC=Com

OperatingSystem       : Windows Server 2008 R2 Standard
OperatingSystemHotfix :
OperatingSystemServicePack :
OperatingSystemVersion : 6.1 (7600)

```

```

OperationMasterRoles      : {SchemaMaster, DomainNamingMaster}
Partitions                 : {DC=ForestDnsZones,DC=NWTraders,DC=Com, DC=DomainDnsZones,
                             DC=NWTraders,DC=Com, CN=Schema,CN=Configuration,DC=NWTraders,DC=Com,
                             CN=Configuration,DC=NWTraders,DC=Com...}

ServerObjectDN             : CN=HYPERV,CN=Servers,CN=NewBerlinSite,CN=Sites,
                             CN=Configuration,DC=NWTraders,DC=Com
ServerObjectGuid           : ab5e2830-a4d6-47f8-b2b4-25757153653c
Site                       : NewBerlinSite
SslPort                    : 636

ComputerObjectDN           : CN=DC1,OU=Domain Controllers,DC=NWTraders,DC=Com
DefaultPartition           : DC=NWTraders,DC=Com
Domain                     : NWTraders.Com
Enabled                    : True
Forest                     : NWTraders.Com
HostName                   : DC1.NWTraders.Com
InvocationId               : fb324ced-bd3f-4977-ae69-d6763e7e029a
IPv4Address                : 192.168.1.101
IPv6Address                :
IsGlobalCatalog            : True
IsReadOnly                 : False
LdapPort                   : 389
Name                       : DC1
NTDSSettingsObjectDN       : CN=NTDS Settings,CN=DC1,CN=Servers,CN=NewBerlinSite,
                             CN=Sites,CN=Configuration,DC=NWTraders,DC=Com
OperatingSystem             : Windows Server 2008 Standard without Hyper-V
OperatingSystemHotfix      :
OperatingSystemServicePack : Service Pack 2
OperatingSystemVersion      : 6.0 (6002)
OperationMasterRoles       : {PDCEmulator, RIDMaster, InfrastructureMaster}
Partitions                 : {DC=ForestDnsZones,DC=NWTraders,DC=Com, DC=DomainDnsZones,
                             DC=NWTraders,DC=Com, CN=Schema,CN=Configuration,DC=NWTraders,DC=Com,
                             CN=Configuration,DC=NWTraders,DC=Com...}

ServerObjectDN             : CN=DC1,CN=Servers,CN=NewBerlinSite,
                             CN=Sites,CN=Configuration,DC=NWTraders,DC=Com
ServerObjectGuid           : 80885b47-5a51-4679-9922-d6f41228f211
Site                       : NewBerlinSite
SslPort                    : 636

PS C:\>

```

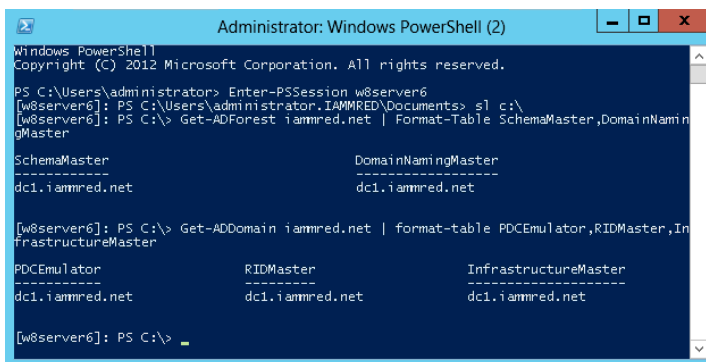
Jeśli informacji jest za dużo, to polecenia Active Directory działają tak samo jak wszystkie inne polecenia Windows PowerShell, więc można ich wynik przekazać potokowo do filtra. Do wyświetlenia jedynie informacji dotyczących FSMO potrzebne są dwa polecenia; trzy są potrzebne, jeśli chcemy dołączyć importowanie modułu Active Directory, a cztery — jeśli w celu wykonania poleceń chcemy nawiązać zdalne połączenie z kontrolerem domeny. Jedną z wielkich zalet narzędzi do pracy zdalnej konsoli Windows PowerShell jest to, że trzeba podać dane poświadczające potrzebne do wykonania polecenia. Jeśli mamy standardowe konto użytkownika, to konta o podwyższonych uprawnieniach używamy tylko wtedy, gdy chcemy wykonać wymagające tego zadania. Jeśli uruchomiłeś już konsolę z większymi uprawnieniami, nie musisz wpisywać danych poświadczających podczas włączania sesji pracy zdalnej (przy założeniu, że wśród tych uprawnień znajduje się też prawo do pracy na zdalnym serwerze). Dwa pierwsze pokazane poniżej polecenia tworzą sesję zdalną na zdalnym kontrolerze domeny i ładują moduł ActiveDirectory:

```
Enter-PSsession w8Server6
```

Po załadowaniu modułu Active Directory należy wpisać jednowierszowe polecenie pobierające role FSMO lasu i kolejne pobierające role FSMO domeny. Polecenia te pokazano poniżej:

```
Get-ADForest iammred.net | Format-Table SchemaMaster,DomainNamingMaster
Get-ADDomain iammred.net | format-table PDCEmulator,RIDMaster,InfrastructureMaster
```

To wszystko — dwa lub trzy polecenia, w zależności od tego, jak się liczy. Nawet w najgorszym przypadku, czyli przy trzech jednowierszowych poleceniach, i tak mamy o wiele lepiej, niż gdybyśmy musieli napisać 33-wierszowy skrypt, nie mając możliwości używania modułu Active Directory. Ponadto kod Windows PowerShell jest znacznie bardziej czytelny i zrozumiały. Przykład użycia opisywanych poleceń i ich wyniku pokazano na rysunku 3.3.



RYSUNEK 3.3. Pobieranie informacji dotyczących FSMO za pomocą narzędzi do pracy zdalnej konsoli Windows PowerShell

Dokumentowanie Active Directory

Przy użyciu poleceń z modułu Active Directory i narzędzi do pracy zdalnej konsoli Windows PowerShell można łatwo znaleźć informacje dotyczące lasu i domeny. Najpierw za pomocą polecenia `Enter-PSSession` należy rozpocząć sesję `PSSession` na komputerze zdalnym. Następnie należy zaimportować moduł Active Directory i ustawić katalog roboczy na folder główny dysku C. Ustawienie to pozwala zyskać trochę wolnego miejsca w wierszu poleceń. Poniżej pokazano opisywane polecenia:

```
PS C:\Users\Administrator.NWTRADERS> Enter-PSSession dc1
[dc1]: PS C:\Users\Administrator\Documents> Import-Module activedirectory
[dc1]: PS C:\Users\Administrator\Documents> Set-Location c:\
```

Po nawiązaniu połączenia ze zdalnym kontrolerem domeny za pomocą polecenia `Get-WmiObject` można sprawdzić system operacyjny na zdalnym komputerze. Poniżej pokazano efekt wykonania tego polecenia:

```
[dc1]: PS C:\> Get-WmiObject win32_operatingsystem
SystemDirectory : C:\Windows\system32
Organization    :
BuildNumber     : 7601
RegisteredUser  : Windows User
SerialNumber    : 55041-507-0212466-84005
Version        : 6.1.7601
```

Teraz trzeba pobrać informacje dotyczące lasu za pomocą polecenia `Get-ADForest`. Polecenie to zwraca mnóstwo bardzo przydatnych informacji, np. Domain Name Master, Forest Mode, Schema Master oraz Domain Controllers. Poniżej przedstawiono przykład użycia tego polecenia i jego wyniku:

```
[dc1]: PS C:\> Get-ADForest
ApplicationPartitions : {DC=DomainDnsZones,DC=nwtraders,DC=com,
DC=ForestDnsZones,DC=nwtraders,DC=com}
CrossForestReferences : {}
DomainNamingMaster    : DC1.nwtraders.com
Domains               : {nwtraders.com}
ForestMode            : Windows2008Forest
GlobalCatalogs       : {DC1.nwtraders.com}
Name                  : nwtraders.com
PartitionsContainer    : CN=Partitions,CN=Configuration,DC=nwtraders,DC=com
RootDomain            : nwtraders.com
SchemaMaster          : DC1.nwtraders.com
Sites                 : {Default-First-Site-Name}
SPNSuffixes           : {}
UPNSuffixes           : {}
```

Aby uzyskać informacje dotyczące domeny, należy użyć polecenia `Get-ADDomain`. Polecenie to zwraca wiele ważnych informacji, takich jak lokalizacja jednostki organizacyjnej domyślnego kontrolera domeny, emulator kontrolera PDC czy nadrzędny identyfikator RID. Poniżej pokazano przykładowy wynik tego polecenia:

```
[dc1]: PS C:\> Get-ADDomain
AllowedDNSSuffixes      : {}
ChildDomains            : {}
ComputersContainer      : CN=Computers,DC=nwtraders,DC=com
DeletedObjectsContainer : CN=Deleted Objects,DC=nwtraders,DC=com
DistinguishedName       : DC=nwtraders,DC=com
DNSRoot                 : nwtraders.com
DomainControllersContainer : OU=Domain Controllers,DC=nwtraders,DC=com
DomainMode              : Windows2008Domain
DomainSID               : S-1-5-21-909705514-2746778377-2082649206
ForeignSecurityPrincipalsContainer : CN=ForeignSecurityPrincipals,DC=nwtraders,DC=com
Forest                  : nwtraders.com
InfrastructureMaster     : DC1.nwtraders.com
LastLogonReplicationInterval :
LinkedGroupPolicyObjects : {CN={31B2F340-016D-11D2-945F-00C04FB984F9},
CN=Policies,CN=System,DC=nwtraders,DC=com}
LostAndFoundContainer    : CN=LostAndFound,DC=nwtraders,DC=com
ManagedBy               :
Name                     : nwtraders
NetBIOSName              : NWTRADERS
ObjectClass              : domainDNS
ObjectGUID               : 0026d1fc-2e4d-4c35-96ce-b900e9d67e7c
ParentDomain             :
PDCEmulator             : DC1.nwtraders.com
QuotasContainer          : CN=NTDS Quotas,DC=nwtraders,DC=com
ReadOnlyReplicaDirectoryServers : {}
ReplicaDirectoryServers  : {DC1.nwtraders.com}
RIDMaster                : DC1.nwtraders.com
```

```
SubordinateReferences      : {DC=ForestDnsZones,DC=nwtraders,DC=com,
                             DC=DomainDnsZones,DC=nwtraders,DC=com,
                             CN=Configuration,DC=nwtraders,DC=com}
SystemsContainer          : CN=System,DC=nwtraders,DC=com
UsersContainer            : CN=Users,DC=nwtraders,DC=com
```

Jeśli chodzi o bezpieczeństwo, to zawsze powinno się sprawdzać zasadę haseł domeny za pomocą polecenia `Get-ADDefaultDomainPasswordPolicy`. Powinno się zwrócić uwagę na poziom złożoności haseł, ich minimalną długość, wiek oraz zasady przechowywania. Ponadto należy sprawdzić zasadę blokady konta. Zasadzie tej należy przyrzeć się szczególnie uważnie, gdy dziedziczy się nową sieć. Poniżej znajduje się przykład odpowiedniego polecenia:

```
[dc1]: PS C:\> Get-ADDefaultDomainPasswordPolicy
ComplexityEnabled          : True
DistinguishedName         : DC=nwtraders,DC=com
LockoutDuration           : 00:30:00
LockoutObservationWindow  : 00:30:00
LockoutThreshold          : 0
MaxPasswordAge            : 42.00:00:00
MinPasswordAge            : 1.00:00:00
MinPasswordLength         : 7
objectClass               : {domainDNS}
objectGuid                : 0026d1fc-2e4d-4c35-96ce-b900e9d67e7c
PasswordHistoryCount      : 24
ReversibleEncryptionEnabled : False
```

Na koniec należy sprawdzić same kontrolery domen za pomocą polecenia `Get-ADDomainController`. Polecenie to również zwraca wiele ważnych informacji, np. czy kontroler jest tylko do odczytu, czy jest serwerem wykazu globalnego, oraz dane systemu operacyjnego. Poniżej przedstawiono przykładowy wynik wykonania tego polecenia:

```
[dc1]: PS C:\> Get-ADDomainController -Identity dc1
ComputerObjectDN          : CN=DC1,OU=Domain Controllers,DC=nwtraders,DC=com
DefaultPartition          : DC=nwtraders,DC=com
Domain                    : nwtraders.com
Enabled                   : True
Forest                    : nwtraders.com
HostName                  : DC1.nwtraders.com
InvocationId              : b51f625f-3f60-44e7-8577-8918f7396c2a
IPv4Address               : 10.0.0.1
IPv6Address               :
IsGlobalCatalog           : True
IsReadOnly                : False
LdapPort                  : 389
Name                      : DC1
NTDSSettingsObjectDN      : CN=NTDS Settings,CN=DC1,CN=Servers,CN=Default-First-Site-
                             Name,CN=Sites,CN=Configuration,DC=nwtraders,DC=com
OperatingSystem           : Windows Server 2008 R2 Enterprise
OperatingSystemHotfix     :
OperatingSystemServicePack : Service Pack 1
OperatingSystemVersion    : 6.1 (7601)
OperationMasterRoles      : {SchemaMaster, DomainNamingMaster, PDCEmulator, RIDMaster...}
Partitions                 : {DC=ForestDnsZones,DC=nwtraders,DC=com, DC=DomainDnsZones,
                             DC=nwtraders,DC=com, CN=Schema,CN=Configuration,
                             DC=nwtraders,DC=com, CN=Configuration,DC=nwtraders, DC=com...}
ServerObjectDN            : CN=DC1,CN=Servers,CN=Default-First-Site-Name,CN=Sites,
                             CN=Configuration,DC=nwtraders,DC=com
```



```

ServerObjectGuid      : 5ae1fd0e-bc2f-42a7-af62-24377114e03d
Site                  : Default-First-Site-Name
SslPort               : 636

```

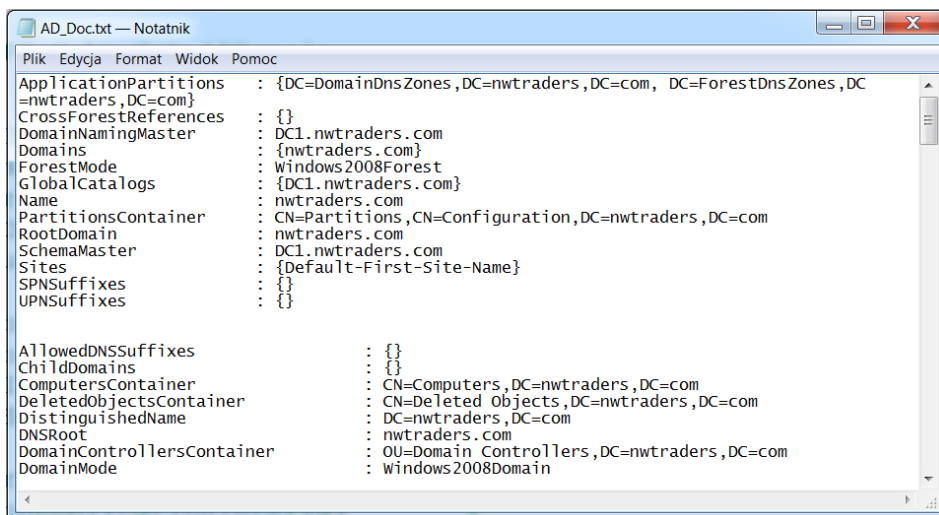
Aby sporządzić raport, wystarczy wysłać wyniki do pliku tekstowego. Poniższe polecenia zbierają informacje opisane w całym podrozdziale i zapisują je w pliku o nazwie *AD_Doc.txt*. Ponadto są one dowodem na to, że można skierować informacje do pliku przechowywanego w udziale sieciowym.

```

Get-ADForest >> \\dc1\shared\AD_Doc.txt
Get-ADDomain >> \\dc1\shared\AD_Doc.txt
Get-ADDefaultDomainPasswordPolicy >> \\dc1\shared\AD_Doc.txt
Get-ADDomainController -Identity dc1 >> \\dc1\shared\AD_Doc.txt

```

Zawartość tego pliku w Notatniku przedstawiono na rysunku 3.4.



RYСУNEK 3.4. Dokumentacja Active Directory w Notatniku

Zmienianie nazw lokalizacji usługi Active Directory

Nazwę lokalizacji zmienia się bardzo łatwo. Wystarczy kliknąć prawym przyciskiem myszy wybraną lokalizację i w menu podręcznym, które zostanie wyświetlone, wybrać opcję zmiany nazwy. Standardowo pierwsza lokalizacja ma niezbyt wyszukaną nazwę *Nazwa-pierwszej-lokacji*. Każdy, kto pracuje z lokalizacjami Active Directory, musi wiedzieć, że są one trochę dziwne. Po pierwsze: znajdują się w kontekście nazw konfiguracji. Nawiązanie połączenia z tym kontekstem z poziomu modułu Active Directory konsoli Windows PowerShell jest łatwe i polega na wykonaniu polecenia `Get-ADRootDSE` oraz wybraniu własności `ConfigurationNamingContext`. Najpierw należy nawiązać połączenie z kontrolerem domeny i zaimportować moduł Active Directory (jeśli na komputerze klienckim nie ma zainstalowanych narzędzi RSAT), jak pokazano poniżej:

```

Enter-PSSession -ComputerName dc3 -Credential iammred\administrator
Import-Module activedirectory

```

Poniższy kod znajduje wszystkie lokalizacje. Użyto w nim polecenia `Get-ADObject` do przeszukania kontekstu nazw konfiguracji dla obiektów, których klasa obiektowa to `site`.

```
Get-ADObject -SearchBase (Get-ADRootDSE).ConfigurationNamingContext -filter "objectclass -eq 'site'"
```

Po znalezieniu szukanej lokalizacji najpierw należy zmienić atrybut `DisplayName`. W tym celu należy przekazać obiekt lokalizacji do polecenia `Set-ADObject`, za pomocą którego można ustawiać wiele różnych atrybutów obiektów. Poniżej pokazano przykład jego użycia (jest to pojedyncze polecenie podzielone na dwie części znakiem potoku).

```
Get-ADObject -SearchBase (Get-ADRootDSE).ConfigurationNamingContext -filter "objectclass -eq 'site'" | Set-ADObject -DisplayName CharlotteSite
```

Po ustawieniu atrybutu `DisplayName` można zmienić nazwę samego obiektu. Służy do tego polecenie `Rename-ADObject`. Ponownie dla uproszczenia obiekt lokalizacji przekazujemy potokowo oraz przypisujemy nową nazwę lokalizacji. Opisywane polecenie znajduje się poniżej (to również jest jednowierszowe polecenie podzielone znakiem potoku).

```
Get-ADObject -SearchBase (Get-ADRootDSE).ConfigurationNamingContext -filter "objectclass -eq 'site'" | Rename-ADObject -NewName CharlotteSite
```

Zarządzanie użytkownikami

Do tworzenia nowej jednostki organizacyjnej służy polecenie `New-ADOrganizationalUnit`:

```
New-ADOrganizationalUnit -Name TestOU -Path "dc=nwtraders,dc=com"
```

Aby utworzyć podrzędną jednostkę organizacyjną, również należy użyć polecenia `New-ADOrganizationalUnit`, tylko w ścieżce należy podać lokalizację nadrzędną, jak pokazano poniżej:

```
New-ADOrganizationalUnit -Name TestOU1 -Path "ou=TestOU,dc=nwtraders,dc=com"
```

Jeśli trzeba utworzyć kilka podrzędnych jednostek organizacyjnych w jednej lokalizacji, można przywrócić poprzednie polecenie za pomocą klawisza strzałki w górę i zmienić nazwę jednostki podrzędnej. Aby przejść na początek wiersza, można nacisnąć klawisz *Home*. Aby przejść na koniec wiersza, należy nacisnąć klawisz *End*. Natomiast za pomocą klawiszy strzałek w lewo i w prawo można poruszać się w obrębie wiersza, aby go zmodyfikować w odpowiednim miejscu. Poniższe polecenie tworzy drugą podrzędną jednostkę organizacyjną:

```
New-ADOrganizationalUnit -Name TestOU2 -Path "ou=TestOU,dc=nwtraders,dc=com"
```

Aby utworzyć konto komputera w jednej z nowo utworzonych jednostek organizacyjnych, należy wpisać pełną ścieżkę do wybranej jednostki organizacyjnej. Do tworzenia nowych kont komputera w AD DS służy polecenie `New-ADComputer`. W poniższym przykładzie jednostka organizacyjna `TestOU1` jest pod jednostką `TestOU` i dlatego w parametrze ścieżki zostały wymienione obie. Pamiętaj, że ścieżka przekazywana w parametrze `path` musi być ujęta w cudzysłów, jak pokazano poniżej:

```
New-ADComputer -Name Test -Path "ou=TestOU1,ou=TestOU,dc=nwtraders,dc=com"
```

Do tworzenia kont użytkowników służy polecenie `New-ADUser`.

```
New-ADUser -Name TestChild -Path "ou=TestOU1,ou=TestOU,dc=nwtraders,dc=com"
```

Jako że trzeba wpisywać sporo kodu, który w znacznej części się powtarza, lepiej jest napisać skrypt tworzący jednostki organizacyjne jednocześnie z tworzeniem kont komputerów i użytkowników. Poniżej znajduje się kod źródłowy przykładowego skryptu tworzącego jednostki organizacyjne oraz konta użytkowników i komputerów (plik *UseADCmdletsToCreateOuComputerAndUser.ps1*).

UseADCmdletsToCreateOuComputerAndUser.ps1

```
Import-Module -Name ActiveDirectory
$Name = "ScriptTest"
$DomainName = "dc=nwtraders,dc=com"
$OUPath = "ou={0},{1}" -f $Name, $DomainName

New-ADOrganizationalUnit -Name $Name -Path $DomainName -ProtectedFromAccidentalDeletion $false

For($you = 0; $you -le 5; $you++)
{
    New-ADOrganizationalUnit -Name $Name$you -Path $OUPath -ProtectedFromAccidentalDeletion $false
}

For($you = 0 ; $you -le 5; $you++)
{
    New-ADComputer -Name "TestComputer$you" -Path $OUPath
    New-ADUser -Name "TestUser$you" -Path $OUPath
}
```

Na początku skrypt *UseADCmdletsToCreateOuComputerAndUser.ps1* importuje moduł Active Directory, a następnie tworzy pierwszą jednostkę organizacyjną. Przy jego testowaniu należy wyłączyć ochronę przed usuwaniem za pomocą parametru `ProtectedFromAccidentalDeletion`. To umożliwi łatwe usunięcie jednostki organizacyjnej i uniknięcie konieczności przechodzenia do widoku zaawansowanego w narzędziu *Użytkownicy i komputery usługi Active Directory* w celu zmiany statusu ochrony każdej jednostki organizacyjnej.

Po utworzeniu jednostki organizacyjnej `ScriptTest` w nowej lokalizacji można utworzyć pozostałe jednostki oraz konta użytkowników i komputerów. Jest oczywiste, że nie można utworzyć podrzędnej jednostki organizacyjnej w nieistniejącej jednostce nadrzędnej, ale łatwo można popełnić taki logiczny błąd.

Do tworzenia globalnych grup zabezpieczeń służy polecenie `New-ADGroup` modułu AD DS konsoli Windows PowerShell. Polecenie to wymaga podania trzech parametrów: **nazwy** (Name) grupy, **ścieżki** (Path) do lokalizacji, w której grupa ta będzie przechowywana, oraz **zakresu grupy** (groupScope), który może być globalny (global), uniwersalny (universal) lub lokalny domenowy (DomainLocal). Zanim wykonasz polecenie pokazane poniżej, nie zapomnij zaimportować do sesji Windows PowerShell modułu Active Directory.

```
New-ADGroup -Name TestGroup -Path "ou=TestOU,dc=nwtraders,dc=com" -groupScope global
```

Aby utworzyć nową grupę uniwersalną, należy tylko odpowiednio zmienić wartość parametru `groupScope`, jak pokazano poniżej:

```
New-ADGroup -Name TestGroup1 -Path "ou=TestOU,dc=nwtraders,dc=com" -groupScope universal
```

Aby dodać użytkownika do grupy, należy podać wartości dla parametrów `identity` i `members`. Wartością parametru `identity` jest nazwa grupy. Nie trzeba stosować składni LDAP `cn=nazwagrupy`, wystarczy tylko podać nazwę. Sprawdź atrybuty LDAP wymagane dla grupy w Edytorze ADSI.

Nazwa parametru `members` zamiast `member` jest dość nietypowa, ponieważ większość nazw parametrów poleceń cmdlet konsoli Windows PowerShell jest w liczbie pojedynczej. Zasada ta jest nawet przestrzegana w przypadku parametrów przyjmujących wiele wartości (np. parametru `computername`). Poniżej pokazano przykładowe polecenie dodające nową grupę o nazwie `TestGroup1` do grupy `UserGroupTest`:

```
Add-ADGroupMember -Identity TestGroup1 -Members UserGroupTest
```

Aby usunąć użytkownika z grupy, należy wykonać polecenie `Remove-ADGroupMember` z nazwą tej grupy. Konieczne jest podanie wartości parametrów `identity` i `members` oraz potwierdzenie chęci wykonania tego polecenia, jak widać poniżej:

```
PS C:\> Remove-ADGroupMember -Identity TestGroup1 -Members UserGroupTest
```

Confirm

Are you sure you want to perform this action?

Performing operation "Set" on Target "CN=TestGroup1,OU=TestOU,DC=NWTraders,DC=Com".

[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help (default is "Y"):

y

```
PS C:\>
```

Jeśli jesteśmy pewni, że chcemy usunąć użytkownika z grupy, możemy wyłączyć potwierdzenie, przypisując parametrowi `confirm` wartość `$false`. Problem w tym, że między nazwą parametru a wartością `$false` należy wstawić dwukropek.

UWAGA

Konieczność wpisania dwukropka po nazwie parametru `confirm` nie jest udokumentowana, ale technika ta działa w kilku różnych poleceniach.

Poniżej przedstawiono przykład użycia opisywanego polecenia:

```
Remove-ADGroupMember -Identity TestGroup1 -Members UserGroupTest -Confirm:$false
```

Możliwość wyłączenia mechanizmu potwierdzania jest potrzebna do tego, aby polecenia `Remove-ADGroupMember` można było używać w skryptach. Skrypt `RemoveUserFromGroup.ps1` najpierw ładuje moduł Active Directory. Potem za pomocą polecenia `Remove-ADGroupMember` usuwa użytkownika z grupy. W celu wyłączenia mechanizmu potwierdzania dodano parametr `-confirm:$false`. Poniżej znajduje się zawartość opisywanego skryptu.

removeUserFromGroup.ps1

```
import-module activedirectory
Remove-ADGroupMember -Identity TestGroup1 -Members UserGroupTest -Confirm:$false
```

Tworzenie użytkownika

Teraz utworzymy w usłudze Active Directory nowego użytkownika o nazwie **ed**. Polecenie służące do tworzenia nowego użytkownika to **New-Aduser** i po nim należy wpisać żadaną nazwę użytkownika. Poniższe polecenie tworzy wyłączone konto użytkownika w kontenerze **users** w domyślnej domenie:

```
new-aduser -name ed
```

Po wykonaniu tego polecenia konsola nie wyświetli żadnych informacji zwrotnych. Aby sprawdzić, czy użytkownik rzeczywiście został utworzony, należy dodatkowo wykonać polecenie **Get-Aduser**, jak pokazano poniżej:

```
Get-aduser ed
```

Po upewnieniu się, że został utworzony nowy użytkownik, można utworzyć jednostkę organizacyjną do przechowywania konta użytkownika. Poniższe polecenie tworzy nową jednostkę organizacyjną z korzenia domeny:

```
new-ADOrganizationalUnit scripting
```

Podobnie jak w przypadku poprzedniego polecenia **New-Aduser**, również w tym przypadku w konsoli nie zostaną wyświetlone żadne informacje zwrotne. Jeśli użyjesz polecenia **Get-ADOrganizationalUnit**, musisz zastosować odmienną strategię, ponieważ wykonanie go w najprostszy sposób spowoduje błąd. Do znalezienia jednostki organizacyjnej należy użyć parametru **LDAPFilter**, jak pokazano poniżej:

```
Get-ADOrganizationalUnit -LDAPFilter "(name=scripting)"
```

Mając nowego użytkownika i nową jednostkę organizacyjną, należy przenieść użytkownika z kontenera **users** do nowo utworzonej jednostki o nazwie **scripting**. Do tego potrzebne będzie polecenie **Move-ADObject**. Najpierw należy pobrać atrybut **distinguishedname** jednostki **scripting** i zapisać go w zmiennej **\$outpath**. Następnie za pomocą polecenia **Move-ADObject** można przenieść użytkownika **ed** do nowej jednostki organizacyjnej. Sztuka w tym przypadku polega na tym, że podczas gdy tam, gdzie polecenie **Get-ADUser** znajdzie użytkownika o nazwie **ed**, polecenie **Move-ADObject** musi mieć nazwę **distinguishedname** obiektu użytkownika **ed**, aby go przenieść.

Kolejną czynnością jest włączenie konta użytkownika, co wymaga przypisania mu hasła w postaci bezpiecznego ciągu. Do tego celu można użyć polecenia **ConvertTo-SecureString**. Domyślnie przy konwersji tekstu na bezpieczny ciąg wyświetlane są ostrzeżenia, ale można je wyłączyć za pomocą parametru **force**. Poniżej znajduje się polecenie tworzące bezpieczny ciąg do użycia jako hasło:

```
$pwd = ConvertTo-SecureString -String "P@ssword1" -AsPlainText -Force
```

Po utworzeniu bezpiecznego ciągu można go ustawić jako hasło do konta użytkownika za pomocą polecenia **Set-ADAccountPassword**. Jako że jest to nowe hasło, należy użyć parametru **newpassword**. Ponadto nie mamy poprzedniego hasła, więc musimy użyć parametru **reset**. Całe polecenie pokazano poniżej:

```
Set-ADAccountPassword -Identity ed -NewPassword $pwd -Reset
```

Po ustawieniu hasła do konta można je włączyć za pomocą polecenia `Enable-ADAccount`, któremu należy przekazać nazwę użytkownika do aktywowania:

```
Enable-ADAccount -Identity ed
```

Podobnie jak wcześniej opisane polecenia niniejsze polecenia nie wyświetlają żadnych informacji zwrotnych. W związku z tym, aby upewnić się, że konto użytkownika `ed` zostało włączone, należy wykonać polecenie `Get-ADUser`. W zwróconych danych należy szukać wartości własności `enabled`. Jako że jest to własność logiczna, powinna mieć wartość reprezentującą prawdę.

Znajdowanie kont użytkowników i ich odblokowywanie

Znalezienie zablokowanych kont użytkowników za pomocą poleceń z modułu Active Directory jest bardzo łatwe. Nawet polecenie `Search-ADAccount` ma przełącznik `LockedOut`. Poniżej pokazano przykład ich użycia:

```
Search-ADAccount -LockedOut
```

UWAGA

Wielu administratorów sieci, którzy przez większość czasu pracują z usługą Active Directory, importuje ten moduł przez profil Windows PowerShell. Dzięki temu unikają chwilowego spowolnienia powodowanego koniecznością automatycznego załadowania modułu Active Directory, gdy jest potrzebny.

Poniżej przedstawiono przykład użycia polecenia `Search-ADAccount` i jego wyniku:

```
[w8server6]: PS C:\> Search-ADAccount -LockedOut
```

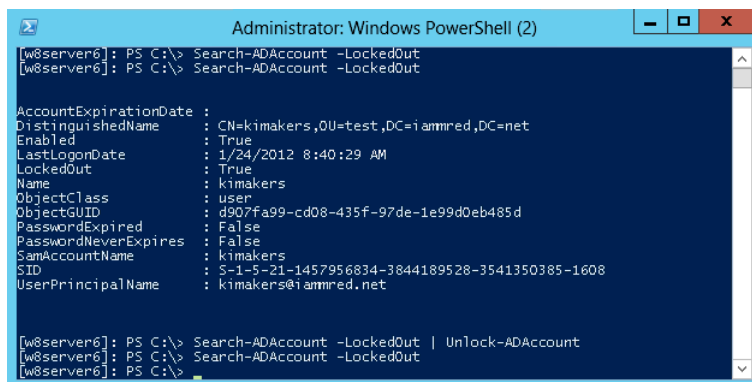
```
AccountExpirationDate :
DistinguishedName      : CN=kimakers,OU=test,DC=iammred,DC=net
Enabled                : True
LastLogonDate          : 1/24/2012 8:40:29 AM
LockedOut              : True
Name                   : kimakers
ObjectClass             : user
ObjectGUID             : d907fa99-cd08-435f-97de-1e99d0eb485d
PasswordExpired        : False
PasswordNeverExpires   : False
SamAccountName         : kimakers
SID                    : S-1-5-21-1457956834-3844189528-3541350385-1608
UserPrincipalName      : kimakers@iammred.net
```

```
[w8server6]: PS C:\>
```

Można także odblokować zablokowane konto użytkownika — oczywiście pod warunkiem, że ma się odpowiednie uprawnienia. Na rysunku 3.5 pokazano próbę odblokowania konta użytkownika przy użyciu konta zwykłego użytkownika, która zakończyła się błędem.

UWAGA

Wielu użytkowników konsoli Windows PowerShell obawia się o kwestie bezpieczeństwa. Ale Windows PowerShell to tylko aplikacja, w której użytkownik nie może zrobić nic, do czego nie ma uprawnień. To bardzo ważna uwaga.



RYСУNEK 3.5. Użycie modułu Active Directory do znalezienia i odblokowania kont użytkowników

Jeśli Twoje konto nie ma uprawnień administracyjnych, musisz uruchomić konsolę Windows PowerShell przy użyciu konta z uprawnieniami do odblokowywania kont użytkowników. W tym celu kliknij jej ikonę prawym przyciskiem myszy, jednocześnie trzymając wciśnięty klawisz *Shift*. Pojawi się menu podręczne, w którym możesz kliknąć opcję *Uruchom jako inny użytkownik*.

Po ponownym uruchomieniu konsoli z odpowiednimi uprawnieniami należy jeszcze raz załadować moduł Active Directory. Później trzeba sprawdzić, czy nadal udaje się zlokalizować zablokowane konta użytkowników. Po sprawdzeniu tego należy przekazać wynik polecenia `Search-ADAccount` do polecenia `Unlock-ADAccount`. Można szybko sprawdzić, czy wszystkie konta zostały odblokowane. Poniżej pokazano wszystkie opisane polecenia:

```
Search-ADAccount -LockedOut
Search-ADAccount -LockedOut | Unlock-ADAccount
Search-ADAccount -LockedOut
```

Wszystkie te polecenia i wynik ich wykonania widać też na rysunku 3.5.

UWAGA

Pamiętaj, że polecenie `Search-ADAccount -LockedOut | Unlock-ADAccount` odblokuje wszystkie konta, które masz prawo odblokować. W większości przypadków przed wykonaniem tej operacji należy dokładnie przejrzeć wszystkie zablokowane konta. Jeśli nie chcesz odblokować ich wszystkich, użyj przełącznika `confirm`, który spowoduje wyświetlenie prośby o potwierdzenie operacji dla każdego odblokowywanego konta.

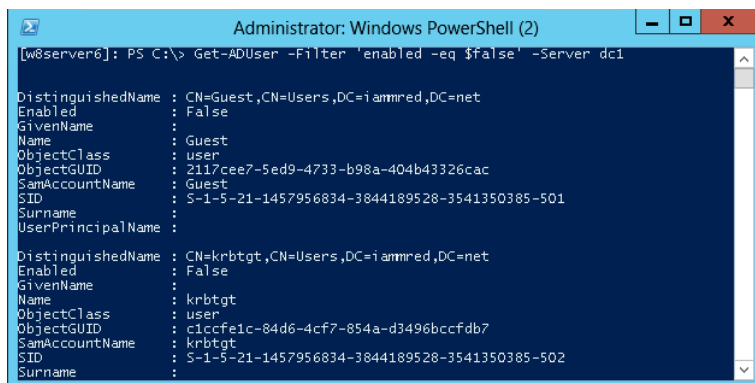
Jeśli nie chcesz odblokować wszystkich użytkowników, użyj parametru `confirm` polecenia `Unlock-ADAccount`. Na przykład najpierw za pomocą polecenia `Search-ADAccount` sprawdzasz, którzy użytkownicy są zablokowani — ale nie chcesz wyświetlać wszystkich informacji, a jedynie nazwy użytkowników. Następnie przekazujesz zablokowanych użytkowników do polecenia `Unlock-ADAccount` z parametrem `confirm`. Zostaniesz poproszony o potwierdzenie chęci odblokowania każdego z trzech użytkowników. Odblokuj pierwszego i trzeciego, a drugiego pozostaw zablokowanego. Na koniec jeszcze raz użyj polecenia `Search-ADAccount`, aby się upewnić, że drugi użytkownik pozostał zablokowany.

Znajdowanie wyłączonych użytkowników

Na szczęście dzięki konsoli Windows PowerShell i poleceniom z modułu Active Directory pobranie listy wyłączonych użytkowników z domeny wymaga wykonania zaledwie jednego wiersza kodu. Jest on pokazany poniżej. (Pamiętaj, że wykonanie tego polecenia spowoduje automatyczne zaimportowanie modułu Active Directory do bieżącego hosta konsoli Windows PowerShell).

```
Get-ADUser -Filter 'enabled -eq $false' -Server dc3
```

Nie dość, że polecenie to zajmuje tylko jedną linię kodu, to na dodatek kod ten jest bardzo czytelny. Pobieramy użytkowników z AD DS i za pomocą filtru szukamy własności `enabled` ustawionej na `false`. Ponadto zaznaczyliśmy, że chcemy odpytać serwer o nazwie `dc3` (taką nazwę ma jeden z kontrolerów domen w mojej sieci). Opisywane polecenie i jego wynik widać na rysunku 3.6.



RYSUNEK 3.6. Znajdowanie wyłączonych kont użytkowników

Aby wybrać konto jednego użytkownika, można użyć parametru `identity`, który przyjmuje kilka wartości: `distinguishedname`, `sid`, `guid` lub `NazwaKonta`. Najłatwiej jest użyć nazwy konta, jak pokazano poniżej:

```
PS C:\Users\ed.IAMMRED> Get-ADUser -Server dc3 -Identity teresa
DistinguishedName : CN=Teresa Wilson,OU=Charlotte,DC=iammred,DC=net
Enabled           : True
GivenName         : Teresa
Name             : Teresa Wilson
ObjectClass       : user
ObjectGUID        : 75f12010-b952-4d3-9b22-3ada7d26eed8
SamAccountName    : Teresa
SID              : S-1-5-21-1457956834-3844189528-3541350385-1104
Surname          : Wilson
UserPrincipalName : Teresa@iammred.net
```

Aby jako parametru `identity` użyć wartości `DistinguishedName`, należy przekazać ją w cudzysłowie — pojedynczym lub podwójnym. Poniżej przedstawiono przykład takiego rozwiązania:


```
PS C:\Users\ed.IAMMRED> Get-ADUser -Server dc3 -Identity 'CN=Teresa
Wilson,OU=Charlotte,DC=iammred,DC=net'
DistinguishedName : CN=Teresa Wilson,OU=Charlotte,DC=iammred,DC=net
Enabled           : True
GivenName        : Teresa
Name             : Teresa Wilson
ObjectClass      : user
ObjectGUID       : 75f12010-b952-4d3-9b22-3ada7d26eed8
SamAccountName   : Teresa
SID              : S-1-5-21-1457956834-3844189528-3541350385-1104
Surname          : Wilson
UserPrincipalName : Teresa@iammred.net
```

Natomiast identyfikatora SID nie trzeba umieszczać w cudzysłowie, jak pokazano poniżej:

```
PS C:\Users\ed.IAMMRED> Get-ADUser -Server dc3 -Identity S-1-5-21-1457956834-3844189528-3541350385-1104
DistinguishedName : CN=Teresa Wilson,OU=Charlotte,DC=iammred,DC=net
Enabled           : True
GivenName        : Teresa
Name             : Teresa Wilson
ObjectClass      : user
ObjectGUID       : 75f12010-b952-4d3-9b22-3ada7d26eed8
SamAccountName   : Teresa
SID              : S-1-5-21-1457956834-3844189528-3541350385-1104
Surname          : Wilson
UserPrincipalName : Teresa@iammred.net
```

Wartością parametru identity może też być identyfikator ObjectGUID, który również nie musi być umieszczony w cudzysłowie. Poniżej pokazano przykład użycia tego identyfikatora:

```
PS C:\Users\ed.IAMMRED> Get-ADUser -Server dc3 -Identity 75f12010-b952-4d3-9b22-3ada7d26eed8
DistinguishedName : CN=Teresa Wilson,OU=Charlotte,DC=iammred,DC=net
Enabled           : True
GivenName        : Teresa
Name             : Teresa Wilson
ObjectClass      : user
ObjectGUID       : 75f12010-b952-4d3-9b22-3ada7d26eed8
SamAccountName   : Teresa
SID              : S-1-5-21-1457956834-3844189528-3541350385-1104
Surname          : Wilson
UserPrincipalName : Teresa@iammred.net
```

Znajdowanie nieużywanych kont użytkowników

Aby wyświetlić listę wszystkich użytkowników usługi Active Directory, należy przekazać symbol wieloznaczny do parametru filter polecenia Get-ADUser, jak pokazano poniżej:

```
Get-ADUser -Filter *
```

Aby zmienić bazę operacji wyszukiwania, należy użyć parametru searchbase. Parametr ten przyjmuje nazwy w stylu LDAP. Poniższe polecenie zmienia bazę wyszukiwania na TestOU:

```
Get-ADUser -Filter * -SearchBase "ou=TestOU,dc=nwtraders,dc=com"
```

Polecenie Get-ADUser wyświetla tylko część własności użytkownika (dokładnie 10). Własności te zostaną wyświetlone po przekazaniu wyników do polecenia Format-List oraz użyciu symbolu wieloznacznego i parametru force, jak pokazano poniżej:

```
PS C:\> Get-ADUser -Identity bob | format-list -Property * -Force
```

```
DistinguishedName : CN=bob,OU=TestOU,DC=NWTraders,DC=Com
Enabled           : True
GivenName        : bob
Name             : bob
ObjectClass      : user
ObjectGUID       : 5cae3acf-f194-4e07-a466-789f9ad5c84a
SamAccountName   : bob
SID              : S-1-5-21-3746122405-834892460-3960030898-3601
Surname          :
UserPrincipalName : bob@NWTraders.Com
PropertyNames    : {DistinguishedName, Enabled, GivenName, Name...}
PropertyCount    : 10
```

```
PS C:\>
```

Każdy, kto dobrze zna usługi domenowe Active Directory (AD DS), wie, że obiekt użytkownika ma o wiele więcej własności niż 10. Jeśli jednak spróbujemy wyświetlić własność niezwróconą przez polecenie `Get-ADUser`, np. `whenCreated`, zamiast jej wartości zostanie zwrócony błąd, jak widać poniżej:

```
PS C:\> Get-ADUser -Identity bob | Format-List -Property name, whenCreated
```

```
name : bob
whencreated :
```

Własność `whenCreated` obiektu użytkownika ma pewną wartość, tylko nie jest ona wyświetlana. Ale wyobraź sobie, że szukasz użytkowników, którzy nigdy się nie zalogowali w systemie. Powiedzmy, że użyłeś w tym celu polecenia pokazanego poniżej i na podstawie otrzymanego wyniku zamierzasz dokonać operacji usuwania. Efekt tego może być katastrofalny.

```
PS C:\> Get-ADUser -Filter * | Format-Table -Property name, LastLogonDate
```

| name | LastLogonDate |
|--|---------------|
| ---- | ----- |
| Administrator | |
| Guest | |
| krbtgt | |
| testuser2 | |
| ed | |
| SystemMailbox{1f05a927-a261-4eb4-8360-8... | |
| SystemMailbox{e0dc1c29-89c3-4034-b678-e... | |
| FederatedEmail.4c1f4d8b-8179-4148-93bf-... | |
| Test | |
| TestChild | |

<results truncated>

Aby pobrać własność nienależącą do 10 domyślnych własności, należy ją specjalnie wybrać za pomocą parametru `property`. Polecenie `Get-ADUser` nie zwraca automatycznie wszystkich własności z wartościami z powodów wydajności, które w dużych sieciach mogą być bardzo znaczące — nie ma sensu zwracać wielkiego zbioru danych, gdy potrzebna jest tylko niewielka jego część. Aby wyświetlić nazwę (`name`) i datę utworzenia (`whenCreated`) użytkownika o nazwie `bob`, należy użyć poniższego polecenia:

```
PS C:\> Get-ADUser -Identity bob -Properties whencreated | Format-List -Property name, whencreated
```

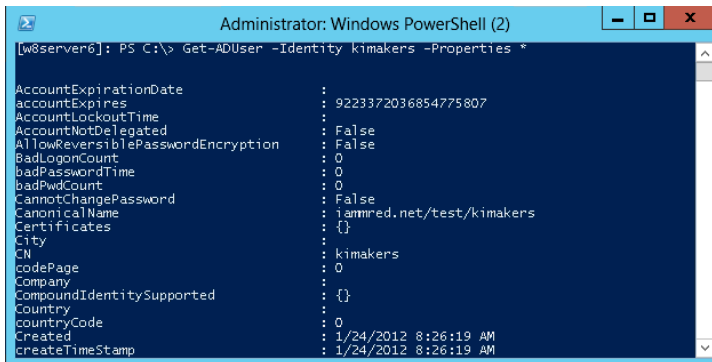
```
name          : bob
whencreated   : 6/11/2010 8:19:52 AM
```

```
PS C:\>
```

Aby pobrać wszystkie własności wybranego obiektu użytkownika, należy jako wartość parametru `properties` podać symbol wieloznaczny `*`, na przykład:

```
Get-ADUser -Identity kimakers -Properties *
```

Na rysunku 3.7 widać wynik pobrania wszystkich własności pewnego użytkownika.

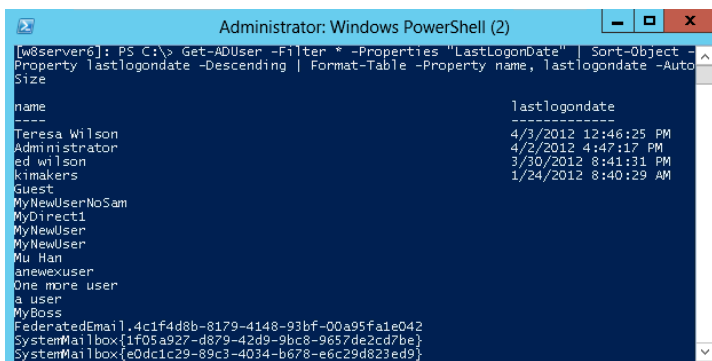


RYSUNEK 3.7. Wyświetlenie wszystkich własności użytkownika za pomocą polecenia `Get-ADUser`

Aby wyświetlić listę wszystkich użytkowników wraz z datami ich ostatniego logowania, można użyć polecenia podobnego do poniższego. Jest to pojedyncze polecenie, ale dość długie, więc może się zawinąć, jeśli szerokość okna będzie niewystarczająca.

```
Get-ADUser -Filter * -Properties "LastLogonDate" |
sort-object -property lastlogondate -descending |
Format-Table -property name, lastlogondate -AutoSize
```

W wyniku zostanie zwrócona zgrabna tabela, jak pokazano na rysunku 3.8.



RYSUNEK 3.8. Wynik sprawdzenia dat ostatniego logowania użytkowników za pomocą polecenia `Get-ADUser`

Zapiski praktyka

Jeff Wouters

MVP Microsoft PowerShell

Jedną z moich ulubionych zasad powtarzanych w społeczności Windows PowerShell jest „pisz narzędzia, nie skrypty”. Od kiedy zacząłem pisać kod Windows PowerShell, mam fioła na punkcie pisania jednowierszowych poleceń.

Łatwość potokowego łączenia poleceń jest czymś, o czym nigdy nie słyszano wśród użytkowników języka VBS.

Ale pewnego razu pewien klient poprosił mnie, abym pozostawił mu część napisanego przeze mnie kodu. Spełniłem tę prośbę. Po tygodniu zadzwonił do mnie spanikowany klient, twierdząc, że mój skrypt usunął połowę danych z Active Directory.

Poprosiłem go, aby przysłał mi kod tego skryptu. Już po kilku sekundach wiedziałem, że to nie był mój kod. Skrypt ten zawierał o wiele więcej kodu, niż ja napisałem. Połączyłem się więc ze swoim domowym środowiskiem i poszukałem w nim kopii zapasowej wszystkich skryptów oraz dokumentacji wykonanych dla tego klienta. Okazało się, że rzeczywiście mój skrypt był o wiele krótszy. Ktoś musiał go zmienić.

Na szczęście klient, o którym mowa, miał włączony koszt w Active Directory, dzięki czemu dało się bez trudu przywrócić usunięte obiekty. Ale dla mnie zdarzenie to było ostrzeżeniem, aby podpisywać swoje skrypty, a przynajmniej zapewnić sobie możliwość weryfikacji ich nienaruszalności.

Potem dowiedziałem się, że mój skrypt zmienił pracownik pomocy technicznej firmy. W tym miejscu właśnie zastosowanie ma zasada „pisz narzędzia, nie skrypty”.

Przepisałem mój skrypt, dodałem graficzny interfejs użytkownika i podpisałem go. Dzięki temu pracownicy pomocy technicznej otrzymali wygodny w obsłudze graficzny interfejs, a sam skrypt jest zabezpieczony przed dostępem przez niepowołane osoby. Jako że skryptów było więcej, utworzyłem moduł o nazwie <NazwaFirmy>Administration. Na koniec pokazałem klientowi, jak przechowywać moduły w centralnym magazynie.

Dla mnie była to dobra nauczka i dziś już stosuję podejście sześciostopniowe:

1. Zapisuj wszystkie możliwe informacje, co robi skrypt i kto go wykonuje.
2. Zaimplementuj obsługę podstawowych parametrów, takich jak `-WhatIf` i `-Confirm`.
3. Utwórz interfejs dostosowany do potrzeb użytkownika — wiersz poleceń dla użytkowników umiejących posługiwać się konsolą Windows PowerShell i graficzny interfejs dla pozostałych.
4. Podpisuj swoje skrypty!
5. Grupuj skrypty w modułach.
6. Używaj centralnego magazynu skryptów, najlepiej tylko do odczytu dla wszystkich tych, którzy nie odpowiadają za moduły.

Przestrzeganie tych kilku reguł znacznie ułatwi Ci życie, gdy inni zaczną szperać w Twoich skryptach.

Dodatkowe źródła informacji

- Centrum skryptów portalu TechNet pod adresem <http://www.microsoft.com/technet/scriptcenter> zawiera wiele przykładowych skryptów.
- Wszystkie skrypty opisane w tym rozdziale są dostępne do pobrania w repozytorium centrum skryptów portalu TechNet pod adresem <http://gallery.technet.microsoft.com/scriptcenter/PowerShell-40-Best-d9e16039>.

Rozdział 4

Znajdowanie możliwości zastosowania skryptów

- Automatyzowanie rutynowych zadań
- Interfejs automatyzacji
- Wymagania strukturalne
- Dodatkowe źródła informacji

Automatyzowanie rutynowych zadań

Jednym z najważniejszych zadań podczas tworzenia skryptów jest śledzenie i koordynowanie prac różnych członków zespołu programistów. Mimo to w większości firm w ogóle się tego nie robi, przez co traci się mnóstwo czasu na tworzenie wielu skryptów robiących to samo.

Jest to jedna z tych dziedzin, w których mądre zastosowanie narzędzi do współpracy może odgrywać bardzo ważną rolę. Jednym z takich narzędzi, które można bardzo łatwo wdrożyć, jest Microsoft SharePoint Portal. Na forum dyskusyjnym można rejestrować prośby o skrypty, a bibliotekę wykorzystać jako centralny punkt dystrybucji gotowych skryptów.

Szukając okazji do napisania skryptów, należy wiedzieć, które zadania dojrzały do automatyzacji, a które jeszcze nie. Ogólnie rzecz biorąc, najważniejszym kryterium przy podejmowaniu decyzji, czy zautomatyzować coś za pomocą skryptu, czy nie, jest powtarzalność. Rutynowe zadania prawie zawsze należy brać pod uwagę jako potencjalnie przydatne do automatyzacji. Ale to, że jakieś zadanie jest powtarzalne, wcale nie znaczy, że z jego automatyzacji za pomocą skryptu będą duże korzyści. Wiele takich zadań nawet nie można zautomatyzować z różnych powodów.

Zapiski praktyka

Jason Hofferle, informatyk

Informatykom trudno nadążyć za rozwojem ich własnej dziedziny, a tym bardziej śledzić najnowsze osiągnięcia w obszarze najlepszych praktyk używania konsoli Windows PowerShell. Jako że jestem wielkim entuzjastą tego narzędzia, pełnię w firmie nieformalną rolę „faceta

od skryptów”. Pomagam innym pracownikom w nauce obsługi konsoli Windows PowerShell i stosowaniu najlepszych rozwiązań zaczerpniętych ze źródeł udostępnianych przez społeczność.

Informacji na temat konsoli Windows PowerShell jest bardzo dużo i codziennie powstają nowe zasoby. Jednym z moich zadań jest filtrowanie tej treści i wybieranie tego, co jest najbardziej przydatne w naszej organizacji. Każdy może zapisać się do listy mailingowej, za pośrednictwem której rozsyłam moim zdaniem przydatne informacje. Czasami ktoś ma konkretny problem, który trzeba rozwiązać, administratorzy próbują zautomatyzować jakieś zadanie albo ktoś opublikuje darmowy kurs obsługi konsoli Windows PowerShell w internecie. Lista mailingowa jest doskonałym narzędziem do rozprowadzania artykułów i przykładów, które mogą być przydatne współpracownikom.

Wewnętrzne strony wiki i SharePoint są świetnymi miejscami do publikowania informacji o skryptach przydatnych tylko w firmie. Istnieją miliony ogólnych przykładów automatyzacji wybranych zadań, ale bywa, że początkujący mają problemy z zastosowaniem ich w konkretnym przypadku. Rolą faceta od skryptów jest pomagać innym w dostrzeganiu przypadków, w których skrypty mogą pomóc zaoszczędzić dużo czasu i pracy. Po pewnym czasie inni zaczynają dostrzegać takie możliwości i wysyłać odpowiedzi **Tobie!**

Jedną z największych zalet konsoli Windows PowerShell jest spójność nazw tworzonych według wzoru czasownik-rzeczownik, nazw parametrów, przekazywania obiektów przez potok oraz metod uzyskiwania pomocy. Nie jestem administratorem serwera Exchange, SQL Server czy VMWare, ale potrafię pomóc naszym ekspertom, ponieważ najlepsze praktyki dotyczące obsługi konsoli Windows PowerShell mają zastosowanie wszędzie. Nie muszę znać się na wdrażaniu serwera Exchange, aby wiedzieć, jak napisać dobrą funkcję albo skrypt przyjmujący potokowe dane wejściowe. Facet od skryptów jest zawsze gotowy pomagać, ponieważ w ten sposób poprawia organizację pracy całej firmy.

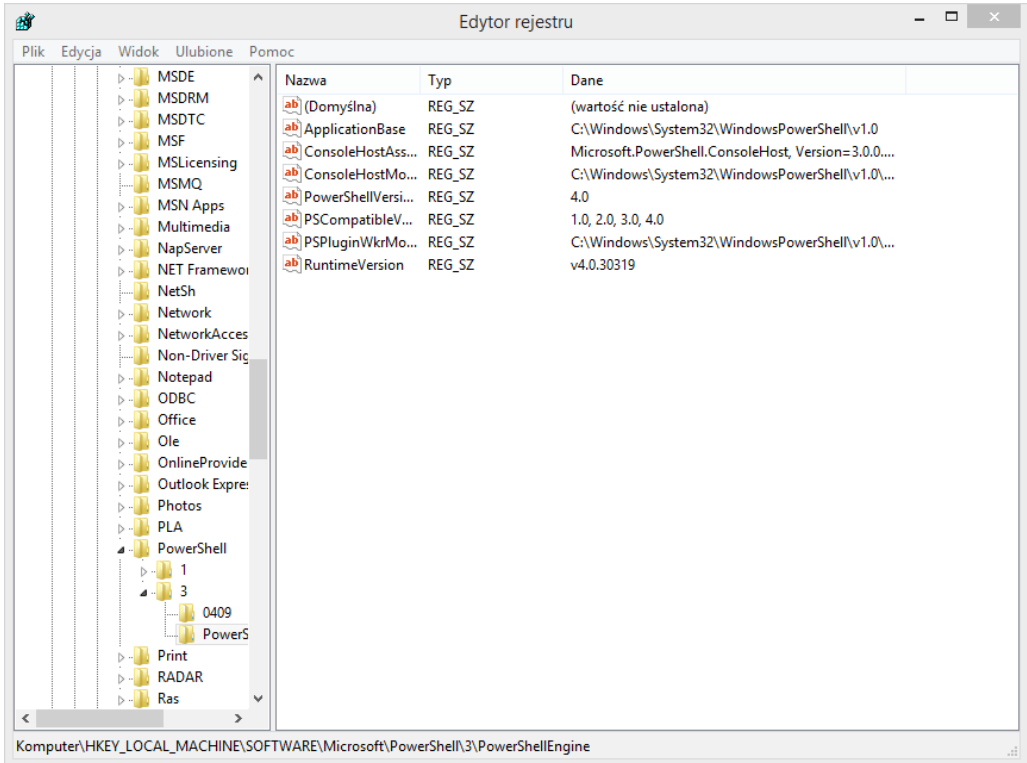
Obsługa konsoli Windows PowerShell jest podstawową umiejętnością informatyka, ale nie każdy musi być ekspertem. Często wystarczy jedna osoba znająca odpowiedź na jakieś pytanie albo potrafiąca podać przykład, aby zaoszczędzić innym wielu godzin frustracji. Organizacje wiele zyskują dzięki pracy „facetów od skryptów”, a ktoś, kto czyta książkę o najlepszych praktykach, może być dobrym kandydatem na takie stanowisko.

Interfejs automatyzacji

Do najbardziej potrzebnych rzeczy można zaliczyć jakiś rodzaj interfejsu automatyzacji. Automatyzację można zaimplementować na wiele sposobów, np. poprzez model COM (ang. *Component Object Model*), klasyczny interfejs programistyczny (ang. *application programming interface* — API), przy użyciu platformy Microsoft .NET Framework, WMI (ang. *Windows Management Instrumentation*), ActiveX Data Object (ADO) w różnych wersjach oraz przy użyciu interfejsu usług Active Directory (ang. *Active Directory Services Interface* — ADSI), nie wspominając już poleceń cmdlet Windows PowerShell czy takich narzędzi wiersza poleceń jak NetSH i NetDom.

Przy takim bogactwie możliwości automatyzacji zadań znalezienie odpowiedniej metody może być zarówno czasochłonne, jak i przytłaczające. Weźmy na przykład prostą czynność, jaką jest odczyt danych z rejestru.

Jeśli chcemy dowiedzieć się, która wersja konsoli Windows PowerShell jest zainstalowana w komputerze, możemy odczytać wartość *PowerShellVersion* z rejestru. Na rysunku 4.1 widać ten klucz rejestru.



RYСУNEK 4.1. Identyfikacja wersji konsoli Windows PowerShell w rejestrze

Jednym ze sposobów na odczyt danych z rejestru jest użycie dostawcy rejestru z poziomu konsoli Windows PowerShell i odczytanie wartości rejestru w taki sam sposób, jak odczytuje się własności z plików oraz folderów. W tym celu należy użyć dysku PowerShell HKLM i podać ścieżkę do klucza rejestru, która w tym przypadku jest następująca: `\SOFTWARE\Microsoft\PowerShell\1\PowerShellEngine`. Później można wybrać interesującą nas własność, czyli `RunTimeVersion`, jak pokazano w poniższym skrypcie *Get-PsVersionRegistry.ps1*.

Get-PsVersionRegistry.ps1

```
$path = "HKLM:\SOFTWARE\Microsoft\PowerShell\3\PowerShellEngine"
$psv = get-itemproperty -path $path
$psv.PowerShellVersion
```

Wiedza tajemna

Wielkie korzyści z automatyzacji przy użyciu konsoli Windows PowerShell 4.0

Keith Mayer, starszy propagator techniczny

Microsoft Corporation

Jestem informatykiem od ponad 20 lat i podczas swojej kariery miałem okazję pracować z wieloma przedsiębiorstwami z branży informatycznej. Wszystkie te firmy kładą duży nacisk na standaryzację różnych procesów, aby zapewnić jak największą niezawodność rozwiązań. Jak to robią? Oczywiście przy użyciu dokumentów SOP (ang. *Standard Operating Procedure*).

Dokumenty SOP mogą zawierać bardzo szczegółowe informacje, a niektóre z nich mają nawet po kilkaset stron. Podczas gdy posiadanie tak szczegółowej dokumentacji jest niewątpliwie czymś pożądanym, dokumenty SOP nie eliminują jednego ze starych problemów branży informatycznej: błędu ludzkiego. Wdrażanie rozwiązań na podstawie dokumentów SOP często wymaga niezwyklej uwagi przy wykonywaniu setek, a czasami nawet tysięcy czynności. Jeśli w tym czasie wystąpią jakieś inne problemy, łatwo się pogubić i przez pomyłkę pominąć jakiś krok.

W takich przypadkach natychmiastowe korzyści można odnieść z wykorzystania konsoli Windows PowerShell 4.0 i usługi konfiguracji żądanego stanu! Gdy zaczniesz uczyć się obsługi konsoli Windows PowerShell 4.0, przyjrzyj się bliżej swoim dokumentom SOP. W szczególności poszukaj w nich żmudnych wielokrotnie powtarzanych procesów. Jeżeli zastosujesz nowo nabyte umiejętności do zautomatyzowania wszystkich lub wybranych procesów, które są często wykonywane przez Twój zespół, uzyskasz trzy bardzo duże korzyści: oszczędzisz czas, który będzie można poświęcić innym, ważniejszym pod względem strategicznym zadaniom, zwiększysz niezawodność i spójność rozwiązań poprzez eliminację błędu ludzkiego oraz pokażesz członkom swojego zespołu, jaką wartość ma konsola Windows PowerShell, dzięki czemu oni również zechcą nauczyć się jej obsługi, aby móc zautomatyzować inne procesy.

Automatyzacja często używanych dokumentów SOP może być świetną okazją do nauki obsługi nowych funkcji konsoli Windows PowerShell przez pracowników firmy. Znam wiele organizacji, w których poświęcono mnóstwo czasu na obmyślenie najlepszych sposobów wykorzystania konsoli Windows PowerShell. Nie przegap okazji, aby zacząć korzystać z tego doskonałego narzędzia także w swoim zespole. W ten sposób skrócisz czas wykonywania wielu procesów.

Odczytywanie rejestru za pomocą metody RegRead

Osoby znające język VBScript mogą zechcieć utworzyć obiekt `WshShell` i użyć metody `RegRead`. W tym celu można użyć skrótu `HKLM` oznaczającego klucz rejestru `HKEY_LOCAL_MACHINE`.

UWAGA

Obiekt `WshShell` rozróżnia wielkość liter w skrócie `HKLM`.

Ścieżkę do danych konfiguracyjnych Windows PowerShell zapisujemy w zmiennej `$path`. Następnie za pomocą polecenia `New-Object` możemy utworzyć egzemplarz obiektu `WshShell`. Obiekt ten ma identyfikator programu `Wscript.Shell`. Zwrócony obiekt można zapisać w zmiennej `$wshShell`. Mając ten obiekt, można za pomocą jego metody `RegRead` odczytać wartość klucza rejestru, którą można określić przez podanie ścieżki i nazwy wartości w łańcuchu `"$path\RuntimeVersion"`. Opisywany skrypt nazwałem *Get-PsVersionRegRead.ps1*, a jego kod można obejrzeć poniżej.

Get-PSVersionregread.ps1

```
$path = "HKLM\SOFTWARE\Microsoft\PowerShell\3\PowerShellEngine"
$WshShell = New-Object -ComObject Wscript.Shell
$WshShell.RegRead("$path\PowerShellVersion")
```

Odczytywanie rejestru za pomocą WMI

Aby odczytać dane z rejestru za pomocą WMI, należy użyć klasy WMI `stdRegProv` należącej do przestrzeni nazw `root\default`. W systemie Windows Vista dodano też klasę WMI `stdRegProv` do przestrzeni nazw `root\cimv2`, ale nie ma to znaczenia, ponieważ jest to ta sama klasa. Ponieważ nie ma różnicy, której klasy użyjemy, zalecam używanie egzemplarza z przestrzeni nazw `root\default` (jak w skrypcie *Get-PsVersionWmi.ps1*), aby zapewnić zgodność ze starszymi wersjami systemu Windows.

W WMI gałęzie drzewa rejestru mają reprezentację liczbowe, które wymieniono w tabeli 4.1.

TABELA 4.1. Wartości WMI gałęzi drzewa rejestru

| Nazwa | Wartość |
|----------------------------------|------------|
| <code>HKEY_CLASSES_ROOT</code> | 2147483648 |
| <code>HKEY_CURRENT_USER</code> | 2147483649 |
| <code>HKEY_LOCAL_MACHINE</code> | 2147483650 |
| <code>HKEY_USERS</code> | 2147483651 |
| <code>HKEY_CURRENT_CONFIG</code> | 2147483653 |

Możemy użyć wartości 2147483650, którą przypiszemy do zmiennej `$hklm`. Wartość ta wskazuje zapytanie WMI do drzewa rejestru `HKEY_LOCAL_MACHINE`. Później przypisujemy łańcuch `SOFTWARE\Microsoft\PowerShell\1\PowerShellEngine` do zmiennej `$key`.

UWAGA

Przy odczytywaniu rejestru przy użyciu WMI klucza nie należy poprzedzać lewym ukośnikiem.

Wartość własności rejestru, którą chcemy odczytać, przypisujemy do zmiennej `$value`. Teraz możemy użyć akceleratora wpisywania `[WMICLASS]`, aby uzyskać egzemplarz klasy WMI `stdRegProv`. Możesz wybrać przestrzeń nazw WMI `root\default`, aby określić, której wersji klasy `stdRegProv` chcesz używać. Ponadto przed przestrzenią nazw możesz dodać nazwę komputera, aby odczytać rejestr ze zdalnej maszyny. Po utworzeniu egzemplarza klasy `stdRegProv` możesz użyć otrzymanej klasy `System.Management.Management` do wywołania metody `GetStringValue`. Metoda ta przyjmuje trzy argumenty: zakodowaną wartość klucza rejestru, łańcuch podklucza rejestru oraz nazwę własności. Metoda ta zwraca dwa obiekty: `returnValue` oznaczający powodzenie lub niepowodzenie operacji oraz `svalue`, czyli wartość łańcuchową, która jest przechowywana we własności rejestru. Poniżej przedstawiono kompletny kod skryptu *Get-PsVersionWmi.ps1*:

Get-PsVersionWmi.ps1

```
$hklm = 2147483650
$key = "SOFTWARE\Microsoft\PowerShell\3\PowerShellEngine"
$value = "PowerShellVersion"
$wmi = [WMICLASS]"root\default:stdRegProv"
($wmi.GetStringValue($hklm,$key,$value)).svalue
```

Odczytywanie rejestru za pomocą klas platformy .NET

Do pobierania informacji z rejestru można też używać klas platformy .NET, a konkretnie klasy `Microsoft.Win32.Registry`. Można użyć metody `GetValue` przyjmującej trzy parametry. Pierwszy parametr jest korzeniem rejestru i nazwą klucza. Drugi parametr jest wartością rejestru, którą chcemy odczytać. Trzeci parametr jest domyślną wartością klucza rejestru. W skrypcie *Get-PsVersionNet.ps1* możemy przypisać wartość łańcuchową `HKEY_LOCAL_MACHINE` do zmiennej `$hklm`. Następnie możemy przypisać łańcuch reprezentujący pozostałą część ścieżki rejestru do zmiennej `$key`. Wartość własności klucza rejestru, którą chcemy odczytać, zapiszemy w zmiennej `$value`. Później użyjemy metody `GetValue` z klasy `Microsoft.Win32.Registry` (dwa dwukropki oznaczają, że wywołujemy metodę statyczną), przekazując jej jako parametry zmienne `$hklm`, `$key` oraz `$value`. Poniżej znajduje się kompletny kod skryptu *Get-PsVersionNet.ps1*:

Get-PsVersionNet.ps1

```
$hklm = "HKEY_LOCAL_MACHINE"
$key = "SOFTWARE\Microsoft\PowerShell\3\PowerShellEngine"
$value = "PowerShellVersion"
[Microsoft.Win32.Registry]::GetValue("$hklm\$key",$value,$null)
```

Do pracy z rejestrem można używać dostawcy Windows PowerShell, obiektów COM, klas WMI oraz klasy platformy .NET. Ta różnorodność dostępnych metod jest jedną z wielkich zalet konsoli Windows PowerShell, ale również źródłem zamieszania w głowie osób dopiero uczących się obsługi tego narzędzia.

Jak w takim razie powinno się odczytywać rejestr? Na pewno nie zrobisz błędu, jeśli użyjesz macierzystych dostawców i poleceń Windows PowerShell. Dostawca rejestru w tej konsoli ma

bardzo duże możliwości, a zarazem jest łatwy w obsłudze z poziomu wiersza poleceń. Ponadto dzięki funkcjom pracy zdalnej dodanym w Windows PowerShell 4.0 potrzeba możliwości uzyskania zdalnego dostępu do rejestru nie jest już kluczowym czynnikiem przy podejmowaniu decyzji, której techniki użyć.

WMI umożliwia odczytywanie rejestru na komputerze zdalnym. Postępuje się bardzo podobnie jak przy użyciu języka VBScript, a więc jeśli przenosisz skrypt VBScript do Windows PowerShell, możesz pozostać przy WMI. Obiekt `COM WshShell` również jest dobrym wyborem, gdy trzeba przenieść stary kod do Windows PowerShell. Techniki oparte na języku VBScript są do siebie bardzo podobne, dzięki czemu zamiana jednej na drugą nie jest trudna. Klasy platformy .NET dają natomiast największe możliwości z wszystkich opisanych technik.

Macierzyste techniki Windows PowerShell do wykonywania niektórych zadań

Wybór sposobu pobierania informacji z komputerów podczas pracy z konsolą Windows PowerShell jest jedną z fundamentalnych decyzji, jakie należy podjąć przed rozpoczęciem pisania skryptów. Jak się przed chwilą przekonałeś, informacje z rejestru można odczytać na wiele sposobów. Ale w przypadku konsoli Windows PowerShell kwestia dotyczy nie tyle sposobu odczytu danych z rejestru, co tego, czy w ogóle odczytywać dane z rejestru. Przecież istnieje automatyczna zmienna zawierająca wersję konsoli. Nazywa się ona `$PSVersionTable` i jest dostępna od Windows PowerShell 2.0. Poniżej pokazano przykład jej użycia i wyniku, jaki zwraca:

```
PS C:\> $PSVersionTable
```

| Name | Value |
|---------------------------|----------------------|
| PSVersion | 4.0 |
| WSManStackVersion | 3.0 |
| SerializationVersion | 1.1.0.1 |
| CLRVersion | 4.0.30319.34003 |
| BuildVersion | 6.3.9600.16394 |
| PSCompatibleVersions | {1.0, 2.0, 3.0, 4.0} |
| PSRemotingProtocolVersion | 2.2 |

Po odkryciu macierzystej techniki Windows PowerShell do wykonywania jakiegoś zadania kolejnym problemem jest, jak ją najlepiej wykorzystać. Ponieważ jest mało prawdopodobne, aby administrator sieci musiał sprawdzać wersję konsoli Windows PowerShell tylko na jednym komputerze, kwestia sprowadza się do znalezienia najlepszego sposobu na sprawdzenie tej informacji dla grupy zdalnych komputerów. Są dwa możliwe sposoby: można użyć przepływu pracy w Windows PowerShell lub narzędzi do pracy zdalnej Windows PowerShell. Te dwie techniki nie różnią się tak bardzo między sobą (ponieważ przepływ pracy w Windows PowerShell używa technologii pracy zdalnej), więc wybór należy oprzeć na tym, ile jest zdalnych komputerów.

Sprawdzanie wersji konsoli Windows PowerShell za pomocą przepływu pracy w Windows PowerShell jest bardzo łatwe (więcej informacji na temat przepływów pracy znajduje się w rozdziale 22.). Przepływ pracy jest podobny do funkcji Windows PowerShell i również podobnie się go wywołuje. Poniżej przedstawiono kod źródłowy skryptu *Get-PSVersionWorkflow.ps1*:

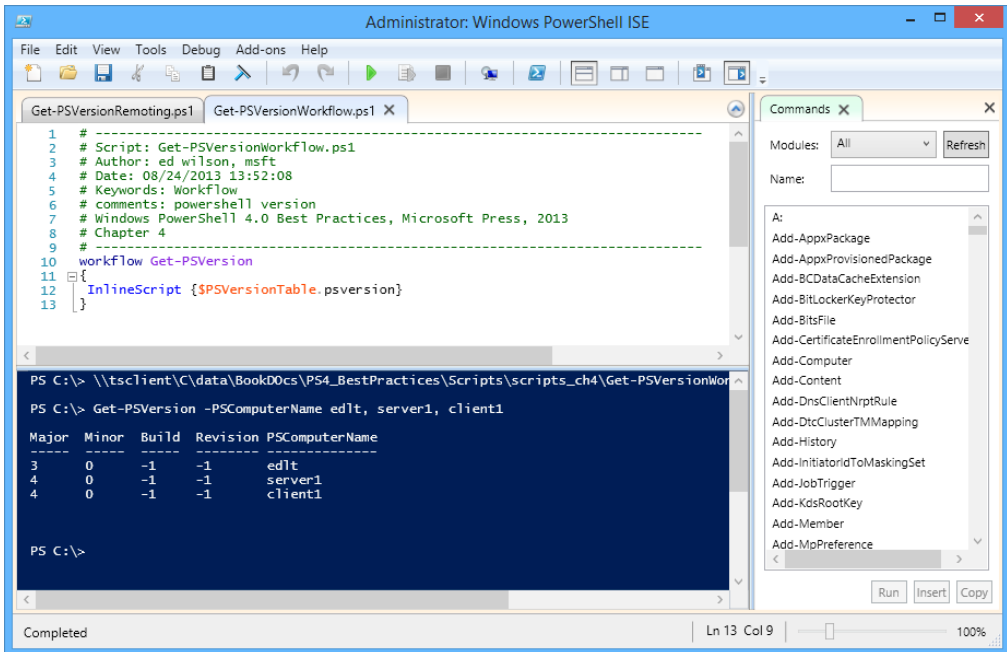
Get-PSVersionWorkflow.ps1

```

workflow Get-PSVersion
{
    InlineScript {$PSVersionTable.psversion}
}

```

Przykład jego użycia widać na rysunku 4.2.

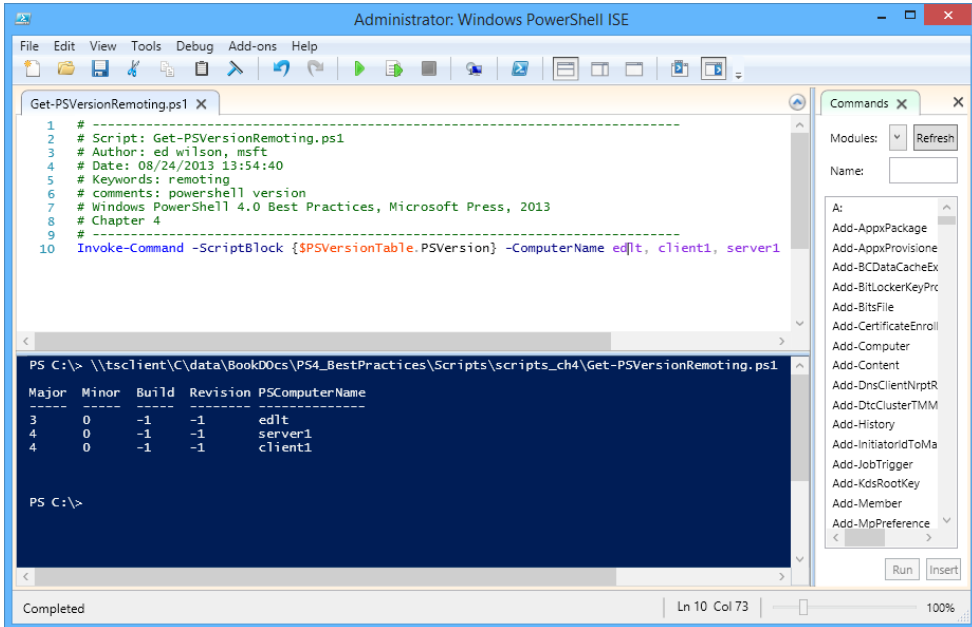
**RYSUNEK 4.2.** Przepływ pracy w Windows PowerShell ułatwia pracę zdalną

Narzędzi do pracy zdalnej konsoli Windows PowerShell można też używać do wykonywania poleceń na zdalnych komputerach (więcej informacji na temat narzędzi do pracy zdalnej znajduje się w rozdziale 21.). Najłatwiej jest użyć w bloku skryptu polecenia `Invoke-Command` i automatycznej zmiennej `$PSVersionTable`. Technikę tę zastosowano w pokazanym poniżej skrypcie `Get-PSVersionRemoting.ps1`:

Get-PSVersionremoting.ps1

```
Invoke-Command -ScriptBlock {$PSVersionTable.PSVersion} -ComputerName edlt, client1, server1
```

Na rysunku 4.3 pokazano przykład wywołania polecenia `Invoke-Command` w konsoli Windows PowerShell ISE.



RYSUNEK 4.3. Narzędzia do pracy zdalnej Windows PowerShell sprawiają, że pobieranie informacji ze zdalnych komputerów jest bardzo łatwe

Wiedza tajemna

Praca z konsolą Windows PowerShell

Jeffrey Snover, Distinguished Engineer

Microsoft Corporation

W konsoli Windows PowerShell 4.0 wykorzystano osiągnięcia architektoniczne trzeciej wersji tej konsoli, dzięki czemu wprowadziliśmy wiele innowacyjnych rozwiązań w krótkim czasie. Dlatego też trudno mi się zdecydować, która z funkcji jest moją ulubioną. Ale gdybym musiał wybrać tylko jedną funkcję, wybrałbym narzędzia do pracy zdalnej, chociaż czuję się, jakbym szukał, bo można wyróżnić przynajmniej sześć różnych zastosowań tych narzędzi:

- Z istniejącymi systemami można łączyć się przy użyciu technologii RPC (ang. *remote procedure call*) i DCOM (ang. *Distributed Component Object Model*). Należy tylko dodać parametr `-computername` do poleceń.
- Przy użyciu WMI można pracować zdalnie na szeroką skalę. Wystarczy użyć paru nowych poleceń wykorzystujących częściowo synchroniczny interfejs API i parametru `-ThrottleLimit`.
- Za pomocą nowych poleceń WS-Management (WSMan) można zdalnie zarządzać sprzętem i komputerami z systemem Unix.

- Na komputerach z systemem Windows można tworzyć zdalne interaktywne sesje Windows PowerShell.
- Można wysyłać żądania wykonania poleceń do dużych grup komputerów i od razu odbierać wyniki albo można uruchomić konsolę Windows PowerShell w tle i odbierać wyniki w wolnym czasie.
- Konsolę Windows PowerShell można hostować jako aplikację IIS (ang. *Internet Information Services*), aby wspomagać zarządzanie zbiorcze, w ramach którego dostawcy usług oferują dostosowane indywidualnie interfejsy skryptów użytkownikom w internecie.

Narzędzia do pracy zdalnej konsoli Windows PowerShell dają administratorom znacznie większą kontrolę nad środowiskiem pracy, niż mieli kiedyś. Już się nie mogę doczekać, aby zobaczyć, jak będą one wykorzystywane. Podejrzewam, że większość użytkowników będzie wykonywać typowe czynności, takie jak tworzenie plików i folderów, tworzenie nowych udziałów oraz praca z rejestrem. Tylko teraz wszystko to będą mogli robić na komputerze zdalnym. Tego typu procedury są udostępniane przez polecenia i dostawców Windows PowerShell. Ponadto myślę, że niektórzy potraktują nasze funkcje jako ogólną platformę do przetwarzania rozproszonego i wykorzystają je do różnych niesamowitych rzeczy.

Gdy muszę zrobić coś na zdalnym komputerze, lubię utworzyć sesję PSSession i używać polecenia `Invoke-Command`, ponieważ jest to bardzo szybka i elastyczna metoda. Polecenia tego używam zarówno wtedy, gdy chcę wykonać proste polecenie, jak i gdy potrzebuję prawdziwej zdalnej interaktywnej sesji z konsolą Windows PowerShell.

Wymagania strukturalne

Rozpatrując potencjalną okazję do napisania skryptu, zawsze należy przeanalizować wymagania ewentualnego skryptu. Poniżej znajduje się lista paru kwestii, które należy wziąć pod uwagę:

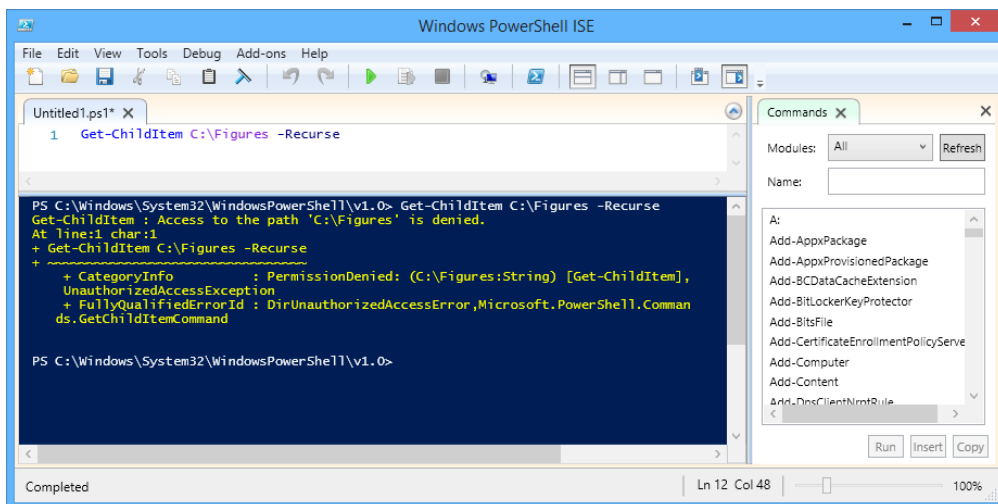
- wymagania dotyczące bezpieczeństwa,
- wymagania dotyczące wersji platformy .NET,
- wymagania dotyczące systemu operacyjnego,
- wymagania dotyczące aplikacji,
- wymagania dotyczące modułu.

Wymagania dotyczące bezpieczeństwa

W systemie Windows Vista wprowadzono funkcję *Kontrola konta użytkownika*, która ułatwia uruchamianie programów bez posiadania uprawnień administratora. Podczas gdy funkcja ta jest dobrodziejstwem dla użytkowników i działów ds. bezpieczeństwa firm, dla administratorów sieci i innych twórców skryptów potrzebujących zwiększonego poziomu uprawnień jest to przekleństwo.

Konsola Windows PowerShell nie omija zabezpieczeń. Jeśli skrypt próbuje wykonać jakąś czynność, której użytkownik nie ma prawa wykonywać, operacja ta nie powiedzie się.

W Windows PowerShell 4.0 poprawiono wykrywanie wymagań dotyczących bezpieczeństwa. Jak widać na rysunku 4.4, jeśli skrypt nie ma odpowiednich uprawnień, użytkownik zostaje o tym powiadomiony w konsoli.



RYСУNEK 4.4. Gdy skrypt nie ma odpowiednich uprawnień, w konsoli zostaje wyświetlona informacja o odmowie dostępu

Dobrze napisany skrypt powinien sprawdzać, jakie ma uprawnienia, i porównywać je z uprawnieniami potrzebnymi do poprawnego działania. Łatwym sposobem na sprawdzenie, czy ma się uprawnienia administratora, jest użycie instrukcji `#Requires`, której dokładniejszy opis zamieszczono w rozdziale 12. Poniżej znajduje się przykład jej użycia do sprawdzenia, czy skrypt ma uprawnienia administratora:

```
#Requires -RunAsAdministrator
```

Najpierw jednak trzeba pobrać informacje o użytkowniku, o czym jest mowa w następnym podrozdziale.

Wykrywanie bieżącego użytkownika

Do pobrania informacji o aktualnie zalogowanych użytkownikach można użyć klasy platformy `.NET Security.Principal.WindowsIdentity`. Znajduje się w niej metoda `GetCurrent` zwracająca egzemplarz obiektu typu `WindowsIdentity` reprezentującego bieżącego użytkownika. W poniższym przykładzie egzemplarz ten został zapisany w zmiennej `$user`:

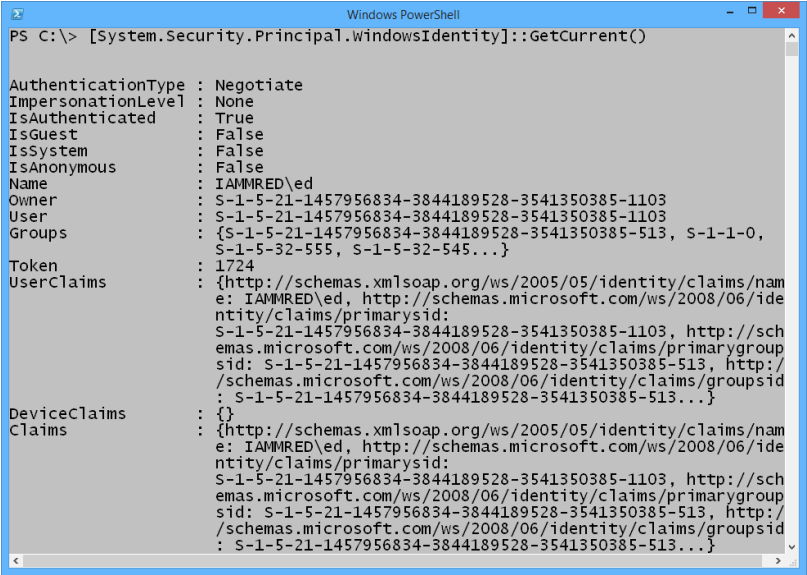
```
$user = [System.Security.Principal.WindowsIdentity]::GetCurrent()
```

W tabeli 4.2 znajduje się wykaz własności obiektu typu `WindowsIdentity`.

TABELA 4.2. Własności obiektu typu WindowsIdentity

| Nazwa | Definicja |
|--------------------|---|
| AuthenticationType | System.String AuthenticationType {get;} |
| Groups | System.Security.Principal.IdentityReferenceCollection Groups {get;} |
| ImpersonationLevel | System.Security.Principal.TokenImpersonationLevel ImpersonationLevel {get;} |
| IsAnonymous | System.Boolean IsAnonymous {get;} |
| IsAuthenticated | System.Boolean IsAuthenticated {get;} |
| IsGuest | System.Boolean IsGuest {get;} |
| IsSystem | System.Boolean IsSystem {get;} |
| Name | System.String Name {get;} |
| Owner | System.Security.Principal.SecurityIdentifir Owner {get;} |
| Token | System.IntPtr Token {get;} |
| User | System.Security.Principal.SecurityIdentifir User {get;} |

Po zapisaniu obiektu typu WindowsIdentity w zmiennej można wpisać nazwę tej zmiennej w wierszu poleceń, aby wyświetlić wartości wszystkich własności wymienionych w tabeli 4.2. Nie trzeba nigdzie zapisywać tej zmiennej, a dane można wyświetlić bezpośrednio, jak pokazano na rysunku 4.5.



```
PS C:\> [System.Security.Principal.WindowsIdentity]::GetCurrent()

AuthenticationType : Negotiate
ImpersonationLevel : None
IsAuthenticated    : True
IsGuest           : False
IsSystem          : False
IsAnonymous       : False
Name              : IAMMRED\ed
Owner             : S-1-5-21-1457956834-3844189528-3541350385-1103
User              : S-1-5-21-1457956834-3844189528-3541350385-1103
Groups            : {S-1-5-21-1457956834-3844189528-3541350385-513, S-1-1-0,
S-1-5-32-555, S-1-5-32-545...}
Token             : 1724
UserClaims        : {http://schemas.xmlsoap.org/ws/2005/05/identity/claims/nam
e: IAMMRED\ed, http://schemas.microsoft.com/ws/2008/06/ide
ntity/claims/primarysid:
S-1-5-21-1457956834-3844189528-3541350385-1103, http://sch
emas.microsoft.com/ws/2008/06/identity/claims/primarygroup
sid: S-1-5-21-1457956834-3844189528-3541350385-513, http://
/schemas.microsoft.com/ws/2008/06/identity/claims/groupsid
: S-1-5-21-1457956834-3844189528-3541350385-513...}

DeviceClaims      : {}
Claims            : {http://schemas.xmlsoap.org/ws/2005/05/identity/claims/nam
e: IAMMRED\ed, http://schemas.microsoft.com/ws/2008/06/ide
ntity/claims/primarysid:
S-1-5-21-1457956834-3844189528-3541350385-1103, http://sch
emas.microsoft.com/ws/2008/06/identity/claims/primarygroup
sid: S-1-5-21-1457956834-3844189528-3541350385-513, http://
/schemas.microsoft.com/ws/2008/06/identity/claims/groupsid
: S-1-5-21-1457956834-3844189528-3541350385-513...}
```

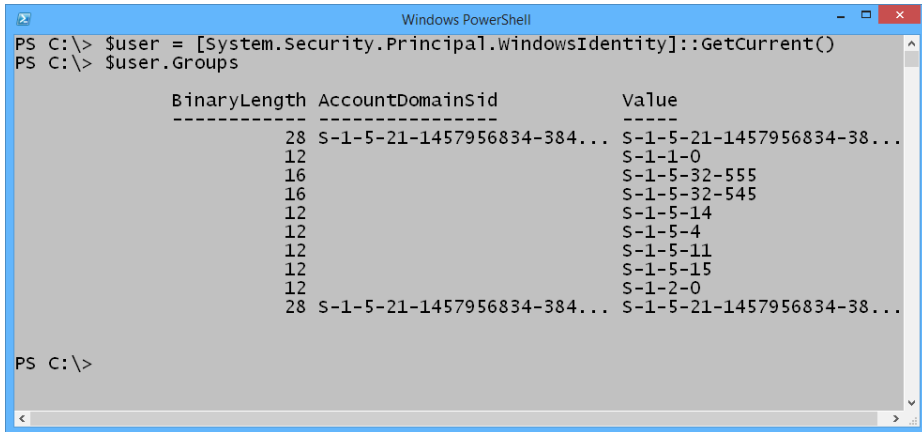
RYСУNEK 4.5. Wartość obiektu klasy WindowsIdentity wyświetlona w konsoli Windows PowerShell

Podczas gdy dane widoczne na tym rysunku mogą robić wrażenie i być przydatne, bo zawierają nazwę użytkownika i identyfikator SID, brak w nich informacji na temat uprawnień użytkownika do konkretnego zasobu czy w ogóle uprawnień administracyjnych. W istocie są dwa wymogi: czy użytkownik ma prawo używać danego zasobu oraz czy ma uprawnienia administratora.

Aby sprawdzić, czy użytkownik ma prawo używać wybranego zasobu, należy zbadać informacje grupowe. Jak widać w tabeli 4.2, klasa `WindowsIdentity` zawiera własność `Groups`. Jej wartość można łatwo wyświetlić w następujący sposób:

```
$user = [System.Security.Principal.WindowsIdentity]::GetCurrent()
$user.Groups
```

Wadą tego rozwiązania jest mało czytelny format wyniku, jak widać na rysunku 4.6.



RYСУNEK 4.6. Informacje z własności `Groups` klasy `WindowsIdentity` są wyświetlane w mało czytelnym formacie

Trzeba zamienić ten identyfikator SID na coś bardziej czytelnego. Gdy zna się rzeczywistą nazwę grupy, łatwiej sprawdzić, czy użytkownik należy do określonej grupy, za pomocą narzędzi konsoli Windows PowerShell do przetwarzania łańcuchów. Można bezpośrednio używać indeksów kolekcji `SecurityIdentifiers` zwracanej przez własność `Groups`, jak pokazano w poniższym przykładzie:

```
$user.Groups[0]
```

Oczywiście metoda ta ma jedną wadę. Skąd wiadomo, czym jest `group[0]`? Można sobie pomóc przy użyciu metody `ToString()`, jak pokazano poniżej:

```
PS C:\Users\bob> $user.Groups[0].ToString()
S-1-4-21-540299044-341859138-929407116-513
```

Teraz bezpośrednio pobraliśmy identyfikator SID grupy. W niektórych przypadkach może to wystarczyć, jeśli chcemy sprawdzać przynależność do grupy po identyfikatorach SID. Ale większość administratorów sieci nie ma takich informacji pod ręką, więc wynik trzeba jeszcze dalej przetworzyć.

Aby zamienić identyfikator SID w nazwę rzeczownikową, można użyć metody `Translate` z klasy platformy .NET `System.Security.Principal.SecurityIdentifier`. W tabeli 4.3 znajduje się wykaz składowych tej klasy.

TABELA 4.3. Tabela 4.3. Składowe klasy SecurityIdentifier

| Nazwa | Typ składowej | Definicja |
|-------------------|---------------|---|
| CompareTo | Metoda | System.Int32 CompareTo(SecurityIdentifier sid) |
| Equals | Metoda | System.Boolean Equals(Object o) System.Boolean Equals(SecurityIdentifier sid) |
| GetBinaryForm | Metoda | System.Void GetBinaryForm(Byte[] binaryForm Int32 offset) |
| GetHashCode | Metoda | System.Int32 GetHashCode() |
| GetType | Metoda | System.Type GetType() |
| IsAccountSid | Metoda | System.Boolean IsAccountSid() |
| IsEqualDomainSid | Metoda | System.Boolean IsEqualDomainSid(SecurityIdentifier sid) |
| IsValidTargetType | Metoda | System.Boolean IsValidTargetType(Type targetType) |
| IsWellKnown | Metoda | System.Boolean IsWellKnown(WellKnownSidType type) |
| ToString | Metoda | System.String ToString() |
| Translate | Metoda | System.Security.Principal.IdentityReference Translate(Type targetType) |
| AccountDomainSid | Własność | System.Security.Principal.SecurityIdentifier AccountDomainSid {get;} |
| BinaryLength | Własność | System.Int32 BinaryLength {get;} |
| Value | Własność | System.String Value {get;} |

Aby przekształcić identyfikator SID na nazwę grupy systemu Windows, należy zaznaczyć, że chce się dokonać zamiany na typ NTAccount, tworząc najpierw egzemplarz tego typu. W tym celu używa się łańcucha reprezentującego klasę System.Security.Principal.NTAccount, a następnie za pomocą operatora -as oznacza ten łańcuch jako typ:

```
$nt = "System.Security.Principal.NTAccount" -as [type]
```

Utworzonego typu NTAccount można używać z metodą Translate, jak pokazano poniżej:

```
PS C:\> $user = [System.Security.Principal.WindowsIdentity]::GetCurrent()
PS C:\> $nt = "System.Security.Principal.NTAccount" -as [type]
PS C:\> $user.Groups[0].translate($nt)
Value
-----
NWTRADERS\Domain Users
```

Może się wydawać, że potrzeba bardzo dużo pracy, aby dowiedzieć się, że aktualnie zalogowany użytkownik należy do grupy Domain Users. Ale biorąc pod uwagę fakt, że własność Groups zwraca kolekcję, można użyć pętli, aby przejrzeć wszystkie grupy jedna po drugiej. Poniżej pokazano przykład użycia polecenia pętlowego ForEach-Object:

```
PS C:\> $user = [System.Security.Principal.WindowsIdentity]::GetCurrent()
PS C:\> $nt = "System.Security.Principal.NTAccount" -as [type]
PS C:\> $user.Groups | ForEach-Object { $_.translate($nt) }
```

Value

```
NWTRADERS\Domain Users
Everyone
BUILTIN\Users
NT AUTHORITY\INTERACTIVE
NT AUTHORITY\Authenticated Users
NT AUTHORITY\This Organization
LOCAL
NWTRADERS\moreBogus
NWTRADERS\bogus
```

Mając nazwy grup, można je przeszukiwać za pomocą różnych operatorów, takich jak `-contains`, `-like` oraz `-match`.

Operatory `-contains`, `-like` oraz `-match`

Do przeszukiwania łańcuchów można używać przynajmniej trzech operatorów: `-contains`, `-like` oraz `-match`. Najtrudniej jest zrozumieć działanie operatora `-contains`. Nie chodzi o to, że jego obsługa jest jakoś wyjątkowo skomplikowana, tylko czasami trudno jest pojąć, kiedy powinno się go używać. Najlepiej to wyjaśnić na paru przykładach. W poniższym kodzie tworzona jest tablica liczb w zmiennej `$a`. Następnie za pomocą operatora `-contains` sprawdzamy, czy tablica zapisana w zmiennej `$a` zawiera liczbę 1. W wyniku otrzymujemy wartość `True` oznaczającą, że liczba ta znajduje się w tablicy. Następnie używamy operatora `-contains` do sprawdzenia, czy tablica zawiera liczbę 6. Nie zawiera, więc została zwrócona wartość `False`.

```
PS C:\> $a = 1,2,3,4,5
PS C:\> $a -contains 1
True
PS C:\> $a -contains 6
False
```

W następnym przykładzie w zmiennej `$b` zapisano liczbę 12345. Za pomocą operatora `-contains` sprawdzamy, czy liczba zapisana w zmiennej `$b` zawiera liczbę 4. Mimo że liczba 4 znajduje się w tej wartości, otrzymujemy wynik `False`. Liczba przechowywana w zmiennej `$b` nie zawiera 4. Następnie sprawdzamy tą samą metodą, czy zmienna `$b` zawiera liczbę 12345, i tym razem otrzymujemy wynik pozytywny.

```
PS C:\> $b = 12345
PS C:\> $b -contains 4
False
PS C:\> $b -contains 12345
True
```

Teraz powiedzmy, że zmienna `$c` zawiera następujący łańcuch: "To jest łańcuch". Jeśli za pomocą operatora `-contains` poszukamy łańcucha "jest", to otrzymamy fałszywy wynik. A jeśli za pomocą tego samego operatora poszukamy łańcucha "To jest łańcuch", to otrzymamy wynik pozytywny.

```
PS C:\> $c = "To jest łańcuch"
PS C:\> $c -contains "jest"
False
PS C:\> $c -contains "To jest łańcuch"
True
```

I jeszcze jeden przykład. Zmienna `$d` zawiera tablicę łańcuchów ("To", "jest", "łańcuch"). Jeśli za pomocą operatora `-contains` poszukamy łańcucha "jest", otrzymamy wynik pozytywny. Jeśli natomiast poszukamy łańcucha "ańcuch", otrzymamy wartość negatywną.

```
PS C:\> $d = "To","jest","łańcuch"
PS C:\> $d -contains "jest"
True
PS C:\> $d -contains "ańcuch"
False
```

Operator `-contains` służy do badania elementów tablicy. Jeśli tablica zawiera określoną wartość, operator ten zwraca wartość `True`. Jeśli w tablicy nie ma dokładnego odpowiednika szukanej wartości, operator `-contains` zwraca wartość `False`. Krótko mówiąc, za pomocą tego operatora można łatwo wyszukiwać elementy w tablicach.

Operator `-like` służy do wyszukiwania łańcuchów przy użyciu symboli wieloznacznych. Jeśli w zmiennej `$a` zapiszemy łańcuch "To jest łańcuch" i za pomocą operatora `-like` poszukamy łańcucha `"*ańcuch"`, to otrzymamy wartość zwrótną `True`.

```
PS C:\> $a = "To jest łańcuch"
PS C:\> $a -like "*ring*"
True
```

Ciekawym zastosowaniem operatora `-like` jest wyszukiwanie elementów tablicy. Jeśli w zmiennej `$b` zapiszemy tablicę "To", "jest", "łańcuch" i za pomocą operatora `-like` poszukamy w niej łańcucha `"*ańcuch"`, zostanie zwrócone każde dopasowanie, a nie tylko wartość `True` lub `False`.

```
PS C:\> $b = "To", "jest", "łańcuch"
PS C:\> $b -like "*ańcuch*"
łańcuch
```

Operator `-match` służy do wyszukiwania przy użyciu wyrażeń regularnych. Gdy znajdzie dopasowanie, zwraca wartość `True`. Jeśli nie znajdzie dopasowania, zwraca wartość `False`. Jeżeli zmiennej `$a` przypiszemy wartość "To jest łańcuch" i za pomocą operatora `-match` poszukamy wartości "jest", to zostanie znalezione dopasowanie tego wzorca i operator zwróci wartość `True`.

```
PS C:\> $a = "To jest łańcuch"
PS C:\> $a -match "jest"
True
```

Można też używać bardziej skomplikowanych wzorców. Na przykład znaku `\w` używa się w wyrażeniach regularnych do szukania białych znaków, np. spacji, przed lub za pewnym ciągiem. Jeśli w zmiennej `$a` zapiszemy łańcuch "To jest łańcuch" i użyjemy wyrażenia regularnego `[\w jest \w]`, to dopasowanie zostanie znalezione, jeśli przed i za słowem jest będzie znajdować się biały znak.

```
PS C:\> $a = "To jest łańcuch"
PS C:\> $a -match "[\w jest \w]"
True
```

A jak operator `-match` działa na tablicach? Jeśli w zmiennej `$c` zapiszemy tablicę "To", "jest", "łańcuch" i użyjemy wyrażenia regularnego "t", operator zwróci 2 dopasowania — łańcuchy zawierające szukany ciąg znaków. Wyniki zwracane są w postaci tablicy.

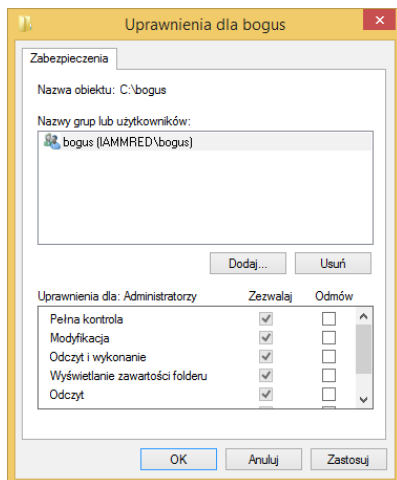
```
PS C:\> $c = "To", "jest", "łańcuch"
PS C:\> $c -match "t"
This
Is
```

Funkcja `Get-MemberOf` używa statycznej metody `GetCurrent` z klasy `System.Security.Principal.WindowsIdentity` w celu utworzenia obiektu klasy `WindowsIdentity`. Po utworzeniu typu `NTAccount` pobiera z własności `Groups` kolekcję grup zabezpieczeń, po czym za pomocą polecenia `ForEach-Object` zamienia identyfikatory SID przychodzących potokowo grup w `NTAccount`. Jeśli nazwa grupy odpowiada grupie używanej przy wywoływaniu tej funkcji, funkcja wyświetla informację, że użytkownik należy do tej grupy. Poniżej przedstawiono kod źródłowy skryptu *Get-MemberOf.ps1*:

Get-MemberOf.ps1

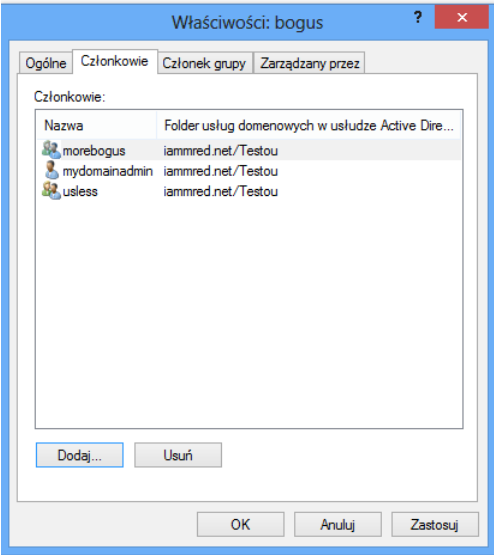
```
Function Get-MemberOf
{
    Param ($group)
    $user = [System.Security.Principal.WindowsIdentity]::GetCurrent()
    $nt = "System.Security.Principal.NTAccount" -as [type]
    If( $user.Groups.translate($NT) -match "$group" )
    { "Użytkownik $($user.name) należy do grupy $group." }
    ELSE
    { "Użytkownik $($user.name) nie należy do grupy $group." }
}
```

Przykład użycia funkcji `Get-MemberOf` ze skryptu *Get-MemberOf.ps1* zawarto w skrypcie *UseGetMemberOf.ps1*. Skrypt ten sprawdza, czy zalogowany użytkownik ma uprawnienia do folderu o nazwie *bogus*. Właściwości zabezpieczeń tego folderu pokazano na rysunku 4.7. Grupa *bogus* ma pełną kontrolę i nikt więcej nie ma do niego uprawnień.



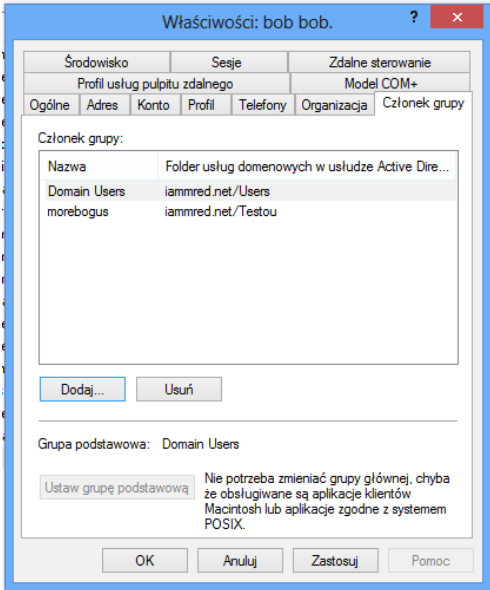
RYСУNEK 4.7. Tylko grupa *bogus* ma uprawnienia dostępu do folderu *bogus*

Grupa *bogus* ma jednego bezpośredniego użytkownika — *MyDomainAdmin*. Ponadto należą do niej dwie grupy: *morebogus* i *useless*. Są one pokazane na rysunku 4.8.



RYSUNEK 4.8. Grupa *bogus* zawiera inne grupy

Zalogowanym użytkownikiem jest Bob. Jak widać na powyższym rysunku, Bob nie jest bezpośrednim członkiem grupy *bogus*, tylko należy do grupy *morebogus*, co widać na rysunku 4.9.



RYSUNEK 4.9. Bob jest członkiem grup *morebogus* i *Domain Users*

W skrypcie *UseGetMemberOf.ps1* najpierw tworzony jest egzemplarz statycznej metody *GetCurrent* z klasy *System.Security.Principal.WindowsIdentity*. Obiekt klasy *WindowsIdentity*, który zostanie zwrócony, należy zapisać w zmiennej.

Następnie należy utworzyć egzemplarz typu *NTAccount* za pomocą operatora *-as* w celu rzutowania łańcucha *"System.Security.Principal.NTAccount"* jako typu. Użyjemy go później.

Najważniejsza część tego skryptu używa instrukcji *If* do ocenienia, czy nazwa grupy użytkowników znajduje się w kolekcji grup. Własność *Groups* zwraca kolekcję grup w obiekcie *\$users*. Przy użyciu funkcji automatycznego rozwijania kolekcji konsoli Windows PowerShell można sprawdzić każdą grupę z kolekcji bez używania polecenia *ForEach-Object*. Dzięki metodzie *Translate* przyjmującej utworzony wcześniej i przypisany do zmiennej *\$NT* typ *NTAccount* mamy teraz przetłumaczoną nazwę grupy i możemy za pomocą operatora *-match* sprawdzić, czy grupa zapisana w zmiennej *\$group* pasuje do jakiegoś obiektu z kolekcji. Wartości tej można użyć w instrukcji *If* tak, jakby była wartością logiczną, jak pokazano poniżej:

```
If( $user.Groups.translate($NT) -match "$group" )
```

Gdy zostanie znalezione dopasowanie, należy sprawdzić, czy plik rzeczywiście istnieje, za pomocą polecenia *Test-Path* przyjmującego ścieżkę zapisaną w zmiennej *\$bogusFile*. Polecenie *Test-Path* zwraca wartość logiczną. Poniżej znajduje się kod sprawdzający istnienie pliku:

```
If(Test-Path -Path $bogusFile)
```

Jeżeli plik istnieje, skrypt wchodzi do bloku kodu, który dodaje tekst do pliku. Do zapisania tekstu w pliku można użyć polecenia *Add-Content* pobierającego ścieżkę do pliku i dane, które mają zostać do niego dodane. Na końcu wiersza znajdują się dwa symbole specjalne: *`r* i *`n*. Pierwszy oznacza powrót karetki, a drugi — nowy wiersz. Razem stanowią odpowiednik słowa kluczowego *vbCrLf* z języka VBScript. W tabeli 4.4 znajduje się zestawienie znaków specjalnych.

TABELA 4.4. Znaki specjalne

| Znak | Definicja |
|------------------|------------------------------|
| <i>`0</i> (zero) | Null |
| <i>`a</i> | Alert |
| <i>`b</i> | Backspace |
| <i>`f</i> | Wysuw strony |
| <i>`n</i> | Nowy wiersz |
| <i>`r</i> | Powrót karetki |
| <i>`t</i> | Poziomy tabulator |
| <i>`v</i> | Pionowy tabulator |
| <i>`r`n</i> | Powrót karetki i nowy wiersz |

Przy dodawaniu tekstu do pliku tekstowego na ekranie wyświetlane jest powiadomienie i plik zostaje otwarty w Notatniku, jak pokazano poniżej:

```
{
  Add-Content -Path $bogusFile -Value "Dodano bzdurną treść.`r`n"
  "Dodano treść do pliku $bogusFile"
  Notepad $bogusFile
} #end if Test-Path
```

Jeżeli plik nie istnieje, na ekranie zostaje wydrukowana odpowiednia informacja:

```
ELSE
{
  "Nie znaleziono pliku $bogusFile"
} #end else
} #end if user
```

Jeśli użytkownik nie należy do grupy mającej uprawnienia do używania tego pliku, na ekranie zostaje wyświetlona jego nazwa wraz z informacją o braku przynależności do grupy:

```
ELSE
{
  "Użytkownik $($user.name) nie należy do grupy $group."
}
} #end GetMemberOf
```

Poniżej znajduje się kompletny kod źródłowy skryptu *UseGetMemberOf.ps1*:

UseGetMemberOf.ps1

```
Function Get-MemberOf
{
  Param ([string]$group,
        [string]$path)
  $user = [System.Security.Principal.WindowsIdentity]::GetCurrent()
  $nt = "System.Security.Principal.NTAccount" -as [type]
  If( $user.Groups.translate($NT) -match "$group" )
  { if(Test-Path -Path $path)
    {
      Add-Content -Path $path -Value "Dodano bzdurną treść.`r`n"
      "Dodano treść do pliku $path."
      Notepad $path
    } #end if Test-Path
  } ELSE
  { "Nie znaleziono pliku $path" } }
  ELSE
  { "Użytkownik $($user.name) nie należy do grupy $group." }
} # end function Get-MemberOf
```

Gdy funkcja *Get-MemberOf* zostanie wywołana, najpierw jest ładowana do pamięci. Aby ją wywołać, wystarczy uruchomić skrypt *UseGetMemberOf.ps1* w konsoli Windows PowerShell ISE. Poniżej pokazano przykład wywołania tej funkcji w celu sprawdzenia przynależności do grupy przed próbą uzyskania dostępu do pliku:

```
PS C:\> Get-MemberOf -group bogus -path 'C:\bogus\bogusfile.txt'
IAMMRED\ed is not a member of a bogus group
```

Wiedza tajemna

Zmiana sposobu pisania skryptów

Jeffrey Snover, Distinguished Engineer

Microsoft Corporation

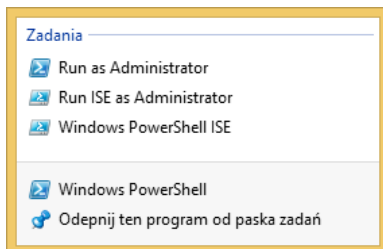
Wersja 2 zmieniła sposób, w jaki piszę funkcje i skrypty. Teraz zawsze w swoich funkcjach używam funkcji poleceń z wersji 2. Wcześniej funkcje były zaledwie bladymi substytutami poleceń cmdlet, ale teraz są im równe. Tak, teraz można pisać pełne polecenia cmdlet w samej konsoli Windows PowerShell. Dodanie tej możliwości wszystko zmienia. Ten niewielki kawałek dodatkowej składni stwarza wielkie możliwości. Jest podstawą tego, co nazywamy metaprogramowaniem, które zmieni świat. Jeszcze wspomnisz moje słowa.

Do metaprogramowania można używać polecenia `Import-PSSession`, za pomocą którego można zaimportować polecenia z sesji na zdalnym komputerze do lokalnego komputera i używać ich tak, jakby były w nim zainstalowane. Można używać funkcji uzupełniania nazw za pomocą klawisza `Tab`, pomocy, formatowania — wszystkiego. Ale wewnętrznie emitujemy funkcję o semantyce polecenia, która do wykonywania pracy używa zdalnej maszyny. Jest to naprawdę bardzo przydatne. Gdy ktoś zaczyna używać polecenia `Import-PSSession`, to jest to porównywalne z epokowym odkryciem przez człowieka możliwości własnoręcznego wykonywania narzędzi.

Mimo tych wszystkich nowych funkcji wiersza poleceń dodanych w Windows PowerShell 2.0 nadal piszę skrypty. W istocie piszę ich więcej, niż gdy używałem konsoli Windows PowerShell 1.0. Dlaczego? Po pierwsze i przede wszystkim: dla przyjemności. Po prostu bardzo lubię pisać skrypty Windows PowerShell. Czuję się, jakbym prowadził BMW. Ta niesamowita maszyna idealnie reaguje na moje polecenia oraz sprawia, że czuję się potężny i kompetentny. Przy użyciu konsoli Windows PowerShell 2.0 można zrobić tak wiele i tak łatwo, że często po napisaniu skryptu przyglądam się mu i myślę sobie: „To jest świetne!”. Podobne odczucia mam, gdy patrzę na skrypty napisane przez innych. Lee Holmes, starszy programista w firmie Microsoft, właśnie przysłał mi zawierający 103 linijki kodu skrypt, na widok którego aż oniemiałem. Nie mogę uwierzyć, jak wiele ten facet potrafi zdziałać przy użyciu 103 linijek kodu źródłowego. Ostatecznie przecież wszystko sprowadza się do efektywnego wykonywania pracy. Konsola Windows PowerShell bardzo w tym pomaga.

Wykrywanie roli użytkownika

Uruchomiona konsola Windows PowerShell może nie mieć uprawnień administratora. W takim przypadku wykonanie skryptu nie powiedzie się z powodu zbyt małych uprawnień, nawet jeśli użytkownik będzie administratorem. Jednym z rozwiązań jest uruchomienie konsoli Windows PowerShell lub Windows PowerShell ISE jako administrator. Robi się to, klikając prawym przyciskiem myszy skrót do programu i w menu podręcznym, które zostanie wyświetlone, klika się opcję `Uruchom jako administrator` (rysunek 4.10).



RYSUNEK 4.10. Kliknij prawym przyciskiem myszy skrót do konsoli Windows PowerShell, aby wyświetlić listę opcji zawierającą pozycję Uruchom jako administrator

Mógłbyś wstawić funkcję `Test-IsAdmin` znajdującą się w skrypcie `Test-IsAdminFunction.ps1` do swojego profilu lub każdego skryptu wymagającego do działania uprawnień administratora. Skrypt `Test-IsAdminFunction.ps1` zaczyna się od deklaracji funkcji `Test-IsAdmin` przyjmującej jedną wartość — zmienną o nazwie `$isAdmin`. Zmienna ta jest przekazywana przez referencję, co znaczy, że jej wartość będzie zmieniana wewnątrz funkcji. Aby przekazać zmienną przez referencję, należy przypisać jej ograniczenie typu `[ref]`. Ograniczenie to jest wymagane w deklaracji funkcji oraz przy jej wywołaniu w głównej części skryptu. Zmienna `$isAdmin` podczas przekazywania do funkcji jest pusta, ponieważ tak jest zadeklarowana w skrypcie. Poniżej pokazano deklarację omawianej funkcji:

```
Function Test-IsAdmin(
```

Kolejną czynnością w funkcji `Test-IsAdmin` jest użycie statycznej metody `GetCurrent` z klasy platformy `.NET Security.Principal.WindowsIdentity` w celu pobrania egzemplarza klasy `WindowsIdentity` reprezentującego bieżącego użytkownika. Zwrócony obiekt zostaje zapisany w zmiennej `$currentUser`, jak pokazano poniżej:

```
$currentUser = [Security.Principal.WindowsIdentity]::GetCurrent()
```

Następnie funkcja `Test-IsAdmin` tworzy reprezentujący bieżącego użytkownika egzemplarz klasy `WindowsPrincipal` poprzez przekazanie przechowywanego w zmiennej `$identity` obiektu `WindowsIdentity`. Jako że potrzebny jest nowy obiekt, użyto polecenia `New-Object`. Utworzony obiekt klasy `WindowsPrincipal` zostaje zapisany w zmiennej `$principal`, jak pokazano poniżej:

```
$principal = new-object security.Principal.WindowsPrincipal $identity
```

Następnie używamy statycznej własności `administrator` z wyliczenia `Security.Principal.WindowsBuiltInRole`. Wyliczenie ról administrator pokazano poniżej:

```
[security.principal.WindowsBuiltInRole]::administrator
```

Metodzie `IsInRole` z klasy `WindowsPrincipal` należy przekazać wyliczenie `WindowsBuiltInRole` utworzone w poprzednim wierszu kodu. Metoda ta zwraca wartość logiczną, `true` lub `false`, która jest zwracana bezpośrednio przez funkcję.

Poniżej znajduje się kompletny kod źródłowy skryptu `Test-IsAdminFunction.ps1`:

test-Isadmin.ps1

```
Function Test-IsAdmin
{
```

```
<#
  .Synopsis
    Sprawdza, czy użytkownik jest administratorem.
  .Description
    Zwraca wartość true, jeśli użytkownik jest administratorem, lub false, jeśli
    użytkownik nie jest administratorem.
  .Example
    Test-IsAdmin
#>
$identity = [Security.Principal.WindowsIdentity]::GetCurrent()
$principal = New-Object Security.Principal.WindowsPrincipal $identity
$principal.IsInRole([Security.Principal.WindowsBuiltInRole]::Administrator)
}
```

Aby użyć funkcji `Test-IsAdmin`, wywołuję ją w bloku ewaluacyjnym po załadowaniu jej do pamięci. Typowym sposobem jest użycie instrukcji `If` w celu sprawdzenia wartości logicznej zwracanej przez tę funkcję, jak pokazano poniżej:

```
PS C:\> if (Test-IsAdmin) {"Konsola ma uprawnienia administracyjne."}
Konsola ma uprawnienia administracyjne.
```

Wyliczenie platformy `.NET Security.Principal.WindowsBuiltInRole` może mieć następujące wartości:

- Administrator
- User
- Guest
- PowerUser
- AccountOperator
- SystemOperator
- PrintOperator
- BackupOperator
- Replicator

Wartości te są udokumentowane w podręczniku na stronach MSDN. Także w „Dodatkowych źródłach informacji” na końcu tego rozdziału znajduje się odnośnik do serwisu MSDN. Ale nie musisz przeszukiwać dokumentacji, jeśli chcesz tylko znaleźć wartości wyliczenia. Informacje te możesz zdobyć w konsoli Windows PowerShell za pomocą statycznej metody `GetNames` z klasy `.NET System.Enum`. Należy podać nazwę klasy wyliczeniowej `Security.Principal.WindowsBuiltInRole` jako parametr metody `GetNames` i nacisnąć klawisz *Enter*, aby otrzymać listę wszystkich elementów znajdujących się w tym wyliczeniu, jak pokazano poniżej:

```
PS C:\> [enum]::getnames("security.principal.WindowsBuiltInRole")
Administrator
User
Guest
PowerUser
AccountOperator
SystemOperator
PrintOperator
BackupOperator
Replicator
```

Klasa Enum zawiera także statyczną metodę `GetValues`, która zwraca listę wartości enumeratorów zamiast ich nazw. W tym przypadku wynik wykonania tej metody nie byłby fascynujący, ponieważ enumeratory i ich wartości w wyliczeniu `WindowsBuiltInRole` są takie same. Aby znaleźć wszystkie metody statyczne klasy Enum, można użyć poniższego wiersza kodu (choć klasy tej używam prawie wyłącznie do sprawdzania nazw wyliczeń platformy .NET):

```
[enum] | Get-Member -Static -MemberType method
```

Jako że w klasie Enum nie ma żadnych statycznych właściwości, możemy opuścić parametr `-MemberType` i skrócić polecenie do następującej postaci:

```
[enum] | Get-Member -Static
```

Jeśli nie lubisz dużo pisać, możesz też użyć następującej formy tego polecenia:

```
[enum] | gm -s
```

Skrypt *GetAdminFunction.ps1* można łatwo zmodyfikować tak, aby dostarczał informacji na podstawie innych ról dostępnych w klasie `Security.Principal.WindowsBuiltInRole`. Przede wszystkim należy zmienić wpisaną na sztywno rolę administratora na zmienną, jak pokazano poniżej:

```
[security.principal.WindowsBuiltInRole]::$roleName
```

Pozostałe zmiany, jakie należy wprowadzić w skrypcie, dotyczą nazw zmiennych i tekstu wyjściowego. Poniżej znajduje się zmodyfikowany skrypt *Test-IsInRole.ps1*:

test-IsInrole.ps1

```
Function Test-Isinrole
{
    <#
        .Synopsis
            Sprawdza, czy użytkownik jest w określonej roli.
        .Description
            Zwraca wartość true, jeśli użytkownik jest w danej roli, lub false, jeśli nie.
        .Example
            Test-Isinrole -role Guest
    #>
    Param($roleName)
    $identity = [Security.Principal.WindowsIdentity]::GetCurrent()
    $principal = New-Object Security.Principal.WindowsPrincipal $identity
    $principal.IsInRole([Security.Principal.WindowsBuiltinRole]::$roleName)
}
```

Aby wywołać funkcję `Test-IsInRole`, najpierw należy ją załadować, a następnie obudować kodem ewaluacyjnym i przekazać jej jedną z dopuszczalnych wartości wyliczenia. Poniżej pokazano przykład:

```
PS C:\> if (Test-Isinrole -roleName 'guest') {"Konsola jest w roli"} ELSE {"Konsola nie jest w roli"}
Konsola nie jest w roli
```

Zapiski praktyka

Jonathan Tyler, informatyk

Gdy pracowałem jako administrator systemu, często używałem konsoli Windows PowerShell do diagnostyki. Teraz jestem programistą, a mimo to nadal często używam tego narzędzia do tego samego celu, tylko robię to nieco inaczej.

Jako że język Windows PowerShell jest pod względem składni bardzo podobny do języka C#, lubię testować kod w tej konsoli, zanim ostatecznie dodam go do projektu. Dzięki takiemu wykorzystaniu konsoli Windows PowerShell mogę wykonywać kod po linijce albo po dwie linijki, aby zobaczyć, jak działa, bez konieczności wielokrotnego przeprowadzania kompilacji. W ten sposób oszczędzam sporo czasu na kompilowaniu i testowaniu. Ponadto używam mniej bloków try-catch, ponieważ mogę z góry zaplanować, jakiego typu wyjątków wypatrywać. Nie jest to absolutnie niezawodne rozwiązanie, ale pomaga w programowaniu.

Czasami też za pomocą konsoli Windows PowerShell przygotowuję dane swoich projektów C#. Przy użyciu narzędzi automatyzacyjnych szybko tworzę pliki wejściowe i inne potrzebne mi zasoby. Przykładowo za pomocą konsoli Windows PowerShell mogę szybko wygenerować plik CSV zawierający kilkaset wierszy losowych danych testowych.

Wymagania dotyczące wersji platformy .NET

Dostępność platformy .NET w konsoli Windows PowerShell oraz jej elastyczność i narzędzia pomocne dla programistów sprawiają, że trzeba wziąć pod uwagę jeszcze jedną rzecz — wersję platformy .NET zainstalowanej w komputerze. Można ją sprawdzić na kilka sposobów.

Aby sprawdzić wersję systemu .NET, można użyć statycznej metody `GetSystemVersion` z klasy `.NET System.Runtime.InteropServices.RuntimeEnvironment`:

```
[runtime.interopServices.RuntimeEnvironment]::GetSystemVersion()
```

Ale gdy wywołuję tę metodę w swoim komputerze, otrzymuję wartość `v4.0.30319` oznaczającą .NET Framework 4.5. Wartość taka jest zwracana ze względu na sposób instalacji platformy — dodatki serwisowe są uważane za rozszerzenia systemu wykonawczego.

Jednym ze sposobów na znalezienie konkretnej wersji platformy .NET jest sprawdzenie w rejestrze. Każda wersja tej platformy podczas instalacji dodaje do rejestru klucz. Od wersji 4.5 platformy .NET należy odczytać wartość klucza `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\NET Framework Setup\NDP\v4\Full\Release`. Do tego celu można użyć polecenia `cmdlet Get-ItemProperty` i dostawy rejestru, jak pokazano poniżej:

```
PS C:\> (Get-ItemProperty -Path 'HKLM:\SOFTWARE\Microsoft\NET Framework
Setup\NDP\v4\Full').Release
378675
```

Numer wydania platformy .NET 4.5 to 378389, natomiast platformy .NET 4.5.1 to 378675.

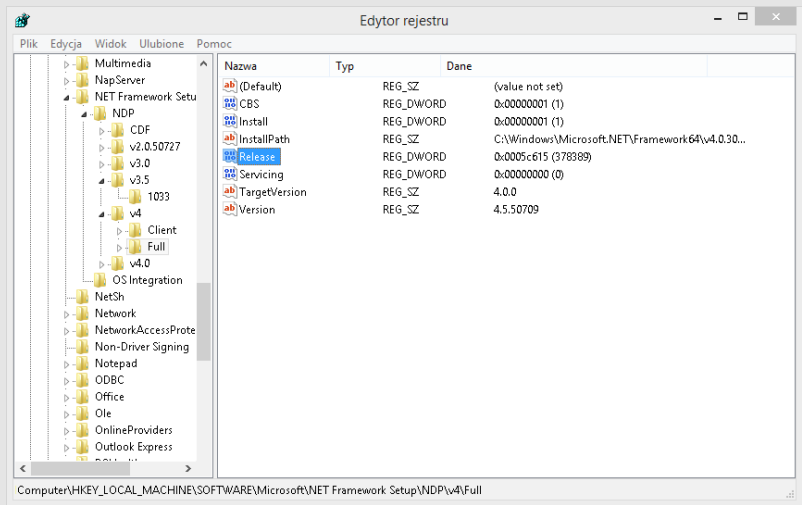
Zapiski praktyka

Sprawdzanie wersji platformy .NET

Luís Canastreiro, Premier Field Engineer
Microsoft, Portugalia

Gdy piszę skrypt wykorzystujący cechę charakterystyczną konkretnej wersji platformy .NET, np. 3.5, zawsze dodaję test sprawdzający, czy w komputerze jest zainstalowana odpowiednia wersja tej platformy. Jeden z najprostszych testów sprawdza, czy istnieje określona ścieżka instalacyjna: `%Systemroot%\Microsoft.NET\Framework\v3.5`. Jako że folder ten nie istnieje, jeśli platforma .NET 3.5 nie jest zainstalowana, to taki prosty test pozwala mi upewnić się, że skrypt zostanie wykonany poprawnie.

Od czasów platformy .NET 2.0 w rejestrze dla każdej nowej wersji tworzony jest klucz. Lubię się nim posługiwać, ponieważ zawiera dodatkowe informacje, np. datę instalacji danej wersji, ścieżkę, numer wersji oraz czy dana wersja została zainstalowana w dodatku serwisowym. Na rysunku 4.11 widać klucz rejestru z komputera z zainstalowaną platformą .NET w wersji 3.5. Została ona zainstalowana w dodatku serwisowym.



RYSUNEK 4.11. W rejestrze można łatwo sprawdzić różne informacje dotyczące wersji platformy .NET

Dezorientujące może być to, że platforma .NET to nie to samo co CLR (ang. *Common Language Runtime*). Jest całkiem możliwe, aby w komputerze były zainstalowane CLR 2.0 SP2 i .NET Framework 3.5 SP1. Gdy instaluje się platformę .NET 3.5 z dodatkiem SP1, uaktualnia się CLR do wersji 2.0 SP2. Platformę .NET można traktować jak pakiet zawierający CLR i zarządzane biblioteki do implementowania graficznych interfejsów użytkownika, używania usług sieciowych, używania funkcji systemu Windows itd. oraz kompilatory i narzędzia do pracy z językami zarządzanymi.

Wymagania dotyczące systemu operacyjnego

Konsolę Windows PowerShell 4.0 można zainstalować w wielu systemach operacyjnych, poczynając od Windows 7. W każdym z nich można uruchamiać skrypty, których działanie zależy od pewnych funkcji dostępnych tylko w określonej wersji systemu.

Jednym ze sposobów na sprawdzenie wersji systemu operacyjnego jest użycie statycznej właściwości `OSVersion` klasy platformy `.NET System.Environment`, jak pokazano poniżej:

```
PS C:\> [System.Environment]::OSVersion | Format-List
```

```
Platform      : Win32NT
ServicePack   :
Version       : 6.3.9478.0
VersionString : Microsoft Windows NT 6.3.9478.0
```

Jak widać, wersja systemu operacyjnego składa się z czterech części liczbowych. Oto opis każdej z nich:

- **Numer wersji głównej (ang. *major*)** — złożenia o tej samej nazwie, ale różnych numerach wersji głównej nie są zamienne. Na przykład numer ten zmienia się, gdy w produkcie są wprowadzane poważne zmiany mogące powodować brak zgodności z poprzednią wersją.
- **Numer wersji pomocniczej (ang. *minor*)** — jeśli dwa złożenia mają takie same nazwy i numery główne, ale różne numery wersji pomocniczej, to znaczy, że wprowadzono ważne poprawki z zachowaniem zgodności z poprzednią wersją. Na przykład numer ten zmienia się w wydaniach poprawkowych produktu oraz nowych wersjach, które są w pełni zgodne z poprzednią wersją.
- **Numer wersji kompilacji (ang. *build number*)** — różnica między numerami kompilacji oznacza różne kompilacje tego samego kodu źródłowego. Część ta odzwierciedla zmiany procesora, platformy lub kompilatora.
- **Numer poprawki (ang. *revision*)** — złożenia opatrzone tą samą nazwą oraz tym samym numerem wersji głównej i pomocniczej, ale różnymi numerami poprawki powinny być w pełni zamienne. Numer ten zmienia się po załatwieniu luki w zabezpieczeniach wykrytej w poprzedniej wersji.

Jeśli zapiszesz wyniki statycznej właściwości `[environment]::OSVersion` w zmiennej, otrzymasz obiekt klasy `System.Version`, który jest zwracany dla właściwości `version`, jak pokazano poniżej:

```
PS C:\> [System.Environment]::OSVersion | Get-Member -MemberType Properties
```

```
TypeName: System.OperatingSystem
```

| Name | MemberType | Definition |
|---------------|------------|-----------------------------------|
| ---- | ----- | ----- |
| Platform | Property | System.PlatformID Platform {get;} |
| ServicePack | Property | string ServicePack {get;} |
| Version | Property | version Version {get;} |
| VersionString | Property | string VersionString {get;} |

Zaletą tego podejścia jest to, że klasa `System.Version` zapewnia łatwy dostęp do wszystkich właściwości numeru wersji. Obiekt tej klasy jest zwracany z właściwości `OSVersion.Version`, jak pokazano poniżej:

```
PS C:\> [System.Environment]::OSVersion.Version | Get-Member -MemberType Properties
```

```
TypeName: System.Version
```

| Name | MemberType | Definition |
|---------------|------------|----------------------------|
| Build | Property | int Build {get;} |
| Major | Property | int Major {get;} |
| MajorRevision | Property | int16 MajorRevision {get;} |
| Minor | Property | int Minor {get;} |
| MinorRevision | Property | int16 MinorRevision {get;} |
| Revision | Property | int Revision {get;} |

Przy użyciu tej techniki można zaznaczyć, że chce się wykonać kod tylko w wybranej wersji systemu operacyjnego. Przykład tego pokazano w skrypcie *Get-OSVersion.ps1*. Na początku wywoływana jest funkcja `Get-OSVersion`. W jej deklaracji zdefiniowana jest zmienna jako typ referencyjny; przy jej użyciu funkcja może zwrócić informację o wersji systemu operacyjnego do tej części skryptu, która ją wywołała. W deklaracji funkcji określona jest jej nazwa i podane są zmienne wejściowe, jak widać poniżej:

```
Function Get-OSVersion([ref]$os)
```

Blok kodu tej funkcji pobiera właściwość `OSVersion` z klasy `.NET System.Environment` i przypisuje ją do właściwości `Value` zmiennej `$os`, jak pokazano poniżej:

```
$os.value = [environment]::OSVersion
```

Aby użyć funkcji `Get-OSVersion`, należy w wywołaniu przekazać jej zmienną typu referencyjnego. Poniższy kod inicjuje zmienną `$os` wartością pustą, a następnie przekazuje ją do funkcji.

```
$os = $null
Get-OSVersion([ref]$os)
```

Do sprawdzenia numeru wersji głównej można użyć instrukcji typu `If-Else`. W sprawdzaniu numerów wersji mogą być pomocne informacje zawarte w tabeli 4.5. Możesz być zaskoczony, że zarówno Windows Vista, jak i Windows Server 2008 mają ten sam numer wersji. Jeśli znajdziesz dopasowanie, możesz przejść do następnej części skryptu. A jeśli nie, możesz zakończyć działanie skryptu, jak pokazano poniżej:

```
if($os.version.major -ge 6)
{
    "Windows Vista lub nowszy"
}
else
{
    "Nie Windows Vista ani nowszy"
    exit
}
```

TABELA 4.5. Nazwy i wersje systemów operacyjnych

| Numer wersji | Nazwa systemu operacyjnego |
|--------------|----------------------------|
| 5.1.2600 | Windows XP |
| 5.2.3790 | Windows Server 2003 |
| 6.0.6001 | Windows Vista |
| 6.0.6001 | Windows Server 2008 |
| 6.1.6801 | Windows 7 |
| 6.1.6801 | Windows Server 2008 R2 |
| 6.2.9200 | Windows 8 |
| 6.2.9200 | Windows Server 2012 |
| 6.3.9600 | Windows 8.1 |
| 6.3.9600 | Windows Server 2012 R2 |

Poniżej znajduje się kompletny kod skryptu *Get-OSVersion.ps1*.

Get-OSVersion.ps1

```
Function Get-OSVersion
{
    [System.Environment]::OSVersion.Version
}
```

*# *** punkt początkowy skryptu ****

```
$os = Get-OSVersion
if($os.major -ge 6 -and $os.Minor -ge 2)
{ "Windows 8 lub nowszy" }
else
{ "Nie Windows 8 ani nowszy" }
```

Klasa .NET `System.Environment` nie zawiera żadnych narzędzi do pracy zdalnej. To znaczy, że nie można jej przekazać łańcucha i oczekiwać, że połączy się ze zdalnym komputerem, aby pobrać informacje. W konsoli Windows PowerShell 2.0 nie jest to problemem, ponieważ do zdalnego wykonywania poleceń można używać polecenia `Invoke-Command`, jak pokazano poniżej:

```
PS C:\> $computers = "berlin","win7"
PS C:\> Invoke-Command -ComputerName $computers -ScriptBlock {[environment]::OSVersion }
```

```
PSComputerName      : berlin
RunspaceId          : d23f85ed-3f2b-465b-877a-37dd43125f40
PSShowComputerName  : True
Platform            : Win32NT
ServicePack         : Service Pack 1
Version             : 6.0.6001.65536
VersionString       : Microsoft Windows NT 6.0.6001 Service Pack 1
```

```
PSComputerName      : win7
RunspaceId          : 04b1ce80-19e9-4dde-9b8d-8725b032dfff
```

```
PSShowComputerName : True
Platform            : Win32NT
ServicePack         :
Version             : 6.1.6801.0
VersionString       : Microsoft Windows NT 6.1.6801.0
```

Wiedza tajemna

Po co pisze się skrypty

Jeffrey Snover, Distinguished Engineer

Microsoft Corporation

Przewiduję, że specjaliści od pisania skryptów zaleją nas skryptami, ponieważ bardzo łatwo się je pisze, udostępnia i diagnozuje. Wystarczy opublikować taki skrypt na swoim blogu i już można poprawiać świat. Inni nie tylko będą mogli użyć naszej funkcji, ale dodatkowo dowiedzą się, jak ją zbudowaliśmy. Niektórzy może nawet napiszą parę słów komentarza i przy okazji czegoś nas nauczą. Sam wiele się nauczyłem, czytając kod źródłowy skryptów napisanych przez innych programistów oraz czytając komentarze na temat moich własnych skryptów.

Uwielbiam graficzne interfejsy użytkownika, ale jeśli korzysta się z nich przez cały dzień, to wieczorem bolą już ręce. Gdy natomiast napisze się skrypt, to ma się coś, czego można użyć wielokrotnie, a tym samym dzięki czemu można pracować sprawniej i szybciej, a więc stać się cenniejszym pracownikiem (to z kolei ułatwia znalezienie pracy i zdobycie wyższych zarobków). Skryptem można podzielić się z innymi, a każdy, kto użyje naszego produktu, będzie nam wdzięczny. Każdy może podziwiać nasze dzieło, skomentować je i czegoś się z niego nauczyć. Dzieło można analizować, krytykować i doskonalić. Pisząc skrypt, dołączasz do wielkiej społeczności, która z każdym kolejnym skryptem uczy się nowych rzeczy.

W konsoli Windows PowerShell 1.0 skrypty pisze się trudno, ponieważ funkcje nie umożliwiają generowania poprawnej semantyki, nie można dostarczać do nich pomocy oraz brakuje udogodnień do udostępniania skryptów innym. W Windows PowerShell 2.0 rozszerzono funkcje oraz dodano moduły rozwiązujące wszystkie problemy, dzięki czemu publikowanie skryptów stało się o wiele łatwiejsze.

Używam zarówno graficznej wersji konsoli Windows PowerShell, jak i tej w postaci wiersza poleceń, ale nie używam już Notatnika do pisania skryptów. Zastąpiła go konsola Windows PowerShell ISE, która jest doskonałym narzędziem do pisania i diagnozowania skryptów.

Co robię z konsolą Windows PowerShell? Badam! Jedną z moich ulubionych cech tej konsoli jest możliwość przejrzenia wielu aspektów systemu operacyjnego. Przy jej użyciu można łatwo i bezpiecznie sprawdzać informacje o takich składnikach jak WMI, .NET, rejestr, COM, pliki itd. Jeśli znajdę coś ciekawego, piszę skrypt, który zwykle udostępniam do użytku innym.

Jeśli dopiero uczysz się obsługi konsoli Windows PowerShell, wiedz jedno: musisz nauczyć się uczyć się. Lubię opowiadać historię pewnej grupy początkujących i doświadczonych administratorów systemu UNIX, którym dano do rozwiązania pisemny test. Specjaliści

wypadli w nim niewiele lepiej od nowicjuszy. Ale kiedy wszystkich posadzono przed komputerami i zlecono im konkretne zadania do wykonania, eksperci zmiażdżyli początkujących rywali. Wniosek z tego jest taki, że eksperci mogą pamiętać niewiele więcej teorii od początkujących, ale z pewnością lepiej rozwiązują praktyczne problemy. Skup się na nauce. Naucz się posługiwać się poleceniami `Get-Help` i `Get-Member`. Naucz się używać narzędzi do pracy z obiektami. Potem zacznij eksplorować. Zdziwisz się, jak dużo można zdziałać przy użyciu kombinacji tylko podstawowych narzędzi. Ale taki sposób działania stanowi fundament systemów kompozycyjnych.

W Windows PowerShell nie ma poleceń w rodzaju „zrób to za mnie”, ale jest zbiór narzędzi pozwalający wykonywać zadania przy użyciu kombinacji kilku poleceń. Musisz nauczyć się łączyć ze sobą różne elementy i w ogóle poznać te elementy. W trakcie nauki wykorzystaj wiedzę zbiorową, co nie będzie trudne, zważywszy na to, że społeczność bardzo chętnie pomaga początkującym.

Wymagania dotyczące aplikacji

Po upewnieniu się, że skrypt ma odpowiednie uprawnienia i w systemie operacyjnym jest zainstalowana właściwa wersja platformy .NET, konieczne może być jeszcze sprawdzenie, czy w docelowej maszynie jest uruchomiona jakaś aplikacja. Do tego celu można wykorzystać polecenie `Get-Process` lub `Get-Service`.

Za pomocą skryptu *GetRunningService.ps1* można sprawdzić, czy w komputerze jest utworzona określona usługa oraz czy została ona uruchomiona. Do sprawdzania istnienia usługi skrypt używa konstrukcji `If-Else`. W instrukcji `If` za pomocą polecenia `Get-Service` pobrano listę wszystkich usług zdefiniowanych w bieżącym komputerze. Za pomocą parametru `-computername` tego polecenia można pobrać informacje z komputera zdalnego. Wyniki zwrócone przez polecenie `Get-Service` są przekazywane do polecenia `Where-Object` w celu przefiltrowania. Zastosowano dwa kryteria filtrowania: status usługi (musi być uruchomiona) oraz jej nazwa. Poniżej pokazano opisywaną część kodu:

```
Get-Service |
Where-Object { $_.status -eq 'running' -AND $_.name -eq $serviceName }
```

Jeśli warunek ten zostanie spełniony, skrypt rozpoczyna wykonywanie bloku kodu związanego z instrukcją `If`. W tym przykładzie skrypt drukuje informację, że usługa jest uruchomiona. Jeżeli usługi nie ma lub nie jest uruchomiona, skrypt również drukuje stosowną informację. W obu tych blokach znajduje się kod, którego wykonanie zależy od stanu badanej usługi. Poniżej pokazano kompletny kod źródłowy skryptu *GetRunningService.ps1*:

GetRunningService.ps1

```
$serviceName = "ZuneBusEnum"
if(
    Get-Service |
    Where-Object { $_.status -eq 'running' -AND $_.name -eq $serviceName }
)
{
    "Usługa $serviceName działa."
} #end if
```

```
ELSE
{
    "Usługa $serviceName nie działa."
} #end else
```

Czasami interesuje nas status procesu, a nie usługi. Aby sprawdzić, czy wybrany proces istnieje, należy użyć polecenia `Get-Process`. W tym przypadku logika jest prostsza, bo procesy istnieją tylko wtedy, gdy są uruchomione, a więc nie trzeba stosować klauzuli `WHERE`. Poniżej pokazano uproszczony kod:

```
Get-Process | Where-Object ProcessName -eq $processName
```

Pozostała pokazana poniżej część skryptu *GerRunningProcess.ps1* jest podobna do skryptu *GetRunningService.ps1*.

GetRunningProcess.ps1

```
$processName = "iexplore"
if(
    Get-Process | Where ProcessName -eq $processName
)
{
    "Proces $processName działa."
} #end if
ELSE
{
    "Proces $processName nie działa."
} #end else
```

Wymagania dotyczące modułów

Rozszerzalność jest jedną z największych zalet konsoli Windows PowerShell. Dzięki tej cesze z internetu można pobierać różne darmowe moduły zawierające dziesiątki dodatkowych poleceń. Można też kupować produkty komercyjne dużych firm rozwiązujące wiele ważnych problemów. Oba rozwiązania mają ze sobą coś wspólnego: polecenia są dostarczane w modułach. Muszą być spełnione dwa warunki: moduł musi być zainstalowany oraz załadowany. Oczywiście załadowanie modułu to żaden problem, ponieważ konsola Windows PowerShell automatycznie ładuje moduły znajdujące się na ścieżce `$env:PSModulePath`. Ale jeśli moduł jest zapisany gdzie indziej, to trzeba go załadować własnoręcznie.

Aby zapewnić dostępność wybranego modułu Windows PowerShell, przed wykonaniem należących do niego poleceń należy użyć dyrektywy `#Requires`. Powinna ona znajdować się w pierwszym wierszu skryptu. Dyrektyw `#Requires` może być kilka, jeśli takie są wymagania. W takim przypadku każda z nich musi zajmować osobny wiersz. Przykład zastosowania tej techniki pokazano w poniższym skrypcie *RequiresModule.ps1*:

requiresModule.ps1

```
#requires -modules activedirectory
#requires -version 4

Get-aduser -filter *
```

Zapiski praktyka

Todd Klindt, MVP Microsoft SharePoint
Konsultant ds. SharePoint

Zawodowo zajmuję się oprogramowaniem SharePoint. Żyję nim i z niego. Gdybym miał wybrać moją drugą ulubioną technologię, prawdopodobnie wybrałbym Windows PowerShell. Robiłem wszystko, aby przed nią uciec, ale w końcu wkradła się do mojego serca i je przejęła.

Pracowałem jako administrator systemu SharePoint długo przed pojawieniem się wersji SharePoint 2010, w której zostałem zmuszony do używania konsoli Windows PowerShell. Administrowałem dużą farmą SharePoint 2003, a potem SharePoint 2007 i całkiem dobrze sobie radziłem z narzędziem wiersza poleceń tego systemu o nazwie *stsadm.exe*. Nie potrzebowałem żadnych wymyślnych konsol do pracy. Ale w końcu skapitulowałem i niechętnie i opornie zacząłem uczyć się obsługi konsoli Windows PowerShell. Każdy ma początkowo problemy z tą konsolą i ja też nie byłem wyjątkiem.

Początkowo szło mi ciężko, ale w końcu zacząłem łapać, o co w tym chodzi. Może nie obudziłem się pewnego ranka ze śpiewem na ustach „W końcu pokonałem konsolę Windows PowerShell; czas na podbój świata”, ale w pewnym momencie dotarło do mnie, że już tak często nie tłukę głową w ścianę. W końcu napisanie jednej linijki działającego kodu przestało zajmować mi godzinę. Konsola uczyła mnie, jak postępować zgodnie z jej zasadami. Nauka szła mi opornie, ale szła.

Teraz ja i konsola Windows PowerShell stanowimy nierozłączny duet. Zawsze mam otwarte przynajmniej jedno okno i często szukam różnych sposobów wykonywania zadań przy jej użyciu. Początkowo w SharePoint konsoli Windows PowerShell używałem tylko dlatego, bo musiałem. Teraz używam jej z wielką przyjemnością do wszelkich prac związanych z produktami firmy Microsoft. I już nie biję głową w ścianę. Prawie nigdy.

Dodatkowe źródła informacji

- Centrum skryptów portalu TechNet pod adresem <http://www.microsoft.com/technet/scriptcenter> zawiera wiele przykładowych skryptów.
- Na stronie <http://blogs.msdn.com/dougste/archive/2007/09/06/version-history-of-the-clr-2-0.aspx> znajduje się opis historii wersji platformy .NET.
- Na stronie <http://msdn.microsoft.com/en-us/library/lh925568.aspx> znajdują się dodatkowe informacje na temat sprawdzania, która wersja platformy .NET jest zainstalowana w komputerze.
- Adres strony głównej serwisu MSDN to <http://msdn.microsoft.com>.
- Wszystkie skrypty opisane w tym rozdziale są dostępne do pobrania w repozytorium centrum skryptów portalu TechNet pod adresem <http://gallery.technet.microsoft.com/scriptcenter/PowerShell-40-Best-d9e16039>.

Rozdział 5

Konfigurowanie środowiska skryptowego

- Konfigurowanie profilu
- Tworzenie profilu
- Używanie funkcji z innych skryptów
- Dodatkowe źródła informacji

W konsoli Windows PowerShell 4.0 jest wiele możliwości konfiguracji i dostosowywania do własnych potrzeb środowiska skryptowego. Użytkownik może zmienić sposób uruchamiania narzędzia, sposób jego działania, a nawet składnię często używanych funkcji. Niestety wygoda ta ma swoją cenę: można tak zmienić środowisko skryptowe, że nie będzie można rozpoznać poleceń, nie będzie wiadomo, jak ich używać ani nawet jak szukać pomocy. W rozdziale tym znajduje się opis technik dostosowywania środowiska stosowanych przez ekspertów oraz udogodnień, które mogą być przydatne zarówno zaawansowanym użytkownikom, jak i pracownikom działów informatycznych firm.

Konfigurowanie profilu

Standardowa instalacja konsoli Windows PowerShell nie zawiera żadnego profilu. Profile służą do konfigurowania środowiska skryptowego oraz ułatwiają pracę z wierszem poleceń Windows PowerShell (lub rozszerzają możliwości konsoli Windows PowerShell ISE). Profil Windows PowerShell to miejsce, w którym można przechowywać następujące elementy:

- aliasy,
- funkcje,
- dyski programu PowerShell,
- zmienne.

Tworzenie aliasów

Aliasy są przydatne z punktu widzenia użyteczności. Weźmy na przykład takie polecenie jak Measure-Object, które zwraca różne dane statystyczne o obiekcie, np. jego minimalną i maksymalną wartość. Wpisywanie nazwy tego polecenia w wierszu poleceń jest dość niewygodne. A biorąc pod uwagę, jak często jest potrzebne, warto rozważyć możliwość utworzenia krótszego aliasu. Zanim utworzy się własny alias, warto sprawdzić, czy dane polecenie nie ma jeszcze standardowego aliasu. W konsoli Windows PowerShell w systemie Windows 8.1 znajduje się ponad 190 gotowych aliasów dla 271 poleceń i funkcji. Jeśli weźmie się pod uwagę, że niektóre polecenia mają więcej niż jeden alias, łatwo się domyslić, że jest jeszcze sporo okazji do zdefiniowania własnych nazw zastępczych. Przedstawiony poniżej skrypt *GetCmdletsWithMoreThanTwoAliases.ps1* wyświetla listę wszystkich poleceń mających więcej niż jeden alias:

```
GetCmdletsWithMorethantwoaliases.ps1

Get-Alias |
Group-Object -Property definition |
Sort-Object -Property count -Descending |
Where-Object count -gt 2
```

Wynik działania tego skryptu jest następujący:

| Count | Name | Group |
|-------|-------------------|-------------------------|
| ----- | ---- | ----- |
| 6 | Remove-Item | {del, erase, rd, ri...} |
| 3 | Get-ChildItem | {dir, gci, ls} |
| 3 | Invoke-WebRequest | {curl, iwr, wget} |
| 3 | Copy-Item | {copy, cp, cpi} |
| 3 | Move-Item | {mi, move, mv} |
| 3 | Get-History | {ghy, h, history} |
| 3 | Get-Content | {cat, gc, type} |
| 3 | Set-Location | {cd, chdir, sl} |

Aby dowiedzieć się, czy polecenie Measure-Object ma alias, można użyć polecenia Get-Alias z parametrem -definition, jak pokazano poniżej:

```
PS C:\> Get-Alias -Definition measure-object
```

| CommandType | Name | ModuleName |
|-------------|---------------------------|------------|
| ----- | ---- | ----- |
| Alias | measure -> Measure-Object | |

Jeśli podoba Ci się alias measure, to możesz go używać i nie definiować własnego. Ale równie dobrze możesz stwierdzić, że oszczędza on tylko dwa naciśnięcia klawisza, ponieważ w konsoli Windows PowerShell istnieje funkcja rozwijania nazw, dzięki której wystarczy wpisać measure-o i nacisnąć klawisz *Tab*. Ogólnie rzecz biorąc, tworząc własne aliasy, często poświęcam czytelność na rzecz wygody. Do moich ulubionych należą aliasy jedno- i dwuliterowe. Jednoliterowe aliasy definiuję dla często używanych poleceń. Ale należy pamiętać, że tak krótkie nazwy są trudne do rozszyfrowania i łatwo zapomnieć, co znaczą. Dlatego najczęściej tworzę aliasy dwuliterowe. Przy użyciu dwóch liter można utworzyć skrót składający się z inicjałów czasownika i rzeczownika, np. mo może być aliasem nazwy polecenia Measure-Object. Czy alias mo nie jest jeszcze używany, można sprawdzić za pomocą polecenia Get-Alias:

```
PS C:\> Get-Alias -Name mo
```

Ile jest dwuliterowych aliasów?

Liczba dwuliterowych kombinacji liter jest dość duża, ale jaka dokładnie? Aby ją obliczyć, należy wziąć każdą literę od *a* do *z* i połączyć ją z drugą literą z tego samego zakresu. Jeśli jesteś dobry z matematyki, to już wiesz, że odpowiedź brzmi 676. Ale jeśli matematyka to nie jest Twój konik, to dla zabawy możesz napisać skrypt Windows PowerShell, który wykona te obliczenia za Ciebie. Problem polega na tym, że nie można użyć operatora zakresu (*..*), który działa tylko z liczbami. Na przykład *1..10* automatycznie tworzy zakres liczb o wartościach od 1 do 10, dzięki czemu można zaoszczędzić sporo pisania. Ale zakres liter można utworzyć za pomocą operatora **zakresu** przy użyciu numerycznych kodów ASCII liter od *a* do *z*. Kod ASCII 97 reprezentuje literę *a*, natomiast kod 122 reprezentuje literę *z*. Po określeniu numerycznego zakresu można za pomocą polecenia *ForEach-Object* przekonwertować wszystkie liczby na litery przy użyciu typu *[char]*. Otrzymaną tablicę liter można zapisać w zmiennej *\$letters*. Potem wystarczy przepuścić tę tablicę przez dwie pętle, aby utworzyć i zapisać dwuliterowe kombinacje w zmiennej *\$lettercombination* typu *[array]*. Do sprawdzenia liczby możliwych kombinacji można użyć polecenia *Measure-Object*. Poniżej znajduje się kod źródłowy skryptu o nazwie *GetTwoLetterAliasCombinations.ps1*, implementującego opisaną technikę:

```
Foreach ($1letter in $letters)
{
    Foreach ($2letter in $letters)
    { [array]$letterCombinations += "$1letter$2letter" } }
"Jest " + ($letterCombinations | Measure-Object).count +
" możliwych kombinacji."
"Oto ich lista: "
$letterCombinations
```

Do utworzenia nowego aliasu można użyć polecenia *New-Alias* lub *Set-Alias*. Można też użyć polecenia *New-Item* i wskazać dysk aliasu. Wadą ostatniej z wymienionych technik jest to, że nie można używać parametru *-description* służącego do definiowania informacji na temat aliasu. Inną wadę metody tworzenia aliasów za pomocą polecenia *New-Item* jest konieczność wpisywania większej ilości tekstu. Dlatego do tworzenia aliasów zawsze używam polecenia *New-Alias* lub *Set-Alias*. Ale które z nich wybrać w konkretnym przypadku? Zanim odpowiem na to pytanie, napiszę, do czego służy każde z tych poleceń. Oczywiście polecenie *New-Alias* tworzy nowy alias. Natomiast *Set-Alias* służy do modyfikowania istniejących aliasów. Jeśli dany alias nie istnieje, tworzy nowy. Zatem wiele osób używa tego polecenia zarówno do tworzenia, jak i modyfikowania aliasów. Wadą tego podejścia jest to, że istnieje niebezpieczeństwo niezamierzonego zmodyfikowania istniejącego aliasu i nawet się o tym nie dowiemy. Ale jeśli tak właśnie chcemy, to użycie polecenia *Set-Alias* będzie doskonałym rozwiązaniem.

Alias lepiej jest tworzyć za pomocą polecenia *New-Alias*. Ma ono parametr *-description* i powiadamia użytkownika, jeśli tworzony przez niego alias już istnieje. Aby zdefiniować opis podczas tworzenia aliasu, należy użyć parametru *-description* w sposób pokazany poniżej:

```
New-Alias -Name mo -Value Measure-Object -Description "Alias Pana Eda"
```

W firmach często definiuje się zestawy aliasów dla całej korporacji, aby zapewnić spójność całego środowiska skryptowego. Administrator sieci pracujący przy wybranym komputerze ma dzięki temu pewność, że może używać danego aliasu. Ponadto praca w takim środowisku jest bardziej przewidywalna. Wpisując tę samą wartość w parametrze `-description` aliasów, ułatwia się tworzenie list wszystkich korporacyjnych aliasów. Wtedy wystarczy tylko przefiltrować listę wszystkich aliasów przez zawartość tego parametru, jak pokazano poniżej:

```
PS C:\> Get-Alias | where description -eq 'alias pana eda'
```

| CommandType | Name | ModuleName |
|-------------|----------------------|------------|
| ----- | ---- | ----- |
| Alias | mo -> Measure-Object | |

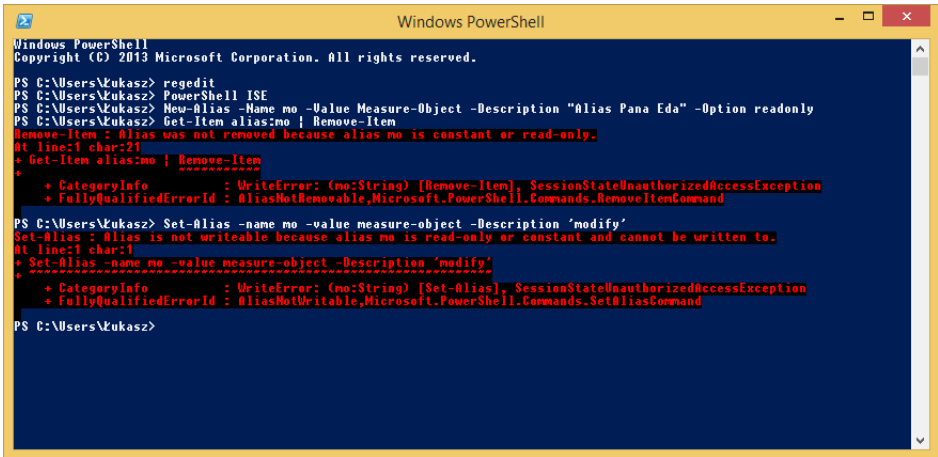
Operator `-eq` w bloku kodu polecenia `Where-Object` sprawia, że filtr nie rozróżnia wielkości liter. Jeśli chcesz włączyć rozróżnianie wielkości liter, użyj parametru `-ceq`. Litera `c` we wszystkich operatorach oznacza rozróżnianie wielkości liter — domyślnie operatory nie rozróżniają wielkości liter. Jak widać poniżej, gdy użyjemy operatora rozróżniającego wielkość liter, polecenie nie zwraca żadnych aliasów.

```
PS C:\> Get-Alias | Where description -ceq 'alias pana eda'
PS C:\>
```

Oprócz parametru `-description` wiele firm używa także parametru `-option` umożliwiającego tworzenie aliasów tylko do odczytu i stałych. Aby utworzyć alias tylko do odczytu, należy parametrowi `-option` przekazać jako wartość słowo kluczowe `read-only`:

```
New-Alias -Name mo -Value Measure-Object -Description "Alias Pana Eda" -Option readonly
```

Zaletą zdefiniowania aliasu tylko do odczytu jest ochrona przed jego przypadkową modyfikacją, jak pokazano na rysunku 5.1.



RYSUNEK 5.1. Próby zmodyfikowania aliasu tylko do odczytu kończą się błędami

Oczywiście w razie potrzeby alias tylko do odczytu można zmodyfikować lub usunąć. Aby zmienić opis, można użyć polecenia `Set-Alias`, w którym należy podać nazwę aliasu i nowy opis oraz dodać parametr `-force`, jak pokazano poniżej:

```
Set-Alias -Name mo -value measure-object -Description "mój alias" -Force
```

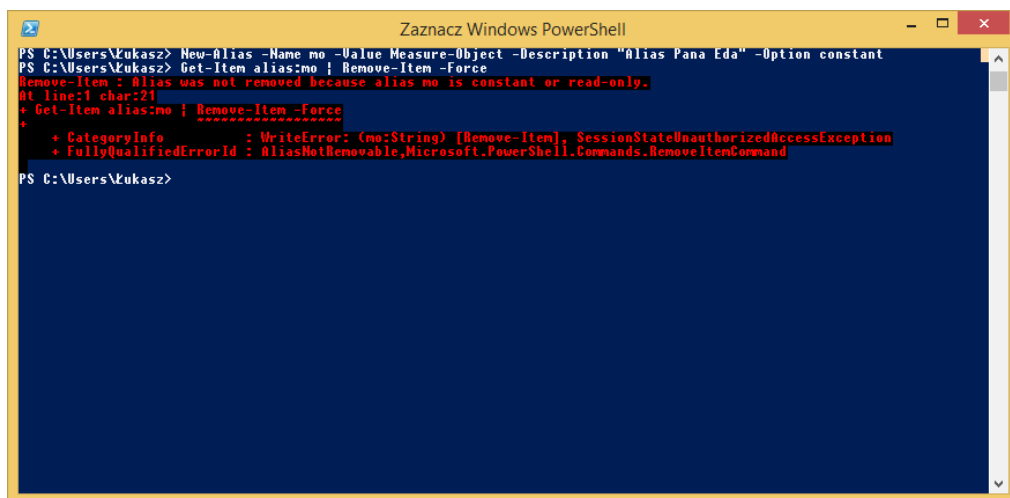
Aby usunąć polecenie tylko do odczytu, należy użyć polecenia `Remove-Item` z parametrem `-force`:

```
Remove-Item Alias:\mo -Force
```

Do tworzenia aliasów stałych służy słowo kluczowe `constant`, które należy przekazać w wartości parametru `-option`:

```
New-Alias -Name mo -Value Measure-Object -Description "Alias Pana Eda" -Option constant
```

Zasadniczo aliasy stałe należy tworzyć tylko wtedy, gdy ma się pewność, że nie trzeba ich będzie modyfikować ani usuwać. Aliasu stałego nie można zmienić ani usunąć — po prostu jest stały. Informacja o błędzie może być trochę myląca, bo jest w niej napisane, że alias może być tylko do odczytu lub stały, i sugeruje użycie parametru `-force`. Problem w tym, że jest ona wyświetlana nawet wtedy, gdy w poleceniu użyje się parametru `-force`. Informację tę widać na rysunku 5.2.



RYSUNEK 5.2. Informacja o błędzie zwrócona w wyniku próby usunięcia aliasu stałego

Tworzenie funkcji

Funkcje dają użytkownikowi prawie nieskończone możliwości dostosowywania środowiska pracy w konsoli Windows PowerShell. Doskonałym miejscem na wprowadzanie takich udoskonaleń jest profil. Powiedzmy na przykład, że wolelibyśmy, aby polecenie `Get-Help` wyświetlało całą treść artykułu. Wiemy jednak, że niektóre artykuły nie zmieszczą się na jednym ekranie, bo są za długie. W takim przypadku można by je wysłać do funkcji `more`, która umożliwia dzielenie treści na strony. Jeśli na przykład szukasz szczegółowych informacji o poleceniu `Get-Process`, możesz użyć poniższego kodu:

```
Get-Help Get-Process -Full | more
```

Nie ma nic złego we wpisywaniu tego polecenia, ale nawet jeśli skorzystamy z funkcji rozwijania nazw klawiszem `Tab`, to i tak musimy wpisać aż 20 znaków. Jeżeli będziesz go często używać,

szybko zacznie Cię to nużyć. Dlatego polecenie to jest doskonałym kandydatem do napisania funkcji. Dobrym zwyczajem jest nadawanie funkcjom nazw składających się z czasownika i rzeczownika, ponieważ użytkownicy konsoli Windows PowerShell są do tego przyzwyczajeni i pozwala im to korzystać z funkcji rozwijania nazw klawiszem *Tab*. Jak widać poniżej, nazwałem funkcję `Get-MoreHelp`.

Get-Morehelp.ps1

```
Function Get-MoreHelp()
{
    Get-Help $args[0] -Full |
    more
} #end Get-MoreHelp
```

Na początku znajduje się słowo kluczowe `Function` oznaczające, że jest to definicja funkcji. Za nim znajduje się nazwa funkcji, którą w tym przypadku jest `Get-MoreHelp`. Pusty nawias za nazwą nie jest obowiązkowy, ponieważ w nawiasie podaje się parametry. Jeśli nie ma parametrów, nawiasu też może nie być. Mimo to zazwyczaj dodaję nawias, aby zaznaczyć, że w razie potrzeby można dodać jakieś parametry, jak pokazano poniżej:

```
Function Get-MoreHelp()
```

Kolejnym elementem definicji funkcji jest otwarcie klamry, które stanowi początek bloku kodu funkcji. Gdy piszę funkcję, zawsze od razu zamykam klamrę w następnym wierszu, aby później o tym nie zapomnieć. Ponadto dobrym zwyczajem jest dodanie za zamknięciem klamry komentarza oznaczającego koniec definicji funkcji. Komentarze takie są też bardzo pomocne przy rozwiązywaniu problemów ze skryptami, ponieważ ułatwiają czytanie kodu i wyraźnie oddzielają kod funkcji od innego kodu. Ponadto jeśli funkcja jest dłuższa niż wysokość ekranu, to końcowy komentarz z przypomnieniem nazwy ułatwia utworzenie aliasu:

```
{
} #end Get-MoreHelp
```

W funkcji `Get-MoreHelp` argument wywołania jest przechowywany w zmiennej automatycznej `$args`. Jako że polecenie `Get-Help` nie przyjmuje w parametrze `name` tablicy, można pobrać pierwszy element tablicy `$args` za pomocą indeksu `[0]`. Jeżeli zgodnie z wymaganiami do funkcji zostanie przekazany tylko jeden element, jest nim zawsze element o numerze zero w tablicy. Funkcja przekazuje parametr `-full` do polecenia `Get-Help`. Otrzymane informacje pomocnicze są przekazywane potokowo, za pomocą symbolu `|`, jak pokazano poniżej:

```
Get-Help $args[0] -full |
```

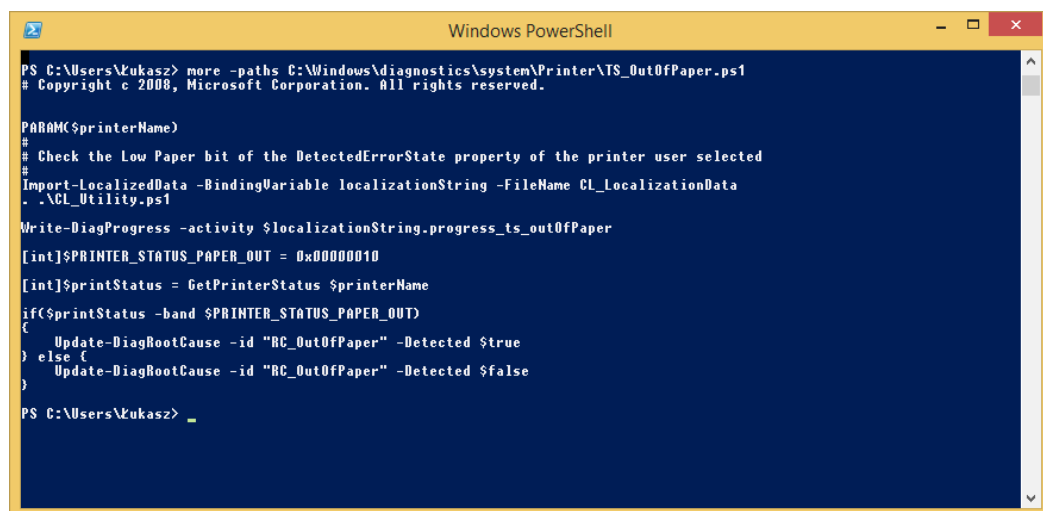
Przesłanie istniejących poleceń

Jako że funkcja `Get-MoreHelp` może zwrócić znacznie więcej tekstu, niż zmieści się na jednym ekranie, przekazuje zwrócone informacje do funkcji `more`. A ponieważ funkcje w konsoli Windows PowerShell są składnikami pierwszej klasy, mają pierwszeństwo przed plikami wykonywalnymi, a nawet macierzystymi poleceniami Windows PowerShell. Dzięki temu za pomocą funkcji można łatwo zmienić zachowanie pliku wykonywalnego lub polecenia `cmdlet`, tworząc funkcję o takiej

samej nazwie, czego dobrym przykładem jest funkcja *more*. Plik *more.com* to plik wykonywalny umożliwiający wyświetlanie informacji po jednej stronie — istnieje już od czasów systemu DOS. Natomiast funkcja *more* modyfikuje działanie tego pliku. Poniżej pokazano jej kod źródłowy:

```
param([string[]]$paths)
if($paths)
{
    foreach ($file in $paths)
    {
        Get-Content $file | more.com
    }
}
else
{
    $input | more.com
}
```

W kodzie tej funkcji widać, że zawiera ona przydatny dodatek wzbogacający funkcjonalność narzędzia *more.com*. Jeśli funkcji tej przekaże się ścieżkę, pobierze ona zawartość pliku i przekaże ją do narzędzia *more.com*, jak widać na rysunku 5.3.



```
Windows PowerShell

PS C:\Users\Lukasz> more -paths C:\Windows\diagnostics\system\Printer\TS_OutOfPaper.ps1
# Copyright c 2008, Microsoft Corporation. All rights reserved.

PARAM($printerName)
# Check the Low Paper bit of the DetectedErrorState property of the printer user selected
#
Import-LocalizedData -BindingVariable localizationString -FileName CL_LocalizationData
- .\CL_Utility.ps1

Write-DiagProgress -activity $localizationString.progress_ts_outOfPaper

[int]$PRINTER_STATUS_PAPER_OUT = 0x00000010

[int]$printStatus = GetPrinterStatus $printerName
if($printStatus -band $PRINTER_STATUS_PAPER_OUT)
{
    Update-DiagRootCause -id "RC_OutOfPaper" -Detected $true
} else {
    Update-DiagRootCause -id "RC_OutOfPaper" -Detected $false
}

PS C:\Users\Lukasz> _
```

RYСУNEK 5.3. Przekazanie ścieżki do funkcji *more* powoduje pobranie treści i przekazanie jej do narzędzia *more.com*

Tworzenie aliasów nazw funkcji

Gdy tworzę funkcje pomocnicze, często od razu definiuję aliasy ich nazw, aby zapewnić sobie do nich szybki dostęp. Funkcję i alias można utworzyć w tym samym skrypcie, ale nie w definicji funkcji. Aliasu nie można utworzyć wewnątrz definicji funkcji, ponieważ w miejscu tym nie jest ona jeszcze utworzona, a przecież nie da się utworzyć aliasu nieistniejącej funkcji. Nie ma natomiast nic złego w utworzeniu aliasu funkcji w tym samym skrypcie, w którym została zdefiniowana, jak widać w skrypcie *Get-MoreHelpWithAlias.ps1*. Co ciekawe, definicja aliasu może znajdować się przed lub za definicją funkcji.

Get-MorehelpWithalias.ps1

```
Function Get-MoreHelp()
{
    Get-Help $args[0] -full |
    more
} #End Get-MoreHelp
New-Alias -name gmh -value Get-MoreHelp -Option allscope
```

Przeglądanie tablicy za pomocą pętli

Jako że zmienna \$args zawiera tablicę, można ją wykorzystać do przekazania dwóch lub większej liczby informacji i pobrania pomocy na kilka tematów. W tym celu można za pomocą instrukcji for przejrzeć elementy tablicy \$args. Instrukcja ta przyjmuje trzy parametry: początek, cel i metodę przeglądania. W tym przykładzie zmienna \$i służy do rejestrowania pozycji w tablicy. Początkowo ustawiamy ją na 0. Natomiast za pomocą operatora -le, mniejszy lub równy, określamy, że pętla ma wykonać tyle cykli, ile jest elementów w tablicy \$args. W miarę postępu działania pętli wartość zmiennej \$i jest zwiększana o jeden w każdym cyklu za pomocą konstrukcji \$i++:

```
For($i = 0 ; $i -le $args.count ; $i++)
```

W wywołaniu polecenia Get-Help potrzebna jest jedna drobna zmiana. Zamiast indeksu \$args[0], który zawsze pobiera pierwszy element tablicy, użyjemy indeksu \$args[\$i]. Wartość zmiennej \$i zmieni się w każdym powtórzeniu, dzięki czemu polecenie Get-Help za każdym razem pobierze kolejny element z tablicy. Poniżej pokazano opisywany zmieniony wiersz kodu:

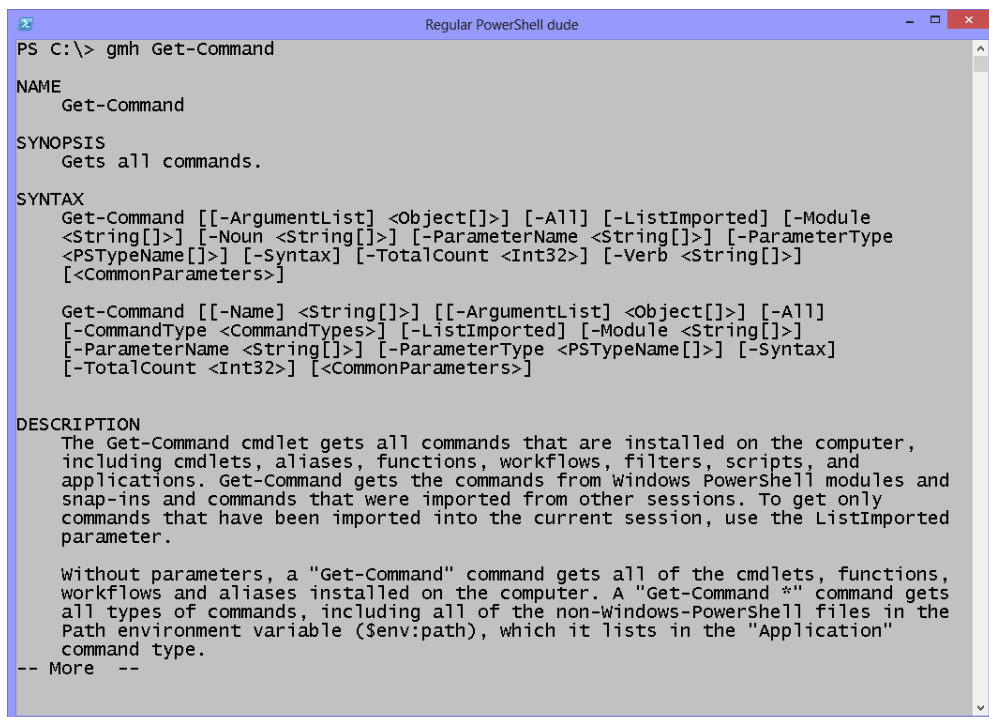
```
Get-Help $args[$i] -full |
```

Pozostała część funkcji Get-MoreHelp jest taka sama jak wcześniej. Poniżej znajduje się jej kod źródłowy z pliku *Get-MoreHelp2.ps1*:

Get-Morehelp2.ps1

```
Function Get-MoreHelp
{
    # .help Get-MoreHelp Get-Command Get-Process
    For($i = 0 ; $i -le $args.count ; $i++)
    {
        Get-Help $args[$i] -full |
        more
    } #end for
} #end Get-MoreHelp
New-Alias -name gmh -value Get-MoreHelp -Option allscope
```

Do uruchamiania funkcji Get-MoreHelp można używać aliasu gmh z nazwą przynajmniej jednego polecenia, o którym chce się przeczytać dodatkowe informacje. Pokazano to na rysunku 4.5, na którym widać kod funkcji wpisany bezpośrednio do konsoli Windows PowerShell.



```

PS C:\> gmh Get-Command

NAME
    Get-Command

SYNOPSIS
    Gets all commands.

SYNTAX
    Get-Command [[-ArgumentList] <Object[]>] [-All] [-ListImported] [-Module
    <String[]>] [-Noun <String[]>] [-ParameterName <String[]>] [-ParameterType
    <PSTypeName[]>] [-Syntax] [-TotalCount <Int32>] [-Verb <String[]>]
    [<CommonParameters>]

    Get-Command [[-Name] <String[]>] [[-ArgumentList] <Object[]>] [-All]
    [-CommandType <CommandTypes>] [-ListImported] [-Module <String[]>]
    [-ParameterName <String[]>] [-ParameterType <PSTypeName[]>] [-Syntax]
    [-TotalCount <Int32>] [<CommonParameters>]

DESCRIPTION
    The Get-Command cmdlet gets all commands that are installed on the computer,
    including cmdlets, aliases, functions, workflows, filters, scripts, and
    applications. Get-Command gets the commands from Windows PowerShell modules and
    snap-ins and commands that were imported from other sessions. To get only
    commands that have been imported into the current session, use the ListImported
    parameter.

    Without parameters, a "Get-Command" command gets all of the cmdlets, functions,
    workflows and aliases installed on the computer. A "Get-Command *" command gets
    all types of commands, including all of the non-Windows-PowerShell files in the
    Path environment variable ($env:path), which it lists in the "Application"
    command type.
    -- More --
  
```

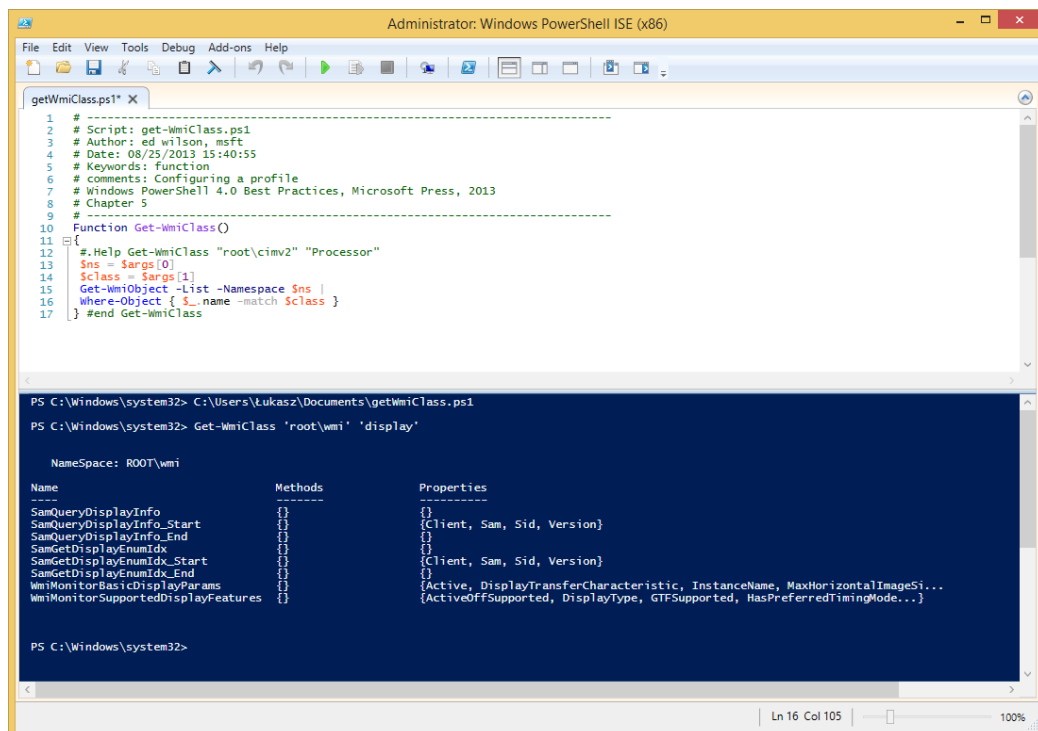
RYСУNEK 5.4. Aby ułatwić sobie pracę, można używać aliasu funkcji

Przekazywanie wielu parametrów

Gdy definiujemy funkcję, często chcemy, żeby przyjmowała pewne parametry wejściowe, które zwiększają jej elastyczność i przydatność. W konsoli Windows PowerShell mamy dwie możliwości do wyboru. Pierwsza metoda przekazywania parametrów polega na użyciu automatycznej zmiennej `$args`, jak pokazano w poprzednim podrozdziale. Druga metoda polega na użyciu parametrów nazwanych. W skryptach parametry nazwane poprzedza się instrukcją `Param`. Natomiast w funkcji instrukcji tej nie należy wpisywać. Po prostu wpisuje się zmienne na każdej pozycji, na której chce się mieć parametr. Nazwa zmiennej jest równocześnie nazwą parametru. W obu metodach przekazywania parametrów są pewne ważne kwestie do zapamiętania. Najpierw bardziej szczegółowo omówimy sobie zmienną `$args`.

Obsługa parametrów za pomocą zmiennej `$args`

Jednym ze sposobów na przekazanie dwóch parametrów jest użycie zmiennej automatycznej `$args`. Gdy do funkcji przekazuje się dwie wartości, można je pobrać z tablicy za pomocą indeksów. W wywołaniu funkcji `Get-WmiClass` przekazywane są dwie wartości. Pierwsza zawiera nazwę przestrzeni nazw WMI, w której należy szukać nazw klas, a druga określa typ szukanej klasy WMI. Funkcja `Get-WmiClass` służy do znajdowania klas WMI. Przykład jej użycia pokazano na rysunku 5.5.



RYSUNEK 5.5. Zmienna \$args obsługuje argumenty pozycyjne

Funkcja `Get-WmiClass` zaczyna pracę od pobrania dwóch wartości ze zmiennej `$args`. Jest to zmienna automatyczna zawierająca wszystko, co użytkownik przekaże do funkcji. Element z pierwszego miejsca zostaje zapisany w zmiennej `$ns`, a drugi element w zmiennej `$class`, jak pokazano poniżej:

```

$ns = $args[0]
$class = $args[1]

```

Polecenie `Get-WmiClass` ma parametr `-list` powodujący utworzenie listy wszystkich klas WMI w przestrzeni nazw. Przestrzeń nazw do przeszukania jest określona jako pierwszy parametr wywołania polecenia. Otrzymana lista klas z danej przestrzeni nazw jest przekazywana do potoku, jak pokazano poniżej:

```
Get-WmiObject -List -Namespace $ns |
```

Aby lista klas była do czegoś przydatna, zostaje przepuszczona przez filtr za pomocą polecenia `Where-Object`. W bloku kodu polecenia `Where-Object` użyto automatycznej zmiennej `$_` w celu odwołania się do bieżącego elementu w potoku. Za pomocą operatora `-match` zdefiniowano wyrażenie regularne do przefiltrowania listy nazw klas. Opisywany wiersz kodu jest pokazany poniżej:

```
Where-Object { $_.name -match $class }
```

Poniżej znajduje się kompletny kod opisywanego skryptu `Get-WmiClass.ps1`:

Get-WmiClass.ps1

```
Function Get-WmiClass()
{
    #.Help Get-WmiClass "root\cimv2" "Processor"
    $ns = $args[0]
    $class = $args[1]
    Get-WmiObject -List -Namespace $ns |
    Where-Object { $_.name -match $class }
} #end Get-WmiClass
```

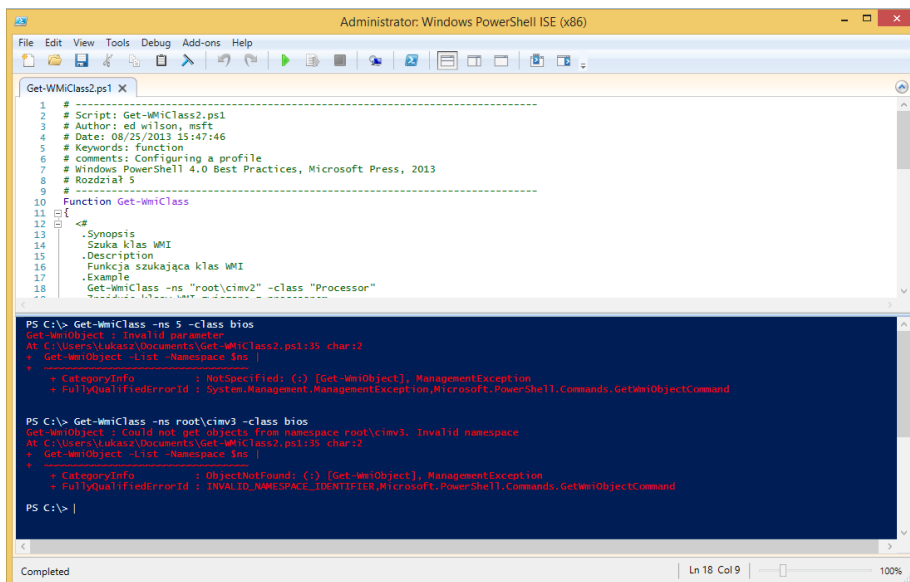
Obsługa parametrów nazwanych

Jeśli do funkcji trzeba przekazać więcej niż dwa parametry, zapamiętanie ich kolejności i znaczenia może być trudne. Ponadto przy użyciu parametrów nazwanych można stosować ograniczenia typów pozwalające wyeliminować podstawowe rodzaje błędów występujące podczas wpisywania wartości w wierszu poleceń.

W skrypcie *Get-WmiClass2.ps1* znajduje się zmieniona funkcja *Get-WmiClass*. Najważniejsza zmiana dotyczy przeniesienia zmiennych *\$ns* i *\$class* do nawiasu za nazwą funkcji. Ponadto, jako że zarówno przestrzeń nazw, jak i nazwa klasy muszą być łańcuchami, dodano ograniczenie typu *[string]*, aby uniemożliwić przekazanie do funkcji nieodpowiedniej wartości, np. liczby. Jako że w zmienionej wersji funkcji użyto parametrów nazwanych, usunięto dwa wiersze kodu pobierające dane ze zmiennej *\$args*. Dzięki temu skrypt *Get-WmiClass2.ps1* jest krótszy i bardziej funkcjonalny niż *Get-WmiClass.ps1*. Poniżej znajduje się pierwszy wiersz funkcji *Get-WmiClass*:

```
Function Get-WmiClass([string]$ns, [string]$class)
```

Na rysunku 5.6 pokazano przykład działania ograniczeń typów.



RYСУNEK 5.6. Ograniczenia typów parametrów funkcji powodują wyświetlenie szczegółowych informacji, gdy zostaną przekazane nieodpowiednie argumenty

W pierwszym wywołaniu funkcji `Get-WmiClass` parametrowi `-ns` przekazano wartość 5, co jest niezgodne z jego ograniczeniem typu `[string]`. W efekcie został zwrócony błąd informujący o użyciu nieprawidłowego parametru.

W drugim przykładzie widocznym na tym rysunku funkcja `Get-WmiClass` została wywołana z parametrem `-ns` o wartości `root/cimv3`. Mimo że w hierarchii WMI nie ma przestrzeni nazw `root/cimv3` (przynajmniej na razie), funkcja została wykonana. Wiadomość o błędzie pochodzi od WMI i informuje, że wystąpił problem z przestrzenią nazw. Definiowanie ograniczeń dotyczących typów parametrów jest bardzo dobrym zwyczajem. Jest to podstawowa warstwa ochrony, którą warto stosować choćby dlatego, że nie sprawia żadnego kłopotu.

W wywołaniu funkcji `Get-WmiClass` można wpisać całą nazwę parametru, skróconą nazwę parametru lub w ogóle można jej nie wpisywać. Poniżej przedstawiono przykłady każdego z tych sposobów wywołania. Gdy podaje się nazwę parametru, wystarczy wpisać tylko taką jego część, która wyklucza pomylenie jej z czymś innym. Należy wziąć to pod uwagę podczas wymyślania nazw parametrów. Jeśli nazwa każdego parametru zaczyna się od innej litery, użytkownik może wpisać tylko pierwszą literę nazwy każdego z nich i zachować podstawową czytelność polecenia. Gdyby na przykład w funkcji `Get-WmiClass` parametr przestrzeni nazw miał nazwę `namespace`, a klasy `name`, konieczne byłoby wpisanie całego słowa **name** dla parametru `-name` i **names** dla parametru `-namespace`. Nazwy parametrów `-Name` i `-namespace` źle się skracają, co widać w poniższym kodzie.

```
Get-WmiClass -ns "root\cimv2" -class "disk"
Get-WmiClass -n root\cimv2 -c disk
Get-WmiClass root\cimv2 disk
```

UWAGA

Wartości nazwanych parametrów funkcji muszą znajdować się w cudzysłowie tylko wtedy, gdy zawierają przecinek, średnik lub inny znak specjalny, który mógłby zostać nieprawidłowo zinterpretowany przez system wykonawczy. Gdy wpisuję polecenia bezpośrednio w wierszu poleceń, często pomijam cudzysłowy, aby mieć mniej pisania. Ale w skryptach dodaję cudzysłowy, aby zwiększyć czytelność i klarowność kodu.

Poniżej znajduje się kompletny kod źródłowy skryptu *Get-WmiClass2.ps1*:

Get-WmiClass2.ps1

```
Function Get-WmiClass
{
    <#
    .Synopsis
    Szuka klas WMI
    .Description
    Funkcja szukająca klas WMI
    .Example
    Get-WmiClass -ns "root\cimv2" -class "Processor"
    Znajduje klasy WMI związane z procesorem
    .Parameter ns
    Przestrzeń nazw
    .Parameter class
    Klasa
    .Notes
    NAME: Get-WmiClass
    AUTHOR: ed wilson, msft
    LASTEDIT: 08/25/2013 15:45:16
```

```

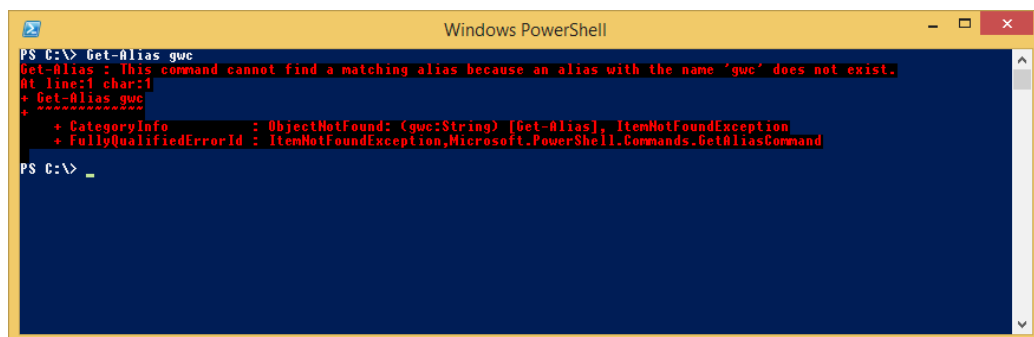
KEYWORDS: WMI, Scripting Technique
HSG:
.Link
Http://www.ScriptingGuys.com
#Requires -Version 2.0
#>
Param ([string]$ns, [string]$class)
Get-WmiObject -List -Namespace $ns |
Where-Object { $_.name -match $class }
} #end Get-WmiClass

```

Definiując funkcję, można też zdefiniować jej alias. Jako że utworzyliśmy już alias dla funkcji `Get-WmiClass`, możemy sprawdzić za pomocą polecenia `Get-Alias`, czy ciąg znaków `gwc` jest już zajęty (pierwsze litery wszystkich składających się na nazwę wyrazów). Do sprawdzenia tej informacji można użyć poniższego polecenia:

```
Get-Alias -Name gwc
```

Jest to jeden z przypadków, w których życzylibyśmy sobie otrzymać informację o błędzie spowodowanym tym, że dany alias jest już zajęty. Błąd ten pokazano na rysunku 5.7.



RYСУNEK 5.7. Informacja o błędzie oznacza, że dany alias nie jest używany

Poniżej przedstawiono kompletny kod źródłowy skryptu *Get-WmiClass2WithAlias.ps1*:

Get-WmiClass2WithAlias.ps1

```

Function Get-WmiClass
{
    <#
    .Synopsis
    Szuka klas WMI
    .Description
    Funkcja szukająca klas WMI
    .Example
    Get-WmiClass -ns "root\cimv2" -class "Processor"
    Znajduje klasy związane z procesorem
    .Parameter ns
    Przestrzeń nazw
    .Parameter class
    Klasa
    .Notes
    NAME: Get-WmiClass

```

```

AUTHOR: ed wilson, msft
LASTEDIT: 08/25/2013 15:45:16
KEYWORDS: WMI, Scripting Technique
HSG:
.Link
Http://www.ScriptingGuys.com
#Requires -Version 2.0
#>
Param ([string]$ns, [string]$class)
Get-WmiObject -List -Namespace $ns |
Where-Object { $_.name -match $class }
} #end Get-WmiClass

New-Alias -Name gwc -Value Get-WmiClass -Description "Mred Alias" '
-Option readonly,allscope

```

Tworzenie zmiennych

Podobnie jak w przypadku aliasów jest kilka sposobów tworzenia zmiennych i przypisywania im wartości. Jedną z nich jest na przykład użycie polecenia `New-Item` na dysku `variable`, jak pokazano poniżej:

```
New-Item -Name temp -Value $env:TEMP -Path variable:
```

Zmienną można też utworzyć za pomocą polecenia `Set-Item`. Jego zaletą jest to, że nie zwraca błędu, gdy zmienna o określonej nazwie już istnieje. Poniżej znajduje się przykład utworzenia zmiennej za pomocą polecenia `Set-Item`. Pamiętaj, że polecenie to nie ma parametru `-name`.

```
Set-Item -Value $env:TEMP -Path variable:\temp
```

Ani polecenie `New-Item`, ani `Set-Item` nie ma parametrów `-option` i `-description`. Jest to ważne w przypadku tworzenia zmiennych, ponieważ nie da się utworzyć stałej ani zmiennej tylko do odczytu bez użycia poleceń `Set-Variable` lub `New-Variable`. Jeśli dana zmienna już istnieje i użyjemy polecenia `Set-Variable`, to wartość tej zmiennej zostanie nadpisana, pod warunkiem że zmienna ta nie jest tylko do odczytu lub stałą. Jeżeli zmienna jest tylko do odczytu, jedynym sposobem na zmienienie jej wartości jest zamknięcie konsoli Windows PowerShell i ponowne jej uruchomienie w celu utworzenia nowej zmiennej.

Można też utworzyć zmienną i jednocześnie przypisać jej wartość. Techniki tej często używa się, gdy wartością zmiennej jest wynik jakichś obliczeń lub łączenia czegoś. W tym przykładzie tworzymy zmienną o nazwie `$wuLog` do przechowywania ścieżki do dziennika usługi aktualizacji systemu Windows, który jest głęboko zaszyty w profilu użytkownika w folderze o nazwie `AppData`. Podczas gdy istnieje zmienna środowiskowa dla lokalnego folderu danych aplikacji, ścieżka do dziennika usługi aktualizacji systemu Windows jest ukryta o kilka poziomów głębiej i kończy się nazwą pliku `WindowsUpdate.log`. Do budowy ścieżek do plików należy używać ścieżkowych poleceń cmdlet, np. `Join-Path`. Pozwala to uniknąć błędów związanych z łączeniem łańcuchów. Przy użyciu zmiennej środowiskowej `$localappdata` i polecenia `Join-Path` z parametrem `-resolve` można zapisać ścieżkę do dziennika usługi aktualizacji systemu Windows na komputerze dowolnego użytkownika. Dokładnie takiego typu zmienną chcemy utworzyć i zapisać w profilu użytkownika konsoli Windows PowerShell. Poniżej znajduje się odpowiednie polecenie:

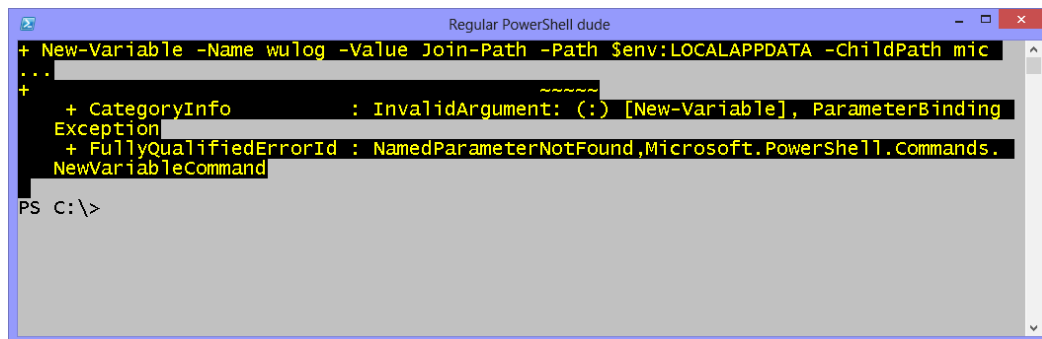
```
PS C:\> $wuLog = Join-Path -Path $env:LOCALAPPDATA '
-ChildPath microsoft\windows\windowsupdate.log -Resolve
PS C:\> $wuLog
C:\Users\edwils.NORTHAMERICA\AppData\Local\microsoft\windows\windowsupdate.log
```

Obliczoną wartość można zapisać w zmiennej nie tylko bezpośrednio. Można też użyć do tego celu polecenia `New-Variable`.

```
PS C:\> New-Variable -Name wuLog -Value (Join-Path -Path $env:LOCALAPPDATA '
-ChildPath microsoft\windows\windowsupdate.log -Resolve)
```

UWAGA

Gdy do zapisania wyniku obliczeń w zmiennej używane jest polecenie `New-Variable`, często należy użyć nawiasu, aby wymusić utworzenie wartości przed próbą przypisania jej do parametru `-value`. W przeciwnym razie może zostać wyświetlone powiadomienie o błędzie polegającym na braku parametru lub podaniu niepoprawnego parametru. Gdy polecenie `New-Variable` znajdzie parametr poza nawiasem, próbuje go zlokalizować. Przykład takiego błędu pokazano na rysunku 5.8.



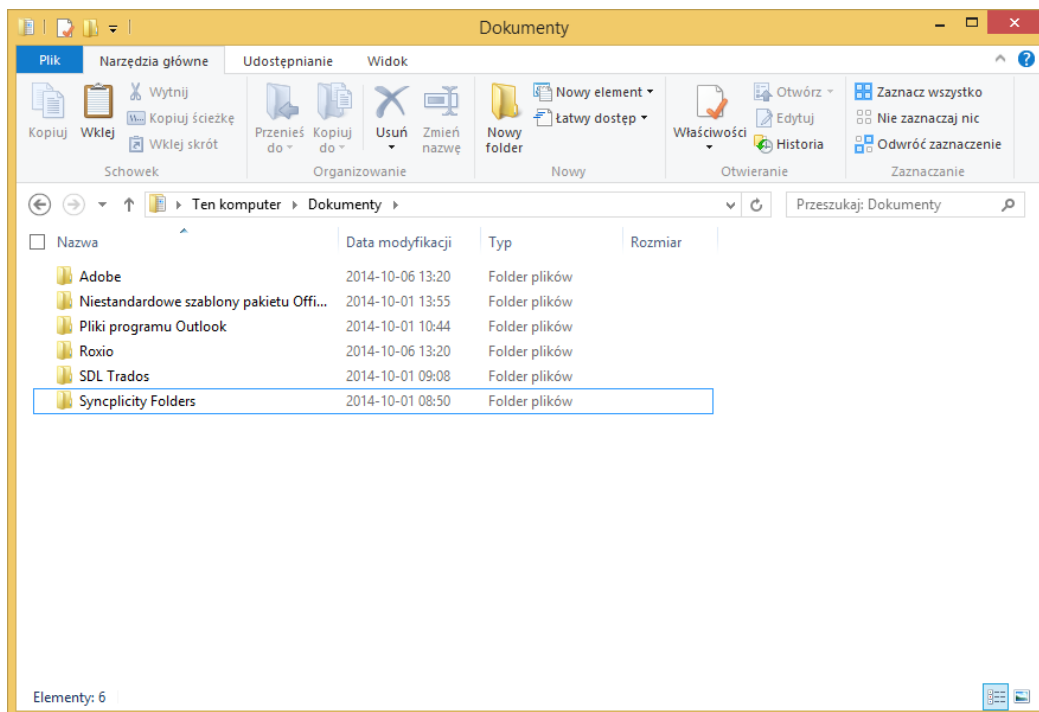
RYСУNEK 5.8. Błąd spowodowany brakiem nawiasu przy tworzeniu nowej zmiennej

Do tworzenia zmiennych dla profilu można też używać zmiennych automatycznych. Wiele aplikacji umieszcza pliki w folderze użytkownika *Dokumenty*. Podczas gdy lokalizacja ta jest dogodna dla aplikacji i użytkowników, którzy mają dostęp do swoich dokumentów poprzez menu startowe, zlokalizowanie tego folderu w wierszu poleceń jest prawie niemożliwe.

Aby ułatwić sobie dostęp do folderu *Dokumenty*, można utworzyć zmienną służącą do odwoływania się do ścieżki. Tworzenie ścieżki do folderu *Dokumenty* jest kolejną dobrą okazją do użycia polecenia `Join-Path`. Istnieje automatyczna zmienna wskazująca katalog główny użytkownika. W moim laptopie katalog główny jest folderem `%username%` w folderze *Users*, jak pokazano poniżej:

```
PS C:\> $home
C:\Users\edwils.NORTHAMERICA
```

Jako że folder *Dokumenty* znajduje się w tym folderze, jak widać na rysunku 5.9, można wykorzystać tę ścieżkę i na jej podstawie zbudować ścieżkę do folderu *Dokumenty*.



RYSUNEK 5.9. Folder Dokumenty jest domyślną lokalizacją do zapisywania plików przez wiele aplikacji

W poleceniu `New-Variable` można zdefiniować parametr `-value`, którego wartość znajduje się w nawiasie, aby wartość polecenia `Join-Path` została obliczona przed przypisaniem jej do zmiennej `docs`. Zmienna ta jest tylko do odczytu, co umożliwia jej modyfikowanie w razie potrzeby, ale zapewnia jej ochronę przed przypadkową zmianą lub usunięciem. Parametr `-description` umożliwia zapanowanie nad wszystkimi tworzonymi zmiennymi, jak pokazano poniżej:

```
New-Variable -Name docs -Value (Join-Path -Path $home -ChildPath documents) `
-Option readonly -Description "Zmienna Pana Eda"
```

WAŻNE

Gdy zaczynałem obsługę konsoli Windows PowerShell, często miałem problemy z używaniem poleceń `New-Variable`, `Set-Variable` oraz `Remove-Variable`. Powodem tych problemów było to, że w wierszu poleceń przed nazwą zmiennej powinien znajdować się znak dolara, natomiast w parametrze `-name` znaku tego być nie powinno.

Przy użyciu obiektu `WshShell` z języka VBScript można pobrać ścieżkę do folderu *Ulubione* (*Favorites*). Jako że konsola Windows PowerShell zapewnia łatwy dostęp do obiektów COM (ang. Component Object Model), nie ma powodu, aby unikać ich używania. Jednym ze sposobów na użycie wspomnianego obiektu jest utworzenie i użycie go w jednym wierszu kodu, jak pokazano poniżej:

```
$f = (New-Object -ComObject Wscript.Shell).specialFolders.item("Favorites")
```


Przedstawiona składnia ma przynajmniej dwie wady z punktu widzenia najlepszych praktyk. Pierwsza, bardziej oczywista, jest taka, że kod ten jest mało czytelny. Wprawdzie często spotyka się takie formy zapisu i większości programistów one nie przeszkadzają, ale uważam, że powinien zwyciężyć zdrowy rozsądek. Lepiej podzielić ten kod na dwie linijki, jak pokazano poniżej:

```
$wshShell = New-Object -ComObject Wscript.Shell
$f = $wshShell.SpecialFolders.Item("Favorites")
```

Dodatkową zaletą podziału długiego polecenia na dwie linijki jest to, że teraz mamy do dyspozycji cały obiekt `WshShell`, który zawiera wiele przydatnych własności i metod. Na przykład oprócz ścieżki do specjalnego folderu *Dokumenty* obiekt `WshSpecialFolders` (zwracany przez wysłanie zapytania `SpecialFolders` do obiektu `WshShell`) daje dostęp także do następujących folderów:

- *AllUsersDesktop*,
- *AllUsersStartMenu*,
- *AllUsersPrograms*,
- *AllUsersStartup*,
- *Ulubione (Favorites)*,
- *Czcionki (Fonts)*,
- *NetHood*,
- *PrintHood*,
- *Programy (Programs)*,
- *Niedawno używane elementy (Recent)*,
- *SendTo*,
- *Menu Start (StartMenu)*,
- *Autostart (Startup)*,
- *Templates*.

Ścieżki wszystkich tych folderów można sprawdzić bez tworzenia zmiennej **pośredniej**. Jeśli obiekt `$wshShell` jest utworzony w profilu, wartości z własności `SystemFolders` są zawsze dostępne do użycia w środowisku skryptowym lub w wierszu poleceń.

```
$wshShell.SpecialFolders.Item("Startup")
```

Ponadto obiekt `WshShell` zawiera jeszcze wiele innych bardzo przydatnych własności i metod. W tabeli 5.1 znajduje się zestawienie składowych tego obiektu.

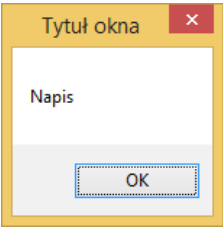
TABELA 5.1. Składowe obiektu `WshShell`

| Nazwa | Typ składowej | Definicja |
|---|---------------|---|
| <code>AppActivate</code> | Metoda | <code>bool AppActivate (Variant, Variant)</code> |
| <code>CreateShortcut</code> | Metoda | <code>IDispatch CreateShortcut (string)</code> |
| <code>Exec</code> | Metoda | <code>IWshExec Exec (string)</code> |
| <code>ExpandEnvironment ↪Strings</code> | Metoda | <code>string ExpandEnvironmentStrings (string)</code> |

TABELA 5.1. Składowe obiektu WshShell — ciąg dalszy

| Nazwa | Typ składowej | Definicja |
|------------------|-----------------------------|---|
| LogEvent | Metoda | bool LogEvent (Variant, string, string) |
| Popup | Metoda | int Popup (string, Variant, Variant, Variant) |
| RegDelete | Metoda | void RegDelete (string) |
| RegRead | Metoda | Variant RegRead (string) |
| RegWrite | Metoda | void RegWrite (string, Variant, Variant) |
| Run | Metoda | int Run (string, Variant, Variant) |
| SendKeys | Metoda | void SendKeys (string, Variant) |
| Environment | Własność parametryzowana | IWshEnvironment Environment (Variant) {get} |
| CurrentDirectory | Własność | string CurrentDirectory () {get} {set} |
| SpecialFolders | Własność | IWshCollection SpecialFolders () {get} |

Bardzo przydatna, a zarazem łatwa w użyciu jest metoda popup. Jak widać na rysunku 5.10, służy ona do wyświetlania okien dialogowych.



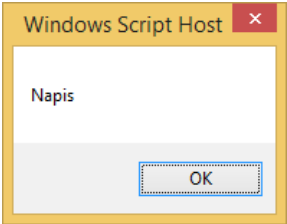
RYSUNEK 5.10. Okno dialogowe z napisem z obiektu WshShell

Aby wyświetlić okno dialogowe za pomocą metody popup, wystarczy w jej wywołaniu przekazać tylko pierwszą wartość z wymienionych w sygnaturze. Wartość ta zostanie wyświetlona jako napis w oknie dialogowym. Druga wartość jest liczbą określającą, ile czasu okno ma być widoczne. Trzecia wartość służy do definiowania tytułu okna. Natomiast ostatnia służy do konfiguracji zestawu przycisków. Jeśli zdefiniuje się tylko pierwszą wartość, zostanie wyświetlone okno dialogowe z przyciskiem OK zawierające wpisany tekst i będzie ono widoczne, aż użytkownik kliknie ten przycisk lub krzyżyk w prawym górnym rogu. W tabeli 5.2 pokazano sygnaturę metody popup.

TABELA 5.2. Sygnatura metody popup obiektu WshShell

| Return | Object.Method | Text | SecondsToWait | Title | Type |
|---------------|------------------|---------|---------------|---------|------|
| \$returnValue | \$wshShell.Popup | "Napis" | 5 | "tytuł" | 0 |

Tytuł okna zostanie wyświetlony w miejscu napisu „Windows Script Host” widocznego na rysunku 5.11.



RYSUNEK 5.11. Domyślnie wyskakujące okienko ma w tytule napis „Windows Script Host”

Kod użyty do utworzenia okienka widocznego na rysunku 5.11 znajduje się poniżej:

```
(New-Object -ComObject wscript.shell).popup("message")
```

Jedną z najbardziej przydatnych cech metody `WshShell.popup` jest możliwość tworzenia różnych konfiguracji przycisków, za pomocą których można współpracować z użytkownikiem poprzez interfejs graficzny. Aby utworzyć wyskakujące okienko z przyciskami *Przerwij*, *Ponów próbę* i *Ignoruj*, należy jako czwarty parametr wpisać wartość 2. W tabeli 5.3 znajduje się zestawienie typowych konfiguracji przycisków. Aby wyświetlić okno dialogowe, które zniknie dopiero po kliknięciu przez użytkownika jednego z przycisków, można wpisać 0 jako wartość drugiego parametru (argumentu określającego czas wyświetlania), jak pokazano poniżej:

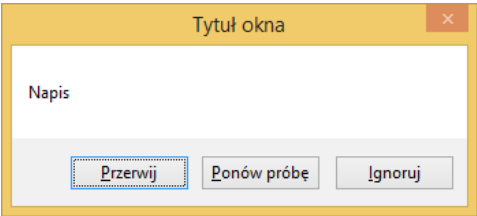
```
$wshShell.Popup("Napis",0,"Tytuł okna",2)
```

TABELA 5.3. Wartości konfiguracji przycisków metody `popup` obiektu `WshShell` i ich znaczenie

| Wartość | Opis |
|---------|---|
| 0 | Przycisk OK |
| 1 | Przyciski OK i Anuluj |
| 2 | Przyciski Przerwij, Ponów próbę i Ignoruj |
| 3 | Przyciski Tak, Nie i Anuluj |
| 4 | Przyciski Tak i Nie |
| 5 | Przyciski Ponów próbę i Anuluj |

Oczywiście różne konfiguracje przycisków są po to, by ułatwić użytkownikowi interakcję ze skryptem. Natomiast od strony skryptu konieczne jest pobranie kodu zwrotnego będącego wartością przypisaną każdemu z przycisków. Poniższy kod tworzy wyskakujące okienko pokazane na rysunku 5.12. Aby dowiedzieć się, który z przycisków kliknął użytkownik, należy pobrać wartość zwrótną. W tabeli 5.4 znajduje się zestawienie wartości zwrótnych poszczególnych przycisków. W tym przykładzie został kliknięty przycisk *Ponów próbę*, co spowodowało zapisanie wartości 4 w zmiennej `$return`.

```
PS C:\> $rtn = (New-Object -ComObject wscript.shell).popup("Napis",0,"Tytuł okna",2)
PS C:\> $rtn
```



RYSUNEK 5.12. Okno dialogowe z przyciskami Przerwij, Ponów próbę i Ignoruj

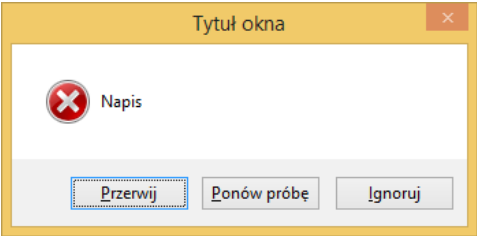
TABELA 5.4. Wartości zwrotne metody popup obiektu WshShell

| Wartość | Opis |
|---------|-----------------------------|
| 1 | Przycisk OK |
| 2 | Przycisk <i>Anuluj</i> |
| 3 | Przycisk <i>Przerwij</i> |
| 4 | Przycisk <i>Ponów próbę</i> |
| 5 | Przycisk <i>Ignoruj</i> |
| 6 | Przycisk <i>Tak</i> |
| 7 | Przycisk <i>Nie</i> |

W wyskakującym okienku można też wyświetlić różne ikony, niezależnie od wybranej konfiguracji przycisków. Jak widać w tabeli 5.5, wartości reprezentujące ikony mają niewiele wspólnego z rzeczywistością. Ponadto dziwi mnie trochę to, że wartości te sumuje się z wartościami przycisków wymienionymi w tabeli 5.3. Aby więc wyświetlić czerwoną ikonę z krzyżem wraz z przyciskami *Przerwij*, *Ponów próbę* i *Ignoruj*, należy do wartości 16 reprezentującej tę ikonę dodać wartość 2 reprezentującą ten zestaw przycisków, jak widać w poniższym przykładzie:

```
$rtn = (New-Object -ComObject wscript.shell).popup("Napis",0,"Tytuł okna",18)
```

Powyższy wiersz kodu powoduje wyświetlenie okna dialogowego widocznego na rysunku 5.13.



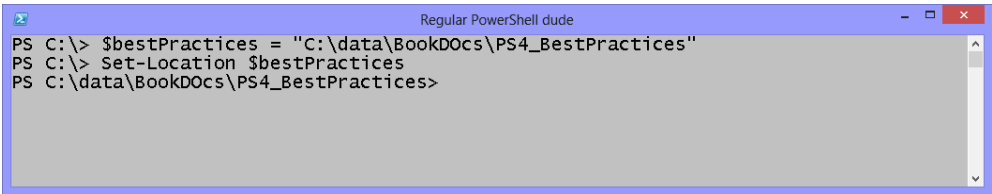
RYSUNEK 5.13. Dodanie wartości reprezentujących ikony do wartości reprezentujących konfigurację przycisków powoduje wyświetlenie różnych rodzajów ikon

TABELA 5.5. Wartości reprezentujące ikony w metodzie popup obiektu WshShell

| Wartość | Opis |
|---------|-----------------------|
| 16 | Ikona symbolu stop |
| 32 | Ikona znaku zapytania |
| 48 | Ikona wykrzyknika |
| 64 | Ikona informacji |

Tworzenie dysków PowerShell

Rozsądne tworzenie dysków Windows PowerShell może uprościć i usprawnić obsługę i modyfikowanie danych w wierszu poleceń. Wprawdzie długą ścieżkę do folderu można zapisać w zmiennej, a następnie zmienić lokalizację roboczą na ścieżkę zapisaną w tej zmiennej, ale wówczas ścieżka ta zajmie sporą część wiersza poleceń, jak widać na rysunku 5.14. Może nie przeszkadza to tak bardzo, ale jeśli dłuższe polecenia zostaną podzielone na kilka wierszy, to łatwo będzie popełnić w nich błąd.



RYSUNEK 5.14. Długie ścieżki często zajmują dużą część wiersza poleceń

Jedną z korzyści wynikających z używania dysków Windows PowerShell jest możliwość ustawienia na katalog główny dowolnej lokalizacji obsługiwanej przez dostawcę PowerShell. Do tworzenia dysków PowerShell służy polecenie `New-PSDrive`. W jego wywołaniu należy podać nazwę dysku oraz określić dostawcę i katalog główny. Poniżej znajduje się przykładowe polecenie tworzące dysk Windows PowerShell z katalogiem głównym w folderze `C:\Data\BookDOcs\PowerShellBestPractices`:

```
New-PSDrive -Name bp -PSProvider filesystem -Root '
C:\data\BookDOcs\PS4_BestPractices -Description "Dysk Pana Eda"
```

Po utworzeniu dysku można zmienić folder roboczy na utworzony dysk za pomocą polecenia `Set-Location`. Dzięki temu odzyskasz znaczną część miejsca w wierszu poleceń, jak widać na rysunku 5.15.

Ponadto dobrym zwyczajem przy tworzeniu dysku jest zdefiniowanie jego opisu za pomocą parametru `description`. Jeśli ustawi się taki sam opis dla wszystkich tworzonych dysków Windows PowerShell, to będzie można łatwo znaleźć wszystkie dyski dostępne w bieżącym środowisku PowerShell. Przykładowe polecenie znajdujące wszystkie dyski w ten sposób pokazano poniżej:

```
Get-PSDrive | Where-Object { $_.description -eq 'Dysk Pana Eda' }
```

```

PS C:\> New-PSDrive -Name BP -PSProvider FileSystem -Root C:\data\BookDOcs\PS4_BestPractices -Description 'mred drive'

Name                Used (GB)    Free (GB) Provider      Root
-----
BP                  92.24      907.76  FileSystem    C:\data\BookDOcs\PS4_BestPra...

PS C:\> Set-Location -Path bp:
PS BP:\> Get-Childitem

Directory: C:\data\BookDOcs\PS4_BestPractices

Mode                LastWriteTime         Length Name
----
d-----      8/24/2013 10:46 PM             <DIR> ADMINstuff
d-----      8/25/2013  3:55 PM             <DIR> Chapters
d-----      8/24/2013 12:59 PM             <DIR> Paging Review
d-----      8/25/2013 11:58 AM             <DIR> Review
d-----      8/24/2013  1:50 PM             <DIR> ScreenShots
d-----      8/24/2013  1:49 PM             <DIR> Scripts

PS BP:\>

```

RYSUNEK 5.15. Utworzenie dysku Windows PowerShell jest dobrym sposobem na odzyskanie miejsca w wierszu poleceń

Dzięki utworzeniu dysków Windows PowerShell dla większości ważnych lokalizacji z danymi można z łatwością zmieniać katalog roboczy za pomocą polecenia `Set-Location`. Jeśli utworzy się tylko jeden dysk Windows PowerShell jako centrum wszystkich działań na danych, to można za pomocą polecenia `Set-Location` zmienić katalog roboczy na własny dysk PowerShell w profilu.

Zapiski praktyka

Praca z profilami

Hal Rottenberg, MVP Microsoft PowerShell

W moim profilu znajduje się wiele różnych rzeczy. Najpierw ładuję moduł Windows Powershell Community Extensions (PSCX). Zawarte w nim polecenia umożliwiają wykonywanie wielu dodatkowych zadań i znacznie ułatwiają pracę z konsolą Windows PowerShell. Jeśli jeszcze nie znasz tego modułu, koniecznie zajrzyj na stronę www.codeplex.com/PowerShellCX.

Następnie tworzę własną funkcję monitorującą i dodaję wiele różnych przystawek. Tworzę kilka aliasów oraz dodaję ścieżki do zmiennej `$env:path`. W ogóle już nie używam narzędzia `cmd.exe`. Dzięki temu profil jest w zasadzie podstawową lokalizacją, w której obsługuję `%path%`. W części dotyczącej tworzenia ścieżek ustawiam też parę zmiennych środowiskowych, np. `$MaximumHistoryCount`. Zmienna ta określa rozmiar bufora historii poleceń, który domyślnie przechowuje 64 ostatnio wpisane polecenia.

Najlepsze w moim profilu jest to, że dołączam w nim wiele funkcji, które napisałem przez kilka ostatnich lat. Są to krótkie, nadające się do wielokrotnego użytku fragmenty kodu, które bardzo ułatwiają mi pracę. Ponadto tworzę też kilka dysków PowerShell. Jeden jest szczególnie przydatny.

Nazywa się *scripts* i wskazuje na lokalizację \$(split-path \$profile)\scripts, w której przechowuję wszystkie swoje biblioteki funkcji oraz samodzielne pliki *.ps1*. Ten folder również znajduje się na mojej ścieżce.

Ponadto mam w profilu sekcję, w której ładuję wiele zestawów .NET. Używam ich nieczęsto i aktualnie nawet wyłączyłem je za pomocą komentarza. Niemniej jednak dla wielu osób możliwość ładowania zestawów .NET może być bardzo przydatna. Na przykład kiedyś często używałem zestawu .NET zawierającego narzędzie do obsługi znaczników ID3 (do plików MP3) i zestawu obsługującego komunikację błyskawiczną poprzez protokół Jabber/Extensible Messaging and Presence Protocol (XMPP).

W ostatniej sekcji profilu ładuję własne typy.

Używam usługi Microsoft SkyDrive, aby zawsze mieć dostęp do profilu. Ponadto w usłudze tej przechowuję kopię zapasową całego swojego folderu Windows PowerShell (i wspomnianych skryptów). Do tego samego celu można też używać innych narzędzi, takich jak Syncplicity (chmura), Foldershare (chmura) czy Synctoy (lokalna).

Ale najlepsze w moim profilu jest to, że dodałem w nim własność ScriptProperty do obiektu System.IO.FileInfo za pomocą systemu ETS (ang. *Extended Type System*) i plików XML *.ps1* w konsoli Windows PowerShell. Własność ta ma nazwę Pages i jest skryptem wywoływanym przy każdej próbie dostępu do tej własności. W skrypcie tym wykorzystałem mało znaną cechę obiektu COM Shell.Application w celu pobierania liczby stron z dokumentów programu Word z pakietu Microsoft Office. Dzięki temu skryptowi mogę tworzyć następujące polecenia:

```
dir | ft name, length, pages
```

a nawet takie:

```
dir | Measure-Object -sum pages
```

Oto kod z mojego profilu, który ładuje ten typ (lub typy, gdybym dodał więcej).

```
Get-ChildItem -path $profiledir\ps1xml\*.ps1xml |
ForEach-Object {
    Update-TypeData $_
    write-host "Updating type data: '$($_.name)'"
}
```

A oto zawartość pliku XML *.ps1*:

```
<?xml version="1.0" encoding="utf-8" ?>
<Types>
  <Type>
    <Name>System.IO.FileInfo</Name>
    <Members>
      <ScriptProperty>
        <Name>Pages</Name>
        <GetScriptBlock>
          $shellApp = new-object -com shell.application
          $myFolder = $shellApp.Namespace($this.Directory.FullName)
          $fileobj = $myFolder.Items().Item($this.Name)
          "$($myFolder.GetDetailsOf($fileobj,13))"
        </GetScriptBlock>
      </ScriptProperty>
    </Members>
  </Type>
</Types>
```

Nieczęsto korzystam z tej funkcjonalności. Najbardziej przydatną rzeczą w moim profilu jest kod dołączający moje biblioteki funkcji. Zwróć uwagę na użycie dysku Windows PowerShell scripts i konwencję nazywania plików lib-nazwa pliku. Kod ten bardzo ułatwia mi ładowanie plików bibliotek bez grzebania w profilu, jak widać w poniższym kodzie:

```
Get-ChildItem scripts:\lib-*.ps1 |
ForEach-Object {
    . $_
    write-host "Loading library file: '$($_.name)'"
}
```

Nie przejmuję się, gdy nie mam ze sobą mojego profilu, ponieważ wierzę w niezawodność technologii chmurowych. Dzięki usłudze SkyDrive zawsze mam dostęp do swojego profilu. Jako że moim podstawowym komputerem jest laptop, kwestie profilu mnie nie martwią.

Mimo że w profilu definiuję kilka aliasów, nigdy nie używam ich w skryptach — jedynie w wierszu poleceń. Podstawowym narzędziem umożliwiającym wygodne tworzenie czytelnych skryptów jest dobry edytor z funkcją uzupełniania nazw poleceń, parametrów, nazw plików, a nawet argumentów.

W skryptach z reguły implementuję podstawowy mechanizm sprawdzania błędów typu: „Jeśli brakuje parametru *x*, zgłoś błąd”. Choć w skryptach, które udostępniam do użytku także innym, zazwyczaj staram się trochę bardziej.

Dla początkujących programistów skryptów mam następującą radę: pobierz i zainstaluj moduł PSCX oraz użyj domyślnego profilu. Ja tak zrobiłem, gdy zaczynałem naukę posługiwania się konsolą Windows PowerShell. Moduł jest bardzo dobrze zbudowany i może być źródłem cennych inspiracji. Poniżej znajduje się zawartość mojego osobistego profilu:

```
# comments: $profiledir, Add-PathVariable come with PSCX

$ErrorPreference = "silentlycontinue"

# -----
# Load PSCX
# -----
. "$home\My Documents\WindowsPowerShell\PSCX_Profile.ps1"

# -----
# Load SQL PSX
# -----
. "$home\My Documents\WindowsPowerShell\Scripts\SQLPSX\LibrarySmo.ps1"

# -----
# Set prompt
# -----

. $profiledir\prompt.ps1

# -----
# Add third-party snapins
# -----

$Snapins =
    # "psmsi", # Windows Installer PowerShell Extensions
    "PshX-SAPIEN", # AD cmdlets from Sapien
```



```

# "GetGPOObjectPSSnapIn", # GPO management
"Quest.ActiveRoles.ADMangement", # more AD stuff
# "Microsoft.Office.OneNote",
"PowerGadgets",
"VMware.VimAutomation.Core",
# "PoshXmpp",
# "PSMobile",
#"PoshHttp",
"NetCmdlets",
"OpenXml.PowerTools",
"IronCowPosh"
$snaps | ForEach-Object {
    if ( Get-PSSnapin -Registered $_ -ErrorAction SilentlyContinue ) {
        Add-PSSnapin $_
    }
}

# -----
# Aliases
# -----
set-alias grep select-string
set-alias ns1 resolve-host
Set-Alias rsps Restart-PowerShell
set-alias which get-command
Set-Alias cvi Connect-VIServer

# -----
# V2 modules
# -----
# dir $profiledir\modules\*.psm1 | Add-Module

# -----
# Setup environment
# -----

New-PSDrive -Name Scripts -PSProvider FileSystem -Root '
    $profiledir\scripts
Add-PathVariable Path $profiledir\scripts
Add-PathVariable Path $profiledir
Add-PathVariable Path "C:\Program Files\OpenSSL\bin"
Add-PathVariable Path "C:\Program Files\Reflector"
$MaximumHistoryCount = 4KB

# -----
# Load function / filter definition library
# -----

Get-ChildItem scripts:\lib-*.ps1 | % {
    . $_
    write-host "Loading library file:'t$($_.name)"
}
write-host

# -----
# PS Drives
# -----

```

```
New-PSDrive -Name Book -PSProvider FileSystem -Root 'C:\Documents and Settings\hrottenberg\
My Documents\MVP-TFM'
Write-Host

# -----
# Load .NET assemblies
# -----
#Get-ChildItem $profiledir\Assemblies\*.dll | % {
# [void][reflection.assembly]::LoadFrom( $_.FullName )
# write-host "Loading .NET assembly: '$($_.name)'"
#}
#Write-Host

# -----
# Load custom types
# -----

Get-ChildItem $profiledir\pslxml\*.pslxml | % {
    Update-TypeData $_
    write-host "Updating type data: '$($_.name)'"
}
Write-Host
if ($?) { Write-Host 'There were errors loading your profile. Check the $error object for
details.' }
```

Włączanie obsługi skryptów

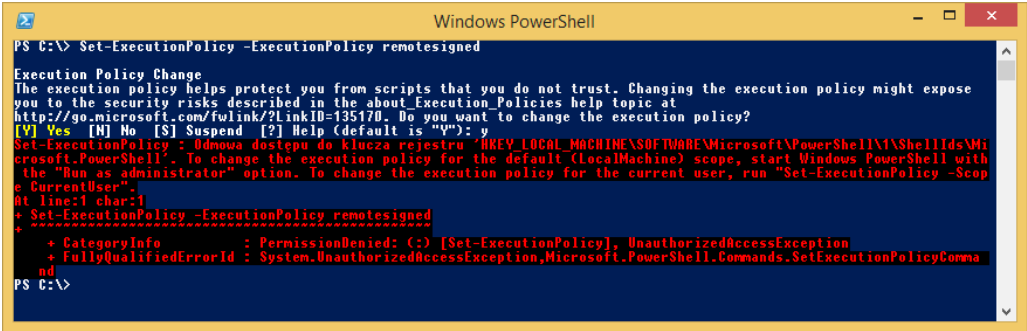
Bezpośrednio po zainstalowaniu konsoli Windows PowerShell ustawione są ograniczone zasady wykonywania skryptów. Nie można wtedy wykonywać żadnych skryptów. Jako że profil jest plikiem *.ps1*, również jest skryptem i domyślnie nie zostanie uruchomiony. W konsoli Windows PowerShell wyróżnia się pięć poziomów zasad ograniczeń wykonawczych, które można ustawiać za pomocą polecenia *Set-ExecutionPolicy*. Wymieniono je w tabeli 5.6. Ograniczoną zasadę wykonawczą można skonfigurować w zasadach grupy przy użyciu ustawienia *Turn On Script Execution* (włącz wykonywanie skryptów) w Active Directory. Ustawienie to można zastosować do obiektu komputera lub użytkownika. Ustawienie obiektu komputera jest ważniejsze od innych ustawień.

TABELA 5.6. Ustawienia poziomu zasady wykonywania

| Poziom | Opis |
|--------------|--|
| Restricted | Brak możliwości wykonywania skryptów i plików konfiguracyjnych |
| AllSigned | Wszystkie skrypty i pliki konfiguracyjne muszą być podpisane przez zaufanego wydawcę |
| RemoteSigned | Wszystkie skrypty i pliki konfiguracyjne pobrane z internetu muszą być podpisane przez zaufanego wydawcę |
| Unrestricted | Można wykonywać wszystkie skrypty i pliki konfiguracyjne. Przed wykonaniem skryptów pobranych z internetu wyświetlane jest pytanie, czy na pewno użytkownik chce to zrobić |
| Bypass | Nic nie jest blokowane i nie są wyświetlane żadne ostrzeżenia ani pytania |

Preferencje użytkownika dotyczące ograniczonej zasady wykonywania programu Windows PowerShell można konfigurować za pomocą polecenia `Set-ExecutionPolicy`, ale ustawienia te nie zmieniają ustawień zasad grupy. Poniżej znajduje się przykład zmieniania bieżącej zasady wykonywania na `RemoteSigned`. Aby można było wykonać polecenie `Set-ExecutionPolicy`, konsola Windows PowerShell musi mieć uprawnienia administratora. Należy więc uruchomić ją, klikając prawym przyciskiem skrót do jej pliku wykonywalnego, a następnie z menu podręcznego wybrać opcję *Uruchom jako administrator*. Różne sposoby rozwiązywania kwestii dotyczących bezpieczeństwa zostały opisane w rozdziale 4. Jeśli spróbujesz wykonać polecenie `Set-ExecutionPolicy` tylko jako administrator komputera lub użytkownik należący do grupy administratorów, to w systemie Windows Vista i nowszych ujrzysz informację o błędzie widoczną na rysunku 5.16.

`Set-ExecutionPolicy -ExecutionPolicy remotesigned`



RYСУNEK 5.16. Próba zmiany ograniczonej zasady wykonywania programu Windows PowerShell, który nie jest uruchomiony jako administrator, powoduje błąd

Zestaw ustawień ograniczonej zasady wykonywania można sprawdzić za pomocą polecenia `Get-ExecutionPolicy`.

Oprócz pięciu wymienionych zasad wykonywania dodatkowo można skonfigurować zakres obowiązywania zasady. Zakres ten dotyczy sposobu zastosowania zasady i można go ustawić na jeden z trzech rodzajów: `Process`, `CurrentUser` oraz `LocalMachine`. W tabeli 5.7 znajduje się opis tych wartości.

TABELA 5.7. Ustawienia zakresu zasady wykonywania

| Zakres | Opis |
|--------------|--|
| Process | Zasada wykonywania dotyczy tylko bieżącego procesu programu Windows PowerShell |
| CurrentUser | Zasada wykonywania dotyczy tylko bieżącego użytkownika |
| LocalMachine | Zasada wykonywania dotyczy wszystkich użytkowników komputera |

Tworzenie profilu

Bezpośrednio po zainstalowaniu programu Windows PowerShell w komputerze nie ma żadnego profilu. Z jednej strony profil można traktować jak rodzaj pliku *Autoexec.bat*, którego używano kiedyś. Z drugiej strony *Autoexec.bat* to tylko plik wsadowy wykonujący jedynie serie poleceń. Mimo to plik ten umieszcza się w katalogu głównym i ma on nazwę *Autoexec.bat*, dzięki czemu jest o wiele ważniejszy niż zwykły plik wsadowy, ponieważ zawiera polecenia konfigurujące wszystkie rodzaje czynności, z konfiguracją środowiska i uruchamianiem samego systemu Windows włącznie. Profil programu Windows PowerShell nie uruchamia konsoli Windows PowerShell. Jest to tylko skrypt o specjalnej nazwie i zapisany w specjalnym miejscu — chociaż tak naprawdę ma dwie specjalne nazwy i jest zapisany w czterech specjalnych miejscach. Tak, to prawda. W rzeczywistości istnieją cztery profile Windows PowerShell. Ich wykaz znajduje się w tabeli 5.8.

TABELA 5.8. Profile Windows PowerShell i ich lokalizacje

| Profil | Lokalizacja |
|------------------------|--|
| AllUsersAllHosts | C:\Windows\system32\WindowsPowerShell\v1.0\profile.ps1 |
| AllUsersCurrentHost | C:\Windows\system32\WindowsPowerShell\v1.0\Microsoft.PowerShell_profile.ps1 |
| CurrentUserAllHosts | C:\Users\UserName\Documents\WindowsPowerShell\profile.ps1 |
| CurrentUserCurrentHost | C:\Users\UserName\Documents\WindowsPowerShell\Microsoft.PowerShell_profile.ps1 |

Wybór odpowiedniego profilu

Dwa z czterech profili są używane przez wszystkich użytkowników programu Windows PowerShell na komputerze. Wszystko, co znajduje się w profilach AllUsers*, jest dostępne dla każdego skryptu i użytkownika, który uruchomi konsolę. W efekcie należy dobrze się zastanowić, co się umieści w takim profilu. Takie ogólne profile są doskonałym miejscem do konfigurowania aliasów potrzebnych wszystkim użytkownikom, zmiennych przeznaczonych do użycia w korporacyjnym środowisku skryptowym oraz definiowania dysków i funkcji Windows PowerShell. W istocie wszystko, co powinno należeć do korporacyjnego środowiska Windows PowerShell, najlepiej jest umieścić w profilach dla wszystkich użytkowników.

Powstaje pytanie: którego z dwóch profili dla wszystkich użytkowników użyć? Profil AllUsersAllHosts jest dla wszystkich użytkowników i każdej instancji programu Windows PowerShell, wliczając konsolę PowerShell, zintegrowane środowisko PowerShell (ISE) oraz każdy inny program mogący hostować Windows PowerShell, np. Exchange Management Environment i konsolę SQL. Nawet jeśli wszystkie aliasy, zmienne, funkcje i dyski Windows PowerShell utworzy się z dużą rozważą, to i tak wszystkie je trzeba przetestować, aby mieć pewność, że są ze sobą zgodne.

Profil AllUsersCurrentHost daje takie same możliwości w zakresie modyfikowania środowiska Windows PowerShell dla wszystkich użytkowników co AllUsersAllHosts, ale dotyczy tylko hosta konsoli.

Profile CurrentUser* służą do modyfikowania środowiska Windows PowerShell dla bieżącego użytkownika. Użytkownicy do konfigurowania własnych ustawień środowiska Windows PowerShell

najczęściej używają profilu `CurrentUserCurrentHost`. Odnosi się do niego zmienna automatyczna `$profile`. W moim komputerze wartość tej zmiennej jest następująca:

```
PS C:\> $PROFILE
C:\Users\edwilson\Documents\WindowsPowerShell\Microsoft.PowerShell_profile.ps1
```

Jak widać, w systemie Windows Vista folder osobisty użytkownika znajduje się w folderze *Dokumenty*. Jeśli nie zostanie utworzony żaden profil, to folder *WindowsPowerShell* nie istnieje. Widać to w poniższym przykładzie, w którym za pomocą polecenia `Test-Path` sprawdzono, czy istnieje folder nadrzędny, który powinien zawierać plik *Microsoft.PowerShell_profile.ps1*. Jako że w moim laptopie jeszcze nie utworzyłem żadnego osobistego profilu, folder ten nie został utworzony.

```
PS C:\> Test-path (Split-Path $PROFILE -Parent)
False
```

Profil `CurrentUserCurrentHost` można utworzyć za pomocą polecenia `New-Item`. Jeśli folder nie istnieje, należy dodać parametr `-force` i określić typ elementu (`itemtype`), jak pokazano poniżej:

```
New-Item -Path $PROFILE -ItemType file -Force
```

Utworzony profil można otworzyć w Notatniku albo konsoli Windows PowerShell ISE w celu modyfikacji. Jeśli wybierzesz Notatnik, to wystarczy w wierszu poleceń wpisać `notepad` i zmienną automatyczną `$profile`, jak pokazano poniżej:

```
Notepad $profile
```

Poniżej znajduje się zawartość profilu `CurrentUserCurrentHost` po dodaniu funkcji, zmiennych, aliasów i dysku Windows PowerShell.

CurrentUserCurrentHostProfile.ps1

```
# *** funkcje ***
Function Set-Profile()
{
    Notepad $profile
    # Funkcja Pana Eda
}

Function Get-MoreHelp()
{
    #.Help Get-MoreHelp Get-Command
    Get-Help $args[0] -Full |
    more
    #MrEd function
} #end Get-MoreHelp

Function Get-WmiClass([string]$ns, [string]$class)
{
    #.Help Get-WmiClass -ns "root\cimv2" -class "Processor"
    $ns = $args[0]
    $class = $args[1]
    Get-WmiObject -List -Namespace $ns |
    Where-Object { $_.name -match $class }
    #MrEd function
}
```

```

} #end Get-WmiClass
# *** aliasy ***

New-Alias -Name mo -Value Measure-Object -Option allscope '
-Description "Alias Pana Eda"
New-Alias -name gmh -value Get-MoreHelp -Option allscope '
-Description "Alias Pana Eda"
New-Alias -Name gwc -Value Get-WmiClass -Option readonly,allscope '
-Description "Alias Pana Eda"

# *** zmienne ***

New-Variable -Name wulog -Value (Join-Path -Path $env:LOCALAPPDATA '
-ChildPath microsoft\windows\windowsupdate.log -Resolve) '
-Option readonly -Description "Zmienna Pana Eda"
New-Variable -Name docs -Value (Join-Path -Path $home -ChildPath documents) '
-Option readonly -Description "Zmienna Pana Eda"
New-Variable -name wshShell -value (New-Object -ComObject Wscript.Shell) '
-Option readonly -Description "Zmienna Pana Eda"

# *** dyski PowerShell ***

New-PSDrive -Name HKCR -PSProvider registry -Root Hkey_Classes_Root '
-Description "Dysk Pana Eda" | out-null

```

Tworzenie innych profili

Do profilu `CurrentUserCurrentHost` można odnosić się przy użyciu zmiennej automatycznej `$profile`, natomiast do innych profili można odwoływać się przy użyciu notacji kropkowej. Aby użyć profilu `AllUsersAllHosts`, można posłużyć się zmienną `$profile` w sposób pokazany poniżej:

```

PS C:\> $PROFILE.AllUsersAllHosts
C:\Windows\system32\WindowsPowerShell\v1.0\profile.ps1

```

Ponadto profil `AllUsersAllHosts` można łatwo utworzyć przy użyciu tej samej techniki, którą tworzy się profil `CurrentUserCurrentHost`:

```
New-Item -Path $PROFILE.AllUsersAllHosts -ItemType file -Force
```

Należy tylko pamiętać, że w systemie Windows Vista i nowszych konsola Windows PowerShell powinna zostać uruchomiona przez kliknięcie prawym przyciskiem myszy skrótu i wybranie z menu podręcznego opcji *Uruchom jako administrator*. Jest to konieczne, ponieważ katalog *System32* należy do chronionej części systemu plików.

Aby utworzyć profil `AllUsersCurrentProfile`, również należy uruchomić program Windows PowerShell jako administrator, a następnie użyć polecenia `New-Item`, jak pokazano poniżej:

```
New-Item -Path $PROFILE.AllUsersCurrentHost -ItemType file -Force
```

Profil `CurrentUserAllHosts` można utworzyć bez posiadania uprawnień administracyjnych, ponieważ jego plik jest zapisywany w folderze *Dokumenty*. Dlatego każdy typowy użytkownik może utworzyć profile `CurrentUserAllHosts` i `CurrentUserCurrentHost`. Poniżej znajduje się polecenie tworzące profil `CurrentUserAllHosts`:

```
New-Item -Path $PROFILE.CurrentUserAllHosts -ItemType file -Force
```

Pracując z profilami, zawsze należy mieć na uwadze skutki zastosowania różnych ustawień. Istnieje możliwość, że ustawienia skonfigurowane w jednym profilu zostaną zmienione przez ustawienia zdefiniowane w innym profilu, oraz może dojść do kumulacji różnych ustawień. W takim przypadku znaczenia nabiera pojęcie RSOP (ang. *Resultant Set of Profiles*). Ustawienia poszczególnych profili są stosowane w określonej kolejności. Pierwszy ma największą szansę, że zostanie nadpisany. Profil będący najbliższym użytkownika — `CurrentUserCurrentProfile` — ma najwyższy priorytet:

- All Users, All Hosts
- All Users, Current Host
- Current User, All Hosts
- Current User, Current Host

Morał

Stosuj standardowe konwencje nazewnictwa, aby uniknąć konfliktów

Standardowe aliasy i zmienne powinno się zawsze oznaczać jako stałe, aby zawsze były dostępne. Przy tworzeniu funkcji i dysków Windows PowerShell należy trzymać się konwencji nazewnictwa, która stwarza niewielkie ryzyko wystąpienia konfliktów nazw. W pewnej znanej mi firmie nazwy funkcji poprzedza się przedrostkiem w postaci nazwy tej firmy, np.:

```
Function FirmaAITWigitFunction() { coś ciekawego do zrobienia }
New-Alias -Name CAWA -Value FirmaAITWigitFunction -Option constant '
-Description "Alias firmowy"
```

Wiedza tajemna

Jak używać profilu

James Brundage, właściciel firmy

Start-Automating

Profile to bardzo ciekawy rodzaj kompromisu. Ich zaletą jest to, że umożliwiają skonfigurowanie spójnego i spersonalizowanego środowiska dostosowanego do potrzeb użytkownika. Wadą natomiast jest to, że spersonalizowany profil zawsze trudniej jest udostępnić innym niż standardowy. Dlatego to, w jaki sposób napiszemy nasz profil, może mieć wielki wpływ na to, czy będzie nadawał się do użytku także przez innych.

Moim zdaniem idealny profil to po prostu seria instrukcji importujących moduły lub dołączających skrypty. Zarówno moduły, jak i skrypty można łatwo skopiować z jednego komputera na inny, co oznacza, że tak zbudowany profil jest przejrzysty, a używane w nim skrypty są łatwe do przeniesienia w inne miejsce. Jeśli w swoich profilach będziesz umieszczać tylko instrukcje importu modułów i dołączać skrypty, to aby przenieść się na inny komputer, wystarczy skopiować profil i parę dodatkowych plików.

Ponadto profil może bardzo ułatwić życie. Konsola Windows PowerShell ISE zawiera model obiektowy umożliwiający dodawanie narzędzi do środowiska, a PowerShell umożliwia dostosowywanie monitorowania przez napisanie własnej funkcji monitorującej. Gdy dodaję do środowiska jakieś ciekawe rzeczy, np. menu *Verb*, to zawsze umieszczam je w profilu.

Przy okazji: najlepszą rzeczą, jaką kiedykolwiek dodałem do swojego profilu, było właśnie menu *Verb*. Napisałem skrypt tworzący hierarchię menu w ISE, aby móc klikać polecenia według czasowników. (Na przykład otwieram menu *Get* i klikam polecenie *Process*, aby wykonać polecenie *Get-Process*). Tego rodzaju usprawnienia są doskonałym dodatkiem do profilu, ponieważ znacznie ułatwiają pracę w środowisku skryptowym.

Raczej unikam definiowania aliasów w profilu, ponieważ utrudniają one przekazywanie skryptów innym użytkownikom poza firmą Microsoft (czasami mogę zapomnieć pousuwać aliasy przed opublikowaniem skryptu na blogu). Oczywiście jeśli ktoś nie planuje udostępniać swoich skryptów innym użytkownikom, to aliasy w profilu są świetnym udogodnieniem. Uważam, że przez aliasy trzeba przenosić swój profil na każdy komputer, przy którym się pracuje. A ponieważ prowadzę blog, staram się minimalizować zależności w swoich skryptach. Dlatego unikam aliasów, które są taką niepotrzebną zależnością.

Mój profil jest niedługi, ponieważ prawie wszystko przechowuję w modułach. Mam kilka elementów więcej w zmiennej `$loadedModules`, ale poniższy kod ilustruje, jak mniej więcej wygląda mój profil:

```
$loadedModules = 'DotNet',
                 'WPF'
```

```
Import-Module $loadedModules -force
```

Moduł *DotNet* jest bardzo prosty. Jest to zwykły plik z rozszerzeniem *.ps1*, który dołącza plik z funkcją *Get-Type*. Znajduje się w folderze `$env:UserProfile\Documents\WindowsPowerShell\Modules\DotNet`.

```
MyDotNetmodule
```

```
. $psScriptRoot\Get-Type.ps1
```

```
Get-Type.ps1:
```

```
Function Get-Type() {
    [AppDomain]::CurrentDomain.GetAssemblies() | Foreach-Object {
        $_.GetTypes()
    }
}
```

Funkcja *Get-Type* to najbardziej przydatny dodatek, jaki kiedykolwiek dodałem do profilu. Zwraca wszystkie aktualnie załadowane typy, które mogę przeszukiwać w konsoli Windows PowerShell, na przykład:

```
Get-Type | Where-Object { $_.FullName -like "***File*" }
```

Używanie funkcji z innych skryptów

Gdy napiszesz dużo funkcji, niektórych z nich możesz zechcieć użyć w różnych skryptach. Wielokrotne wykorzystanie kodu to doskonały pomysł. Najprostszym sposobem na jego realizację jest skopiowanie funkcji z jednego skryptu i wklejenie jej do innego. Wyobraźmy sobie, że mamy

skrypt zawierający kod wykonujący konwersję stopni Celsjusza na stopnie Fahrenheita i chcemy wykorzystać ten algorytm w innym skrypcie. Jedną z możliwości jest po prostu skopiowanie kodu do nowego skryptu. Po zrobieniu tego skrypt może wyglądać mniej więcej tak:

ConvertToFahrenheit.ps1

```
Param($Celsius)
Function ConvertToFahrenheit($Celsius)
{
    "$Celsius stopni Celsjusza równa się $((1.8 * $Celsius) + 32) stopni Fahrenheita"
} #end ConvertToFahrenheit
ConvertToFahrenheit($Celsius)
```

Nie mam nic do zarzucenia temu skryptowi. Wykonuje jedną czynność i robi to całkiem dobrze. Aby go użyć, należy przekazać argument w wierszu poleceń. W wywołaniu tego skryptu nie trzeba wpisywać całej nazwy parametru, jak widać poniżej:

```
PS C:\> C:\BestPracticesBook\ConvertToFahrenheit.ps1 -c 24
24 stopni Celsjusza równa się 75.2 stopni Fahrenheita
```

Tworzenie biblioteki funkcji

Problem z wielokrotnym użyciem kodu poprzez jego kopiowanie w nowe miejsce polega na tym, że gdy chcemy coś zmienić w funkcji, musimy znaleźć wszystkie jej wystąpienia. Jeśli tego nie zrobimy, powstanie kilka nieco różniących się między sobą wersji tej samej funkcji, co przysporzy nam wiele problemów z utrzymaniem tego kodu.

Jak to rozwiązać? Jedną z możliwości jest zapisanie wszystkich funkcji w jednym skrypcie, takim jak pokazany poniżej *ConversionFunctions.ps1*.

ConversionFunctions.ps1

```
Function ConvertToMeters($feet)
{
    "$feet stóp równa się $($feet*.31) metrów"
} #end ConvertToMeters
Function ConvertToFeet($meters)
{
    "$meters metrów równa się $($meters * 3.28) stóp"
} #end ConvertToFeet
Function ConvertToFahrenheit($celsius)
{
    "$celsius stopni Celsjusza równa się $((1.8 * $celsius) + 32 ) stopni Fahrenheita"
} #end ConvertToFahrenheit
Function ConvertToCelsius($fahrenheit)
{
    "$fahrenheit stopni Fahrenheita równa się $( ( ($fahrenheit - 32)/9)*5 ) stopni Celsjusza"
} #end ConvertToCelsius
Function ConvertToMiles($kilometer)
{
    "$kilometer kilometrów równa się $( ($kilometer *.6211) ) mil"
} #end convertToMiles
Function ConvertToKilometers($miles)
{
    "$miles mil równa się $( ($miles * 1.61) ) kilometrów"
} #end convertToKilometers
```

Dołączanie pliku

Jeśli chcesz użyć jednej ze swoich funkcji konwersji, możesz ją dołączyć do skryptu przez postawienie kropki przed ścieżką do skryptu. Po dołączeniu całego skryptu z funkcjami konwersji ma się dostęp do wszystkich tych funkcji i można ich używać w taki sposób, jakby znajdowały się bezpośrednio w pliku, do którego zostały dołączone. Ilustracja zastosowania tej techniki znajduje się w skrypcie *ConvertToFahrenheit_Include.ps1*. Nadal można używać parametru wiersza poleceń `$Celsius` w celu przekazania wartości temperaturowej, która ma zostać przekonwertowana. Później za pomocą kropki i ścieżki do skryptu dołączamy plik. Na końcu wywołujemy funkcję przy użyciu jej nazwy i podajemy jej wartość przekazaną do skryptu poprzez wiersz poleceń. Poniżej znajduje się kod źródłowy skryptu *ConvertToFahrenheit_Include.ps1*:

ConvertToFahrenheit_Include.ps1

```
Param($Celsius)
. C:\data\scriptingGuys\ConversionFunctions.ps1
ConvertToFahrenheit($Celsius)
```

Jak widać, ten skrypt jest znacznie bardziej zwięzły i przejrzysty, a dzięki temu również czytelniejszy. A bardziej czytelny skrypt łatwiej jest zrozumieć, co z kolei sprawia, że łatwiej jest też go utrzymać. Oczywiście takie podejście ma nie tylko zalety. Po pierwsze: teraz skrypty są ze sobą powiązane. Zmiana w jednym może spowodować zmianę w drugim. Ale jeszcze ważniejsze jest to, że od tej pory skrypty te muszą być wszędzie zabierane razem. Jeśli wystąpią jakieś problemy, to ich rozwiązanie przy takiej zewnętrznej zależności może być trudne.

Jednym ze sposobów na ułatwienie rozwiązywania problemów ze skryptem jest sprawdzenie dostępności dołączanego pliku za pomocą polecenia `Test-Path`. Jeśli pliku tego nie ma, można wygenerować stosowne powiadomienie, co znacznie ułatwi pracę. Używanie polecenia `Test-Path` zawsze, gdy dołącza się jakiś plik, jest bardzo dobrym zwyczajem. Poniżej pokazano zmieniony skrypt, który można znaleźć też w pliku *ConvertToFahrenheit_Include2.ps1*:

ConvertToFahrenheit_Include2.ps1

```
Param($Celsius)
$includeFile = "c:\data\scriptingGuys\ConversionFunctions.ps1"
if(!(test-path -path $includeFile))
{
    "Nie znaleziono pliku $includeFile"
    Exit
}
. $includeFile
ConvertToFahrenheit($Celsius)
```

Jak widać, zaczynamy popadać w paranoję. Aby użyć funkcji zawierającej trzy wiersze kodu, napisaliśmy skrypt składający się z dziewięciu wierszy kodu. Jeśli chcesz użyć dołączanego pliku, musisz wykonać wywołanie. Korzyści z zastosowania tej metody stają się ewidentne, gdy używa się bardziej rozbudowanych skryptów. W skrypcie *ConvertUseFunctions.ps1* znajduje się funkcja `ParseAction`, która bada czynność i wartość przekazane w wierszu poleceń i na podstawie zdobytych informacji wywołuje odpowiednią funkcję, jak widać w skrypcie *ConvertUseFunctions.ps1*.

ConvertUseFunctions.ps1

```

Param($action,$value,[switch]$help)
Function GetHelp()
{
    if($help)
    {
        "Wybierz rodzaj konwersji: M (metry), F (stopy) C (stopnie Celsjusza),Fa (stopnie
        Fahrenheita),Mi(mile),K (kilometry) oraz wartość"
        " Polecenie Convert -a M -v 10 konwertuje 10 metrów na stopy."
    } #end if help
} #end getHelp
Function GetInclude()
{
    $includeFile = "c:\data\scriptingGuys\ConversionFunctions.ps1"
    if(!(test-path -path $includeFile))
    {
        "Nie znaleziono pliku $includeFile"
        Exit
    }
    . $includeFile
} #end GetInclude
Function ParseAction()
{
    switch ($action)
    {
        "M" { ConvertToFeet($value) }
        "F" { ConvertToMeters($value) }
        "C" { ConvertToFahrenheit($value) }
        "Fa" { ConvertToCelsius($value) }
        "Mi" { ConvertToKilometers($value) }
        "K" { ConvertToMiles($value) }
        DEFAULT { "Nieprawidłowa wartość." ; GetHelp ; exit }
    } #end action
} #end ParseAction
# *** punkt wejściowy ***
If($help) { GetHelp ; exit }
if(!$action) { "Nie określono czynności" ; GetHelp ; exit }
GetInclude
ParseAction

```

Pamiętaj, że trzeba zmienić dołączany plik. Jako że ładujemy funkcje z wnętrza funkcji, należą one domyślnie do przestrzeni nazw tej funkcji. Są niedostępne w innych funkcjach — tylko z elementów potomnych. Aby uniknąć problemu z dziedziczeniem, do każdej funkcji można dodać znacznik skryptu, jak pokazano poniżej:

```

Function Script:ConvertToMeters($feet)
{
    "$feet stóp równa się $($feet*.31) metrów"
} #end ConvertToMeters

```

Dodatkowe źródła informacji

- Centrum skryptów portalu TechNet pod adresem <http://www.microsoft.com/technet/scriptcenter> zawiera wiele przykładowych skryptów.
- W portalu MSDN na stronie [http://msdn.microsoft.com/en-us/library/bb613488\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/bb613488(VS.85).aspx) znajduje się opis profili programu Windows PowerShell.
- W portalu MSDN na stronie [http://msdn.microsoft.com/en-us/library/bb648601\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/bb648601(VS.85).aspx) znajduje się opis zasad wykonywania programu Windows PowerShell.
- Wszystkie skrypty opisane w tym rozdziale są dostępne do pobrania w repozytorium centrum skryptów portalu TechNet pod adresem <http://gallery.technet.microsoft.com/scriptcenter/PowerShell-40-Best-d9e16039>.

Rozdział 6

Unikanie pułapek podczas pisania skryptów

- Brak obsługi poleceń
- Skomplikowane konstruktory
- Kwestie dotyczące zgodności wersji
- Brak obsługi WMI
- Praca z obiektami i przestrzeniami nazw
- Pobieranie listy dostawców WMI
- Praca z klasami WMI
- Brak obsługi platformy .NET
- Dodatkowe źródła informacji

Wiedza, czego nie należy obejmować skrypcem, jest tak samo cenna jak wiedza, co należy zapisać w postaci skryptu. Czasami utworzenie skryptu nie jest najlepszym rozwiązaniem z powodu braku obsługi jakiejś technologii albo poziomu złożoności projektu. W tym rozdziale opisuję pewne niebezpieczeństwa zagrażające projektom skryptów.

Brak obsługi poleceń

Jest oczywiste, że to właśnie polecenia cmdlet są solą narzędzia Windows PowerShell. Jeśli trzeba sprawdzić status usługi *bits*, najłatwiej jest to zrobić za pomocą polecenia `Get-Service`:

```
Get-Service -name bits
```

Aby dowiedzieć się czegoś o procesie *explorer*, można użyć polecenia `Get-Process`:

```
Get-Process -Name explorer
```

Natomiast gdy trzeba zatrzymać jakiś proces, można użyć polecenia `Stop-Process`:

```
Stop-Process -Name notepad
```

Przy użyciu przełącznika `-ComputerName` polecenia `Get-Service` można nawet sprawdzać status usług na zdalnych komputerach:

```
Get-Service -Name bits -ComputerName vista
```

WAŻNE

Jeśli pracujesz w środowisku międzydomenowym, w którym wymagane jest uwierzytelnienie się, to nie będziesz mógł używać poleceń `Get-Service` ani `Get-Process`, ponieważ nie mają one parametru `-credential`. W takim przypadku należy użyć poleceń do pracy zdalnej, takich jak `Invoke-Command`, które umożliwiają wpisanie danych poświadczających.

Informacje na temat systemu BIOS komputera można odczytać i zapisać w pliku CSV za pomocą tylko paru linijek kodu. Poniżej znajduje się kod przykładowego skryptu służącego do tego celu, o nazwie *ExportBiosToCsv.ps1*:

ExportBiosToCsv.ps1

```
$path = "c:\fso\bios.csv"
Get-CimInstance -ClassName win32_bios |
Select-Object -property name, version |
Export-CSV -path $path -noTypeInfoation
```

Gdyby nie było poleceń do wybierania obiektów i eksportowania ich do formatu CSV, pewnie zwrócilibyśmy się ku obiektowi `filesystemobject` z języka VBScript firmy Microsoft. Skrypt z użyciem tego obiektu byłby prawie dwa razy dłuższy i o wiele mniej czytelny. Poniżej znajduje się przykład użycia tego obiektu (skrypt *FSOBiosToCsv.ps1*):

FSOBiosToCsv.ps1

```
$path = "c:\fso\bios1.csv"
$bios = Get-CimInstance -ClassName win32_bios
$csv = "Name,Version`r`n"
$csv += $bios.name + "," + $bios.version
$fso = new-object -comobject scripting.filesystemobject
$file = $fso.CreateTextFile($path,$true)
$file.write($csv)
$file.close()
```

Niewątpliwie dostępność wbudowanych poleceń jest wielką zaletą narzędzia Windows PowerShell. Ale jednym z problemów, jakie można napotkać podczas używania programu Windows PowerShell 4.0 i systemu Windows Server 2012, jest liczba poleceń. Podobny problem mają administratorzy serwera Windows Exchange Server. Tych poleceń jest tak dużo, że nie wiadomo, od czego zacząć. Wystarczy pobieżnie przejrzeć blog zespołu ds. Microsoft Exchange i fora Exchange, aby dowiedzieć się, że problem nie dotyczy pisania skryptów, tylko znalezienia wśród setek poleceń tych, które najlepiej nadają się do danego celu. Jeśli doda się do tego jeszcze polecenia tworzone przez ochotników i różne firmy programistyczne, potencjalnie Twoje środowisko może obsługiwać nawet tysiące poleceń.

Na szczęście programiści programu Windows PowerShell mają plan, jak rozwiązać ten problem — standardowe konwencje nazewnictwa. Polecenia `Get-Help`, `Get-Command` oraz `Get-Member` zostały opisane już w rozdziale 1., ale warto przywołać je jeszcze raz. Jeśli nie wiesz o jakiejś funkcji lub istnieniu jakiegoś polecenia, musisz zaimplementować własne rozwiązanie, co wymaga

dotądowej pracy i może zamaskować ukryte błędy. Jeśli masz wybór między gotowym poleceniem należącym do systemu operacyjnego i własnym rozwiązaniem, prawie zawsze powinieneś wybrać gotowe polecenie. Dlatego zamiast od razu zakładać, że określone polecenie nie istnieje, powinno się go poszukać za pomocą narzędzi `Get-Help`, `Get-Command` oraz `Get-Member` i dopiero potem ewentualnie napisać własne polecenie. W rozdziale tym opisuję niektóre trudności, jakie mogą wyniknąć z powodu używania niestandardowych poleceń.

Skomplikowane konstruktory

Jeśli podczas projektowania skryptu nie ma pomocy ze strony poleceń, może istnieć lepszy sposób na wykonanie tego zadania i powinno się przynajmniej rozważyć alternatywne możliwości.

W skrypcie *GetRandomObject.ps1* tworzona jest funkcja o nazwie `GetRandomObject`. Funkcja ta przyjmuje dwa parametry wejściowe: jeden, o nazwie `$in`, przechowuje tablicę obiektów, a drugi nazywa się `$count` i kontroluje liczbę elementów losowo wybieranych z obiektu wejściowego.

Za pomocą polecenia `New-Object` tworzony jest egzemplarz klasy `.NET System.Random`. Do utworzenia tego egzemplarza użyto domyślnego konstruktora (bez ziarna) i zapisano go w zmiennej `$rnd`.

Do przeglądania kolekcji użyto pętli `for-next`. Metoda `next` klasy `System.Random` służy do wybierania losowej liczby z przedziału od 1 do liczby wszystkich elementów w obiekcie wejściowym. Losowa liczba jest wykorzystywana do lokalizacji elementu w tablicy za pomocą indeksu. Kod źródłowy opisywanego skryptu *GetRandomObject.ps1* pokazano poniżej.

GetRandomObject.ps1

```
Function GetRandomObject($in,$count)
{
    $rnd = New-Object system.random
    for($i = 1 ; $i -le $count; $i ++ )
    {
        $in[$rnd.Next(1,$a.length)]
    } #end for
} #end GetRandomObject

# *** punkt wejściowy ***
$a = 1,2,3,4,5,6,7,8,9
$count = 3
GetRandomObject -in $a -count $count
```

Podczas gdy teoretycznie nic nie można zarzucić temu skryptowi, użytkownicy konsoli Windows PowerShell 2.0 i jej nowszych wersji do tego samego celu mogą używać polecenia `Get-Random`, jak pokazano poniżej:

```
$a = 1,2,3,4,5,6,7,8,9
Get-Random -InputObject $a -Count 3
```

Jak widać, korzystając z tego polecenia, można oszczędzić dużo czasu i wysiłku.

Jedną z zalet używania standardowych poleceń jest to, że można być pewnym poprawności ich implementacji. Niektóre klasy platformy .NET mają skomplikowane konstruktory służące do tworzenia egzemplarzy w określony sposób. Przekazanie błędnej wartości do takiego konstruktora

nie zawsze powoduje wygenerowanie błędu. Taki kod może sprawiać wrażenie, że działa poprawnie, przez co wykrycie błędu może być bardzo trudne.

Przykład tego rodzaju błędu przedstawiono w poniższym skrypcie *BadGetRandomObject.ps1*, w którym do konstruktora klasy `.NET System.Random` przekazano liczbę całkowitą. Problem polega na tym, że skrypt ten za każdym razem wygeneruje tę samą liczbę całkowitą. Ten konkretny błąd jest banalny, ilustruje jednak, jakie pułapki czekają na programistę. Musi on bardzo dokładnie znać klasy `.NET`, aby je wykryć.

BadGetRandomObject.ps1

```
Function GetRandomObject($in,$count,$seed)
{
    $rnd = New-Object system.random($seed)
    for($i = 1 ; $i -le $count; $i ++ )
    {
        $in[$rnd.Next(1,$a.length)]
    } #end for
} #end GetRandomObject

# *** punkt wejściowy ***
$a = 1,2,3,4,5,6,7,8,9
$count = 3
GetRandomObject -in $a -count $count -seed 5
```

Informacje na temat klasy `System.Random` można znaleźć w portalu MSDN, ale łatwo przeoczyć jakiś szczegół, ponieważ dokumentacja jest bardzo obszerna, a niektóre klasy są bardzo skomplikowane. Gdy przeoczony szczegół nie powoduje błędu wykonawczego, a skrypt sprawia wrażenie, jakby działał prawidłowo, to w najlepszym przypadku mamy krępującą sytuację.

Kwestie dotyczące zgodności wersji

Internet jest doskonałym źródłem informacji, ale często zamiast pomagać, może tylko przeszkadzać. Znalezione źródło informacji może nie być aktualne i zawierać dane dotyczące przestarzałej wersji systemu operacyjnego. Sytuację pogarsza jeszcze istnienie wielu innych czynników mogących mieć wpływ na działanie programu, jak funkcja kontroli użytkownika, zapora systemowa i inne zabezpieczenia, które mogą być skonfigurowane na tak wiele sposobów, że czasami trudno jest ocenić, czy błąd jest wynikiem zmian wprowadzonych w systemie, czy efektem pomyłki w kodzie źródłowym. Powiedzmy na przykład, że chcemy użyć klasy `WMI Win32_Volume` w celu zdobycia informacji o dyskach. Po pierwsze: musimy wiedzieć, że klasa ta jest dostępna dopiero od systemu Microsoft Windows Server 2003. To zaskakujące, że nie ma jej w systemie Windows XP. Ale jeśli spróbujemy wykonać poniższe polecenie w systemie Windows Vista, zostanie zgłoszony błąd:

```
Get-WmiObject -Class win32_volume -Filter "Name = 'c:\'"
```

Pierwszym podejrzanym przy rozwiązywaniu problemów w systemie Windows Vista i nowszych wersjach systemu Windows są uprawnienia użytkownika. Należy uruchomić konsolę Windows PowerShell jako administrator i ponowić próbę wykonania kodu. Znow się nie uda. W następnej kolejności można sprawdzić, czy problem nie ma związku z rodzajem cudzysłówów. Po chwili

namysłu postanawiasz użyć tylko cudzysłowów literalnych. To wymaga zastosowania sekwencji specjalnych, czyli większego nakładu pracy. Ale jeśli kod zadziała, to ta praca się opłaci. W związku z tym otrzymujemy poniższy kod, który jednak niestety też nie zadziała:

```
Get-WmiObject -Class win32_volume -Filter 'Name = 'c:\'''
```

W końcu postanawiamy przeczytać informację o błędzie. Oto ona:

```
Get-WmiObject : Invalid query
At line:1 char:14
+ Get-WmiObject <<<< -Class win32_volume -Filter "Name = 'c:\' "
+ ~~~~~
+ CategoryInfo          : InvalidOperation: (:) [Get-WmiObject], ManagementException
+ FullyQualifiedErrorId : GetWMIManagementException,
    Microsoft.PowerShell.Commands.GetWmiObjectCommand
```

Patrzmy na linijkę zawierającą tekst `InvalidOperation` i dochodzimy do wniosku, że ukośnik prawdopodobnie jest znakiem specjalnym. Skoro tak, to wystarczy zamienić go na sekwencję specjalną i spróbować jeszcze raz wykonać kod, który po tym zabiegu powinien wyglądać tak jak poniżej:

```
Get-WmiObject -Class win32_volume -Filter "Name = 'c:\' "
```

Pomysł był dobry, ale kod i tak nie działa, tylko powoduje zwrócenie następującego błędu:

```
Get-WmiObject : Invalid query
At line:1 char:14
+ Get-WmiObject <<<< -Class win32_volume -Filter "Name = 'c:\' "
+ ~~~~~
+ CategoryInfo          : InvalidOperation: (:) [Get-WmiObject], ManagementException
+ FullyQualifiedErrorId : GetWMIManagementException,
    Microsoft.PowerShell.Commands.GetWmiObjectCommand
```

W następnej kolejności sprawdzamy, czy mamy uprawnienia do wykonania tego zapytania. (Wiem, że konsola jest uruchomiona z uprawnieniami administratora, ale niektóre procesy odmawiają dostępu nawet administratorowi, więc warto spróbować). Najprostszym sposobem na sprawdzenie posiadanych uprawnień jest wykonanie zapytania WMI z pominięciem parametru `-filter`, jak pokazano poniżej:

```
Get-WmiObject -Class win32_volume
```

To polecenie zostaje wykonane bez żadnych błędów. Możemy pomyśleć, że w ogóle nie możemy filtrować klasy WMI, i dojść do wniosku, że jest ona jakaś dziwna. Postanawiamy napisać inny filtr, aby sprawdzić, czy przyjmie składnię nowego filtra, jak pokazano poniżej:

```
Get-WmiObject -Class win32_volume -Filter "DriveLetter = 'c:'"
```

Powyższe polecenie zwróci wynik podobny do przedstawionego poniżej:

```
PS C:\> Get-WmiObject -Class win32_volume -Filter "DriveLetter = 'c:'"
```

```
__GENUS                : 2
__CLASS                : Win32_Volume
__SUPERCLASS           : CIM_StorageVolume
__DYNASTY               : CIM_ManagedSystemElement
__RELPATH              : Win32_Volume.DeviceID="\\\\\\?\\Volume{194e9837-4954-11e4-8250-
                        806e6f6e6963}\\"
```

```
__PROPERTY_COUNT      : 44
__DERIVATION          : {CIM_StorageVolume, CIM_StorageExtent, CIM_LogicalDevice,
                        CIM_LogicalElement...}
__SERVER              : WIN-5I6S4NUE99G
__NAMESPACE          : root\cimv2
__PATH                : \\WIN-5I6S4NUE99G\root\cimv2:Win32_Volume.DeviceID="\
                        \\?\Volume{194e9837-4954-11e4-8250-806e6f6e6963}\\\"
Access                :
Automount              : True
Availability           :
BlockSize             : 4096
BootVolume            : True
Capacity              : 64422408192
Caption               : C:\
Compressed             : False
ConfigManagerErrorCode :
ConfigManagerUserConfig :
CreationClassName     :
Description           :
DeviceID              : \\?\Volume{194e9837-4954-11e4-8250-806e6f6e6963}\
DirtyBitSet           :
DriveLetter           : C:
DriveType             : 3
ErrorCleared          :
ErrorDescription       :
ErrorMethodology      :
FileSystem             : NTFS
FreeSpace             : 45035909120
IndexingEnabled       : True
InstallDate           :
Label                 :
LastErrorCode          :
MaximumFileNameLength : 255
Name                  : C:\
NumberOfBlocks        :
PageFilePresent       : True
PNPDeviceID           :
PowerManagementCapabilities :
PowerManagementSupported :
Purpose               :
QuotasEnabled         :
QuotasIncomplete      :
QuotasRebuilding      :
SerialNumber          : 2392445071
Status                :
StatusInfo            :
SupportsDiskQuotas    : True
SupportsFileBasedCompression : True
SystemCreationClassName :
SystemName            :
SystemVolume          : True
PSComputerName        : WIN-5I6S4NUE99G
```

Wiedza tajemna

Podczas pracy ze skryptami administratorzy sieci i konsultanci często wykorzystują różne sztuczki, bo przecież ich zadaniem jest „sprawienie, aby wszystko działało”.

Sam też od czasu do czasu używam jakichś sztuczek, bo przecież moim zadaniem jest codzienne opublikowanie przynajmniej jednego artykułu, a to znaczy, że codziennie mam termin wykonania pracy. Wracając do opisywanego powyżej przykładu z klasą WMI Win32_Volume: przy wykonywaniu tego zapytania WMI zawsze używam własności DriveLetter. Wiele lat temu po kilku godzinach ślęczenia nad tym przykładem doszedłem do wniosku, że prawdopodobnie coś jest nie tak z własnością name, i postanowiłem jej unikać podczas prowadzenia wykładów.

Na szczęście żaden student nigdy nie spytał, dlaczego używam DriveLetter zamiast name w zapytaniach.

Jeśli wrócimy do wygenerowanej przez wcześniejsze zapytania wiadomości o błędzie, to pole InvalidOperation CategoryInfo może skłonić nas do ponownego przemyślenia kwestii ukośnika. Próby wyeliminowania gołego ukośnika były dobrym tropem. Problem dotyczy dziwnej mieszanki składni WMI Query Language (WQL) i Windows PowerShell. Parametr -filter bez wątpienia należy do składni Windows PowerShell, ale w parametrze tym należy przekazać łańcuch zgodny z dialektem WQL. Dlatego właśnie użyliśmy znaku równości zamiast operatora Windows PowerShell -eq w cudzysłowie zawierającym wartość parametru -filter. Aby pozbyć się gołego ukośnika w składni WQL, należy dodać jeszcze jeden taki ukośnik, podobnie jak się robi w językach C i C++. Poniższy kod odfiltrowuje dysk na podstawie nazwy:

```
Get-WmiObject -Class win32_volume -Filter "Name = 'c:\\'"
```

WAŻNE

Stosowanie podwójnych ukośników podczas pracy z WMI jest niekomfortowe. Mimo że dokumentacja w portalu MSDN jest ciągle doskonalona, to jednak cały czas jest bardzo dużo do zrobienia w tym zakresie. Jako że klasa WMI nie działa zgodnie z oczekiwaniami, zgłosiłem błąd w dokumentacji dotyczący własności name klasy Win32_Volume. Efektem będzie dodanie kolejnej uwagi do opisu tej własności. Od tamtej pory znalazłem jeszcze kilka miejsc, w których trzeba podwajać ukośnik. Te błędy również zgłoszę.

Najlepiej napisać skrypt zwracający informacje z klasy Win32_Volume i ukryć szczegóły implementacji przed użytkownikiem. Użyj funkcji Get-Volume, która jest dostępna od systemu Windows 8. Funkcja ta przyjmuje wiele parametrów wiersza poleceń, z których dwa to -driveletter i -cimsession. Dysk określa się przez podanie litery bez średnika. Domyślnie funkcja ta zwraca informacje z wszystkich woluminów w lokalnym komputerze. Parametr -driveletter funkcji Get-Volume jest pojedynczą literą. Nie trzeba jej wpisywać w cudzysłowie ani dodawać do niej średnika. Parametr -cimsession przyjmuje nazwę komputera lub sesję CIM. Poniżej znajduje się kod źródłowy skryptu *GetVolume.ps1*.

GetVolume.ps1

```
Get-Volume -DriveLetter 'c' -CimSession client1
```

Jeśli pracujesz w środowisku międzydomenowym, musisz podać dane uwierzytelniające na zdalnym komputerze. Funkcja `Get-Volume` nie ma parametru `-credential`. Ale dzięki temu, że przyjmuje sesję CIM, można przekazać dane poświadczające.

W tym celu najpierw należy utworzyć sesję CIM za pomocą polecenia `New-CimSession` oraz podać dane poświadczające i nazwę komputera. Utworzonej sesji CIM można następnie użyć w wywołaniu polecenia `Get-Volume` w celu nawiązania zdalnego połączenia i pobrania informacji. Opisaną technikę zastosowano w skrypcie *GetVolumeWithCredentials.ps1*, którego kod źródłowy znajduje się poniżej:

GetVolumeWithCredentials.ps1

```
$cim = New-CimSession -Credential iammred\administrator -ComputerName client1
Get-Volume -CimSession $cim
```

Wybór odpowiedniej metody pisania skryptów

Luis Canastreiro, Premier Field Engineer

Microsoft Corporation, Portugalia

Gdy piszę skrypt, często dostrzegam wiele różnych możliwości osiągnięcia tego samego celu. Jeśli na przykład piszę skrypt w języku VBScript, wolę korzystać z obiektów COM niż wywoływać zewnętrzne pliki wykonywalne, ponieważ COM jest technologią macierzystą dla VBScript. To samo dotyczy skryptów Windows PowerShell. Wolę używać klas platformy .NET, jeśli brakuje mi jakiegos polecenia Windows PowerShell, ponieważ program ten jest oparty na platformie .NET.

Oczywiście jeśli dostępne jest polecenie wykonujące potrzebną mi czynność, to zawsze używam tego polecenia, ponieważ w ten sposób ukrywam złożone kwestie bezpośredniej pracy z platformą .NET. Chcę przez to powiedzieć, że istnieją klasy .NET, które na pierwszy rzut oka wydają się proste. Ale gdy zacznie się ich używać, okazuje się, że zawierają skomplikowane konstruktory. Jeśli ktoś nie jest ekspertem od danej klasy, to może popełnić błąd, którego wykrycie będzie go kosztowało dużo wysiłku i testowania. Jeśli istnieje polecenie spełniające moje wymagania, to zawsze używam polecenia.

Na przykład zapisu i odczytu rejestru można dokonywać na wiele sposobów. Można użyć metod VBScript `regread` i `regwrite`, klasy `WMI stdRegProv`, klas platformy .NET, a nawet różnych narzędzi wiersza poleceń. Moją ulubioną metodą pracy z rejestrem jest używanie dostawcy rejestru Windows PowerShell oraz poleceń z rodzin `*-item` i `*-itemproperty`. Polecenia te są łatwe w obsłudze i aby ich użyć, wystarczy tylko uruchomić konsolę Windows PowerShell.

Kiedy piszę nowy skrypt, lubię tworzyć niewielkie ogólne funkcje, które mają wiele zalet. Ułatwiają testowanie skryptu podczas jego pisania. Muszę wywołać tylko funkcję lub funkcje, nad którymi pracuję. Pozostałą część kodu mogę zostawić niedokończoną, aby wrócić do niej później. Takie funkcje łatwo jest poprawić i ponownie wykorzystać. Do uruchamiania skryptu tworzę funkcję główną, której głównym celem jest inicjacja środowiska i obsługa globalnego przepływu skryptu poprzez wywołanie odpowiednich funkcji w odpowiednim czasie.

Sprawdzanie wersji systemu operacyjnego

Biorąc pod uwagę to, jak wiele jest wersji systemu operacyjnego Windows, dobrym zwyczajem jest sprawdzanie, w którym systemie jest uruchamiany skrypt, zwłaszcza gdy wiadomo, że mogą wystąpić jakieś problemy ze zgodnością. Jest kilka sposobów na sprawdzenie zgodności wersji. W rozdziale 4. do sprawdzania wersji systemu operacyjnego używaliśmy skryptu o nazwie *Get-OsVersion.ps1*, w którym wykorzystywaliśmy klasę .NET `System.Environment`. Można pobierać informacje z tej klasy zdalnie, ale podobne efekty można osiągnąć przy użyciu klasy `WMI Win32_OperatorSystem`. Zaletą jej użycia jest to, że klasa ta automatycznie łączy się zdalnie z wybranym komputerem.

Skrypt *Get-Version.ps1* przyjmuje dwa parametry wiersza poleceń: `computername` i `credential`. Parametr `computername` przyjmuje tablicę łańcuchów dla lokalnych lub zdalnych połączeń. Natomiast parametr `credential` musi być typem klasy `PSCredential`. Najłatwiejszym sposobem na uzyskanie obiektu poświadczającego jest użycie polecenia `Get-Credential`. Funkcja `Get-Version` używa automatycznej zmiennej `$PSBoundParameters` w celu spakowania do tablicy i przekazania parametrów do polecenia `New-CimSession`. W ten sposób, jeśli wywołamy funkcję i nie prześlemy obiektu `Credential`, nie wystąpi żaden błąd, ponieważ niepusty obiekt poświadczeń nie jest przekazywany do polecenia. Jeżeli nie prześlemy obiektu poświadczeń, polecenie zostanie uruchomione i poda się za zalogowanego użytkownika. Analogicznie, jeśli do parametru `ComputerName` nie zostanie przekazana żadna wartość, funkcja będzie działała na komputerze lokalnym. Dzięki użyciu poleceń CIM w celu dostarczenia informacji z obiektu zarządzania `Win32_OperatingSystem` funkcja może nawiązać połączenie lokalne przy użyciu alternatywnych danych poświadczających — WMI ma takie ograniczenie od najwcześniejszej implementacji. Funkcja zwraca cały obiekt zarządzania `Win32_OperatingSystem`. Jego własności `ProductType` można użyć do odróżnienia stacji roboczej od serwera. Jej możliwe wartości są przedstawione w tabeli 6.1.

TABELA 6.1. Wartości własności `ProductType` obiektu klasy `Win32_OperatingSystem` i ich znaczenie

| Wartość | Opis |
|---------|------------------|
| 1 | Stacja robocza |
| 2 | Kontroler domeny |
| 3 | Serwer |

Po sprawdzeniu wersji systemu operacyjnego można wybrać, ile informacji zwrócić. Na przykład własność `caption` zwraca łańcuch identyfikujący system operacyjny, ale jeśli trzeba podjąć decyzję, lepszym rozwiązaniem może być ewaluacja rzeczywistego numeru wersji. Poniżej znajduje się kompletny kod źródłowy skryptu *Get-Version.ps1*.

Get-Version.ps1

```
Function Get-Version
{
    <#
        .Synopsis
            Zwraca informacje o systemie operacyjnym lokalnego lub zdalnego komputera
        .Description
```

```

Funkcja zwracająca informacje o systemie operacyjnym lokalnego lub zdalnego komputera
.Example
Get-Version -computername client1, server1 -credential (Get-Credential iammred\administrator)
Zwraca informacje o systemie operacyjnym dwóch zdalnych komputerów przy użyciu danych
poświadczających podanych przy uruchamianiu funkcji
.Example
$cred = Get-Credential iammred\administrator
Get-Version -computername client1, server1, edlt -credential $cred | select caption, version
Zwraca wartości własności caption i version z dwóch zdalnych komputerów przy użyciu danych
poświadczających zapisanych w zmiennej
.Parameter Computername
Nazwa komputera docelowego lub komputerów docelowych
.Parameter Credential
Dane poświadczające potrzebne do nawiązania połączenia
.Notes
NAME: Get-Version
AUTHOR: ed wilson, msft
LASTEDIT: 08/26/2013 13:25:38
KEYWORDS: CIM, OS
HSG:
.Link
Http://www.ScriptingGuys.com
#Requires -Version 2.0
#>
Param([string[]]$computername,
[System.Management.Automation.PSCredential]$credential)
$cim = New-CimSession @PSBoundParameters
Get-CimInstance -CimSession $cim -ClassName Win32_OperatingSystem
}

```

Po uruchomieniu skrypt ten ładuje do pamięci funkcję `Get-Version`. Funkcja ta zwraca cały obiekt zarządzania `Win32_OperatingSystem`, więc trzeba przetworzyć otrzymane informacje w kontekście swojej aplikacji.

Wiedza tajemna

Georges Maheu, Security Premier Field Engineer

Microsoft Corporation, Kanada

Większość czasu pracy z konsolą Windows PowerShell poświęcam na pisanie skryptów. Chociaż czasami konsoli tej używam też do testowania składni i eksperymentowania. Należy jednak pamiętać, że obsługa tego narzędzia może być zarówno bardzo łatwa (gdy używa się prostych poleceń bez parametrów), jak i niezwykle skomplikowana, gdy użyje się rozbudowanych skryptów. Pisząc skrypty, należy zawsze zachować szczególną ostrożność, ponieważ to, co wygląda jak prosty kawałek kodu, może zablokować procesor. Na przykład ten skrypt może spowodować 100-procentowe wykorzystanie pojedynczego procesora w komputerze: `1..2000 | ForEach-Object {Get-WmiObject win32_bios};`. Natomiast ten skrypt nie stwarza takiego ryzyka: `1..2000 | ForEach-Object {Get-WmiObject win32_bios;start-sleep -milli 2}`. Żle napisany skrypt może łatwo przekroczyć możliwości starego serwera.

Brak obsługi WMI

Instrumentacja zarządzania Windows (ang. *Windows Management Instrumentation* — WMI) to rozszerzenie obecne w systemie Windows od wersji NT 4.0. W każdej kolejnej wersji systemu dodawano do niego nowe klasy i od czasu do czasu wzbogacano istniejące klasy o nowe metody. Jedną z zalet WMI jest w miarę spójne podejście do pracy z oprogramowaniem i sprzętem. Inną zaletą jest to, że WMI jest dobrze znaną technologią, dzięki czemu w internecie można znaleźć wiele przykładowych skryptów. Dzięki udoskonaleniu obsługi WMI w konsoli Windows PowerShell 2.0, wprowadzeniu poleceń CIM w Windows PowerShell 3.0 oraz rozszerzonemu wsparciu CIM w systemach Windows 8.1 i Windows Server 2012 R2 konsola Windows PowerShell praktycznie dorównuje możliwościami językowi VBScript. W istocie dzięki funkcjom opartym na CIM wiele czynności konfiguracyjnych można wykonać przy użyciu pojedynczego polecenia wiersza poleceń. Zanim przejdziemy do praktycznego omówienia zasad posługiwania się klasami WMI z poziomu konsoli Windows PowerShell, musimy poznać podstawy technologii WMI.

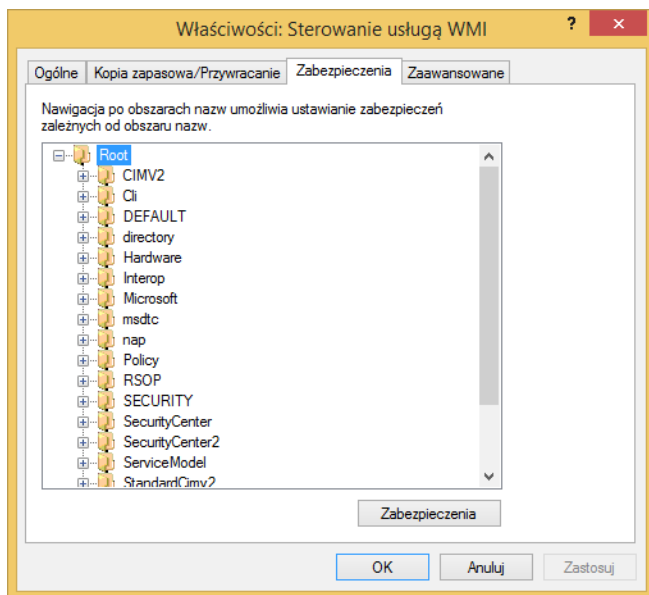
Czasami mówi się, że WMI to hierarchiczna **przestrzeń nazw**. To określenie bierze się stąd, że poszczególne warstwy są budowane jedne na drugich, podobnie jak katalog LDAP (ang. *Lightweight Directory Access Protocol*) w Active Directory lub struktura systemu plików na dysku twardym komputera. Ale mimo że w rzeczywistości WMI jest hierarchiczną przestrzenią nazw, określenie to nie oddaje w pełni bogactwa możliwości tej technologii. Model WMI składa się z trzech części: zasobów, infrastruktury oraz konsumentów. Poniżej znajduje się ich bardziej szczegółowy opis.

- **Zasoby WMI** — do zasobów zalicza się wszystko, do czego można uzyskać dostęp za pomocą WMI: system plików, składniki sieciowe, dzienniki zdarzeń, pliki, foldery, dyski, Active Directory itd.
- **Infrastruktura WMI** — infrastrukturę tworzą trzy składniki: usługa WMI, repozytorium WMI oraz dostawcy WMI. Z tych trzech składników najważniejsi są dostawcy, ponieważ dzięki nim WMI może zbierać potrzebne informacje.
- **Konsument WMI** — konsument „konsumuje” dane z WMI. Może być skryptem VBScript, programem do zarządzania firmą albo innym narzędziem wykonującym zapytania WMI.

Praca z obiektami i przestrzeniami nazw

Wróćmy do wprowadzonego w poprzednim podrozdziale pojęcia przestrzeni nazw. **Przestrzeń nazw** można traktować jak sposób organizacji lub gromadzenia danych odnoszących się do podobnych elementów. Wyobraź sobie staroświecką szafę na dokumenty. Każda szuflada tej szafy reprezentuje jedną przestrzeń nazw. W każdej szufladzie znajdują się teczki zawierające pewien podzbiór wszystkich zawartych w szafie informacji. Na przykład w moim domu znajduje się szafa zawierająca szufladę przeznaczoną do przechowywania informacji o moich narzędziach stolarskich. W szufladzie tej można znaleźć teczkę z instrukcjami obsługi piły stołowej, hebla, ścisku stolarskiego, odkurzacza itd. W teczce piły stołowej znajdują się informacje o silniku, ostrzach oraz różnych dokupionych akcesoriach (np. osłonie ostrza).

W podobny sposób zorganizowana jest przestrzeń nazw WMI. Przestrzenie nazw są szafami na dokumenty. Szuflady są dostawcami. Teczki są klasami WMI. Strukturę tych przestrzeni nazw pokazano na rysunku 6.1.



RYСУNEK 6.1. Przestrzeń nazw WMI w systemie Windows 8.1

Przestrzenie nazw zawierają obiekty. Obiekty z kolei zawierają własności, przy których można szperać. Użyjemy jednego z poleceń WMI, aby zobaczyć, jak jest zorganizowana ta przestrzeń nazw. Polecenie `Get-WmiObject` służy do łączenia się z WMI. W argumencie `class` określono klasę `__Namespace`, a w argumencie `namespace` określono poziom hierarchii przestrzeni nazw. Opisywane polecenie jest pokazane poniżej:

```
Get-WmiObject -class __Namespace -namespace root |  
Select-Object -property name
```

Wykonanie tego kodu w systemie Windows Vista spowoduje pojawienie się następujących informacji:

```
name  
----  
subscription  
DEFAULT  
MicrosoftDfs  
CIMV2  
Cli  
nap  
SECURITY  
SecurityCenter2  
RSOP  
WMI  
directory  
Policy
```



```
ServiceModel
SecurityCenter
Microsoft
aspnet
```

Za pomocą skryptu *RecursiveWMINameSpaceListing.ps1* możesz sprawdzić, jak wiele różnorodnych przestrzeni nazw WMI znajduje się w Twoim komputerze. Przy okazji sporo się dowiesz na temat tej technologii. Poniżej znajduje się cały kod źródłowy tego skryptu.

RecursiveWMINameSpaceListing.ps1

```
Function Get-WmiNameSpace($namespace, $computer)
{
    Get-WmiObject -class __NameSpace -computer $computer '
    -namespace $namespace -ErrorAction "SilentlyContinue" |
    Foreach-Object '
    -Process '
    {
        $subns = Join-Path -Path $_.__namespace -ChildPath $_.name
        $subns
        $script:i ++
        Get-WmiNameSpace -namespace $subNS -computer $computer
    }
} #end Get-WmiNameSpace

# *** punkt wejściowy ***

$script:i = 0
$namespace = "root"
$computer = "LocalHost"
"Pobieranie przestrzeni nazw WMI z komputera $computer..."
Get-WmiNameSpace -namespace $namespace -computer $computer
"Liczba przestrzeni nazw na komputerze $computer: $script:i"
```

Poniżej pokazano wynik zwrócony przez skrypt *RecursiveWMINameSpaceListing.ps1* na tym samym komputerze z systemem Windows Vista co poprzednio. Jak widać, hierarchia przestrzeni nazw we współczesnych systemach operacyjnych jest dość mocno rozbudowana.

```
Pobieranie przestrzeni nazw WMI z komputera LocalHost ...
ROOT\subscription
ROOT\subscription\ms_409
ROOT\DEFAULT
ROOT\DEFAULT\ms_409
ROOT\MicrosoftDfs
ROOT\MicrosoftDfs\ms_409
ROOT\CIMV2
ROOT\CIMV2\Security
ROOT\CIMV2\Security\MicrosoftTpm
ROOT\CIMV2\ms_409
ROOT\CIMV2\TerminalServices
ROOT\CIMV2\TerminalServices\ms_409
ROOT\CIMV2\Applications
ROOT\CIMV2\Applications\Games
ROOT\Cli
ROOT\Cli\MS_409
ROOT\nap
```

```

ROOT\SECURITY
ROOT\SecurityCenter2
ROOT\RSOP
ROOT\RSOP\User
ROOT\RSOP\User\S_1_5_21_540299044_341859138_929407116_1133
ROOT\RSOP\User\S_1_5_21_540299044_341859138_929407116_1129
ROOT\RSOP\User\S_1_5_21_540299044_341859138_929407116_1118
ROOT\RSOP\User\S_1_5_21_918056312_2952985149_2686913973_500
ROOT\RSOP\User\S_1_5_21_135816822_1724403450_2350888535_500
ROOT\RSOP\User\ms_409
ROOT\RSOP\User\S_1_5_21_540299044_341859138_929407116_500
ROOT\RSOP\Computer
ROOT\RSOP\Computer\ms_409
ROOT\WMI
ROOT\WMI\ms_409
ROOT\directory
ROOT\directory\LDAP
ROOT\directory\LDAP\ms_409
ROOT\Policy
ROOT\Policy\ms_409
ROOT\ServiceModel
ROOT\SecurityCenter
ROOT\Microsoft
ROOT\Microsoft\HomeNet
ROOT\aspnet
Liczba przestrzeni nazw na komputerze LocalHost: 42

```

Co to wszystko znaczy? Że na komputerze z systemem Windows Vista jest kilkadziesiąt różnych przestrzeni nazw, z których można pobierać informacje na temat tego komputera. Uświadomienie sobie istnienia tych wszystkich przestrzeni jest pierwszym krokiem do rozpoczęcia korzystania z nich w celu zdobycia potrzebnych danych. Studenci i inne początkujące osoby uczą się obsługi WMI przez konsolę Windows PowerShell, pisząc skrypt wykonujący jakieś konkretne zadanie. Jednak większość z nich nie wie, z którą przestrzenią nazw się połączyć, aby zrobić to, co chce. Kiedy im podpowiadam, to często odpowiadają: „Dla pana to jest oczywiste, ale skąd ja mam wiedzieć, że taka przestrzeń nazw w ogóle istnieje?”. Za pomocą skryptu *RecursiveWMINameSpaceListing.ps1* można wygenerować listę wszystkich przestrzeni nazw dostępnych w komputerze i na podstawie zdobytych informacji poszukać dodatkowych informacji w portalu MSDN. A jeśli ktoś lubi pobawić się w badacza, to może przejść do następnego podrozdziału, zatytułowanego „Pobieranie listy dostawców WMI”.

Wiedza tajemna

Jason Walker, Premier Field Engineer
Microsoft Corporation

W pracy z klientami korporacyjnymi często muszę szukać rozwiązań dotyczących obsługi dużych ilości danych. Gdy w grę wchodzi duże zbiory informacji, na porządku dziennym są błędy w rodzaju `System.OutOfMemoryException`. Jak pisać skrypty, by nie napotykać takich błędów? Zawsze powtarzam klientom, aby nie bawili się z dużymi obiektami.

Dwa błędy, jakie najczęściej spotykam w skryptach, to zapisywanie obiektów, aby potem wyświetlić je wszystkie naraz, i gromadzenie wszystkich obiektów do przetworzenia i rozpoczęcie przetwarzania dopiero po zebraniu ich wszystkich.

Poniżej znajduje się przykład zapisania obiektów, aby je później wszystkie wyświetlić:

```
Results = @()
$Servers = Get-Content ServerList.txt

foreach($Server in $Servers)
{
    $wmi = Get-WmiObject -ComputerName $Server -Class Win32_OperatingSystem

    $Results += New-Object -TypeName PSObject -Property @{
        ComputerName = $wmi.__Server
        OperatingSystem = $wmi.caption
    }
}
```

\$Results

W każdej iteracji pętli foreach do zmiennej \$Results jest dodawana nowa wartość. Jeśli w zmiennej \$Servers znajduje się tylko dziesięć nazw komputerów, to nie ma problemu. Ale gdyby zmienna ta zawierała 1000 nazw serwerów, skrypt zużyłby bardzo dużo pamięci i wydawałoby się, że działa bardzo powoli, ponieważ wynik zostałby wyświetlony dopiero po skontaktowaniu się z ostatnim serwerem. Rozwiązanie tego problemu jest bardzo proste — wystarczy pozbyć się zmiennej \$Results:

```
$Servers = "mail01","mail02","mbx01","mbx02"

foreach($Server in $Servers)
{
    $wmi = Get-WmiObject -ComputerName $Server -Class Win32_OperatingSystem

    New-Object -TypeName PSObject -Property @{
        ComputerName = $wmi.__Server
        OperatingSystem = $wmi.caption
    }
}
```

Teraz skrypt będzie zwracał wyniki na bieżąco, dzięki czemu wyda się szybszy i zużyje bardzo mało pamięci.

Podobny efekt ma gromadzenie wszystkich obiektów do przetworzenia, zanim rozpocznie się przetwarzanie, tzn. nastąpi zużycie bardzo dużej ilości pamięci. Jeden z moich klientów wysyłał raporty o niedostarczeniu do zewnętrznych adresatów. Jego administrator systemu Exchange potrzebował skryptu dodającego adres SMTP do zewnętrznego kontaktu w celu wyeliminowania tych raportów. Napisany przez niego skrypt doskonale działał, gdy był uruchamiany dla pojedynczych kontaktów. Ale administrator chciał być sprytny i uruchomił go dla 100 000 kontaktów z książki adresowej, co spowodowało zajęcie całej pamięci serwera i jego zawieszenie. Rozwiązanie jest proste: niech skrypt przyjmuje dane z potoku. Dane z tego źródła są przetwarzane na bieżąco i usuwane, więc nie trzeba ich zapisywać w gigantycznej tablicy obiektów. Oto dwa proste przykłady na dowód moich twierdzeń:

Przykład 1: `ForEach-Object -InputObject (dir c:\ -Recurse) -Process {$_.Fullname}`

Przykład 2: `dir c:\ -Recurse | select Fullname`

W przykładzie 1. wszystkie ścieżki przed przetworzeniem w skrypcie muszą zostać zebrane w jednym miejscu. Gdy kod rozpocznie przetwarzanie, nastąpi zawieszenie aż do czasu zwrócenia wyniku. Natomiast w przykładzie 2. wyniki będą zwracane natychmiast.

Wracamy do przykładu klienta.

Poniższe rozwiązanie zużywa dużo zasobów:

```
Set-x500Address -Identity (Get-MailContact -ResultSize Unlimited)
```

W tym przypadku przed rozpoczęciem przetwarzania trzeba zapisać wszystkie 100 000 kontaktów w pamięci.

Poniższe rozwiązanie jest znacznie bardziej efektywne:

```
Get-MailContact -ResultSize Unlimited | Set-X500Address
```

Dzięki użyciu potoku przychodzące dane są przetwarzane na bieżąco.

Pisząc skrypty, należy mieć na uwadze, co dokładnie robi nasz kod, ponieważ konsola Windows PowerShell, jeśli tylko się jej na to pozwoli, potrafi zużyć ogromne ilości zasobów.

Pobieranie listy dostawców WMI

Wiedza na temat struktury przestrzeni nazw WMI jest administratorowi potrzebna do poprawnego posługiwania się skryptami w pracy. Ale jak wspomniałem wcześniej, aby dostać się do informacji poprzez WMI, należy mieć dostęp do dostawcy WMI. Dopiero po jego zaimplementowaniu można czerpać udostępnione informacje.

Wszyscy dostawcy WMI bazują na klasie szablonowej lub systemowej o nazwie `_provider`. Aby więc zdobyć listę wszystkich dostępnych dostawców WMI, można wyszukać egzemplarze tej klasy należące do przestrzeni nazw WMI. Dokładnie to robi skrypt *Get-WmiProviders.ps1*.

Na początku skryptu *Get-WmiProviders.ps1* znajduje się przypisanie łańcucha `"root\cimv2"` do zmiennej `$wmiNS`. Wartość ta jest używana w poleceniu `Get-WmiObject` do określenia, w którym miejscu ma zostać wykonane zapytanie WMI. Należy zauważyć, że `root\cimv2` to domyślna przestrzeń nazw WMI w każdym systemie operacyjnym Windows od wersji 2000.

Do odpytywania WMI służy polecenie `Get-WmiObject`. Za pomocą dostawcy klas ogranicza się zapytanie do klas pochodnych od klasy `_provider`. Argument `namespace` nakazuje poleceniu przeszukać tylko przestrzeni nazw WMI `root\cimv2`. Zwrócona przez to polecenie tablica obiektów zostaje przekazana potokowo do polecenia `Sort-Object` w celu posortowania według własności `name`. Posortowane obiekty są przekazywane do polecenia `Format-List` w celu wydrukowania nazw wszystkich dostawców. Poniżej znajduje się cały kod źródłowy skryptu *Get-WmiProviders.ps1*:

Get-WmiProviders.ps1

```
Function Get-WmiProviders(
    $namespace="root\cimv2",
    $computer="localhost"
)
{
    Get-WmiObject -class __Provider -namespace $namespace '
    -computername $computer |
```

```
Sort-Object -property Name |
Select-Object -property Name
} #end Get-WmiProviders
Get-WmiProviders
```

Praca z klasami WMI

Oprócz przestrzeni nazw ciekawski administrator na pewno zechce też zbadać klasy. W dialekcie WMI wyróżnia się klasy rdzenne, klasy pospolite oraz klasy dynamiczne. **Klasy rdzenne** (ang. *core class*) reprezentują obiekty zarządzane dotyczące wszystkich dziedzin zarządzania. Dostarczają one podstawowego słownika do analizowania i opisywania systemów zarządzanych. Do przykładów tych klas zaliczają się parametry i klasa `System.Security`. **Klasy pospolite** (ang. *common class*) to rozszerzenia klas rdzennych reprezentujące obiekty zarządzane dotyczące konkretnych dziedzin zarządzania. Mimo to klasy pospolite są niezależne od jakiejkolwiek implementacji lub technologii. Przykładem takiej klasy jest `CIM_UniformComputerSystem`. Administratorzy sieci nieczęsto używają klas rdzennych i pospolitych, ponieważ są one szablonami do tworzenia innych klas.

Z powyższego wynika, że przestrzeń nazw `root\cimv2` zawiera wiele klas abstrakcyjnych, których używa się jako szablonów. Ale jest w niej też parę klas dynamicznych służących do pobierania realnych informacji. W odniesieniu do **klas dynamicznych** należy tylko pamiętać, że ich egzemplarze są generowane przez dostawców, dzięki czemu powinny zwracać „żywe” dane z systemu.

Prostą listę klas WMI można wyświetlić za pomocą polecenia `Get-WmiObject` z argumentem `list`, jak pokazano poniżej:

```
Get-WmiObject -list
```

Poniżej pokazano część wyniku tego polecenia:

| | |
|-----------------------------------|------------------------------|
| Win32_TSGeneralSetting | Win32_TSPermissionsSetting |
| Win32_TSClientSetting | Win32_TSEnvironmentSetting |
| Win32_TSNetworkAdapterListSetting | Win32_TSLogonSetting |
| Win32_TSSessionSetting | Win32_DisplayConfiguration |
| Win32_COMSetting | Win32_ClassicCOMClassSetting |
| Win32_DCOMApplicationSetting | Win32_MSISResource |
| Win32_ServiceControl | Win32_Property |

Zapiski praktyka

Praca z usługami

Clint Huffman, Senior Premier Field Engineer (PFE)

Microsoft Corporation

Dużo podróżuję, a niestety akumulator w moim laptopie nie trzyma zbyt długo. Dlatego sporo czasu poświęciłem na znalezienie w moim komputerze wszystkich usług wykonujących operacje wejścia i wyjścia na dysku twardym, które mogą pobierać najwięcej energii, nie licząc

monitora. Określiłem wiele usług, które podczas lotu nie są mi potrzebne, np. oprogramowanie antywirusowe, Usługa wyszukiwania systemu Windows, usługa Pliki trybu offline, ReadyBoost itd. Ponieważ dość często musiałem wyłączać i włączać te usługi, postanowiłem napisać skrypt, który robiłby to za mnie.

WMI to potężny model obiektowy, dzięki któremu za pomocą języków skryptowych, takich jak VBScript i Windows PowerShell, można wykonywać zadania, które kiedyś mogli robić tylko doświadczeni programiści języka C++. Co więcej, dzięki temu modelowi nie trzeba pisać tak dużo kodu źródłowego jak kiedyś, ponieważ skrypty znacznie ułatwiają automatyzację.

Wracając do opisywanego skryptu: na początku muszę wybrać odpowiednie usługi. W WMI używana jest podobna do języka SQL składnia o nazwie WMI Query Language (WQL). Nie nazywa się SQL, ponieważ zawiera parę dziwactw ściśle związanych z WMI. Chcę, aby moje zapytanie WQL zwracało te usługi systemu Windows, które wcześniej wytypowałem jako najbardziej energochłonne, takie jak usługa Pliki trybu offline, usługa ReadyBoost, usługi programu antywirusowego zaczynające się od słów *Microsoft Forefront* (Microsoft Forefront Client Security Antimalware Service i Microsoft Forefront Client Security State Assessment Service) i w końcu osobisty indeksator plików Usługa wyszukiwania systemu Windows.

```
$WQL = "SELECT Name, State, Caption FROM Win32_Service WHERE Caption LIKE
'Microsoft Forefront%' OR Name = 'WSearch' OR Caption = 'Offline files'
OR Caption = 'ReadyBoost'"
Get-WmiObject -Query $WQL
```

W moim komputerze powyższy skrypt zwraca następujące usługi:

```
Offline Files
ReadyBoost
Microsoft Forefront Client Security Antimalware Service
Microsoft Forefront Client Security State Assessment Service
Windows Search
```

Własność *Caption* to tekst, który widać w Panelu sterowania, w oknie *Usługi*, a własność *name* to krótka nazwa usługi, którą możesz lepiej znać, jeśli używasz narzędzi wiersza poleceń *Net Start* i *Net Stop*. Natomiast własność *State* informuje o tym, czy dana usługa jest uruchomiona.

Klauzula *WHERE* umożliwia ograniczenie zbioru zwracanych informacji. Gdybym jej nie użył, otrzymałbym wszystkie usługi jako obiekty. Byłoby to dobre, gdybym chciał się dowiedzieć, jakie w ogóle usługi znajdują się w komputerze, ale nie jest zbyt przydatne, gdy chcę po prostu zamknąć parę z nich. Więcej informacji na temat języka WQL można znaleźć w artykule pt. „*Quering with WQL*” na stronie <http://msdn.microsoft.com/en-us/library/aa392902.aspx>.

Jako że parametr *Query* zawsze zwraca obiekt będący **kolekcją**, muszę przejrzeć tę kolekcję w celu dostania się do każdego obiektu z osobna. Proces ten jest podobny do otrzymania paczki pocztą: zanim będę mógł użyć tego, co znajduje się w opakowaniu, muszę zdjąć to opakowanie. Do tego celu używam instrukcji sterującej *Foreach*. Za jej pomocą mogę pracować z każdym obiektem (np. usługą) po kolei, tak jak mogę wyjmować z pudełka pojedyncze przedmioty. W tym przypadku przekazuję wartość zwrótną polecenia *Get-WmiObject* do zmiennej o nazwie *\$CollectionOfServices* (mojego pudełka). Następnie za pomocą instrukcji *Foreach* wykonuję odpowiednie czynności na każdej usłudze w ten sposób, że w każdej iteracji zmienna *\$Service* reprezentuje inny obiekt usługi. Poniżej znajduje się powyższy kod źródłowy z dodatkiem instrukcji *Foreach*:

```
$WQL = "SELECT Name, State, Caption FROM Win32_Service WHERE Caption LIKE
'Microsoft ForeFront%' OR Name = 'WSearch' OR Caption = 'Offline Files'
OR Caption = 'ReadyBoost'"
$CollectionOfServices = Get-WmiObject -Query $WQL
Foreach ($Service in $CollectionOfServices)
{
    $Service.Caption
}
```

Po wybraniu konkretnych usług, które chcę zamknąć, mogę rzeczywiście je pozamykać. Do tego używam metody `StopService()` w sposób pokazany poniżej:

```
$WQL = "SELECT Name, State, Caption FROM Win32_Service WHERE Caption
LIKE 'Microsoft ForeFront%' OR Name = 'WSearch' OR Caption = 'Offline Files' OR Caption =
'ReadyBoost'"
$CollectionOfServices = Get-WmiObject -Query $WQL
Foreach ($Service in $CollectionOfServices)
{
    $Service.Caption
    $Service.StopService()
}
```

Jeśli usługi nie zostaną zatrzymane, to najczęściej z tego powodu, że nie mam uprawnień administratora, a jeśli pracuję w systemie Windows Vista, to dlatego, że muszę uruchomić skrypt w konsoli PowerShell o zwiększonych uprawnieniach. Aby uruchomić konsolę Windows PowerShell ze zwiększonymi uprawnieniami, klikam prawym przyciskiem myszy jej ikonę i z menu, które się pojawia, wybieram opcję *Uruchom jako administrator*. Potem jeszcze raz próbuję wykonać skrypt.

Świetnie! Wszystkie niepotrzebne usługi zostały zatrzymane. Ale niektóre z nich potrafią być namolne i z powrotem się uruchamiają przy pierwszej nadarzającej się okazji. Jak im to uniemożliwić? Trzeba je całkowicie wyłączyć. W jaki sposób? Za pomocą metody `ChangeStartMode()` z argumentem `Disabled`, jak pokazano poniżej:

```
$WQL = "SELECT Name, State, Caption FROM Win32_Service WHERE Caption LIKE
'Microsoft ForeFront%' OR Name = 'WSearch' OR Caption = 'Offline Files'
OR Caption = 'ReadyBoost'"
$CollectionOfServices = Get-WmiObject -Query $WQL
Foreach ($Service in $CollectionOfServices)
{
    $Service.Caption
    $Service.StopService()
    $Service.ChangeStartMode("Disabled")
}
```

To rozumiem! Teraz te nieznosne usługi są uziemione na dobre.

Pobawiłem się, doleciałem na miejsce i muszę połączyć się z siecią firmową. Zabezpieczenia w tej sieci pozwalają na podłączanie tylko komputerów z włączonym oprogramowaniem antywirusowym. Nie ma problemu. Wystarczy dwie drobne zmiany w skrypcie i usługi z powrotem działają:

```
$WQL = "SELECT Name, State, Caption FROM Win32_Service WHERE Caption LIKE
'Microsoft ForeFront%' OR Name = 'WSearch' OR Caption = 'Offline Files'
OR Caption = 'ReadyBoost'"
$CollectionOfServices = Get-WmiObject -Query $WQL
Foreach ($Service in $CollectionOfServices)
```

```
{
    $Service.Caption
    $Service.StartService()
    $Service.ChangeStartMode("Automatic")
}
```

Zamieniłem metodę `StopService()` na `StartService()` i argument metody `ChangeStartMode()` na `Automatic`.

Pewnie myślisz, że wszystkie te zabiegi znacznie przedłużają czas pracy na akumulatorze, ale czy zastanawiałeś się nad masowym wyłączaniem i włączaniem usług? Jedną z ciekawych modyfikacji tego skryptu jest przystosowanie go do współpracy ze zdalnymi serwerami. Załóżmy na przykład, że musimy uruchomić ponownie usługi w farmie 10 serwerów sieciowych. W tym celu wystarczy tylko wprowadzić drobną modyfikację do naszego skryptu — dodać argument `-ComputerName`.

```
$WQL = "SELECT Name, State, Caption FROM Win32_Service WHERE Caption LIKE
'Microsoft ForeFront%' OR Name = 'WSearch' OR Caption = 'Offline Files'
OR Caption = 'ReadyBoost'"
$CollectionOfServices = Get-WmiObject -Query $WQL -ComputerName
demoserver
Foreach ($Service in $CollectionOfServices)
{
    $Service.Caption
    $Service.StartService()
    $Service.ChangeStartMode("Automatic")
}
```

Te skrypty bardzo dobrze mi służą i mam nadzieję, że okażą się przydatne również Tobie.

Zmienianie ustawień

Mimo niewątpliwych korzyści wynikających z używania WMI technologia ta ma parę irytujących ograniczeń. Do pobierania informacji WMI nadaje się doskonale, ale gorzej jest z ich zmienianiem. Poniższy przykład ilustruje, co mam na myśli. Klasa WMI `Win32_Desktop` dostarcza informacji na temat ustawień pulpitu, jak widać poniżej:

```
PS C:\> Get-WmiObject Win32_Desktop
__GENUS           : 2
__CLASS           : Win32_Desktop
__SUPERCLASS      : CIM_Setting
__DYNASTY         : CIM_Setting
__RELPATH         : Win32_Desktop.Name="NT AUTHORITY\SYSTEM"
__PROPERTY_COUNT  : 21
__DERIVATION      : {CIM_Setting}
__SERVER          : MRED1
__NAMESPACE      : root\cimv2
__PATH           : \\MRED1\root\cimv2:Win32_Desktop.Name="NT AUTHORITY\SYSTEM"
BorderWidth       : 1
Caption          :
CoolSwitch        :
CursorBlinkRate   : 500
Description       :
```



```

DragFullWindows      : True
GridGranularity      :
IconSpacing           :
IconTitleFaceName     : Segoe UI
IconTitleSize         : 9
IconTitleWrap         : True
Name                  : NT AUTHORITY\SYSTEM
Pattern               : (None)
ScreenSaverActive      : True
ScreenSaverExecutable : C:\Windows\system32\logon.scr
ScreenSaverSecure      : True
ScreenSaverTimeout    : 600
SettingID             :
Wallpaper             :
WallpaperStretched    : False
WallpaperTiled        :

```

Jak widać, polecenie `Get-WmiObject` zwraca wiele cennych informacji. Dla administratorów bardzo ważne jest np. po jakim czasie włącza się wygaszacz ekranu oraz czy pulpit jest chroniony hasłem. Wprawdzie ustawienia te można i powinno się definiować poprzez zasady grup, ale czasami administrator sieci również chciałby je zmienić za pomocą skryptu. Jeśli zbadasz własności klasy `Win32_Desktop` za pomocą polecenia `Get-Member`, otrzymasz następujące informacje:

```

PS C:\> Get-WmiObject Win32_Desktop | Get-Member
      TypeName: System.Management.ManagementObject#root\cimv2\Win32_Desktop

```

| Name | MemberType | Definition |
|-----------------------|------------|---|
| ---- | ----- | ----- |
| BorderWidth | Property | System.UInt32 BorderWidth {get;set;} |
| Caption | Property | System.String Caption {get;set;} |
| CoolSwitch | Property | System.Boolean CoolSwitch {get;set;} |
| CursorBlinkRate | Property | System.UInt32 CursorBlinkRate {get;set;} |
| Description | Property | System.String Description {get;set;} |
| DragFullWindows | Property | System.Boolean DragFullWindows {get;set;} |
| GridGranularity | Property | System.UInt32 GridGranularity {get;set;} |
| IconSpacing | Property | System.UInt32 IconSpacing {get;set;} |
| IconTitleFaceName | Property | System.String IconTitleFaceName {get;set;} |
| IconTitleSize | Property | System.UInt32 IconTitleSize {get;set;} |
| IconTitleWrap | Property | System.Boolean IconTitleWrap {get;set;} |
| Name | Property | System.String Name {get;set;} |
| Pattern | Property | System.String Pattern {get;set;} |
| ScreenSaverActive | Property | System.Boolean ScreenSaverActive {get;set;} |
| ScreenSaverExecutable | Property | System.String ScreenSaverExecutable {get;...} |
| ScreenSaverSecure | Property | System.Boolean ScreenSaverSecure {get;set;} |
| ScreenSaverTimeout | Property | System.UInt32 ScreenSaverTimeout {get;set;} |
| SettingID | Property | System.String SettingID {get;set;} |
| Wallpaper | Property | System.String Wallpaper {get;set;} |
| WallpaperStretched | Property | System.Boolean WallpaperStretched {get;set;} |
| WallpaperTiled | Property | System.Boolean WallpaperTiled {get;set;} |
| __CLASS | Property | System.String __CLASS {get;set;} |
| __DERIVATION | Property | System.String[] __DERIVATION {get;set;} |
| __DYNASTY | Property | System.String __DYNASTY {get;set;} |
| __GENUS | Property | System.Int32 __GENUS {get;set;} |
| __NAMESPACE | Property | System.String __NAMESPACE {get;set;} |
| __PATH | Property | System.String __PATH {get;set;} |
| __PROPERTY_COUNT | Property | System.Int32 __PROPERTY_COUNT {get;set;} |
| __RELPATH | Property | System.String __RELPATH {get;set;} |
| __SERVER | Property | System.String __SERVER {get;set;} |

```
__SUPERCLASS      Property      System.String __SUPERCLASS {get;set;}
ConvertFromDateTime ScriptMethod System.Object ConvertFromDateTime();
ConvertToDateTime  ScriptMethod System.Object ConvertToDateTime();
```

Jeśli użyje się parametru `-filter` do pobrania określonego egzemplarza klasy WMI `Win32_Desktop` i zapisania go w zmiennej, później można bezpośrednio używać własności tej klasy. W poniższym przykładzie konieczne było zastosowanie sekwencji specjalnej dla ukośnika, który służy jako separator części NT `AUTHORITY` i `SYSTEM`:

```
PS C:\> $desktop = Get-WmiObject Win32_Desktop -Filter '
>> "name = 'NT AUTHORITY\SYSTEM'"
```

Po otrzymaniu dostępu do egzemplarza klasy WMI można przypisać nową wartość parametrowi `ScreenSaverTimeout`. Jak widać poniżej, wartość zostaje zmieniona natychmiast:

```
PS C:\> $Desktop.ScreenSaverTimeout = 300
PS C:\> $Desktop.ScreenSaverTimeout
300
```

Ale jeśli jeszcze raz wykonamy zapytanie WMI, własność `ScreenSaverTimeout` nie zostanie już zmieniona. Metody pobierające i ustawiające zgłaszane przez polecenie `Get-Member` dotyczą kopii obiektu zwróconej przez zapytanie WMI, a nie rzeczywistego egzemplarza obiektu klasy WMI reprezentowanego przez tę klasę, jak pokazano poniżej:

```
PS C:\> $desktop = Get-WmiObject Win32_Desktop -Filter '
>> "name = 'NT AUTHORITY\SYSTEM'"
>>
PS C:\> $Desktop.ScreenSaverTimeout
600
```

Modyfikowanie wartości przez rejestr

W skrypcie *Set-SaverTimeout.ps1* używane są trzy parametry, ale tylko jeden z nich jest często modyfikowany — `timeoutvalue`. Parametr ten ustawia wartość limitu czasu wygaszacza ekranu. Ponieważ funkcja `Set-ScreenSaverTimeout` używa atrybutu `[cmdletbinding()]`, otrzymujemy dostęp do często używanych parametrów, takich jak `verbose`. Zwróć uwagę, że nie trzeba robić już nic więcej, aby otrzymać dostęp do tych parametrów. Polecenie `Write-Verbose` wyświetla rozszerzony strumień tylko wtedy, gdy funkcja jest wywoływana z przełącznikiem `-Verbose`. Jeśli w wywołaniu funkcji nie ma tego przełącznika, nie zostaje wyświetlony rozszerzony strumień. To oznacza, że można użyć funkcji `Set-ScreenSaverTimeout` i nic nie wyświetlić — jest to idealne rozwiązanie na przykład w skryptach logowania.

Jako że funkcja ta modyfikuje rejestr, dobrym pomysłem jest użycie **transakcji**. Dzięki temu jeśli nie uda się w całości wprowadzić zmian, można cofnąć to, co zostało zrobione. Transakcje rozpoczyna się bardzo łatwo. Służy do tego polecenie `Start-Transaction`,

```
Start-Transaction
```

Następnie, podczas inicjowania zmiany, należy użyć przełącznika `-UseTransaction`. Poniżej znajduje się polecenie z tym przełącznikiem:

```
Set-ItemProperty -Path $path -name $name -value $timeoutValue -UseTransaction
```

Ta konkretna funkcja nie cofa transakcji, ale w razie potrzeby łatwo dałoby się to zrobić. Dzięki użyciu przełącznika `-Verbose` wraz ze zmienioną wartością limitu czasu wygaszacza ekranu wyświetlona zostanie także wartość początkowa. Aby dowiedzieć się, jaka będzie nowa wartość po zakończeniu transakcji, należy użyć przełącznika `-UseTransaction` w poleceniu `Get-ItemProperty`. Poniżej znajduje się przykład:

```
Get-ItemProperty -path $spath -name $name -UseTransaction
```

Jeśli wszystko pójdzie zgodnie z planem, można użyć polecenia `Complete-Transaction`, jak pokazano poniżej:

```
Complete-Transaction
```

Poniżej pokazano przykład wywołania funkcji z parametrem `-Verbose`:

```
Set-ScreenSaverTimeout -timeoutValue 600 -Verbose
```

Wynik wykonania tej funkcji pokazano na rysunku 6.2.

```

1 Function Set-ScreenSaverTimeout
2 {
3     [cmdletbinding()]
4     Param(
5         [int]$timeoutValue = 600,
6         [string]$spath = 'HKCU:\Control Panel\Desktop',
7         [string]$name = 'ScreenSaverTimeout' )
8     Write-Verbose "Wywołano funkcję $($MyInvocation.MyCommand.name)"
9     Write-Verbose "Aktualna wartość własności $name $($Get-ItemProperty -path $spath -name $name).$name)"
10    Start-Transaction
11    Set-ItemProperty -Path $spath -name $name -value $timeoutValue -UseTransaction
12    Write-Verbose "Nowa wartość własności $name $($Get-ItemProperty -path $spath -name $name -UseTransaction).$name)"
13    Complete-Transaction
14 } #end Set-ScreenSaverTimeout

```

```

PS C:\> Set-ScreenSaverTimeout -timeoutValue 600 -Verbose
VERBOSE: Wywołano funkcję Set-ScreenSaverTimeout
VERBOSE: Aktualna wartość własności ScreenSaverTimeout 550
VERBOSE: Nowa wartość własności ScreenSaverTimeout 600
PS C:\>

```

RYСУNEK 6.2. Dzięki implementacji parametru `verbose` łatwo można uzyskać szczegółowe informacje

Zamiast polecenia `Write-Verbose` można było użyć polecenia `Write-Debug`, które automatycznie formatuje tekst oraz koloruje go na żółto i czarno (ale można zmienić te ustawienia), a także zapisuje tekst w konsoli tylko wtedy, gdy tego zażądamy. Domyślnie polecenie `Write-Debug` nic nie drukuje w konsoli, dzięki czemu nie trzeba go usuwać ze skryptów przed przekazaniem ich do użytku. Zmienna automatyczna `$DebugPreference` służy do sterowania działaniem polecenia `Write-Debug`. Domyślnie jest ustawiona na `SilentlyContinue`, dzięki czemu gdy konsola Windows PowerShell napotka polecenie `Write-Debug`, pomija je lub po cichu przechodzi do następnego wiersza. Zmienną `$DebugPreference` można konfigurować za pomocą czterech wartości zdefiniowanych w klasie wyliczeniowej `System.Management.Automation.ActionPreference`. Wartości te można znaleźć w portalu MSDN oraz sprawdzić za pomocą statycznej metody `GetNames` klasy `.NET System.Enum`, jak pokazano poniżej:

```
PS C:\> [enum]::GetNames("System.Management.Automation.ActionPreference")
SilentlyContinue
Stop
Continue
Inquire
```

Polecenie Write-Debug służy do drukowania wartości własności name z obiektu System.Management.Automation.ScriptInfo. Obiekt ten pobiera się, wysyłając zapytanie do własności MyCommand klasy System.Management.Automation.InvocationInfo. Obiekt System.Management.Automation.InvocationInfo jest zwracany po wysłaniu zapytania do zmiennej automatycznej \$MyInvocation. Własności obiektu System.Management.Automation.InvocationInfo przedstawiono w tabeli 6.2.

TABELA 6.2. Własności klasy System.Management.Automation.InvocationInfo

| Własność | Definicja |
|------------------|---|
| BoundParameters | System.Collections.Generic.Dictionary'2[[System.String, mscorlib, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089],[System.Object, mscorlib, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089]] BoundParameters {get;} |
| CommandOrigin | System.Management.Automation.CommandOrigin CommandOrigin {get;} |
| ExpectingInput | System.Boolean ExpectingInput {get;} |
| InvocationName | System.String InvocationName {get;} |
| Line | System.String Line {get;} |
| MyCommand | System.Management.Automation.CommandInfo MyCommand {get;} |
| OffsetInLine | System.Int32 OffsetInLine {get;} |
| PipelineLength | System.Int32 PipelineLength {get;} |
| PipelinePosition | System.Int32 PipelinePosition {get;} |
| PositionMessage | System.String PositionMessage {get;} |
| ScriptLineNumber | System.Int32 ScriptLineNumber {get;} |
| ScriptName | System.String ScriptName {get;} |
| UnboundArguments | System.Collections.Generic.List'1[[System.Object, mscorlib, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089]] UnboundArguments {get;} |

Do polecenia Write-Debug można dodać każdą własność, którą uzna się za przydatną w rozwiązywaniu problemów. Własności te stają się jeszcze bardziej przydatne podczas pracy z obiektem System.Management.Automation.ScriptInfo, którego własności wymieniono w tabeli 6.3.

TABELA 6.3. Własności obiektu System.Management.Automation.ScriptInfo

| Własność | Definicja |
|---------------|--|
| CommandType | System.Management.Automation.CommandTypes CommandType {get;} |
| Defiition | System.String Defiition {get;} |
| Module | System.Management.Automation.PSModuleInfo Module {get;} |
| ModuleName | System.String ModuleName {get;} |
| Name | System.String Name {get;} |
| Parameters | System.Collections.Generic.Dictionary'2[[System.String, mscorlib, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089], [System.Management.Automation.ParameterMetadata, System.Management.Automation, Version=1.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35]] Parameters {get;} |
| ParameterSets | System.Collections.ObjectModel.ReadOnlyCollection'1 [[System.Management.Automation.CommandParameterSetInfo, System.Management.Automation, Version=1.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35]] ParameterSets {get;} |
| ScriptBlock | System.Management.Automation.ScriptBlock ScriptBlock {get;} |
| Visibility | System.Management.Automation.SessionStateEntryVisibility Visibility {get;set;} |

Poniżej znajduje się kompletny kod źródłowy skryptu *Set-SaverTimeOut.ps1*.

Set-SavertimeOut.ps1

```
Function Set-ScreenSaverTimeOut
{
    [cmdletbinding()]
    Param( [int]$timeoutValue = 600,
           [string]$path = 'HKCU:\Control Panel\Desktop',
           [string]$name = 'ScreenSaverTimeOut' )
    Write-Verbose "Wywołano funkcję $($MyInvocation.MyCommand.name)"
    Write-Verbose "Aktualna wartość własności $name $($Get-ItemProperty -path $path -name $name).$name)"
    Start-Transaction
    Set-ItemProperty -Path $path -name $name -value $timeoutValue -UseTransaction
    Write-Verbose "Nowa wartość własności $name $($Get-ItemProperty -path $path -name $name - UseTransaction).$name)"
    Complete-Transaction
} #end Set-ScreenSaverTimeOut
```

Brak obsługi platformy .NET

Możliwość pracy z platformą .NET z poziomu konsoli Windows PowerShell jest bardzo przydatna. Jako że sama konsola jest aplikacją .NET, komunikacja z tą platformą przebiega bardzo sprawnie i naturalnie. Czasami pytaniem nie jest, co można zrobić przy użyciu klas platformy .NET, tylko czego nie da się przy ich użyciu zrobić. Konstruktory niektórych klas platformy .NET są skomplikowane i trudne do zapamiętania. **Konstruktor** służy do tworzenia egzemplarza klasy. W wielu przypadkach, aby używać klasy, należy utworzyć jej egzemplarz. Czasami jednak konstruktor nie jest potrzebny i metody wywoływane bez tworzenia obiektu są statyczne.

Używanie statycznych metod i własności

Metody i własności statyczne to składowe, które są zawsze dostępne. Aby użyć statycznej metody, należy wpisać nazwę klasy w kwadratowym nawiasie, dwa dwukropki i dopiero po nich nazwę tej metody. Przykładem metody statycznej jest metoda `tan` z klasy `System.Math`, która służy do obliczania tangensa dla dowolnej liczby. Poniżej znajduje się przykład obliczenia za pomocą metody `tan` z klasy `System.Math` tangensa dla kąta 45 stopni:

```
PS C:\> [system.math]::tan(45)
1,61977519054386
```

W zbitce `system.math` część `system` reprezentuje przestrzeń nazw, w której znajduje się klasa `math`. W większości przypadków można pominąć słowo `system` bez uszczerbku dla działania procesu. Podczas pracy w wierszu poleceń można w ten sposób zmniejszyć trochę ilość tekstu do wpisania. Natomiast uważam, że w skryptach nigdy nie powinno się tego robić. Bez słowa `system` polecenie wygląda następująco:

```
PS C:\> [math]::tan(45)
1,61977519054386
```

Za pomocą polecenia `Get-Member` z parametrem `static` można sporządzić listę składowych klasy .NET `System.Math`. Poniżej znajduje się to polecenie w całej okazałości:

```
[math] | Get-Member -static
```

W tabeli 6.4 znajduje się zestawienie statycznych składowych klasy `System.Math`. Metody są bardzo ważne, ponieważ za ich pomocą można wykonać większość czynności obsługiwanych przez tę klasę. Na przykład w konsoli Windows PowerShell nie ma funkcji `tan`. W związku z tym, jeśli chce się obliczyć tangens jakiegoś kąta, należy użyć statycznej metody z klasy `System.Math` lub napisać własną funkcję. Do wykonywania działań matematycznych należy używać narzędzi platformy .NET, która nie jest żadnym obciążeniem — stanowi wartość dodaną, ponieważ jest dojrzałą i dobrze udokumentowaną technologią.

TABELA 6.4. Składowe klasy System.Math

| Nazwa | Typ składowej | Definicja |
|---------------|---------------|--|
| Abs | Metoda | static System.SByte Abs(SByte value) static System.Int16 Abs(Int16 value) static System.Int32 Abs(Int32 value) static System.Int64 Abs(Int64 value) static System.Single Abs(Single value) static System.Double Abs(Double value) static System.Decimal Abs(Decimal value) |
| Acos | Metoda | static System.Double Acos(Double d) |
| Asin | Metoda | static System.Double Asin(Double d) |
| Atan | Metoda | static System.Double Atan(Double d) |
| Atan2 | Metoda | static System.Double Atan2(Double y Double x) |
| BigMul | Metoda | static System.Int64 BigMul(Int32 a Int32 b) |
| Ceiling | Metoda | static System.Decimal Ceiling(Decimal d) static System.Double Ceiling(Double a) |
| Cos | Metoda | static System.Double Cos(Double d) |
| Cosh | Metoda | static System.Double Cosh(Double value) |
| DivRem | Metoda | static System.Int32 DivRem(Int32 a Int32 b Int32& result) static System.Int64 DivRem(Int64 a Int64 b Int64& result) |
| Equals | Metoda | static System.Boolean Equals(Object objA Object objB) |
| Exp | Metoda | static System.Double Exp(Double d) |
| Floor | Metoda | static System.Decimal Floor(Decimal d) static System.Double Floor(Double d) |
| IEEERemainder | Metoda | static System.Double IEEERemainder(Double x Double y) |
| Log | Metoda | static System.Double Log(Double d) static System.Double Log(Double a Double newBase) |
| Log10 | Metoda | static System.Double Log10(Double d) |
| Max | Metoda | static System.SByte Max(SByte val1 SByte val2) static System.Byte Max(Byte val1 Byte val2) static System.Int16 Max(Int16 val1 Int16 val2) static System.UInt16 Max(UInt16 val1 UInt16 val2) static System.Int32 Max(Int32 val1 Int32 val2) static System.UInt32 Max(UInt32 val1 UInt32 val2) static System.Int64 Max(Int64 val1 Int64 val2) static System.UInt64 Max(UInt64 val1 UInt64 val2) static System.Single Max(Single val1 Single val2) static System.Double Max(Double val1 Double val2) static System.Decimal Max(Decimal val1 Decimal val2) |

TABELA 6.4. Składowe klasy System.Math — ciąg dalszy

| Nazwa | Typ składowej | Definicja |
|----------------------|---------------|---|
| Min | Metoda | <code>static System.SByte Min(SByte val1 SByte val2) static System.Byte</code> <code>Min(Byte val1 Byte val2) static</code> <code>System.Int16 Min(Int16 val1 Int16 val2) static</code> <code>System.UInt16 Min(UInt16 val1 UInt16 val2) static</code> <code>System.Int32 Min(Int32 val1 Int32 val2) static</code> <code>System.UInt32 Min(UInt32 val1 UInt32 val2) static</code> <code>System.Int64 Min(Int64 val1 Int64 val2) static</code> <code>System.UInt64 Min(UInt64 val1 UInt64 val2) static</code> <code>System.Single Min(Single val1 Single val2) static</code> <code>System.Double Min(Double val1 Double val2) static System.Decimal</code> <code>Min(Decimal val1 Decimal val2)</code> |
| Pow | Metoda | <code>static System.Double Pow(Double x Double y)</code> |
| Reference ↪Equals | Metoda | <code>static System.Boolean ReferenceEquals(Object objA Object objB)</code> |
| Round | Metoda | <code>static System.Double Round(Double a) static</code> <code>System.Double Round(Double value Int32 digits) static</code> <code>System.Double</code> <code>Round(Double value MidpointRounding mode) static System</code> <code>.Double Round(Double value Int32 digits MidpointRounding mode)</code> <code>static System.Decimal Round(Decimal d) static System.Decimal</code> <code>Round(Decimal d Int32 decimals) static System.Decimal</code> <code>Round(Decimal</code> <code>d MidpointRounding mode) static System.Decimal Round(Decimal d</code> <code>Int32 decimals MidpointRounding mode)</code> |
| Sign | Metoda | <code>static System.Int32 Sign(SByte value) static System.Int32</code> <code>Sign(Int16 value)</code> <code>static System.Int32 Sign(Int32 value) static System.Int32</code> <code>Sign(Int64 value)</code> <code>static System.Int32 Sign(Single value) static System.Int32</code> <code>Sign(Double</code> <code>value) static System.Int32 Sign(Decimal value)</code> |
| Sin | Metoda | <code>static System.Double Sin(Double a)</code> |
| Sinh | Metoda | <code>static System.Double Sinh(Double value)</code> |
| Sqrt | Metoda | <code>static System.Double Sqrt(Double d)</code> |
| Tan | Metoda | <code>static System.Double Tan(Double a)</code> |

TABELA 6.4. Składowe klasy System.Math — ciąg dalszy

| Nazwa | Typ składowej | Definicja |
|----------|---------------|--|
| Tanh | Metoda | static System.Double Tanh(Double value) |
| Truncate | Metoda | static System.Decimal Truncate(Decimal d) static System.Double Truncate(Double d) |
| E | Własność | static System.Double E {get;} |
| PI | Własność | static System.Double PI {get;} |

Zależność od wersji

Jedną z wielu ciekawych cech platformy .NET jest to, że zawsze wydaje się, że jest nowa wersja i oczywiście między wersjami występują jeszcze dodatki serwisowe. Jednak choć sama platforma .NET jest częścią systemu operacyjnego, nie jest aktualizowana w dodatkach serwisowych. W związku z tym na administratora sieci spada obowiązek spakowania i wdrożenia aktualizacji platformy. Przed pojawieniem się konsoli Windows PowerShell administratorzy sieci niechętnie dokonywali aktualizacji, ponieważ nie mieli w tym żadnego interesu. Często wynikało to z braku zainteresowania lub niezrozumienia platformy .NET. Dopóki programiści nie zaczęli się domagać aktualizacji, platforma .NET nie była aktualizowana.

Brak obsługi COM

Wiele bardzo przydatnych funkcji występuje pod postacią składników COM (ang. *Component Object Model*), ale żeby je znaleźć, czasami trzeba mieć szczęście. Oczywiście zawsze można przeczytać dokumentację w portalu MSDN, ale w artykułach często brakuje identyfikatora programu potrzebnego do utworzenia danego obiektu COM i dotyczy to nawet artykułów odnoszących się do interfejsów skryptowych. Przykład tego można znaleźć w skryptowym modelu obiektowym programu Windows Media Player. Można przekopać całą dokumentację SDK (ang. *Software Development Kit*) i nie dowiedzieć się, że identyfikator tego programu to `wmplayer.ocx`, a nie `player`, którego używa się w celach demonstracyjnych. Najbardziej naturalnym sposobem pracy z obiektami COM w konsoli Windows PowerShell jest użycie polecenia `New-Object` z parametrem `-ComObject` o wartości będącej identyfikatorem wybranego programu. Ale jeśli nie znamy identyfikatora, to mamy problem. Możemy go znaleźć w rejestrze, co będzie wymagać trochę pracy detektywistycznej.

Przykładem obiektu COM, którego identyfikator programu trudno znaleźć, jest obiekt o identyfikatorze programu `makecab.makecab`. Obiekt ten służy do tworzenia **plików cabinet**, które są silnie skompresowanymi plikami używanymi przez programistów do instalowania programów. Nie ma żadnego powodu, dla którego administratorzy sieci mieliby nie używać tych plików do kompresowania dzienników przed ich przesłaniem. Jedyne problemy polegają na tym, że obiekt `makecab.makecab` jest dostępny tylko w systemach Windows XP i Windows Server 2003, a w systemie Windows Vista został usunięty i nie ma go też w nowszych wersjach. Dlatego podczas pracy z nowszymi systemami operacyjnymi należy zastosować inne podejście.

Aby skrypt był łatwy w obsłudze, najpierw za pomocą instrukcji `Param` należy utworzyć kilka parametrów wiersza poleceń. Instrukcja ta musi zajmować pierwszy niezawierający komentarza wiersz skryptu. Gdy skrypt zostanie uruchomiony w konsoli Windows PowerShell lub edytorze skryptów, parametry wiersza poleceń zostaną użyte do sterowania jego wykonywaniem. W ten sposób skrypt można uruchamiać bez modyfikowania go za każdym razem, gdy trzeba utworzyć plik *.cab* z innego katalogu. Wystarczy tylko przekazać nową ścieżkę do parametru `-filepath`, jak pokazano poniżej:

```
CreateCab.ps1 -filepath C:\fso1
```

Zaletą parametrów wiersza poleceń jest to, że podlegają funkcji uzupełniania częściowo wpisanych nazw, tzn. wystarczy wpisać tylko taką część ich nazw, aby nie dało się jej pomylić z niczym innym. Dzięki temu w wierszu poleceń można też wpisać polecenie o następującej składni:

```
CreateCab.ps1 -f c:\fso1 -p c:\fso2\bcab.cab -d
```

Powyższe polecenie przeszukuje katalog *c:\fso* i pobiera z niego wszystkie pliki. Następnie tworzy z nich plik cabinet o nazwie *bcab.cab*, który zapisuje w folderze *fso2* na dysku C. Dodatkowo polecenie to podczas pracy wyświetla dane diagnostyczne. Należy zauważyć, że parametr `debug` jest przełącznikiem, ponieważ ma wpływ na skrypt tylko wtedy, gdy jest wpisany. Poniżej znajduje się opisana do tej pory część skryptu.

```
Param(
    $filepath = "C:\fso",
    $path = "C:\fso\acab.cab",
    [switch]$debug
)
```

Teraz przejdziemy do tworzenia funkcji `New-Cab`, która będzie przyjmowała dwa parametry: `-path` i `-files`.

```
Function New-Cab($path,$files)
```

Identyfikator programu `makecab` można przypisać do zmiennej `$makecab`, aby trochę uprościć kod źródłowy skryptu. Ponadto w tym samym miejscu można umieścić pierwszą instrukcję `Write-Debug`.

```
{
    $makecab = "makecab.makecab"
    Write-Debug "Ścieżka tworzenia pliku Cab: $path"
```

Teraz musimy utworzyć obiekt COM:

```
$cab = New-Object -ComObject $makecab
```

Warto także zaimplementować podstawową ochronę przed błędami. Użyjemy do tego zmiennej automatycznej `??`.

```
if(!$?) { $(Throw "Nie udało się utworzyć obiektu $makecab.")}
```

Jeżeli podczas próby utworzenia obiektu `makecab` nie wystąpią żadne błędy, można użyć obiektu zapisanego w zmiennej `$cab` i wywołać metodę `createcab`:

```
$cab.CreateCab($path,$false,$false,$false)
```

Po utworzeniu pliku *.cab* należy dodać do niego pliki za pomocą instrukcji *Foreach*.

```
Foreach ($file in $files)
{
    $file = $file.fullname.toString()
    $fileName = Split-Path -path $file -leaf
```

Po zamienieniu pełnej nazwy pliku w łańcuch i usunięciu informacji dotyczących folderu za pomocą polecenia *Split-Path* należy dodać kolejną instrukcję *Write-Debug* informującą użytkownika o postępie działania skryptu, jak pokazano poniżej:

```
Write-Debug "Dodawanie z $file"
Write-Debug "Nazwa pliku to $fileName"
```

Następnie należy dodać plik do pliku cabinet.

```
$cab.AddFile($file,$filename)
}
Write-Debug "Zamykanie pliku .cab $path"
```

Plik cabinet można zamknąć za pomocą metody *closecab*.

```
$cab.CloseCab()
} #end New-Cab
```

Teraz czas przejść do punktu wejściowego skryptu. Najpierw trzeba dowiedzieć się, czy został uruchomiony w trybie diagnostycznym, sprawdzając, czy istnieje zmienna *\$debug*. Jeśli tak, należy ustawić zmienną *\$DebugPreference* na *continue*, co umożliwi instrukcjom *Write-Debug* drukowanie na ekranie. Domyślnie zmienna ta jest ustawiona na *SilentlyContinue*, przez co nie są wyświetlane żadne dane instrukcji diagnostycznych i konsola Windows PowerShell w ogóle ignoruje wszelkie instrukcje *Write-Debug*.

```
if($debug) {$DebugPreference = "continue"}
```

Teraz należy pobrać zbiór plików za pomocą polecenia *Get-ChildItem*.

```
$files = Get-ChildItem -path $filePath | Where-Object { !$_.psiscontainer }
```

Pobraną kolekcję plików można przekazać do funkcji *New-Cab*, jak pokazano poniżej:

```
New-Cab -path $path -files $files
```

Poniżej znajduje się kompletny kod skryptu *CreateCab.ps1*.

UWAGA

Skrypt *CreateCab.ps1* nie działa w systemie Windows Vista i nowszych wersjach systemu Windows z powodu braku w nich obsługi obiektu COM *makecab.makecab*. Alternatywny sposób tworzenia plików *.cab* opisano w sekcji „Brak obsługi aplikacji zewnętrznych”.

CreateCab.ps1

```
Param(
    $filepath = "C:\fso",
    $path = "C:\fso\acab.cab",
    [switch]$debug
)
```

```
Function New-Cab($path,$files)
{
    $makecab = "makecab.makecab"
    Write-Debug "Ścieżka tworzenia pliku Cab: $path"
    $cab = New-Object -ComObject $makecab
    if(!$?) { $(Throw "Nie udało się utworzyć obiektu $makecab.")}
    $cab.CreateCab($path,$false,$false,$false)
    Foreach ($file in $files)
    {
        $file = $file.fullname.toString()
        $fileName = Split-Path -path $file -leaf
        Write-Debug "Dodawanie z $file"
        Write-Debug "Nazwa pliku to $fileName"
        $cab.AddFile($file,$filename)
    }
    Write-Debug "Zamykanie pliku .cab $path"
    $cab.CloseCab()
} #end New-Cab

# *** punkt wejściowy skryptu ***
if($debug) {$DebugPreference = "continue"}
$files = Get-ChildItem -path $filePath | Where-Object { !$_.psiscontainer }
New-Cab -path $path -files $files
```

Za pomocą obiektu `makecab.makecab` nie można rozpakować pliku `cabinet`, ponieważ obiekt ten nie zawiera służącej do tego metody. Ponadto nie można użyć obiektu `makecab.expandcab`, ponieważ taki obiekt nie istnieje. Jako że funkcja rozpakowywania plików `cabinet` jest integralną częścią powłoki systemu Windows, do rozpakowywania tych plików można używać obiektu `shell`. Dostęp do niego można uzyskać przez obiekt `COM Shell.Application`.

Najpierw należy utworzyć parametry wiersza poleceń. Ta część skryptu jest bardzo podobna do sekcji parametrycznej w skrypcie *CreateCab.ps1*. Parametry te pokazano poniżej:

```
Param(
    $cab = "C:\fso\acab.cab",
    $destination = "C:\fso1",
    [switch]$debug
)
```

Po utworzeniu parametrów kolej na funkcję `ConvertFrom-Cab`, która będzie przyjmować dwa parametry z wiersza poleceń. Pierwszy będzie zawierał plik `.cab`, a drugi miejsce, w którym mają zostać wypakowane pliki.

```
Function ConvertFrom-Cab($cab,$destination)
```

Teraz powinniśmy utworzyć egzemplarz obiektu `Shell.Application`, który zawiera wiele bardzo przydatnych metod. W tabeli 6.5 znajduje się zestawienie wszystkich składowych obiektu `Shell.Application`.

TABELA 6.5. Składowe obiektu `Shell.Application`

| Nazwa | Typ składowej | Definicja |
|---------------------|---------------|--|
| AddToRecent | Metoda | void AddToRecent (Variant, string) |
| BrowseForFolder | Metoda | Folder BrowseForFolder (int, string, int, Variant) |
| CanStartStopService | Metoda | Variant CanStartStopService (string) |

TABELA 6.5. Składowe obiektu Shell.Application — ciąg dalszy

| Nazwa | Typ składowej | Definicja |
|----------------------|---------------|--|
| CascadeWindows | Metoda | void CascadeWindows () |
| ControlPanelItem | Metoda | void ControlPanelItem (string) |
| EjectPC | Metoda | void EjectPC () |
| Explore | Metoda | void Explore (Variant) |
| ExplorerPolicy | Metoda | Variant ExplorerPolicy (string) |
| FileRun | Metoda | void FileRun () |
| FindComputer | Metoda | void FindComputer () |
| FindFiles | Metoda | void FindFiles () |
| FindPrinter | Metoda | void FindPrinter (string, string, string) |
| GetSetting | Metoda | bool GetSetting (int) |
| GetSystemInformation | Metoda | Variant GetSystemInformation (string) |
| Help | Metoda | void Help () |
| IsRestricted | Metoda | int IsRestricted (string, string) |
| IsServiceRunning | Metoda | Variant IsServiceRunning (string) |
| MinimizeAll | Metoda | void MinimizeAll () |
| NameSpace | Metoda | Folder NameSpace (Variant) |
| Open | Metoda | void Open (Variant) |
| RefreshMenu | Metoda | void RefreshMenu () |
| ServiceStart | Metoda | Variant ServiceStart (string, Variant) |
| ServiceStop | Metoda | Variant ServiceStop (string, Variant) |
| SetTime | Metoda | void SetTime () |
| ShellExecute | Metoda | void ShellExecute (string, Variant, Variant, Variant, Variant) |
| ShowBrowserBar | Metoda | Variant ShowBrowserBar (string, Variant) |
| ShutdownWindows | Metoda | void ShutdownWindows () |
| Suspend | Metoda | void Suspend () |
| TileHorizontally | Metoda | void TileHorizontally () |
| TileVertically | Metoda | void TileVertically () |
| ToggleDesktop | Metoda | void ToggleDesktop () |
| TrayProperties | Metoda | void TrayProperties () |
| UndoMinimizeALL | Metoda | void UndoMinimizeALL () |

TABELA 6.5. Składowe obiektu Shell.Application — ciąg dalszy

| Nazwa | Typ składowej | Definicja |
|-----------------|---------------|--------------------------------|
| Windows | Metoda | IDispatch Windows () |
| WindowsSecurity | Metoda | void WindowsSecurity () |
| WindowSwitcher | Metoda | void WindowSwitcher () |
| Application | Własność | IDispatch Application () {get} |
| Parent | Własność | IDispatch Parent () {get} |

Ponieważ chcemy użyć nazwy obiektu COM więcej niż raz, powinniśmy zapisać jego identyfikator programu w zmiennej. Później zapisanego łańcucha będzie można używać w poleceniu New-Object i przy dostarczaniu użytkownikowi informacji zwrotnych. Poniżej pokazany jest wiersz kodu przypisujący identyfikator programu obiektu Shell.Application do łańcucha:

```
{
$comObject = "Shell.Application"
```

Teraz powinniśmy zatroszczyć się o dostarczenie informacji zwrotnych użytkownikowi. W tym celu użyjemy polecenia Write-Debug, za pomocą którego poinformujemy o próbie utworzenia obiektu typu Shell.Application, jak pokazano poniżej:

```
Write-Debug "Tworzenie obiektu $comObject"
```

Po poinformowaniu użytkownika, że zamierzamy utworzyć obiekt, możemy go rzeczywiście utworzyć, jak widać poniżej:

```
$shell = New-Object -Comobject $comObject
```

Teraz powinniśmy sprawdzić, czy nie wystąpiły błędy, za pomocą automatycznej zmiennej \$? Zmienna ta informuje o tym, czy ostatnie polecenie zostało wykonane pomyślnie. Jest to zmienna logiczna, co można wykorzystać w celu uproszczenia kodu, a dokładniej: można użyć operatora negacji ! i instrukcji if. Jeśli wartością zmiennej nie jest true, można zgłosić błąd za pomocą instrukcji Throw i zatrzymać wykonywanie skryptu. Poniżej znajduje się omawiana część skryptu:

```
if(!$?) { $(Throw "Nie udało się utworzyć obiektu $comObject.")}
```

Jeśli skrypt bez przeszkód utworzy obiekt Shell.Application, należy o tym poinformować użytkownika, jak pokazano poniżej:

```
Write-Debug "Tworzenie źródłowego obiektu cab dla $cab."
```

Kolejną czynnością jest połączenie się z plikiem .cab za pomocą metody Namespace obiektu Shell.Application, jak pokazano poniżej. Jest to kolejny ważny etap procesu, więc warto dodać następną instrukcję Write-Debug, aby poinformować użytkownika o postępie pracy.

```
$sourceCab = $shell.Namespace($cab).items()
Write-Debug "Tworzenie obiektu folderu docelowego dla $destination."
```

Czas połączyć się z folderem docelowym za pomocą metody Namespace w sposób pokazany poniżej. Dodatkowo informujemy za pomocą instrukcji Write-Debug, z którym folderem nawiązaliśmy połączenie.

```
$DestinationFolder = $shell.Namespace($destination)
Write-Debug "Rozpakowywanie $cab do $destination"
```

Po tych wszystkich przygotowaniach rzeczywiste polecenie służące do rozpakowania pliku cabinet wydaje się nieco od rzeczy. Możemy użyć metody `copyhere` z obiektu `folder` zapisanego w zmiennej `$destinationFolder`. Jako parametr wejściowy należy przekazać odwołanie do pliku `.cab` zapisane w zmiennej `$sourceCab`:

```
$DestinationFolder.CopyHere($sourceCab)
}
```

Punkt początkowy skryptu wykonuje dwie czynności. Po pierwsze: sprawdza, czy istnieje zmienna `$debug`. Jeśli tak, ustawia zmienną `$debugPreference` na `continue`, aby zmusić polecenie `Write-Debug` do wydrukowania wiadomości w oknie konsoli. Po drugie: wywołuje funkcję `ConvertFrom-Cab` i przekazuje ścieżkę do pliku `.cab` przez parametr wiersza poleceń `-cab` oraz folder docelowy dla rozpakowanych plików przez parametr `-destination`.

```
if($debug) { $debugPreference = "continue" }
ConvertFrom-Cab -cab $cab -destination $destination
```

Poniżej znajduje się kompletny kod źródłowy skryptu *ExpandCab.ps1*.

ExpandCab.ps1

```
Param(
    $cab = "C:\fso\acab.cab",
    $destination = "C:\fso1",
    [switch]$debug
)
Function ConvertFrom-Cab($cab,$destination)
{
    $comObject = "Shell.Application"
    Write-Debug "Tworzenie obiektu $comObject"
    $shell = New-Object -Comobject $comObject
    if(!$?) { $(Throw "Nie udało się utworzyć obiektu $comObject.")}
    Write-Debug "Tworzenie źródłowego obiektu cab dla $cab."
    $sourceCab = $shell.Namespace($cab).items()
    Write-Debug "Tworzenie folderu docelowego dla $destination."
    $DestinationFolder = $shell.Namespace($destination)
    Write-Debug "Rozpakowywanie $cab do $destination."
    $DestinationFolder.CopyHere($sourceCab)
}

# *** punkt wejściowy ***
if($debug) { $debugPreference = "continue" }
ConvertFrom-Cab -cab $cab -destination $destination
```

Brak obsługi aplikacji zewnętrznych

Działanie wielu funkcji zarządzania nadal zależy od wsparcia ze strony wiersza poleceń. Typowym przykładem jest narzędzie NETSH. Innym przykładem jest *MakeCab.exe*. W systemie Windows Vista i nowszych usunięto obiekt `makecab`, przez co w systemach tych do tworzenia plików `.cab` należy używać narzędzia *MakeCab.exe*.

Najpierw należy zdefiniować kilka parametrów wiersza poleceń, jak widać poniżej:

```
Param(
    $filepath = "C:\fso",
    $path = "C:\fso\cabfiles",
    [switch]$debug
)
```

Następnie należy utworzyć funkcję New-DDF tworzącą podstawowy plik *.ddf* używany przez narzędzie *MakeCab.exe* do utworzenia pliku *.cab*. Dokumentację składni tych typów plików można znaleźć w SDK Microsoft Cabinet w portalu MSDN. Jeśli przy tworzeniu funkcji New-DDF użyje się słowa kluczowego *Function*, to można za pomocą polecenia *Join-Path* utworzyć ścieżkę do tymczasowego pliku *.ddf*. Można połączyć nazwy dysku, folderu i pliku, ale byłoby to kłopotliwe i łatwo byłoby przy tym popełnić błąd. Dobrym zwyczajem jest tworzenie wszystkich ścieżek za pomocą polecenia *Join-Path*, jak pokazano poniżej:

```
Function New-DDF($path,$filePath)
{
    $ddfFile = Join-Path -path $filePath -childpath temp.ddf
```

Jeśli skrypt zostanie uruchomiony z przełącznikiem *-debug*, w tym miejscu powinniśmy wyświetlić jakąś informację za pomocą polecenia *Write-Debug*, na przykład:

```
Write-Debug "Ścieżka do pliku DDF: $ddfFile"
```

Teraz utworzymy pierwszą część pliku *.ddf* przy użyciu rozwijanego łańcucha miejscowego (ang. *here-string*). Zaletą tego rodzaju łańcuchów jest to, że nie trzeba w nich zastępować żadnych znaków sekwencjami specjalnymi. Na przykład w plikach *.ddf* komentarze oznacza się średnikiem, który jest znakiem zarezerwowanym przez konsolę Windows PowerShell. Jeśli tekst pliku *.ddf* napiszemy bez użycia łańcucha miejscowego, będziemy musieli zastąpić wszystkie średniki sekwencjami specjalnymi, aby uniknąć błędów kompilacji. Dzięki użyciu rozwijanego łańcucha miejscowego możemy wykorzystać możliwość rozwijania zmiennych. Łańcuch miejscowy zaczyna się od znaku *@* i cudzysłowu, a kończy cudzysłowem i znakiem *@*, jak pokazano poniżej:

```
$ddfHeader =@"
;*** MakeCAB Directive file
;
.OPTION EXPLICIT
.set CabinetNameTemplate=Cab.*.cab
.set DiskDirectory1=C:\fso\cabfiles
.set MaxDiskSize=CDROM
.set Cabinet=on
.set Compress=on
"@
```

Teraz możemy wyświetlić dodatkowe informacje dla użytkownika za pomocą polecenia *Write-Debug*, jak pokazano poniżej:

```
Write-Debug "Zapisywanie nagłówka pliku DDF do $ddfFile."
```

Po wyświetleniu informacji dla użytkownika przechodzimy do sekcji, która może stwarzać problemy. Plik *.ddf* musi być w formacie ASCII, a konsola Windows PowerShell domyślnie używa standardu Unicode. Aby zapewnić odpowiedni format pliku, należy użyć polecenia *Out-File*. Zazwyczaj można uniknąć konieczności użycia tego polecenia poprzez użycie strzałek przekierowania plikowego, ale nie tym razem. Oto opisywana składnia.

```
$ddfHeader | Out-File -filepath $ddfFile -force -encoding ASCII
```


Przed rozpoczęciem gromadzenia kolekcji plików za pomocą polecenia `Get-ChildItem` możemy wyświetlić kolejną informację dla użytkownika, jak pokazano poniżej:

```
Write-Debug "Generowanie kolekcji plików z $filePath"
Get-ChildItem -path $filePath |
```

Nie można zapomnieć o odfiltrowaniu folderów z kolekcji, ponieważ narzędzie *MakeCab.exe* nie potrafi ich kompresować. W tym celu użyjemy polecenia `Where-Object` z operatorem negacji oznaczającym w tym przypadku, że obiekt nie może być kontenerem.

```
Where-Object { !$_.psiscontainer } |
```

Po odfiltrowaniu folderów bierzemy się do pracy z pojedynczymi plikami przychodzącymi przez potok za pomocą polecenia `ForEach-Object`. Jako że `ForEach-Object` nie jest instrukcją językową, tylko poleceniem, nawias klamrowy musi znajdować się w tym samym wierszu co nazwa tego polecenia. Problem w tym, że klamry często zakupuje się w kodzie źródłowym. Osobiście lubię ustawiać klamry w jednej linii w pionie, chyba że polecenie jest bardzo krótkie, jak powyższe `Where-Object`, chociaż to wymaga użycia znaku kontynuacji wiersza (tzw. *grawis* lub *backtick*). Niektórzy programiści nie lubią tego robić, ale moim zdaniem korzyści przewyższają wady, ponieważ dzięki wyrównaniu klamer kod jest bardziej czytelny. Poniżej znajduje się początek omawianego polecenia `ForEach-Object`:

```
Foreach-Object {
```

Jako że plik *.ddf* używany przez narzędzie *MakeCab.exe* jest w formacie ASCII, należy przekształcić własność `FullName` obiektu `System.IO.FileInfo` zwróconą przez polecenie `Get-ChildItem` w łańcuch. Ponadto w nazwach plików mogą występować spacje, więc wartość `FullName` dobrze jest umieścić w cudzysłowie, jak pokazano poniżej:

```
{
    '"' + $_.fullname.toString() + '"' |
```

Następnie przekazujemy potokowo nazwy plików do polecenia `Out-File`, nie zapominając o określeniu kodowania ASCII, i dodajemy przełącznik `-append`, aby nie spowodować nadpisania istniejącej zawartości pliku.

```
Out-File -filepath $ddfFile -encoding ASCII -append
}
```

Teraz możemy wyświetlić kolejną porcję informacji dla użytkownika i wywołać funkcję `New-Cab`, jak pokazano poniżej:

```
Write-Debug "Plik DDF został utworzony. Wywoływanie funkcji New-Cab."
New-Cab($ddfFile)
} #end New-DDF
```

Przy rozpoczynaniu wykonywania funkcji `New-Cab` można poinformować o tym fakcie użytkownika, jak pokazano poniżej:

```
Function New-Cab($ddfFile)
{
    Write-Debug "Rozpoczynanie wykonywania funkcji New-Cab. Plik DDF nazywa się $ddfFile."
```

Jeśli skrypt zostanie uruchomiony z przełącznikiem `-debug`, to można użyć parametru `/V` narzędzia *MakeCab.exe* w celu wyświetlenia szczegółowych informacji diagnostycznych. Jeżeli skrypt zostanie uruchomiony bez tego przełącznika, to znaczy, że nie chcemy zapelniać ekranu nadmiarem informacji, więc pozostajemy przy standardowym poziomie szczegółowości narzędzia, jak pokazano poniżej:

```
if($debug)
{ makecab /f $ddfFile /V3 }
Else
{ makecab /f $ddfFile }
} #end New-Cab
```

Punkt wejściowy skryptu sprawdza, czy istnieje zmienna `$debug`. Jeśli tak, ustawia zmienną `$debugPreference` na `continue`, dzięki czemu na ekranie będą wyświetlane informacje dostarczane przez polecenie `Write-Debug`. Następnie wywoływane jest polecenie `New-DDF` z wartościami `path` i `filepath` przekazanymi przez wiersz poleceń, jak widać poniżej:

```
if($debug) {$DebugPreference = "continue"}
New-DDF -path $path -filepath $filepath
```

Poniżej pokazano ukończony kod źródłowy skryptu *CreateCab2.ps1*.

CreateCab2.ps1

```
Param(
    $filepath = "C:\fso",
    $path = "C:\fso1\cabfiles",
    [switch]$debug
)
Function New-DDF($path,$filePath)
{
    $ddfFile = Join-Path -path $filePath -childpath temp.ddf
    Write-Debug "Ścieżka do pliku DDF: $ddfFile"
    $ddfHeader =@''
    ,*** Plik dyrektyw MakeCAB
    ;
    .OPTION EXPLICIT
    .Set CabinetNameTemplate=Cab.*.cab
    .set DiskDirectory1=C:\fso1\Cabfiles
    .Set MaxDiskSize=CDROM
    .Set Cabinet=on
    .Set Compress=on
    ""
    Write-Debug "Zapisywanie nagłówka DDF do $ddfFile."
    $ddfHeader | Out-File -filepath $ddfFile -force -encoding ASCII
    Write-Debug "Generowanie kolekcji plików z $filePath."
    Get-ChildItem -path $filePath |
    Where-Object { !$_.psiscontainer } |
    Foreach-Object {
        {
            '$' + $_.fullname.toString() + '$' |
            Out-File -filepath $ddfFile -encoding ASCII -append
        }
    }
    Write-Debug "Plik DDF został utworzony. Wywoływanie funkcji New-Cab."
    New-Cab($ddfFile)
} #end New-DDF
```

```
Function New-Cab($ddfFile)
{
    Write-Debug "Rozpoczynanie wykonywania funkcji New-Cab. Plik DDF nazywa się $ddfFile."
    if($debug)
    { makecab /f $ddfFile /V3 }
    Else
    { makecab /f $ddfFile }
} #end New-Cab

# *** punkt początkowy skryptu ***
if($debug) {$DebugPreference = "continue"}
New-DDF -path $path -filepath $filepath
```

Dodatkowe źródła informacji

- Centrum skryptów portalu TechNet pod adresem <http://www.microsoft.com/technet/scriptcenter> zawiera wiele przykładowych skryptów, w których użyto wszystkich opisanych w tym rozdziale technik.
- W książce *Windows PowerShell™ Scripting Guide* (Microsoft Press 2008) znajdują się przykłady użycia klas WMI i różnych klas platformy .NET w konsoli Windows PowerShell.
- Dobrym podręcznikiem do WMI jest książka *Windows Scripting with WMI Self Paced Learning Edition* (Microsoft Press 2005).
- W bibliotece MSDN na stronie <http://msdn.microsoft.com/en-us/library/default.aspx> znajduje się pełna dokumentacja różnych produktów firmy Microsoft. Biblioteka ta jest autorytatywnym źródłem informacji na temat wszystkich produktów firmy Microsoft.
- Na stronie <http://gallery.technet.microsoft.com/scriptcenter/Get-on-thousands-of-ef3175c7> znajduje się seria artykułów dotyczących wydajności konsoli Windows PowerShell autorstwa inżyniera z firmy Microsoft Georgesa Maheu.
- Wszystkie skrypty opisane w tym rozdziale są dostępne do pobrania w repozytorium centrum skryptów portalu TechNet pod adresem <http://gallery.technet.microsoft.com/scriptcenter/PowerShell-40-Best-d9e16039>.

Rozdział 7

Śledzenie możliwości zastosowania skryptów

- Ocenianie, czy skrypt jest potrzebny
- Obliczanie korzyści z zastosowania skryptu
- Współpraca nad skryptami
- Dodatkowe źródła informacji

Należy śledzić możliwości zastosowania skryptów, aby mieć pewność, że najbardziej korzystne skrypty zostaną napisane najszybciej. Taki spis możliwych do oskryptowania przypadków może efektywnie służyć do zarządzania skryptami w przedsiębiorstwie. Kluczem jest właściwe posługiwanie się tym zbiorem informacji.

Ewaluacja potrzeby napisania skryptu

Nie wszystko w konsoli Windows PowerShell 4.0, podobnie jak w Windows PowerShell 3.0, musi być oskryptowane. Programiści języków Microsoft VBScript i Perl często mają odczucie, że w danej sytuacji powinni napisać skrypt. Ale wiele czynności można wykonać w wierszu poleceń bez używania jakiegokolwiek skryptu.

Jedną z wielkich zalet konsoli Windows PowerShell jest możliwość używania instrukcji językowych bezpośrednio w wierszu poleceń. Instrukcja `for` umożliwia tworzenie pętli, do obsługi których w innych językach konieczne byłoby napisanie skryptu. Aby ułatwić pracę w wierszu poleceń, twórcy konsoli Windows PowerShell umożliwili dzielenie poleceń na kilka wierszy. Po zakończeniu wpisywania można po raz drugi nacisnąć klawisz *Enter*, aby je wykonać. Poniżej znajduje się polecenie wysyłające polecenie `ping` do wszystkich adresów IP z przedziału od 192.168.2.1 do 192.168.2.10:

```
PS C:\> for($i = 1 ; $i -le 10 ; $i++)
>> { Test-Connection -Destination 192.168.2.$i -Count 1 -ErrorAction SilentlyContinue
|
>> Format-Table -property Address, statusCode, ResponseTime -AutoSize }
>>
```

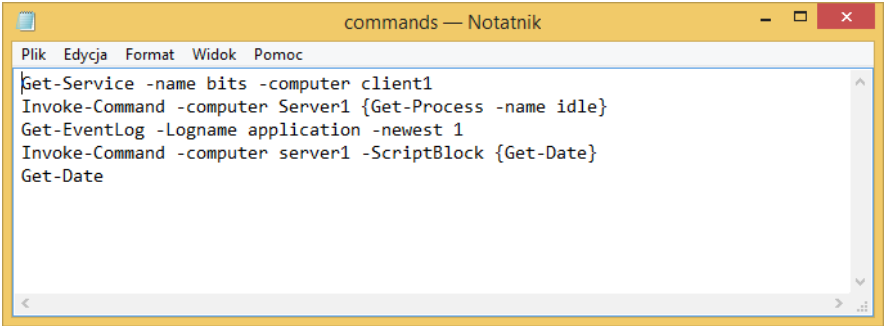
| Address | statusCode | ResponseTime |
|--------------|------------|--------------|
| ----- | ----- | ----- |
| 192.168.2.1 | 0 | 1 |
| Address | statusCode | ResponseTime |
| ----- | ----- | ----- |
| 192.168.2.3 | 0 | 2 |
| Address | statusCode | ResponseTime |
| ----- | ----- | ----- |
| 192.168.2.5 | 0 | 0 |
| Address | statusCode | ResponseTime |
| ----- | ----- | ----- |
| 192.168.2.10 | 0 | 10 |

Jeśli zastosuje się parę sztuczek składni Windows PowerShell, np. aliasy, częściowe parametry i argumenty pozycyjne, powyższe polecenie można znacznie skrócić. Poniżej znajduje się właśnie taka skrócona wersja:

```
1..10 | % {Test-Connection 10.1.1.$_ -cou 1 -ea 0 | ft Address, StatusCode, ResponseTime -au}
```

Odczytywanie pliku tekstowego

Najprostszy skrypt Windows PowerShell jest zbiorem poleceń PowerShell zapisanych w pliku o określonym rozszerzeniu. Jeśli ktoś nie chce pisać skryptu, może zapisać zbiór poleceń w pliku tekstowym, jak pokazano na rysunku 7.1.



RYSUNEK 7.1. Plik zawierający zbiór poleceń Windows PowerShell

Przy użyciu konsoli Windows PowerShell można odczytać plik *commands.txt* i wykonać znajdujące się w nim polecenia za pomocą polecenia *Get-Content*, które służy właśnie do pobierania poleceń z plików tekstowych. Domyślnym parametrem polecenia *Get-Content* jest *path*. Podczas pracy w wierszu poleceń nie trzeba go podawać. Ścieżka może być lokalna lub nawet UNC (ang. *Universal Naming Convention*), pod warunkiem że ma się odpowiednie uprawnienia do odczytu pliku tekstowego. Najlepszym sposobem użycia tej techniki jest przekazanie wyników do polecenia *Invoke-Expression*. Każde polecenie przekazywane przez potok do tego polecenia zostaje przez nie wykonane, jak pokazano poniżej:

```
Get-Content -Path C:\fso\Commands.txt | Invoke-Expression
```

Wynik tego polecenia pokazano na rysunku 7.2.

```

Administrator: Windows PowerShell
PS C:\> Get-Content -Path .\bin\Commands.txt | Invoke-Expression

Status   Name                DisplayName
-----
Running  bits                Background Intelligent Transfer Ser...

Id        : 0
Handles   : 0
CPU       :
Name      : Idle
PSComputerName : Server1

MachineName : client1.iammred.net
Data        : {}
Index       : 685
Category    : (0)
CategoryNumber : 0
EventID     : 1005
EntryType   : Information
Message     : The description for Event ID '1073742829' in Source
              'Customer Experience Improvement Program' cannot be
              found. The local computer may not have the necessary
              registry information or message DLL files to display the
              message, or you may not have permission to access them.
              The following information is part of the event:
Source      : Customer Experience Improvement Program
ReplacementsStrings : {}
InstanceId  : 1073742829
TimeGenerated : 8/27/2013 3:00:01 PM

```

RYSUNEK 7.2. Polecenia Windows PowerShell bez problemu przetwarzają zawartość plików tekstowych i wykonują znajdujące się w nich polecenia

Podczas korzystania z funkcji pracy zdalnej konsoli Windows PowerShell do pracy z niezaufaną domeną łatwo się pomylić przy używaniu takich poleceń jak `Get-Content`. Parametr `-path` odnosi się do ścieżki lokalnej dla komputera docelowego, a nie tego, z którego wykonuje się polecenie. W poniższym przykładzie ścieżka `c:\fso\commands.txt` wskazuje plik tekstowy o nazwie `commands.txt` znajdujący się w folderze `fso` na dysku C komputera o nazwie Sydney w domenie `Woodbridgebank.com`. Jeśli plik ten nie zostanie tam znaleziony, wystąpi błąd.

```

PS C:\> invoke-command -ComputerName sydney.woodbridgebank.com -Credential
administrator@woodbridgebank.com -ScriptBlock {get-content -Path C:\fso\Commands.txt
| Invoke-Expression}
Invoke-Command : Cannot find path 'C:\fso\Commands.txt' because it does not exist.At
line:1 char:15
+ invoke-command <<<< -ComputerName sydney.woodbridgebank.com -Credential
administrator@woodbridgebank.com -ScriptBlock {get-content -Path
C:\fso\Commands.txt | Invoke-Expression}
+ CategoryInfo          : ObjectNotFound: (C:\fso\Commands.txt:String)
[Get-Content], ItemNotFoundException
+ FullyQualifiedErrorId : PathNotFound,Microsoft.PowerShell.Commands.GetContentCommand

```

Pewnie zastanawiasz się, czy możesz wskazać plik `commands.txt` na komputerze, z którego wykonywane jest polecenie, za pomocą ścieżki UNC. Jako że zdalna domena jest niezaufana, nie ma żadnego kontekstu zabezpieczeń, który pozwoliłby zdalnemu poleceniu uzyskać dostęp do systemu plików komputera lokalnego. Polecenie znajdujące się w parametrze `ScriptBlock` zostanie wykonane w kontekście komputera docelowego, którym w tym przypadku jest `Sydney.Woodbridgebank.com`.

Komputer lokalny, z którego wykonywane jest polecenie, nazywa się `Windows 8.NWTraders.com`. Jako że te dwie domeny nie mają zaufanych relacji, nie można podać danych poświadczających, które umożliwiłyby wykonanie polecenia. Wynik próby wykonania polecenia przedstawiono poniżej:

```
PS C:\> invoke-command -ComputerName sydney.woodbridgebank.com -Credential
administrator@woodbridgebank.com -ScriptBlock {get-content -Path '\\Windows 8\fso\Commands.txt'
| Invoke-Expression}
Invoke-Command : Cannot find path '\\Windows 8\fso\Commands.txt' because it does not exist.
At line:1 char:15
+ invoke-command <<<< -ComputerName sydney.woodbridgebank.com -Credential
administrator@woodbridgebank.com -ScriptBlock {get-content -Path '\\Windows 8\fso\Commands.txt'
| Invoke-Expression}
+ ~~~~~
+ CategoryInfo          : ObjectNotFound: (\\Windows 8\fso\Commands.txt:String)
  [Get-Content],ItemNotFoundException
+ FullyQualifiedErrorId : PathNotFound,Microsoft.PowerShell.Commands.GetContentCommand
```

Mylące może być to, że polecenie `Get-Content` w pojedynkę działa bardzo dobrze. Na komputerze o nazwie *client1* zawierającym folder o nazwie *bin*, w którym znajduje się plik tekstowy o nazwie *commands.txt*, polecenie to zostanie wykonane, pod warunkiem że umieścimy je w pojedynczym cudzysłowie, jak pokazano poniżej:

```
PS C:\> Get-Content -Path '\\client1\bin\Commands.txt'
Get-Service -Name bits -ComputerName Windows 8
Get-Process -Name explorer -ComputerName berlin
Get-EventLog -LogName application -Newest 1 -ComputerName berlin,Windows 8
Invoke-Command -ComputerName Berlin { Get-Date }
Get-Date
```

Ale tego należało się spodziewać, ponieważ zalogowany użytkownik ma uprawnienia dostępu do użytego folderu, więc może też wczytać za pomocą polecenia `Get-Content` ścieżkę UNC do pliku *commands.txt*.

Można zmapować dysk na zdalnej domenie i skopiować plik z lokalnego komputera do odpowiedniego folderu na serwerze zdalnym. Oczywiście konieczne będzie otwarcie dodatkowych portów w zaporze systemu Windows, co może, ale nie musi być wykonalne, w zależności od konfiguracji sieci. Jeśli zdecydujesz się na takie rozwiązanie, możesz dokonać zmian konfiguracyjnych za pomocą konsoli Windows PowerShell, jak pokazano poniżej:

```
PS C:\> Invoke-Command -ComputerName Sydney.WoodBridgeBank.Com -Credential
Administrator@WoodbridgeBank.com -ScriptBlock { netsh advfirewall firewall set rule group="File
and Printer Sharing" new enable=Yes }

Updated 28 rule(s).
Ok.
```

Po wprowadzeniu wyjątku do zapory można zmapować dysk przy użyciu graficznego interfejsu użytkownika, polecenia `Net Use` z poziomu konsoli Windows PowerShell lub dowolną inną metodą. Po zmapowaniu dysku można skopiować plik *commands.txt* na serwer za pomocą polecenia `Copy-Item`, jak pokazano poniżej:

```
Copy-Item -Path C:\fso\Commands.txt -Destination z:
```

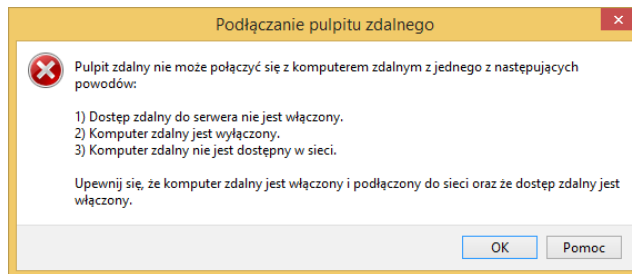
UWAGA

Podczas kopiowania obiektów na zmapowany dysk za pomocą polecenia `Copy-Item` należy mieć na uwadze strukturę tego zmapowanego dysku. Często mapuje się dyski w celu umożliwienia używania ich na zdalnym komputerze. Zdalny udział prawie zawsze jest udziałem folderu, a nie całego dysku. Jako że nasz dysk zdalny jest punktem mapowania jednego folderu, miejsce docelowe ulega zmianie. W poniższym poleceniu dysk *Z* jest udziałem folderu *Fso* na serwerze zdalnym. Parametr `-destination` kieruje do korzenia zmapowanego dysku, a nie do folderu *Z:\fso*.

Teraz można użyć pliku *commands.txt* bezpośrednio w poleceniu Windows PowerShell, jak pokazano poniżej:

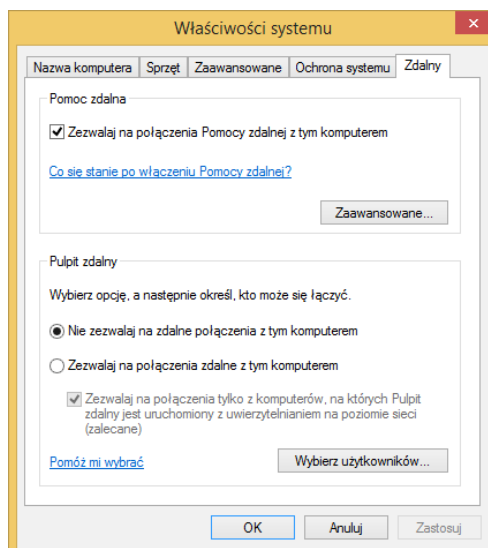
```
PS C:\> invoke-command -ComputerName Sydney.WoodbridgeBank.com -Credential administrator@WoodBridgeBank.com -ScriptBlock { Get-Content -Path C:\fso\Commands.txt | Invoke-Expression }
```

Jednym z rozwiązań dylematu mapowania dysków jest użycie pulpitu zdalnego, który umożliwia dostęp do zasobów lokalnych, jeśli chcemy mieć je dostępne w swojej sesji. Wybierając pulpit zdalny, klikając przycisk *Options* i wybierając kartę *Local resources*, możemy wybrać połączenia drukarki, dostęp do schowka oraz udostępnić lokalne dyski w sesji RDP (ang. *Remote Desktop Protocol*). Ustawienia pulpitu zdalnego można otworzyć, klikając kolejno *Start/Wszystkie programy/Akcesoria/Podłączanie pulpitu zdalnego*. Jeśli pulpit zdalny nie został jeszcze skonfigurowany, system wyświetli informację o braku dostępu widoczną na rysunku 7.3.



RYSUNEK 7.3. Odmowa dostępu podczas próby połączenia się z pulpitem zdalnym

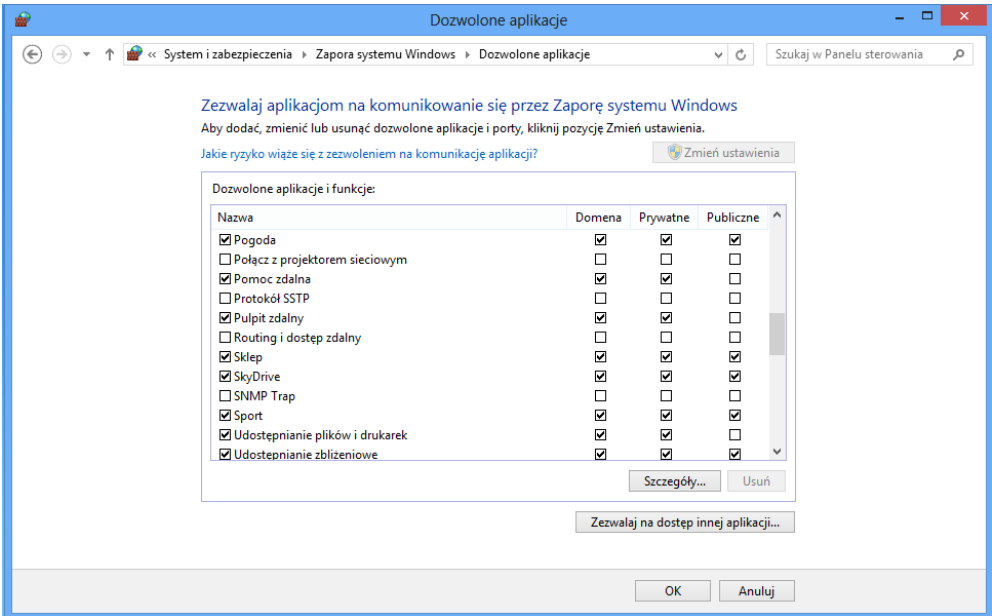
Aby włączyć zdalny pulpit w systemach Microsoft Windows 2012 i Windows 2012 R2, należy w menedżerze serwera w węźle *Server lokalny* wybrać opcję *Pulpit zdalny*. W systemach Windows 8 i Windows 8.1 należy w Panelu sterowania otworzyć aplet *System*, kliknąć odnośnik *Ustawienia zdalne*. Pojawi się okno dialogowe *Właściwości systemu* z otwartą kartą *Zdalny*, jak widać na rysunku 7.4.



RYSUNEK 7.4. Pulpit zdalny musi być włączony

W dolnej części okna znajdują się trzy opcje dotyczące pulpitu zdalnego. Domyślnie połączenie z pulpitem zdalnym jest zabronione. Najbezpieczniejszym rozwiązaniem jest zaznaczenie opcji *Zezwalaj na połączenia tylko z komputerów, na których Pulpit zdalny jest uruchomiony z uwierzytelnianiem na poziomie sieci (zalecane)*. Dodatkowo można określić, którzy użytkownicy mogą nawiązywać połączenie. Domyślnie uprawnienia takie mają wszyscy członkowie grupy administratorów domeny.

Po włączeniu pulpitu zdalnego automatycznie zostanie utworzony wyjątek umożliwiający przepływ ruchu RDP przez zaporę systemu Windows. Warto dokładnie sprawdzić, czy na pewno wyjątek został utworzony. Na rysunku 7.5 widać wyjątek zapory w systemie Windows 8.1.



RYSUNEK 7.5. Pulpit zdalny musi mieć wyjątek w zaporze systemu Windows

Zapiski praktyka

Brian Wilhite

Premier Field Engineer (PFE), Microsoft Corporation

W ramach swoich codziennych obowiązków administratora systemów pracującego w dość dużej firmie często muszę rozwiązywać problemy wymagające użycia konsoli Windows PowerShell. Najczęściej korzystam z Instrumentacji zarządzania Windows (WMI), która jest moją ulubioną technologią w systemach Windows. Konsola PowerShell ułatwia pracę z WMI na niespotykaną wcześniej skalę. Jedną z najlepszych rzeczy, jakie można robić za pomocą narzędzi WMI i Windows PowerShell, jest tworzenie procesów na zdalnych komputerach. To bardzo przydatna możliwość dla każdego, kto musi instalować, aktualizować lub odinstalowywać programy na komputerach z systemem Windows w swoim środowisku, w którym nie ma programów do zarządzania typu System Center Configuration Manager.

Ostatnio na naszych serwerach ktoś przez pomyłkę zainstalował pewien program. Aby go szybko usunąć, postanowiłem za pomocą WMI zdalnie utworzyć proces *MSIExec.exe* z ciągiem dezinstalacji dla MSI. Ciąg dezinstalacji (*UninstallString*) dla każdego programu można znaleźć w rejestrze, typowo w następującej lokalizacji (<SID> to identyfikator SID użytkownika, z którym został zainstalowany dany program, a <GUID produktu> to niepowtarzalny identyfikator interesującego nas produktu): *HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Installer\UserData\<SID>\Products\<GUID produktu>\InstallProperties*.

W ten sposób można znaleźć ciąg dezinstalacji (*UninstallString*) wybranego programu, który chcemy usunąć z systemu. Usunąłem prawdziwy identyfikator GUID produktu, aby nie oczerniać niewinnej firmy:

```
MSIExec.exe /X{<GUID produktu>} /quiet
```

W tym przypadku wiem, bo to sprawdziłem, że przełącznik */quiet* zadziała dla produktu, który chcę usunąć. Ale Ty możesz mieć inne doświadczenia.

Do wykonania tego zadania użyjemy polecenia *Invoke-WmiMethod*. Ponadto użyjemy metody *Create* klasy *Win32_Process* w celu utworzenia procesu na zdalnym komputerze. Najlepiej mieć włączone narzędzia do pracy zdalnej Windows PowerShell w swoim komputerze, ale jeśli nie są włączone, nie panikuj. Opisuję kilka technik realizacji tego zadania.

W pierwszej technice przyjąłem założenie, że na docelowych komputerach zdalnych włączone są narzędzia do pracy zdalnej Windows PowerShell. Zmienna *\$Computers* zawiera zbiór nazw komputerów, których użyjemy jako urządzeń docelowych:

```
Invoke-Command -ScriptBlock {
    Invoke-WmiMethod -Class Win32_Process '
        -Name Create '
        -ArgumentList 'MSIExec.exe /X{<Product GUID>} /quiet' '
        -ComputerName $Computers
```

Drugą technikę można zastosować, gdy narzędzia do pracy zdalnej Windows PowerShell na komputerach zdalnych nie są włączone:

```
$Computers | ForEach-Object {
    Invoke-WmiMethod -Class Win32_Process '
        -Name Create '
        -ArgumentList 'MSIExec.exe /X{<Product GUID>} /quiet' '
        -ComputerName $_}
```

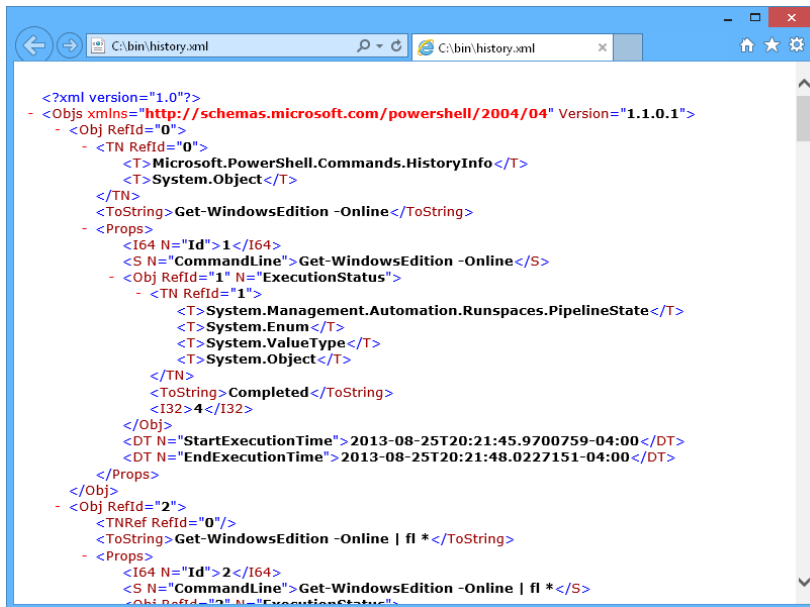
Z własnego doświadczenia wiem, że w takich sytuacjach jak ta, gdy komputerów są setki lub nawet tysiące, bardziej niezawodnym rozwiązaniem jest przekazanie nazw komputerów do polecenia *ForEach-Object* zamiast przekazywania całej listy do parametru *ComputerName*. Oba przedstawione rozwiązania powinny zwrócić obiekt zawierający *ReturnValue* i *ProcessId*. Jeśli tworzenie procesu powiedzie się, *ReturnValue* będzie mieć wartość 0. Zero wskazuje, że proces został utworzony, ale nie mówi nic o tym, czy dezinstalacja się powiodła. Dlatego należy to jeszcze sprawdzić w dziennikach zdarzeń wszystkich systemów.

Eksportowanie historii poleceń

Wiele prac administracyjnych wykonywanych przy użyciu konsoli Windows PowerShell polega na wpisywaniu serii poleceń w konsoli. Niezależnie od tego, czy edytujemy rejestr, czy zatrzymujemy różne procesy i usługi, aby zapewnić spójne środowisko operacyjne, należy skopiować prace konfiguracyjne na kilka różnych serwerów. W przeszłości trzeba było do tego tworzyć skrypty. Jeśli polecenia do zduplikowania są serią poleceń wpisywanych w konsoli, można obejść się bez skryptu, eksportując historię poleceń do pliku *.xml* za pomocą polecenia `Get-History`, jak pokazano poniżej:

```
Get-History | Export-Clixml -Path C:\fso\history.xml
```

Wynikiem tego polecenia jest plik *.xml* zawierający wszystkie polecenia, jakie wpisano w konsoli. Na rysunku 7.6 widać zawartość takiego przykładowego pliku:



RYСУNEK 7.6. Plik *.xml* z historią poleceń

Polecenia z tego pliku można importować za pomocą polecenia `Import-Clixml`. Jego wynik przekazuje się do polecenia `Add-History`, aby polecenia zawarte w zaimportowanym pliku dodać do historii konsoli. Należy użyć przełącznika `-passthru`, aby polecenia przekazać zarówno do polecenia `Add-History`, jak i `ForEach-Object`. W poleceniu `ForEach-Object` można wykonać każde polecenie z historii za pomocą polecenia `Invoke-History`. Poniżej przedstawiono opisywany zestaw poleceń wraz z wynikiem jego wykonania:

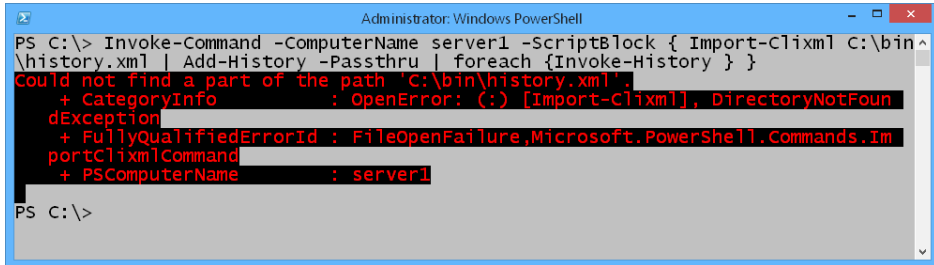
```
PS C:\> Import-Clixml -Path C:\fso\history.xml | Add-History -Passthru |
>> ForEach-Object { Invoke-History }
>>
if(!(test-path -path c:\fso4)) { new-item c:\fso4 -ItemType directory }

Directory: C:\
```

| Mode | LastWriteTime | Length | Name |
|-------|-------------------|--------|------|
| d---- | 1/9/2009 12:33 AM | | fso4 |

```
Get-Command >> C:\fso4\commands.txt
notepad C:\fso4\commands.txt
```

Technikę tę można też zastosować zdalnie, posługując się poleceniem `Invoke-Command`. Pamiętaj, że ścieżka `path` jest względna w odniesieniu do komputera będącego celem, a nie komputera wykonującego polecenie. Jeśli o tym zapomnisz, wystąpi błąd, jak widać na rysunku 7.7.



RYСУNEK 7.7. Błąd spowodowany użyciem lokalnych ścieżek do plików

Jeśli najpierw skopiujesz plik na komputer docelowy i dostosujesz swój wiersz poleceń, to technika polegająca na zaimportowaniu i wykonaniu historii sprawdzi się wyśmienicie. Zaletą konsoli Windows PowerShell jest to, że z poleceniem `Copy-Item` można używać ścieżek UNC. Właśnie to sprawia, że technika ta jest godna uwagi, ponieważ umożliwia łatwe przenoszenie plików na zdalny komputer, jak widać poniżej:

```
PS C:\> Copy-Item C:\fso\history.xml \\berlin\c$\fso
PS C:\> Import-Clixml -Path C:\fso\history.xml | Add-History -Passthru | ForEach -Object {
Invoke-History }
if(!(test-path -path c:\fso4)) { new-item c:\fso4 -ItemType directory }
```

Directory: C:\

| Mode | LastWriteTime | Length | Name |
|-------|-------------------|--------|------|
| d---- | 1/9/2009 12:40 AM | | fso4 |

```
Get-Command >> C:\fso4\commands.txt
notepad C:\fso4\commands.txt
```

Polecenia promieniste

Polecenia promieniste (ang. *fan-out command*) to takie, które uruchamia się z centralnego komputera na pewnej liczbie zdalnych komputerów. Jednym ze sposobów ich wykonywania jest użycie polecenia `Invoke-Command`, jak pokazano poniżej:

```
PS C:\> Invoke-Command -Computer berlin,Windows 8 -Script '
>> {"$env:computername $(get-date)"}
>>
WINDOWS 8 01/09/2009 08:31:42
BERLIN 01/09/2009 08:31:47
```

W sposób promienisty można wykonać wiele poleceń, przekazując im tablicę nazw komputerów jako parametr `-computername`. Wadą tej metody jest to, że zwrócone wyniki są prawie bezużyteczne. Najlepiej wytłumaczyć to na konkretnym przykładzie. Poniżej użyto polecenia `Get-Service` do pobrania informacji o konfiguracji usług z dwóch komputerów. Pierwszy z nich nazywa się `Windows 8`, a drugi `Berlin`. Jak widać, wyniki polecenia dla tych dwóch komputerów są pomieszczone i nie ma kolumny informującej, do którego komputera odnosi się dana informacja. W związku z tym wynik ten jest przydatny tylko w tym względzie, że pozwala szybko przejrzeć usługi na dwóch komputerach i znaleźć różnice w ich konfiguracji. Poniżej znajduje się omawiane polecenie i część zwróconego przez nie wyniku:

```
PS C:\> Get-Service -ComputerName Windows 8, Berlin
Status   Name                DisplayName
-----
Running  1-vmsvc             Virtual Machine Additions Services ...
Running  1-vmsvc             Virtual Machine Additions Services ...
Running  AeLookupSvc         Application Experience
Stopped  AeLookupSvc         Application Experience
Stopped  ALG                 Application Layer Gateway Service
Stopped  ALG                 Application Layer Gateway Service
Stopped  Appinfo             Application Information
Stopped  Appinfo             Application Information
Stopped  AppMgmt             Application Management
Stopped  AppMgmt             Application Management
Stopped  AudioEndpointBu...  Windows Audio Endpoint Builder
Stopped  AudioEndpointBu...  Windows Audio Endpoint Builder
Stopped  Audiosrv Windows    Audio
Stopped  Audiosrv Windows    Audio
Running  BFE Base Filtering Engine
Running  BFE Base Filtering Engine
Running  BITS Background     Intelligent Transfer Ser...
Stopped  BITS Background     Intelligent Transfer Ser...
Stopped  Browser Computer    Browser
Running  Browser Computer    Browser
>>> reszta wyników opuszczona >>>
```

Jak widać w powyższych danych, na jednym komputerze usługa `AeLookupSvc` jest uruchomiona, a na drugim zatrzymana. Aby dowiedzieć się, na którym działa, a na którym nie, wystarczy użyć polecenia `Get-Service`.

```
PS C:\> Get-Service -Name AeLookupSvc -computer Windows 8
Status   Name                DisplayName
-----
Stopped  AeLookupSvc         Application Experience

PS C:\> Get-Service -Name AeLookupSvc -computer Berlin
Status   Name                DisplayName
-----
Running  AeLookupSvc         Application Experience
```

Możesz pomyśleć, że pierwsze wystąpienie nazwy usługi należy do komputera, który jest pierwszy na liście. Ale jak widać, usługa `AeLookupSvc` jest uruchomiona na komputerze `Berlin`, a zatrzymana na komputerze `Windows 8`. W wyniku jest właśnie taka kolejność, ale w poleceniu promienistym najpierw wymieniony jest komputer `Windows 8`. Teraz pewnie pomyślisz, że najpierw wyświetlane są wyniki dla pierwszego komputera, a potem dla drugiego. Ale zanim

dojdiesz do wniosku, że tak musi być, sprawdź jeszcze jakąś inną usługę. W wyniku polecenia usługa BITS najpierw jest oznaczona jako uruchomiona, a potem jako zatrzymana. Sprawdźmy, jaki jest jej stan na poszczególnych komputerach:

```
PS C:\> Get-Service -Name Bits -computer berlin
Status Name DisplayName
-----
Stopped BITS Background Intelligent Transfer Ser...
```

```
PS C:\> Get-Service -Name Bits -computer Windows 8
Status Name DisplayName
-----
Running BITS Background Intelligent Transfer Ser...
```

Jak widać, usługa BITS jest zatrzymana na komputerze Berlin i uruchomiona na komputerze Windows 8. Wniosek z tego taki, że używając polecenia `Get-Service` w sposób promienisty przez dostarczenie mu tablicy nazw komputerów za pomocą parametru `-computername`, można uzyskać przydatne informacje. Chociaż jeśli chcemy dowiedzieć się, jaki jest status konkretnych usług na konkretnym komputerze, to dane te nie będą już tak pomocne. Najlepiej wynik tego polecenia przekazać do polecenia `Format-Table` i wybrać własność `machineName`. Wartość własności `displayName` jest taka sama jak w kolumnie *Nazwa* w konsoli MMC. Omawiane polecenie i część jego wyniku pokazano poniżej:

```
PS C:\> Get-Service -ComputerName berlin,Windows 8 |
format-table name, status, machinename, displayName -AutoSize
```

| Name | Status | MachineName | DisplayName |
|----------------------|---------|-------------|------------------------------|
| 1-vmsrvc | Running | Windows 8 | Virtual Machine Additions... |
| 1-vmsrvc | Running | berlin | Virtual Machine Additions... |
| AeLookupSvc | Running | berlin | Application Experience |
| AeLookupSvc | Stopped | Windows 8 | Application Experience |
| ALG | Stopped | berlin | Application Layer Gateway... |
| ALG | Stopped | Windows 8 | Application Layer Gateway... |
| Appinfo | Stopped | berlin | Application Information |
| Appinfo | Stopped | Windows 8 | Application Information |
| AppMgmt | Stopped | Windows 8 | Application Management |
| AppMgmt | Stopped | berlin | Application Management |
| AudioEndpointBuilder | Stopped | berlin | Windows Audio Endpoint Bu... |
| AudioEndpointBuilder | Stopped | Windows 8 | Windows Audio Endpoint Bu... |
| Audiosrv | Stopped | berlin | Windows Audio |
| Audiosrv | Stopped | Windows 8 | Windows Audio |
| BFE | Running | Windows 8 | Base Filtering Engine |
| BFE | Running | berlin | Base Filtering Engine |
| BITS | Stopped | berlin | Background Intelligent Tr... |
| BITS | Running | Windows 8 | Background Intelligent Tr... |
| Browser | Running | Windows 8 | Computer Browser |
| Browser | Stopped | berlin | Computer Browser |

Jako że wartość własności `displayName` często jest długa, więc i często nie mieści się w kolumnie o szerokości 80 znaków. Jeśli wymienisz ją na początku kolejki wybranych własności w poleceniu `Format-Table`, kilka kolumn może nie zostać wyświetlonych, jak pokazano poniżej:

```
PS C:\> Get-Service -ComputerName berlin,Windows 8 | format-table name, displayname, status,
machinename -AutoSize
```

```
WARNING: 2 columns do not fit into the display and were removed.
```

| Name | DisplayName |
|-------------|--|
| ----- | ----- |
| 1-vmsrvc | Virtual Machine Additions Services Application |
| 1-vmsrvc | Virtual Machine Additions Services Application |
| AeLookupSvc | Application Experience |
| AeLookupSvc | Application Experience |

Jak widać, w kodzie tym w ogóle nie było sensu wybierać własności `machineName`, bo i tak nie jest widoczna. Rozwiązaniem tego problemu może być umieszczanie potencjalnie długiej własności na końcu listy argumentów polecenia. Dzięki temu konsola Windows PowerShell skróci wartość tej własności, zamiast zapełniać cały ekran danymi, które i tak pewnie udałoby się odgadnąć, dysponując tylko ich częścią.

Innym rozwiązaniem jest pozbycie się parametru `-autosize` polecenia `Format-Table` i użycie zamiast niego parametru `Wrap`. Parametr ten sprawia, że długie wartości są zawijane, a nie obcinane. W zależności od tego, czego potrzebujesz, może to być przydatne lub irytujące. Poniżej znajduje się przykład danych zwróconych dzięki użyciu parametru `-Wrap`:

```
PS C:\> Get-Service -ComputerName berlin,Windows 8 | format-table name, displayname, status,
machinename -Wrap
```

| Name | DisplayName | Status MachineName |
|-------------|--|--------------------|
| ----- | ----- | ----- |
| 1-vmsrvc | Virtual Machine Additions Services Application | Running Windows 8 |
| 1-vmsrvc | Virtual Machine Additions Services Application | Running berlin |
| AeLookupSvc | Application Experience | Running berlin |
| AeLookupSvc | Application Experience | Stopped Windows 8 |

UWAGA

Może myślisz, że najlepszym rozwiązaniem jest użycie zarówno parametru `-autosize`, jak i `-Wrap`. Pozwoli to zmaksymalizować wykorzystanie przestrzeni ekranowej na prezentację zwróconych danych (dzięki parametrowi `-autosize`) i zawijać zbyt długie wiersze (dzięki parametrowi `-Wrap`). Tak się nie da, ale jeśli to zrobisz, program nie zwróci żadnego błędu. Konsola Windows PowerShell priorytetowo traktuje parametr `-autosize` i jeśli jest użyty, ignoruje parametr `-Wrap`. Kolejność ich wpisania nie ma przy tym znaczenia.

Wysyłanie zapytań do Active Directory

Większość administratorów sieci myśli, że aby wysłać zapytanie do Active Directory przy użyciu konsoli Windows PowerShell 1.0, trzeba napisać skrypt. Częściowo przekonanie to jest pozostałością po dniach świetności języka VBScript i odzwierciedla konieczność używania technologii ADO (ang. *ActiveX Data Object*) do wywoływania zapytań LDAP (ang. *Lightweight Directory Access Protocol*) do Active Directory. Wprawdzie można używać klasy `System.DirectoryServices.DirectorySearcher` w wierszu poleceń Windows PowerShell, ale nie jest to zbyt wygodne. Istnieją co prawda zewnętrzne polecenia `cmdlet` i dostawcy umożliwiające wykonywanie zapytań do Active Directory z wiersza poleceń, ale wielu administratorów sieci ma słuszne opory przed

instalowaniem na serwerach produkcyjnych oprogramowania społecznościowego bez wsparcia. Istnieje jeszcze drugie rozwiązanie w wierszu poleceń, polegające na użyciu narzędzia *DSQuery.exe*, ale mało komu przychodzi ono do głowy. Ale w konsoli Windows PowerShell 2.0 i jej nowszych wersjach sytuacja radykalnie się zmieniła. Za pomocą technik opisanych w tym podrozdziale informatyk może korzystać z dobrych rozwiązań do wykonywania zapytań do Active Directory w wierszu poleceń.

Zastosowanie akceleratora [ADSIEnumerator]

Jeśli ktoś chce wysyłać zapytania do Active Directory z poziomu wiersza poleceń konsoli Windows PowerShell, to ma kilka możliwości do wyboru. Jedną z nich jest użycie akceleratora typu [ADSIEnumerator], który jest skrótem nazwy klasy `System.DirectoryServices.DirectoryEnumerator`. Akcelerator ten, jak nietrudno się domyślić, jedynie pomaga zmniejszyć ilość wpisywanego tekstu. Oczywiście i tak należy mu podać odpowiedni konstruktor, aby rzeczywiście utworzyć egzemplarz klasy. Jeśli nie użyjesz akceleratora [ADSIEnumerator], musisz użyć polecenia `New-Object`, aby utworzyć obiekt. Polecenie to możesz wpisać w nawiasie, aby wymusić utworzenie obiektu, a następnie możesz wywołać metodę `FindAll` z obiektu `DirectoryEnumerator`. Otrzymana kolekcja obiektów `DirectoryEntry` zostaje potokowo przekazana do polecenia `Select-Object`, w którym zwracana jest własność `path`, jak pokazano poniżej:

```
PS C:\> (New-Object DirectoryServices.DirectoryEnumerator "ObjectClass=user").Find All() | Select path
Path
----
LDAP://CN=Administrator,CN=Users,DC=nwtraders,DC=com
LDAP://CN=Guest,CN=Users,DC=nwtraders,DC=com
LDAP://CN=BERLIN,OU=Domain Controllers,DC=nwtraders,DC=com
LDAP://CN=krbtgt,CN=Users,DC=nwtraders,DC=com
LDAP://CN=WINDOWS 8,CN=Computers,DC=nwtraders,DC=com
LDAP://CN=Windows 8Admin,OU=Students,DC=nwtraders,DC=com
List Truncated -
```

Aby użyć akceleratora typu [ADSIEnumerator], należy przekazać mu odpowiedni konstruktor, którym w wielu przypadkach jest filtr według składni LDAP. Składnia filtrów wyszukiwania LDAP jest zdefiniowana w dokumencie RFC 2254 i jest reprezentowana przez łańcuchy Unicode. Za pomocą filtrów wyszukiwania można definiować efektywne kryteria wyszukiwania. W tabeli 7.1 zamieszczono parę przykładów zastosowania tej składni.

TABELA 7.1. Przykłady filtrów wyszukiwania LDAP

| Filtr wyszukiwania | Opis |
|--|--|
| <code>ObjectClass=Computer</code> | Wszystkie obiekty komputerów |
| <code>ObjectClass=OrganizationalUnit</code> | Wszystkie obiekty jednostek organizacyjnych |
| <code>ObjectClass=User</code> | Wszystkie obiekty użytkowników oraz wszystkie obiekty komputerów |
| <code>ObjectCategory=User</code> | Wszystkie obiekty użytkowników |
| <code>(&(ObjectCategory=User) (ObjectClass=Person))</code> | Wszystkie obiekty użytkowników |

TABELA 7.1. Przykłady filtrów wyszukiwania LDAP — ciąg dalszy

| Filtr wyszukiwania | Opis |
|--|--|
| L=Berlin | Wszystkie obiekty o lokalizacji Berlin |
| Name=*Berlin* | Wszystkie obiekty o nazwach zawierających słowo Berlin |
| (&(L=Berlin)(ObjectCategory=↪OrganizationalUnit)) | Wszystkie jednostki organizacyjne o lokalizacji Berlin |
| (&(ObjectCategory=OrganizationalUnit)(Name=*Berlin*)) | Wszystkie jednostki organizacyjne, których nazwa zawiera słowo Berlin |
| (&(ObjectCategory=OrganizationalUnit)(Name=*Berlin*)(!L=Berlin)) | Wszystkie jednostki organizacyjne o nazwie zawierającej słowo Berlin, ale nie o lokalizacji Berlin |
| (&(ObjectCategory=OrganizationalUnit)(Name=*Berlin*)(!L=*)) | Wszystkie jednostki organizacyjne o nazwie zawierającej słowo Berlin, ale niemające określonej lokalizacji |
| (&(ObjectCategory=OrganizationalUnit)((L=Berlin)(L=Charlotte))) | Wszystkie jednostki organizacyjne o lokalizacji Berlin lub Charlotte |

Jak widać w przedstawionych przykładach, filtr wyszukiwania można zdefiniować na dwa sposoby. Pierwsza metoda polega na prostym przypisaniu filtra. Składa się on z atrybutu, operatora oraz wartości, jak poniżej:

```
PS C:\> ([ADSI::Searcher]"Name=Charlotte").FindAll() | Select Path
```

Path

```
LDAP://OU=Charlotte,DC=nwtraders,DC=com
```

Druga metoda użycia filtra wyszukiwania LDAP polega na łączeniu kilku filtrów. Najpierw wpisuje się operator, potem filtr A, następnie filtr B. Można łączyć wiele filtrów i operatorów, jak widać w przykładach przedstawionych w tabeli 7.1. Poniżej znajduje się przykład użycia takiego filtra złożonego:

```
PS C:\> ([ADSI::Searcher]"(|(Name=Charlotte)(Name=Atlanta))").FindAll() | Select Path
```

Path

```
LDAP://OU=Atlanta,DC=nwtraders,DC=com
```

```
LDAP://OU=Charlotte,DC=nwtraders,DC=com
```

W tabeli 7.2 znajduje się zestawienie operatorów, których można używać zarówno w prostych, jak i złożonych filtrach.

W tabeli 7.3 znajduje się zestawienie znaków specjalnych. Jeśli któryś z nich musi wystąpić w filtrze wyszukiwania w dosłownym znaczeniu, należy go zastąpić sekwencją specjalną.

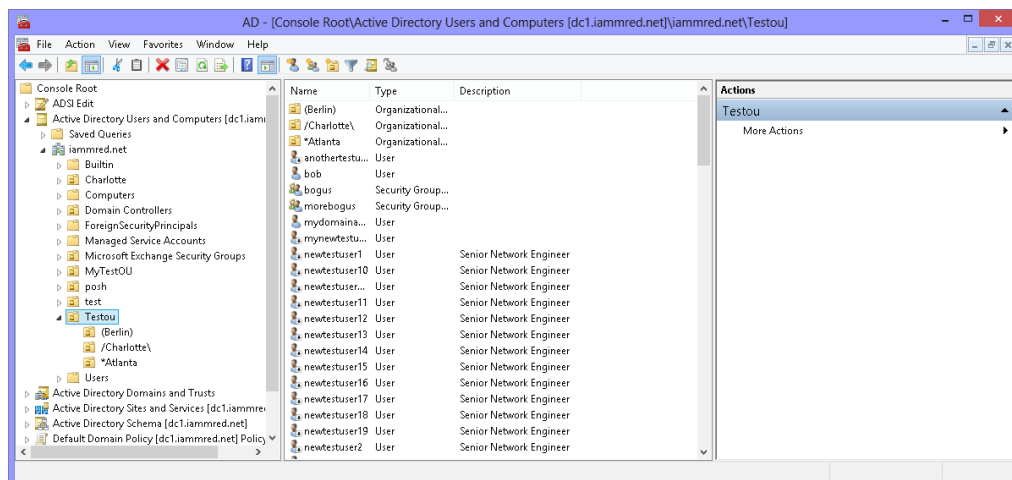
Jak widać na rysunku 7.8, znaki specjalne mogą być używane w nazwach jednostek organizacyjnych w Active Directory.

TABELA 7.2. Logiczne operatory filtrów wyszukiwania LDAP

| Operator | Opis |
|----------|---|
| = | Równość |
| ~= | Równość w przybliżeniu |
| <= | Mniejszy niż lub równy w porządku leksykograficznym |
| >= | Większy niż lub równy w porządku leksykograficznym |
| & | Koniunkcja |
| | Alternatywa logiczna |
| ! | Negacja logiczna |

TABELA 7.3. Znaki specjalne filtrów wyszukiwania LDAP

| Znak ASCII | Zastępcza sekwencja specjalna |
|------------|-------------------------------|
| * | \2a |
| (| \28 |
|) | \29 |
| \ | \5c |
| NUL | \00 |
| / | \2f |



RYSUNEK 7.8. Nazwy jednostek organizacyjnych w Active Directory

Na rysunku tym widać jednostkę organizacyjną o nazwie **Atlanta*. Aby pobrać tę konkretną jednostkę, należy użyć znaku \2a, jak pokazano poniżej:

```
PS C:\> ([ADSI]Searcher)"name=\2aAtlanta").FindAll() | Select Path
```

Path

```
LDAP://OU=*Atlanta,DC=nwtraders,DC=com
```

Aby pobrać jednostkę organizacyjną o nazwie (*Berlin*), należy zgodnie z tabelą 7.3 użyć sekwencji specjalnych \28 i \29, jak pokazano poniżej:

```
PS C:\> ([ADSI]Searcher]"name=\28Berlin\29").FindAll() | Select Path
```

Path

```
LDAP://OU=(Berlin),DC=nwtraders,DC=com
```

Na rysunku 7.8 widać też jednostkę organizacyjną o nazwie (*Charlotte*). Sekwencja specjalna zastępująca ukośnik / to \2f, natomiast lewy ukośnik ma sekwencję specjalną \5c. Aby więc pobrać jednostkę organizacyjną o nazwie (*Charlotte*) przy użyciu filtru wyszukiwania LDAP i akceleratora typu [ADSI]Searcher], należy użyć poniższego zapytania:

```
PS C:\> ([ADSI]Searcher]"name=\2fCharlotte\5c").FindAll() | Select Path
```

Path

```
LDAP://OU=\\Charlotte\\,DC=nwtraders,DC=com
```

Morał

Unikaj stosowania specjalnych znaków w nazwach jednostek organizacyjnych

Zasadniczo staram się nie używać znaków specjalnych w nazwach jednostek organizacyjnych, nazwach użytkowników, nazwach grup, nazwach komputerów itd. Podejrzewam, że nie wszystkie aplikacje potrafią je poprawnie zinterpretować, i zawsze obawiam się, czy nazwa zawierająca znak specjalny na pewno zadziała. Ponadto nawet mimo możliwości zastąpienia znaków specjalnych sekwencjami specjalnymi w wyszukiwaniu nigdy nie jest to intuicyjne i trzeba poświęcić dużo czasu na znalezienie sposobu na zastąpienie znaku sekwencją specjalną. Jeśli doda się do tego fakt, że problemy zwykle występują o godzinie 2 w nocy w niedziele (tak już jest, że wszystkie problemy z siecią zdarzają się o 2 w nocy w niedziele), gdy jest spore ryzyko, że zapomnisz użyć sekwencji specjalnej, to jesteś na najlepszej drodze do katastrofy. To, że coś można zrobić, nie znaczy, że jest to zalecane.

Znaki specjalne filtrów LDAP i odpowiadające im sekwencje specjalne są wymienione w tabeli 7.3.

Przy użyciu polecenia Invoke-Command można z łatwością wykorzystać akcelerator [ADSI]Searcher] do wysyłania zapytań do katalogu Active Directory niezaufanego lasu lub niezaufanej domeny. Wówczas należy podać pełną nazwę domeny komputera, ponieważ gdy używana jest tylko nazwa NetBIOS serwera, istnieje ryzyko, że nazwy nie będą w pełni rozwiązywane. Ponadto najlepiej jest przekazać dane poświadczające jako główną nazwę użytkownika (ang. *User Principal Name* — UPN). Po uruchomieniu polecenia zostaje wyświetlone okno dialogowe, w którym należy wpisać hasło. Polecenie to pokazano poniżej:

```
PS C:\> Invoke-Command -ComputerName Sydney.WoodBridgeBank.Com -Credential '
administrator@WoodBridgeBank.com -ScriptBlock {[ADSI]Searcher]"L=Berlin").FindAll()}
PSComputerName      : sydney.woodbridgebank.com
RunspaceId          : 112f974a-00aa-417c-8a13-9033a49354bd
PSShowComputerName  : True
Path                : LDAP://OU=Berlin Bank,DC=woodbridgebank,DC=com
Properties           : {ou, dscorepropagationdata, whencreated, name...}
```

Posługiwanie się poleceniami Active Directory

Polecenia Active Directory są dostępne od systemu Windows 2008 R2. Znajdują się w module, więc najpierw należy je załadować za pomocą polecenia `Import-Module`. Oczywiście można po prostu wybrać ikonę Active Directory Windows PowerShell, która powoduje uruchomienie konsoli PowerShell z od razu załadowanymi poleceniami Active Directory. Bardzo dobrze, że polecenia Active Directory znajdują się w module, ponieważ dzięki temu za pomocą polecenia `Import-Module` można je dodać ze zdalnego komputera do sesji Windows PowerShell, w której ich brakuje. W tym celu należy wykonać następujące czynności:

1. Ustanów zdalną sesję z serwerem z systemem Windows 2008 R2.
2. Zaimportuj polecenia Active Directory za pomocą polecenia `Import-Module`.
3. Wykonaj zapytanie Active Directory.
4. Zamknij połączenie ze zdalną sesją.
5. Usuń zdalną sesję.

UWAGA

Używając polecenia `Remove-PSSession` z parametrem `-id`, należy pamiętać, że identyfikator sesji nie zawsze jest znany. Identyfikator pierwszej sesji to 1, a drugiej 2. Konsola Windows PowerShell prowadzi rejestr wszystkich sesji. Ale użytkownik może nie wiedzieć, który numer sesji jest aktualny. Dlatego najlepiej jest pobrać listę wszystkich sesji PowerShell za pomocą polecenia `Get-PSSession`. Ponadto mam zwyczaj usuwać odłączone sesje, których nie planuję używać w najbliższej przyszłości. W ten sposób zwalnim trochę zasobów.

Poniżej przedstawiono przykład usuwania nieużywanej sesji:

```
PS C:\> $ps = New-PSSession -ComputerName Sydney.WoodBridgeBank.Com -Credential
administrator@WoodBridgeBank.Com
PS C:\> Enter-PSSession $ps
[sydney.woodbridgebank.com]: PS C:\> Import-Module ActiveDirectory
[sydney.woodbridgebank.com]: PS C:\> Get-ADOrganizationalUnit -Filter "L -eq 'Berlin'"
Name                : Berlin Bank
Country              : DE
PostalCode           :
City                 : Berlin
ManagedBy           :
StreetAddress        :
State                : Berlin
ObjectGUID           : dde90f41-128c-4567-9822-00de5a4c96cc
ObjectClass           : organizationalUnit
DistinguishedName    : OU=Berlin Bank,DC=woodbridgebank,DC=com
[sydney.woodbridgebank.com]: PS C:\> Exit-PSSession
PS C:\> Get-PSSession

    Id Name                ComputerName    State    Configuration
    -- --
    1 Session1            sydney.woodb... Broken    Microsoft.PowerShell

PS C:\> Remove-PSSession -Id 1
```

Oprócz składni filtrów Active Directory, w których używa się operatorów Windows PowerShell i obsługiwane są konwersje typów wzbogaconych, można też używać składni filtrów LDAP opisanej w poprzednim podrozdziale. Aby użyć składni LDAP, należy zamiast parametru `-filter` użyć parametru `-LDAPFilter` oraz w cudzysłowie podać filtr wyszukiwania LDAP, jak pokazano poniżej:

```
PS C:\> Get-ADOrganizationalUnit -LDAPFilter '(L=Berlin)'  
Name                : Berlin Bank  
Country              : DE  
PostalCode           :  
City                 : Berlin  
ManagedBy           :  
StreetAddress        :  
State                : Berlin  
ObjectGUID           : dde90f41-128c-4567-9822-00de5a4c96cc  
ObjectClass           : organizationalUnit  
DistinguishedName    : OU=Berlin Bank,DC=woodbridgebank,DC=com
```

Po prostu używaj wiersza poleceń

Wiele poleceń można wykonać bezpośrednio w zwykłym wierszu poleceń przy użyciu starych narzędzi. Nie ma w tym nic złego i polecenia te są też obsługiwane przez konsolę Windows PowerShell. Wskazówką, że stare narzędzia wiersza poleceń są obsługiwane przez Windows PowerShell, powinno być to, że można je wyszukiwać za pomocą polecenia cmdlet `Get-Command`. Do wyszukiwania plików wykonywalnych za pomocą tego polecenia można używać symboli wieloznacznych, jak pokazano poniżej:

```
PS C:\> Get-Command ds*  
CommandType  Name                Definition  
-----  
Application  ds16gt.dll          C:\Windows\system32\ds16gt.dll  
Application  ds32gt.dll          C:\Windows\system32\ds32gt.dll  
Application  dsa.msc             C:\Windows\system32\dsa.msc  
Application  dsac ls.exe         C:\Windows\system32\dsac ls.exe  
Application  dsadd.exe           C:\Windows\system32\dsadd.exe  
Application  dsadmin.dll         C:\Windows\system32\dsadmin.dll  
Application  dsauth.dll          C:\Windows\system32\dsauth.dll  
Application  dsdbutil.exe        C:\Windows\system32\dsdbutil...  
Application  dsdmo.dll           C:\Windows\system32\dsdmo.dll  
Application  dsget.exe           C:\Windows\system32\dsget.exe  
Application  dskquota.dll        C:\Windows\system32\dskquota...  
Application  dskquoui.dll        C:\Windows\system32\dskquoui...  
Application  ds mgmt.exe         C:\Windows\system32\ds mgmt.exe  
Application  dsmod.exe           C:\Windows\system32\dsmod.exe  
Application  dsmove.exe          C:\Windows\system32\dsmove.exe  
Application  dsound.dll          C:\Windows\system32\dsound.dll  
Application  dsprop.dll          C:\Windows\system32\dsprop.dll  
Application  dsprov.dll          C:\Windows\System32\Wbem\dsp...  
Application  dsprov.mof          C:\Windows\System32\Wbem\dsp...  
Application  dsquery.dll         C:\Windows\system32\dsquery.dll  
Application  dsquery.exe         C:\Windows\system32\dsquery.exe  
Application  dsrm.exe            C:\Windows\system32\dsrm.exe  
Application  dssec.dat           C:\Windows\system32\dssec.dat  
Application  dssec.dll           C:\Windows\system32\dssec.dll  
Application  dsenh.dll           C:\Windows\system32\dsenh.dll  
Application  ds site.msc         C:\Windows\system32\ds site.msc  
Application  dsuixt.dll          C:\Windows\system32\dsuixt.dll  
Application  dsuiwiz.dll         C:\Windows\system32\dsuiwiz.dll  
Application  dswave.dll          C:\Windows\system32\dswave.dll
```

Powyższe polecenie zwraca wszystkie poprawne polecenia Windows PowerShell, wliczając funkcje, polecenia cmdlet oraz pliki wykonywalne. Jeśli chcesz znaleźć tylko narzędzia wiersza poleceń, użyj parametru `commandtype`, jak pokazano poniżej:

```
PS C:\> Get-Command -Name ds* -CommandType application
```

| CommandType | Name | Definition |
|-------------|--------------|---------------------------------|
| ----- | ---- | ----- |
| Application | ds16gt.dll | C:\Windows\system32\ds16gt.dll |
| Application | ds32gt.dll | C:\Windows\system32\ds32gt.dll |
| Application | dsa.msc | C:\Windows\system32\dsa.msc |
| Application | dsacis.exe | C:\Windows\system32\dsacis.exe |
| Application | dsadd.exe | C:\Windows\system32\dsadd.exe |
| Application | dsadmin.dll | C:\Windows\system32\dsadmin.dll |
| Application | dsauth.dll | C:\Windows\system32\dsauth.dll |
| Application | dsdbutil.exe | C:\Windows\system32\dsdbutil... |
| Application | dsdmo.dll | C:\Windows\system32\dsdmo.dll |
| Application | dsget.exe | C:\Windows\system32\dsget.exe |
| Application | dskquota.dll | C:\Windows\system32\dskquota... |
| Application | dskquoui.dll | C:\Windows\system32\dskquoui... |
| Application | dsmgmt.exe | C:\Windows\system32\dsmgmt.exe |
| Application | dsmod.exe | C:\Windows\system32\dsmod.exe |
| Application | dsmove.exe | C:\Windows\system32\dsmove.exe |
| Application | dsound.dll | C:\Windows\system32\dsound.dll |
| Application | dsprop.dll | C:\Windows\system32\dsprop.dll |
| Application | dsprov.dll | C:\Windows\System32\Wbem\dsp... |
| Application | dsprov.mof | C:\Windows\System32\Wbem\dsp... |
| Application | dsquery.dll | C:\Windows\system32\dsquery.dll |
| Application | dsquery.exe | C:\Windows\system32\dsquery.exe |
| Application | dsrm.exe | C:\Windows\system32\dsrm.exe |
| Application | dssec.dat | C:\Windows\system32\dssec.dat |
| Application | dssec.dll | C:\Windows\system32\dssec.dll |
| Application | dsenh.dll | C:\Windows\system32\dsenh.dll |
| Application | dssite.msc | C:\Windows\system32\dssite.msc |
| Application | dsuixt.dll | C:\Windows\system32\dsuixt.dll |
| Application | dsuiwiz.dll | C:\Windows\system32\dsuiwiz.dll |
| Application | dswave.dll | C:\Windows\system32\dswave.dll |

Łatwość obsługi i elastyczność konsoli Windows PowerShell przyczyniły się do nawrotu zainteresowania programami wiersza poleceń. Przykładem może być program *DSQuery.exe*, za pomocą którego użytkownik może szybko wysłać zapytanie do Active Directory. Biorąc pod uwagę, że w systemie Windows Server 2008 R2 dodano akcelerator typu [ADSISeacher] i różne polecenia cmdlet do obsługi Active Directory, możesz się zastanawiać, po co w ogóle ktoś miałby używać takich narzędzi jak *DSQuery.exe*. Poniżej znajduje się polecenie *DSQuery.exe* zwracające listę jednostek organizacyjnych w domenie:

```
PS C:\> dsquery ou
"OU=Domain Controllers,DC=nwtraders,DC=com"
"OU=Students,DC=nwtraders,DC=com"
"OU=ManagedComputers,DC=nwtraders,DC=com"
"OU=TestOU,DC=nwtraders,DC=com"
```

Poniżej znajduje się polecenie zwracające listę jednostek organizacyjnych w składni opartej na użyciu akceleratora [ADSISeacher]:

```
PS C:\> ([ADSI searcher]"objectClass=OrganizationalUnit").findall() | select-object -property path
```

```
Path
```

```
----
```

```
LDAP://OU=Domain Controllers,DC=nwtraders,DC=com
```

```
LDAP://OU=Students,DC=nwtraders,DC=com
```

```
LDAP://OU=ManagedComputers,DC=nwtraders,DC=com
```

```
LDAP://OU=TestOU,DC=nwtraders,DC=com
```

Składnia oparta na poleceniu `Get-ADOrganizationalUnit`, dostępnym w module Active Directory od systemu Windows Server 2008 R2, jest nieco prostsza. Podczas pracy w wierszu poleceń Windows PowerShell Active Directory nie zawsze trzeba podawać nazwy parametrów. Można też używać aliasów (np. `Select` zamiast `Select-Object`). Dzięki aliasom polecenia są krótsze, ale trudniejsze do zmodyfikowania. Poniżej znajduje się przykład użycia polecenia `Get-ADOrganizationalUnit`:

```
PS C:\> Get-ADOrganizationalUnit -Filter "name -like '*'" | Select DistinguishedName
```

```
DistinguishedName
```

```
-----
```

```
OU=Domain Controllers,DC=woodbridgebank,DC=com
```

```
OU=Test1,DC=woodbridgebank,DC=com
```

Jeśli dążysz do maksymalnego skrócenia składni, to najlepszym rozwiązaniem będzie użycie programu *DSQuery.exe*. Ale jeśli masz jeszcze inne wymagania, to lepsze mogą być inne rozwiązania. Program *DSQuery.exe* zwraca łańcuch, podczas gdy akcelerator `[ADSI searcher]` zwraca obiekt `DirectoryEntry`. Z kolei polecenie `Get-ADOrganizationalUnit` zwraca obiekt `Microsoft.ActiveDirectory.Management.ADOrganizationalUnit`. Każda metoda ma wady i zalety, więc wybór jednej z nich zależy od konkretnych potrzeb.

UWAGA

Uważam, że dobrym zwyczajem jest wpisywanie pełnych nazw parametrów nawet podczas pracy w wierszu poleceń w konsoli Windows PowerShell. Choć parametry często można definiować według ich pozycji, to trzeba pamiętać, który jest domyślny, oraz znać kolejność parametrów, jeśli jest ich więcej niż jeden. Często żądane informacje udaje mi się zdobyć dopiero po paru próbach. Po zmianie domyślnego parametru na kilka parametrów modyfikujących, jeśli nie używa się nazw parametrów, składnia ulega zmianie, jak pokazano poniżej:

```
PS C:\> Get-Command ds* application
Get-Command : The command could not be retrieved because the ArgumentList parameter can be
specified only when retrieving a single cmdlet or script.
At line:1 char:12
+ Get-Command <<<< ds* application
+ CategoryInfo          : InvalidArgument: (ds16gt.dLL:ApplicationInfo)
[Get-Command], PSArgumentException
+ FullyQualifiedErrorId :
CommandArgsOnlyForSingleCmdlet,Microsoft.PowerShell.Commands.GetCommandCommand
```

Oprócz kwestii związanej z typem zwrótnym narzędzie *DSQuery.exe* jest obciążone jeszcze pewnymi innymi problemami. Przede wszystkim poświęcono w nim możliwości na rzecz prostoty, co oznacza, że w zapytaniach można używać tylko paru atrybutów. Aby na przykład znaleźć

wszystkie jednostki organizacyjne w Active Directory zawierające nazwę Berlin, można użyć poniższego polecenia:

```
dsquery ou -name *berlin*
```

A żeby znaleźć wszystkie jednostki organizacyjne w Active Directory, których atrybut lokalizacji jest ustawiony na Berlin, należy użyć poleceń cmdlet Active Directory lub akceleratora [ADSISeacher]. Jeśli ktoś dobrze wie, jakie są możliwości narzędzia *DSQuery.exe*, to nie ma nic złego w tym, że go używa. Zwrócone przezeń wyniki można nawet przekazać potokowo do innych narzędzi, np. *DSMove.exe*. Program *DSMove.exe* przenosi obiekt do innej lokalizacji w Active Directory, program *DSMode.exe* służy do zmieniania wartości atrybutów, a *DSrm.exe* służy do usuwania obiektów z Active Directory.

Obliczanie korzyści z użycia skryptu

Jeśli podliczy się czas potrzebny na napisanie, przetestowanie i umieszczenie skryptu w systemie kontroli wersji, może się okazać, że cały proces jest dość czasochłonny. Dlatego też informatycy powinni oszacować potencjalne korzyści z napisania skryptu, zanim zaczną go pisać. Jak już wspomniałem w tym rozdziale, wiele tradycyjnych powodów do pisania skryptów Windows PowerShell straciło już ważność. Nie znaczy to, że nigdy nie trzeba pisać skryptów, tylko że wiele skomplikowanych zadań można wykonać przy użyciu prostych poleceń. Jeśli zastanawiasz się, czy pisać skrypt, czy nie, w tym podrozdziale znajdziesz parę wskazówek.

Zapiski praktyka

Jason A. Yoder, MCT

Prezes firmy MCTExpert, Inc.

Często polecam używanie konsoli Windows PowerShell informatykom, którzy są przyzwyczajeni do tego, że traktuje się ich jako koszt, a nie wartościowe aktywa firmy. Ale nie musi tak być. Odkąd istnieje konsola Windows PowerShell, a jest już czwarta wersja, trudno znaleźć jakiegokolwiek uzasadnienie dla ręcznego wykonywania jakichkolwiek czynności administracyjnych na serwerach firmy Microsoft. W istocie konsola ta umożliwia informatykom pokazanie, że są cennymi pracownikami, i udowodnienie, że warto inwestować w ich pracę.

Wyobraź sobie taką sytuację: za każdym razem, gdy ktoś w firmie uruchamia napisany przez Ciebie skrypt, program ten wysyła pewne ważne informacje do znajdującego się na centralnym serwerze pliku CSV. Zapisywane jest, który skrypt został uruchomiony, kto go uruchomił oraz ile czasu dzięki temu zaoszczędzono. Przy użyciu tych danych informatyk może udowodnić, jak wiele udało się zyskać dzięki niewielkiej inwestycji w napisanie paru skryptów. W niektórych przypadkach można nawet wykazać, że zaoszczędzenie wielu godzin dzięki użyciu konsoli Windows PowerShell pozwoliło odłożyć konieczność zatrudnienia dodatkowych pracowników.

Kiedyś pomagałem pewnemu klientowi, który musiał wyznaczyć jednego pracownika do robienia kopii zapasowej i czyszczenia dzienników zabezpieczeń co trzy godziny każdego dnia na wielu zdalnych serwerach. Dzięki poświęceniu trzech godzin na napisanie i przetestowanie skryptu klient ten poświęca na to samo zadanie mniej niż godzinę rocznie — kiedyś potrzebował 1095 godzin. Nie trzeba być geniuszem finansowym, aby dostrzec, jak dużo można potencjalnie zaoszczędzić. Pamiętam też pewnego użytkownika, który co tydzień musiał wykonywać pewne zadanie związane z używaniem Active Directory i Excela. Zajmowało mu to dwie godziny. Dzięki użyciu konsoli Windows PowerShell zadanie jest wykonywane automatycznie, a osoba, o której piszę, ma gotowy produkt, gdy przychodzi do pracy w poniedziałek — oszczędza 104 godziny pracy.

Pytanie, jakie należy sobie zadać, nie brzmi: „Co mogę zrobić przy użyciu konsoli Windows PowerShell?”, tylko: „Co **chcę** zrobić przy użyciu konsoli Windows PowerShell?”. Gdy odpowiesz sobie na pytanie i wprowadzisz swoją wizję w życie, nie zapomnij oszacować efektów. Może staniesz się w firmie informatyczną gwiazdą i nowym cennym pracownikiem.

Powtarzalność

Jeśli jakieś zadanie trzeba powtórzyć wiele razy, to jest to dobra okazja do napisania skryptu (choć oczywiście nie zawsze). Informatycy często potrzebują informacji o stanie różnych usług, który bez problemu można sprawdzić za pomocą polecenia `Get-Service`. Aby sprawdzić status wybranego procesu, należy użyć parametru `Name`, jak pokazano poniżej:

```
PS C:\> Get-Process -Name powershell
```

| Handles | NPM(K) | PM(K) | WS(K) | VM(M) | CPU(s) | Id | ProcessName |
|---------|--------|-------|-------|-------|--------|-----|-------------|
| 661 | 9 | 42616 | 46024 | 202 | 3.61 | 880 | powershell |

Jeśli trzeba odczytać najnowszy wpis w dzienniku aplikacji, należy użyć dwóch parametrów, jak pokazano poniżej:

```
PS C:\> Get-WinEvent -LogName application -MaxEvents 1
```

| TimeCreated | ProviderName | Id | Message |
|---------------------|--------------|------|---------------------|
| 1/26/2009 10:47:... | VSS | 8193 | Volume Shadow Co... |

W tych przypadkach pisanie skryptu nie ma sensu, ponieważ składnia poleceń jest prosta, łatwa w użyciu i bez trudu można ją sobie przypomnieć, jeśli się zapomni. Wystarczy w razie potrzeby użyć polecenia `Get-Help`.

Jeśli jakąś czynność wykonuje się rutynowo na grupie komputerów, to warto zastanowić się nad napisaniem skryptu. Załóżmy, że musimy sprawdzać poziom fragmentacji na kilku komputerach. Pewnie uda się znaleźć sposób na uruchomienie polecenia bezpośrednio z konsoli Windows PowerShell, ale następnym razem, gdy będzie trzeba to zrobić, znowu spędzisz kilkadziesiąt minut na przypominaniu sobie składni. W takim przypadku lepiej napisać skrypt, np. o nazwie *DefragAnalysisReport.ps1*, co powinno zająć nie więcej niż godzinę. W skrypcie tym można użyć klasy `WMI Win32_Volume`, wywołać metodę `DefragAnalysis` dla każdego dysku komputera i zapisać wyniki w pliku tekstowym.

W skrypcie *DefragAnalysisReport.ps1* najpierw należy utworzyć tablicę nazw komputerów i przypisać ją do zmiennej `$arycomputer`. Można to zrobić zarówno przez wpisanie wartości bezpośrednio w kodzie, jak w tym przykładzie, lub odczytanie wartości z pliku tekstowego za pomocą polecenia `Get-Content`. Następnie należy przypisać wartość ścieżki wyjściowej do zapisania raportu. Ścieżka ta wskazuje folder istniejący na komputerze, na którym zostanie uruchomiony skrypt. Folderu tego nie musi być na komputerze docelowym, ponieważ wszystkie raporty będą przechowywane lokalnie. Poniżej znajduje się opisywana część kodu:

```
$arycomputer = "Windows 8","Berlin"
$FilePath = "C:\fso"
```

Teraz za pomocą instrukcji `Foreach` trzeba przejrzeć tablicę nazw komputerów zapisaną w zmiennej `$arycomputer`, jak pokazano poniżej:

```
Foreach($Computer in $aryComputer)
{
```

Do klasy `WMI Win32_Volume` można wysyłać zapytania za pomocą polecenia `Get-WmiObject`. Klasa ta jest dostępna od systemu Windows 2003. Jeśli planowane jest uruchamianie skryptu w starszych systemach, dobrym zwyczajem jest dodanie mechanizmu obsługi błędów wykrywającego wersję systemu operacyjnego i w razie potrzeby elegancko przechodzącego do następnego komputera. Opis tej techniki znajduje się w rozdziale 6. Poniżej przedstawiono opisywane zapytanie:

```
Get-WmiObject -Class win32_volume -Filter "DriveType = 3" '
-ComputerName $computer |
```

Wyniki zapytania WMI są przekazywane do polecenia `ForEach-Object`. Technika polegająca na przekazywaniu danych przez potok jest nieco bardziej efektywna niż zapisywanie wyników w zmiennej i iterowanie po nich, ponieważ przetwarzanie rozpoczyna się od razu po otrzymaniu pierwszego obiektu. Pierwszą czynnością w poleceniu `ForEach-Object` jest wydrukowanie za pomocą parametru `-Begin` wiadomości informującej o tym, który komputer jest aktualnie sprawdzany, jak pokazano poniżej:

```
ForEach-Object '
-Begin { "Sprawdzanie komputera $computer" } '
```

Rzeczywistą analizę defragmentacji, która ma miejsce raz dla każdego dysku, można wykonać w bloku `Process`. Metoda `DefragAnalysis` jest wywoływana dla obiektu znajdującego się aktualnie w potoku. Zmienna `$_` to zmienna automatyczna odnosząca się do tego obiektu. Metoda `DefragAnalysis` zwraca zarówno raport o błędzie, jak i egzemplarz klasy `WMI Win32_DefragAnalysis`. Obie te informacje są zapisywane w zmiennej `$rtn`, jak pokazano poniżej:

```
-Process {
    "Sprawdzanie poziomu fragmentacji dysku $($_.name). Czekaj..."
    $RTN = $_.DefragAnalysis()
```

W celu utworzenia raportu o poziomie defragmentacji można użyć przekierowania. Pojedynczy znak nawiasu trójkątnego skierowany w prawo (`>`) nadpisuje poprzednie raporty. Jako że istnieje duża szansa, że serwer będzie zawierał więcej niż jeden dysk, lepiej użyć podwójnego nawiasu (`>>`). Inną możliwością jest użycie polecenia `Out-File`, którego zaletą jest to, że pozwala

określić sposób kodowania pliku. Ponadto polecenie to jest bardziej czytelne niż strzałki przekierowania. Dlatego z reguły używam polecenia `Out-File`. Poniżej znajduje się sekcja nagłówka raportu:

```
"Raport nt. defragmentacji dla komputera $computer" >> "$FilePath\Defrag$computer.txt"
"Raport dla dysku $(_.Name)" >> "$FilePath\Defrag$computer.txt"
"Data sporządzenia raportu: $(Get-Date)" >> "$FilePath\Defrag$computer.txt"
"-----" >> "$FilePath\Defrag$computer.txt"
```

Jedną z wielkich zalet konsoli Windows PowerShell jest sposób, w jaki automatycznie wyświetla własności i wartości obiektów. Aby w języku VBScript wydrukować wartość każdej własności, trzeba napisać kilkanaście wierszy kodu. Jak widać tutaj, obiekt zarządzania `Win32_DefragAnalysis`, który jest zapisany we własności `DefragAnalysis`, zostaje przekazany do polecenia `Format-List` w celu usunięcia własności systemowych klasy WMI. Nazwy wszystkich własności systemowych zaczynają się od dwóch znaków podkreślenia (`_`), dzięki czemu można je łatwo wyeliminować za pomocą wyrażenia regularnego wyszukującego tylko własności o nazwach zaczynających się od liter, po których występuje dowolna liczba innych znaków. Otrzymana lista własności z wartościami zostaje przekazana do pliku znajdującego się w lokalizacji określonej przez własność `$filePath`, jak pokazano poniżej:

```
$RTN.DefragAnalysis |
Format-List -Property [a-z]* >> "$FilePath\Defrag$computer.txt"
}
```

Na koniec za pomocą parametru `End` można wydrukować informację, że testowanie na danym komputerze zostało zakończone, jak pokazano poniżej:

```
-END { "Zakończono sprawdzanie komputera $computer" }
} #end foreach computer
```

Poniżej znajduje się cały skrypt *DefragAnalysisReport.ps1*.

DefragAnalysisReport.ps1

```
$arycomputer = "Windows 8","Berlin"
$FilePath = "C:\fso"
Foreach($Computer in $aryComputer)
{
  Get-WmiObject -Class win32_volume -Filter "DriveType = 3" `
    -ComputerName $computer |
  ForEach-Object {
    -Begin { "Sprawdzanie komputera $computer" } `
    -Process {
      "Sprawdzanie poziomu defragmentacji dysku $(_.name). Czekaj..."
      $RTN = $_.DefragAnalysis()
      "Raport nt. defragmentacji dla komputera $computer" >> "$FilePath\Defrag$computer.txt"
      "Raport dla dysku $(_.Name)" >> "$FilePath\Defrag$computer.txt"
      "Data sporządzenia raportu: $(Get-Date)" >> "$FilePath\Defrag$computer.txt"
      "-----" >> "$FilePath\Defrag$computer.txt"
      $RTN.DefragAnalysis |
      Format-List -Property [a-z]* >> "$FilePath\Defrag$computer.txt"
    }
  }
  -END { "Completed testing $computer" }
} #end foreach computer
```

Wiedza tajemna

Stefan Stranger, Senior Premier Field Engineer
Microsoft Corporation

Program System Center Operations Manager zaczął odbierać pakiety zbiorcze aktualizacji przy użyciu usługi Windows Update. Jako że nie wszystkie moje komputery demonstracyjne i testowe są połączone z internetem, potrzebowałem sposobu na odszukanie łączy używanych przez program Operations Manager do pobierania danych z tej usługi.

Usługa Windows Update używa pliku *.cab* znajdującego się pod adresem <http://go.microsoft.com/fwlink/?LinkId=76054>. Jest w nim plik *Package.xml* zawierający łączy, przy użyciu których Operations Manager (i inne produkty) pobiera pliki aktualizacyjne.

Utworzyłem skrypt pobierający ten plik *.cab*, wypakowujący z niego potrzebne pliki i wyszukujący w pliku *.xml* łączy do plików aktualizacji, co pozwoliło mi na aktualizowanie moich środowisk testowych bez podłączania ich do internetu.

Agent aktualizacji pobiera te informacje z archiwum *wsusscn2.cab* udostępnianego do pobrania przez firmę Microsoft pod stałym adresem URL <http://go.microsoft.com/fwlink/?linkid=76054>. Plik ten pobiera także program Microsoft Baseline Security Analyzer (MBSA), który przy jego użyciu sprawdza, czy system ma zainstalowane wszystkie poprawki.

Archiwum zawiera plik katalogowy o nazwie *package.xml*, w którym firma Microsoft indeksuje wszystkie poprawki dotyczące bezpieczeństwa (wraz z zależnościami) dla wszystkich swoich systemów operacyjnych. W pliku tym znajdują się też adresy URL do plików do pobrania, za pomocą których można pobrać wybrane poprawki bezpośrednio z serwerów Microsoftu.

Najpierw należy za pomocą konsoli Windows PowerShell pobrać plik *wsusscn2.cab*:

Pobiera plik .cab Windows Update, aby znaleźć w nim łączy do aktualizacji.

```
$download = "http://go.microsoft.com/fwlink/?LinkId=76054"
```

Pobiera plik .cab.

```
Start-BitsTransfer -Source "http://go.microsoft.com/fwlink/?LinkId=76054" -Destination  
"$env:temp\wsusscn2.cab"
```

Kolejną czynnością jest wydobycie pliku *package.xml* z właśnie pobranego pliku *wsusscn2.cab*.

Użyłem funkcji COM Expand-Cab:

```
Function Expand-Cab ($SourceFile, $TargetFolder, $Item)  
{  
  
    $ShellObject = new-object -com shell.application  
    # Plik ZIP do rozpakowania:  
    $zipFolder = $ShellObject.namespace($sourceFile) # miejsce przechowywania pliku ZIP  
    $item = $zipFolder.parse($Item) # element w pliku ZIP  
    $targetFolder = $ShellObject.namespace("$targetFolder")  
    $targetFolder.copyhere($item)  
}
```

Za pomocą tej funkcji można wypakować zawartość pliku *wsussnc2.cab* do folderu tymczasowego użytkownika:

```
# Wypakowuje plik Package.cab z pliku WSUSsnc2.cab
Expand-Cab -SourceFile "$env:temp\wsussnc2.cab" -TargetFolder "$env:temp" -Item "Package.cab"
```

Po rozpakowaniu pliku *wsussnc2.cab* mamy plik *Package.cab*, który też trzeba rozpakować, aby wydobyć z niego plik *Package.xml*:

```
# Wypakowuje plik Package.xml z pliku Package.cab.
Expand-Cab -SourceFile "$env:temp\Package.cab" -TargetFolder "$env:temp" -Item "Package.xml"
```

Ostatnią czynnością jest odczytanie zawartości pliku *Package.xml* za pomocą konsoli Windows PowerShell:

```
[xml]$wsusupdate = get-content -path "$env:temp\package.xml"
$urls = $wsusupdate.OfflineSyncPackage.FileLocations.FileLocation
```

Stosując filtrowanie przy użyciu nazw artykułów Bazy wiedzy, można znaleźć pliki pakietu zbiorczego aktualizacji dla każdego takiego artykułu:

```
$KBArticle = "KB2750631"
$urls | ? { $_.Url -like "$KBArticle*" } | ft -Property Url -wrap
```

Teraz pobieramy pliki *.cab* dla wybranego artykułu Bazy wiedzy za pomocą polecenia *Start-BitsTransfer*.

```
$urls | ? { $_.Url -like "*KB2750631*" } | select @{L="Source";E={$_.Url}},
@{L="Destination";E={"$env:temp\"+($_.Url) -split "/" }[(($_.Url) -split "/").count -1]}} |
Start-BitsTransfer
```

Możliwość opisanania w dokumentacji

Skrypt zapewnia, że pewne czynności zawsze będą wykonywane w ten sam sposób. Jest to bardzo ważne przy wykonywaniu skomplikowanych zadań konfiguracyjnych oraz przy dokonywaniu prostych zmian w rejestrze. Skrypt dokładnie dokumentuje, co wydarzyło się podczas zmieniania konfiguracji. Jeśli później zostanie w niej znaleziony błąd, można sprawdzić dokumentację wykonanych przez skrypt poleceń i go odpowiednio zmodyfikować, aby cofnąć wprowadzone zmiany.

W poniższym przykładzie w gałęzi rejestru *HKEY_CURRENT_USER* tworzony jest nowy klucz o nazwie *Scripting* oraz kolejny o nazwie *Logon*. Po utworzeniu tych kluczy zostaje utworzona własność o nazwie *ScriptName* i wartości *temp*. Powstałe w ten sposób klucze rejestru widać na rysunku 7.9, a kod użyty do ich utworzenia pokazano poniżej:

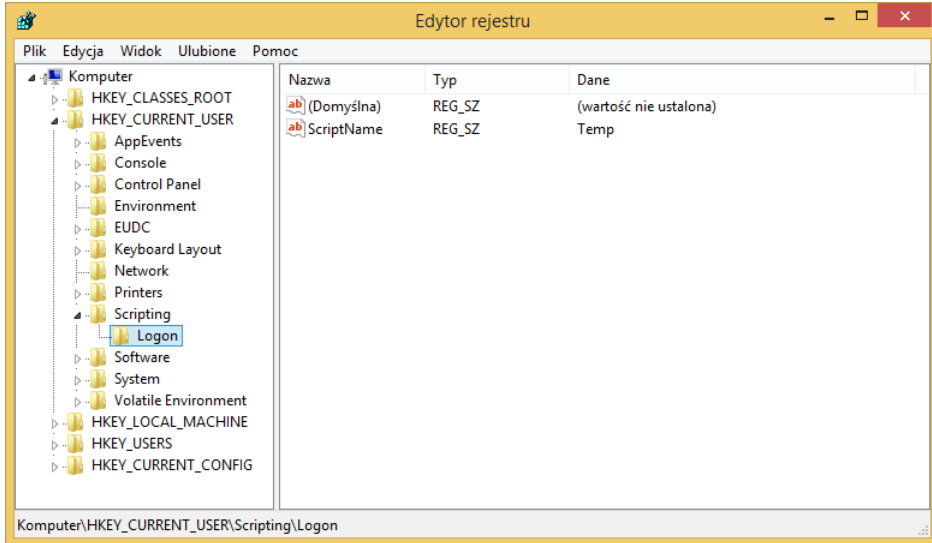
```
PS C:\> New-Item -Path HKCU:\Scripting\Logon -Force
```

```
Hive: HKEY_CURRENT_USER\Scripting
```

| Name | Property |
|-------|----------|
| ---- | ----- |
| Logon | |

```
PS C:\> New-ItemProperty -Path HKCU:\Scripting\Logon -Name ScriptName -Value "Temp"
```

```
ScriptName : Temp
PSPath      : Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER\Scripting\Logon
PSParentPath : Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER\Scripting
PSChildName  : Logon
PSDrive      : HKCU
PSProvider   : Microsoft.PowerShell.Core\Registry
```



RYСУNEK 7.9. Własność ScriptName w kluczu rejestru Logon

Jeśli podczas tworzenia kluczy rejestru i ich wartości wystąpi problem, należy uruchomić Edytor rejestru lub wpisać w konsoli Windows PowerShell parę poleceń. Skrypt z zasady łatwiej się modyfikuje, ponieważ widać w całości wykonywany kod. Polecenia, które wpisano wcześniej w konsoli Windows PowerShell, są pokazane w skrypcie *CreateScriptingRegistryKey.ps1*.

CreateScriptingRegistryKey.ps1

```
New-Item -Path HKCU:\Scripting\Logon -Value "Temp" -Force
New-ItemProperty -Path HKCU:\Scripting\Logon -Name ScriptName -Value "Temp"
```

Jeśli problem wystąpił z wcześniejszym poleceniem, to na podstawie pierwszego skryptu bez problemu można utworzyć nowy skrypt cofający zmiany, jak widać w skrypcie *DeleteScriptingRegistryKey.ps1*. Drugi wiersz tego skryptu jest wyłączony za pomocą komentarza, a w pierwszym zmieniono polecenie `New-Item` na `Remove-Item`. Parametr `-force` zmieniono na `Recurse`, a parametr `value` w poleceniu `Remove-Item` jest niepotrzebny. Zawartość zmodyfikowanego skryptu *DeleteScriptingRegistryKey.ps1* pokazano poniżej:

DeleteScriptingRegistryKey.ps1

```
Remove-Item -Path HKCU:\Scripting -Recurse
#New-ItemProperty -Path HKCU:\Scripting\Logon -Name ScriptName -Value "Temp"
```

Zdolność do adaptacji

W zależności od struktury skryptu można go wykorzystać do wykonywania różnych innych zadań. Jeśli skrypt ma budowę modułową oraz użyto w nim funkcji i argumentów wiersza poleceń, to może być używany do wykonywania rozmaitych czynności. Funkcje można zaimportować przez dołączenie skryptu do innego skryptu. Sam skrypt można też przekształcić w moduł, który później można zaimportować do sesji za pomocą polecenia `Import-Module`.

Przykładem skryptu o budowie modułowej jest skrypt *SaveWmiInformationAsDocument.ps1*. Jego najważniejsze składniki są funkcjami, które można łatwo wykorzystać w innych skryptach.

Wielokrotne wykorzystanie kodu

Możliwość przystosowywania funkcji z jednego skryptu do użycia w innych skryptach często uzasadnia koszt finansowy i czas, jaki należy włożyć w napisanie tego skryptu. Niemniej jednak możliwość wielokrotnego wykorzystania kodu nie powinna być podstawowym celem programisty. Pisanie skryptów o modułowej budowie trwa znacznie dłużej niż zwykłych. Ponadto inwestowanie czasu w możliwość ponownego użycia kodu w bliżej nieokreślonej przyszłości nie zawsze jest dobrym rozwiązaniem. Oczywiście budowanie kodu modułowego jest bardzo dobrym podejściem do programowania, które pozwala uzyskać czytelny i łatwy w modyfikacji kod. Warto dążyć do tych celów projektowych, ale sam potencjał ponownego użycia kodu w przyszłości nie powinien być ostatecznym argumentem za.

Pierwsza funkcja w skrypcie *SaveWmiInformationAsDocument.ps1* nazywa się *CreateWordDoc*. Tworzy ona egzemplarz klasy *Word.Application*, który zapisuje w zmiennej skryptowej *\$word*. Następnie funkcja ta sprawia, że aplikacja Microsoft Office Word staje się widoczna, i dodaje dokument do kolekcji dokumentów, jak pokazano poniżej:

```
Function CreateWordDoc()
{
    $script:word = New-Object -ComObject word.application
    $word.visible = $true
    $Script:doc = $word.documents.add()
} #end CreateWordDoc
```

Kolejna funkcja nazywa się *CreateSelection* i przyjmuje łańcuch, który powinien zostać użyty jako nagłówek dokumentu Office Word. Do utworzenia zaznaczenia w programie Word potrzebny jest egzemplarz klasy *Word.Application*. Jako że zmienna *\$word* należy do zakresu skryptowego, jest dostępna także w funkcji *CreateSelection*. Obiekt *selection* zostaje utworzony przez wysłanie zapytania do własności *selection*. Do wpisania nagłówka do dokumentu Word użyto metody *TypeText*. Następnie tworzony jest pusty akapit i funkcja kończy działanie.

```
Function CreateSelection($Heading)
{
    $script:selection = $word.selection
    $selection.typeText($Heading)
    $selection.TypeParagraph()
} #end CreateSelection
```

Funkcja *GetWmiData* odpytuje klasę WMI, wynik przekształca na łańcuch i zapisuje informacje w dokumencie Word jako zaznaczenie:

```
Function GetWmiData($WmiClass)
{
    Get-WmiObject -class $wmiClass | Out-String |
    ForEach-Object {$selection.typeText($_)}
} #end GetWmiData
```


Podczas pobierania informacji WMI należy utworzyć ścieżkę do pliku za pomocą funkcji `CreateFilePath`, aby można było gdzieś zapisać dokument Word. Funkcja ta pobiera nazwę klasy WMI przez zmienną `$wmiClass`. Następnie za pomocą metody `substring` z klasy `System.String` usuwa sześć pierwszych znaków z nazwy klasy WMI. Te sześć znaków to przedrostek `Win32_` obecny w nazwach prawie wszystkich klas WMI. Aby być dokładniejszym, powinno się sprawdzić jeszcze inne wzorce nazw klas WMI i odpowiednio dostosować polecenie `substring` do znalezionej nazwy klasy. Następnie polecenie `Join-Path` buduje ścieżkę do pliku, która zostaje użyta przy zapisywaniu dokumentacji WMI. Poniżej znajduje się kod źródłowy tej funkcji:

```
Function CreateFilePath($wmiClass)
{
    $script:filename = $wmiClass.substring(6)
    $script:path = Join-Path -Path $folder -childpath $filename
} #end CreateFilePath
```

Następnie trzeba zapisać dokument Word. W tym celu najpierw należy utworzyć egzemplarz wyliczenia `Microsoft.Office.Interop.Word.WdSaveFormat` poprzez dokonanie rzutowania reprezentacji łańcuchowej tego wyliczenia jako typu. Musi to być typ referencyjny, dlatego dodano `[ref]`. Metoda `saveas` z obiektu `Word.Document` wymaga, aby zarówno ścieżka, jak i format zapisu były typami referencyjnymi. Po zapisaniu dokumentu obiekt `Word.Application` można usunąć z pamięci za pomocą metody `quit`. Poniżej znajduje się kompletny kod źródłowy funkcji `SaveWordData`:

```
Function SaveWordData($path)
{
    [ref]$SaveFormat = "microsoft.office.interop.word.WdSaveFormat" -as [type]
    $doc.saveas([ref]$path, [ref]$saveFormat::wdFormatDocument)
    $word.quit()
} #end SaveWordData
```

Punkt początkowy skryptu tworzy parę zmiennych i wywołuje odpowiednie funkcje, jak pokazano poniżej:

```
$folder = "C:\fso"
$wmiClass = "Win32_Bios"
$heading = "$wmiClass information:"
CreateWordDoc
CreateSelection($Heading)
GetWmiData($wmiClass)
CreateFilePath($wmiClass)
SaveWordData($path)
```

Poniżej znajduje się kompletny kod źródłowy opisywanego skryptu *SaveWmiInformation-AsDocument.ps1*:

SaveWmiInformationAsDocument.ps1

```
Function CreateWordDoc()
{
    $script:word = New-Object -ComObject word.application
    $word.visible = $true
    $Script:doc = $word.documents.add()
} #end CreateWordDoc
Function CreateSelection($Heading)
```

```

{
    $script:selection = $word.selection
    $selection.typeText($Heading)
    $selection.TypeParagraph()
} #end CreateSelection

Function GetWmiData($WmiClass)
{
    Get-WmiObject -class $wmiClass | Out-String |
    ForEach-Object {$selection.typeText($_)}
} #end GetWmiData

Function CreateFilePath($wmiClass)
{
    $script:filename = $wmiClass.substring(6)
    $script:path = Join-Path -Path $folder -childpath $filename
} #end CreateFilePath

Function SaveWordData($path)
{
    [ref]$SaveFormat = "microsoft.office.interop.word.WdSaveFormat" -as [type]
    $doc.saveas([ref]$path, [ref]$SaveFormat::wdFormatDocument)
    $word.quit()
} #end SaveWordData

# *** punkt początkowy ***
$folder = "C:\fso"
$wmiClass = "Win32_Bios"
$heading = "$wmiClass information:"
CreateWordDoc
CreateSelection($Heading)
GetWmiData($wmiClass)
CreateFilePath($wmiClass)
SaveWordData($path)

```

Wiedza tajemna

Rejestrowanie pomysłów na skrypty

Chris Bellée, Premier Field Engineer

Microsoft Corporation, Australia

Klienci często proszą mnie o przykładowe skrypty. Często podsuwają dobre pomysły, więc piszę skrypty, które zachowuję do użycia kiedy indziej. Używam do tego Notatnika, który jest łatwy w obsłudze i szybki. Pliki tekstowe odczytują programy wszelkiego typu, więc nie muszę się martwić, czy mam zainstalowany odpowiedni program z pakietu Microsoft Office. Podczas pisania skryptów często odkrywam nowe techniki i technologie. Wówczas od razu piszę przykładowy skrypt z ich użyciem, aby mieć jakiś przykład na przyszłość. Nie jest to formalna technika, ale jej zaletą jest prostota.

Doskonałym pomysłem jest utworzenie bazy danych skryptów, np. w programie Microsoft Office Access. Skrypty można podzielić na kategorie tematyczne według dowolnego kryterium, a następnie można sporządzać raporty wskazujące, w jakich dziedzinach brakuje jeszcze określonych rodzajów skryptów. Później wystarczy tylko przejrzeć raport i uzupełnić braki. Baza danych może służyć nie tylko do przechowywania, ale również do wyszukiwania skryptów. Bardzo ciekawym pomysłem jest utworzenie konstruktora skryptów bazującego na ogólnych procedurach zapisanych w bazie danych. Oczywiście jest to przede wszystkim metoda tworzenia nowych skryptów na podstawie zapisanych pomysłów, a nie wyszukiwania kolejnych okazji do napisania skryptu. Konstruktor skryptów może być rozszerzeniem przenośnego centrum skryptowego dostępnego z Microsoft Script Center.

Gdy zabieram się do napisania nowego skryptu, z reguły używam klas platformy Microsoft .NET, jeśli są dostępne, zamiast funkcji VBScript czy nawet klas WMI, ponieważ lepiej znam platformę .NET i uważam, że jest to najlepsze rozwiązanie. Platforma .NET jest technologią macierzystą dla konsoli Windows PowerShell i o wiele łatwiej jest wywołać interfejsy API Win32 w Windows PowerShell 2.0. Używam tych API, gdy nie mam dostępu do jakiejś klasy w bibliotece .NET, np. do tworzenia udziałów sieciowych i ustawiania ich uprawnień.

Współpraca nad skryptami

Podczas gdy pisanie skryptów może być przyjemne i chyba rzeczywiście wielu administratorów sieci czerpie z tego przyjemność, bywa też czasochłonne. Dlatego proces tworzenia skryptów powinien być tak zaplanowany, aby korzyści z niego miała cała organizacja. Należy pamiętać, że istnieje różnica między nauką pisania skryptów a pisanem skryptów. To prawda, że administratorzy sieci często uczą się poprzez pisanie skryptów, ale mimo wszystko należy rozdzielać te dwie czynności. Nauka pisania skryptów to działalność szkoleniowa i należy ją traktować jako obciążenie budżetu na szkolenia. Natomiast pisanie skryptów to działalność operacyjna. Jeśli administrator sieci osiem godzin pisze skrypt sprawdzający, ile jest wolnego miejsca na dysku serwera, to znaczy, że nie umie pisać skryptów. W związku z tym te osiem godzin należy zaliczyć do obciążeń szkoleniowych, a nie produkcyjnych. Posiadanie przez firmę kilkunastu różnych skryptów sprawdzających ilość wolnego miejsca na dysku komputera nie jest efektywnym ekonomicznie rozwiązaniem.

Jeśli pracownicy różnych działów piszą takie same skrypty, marnuje się czas oraz wysiłek i trzeba pomyśleć o wdrożeniu jakichś narzędzi do współpracy. Przy ich użyciu pracownicy mogą udostępniać swoje skrypty innym oraz zgłaszać zapotrzebowanie na konkretne skrypty. W ten sposób oddziela się działalność szkoleniową od produkcyjnej, a więc oszczędza czas i wysiłek.

Dodatkowe źródła informacji

- Centrum skryptów portalu TechNet pod adresem <http://www.microsoft.com/technet/scriptcenter> zawiera wiele przykładowych skryptów z filtrami wyszukiwania LDAP.

- W artykule <http://support.microsoft.com/kb/947709> w Bazie wiedzy znajdują się informacje na temat sposobów użycia poleceń kontekstu NetSh Advanced Firewall.
- W portalu MSDN na stronie [http://msdn.microsoft.com/en-us/library/aa746385\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/aa746385(VS.85).aspx) znajduje się dokumentacja składni filtrów wyszukiwania LDAP.
- Wszystkie skrypty opisane w tym rozdziale są dostępne do pobrania w repozytorium centrum skryptów portalu TechNet pod adresem <http://gallery.technet.microsoft.com/scriptcenter/PowerShell-40-Best-d9e16039>.

CZĘŚĆ III

Projektowanie skryptów

Rozdział 8. Projektowanie skryptów

Rozdział 9. Projektowanie pomocy do skryptów

Rozdział 10. Projektowanie modułów

Rozdział 11. Obsługa wejścia i wyjścia

Rozdział 12. Obsługa błędów

Rozdział 13. Testowanie skryptów

Rozdział 14. Dokumentowanie skryptów

Rozdział 8

Projektowanie skryptów

- Podstawowe wiadomości o funkcjach
- Wielokrotne wykorzystanie kodu dzięki użyciu funkcji
- Używanie więcej niż dwóch parametrów
- Zapisywanie logiki biznesowej w funkcjach
- Ułatwianie modyfikacji dzięki użyciu funkcji
- Podstawowe wiadomości o filtrach
- Dodatkowe źródła informacji

Aby projektować łatwe w obsłudze i wygodne w użyciu skrypty, wystarczy postępować według prostych zasad. W rozdziale tym poznasz te zasady oraz obejrzysz przykłady dobrze i źle zaprojektowanego kodu.

Podstawowe wiadomości o funkcjach

W konsoli Windows PowerShell 4.0 funkcje stały się pierwszorzędnym elementem programowania skryptów. Stało się tak nie tyle z powodu udoskonalenia samych funkcji jako takich, co raczej dzięki kilku czynnikom, wśród których należy wymienić także dojrzewanie programistów skryptów Windows PowerShell. W starszych wersjach konsoli funkcje były źle rozumiane z powodu braku jasnej dokumentacji na temat sposobów ich użycia, przeznaczenia oraz zastosowań.

Zarówno podprocedury, jak i funkcje są dostępne w języku VBScript. Według klasycznych definicji podprocedury służą do wykonywania takich czynności jak zapis danych w bazie danych czy tworzenie dokumentów Microsoft Office Word. Natomiast funkcje służą do obliczania wartości, które zwracają. Klasycznym przykładem funkcji VBScript jest funkcja konwertująca temperaturę w stopniach Fahrenheita na stopnie Celsjusza. Pobiera ona wartość wyrażoną w stopniach Fahrenheita i zwraca wartość w stopniach Celsjusza. Zgodnie z klasyczną definicją funkcja zawsze powinna zwracać jakąś wartość. Jeśli tego nie robi, to znaczy, że trzeba użyć podprocedury.

UWAGA

Nie trzeba chyba dodawać, że wielu programistów VBScript mieszało pojęcia podprocedury i funkcji. Gdy uczyłem programowania w tym języku, często pytano mnie o to, kiedy należy użyć funkcji, a kiedy podprocedury. Zawsze najpierw przytaczałem klasyczną definicję, a potem pokazywałem studentom, że da się napisać funkcję działającą jak procedura. To była świetna zabawa i studenci bardzo to lubili. Programiści konsoli Windows PowerShell doszli więc do słusznego wniosku. Nie trzeba już zastanawiać się, kiedy używać podprocedur, a kiedy funkcji, bo w Windows PowerShell podprocedur już po prostu nie ma. Są tylko funkcje.

Aby utworzyć funkcję, należy wpisać słowo kluczowe `Function` i nazwę funkcji. Dobrym zwyczajem jest nadawanie funkcjom nazw według wzoru Czasownik-Rzeczownik. Aby nazwa funkcji była jak najłatwiejsza do zapamiętania, czasownik najlepiej wybrać z listy standardowych czasowników Windows PowerShell. Nie powinno się używać czasowników spoza listy, jeśli nie jest to absolutnie konieczne.

UWAGA

Prawie nigdy nie trzeba używać czasowników spoza listy Windows PowerShell. Przez ostatnie pięć lat napisałem ponad 3000 skryptów i nigdy nie musiałem używać jakiegoś niestandardowego czasownika. Standardowe czasowniki są tak dobrze wybrane, że powinny zaspokoić wszystkie potrzeby informatyka piszącego skrypty.

Jeśli chcesz sprawdzić, jak używane są poszczególne czasowniki, możesz wykonać polecenie `Get-Command` i przekazać jego wynik do polecenia `Group-Object`, jak pokazano poniżej:

```
Get-Command -CommandType cmdlet | Group-Object -Property Verb |
Sort-Object -Property count -Descending
```

Jeśli polecenie to zostanie wykonane w systemie Windows 8.1, zwróci wynik pokazany poniżej. Uwzględnione są tylko polecenia `cmdlet`. Jak widać, w standardowych poleceniach najczęściej używany jest czasownik `Get`, a na drugim miejscu znajdują się czasowniki `Set`, `New` i `Remove`.

```
Get-Command -CommandType cmdlet | Group-Object -Property Verb |
Sort-Object -Property count -Descending
```

| Count | Name | Group |
|-------|---------|--|
| ----- | ---- | ----- |
| 91 | Get | {Get-Acl, Get-Alias, Get-AppLockerFileInformation, Get-AppLockerPolicy...} |
| 44 | Set | {Set-Acl, Set-Alias, Set-AppBackgroundTaskResourcePolicy, Set-AppLockerPolicy...} |
| 36 | New | {New-Alias, New-AppLockerPolicy, New-CertificateNotificationTask, New-CimInstance...} |
| 26 | Remove | s {Remove-AppxPackage, Remove-AppxProvisionedPackage, Remove-BitsTransfer, Remove-CertificateEnrollmentPol...} |
| 15 | Add | {Add-AppxPackage, Add-AppxProvisionedPackage, Add-BitsFile, Add-CertificateEnrollmentPolicyServer...} |
| 14 | Export | {Export-Alias, Export-BinaryMiLog, Export-Certificate, Export-Clixml...} |
| 12 | Import | {Import-Alias, Import-BinaryMiLog, Import-Certificate, Import-Clixml...} |
| 12 | Enable | {Enable-AppBackgroundTaskDiagnosticLog, Enable-ComputerRestore, Enable-JobTrigger, Enable-PSBreakpoint...} |
| 12 | Disable | {Disable-AppBackgroundTaskDiagnosticLog, Disable-ComputerRestore, Disable-JobTrigger, Disable-PSBreakpoi...} |
| 10 | Invoke | {Invoke-CimMethod, Invoke-Command, Invoke-Expression, Invoke-History...} |

| | |
|---------------|--|
| 9 Test | {Test-AppLockerPolicy, Test-Certificate, Test-ComputerSecureChannel, Test-Connection...} |
| 9 Clear | {Clear-Content, Clear-EventLog, Clear-History, Clear-Item...} |
| 9 Start | {Start-BitsTransfer, Start-DscConfiguration, Start-DtcDiagnosticResourceManager, Start-Job...} |
| 8 Write | {Write-Debug, Write-Error, Write-EventLog, Write-Host...} |
| 7 Out | {Out-Default, Out-File, Out-GridView, Out-Host...} |
| 6 Stop | {Stop-Computer, Stop-DtcDiagnosticResourceManager, Stop-Job, Stop-Process...} |
| 6 Register | {Register-CimIndicationEvent, Register-EngineEvent, Register-ObjectEvent, Register-PSSessionConfiguratio...} |
| 6 ConvertTo | {ConvertTo-Csv, ConvertTo-Html, ConvertTo-Json, ConvertTo-SecureString...} |
| 5 Format | {Format-Custom, Format-List, Format-SecureBootUEFI, Format-Table...} |
| 4 Update | {Update-FormatData, Update-Help, Update-List, Update-TypeData} |
| 4 ConvertFrom | {ConvertFrom-Csv, ConvertFrom-Json, ConvertFrom-SecureString, ConvertFrom-StringData} |
| 3 Complete | {Complete-BitsTransfer, Complete-DtcDiagnosticTransaction, Complete-Transaction} |
| 3 Receive | {Receive-DtcDiagnosticTransaction, Receive-Job, Receive-PSSession} |
| 3 Rename | {Rename-Computer, Rename-Item, Rename-ItemProperty} |
| 3 Suspend | {Suspend-BitsTransfer, Suspend-Job, Suspend-Service} |
| 3 Resume | {Resume-BitsTransfer, Resume-Job, Resume-Service} |
| 3 Unregister | {Unregister-Event, Unregister-PSSessionConfiguration, Unregister-ScheduledJob} |
| 3 Select | {Select-Object, Select-String, Select-Xml} |
| 3 Wait | {Wait-Event, Wait-Job, Wait-Process} |
| 3 Show | {Show-Command, Show-ControlItem, Show-EventLog} |
| 2 Resolve | {Resolve-DnsName, Resolve-Path} |
| 2 Disconnect | {Disconnect-PSSession, Disconnect-WSMan} |
| 2 Save | {Save-Help, Save-WindowsImage} |
| 2 Split | {Split-Path, Split-WindowsImage} |
| 2 Restart | {Restart-Computer, Restart-Service} |
| 2 Connect | {Connect-PSSession, Connect-WSMan} |
| 2 Move | {Move-Item, Move-ItemProperty} |
| 2 Measure | {Measure-Command, Measure-Object} |
| 2 Join | {Join-DtcDiagnosticResourceManager, Join-Path} |
| 2 Unblock | {Unblock-File, Unblock-Tpm} |
| 2 Undo | {Undo-DtcDiagnosticTransaction, Undo-Transaction} |
| 2 Copy | {Copy-Item, Copy-ItemProperty} |
| 2 Use | {Use-Transaction, Use-WindowsUnattend} |
| 2 Send | {Send-DtcDiagnosticTransaction, Send-MailMessage} |
| 1 Compare | {Compare-Object} |
| 1 Switch | {Switch-Certificate} |
| 1 Tee | {Tee-Object} |
| 1 Sort | {Sort-Object} |
| 1 Checkpoint | {Checkpoint-Computer} |
| 1 Trace | {Trace-Command} |
| 1 Dismount | {Dismount-WindowsImage} |
| 1 Repair | {Repair-WindowsImage} |
| 1 Reset | {Reset-ComputerMachinePassword} |
| 1 Debug | {Debug-Process} |
| 1 Enter | {Enter-PSSession} |
| 1 Exit | {Exit-PSSession} |
| 1 Expand | {Expand-WindowsImage} |
| 1 ForEach | {ForEach-Object} |
| 1 Group | {Group-Object} |
| 1 Restore | {Restore-Computer} |
| 1 Convert | {Convert-Path} |

| | |
|--------------|--------------------------|
| 1 Limit | {Limit-EventLog} |
| 1 Mount | {Mount-WindowsImage} |
| 1 Where | {Where-Object} |
| 1 Push | {Push-Location} |
| 1 Read | {Read-Host} |
| 1 Confirm | {Confirm-SecureBootUEFI} |
| 1 Initialize | {Initialize-Tpm} |
| 1 Pop | {Pop-Location} |

Zapiski praktyka

Juan Carlos Ruiz Lopez, Premier Field Engineer
Microsoft Corporation

Wielu administratorów systemów informatycznych drży na samą myśl o **programowaniu**. Mówią sobie: „Przecież ja jestem administratorem, a nie programistą”. W niektórych krajach krąży nawet złośliwa definicja: „Programista to tania maszyna do zamieniania kawy w kod źródłowy”. Dlatego raczej nie chcesz nim zostać.

Ale niektóre czynności są wielokrotnie powtarzane. Jak utworzyć tysiąc użytkowników, przenieść 200 jednostek organizacyjnych albo szybko przywrócić do działania serwer, który padł ofiarą hakerów w środku nocy, na dodatek mając za plecami ponaglącego nas szefa oraz nie mogąc popełnić nawet najdrobniejszego błędu?

W takim przypadku idealny byłby skrypt. Za pomocą konsoli Windows PowerShell można bardzo łatwo oskryptować dowolne zadania. Większość potrzebnego kodu znajduje się w poleceniach cmdlet, więc nam pozostaje tylko poukładać elementy układanki w potoku (albo kilku potokach).

Nie piszemy kodu, tylko tworzymy skrypty. Każdy to potrafi i nie jest to żadna wielka filozofia, a jaka oszczędność czasu.

Nabierając doświadczenia, będziesz pisać coraz to bardziej złożone skrypty i nawet się nie obejrzyysz, gdy zaczniesz w nich stosować zagnieżdżone pętle, wywoływać metody obiektów platformy .NET itd. Możliwe, że w międzyczasie porzucisz książki o Windows PowerShell i przerzucisz się na podręcznik platformy .NET, a potem **zamiast mówić, że tylko tworzysz skrypty, w rzeczywistości będziesz programować**. Nie ma w tym nic złego. Będzie to dowodem na to, że czegoś się nauczyłeś i że teraz potrafisz jeszcze lepiej wykonywać swoje obowiązki.

Najważniejsze, aby obsługi konsoli Windows PowerShell nauczyć się wtedy, gdy można, a nie wtedy, gdy się musi. W tym drugim przypadku będzie już za późno.

Powinno się opisać za pomocą skryptu każdą czynność, którą trzeba będzie powtórzyć przynajmniej raz. Skrypty mają to do siebie, że jak już są, to używa się ich często.

Funkcja nie musi przyjmować żadnych parametrów i rzeczywiście jest wiele funkcji działających bez podawania im danych wejściowych. Objaśnię to na przykładzie. Administratorzy sieci często muszą sprawdzać wersję systemu operacyjnego. Twórcy skryptów również muszą to robić, aby zapewnić poprawne działanie skryptu lub jego eleganckie zamknięcie, gdyby odpowiednie

interfejsy były niedostępne. Równie często jest tak, że jeden zbiór plików może zostać skopiowany na komputer z jednym systemem operacyjnym, a inny zbiór plików można skopiować do innej wersji systemu operacyjnego.

Pierwszą czynnością przy tworzeniu funkcji jest wybranie nazwy. Jako że nasza funkcja ma pobierać informacje, najlepszym czasownikiem ze standardowej listy czasowników jest `Get`. Jeśli chodzi o część rzeczownikową, to powinna ona odzwierciedlać rodzaj pobieranych przez funkcję informacji. Myślę, że w tym przypadku dobry będzie człon `OperatingSystemVersion`. Przykład implementacji opisywanej funkcji znajduje się w skrypcie *Get-OperatingSystemVersion.ps1*. Funkcja `Get-OperatingSystemVersion` sprawdza wersję systemu operacyjnego przy użyciu Instrumentacji zarządzania Windows (WMI). Najprostsza możliwa forma funkcji składa się ze słowa kluczowego `Function`, nazwy funkcji oraz bloku skryptu w klamrze, jak pokazano poniżej:

```
Function Function-Name
{
    # kod źródłowy
}
```

Na początku skryptu *Get-OperatingSystemVersion.ps1* znajduje się definicja funkcji `Get-OperatingSystemVersion`. Składa się ona ze słowa kluczowego `Function`, za którym znajduje się nazwa `Get-OperatingSystemVersion`. Dalej mamy otwarcie klamry i kod źródłowy. W kodzie tym za pomocą polecenia `Get-WmiObject` pobieramy egzemplarz klasy WMI `Win32_OperatingSystem`. Jako że klasa ta zwraca tylko jeden egzemplarz, jej własności są bezpośrednio dostępne. Nas interesuje wersja systemu operacyjnego, a nawiasy wymuszają wykonanie znajdującego się w nich kodu. Zwrócony obiekt zarządzania zostaje wykorzystany do emisji wersji systemu, a następnie zamykamy funkcję przez zamknięcie klamry. Wersja systemu operacyjnego zostaje zwrócona do kodu, który wywołał funkcję. W tym przykładzie użyto łańcucha "Wersja tego systemu operacyjnego to: ". W celu wymuszenia wykonania funkcji użyto podwyrażenia. Wersja systemu operacyjnego zostaje zwrócona w miejscu wywołania funkcji. Poniżej znajduje się kod źródłowy opisywanego skryptu *Get-OperatingSystemVersion.ps1*.

Get-OperatingSystemVersion.ps1

```
Function Get-OperatingSystemVersion
{
    (Get-WmiObject -Class Win32_OperatingSystem).Version
} #end Get-OperatingSystemVersion

"Wersja tego systemu operacyjnego to: $(Get-OperatingSystemVersion)"
```

W przykładzie wyświetlającym listę czasowników użyłem polecenia `Read-Host`. Czasownikiem w jego nazwie jest słowo `Read` (czytaj), które oznacza w tym przypadku wczytywanie danych z wiersza poleceń. Oznacza to, że czasownik ten nie jest używany w nazwach poleceń odczytujących pliki. Na standardowej liście nie ma czasownika `Display` (wyświetl), a czasownik `Write` (zapisz) jest używany w takich nazwach poleceń, jak `Write-Error` i `Write-Debug`, z których żadne nie odpowiada koncepcji wyświetlania informacji. Jeśli piszesz funkcję wczytującą informacje z pliku tekstowego i wyświetlającą dane statystyczne o tym pliku, możesz ją nazwać `Get-TextStatistics`. Nazwa ta będzie zgodna ze standardowymi nazwami poleceń, takimi jak `Get-Process` czy `Get-Service`, które emitują zdobytą treść. Funkcja `Get-TextStatistics` przyjmuje jeden parametr o nazwie `-path`. Z parametrami funkcji wiąże się taka ciekawostka, że gdy przekazuje się im wartość, to przed ich

nazwami należy wpisać łącznik. Przy odwoływaniu się do wartości wewnątrz funkcji używa się zmiennej, np. `$path`. Funkcję `Get-TextStatistics` można wywołać na kilka sposobów. Pierwszy to wpisanie jej nazwy i podanie wartości parametru w nawiasie, jak pokazano poniżej:

```
Get-TextStatistics("C:\fso\mytext.txt")
```

Jest to typowy sposób wywoływania funkcji, ale metodę tę można stosować tylko wtedy, gdy jest tylko jeden parametr. Innym sposobem na przekazanie wartości do funkcji jest użycie łącznika z nazwą parametru, jak poniżej:

```
Get-TextStatistics -path "C:\fso\mytext.txt"
```

Jak widać, w tym przykładzie nie ma nawiasu. Ponadto można też użyć argumentów pozycyjnych. W takim przypadku opuszcza się nazwę parametru i wpisuje się tylko jego wartość, jak pokazano poniżej:

```
Get-TextStatistics "C:\fso\mytext.txt"
```

Kompromis

Zasady używania argumentów pozycyjnych

Argumenty pozycyjne są dobrym rozwiązaniem, gdy ktoś pracuje w wierszu poleceń i chce przyspieszyć pracę przez wyeliminowanie zbędnego wpisywania. Ale metoda ta jest nieco myląca i dlatego staram się jej unikać, nawet podczas pracy w wierszu poleceń. Staram się nie używać argumentów pozycyjnych, ponieważ często kopiuję kod z konsoli bezpośrednio do skryptów. Wtedy muszę przepisywać polecenia, aby pozbyć się z nich aliasów i argumentów bez nazw. Dzięki funkcji rozwijania nazw za pomocą klawisza *Tab* wprowadzonej w Windows PowerShell 4.0 nie ma sensu oszczędzać czasu przez wpisywanie częściowych nazw argumentów lub całkowite ich opuszczanie. Innym powodem, dla którego zawsze wpisuję nazwy argumentów, jest to, że dzięki temu nie zapominam składni.

Jeszcze innym sposobem przekazywania wartości do funkcji jest używanie częściowych nazw parametrów. Należy wpisać tylko taką część nazwy każdego parametru, która uniemożliwia pomylenie jej z jakąkolwiek inną nazwą. Jeśli na przykład funkcja ma dwa parametry o nazwach zaczynających się od litery *p*, to należy wpisać tyle liter ich nazw, aby nie dało się ich pomylić, jak pokazano poniżej:

```
Get-TextStatistics -p "C:\fso\mytext.txt"
```

Kompletny kod źródłowy funkcji `Get-TextStatistics` znajduje się w skrypcie *Get-TextStatistics.ps1* i poniżej:

Get-TextStatistics.ps1

```
Function Get-TextStatistics($path)
{
    Get-Content -path $path |
    Measure-Object -line -character -word
}
```

W konsoli Windows PowerShell 1.0 było tylko 40 standardowych czasowników, a w konsoli Windows PowerShell 4.0 (podobnie zresztą jak w 3.0) jest ich już 98. Programiści tego narzędzia z dużą rozważą dodają nowe czasowniki i bez wątplenia obecna lista powinna wystarczyć każdemu administratorowi.

UWAGA

Wybór odpowiedniego czasownika do nazwy funkcji jest niezmiernie ważny. Kombinacja Czasownik-Rzeczownik to jedna z najważniejszych cech poleceń; pozwala użytkownikom intuicyjnie domyślać się ich nazw. Stosując konwencjonalne nazwy funkcji, bardzo ułatwiasz ich używanie. Jeśli to zaniedbasz, prawdopodobnie nie zdobędziesz wielu użytkowników.

Po wybraniu czasownika do nazwy funkcji należy wybrać rzeczownik jak najlepiej opisujący jej przeznaczenie. Zwyczajowo, choć nie jest to obowiązująca zasada, używa się rzeczowników w liczbie pojedynczej. Najczęściej zasada ta jest ściśle przestrzegana, podobnie jak zasada dotycząca czasowników. Na przykład poniższe polecenie wyświetla listę funkcji i poleceń, których nazwy kończą się literą s:

```
Get-Command -CommandType cmdlet, function | where noun -match "s$" | select noun -Unique
```

Jak widać, tylko siedem z nich kończy się rzeczownikiem w liczbie mnogiej:

Noun

```
NetNatExternalAddress
SmbShareAccess
AssignedAccess
NetAdapterQos
NetAdapterRss
NetAdapterQos
NetAdapterRss
AssignedAccess
BCStatus
DACConnectionStatus
DnsClientServerAddress
DtcTransactionsStatistics
LogProperties
MpComputerStatus
NetAdapterQos
NetAdapterRss
NetAdapterStatistics
NetIPAddress
NetNatExternalAddress
SmbShareAccess
StartApps
SupportedClusterSizes
SupportedFileSystems
SmbShareAccess
NetIPAddress
VpnServerAddress
NetIPAddress
NetNatExternalAddress
SmbShareAccess
AssignedAccess
DnsClientServerAddress
LogProperties
NetAdapterQos
```

```

NetAdapterRss
NetIPAddress
SmbShareAccess
Process
Alias
Alias
CimClass
Process
Alias
Alias
Alias
Process
Process
Process
Progress

```

Po wybraniu nazwy dla funkcji można utworzyć jej parametry; należy je wpisać w nawiasie okrągłym. Funkcja `Get-TextStatistics` przyjmuje jeden parametr o nazwie `-path`. Jeśli funkcja przyjmuje tylko jeden parametr, to jego wartość można przekazać w nawiasie okrągłym, jak pokazano poniżej:

```
Get-TextLength("C:\fso\test.txt")
```

Ścieżka `C:\fso\test.txt` jest przekazywana do funkcji `Get-TextStatistics` poprzez parametr `-path`. W funkcji tej łańcuch `"C:\fso\test.txt"` znajduje się w zmiennej `$path`. Zmienna ta istnieje tylko w obrębie tej funkcji i jest niedostępna poza nią. Jest natomiast dostępna w zakresach podrzędnych funkcji `Get-TextStatistics`. Zakres podrzędny funkcji `Get-TextStatistics` to taki, który został utworzony w jej wnętrzu. W skrypcie *Get-TextStatisticsCallChildFunction.ps1* funkcja `Write-Path` jest wywoływana w funkcji `Get-TextStatistics`, co oznacza, że funkcja `Write-Path` ma dostęp do zmiennych utworzonych w funkcji `Get-TextStatistics`. Zagadnienie to nazywa się zakresem dostępności zmiennych i jest ono bardzo ważne w kontekście pracy z funkcjami. Jako że w funkcjach tworzy się obiekty, należy uważać, w jakim miejscu dany obiekt jest tworzony i gdzie ma zostać użyty. W skrypcie *Get-TextStatisticsCallChildFunction.ps1* zmienna `$path` otrzymuje wartość dopiero po przekazaniu jej do funkcji, więc istnieje w funkcji `Get-TextStatistics`. Ale ponieważ funkcja `Write-Path` jest wywoływana w funkcji `Get-TextStatistics`, dziedziczy zmienne z jej zakresu. Gdy wywołuje się funkcję lokalną w innej funkcji, zmienne utworzone w funkcji nadrzędnej są dostępne także w funkcji podrzędnej, jak pokazano w poniższym skrypcie *Get-TextStatisticsCallChildFunction.ps1*:

Get-TextStatisticsCallChildFunction.ps1

```

Function Get-TextStatistics($path)
{
    Get-Content -path $path |
    Measure-Object -line -character -word
    Write-Path
}

Function Write-Path()
{
    "Wewnątrz funkcji Write-Path zmienna '$path' ma wartość '$path'"
}

Get-TextStatistics("C:\fso\test.txt")
"Poza funkcją Get-TextStatistics zmienna '$path' ma wartość '$path'"

```

W funkcji `Get-TextStatistics` zmienna `$path` dostarcza ścieżkę do polecenia `Get-Content`. Gdy funkcja `Write-Path` zostaje wywołana, nic nie zostaje do niej przekazane, ale wartość zmiennej `$path` zostaje podtrzymana. Natomiast poza tymi dwiema funkcjami zmienna `$path` nie ma żadnej wartości. Poniżej pokazano wynik wykonania tego skryptu:

| Lines ---- | Words ----- | Characters ----- | Property ----- |
|---------------|----------------|---------------------|-------------------|
| 3 | 41 | 210 | |

Wewnątrz funkcji `Write-Path` zmienna `$path` ma wartość `C:\fso\test.txt`
 Poza funkcją `Get-TextStatistics` zmienna `$path` ma wartość

Następnie należy otworzyć i zamknąć blok skryptu, który powinien znajdować się w klamrze. Dobrym zwyczajem przy definiowaniu funkcji jest wpisanie od razu słowa kluczowego `Function`, nazwy funkcji, parametrów wejściowych funkcji oraz klamry bloku skryptowego, jak pokazano poniżej:

```
Function Moja-Funkcja
{

Param()
# kod źródłowy
} #end My-Function
```

Dzięki temu mamy pewność, że nie zapomnimy zamknąć klamry. W długim skrypcie może być trudno znaleźć, w którym miejscu brakuje klamry, ponieważ błędy nie zawsze występują w tym samym wierszu, w którym powinna znajdować się brakująca klamra. Przypuśćmy, że zapomnieliśmy zamknąć klamry w funkcji `Get-TextStatistics`, jak w skrypcie *GetTextStatisticsCallChildFunction-DoesNOTWork-MissingClosingBracket.ps1*. Spowoduje to wyświetlenie poniższej informacji o błędzie:

```
Missing closing '}' in statement block.
At C:\BestPracticesBook\Get-TextStatisticsCallChildFunction-DoesNOTWork
MissingClosingBracket.ps1:28 char:1
```

Problem polega na tym, że według wyświetlonych informacji błąd dotyczy pierwszego znaku w 28. wierszu skryptu. Ale 28. wiersz tego skryptu jest pustą linią. To znaczy, że konsola Windows PowerShell przeskanowała cały skrypt w poszukiwaniu zamknięcia klamry. Nie znalazła go, więc wydrukowała wiadomość informującą, że problem dotyczy końca skryptu. Jeśli wpiszemy brakujący znak, błąd przestanie występować, ale skrypt i tak nie będzie działał. Poniżej znajduje się kod źródłowy skryptu *GetTextStatisticsCallChildFunction-DoesNOTWork-MissingClosingBracket.ps1* z komentarzem wskazującym miejsce, w którym powinno znajdować się brakujące zamknięcie klamry. Inną techniką pozwalającą ochronić się przed problemem brakującej klamry jest dodawanie komentarza do każdego zamknięcia.

Get-textStatisticsCallChildFunction-DoesNOTWork-MissingClosingBracket.ps1

```
Function Get-TextStatistics($path)
{
  Get-Content -path $path |
  Measure-Object -line -character -word
  Write-Path
  # Tu powinna znajdować się brakująca klamra.
```

```
Function Write-Path()
{
    "Wewnątrz funkcji Write-Path zmienna '$path' ma wartość '$path'"
}
Get-TextStatistics("C:\fso\test.txt")
Poza funkcją Get-TextStatistics zmienna '$path' ma wartość '$path'
```

Zapiski praktyka

Douglas Finke

MVP Microsoft PowerShell

Lubię minimalizować ilość wpisywanych znaków. Dzięki temu oszczędzam czas i robię mniej błędów, zwłaszcza podczas czynności wykonywanych wiele razy w ciągu dnia.

Jedną z takich czynności jest wyszukiwanie wzorców łańcuchowych w plikach tekstowych. Powiedzmy, że chcę znaleźć wszystkie pliki Windows PowerShell, w których używam łańcucha `Test-Path`.

Wówczas przechodzę do katalogu `C:\PowerShell` i wykonuję następujące polecenie:

```
dir . -recurse *.ps1
```

To polecenie zwraca listę wszystkich plików `.ps1` znajdujących się w tym katalogu i jego podkatalogach. Teraz mogę poszukać polecenia `Test-Path`:

```
dir . -recurse *.ps1 | select-string test-path
```

Utworzę funkcję o nazwie `fps`, rekursywnie wyszukującą wszystkie pliki `.ps1`.

```
function fps {dir . -recurse *.ps1}
```

Dzięki temu teraz, będąc w dowolnym miejscu na dysku, mogę wpisać `fps`, aby otrzymać listę plików Windows PowerShell. Połączmy to z aliasem polecenia `Select-String`:

```
fps|sls test-path
```

Tym sposobem zaoszczędziłem 29 znaków. Jeśli operację szukania tekstu przy użyciu konsoli Windows PowerShell wykonam 10 razy dziennie, to zaoszczędzę 290 naciśnieć klawisza, ponad 1000 tygodniowo i ponad 75 000 rocznie.

Oczywiście technika ta nie ogranicza się do plików Windows PowerShell. Za jej pomocą równie dobrze można szukać plików `C#` czy `XAML`:

```
function fx {dir . -recurse *.xml}
function fcs {dir . -recurse *.cs}
```

Definiowanie funkcji w celu ułatwienia wielokrotnego wykorzystania kodu

Gdy do budowy skryptów używa się poprawnie zaprojektowanych funkcji, łatwiej jest je wykorzystać także w innych skryptach oraz wywoływać w konsoli Windows PowerShell.

Aby uzyskać dostęp do funkcji znajdujących się w skrypcie, skrypt ten należy dołączyć za pomocą

notacji kropkowej. Wadą tego rozwiązania jest to, że dołączony skrypt może zawierać zmienne globalne lub inne elementy, których nie powinno być w bieżącym środowisku.

Przykład dobrze napisanej funkcji znajduje się w skrypcie *ConvertToMeters.ps1*. Poza funkcją nie ma żadnych definicji zmiennych globalnych, a sama funkcja nie zawiera polecenia `Write-Host` dzielącego potok. Wyniki konwersji są zwracane bezpośrednio do kodu wywołującego. Jedyny problem z tym skrypcem polega na tym, że gdy zostanie dołączony do konsoli Windows PowerShell, nastąpi jego uruchomienie i zwrot danych, ponieważ cały jego wykonywalny kod zostanie wykonany. Poniżej znajduje się kod źródłowy omawianego skryptu:

ConvertToMeters.ps1

```
Function Script:ConvertToMeters($feet)
{
    "$feet stóp równa się $($feet*.31) metrów"
} #end ConvertToMeters
$feet = 5
ConvertToMeters -Feet $feet
```

Jeśli funkcje są poprawnie zbudowane, to nie ma problemu ze zgromadzeniem ich w jednym skrypcie — wystarczy je skopiować z oryginalnych skryptów i wkleić w nowym miejscu. W ten sposób można utworzyć bibliotekę funkcji.

Wklejając funkcję do biblioteki funkcji, należy zwrócić uwagę na znajdujący się na jej końcu komentarz. Komentarz ten wskazuje koniec funkcji i dodatkowo jest wizualną wskazówką dla programisty szukającego błędów. Przykładem takiej biblioteki funkcji jest skrypt *ConversionFunctions.ps1*, którego zawartość przedstawiono poniżej:

ConversionFunctions.ps1

```
Function Script:ConvertToMeters($feet)
{
    "$feet stóp równa się $($feet*.31) metrów"
} #end ConvertToMeters

Function Script:ConvertToFeet($meters)
{
    "$meters metrów równa się $($meters * 3.28) stóp"
} #end ConvertToFeet

Function Script:ConvertToFahrenheit($celsius)
{
    "$celsius stopni Celsjusza równa się $((1.8 * $celsius) + 32 ) Fahrenheita"
} #end ConvertToFahrenheit

Function Script:ConvertToCelsius($fahrenheit)
{
    "$fahrenheit stopni Fahrenheita równa się $( (($fahrenheit - 32)/9)*5 ) stopni Celsjusza"
} #end ConvertToCelsius

Function Script:ConvertToMiles($kilometer)
{
    "$kilometer kilometrów równa się $( ($kilometer *.6211) ) mil"
} #end convertToMiles

Function Script:ConvertToKilometers($miles)
```

```
{
    "$miles mil równa się $( ($miles * 1.61) ) kilometrów"
} #end convertToKilometers
```

Jednym ze sposobów użycia funkcji ze skryptu *ConversionFunctions.ps1* jest użycie operatora kropki w celu jego wykonania, tak aby znajdujące się w nim funkcje weszły do zakresu wywołań. Aby dołączyć skrypt za pomocą operatora kropki, należy po prostu wpisać kropkę, a po niej ścieżkę do skryptu. Po wykonaniu tej czynności można bezpośrednio wywoływać funkcje, jak pokazano poniżej:

```
PS C:\> . C:\scripts\ConversionFunctions.ps1
PS C:\> convertToMiles 6
6 kilometrów równa się 3.7266 mil
```

Wszystkie funkcje z dołączonego skryptu są dostępne w bieżącej sesji. Można to udowodnić, wyświetlając listę zawartości dysku funkcji, jak pokazano poniżej:

```
PS C:\> dir function: | Where { $_.name -like 'co*' } | Format-Table -Property name, definition -AutoSize
Name                                     Definition
----                                     -
ConvertToMeters                        param($feet) "$feet stóp równa się $($feet*.31) metrów"...
ConvertToFeet                          param($meters) "$meters metrów równa się $($meters * 3.28) stóp"...
ConvertToFahrenheit                    param($celsius) "$celsius stopni Celsjusza równa się $((1.8 * $celsius) + 32 )
stopni Fahrenheita"...
ConvertToCelsius                      param($fahrenheit) "$fahrenheit stopni Fahrenheita równa się
$( (($fahrenheit - 32)/9)*5 ) stopni Celsjusza..."
ConvertToMiles                        param($kilometer) "$kilometer kilometrów równa się $( ($kilometer
*.6211) ) mil"...
ConvertToKilometers                    param($miles) "$miles mil równa się $( ($miles * 1.61) )
kilometrów"...
```

Zapiski praktyka

Czym w istocie są funkcje

Brandon Shell, MVP Windows PowerShell

Moim zdaniem funkcje to ogólnie rzecz biorąc niewielkie narzędzia do wykonywania pojedynczych zadań (jak śrubokręt albo młotek). Wykonują tylko jedną czynność i robią to w sposób niezawodny. Jeśli oprzesz swój kod na funkcjach, będzie on łatwiejszy w diagnozowaniu i bardziej zwięzły. Dlaczego bardziej zwięzły? Ponieważ funkcje można przenosić między skryptami, a nawet wykorzystywać do wykonywania codziennych zadań.

Mam trzy podstawowe wskazówki dla tych, którzy nie wiedzą, kiedy należy pisać funkcje:

1. **Po pierwsze:** odkrywam, że pewien blok kodu wykonuję wielokrotnie. Na przykład mam kod sprawdzający kilka usług na komputerze. Dobrym pomysłem może być zamienienie go w funkcję i wywołanie jej dla każdego serwera. Dzięki temu o wiele łatwiej jest mi rozwiązywać problemy.

2. **Po drugie:** odkrywam, że mogę użyć jakiegoś fragmentu kodu w innych skryptach. Na przykład napisałem elegancki blok rekurencyjnie przetwarzający dane i chciałbym wykorzystać ten kod także w innych skryptach.
3. **Po trzecie:** kod jest przydatny poza skryptem. Ta wskazówka nieco różni się od poprzedniej. Dobrym przykładem jest funkcja wysyłająca ping do serwera, która przydaje się zarówno w skryptach, jak i w codziennej pracy.

Ogólnie rzecz biorąc, każdy pisany fragment kodu należy traktować jako potencjalnego kandydata do wielokrotnego użycia. W szczególności dotyczy to funkcji, które definiuje się właśnie po to, by móc ich użyć wiele razy. Przemyśl, jak i gdzie dana funkcja będzie używana, aby określić odpowiednią listę parametrów i ewentualnych ustawień domyślnych.

Ponieważ kod projektujemy tak, aby nadawał się do wielokrotnego użytku, nie należy w nim oszczędzać na pisaniu. Podstawową zasadą jest, aby nic nie wpisywać na sztywno. Wszystkie dane powinny być przekazywane przez parametry. Oczywiście parametry mogą mieć wartości domyślne, ale powinna być też możliwość ich zmiany w wywołaniu funkcji. Podejście to nazywa się programowaniem czarnej skrzynki. Należy przemyśleć efekt każdej zmiany w oryginalnej funkcji i jak ta zmiana wpłynie na cały skrypt.

Kiedy używałem konsoli Windows PowerShell 1.0, zawsze próbowałem implementować parametry `-verbose` i `-whatif` z własnymi przełącznikami. Od wersji 2.0 jest to robione automatycznie.

Projektując funkcje, należy mieć na uwadze logikę pętlową i przetwarzania. Z reguły dotyczy ona samego skryptu, więc powinna być zaimplementowana poza jakąkolwiek funkcją. Najlepiej, jeśli logika jest ograniczona do tej części skryptu, w której jest potrzebna. Na przykład: jeśli w skrypcie znajduje się logika przetwarzająca w jakiś sposób serwery, powinna znajdować się poza funkcjami, bo nie ma sensu implementować jej w każdej funkcji osobno. Ale jeśli logika bezpośrednio dotyczy dziedziny jakiejś funkcji, to powinna się w niej znaleźć.

Potrzeba jest matką wspaniałych funkcji, ale dojrzewają one dzięki używaniu. Im lepiej będziesz rozumieć zastosowanie funkcji, tym bardziej dojrzale będą stawać się Twoje funkcje. Są one niczym zawsze gotowe do pomocy przyjaciółki, ale podobnie jak przyjaciółki wymagają też uwagi i troski. Poniżej znajduje się lista podstawowych cech, jakie powinna mieć każda funkcja.

Dobrze zdefiniowane parametry

Nie powinno być wątpliwości, jakich danych potrzebuje funkcja, aby wytworzyć oczekiwane informacje. Dlatego należy zdefiniować konkretne parametry (często wraz z podaniem typu danych). Jeśli do działania funkcji jakaś wartość jest bezwzględnie potrzebna, należy zadbać o jej dostarczenie do funkcji. Dobrym sposobem jest przypisanie domyślnej wartości parametru do `(Throw 'Wymagany jest parametr $ThisParam')`.

Spójność wyników i brak niespodzianek

Ta cecha jest absolutnie niezbędna. Niedopuszczalne jest zgadywanie, jakiego typu dane zwróci funkcja. Muszą być takiego typu, jakiego się spodziewasz. Dlatego tak projektuj

funkcje, aby zwracały konkretne typy danych (np. `string`, `DateTime` lub `Boolean`). Uważaj, aby nie zaśmiecić strumienia danych wiadomościami wysyłanymi przez polecenie `Write-Output`.

Samodzielność

Działanie funkcji nie powinno zależeć od jakichkolwiek zmiennych skryptu. Jeśli funkcja potrzebuje do działania danych z zewnątrz, należy je dostarczyć za pomocą parametru.

Przenośność

Bardzo ważną cechą każdej funkcji jest możliwość jej przenoszenia. Jeśli nie planujesz powtórnie wykorzystywać kodu, równie dobrze możesz napisać go bezpośrednio w miejscu użycia zamiast w postaci funkcji. Kluczem do przenośności jest to, aby nazwy zmiennych nie kolidowały ze skrypcem wywołującym.

Definiowanie funkcji z dwoma parametrami

Aby utworzyć funkcję przyjmującą więcej niż jeden parametr wejściowy, należy wpisać słowo kluczowe `Function`, nazwę funkcji, zdefiniować zmienną dla każdego parametru wejściowego w instrukcji `Param` oraz zdefiniować blok kodu w klamrze. Poniżej pokazano opisany wzorzec:

```
Function Moja-Funkcja
{
    Param ($input1, $input2)
    # kod źródłowy
}
```

Przykładem funkcji pobierającej kilka parametrów jest funkcja `Get-FreeDiskSpace` ze skryptu *Get-FreeDiskSpace.ps1*.

Na początku tego skryptu znajduje się słowo kluczowe `Function`, nazwa funkcji oraz dwa parametry wejściowe. Parametry te znajdują się w nawiasie okrągłym, jak widać poniżej:

```
Function Get-FreeDiskSpace
{ Param ($drive,$computer)
```

W bloku kodu funkcji `Get-FreeDiskSpace` wysłano zapytanie do klasy WMI `Win32_LogicalDisk` za pomocą polecenia `Get-WmiObject`. Funkcja `Get-FreeDiskSpace` łączy się z komputerem określonym w parametrze `-computer` i odfiltrowuje wszystkie dyski oprócz określonego w parametrze `-drive`. W wywołaniu podaje się nazwy tych parametrów `-drive` i `-computer`. W definicji funkcji wartości przekazane do tych parametrów są przechowywane w zmiennych `$drive` i `$computer`.

Dane pobrane z WMI zostają zapisane w zmiennej `$driveData` jako egzemplarz klasy `Win32_LogicalDisk`. Zmienna ta zawiera kompletny egzemplarz tej klasy. W tabeli 8.1 znajduje się wykaz jej wszystkich składowych.

TABELA 8.1. Składowe klasy Win32_LogicalDisk

| Nazwa | Typ składowej | Definicja |
|------------------------|---------------|---|
| Chkdsk | Metoda | System.Management.ManagementBase-Object Chkdsk(System.Boolean FixErrors, System.Boolean VigorousIndexCheck, System.Boolean SkipFolderCycle, System.Boolean ForceDismount, System.Boolean RecoverBadSectors, System.Boolean OkToRunAtBootUp) |
| Reset | Metoda | System.Management.ManagementBaseObject Reset() |
| SetPowerState | Metoda | System.Management.ManagementBaseObject SetPowerState(System.UInt16 PowerState, System.String Time) |
| Access | Własność | System.UInt16 Access {get;set;} |
| Availability | Własność | System.UInt16 Availability {get;set;} |
| BlockSize | Własność | System.UInt64 BlockSize {get;set;} |
| Caption | Własność | System.String Caption {get;set;} |
| Compressed | Własność | System.Boolean Compressed {get;set;} |
| ConfiManagerErrorCode | Własność | System.UInt32 ConfiManagerErrorCode {get;set;} |
| ConfiManagerUserConfi | Własność | System.Boolean ConfiManagerUserConfi {get;set;} |
| CreationClassName | Własność | System.String CreationClassName {get;set;} |
| Description | Własność | System.String Description {get;set;} |
| DeviceID | Własność | System.String DeviceID {get;set;} |
| DriveType | Własność | System.UInt32 DriveType {get;set;} |
| ErrorCleared | Własność | System.Boolean ErrorCleared {get;set;} |
| ErrorDescription | Własność | System.String ErrorDescription {get;set;} |
| ErrorMethodology | Własność | System.String ErrorMethodology {get;set;} |
| FileSystem | Własność | System.String FileSystem {get;set;} |
| FreeSpace | Własność | System.UInt64 FreeSpace {get;set;} |
| InstallDate | Własność | System.String InstallDate {get;set;} |
| LastErrorCode | Własność | System.UInt32 LastErrorCode {get;set;} |
| MaximumComponentLength | Własność | System.UInt32 MaximumComponentLength {get;set;} |
| MediaType | Własność | System.UInt32 MediaType {get;set;} |
| Name | Własność | System.String Name {get;set;} |
| NumberOfBlocks | Własność | System.UInt64 NumberOfBlocks {get;set;} |

TABELA 8.1. Składowe klasy Win32_LogicalDisk — ciąg dalszy

| Nazwa | Typ składowej | Definicja |
|------------------------------|---------------|--|
| PNPDeviceID | Własność | System.String PNPDeviceID {get;set;} |
| PowerManagementCapabilities | Własność | System.UInt16[] PowerManagementCapabilities {get;set;} |
| PowerManagementSupported | Własność | System.Boolean PowerManagementSupported {get;set;} |
| ProviderName | Własność | System.String ProviderName {get;set;} |
| Purpose | Własność | System.String Purpose {get;set;} |
| QuotasDisabled | Własność | System.Boolean QuotasDisabled {get;set;} |
| QuotasIncomplete | Własność | System.Boolean QuotasIncomplete {get;set;} |
| QuotasRebuilding | Własność | System.Boolean QuotasRebuilding {get;set;} |
| Size | Własność | System.UInt64 Size {get;set;} |
| Status | Własność | System.String Status {get;set;} |
| StatusInfo | Własność | System.UInt16 StatusInfo {get;set;} |
| SupportsDiskQuotas | Własność | System.Boolean SupportsDiskQuotas {get;set;} |
| SupportsFileBasedCompression | Własność | System.Boolean SupportsFileBasedCompression {get;set;} |
| SystemCreationClassName | Własność | System.String SystemCreationClassName {get;set;} |
| SystemName | Własność | System.String SystemName {get;set;} |
| VolumeDirty | Własność | System.Boolean VolumeDirty {get;set;} |
| VolumeName | Własność | System.String VolumeName {get;set;} |
| VolumeSerialNumber | Własność | System.String VolumeSerialNumber {get;set;} |
| __CLASS | Własność | System.String __CLASS {get;set;} |
| __DERIVATION | Własność | System.String[] __DERIVATION {get;set;} |
| __DYNASTY | Własność | System.String __DYNASTY {get;set;} |
| __GENUS | Własność | System.Int32 __GENUS {get;set;} |
| __NAMESPACE | Własność | System.String __NAMESPACE {get;set;} |
| __PATH | Własność | System.String __PATH {get;set;} |
| __PROPERTY_COUNT | Własność | System.Int32 __PROPERTY_COUNT {get;set;} |
| __RELPATH | Własność | System.String __RELPATH {get;set;} |
| __SERVER | Własność | System.String __SERVER {get;set;} |
| __SUPERCLASS | Własność | System.String __SUPERCLASS {get;set;} |

TABELA 8.1. Składowe klasy Win32_LogicalDisk — ciąg dalszy

| Nazwa | Typ składowej | Definicja |
|---------------------|------------------|---|
| PSStatus | Zbiór własności | PSStatus {Status, Availability, DeviceID, StatusInfo} |
| ConvertFromDateTime | Metoda skryptowa | System.Object ConvertFromDateTime(); |
| ConvertToDateTime | Metoda skryptowa | System.Object ConvertToDateTime(); |

Pobieranie danych WMI

Wprawdzie zapisanie całego egzemplarza obiektu w zmiennej `$driveData` nie jest najbardziej efektywnym rozwiązaniem, ale opisywana klasa jest niewielka, a zalety polecenia `Get-WmiObject` przeważają nad tą wadą. Gdyby jednak wydajność była priorytetem, lepiej byłoby użyć akceleratora [WMI]. Aby sprawdzić ilość wolnego miejsca na dysku, można użyć następującego polecenia:

```
([wmi]"Win32_logicalDisk.DeviceID='c:'").FreeSpace
```

Aby polecenie to umieścić w przydatnej funkcji, należałoby zamienić wpisaną na sztywno literę dysku na zmienną. Ponadto należałoby zmienić konstruktor klasy, aby pobierał ścieżkę do zdalnego komputera. Nowo utworzona funkcja znajduje się w skrypcie *Get-DiskSpace.ps1*, którego zawartość pokazano poniżej:

Get-DiskSpace.ps1

```
Function Get-DiskSpace($drive,$computer)
{
    ([wmi]"\\$computer\root\cimv2:Win32_logicalDisk.DeviceID='$drive']").FreeSpace
}
Get-DiskSpace -drive "C:" -computer "Office"
```

Po wprowadzeniu tych zmian kod zwraca tylko wartość własności `FreeSpace` dla określonego dysku. Jeśli wynik ten przekaże się do polecenia `Get-Member`, można się dowiedzieć, że została zwrócona liczba całkowita. Technika ta jest bardziej efektywna niż zapisywanie całego egzemplarza klasy `Win32_LogicalDisk` i pobieranie z niego pojedynczej wartości.

Po zapisaniu danych w zmiennej `$driveData` należy gdzieś wydrukować informacje dla użytkownika skryptu. Przede wszystkim można wydrukować nazwę komputera i dysku przez umieszczenie zmiennych w podwójnych cudzysłowach. Zmienne w takich cudzysłowach są zamieniane na ich wartości. Poniżej znajduje się opisywany fragment kodu:

```
"Wolna przestrzeń w komputerze $computer na dysku $drive"
```

Następnie za pomocą łańcuchów formatujących platformy Microsoft .NET można zdefiniować format danych zawierający dwa miejsca po przecinku. Aby uniknąć rozwinięcia całego obiektu WMI w cudzysłowie, należy użyć podwyrażenia. W podwyrażeniu tym wpisujemy znak dolara

i nawias okrągły wymuszający wykonanie kodu przed zwróceniem danych do łańcucha, jak pokazano poniżej:

```
$(("{0:n2}" -f ($driveData.FreeSpace/1MB)) megabajtów
```

Get-FreeDiskSpace.ps1

```
Function Get-FreeDiskSpace
{
    Param ($drive,$computer)
    $driveData = Get-WmiObject -class win32_LogicalDisk '
    -computername $computer -filter "Name = '$drive'"
    "
    Wolna przestrzeń w komputerze $computer na dysku $drive
    $("{0:n2}" -f ($driveData.FreeSpace/1MB)) megabajtów
    "
}

Get-FreeDiskSpace -drive "C:" -computer "dc1"
```

Zapiski praktyka

Jason Hofferle

Informatyk

Wiele źródeł informacji i dyskusji dotyczących konsoli Windows PowerShell jest przeznaczonych dla pracujących w firmach informatyków zarządzających serwerami. Przy całej masie książek na temat zastosowań Windows PowerShell do zarządzania takimi produktami jak Exchange, vSphere i inne pracownicy obsługi klienta mogą odnosić wrażenie, że dla nich to narzędzie nie jest przydatne. Ale w rzeczywistości pracownicy obsługi klienta mają tyle samo powodów do używania konsoli Windows PowerShell co administratorzy serwerów i nawet osoby rzadko mające styczność z serwerami powinny jak najszybciej nauczyć się obsługi tego programu.

Zdalny pulpit i inne podobne technologie są niezwykle przydatne przy pomaganiu zdalnym użytkownikom, ale niektóre zadania można o wiele efektywniej wykonać przy użyciu konsoli Windows PowerShell. Wśród przykładów takich czynności można wymienić skopiowanie nowej wersji pliku konfiguracyjnego, ponowne uruchomienie usługi czy odblokowanie konta. Podczas gdy początkowo wykonywanie zadań bez użycia myszy może zabierać więcej czasu niż przy jej użyciu, inwestycja ta zwróci się dziesięciokrotnie już przy pierwszej okazji, gdy jakieś zadanie trzeba będzie wykonać na wielu komputerach.

W wielu organizacjach do wdrażania programów, wprowadzania zmian i wykonywania innych zautomatyzowanych czynności używa się różnego rodzaju rozwiązań dla przedsiębiorstw. Narzędzia te są świetne, ale czasami pracownicy obsługi klienta nie mają do nich dostępu, przez co nawet tak proste czynności jak utworzenie skrótu na pulpicie dla użytkownika trzeba odłożyć na później. Jako że większość lokalnych administratorów systemów informatycznych ma dostęp z uprawnieniami administratora do stacji roboczych, za które odpowiada, wykonanie czynności typu utworzenie skrótu na pulpicie przy użyciu

konsoli Windows PowerShell jest dla nich bardzo łatwe. Przy użyciu tej konsoli można tworzyć automatyczne poprawki, dzięki którym oszczędzamy czas bez polegania na rozwiązaniach dla przedsiębiorstw.

Nie każdy musi być ekspertem od Windows PowerShell, ponieważ zawsze można skorzystać z modułów zawierających skomplikowane skrypty napisane przez bardziej zaawansowanych użytkowników. W czasach gdy prym wiodł język VBS, udokumentowanie skryptu lub napisanie dla niego wewnątrzskryptowej dokumentacji byłoby męką. Natomiast dzięki pomocy opartej na komentarzach i modułom skryptów udostępnianie zautomatyzowanych poprawek innym jest bardzo łatwe. W organizacji, w której pracuję, opiekuję się modulem zawierającym parę funkcji wykonujących pewne skomplikowane zadania. Dzięki nim nawet początkujący może od razu zacząć pracować produktywnie; wystarczy, że zdobędzie podstawowe wiadomości, na przykład do czego służy polecenie `Get-Help`. Gdy użytkownik nabędzie doświadczenia w pracy z konsolą Windows PowerShell, może zacząć tworzyć rozwiązania przydatne także dla innych i w ten sposób zyskać uznanie w branży.

Ale najważniejszym bodajże powodem, dla którego warto nauczyć się biegłej obsługi konsoli Windows PowerShell, jest możliwość rozwoju kariery zawodowej. Niewiele jest firm, które nie używają produktów firmy Microsoft. A jak wiadomo, produktami firmy Microsoft można zarządzać za pomocą konsoli Windows PowerShell. Dzięki jej spójności podstawowe koncepcje można zastosować w taki sam sposób praktycznie wszędzie. Naucz się obsługi Windows PowerShell teraz, a gdy zostaniesz administratorem Directory lub Exchange, umiejętności te bardzo Ci się przydadzą.

Ograniczenia typu

Jeśli funkcja przyjmuje jakieś parametry, powinno się zdefiniować ograniczenia typów, aby uniemożliwić przekazanie do niej danych niewłaściwego typu. W tym celu można wpisać alias typu w kwadratowym nawiasie przed nazwą parametru wejściowego. To spowoduje, że parametr ten będzie przyjmował dane wyłącznie określonego typu. W tabeli 8.2 znajduje się zestawienie dopuszczalnych skrótów nazw typów.

TABELA 8.2. Aliasy typów danych

| Alias | Typ |
|----------|--|
| [int] | 32-bitowa liczba całkowita ze znakiem |
| [long] | 64-bitowa liczba całkowita ze znakiem |
| [string] | Łańcuch o stałej długości znaków Unicode |
| [char] | 16-bitowy znak Unicode |
| [bool] | Wartość typu prawda lub fałsz |
| [byte] | 8-bitowa liczba całkowita bez znaku |
| [double] | 64-bitowa liczba zmiennoprzecinkowa o podwójnej precyzji |

TABELA 8.2. Aliasy typów danych — ciąg dalszy

| Alias | Typ |
|-------------|--|
| [decimal] | 128-bitowa liczba dziesiętna |
| [single] | 32-bitowa liczba zmiennoprzecinkowa o pojedynczej precyzji |
| [array] | Tablica wartości |
| [xml] | Obiekt klasy XmlDocument |
| [hashtable] | Obiekt tablicy mieszającej (podobny do słownika) |

W funkcji `Resolve-ZipCode`, zdefiniowanej w skrypcie *Resolve-ZipCode.ps1*, parametr `-zip` przyjmuje tylko 32-bitowe liczby całkowite ze znakiem. (Oczywiście ogranicznik typu `[int]` uniemożliwia wpisanie kodów pocztowych z większości państw na świecie, ale ten skrypt dotyczy tylko kodów pocztowych używanych w USA).

W funkcji `Resolve-ZipCode` najpierw można użyć łańcucha wskazującego kod WSDL (ang. *Web Services Description Language*) usługi internetowej. Następnie polecenie `New-WebServiceProxy` tworzy nowego pośrednika dla usługi sieciowej `ZipCode`. Kod WSDL usługi `ZipCode` zawiera definicję metody o nazwie `GetInfoByZip`, która przyjmuje kod pocztowy w standardowym formacie obowiązującym w USA. Wyniki są wyświetlane w tabeli. Kod źródłowy skryptu *Resolve-ZipCode.ps1* jest pokazany poniżej.

```
Resolve-ZipCode.ps1
#Requires -Version 4.0
Function Resolve-ZipCode([int]$zip)
{
    $URI = "http://www.websvc.net/uszip.aspx?WSDL"
    $zipProxy = New-WebServiceProxy -uri $URI -namespace WebServiceProxy -class ZipClass
    $zipProxy.getinfozip($zip).table
} #end Get-ZipCode

Resolve-ZipCode 28273
```

Gdy parametr wejściowy ma zdefiniowane ograniczenie typu, wszelkie odstępstwa od wyznaczonego typu powodują błąd podobny do poniższego:

```
Resolve-ZipCode : Cannot process argument transformation on parameter 'zip'. Cannot convert value
"COW" to type "System.Int32". Error: "Input string was not in a correct format."
At C:\Users\edwils.NORTHAMERICA\AppData\Local\Temp\tmp3351.tmp.ps1:22 char:16
+ Resolve-ZipCode <<<< "COW"
    + CategoryInfo          : InvalidData: (:) [Resolve-ZipCode], ParameterBindin...mationException
    + FullyQualifiedErrorId : ParameterArgumentTransformationError,Resolve-ZipCode
```

Nie trzeba dodawać, że takie powiadomienie o błędzie może drażnić użytkowników funkcji. Jednym z możliwych rozwiązań jest użycie słowa kluczowego `Trap`. W skrypcie *DemoTrapSystemException.ps1* znajduje się funkcja `My-Test` zawierająca zmienną o nazwie `$myinput`, w której dzięki ograniczeniu typu `[int]` można zapisywać tylko 32-bitowe liczby całkowite bez znaku. Gdy w wywołaniu funkcji zostanie przekazana taka właśnie liczba, funkcja zwróci napis `To działa`. Natomiast w pozostałych przypadkach nastąpi wyświetlenie informacji o błędzie, podobnej do przedstawionej wcześniej.

Zamiast wyświetlać surową wiadomość o błędzie, która dla większości zwykłych użytkowników i niektórych informatyków jest niezrozumiała, lepiej ją stłumić i wyświetlić własne, bardziej rzeczowe informacje, które użytkownik będzie mógł później przekazać pracownikowi obsługi klienta. W powiadomieniach o błędzie często wyświetla się numer telefonu do odpowiedniego działu firmy albo odnośnik do wewnętrznej strony internetowej zawierającej wskazówki dotyczące samodzielnego rozwiązania problemu. Można nawet dostarczyć stronę zawierającą skrypt rozwiązujący dany problem. Jest to rozwiązanie w rodzaju stron „Napraw mi to” firmy Microsoft.

W chwili utworzenia egzemplarza klasy `System.SystemException` (gdy wystąpi wyjątek systemowy) instrukcja `Trap` przechwytuje błąd, uniemożliwiając wyświetlenie na ekranie informacji o tym błędzie. Jeśli sprawdzisz zawartość zmiennej `$error`, odkryjesz, że błąd rzeczywiście wystąpił i został zarejestrowany. Ponadto masz też dostęp do klasy `ErrorRecord` poprzez zmienną automatyczną `$_`, co oznacza, że informacja o błędzie jest przekazywana przez potok, dzięki czemu można opracować zaawansowany mechanizm obsługi błędów. W tym przykładzie wyświetlany jest napis `Błąd został przechwycony` i instrukcja `Continue` powoduje dalsze wykonywanie skryptu od następnego wiersza kodu. W wierszu tym znajduje się instrukcja wyświetlająca napis `Po błędzie`. Po uruchomieniu skryptu `DemoTrapSystemException.ps1` zostanie wyświetlony następujący wynik:

```
Błąd został przechwycony
Po błędzie
```

Poniżej znajduje się kompletny kod źródłowy skryptu `DemoTrapSystemException.ps1`:

DemoTrapSystemException.ps1

```
Function My-Test([int]$myinput)
{

    "To działa"
} #End my-test function
# *** punkt początkowy skryptu ***

Trap [SystemException] { "Błąd został przechwycony" ; continue }
My-Test -myinput "string"
"Po błędzie"
```

Zapiski praktyka

Juan Carlos Ruiz Lopez, Premier Field Engineer
Microsoft Corporation

Wszystkie moje skrypty zaczynam pisać w taki sam sposób, bez względu na to, jak są proste (z czasem większość z nich i tak znacznie rozbudowuję).

Na początku wpisuję komentarz zawierający nazwę, wersję, datę utworzenia i opis skryptu. Aby zapewnić jak najlepszą czytelność, komentarz ten formatuję w sposób podobny do pokazanego poniżej:

```
# -----
# Script      : Murgifly.ps1
# Description : Skrypt murgujący borostwory...
# Author      : JC@company.com
# Date        : 32 sierpnia 2013
# Version     : V 1.0, teraz V2 zawiera dodatkowe cechy...
# -----
```

Ponadto często dodają jakieś słowa kluczowe, jak poniżej:

```
# -----
# Keywords    : Jabberwocky, borostwory
# -----
```

Oprócz pełnienia funkcji dokumentacyjnej ważnej dla innych użytkowników (jak również po pewnym czasie dla samego autora) komentarz ten pomaga także w znalezieniu zagubionego skryptu. Zazwyczaj pamiętam jakieś słowo kluczowe albo co dany skrypt robił, ale zapominam ścieżki do niego, i wtedy znajduję go za pomocą poniższego polecenia:

```
Dir -Recurse -Include *.ps1 | Select-String "boros"
```

Następnie wpisuję kod źródłowy, na przykład:

```
# wymaga-Version 4
$Error.Clear ()
Set-StrictMode -Version Latest
```

Pierwszy wiersz wygląda jak komentarz, ale wymusza użycie odpowiedniej wersji konsoli Windows PowerShell. W ten sposób zapewniamy, że skrypt nie przestanie nagle działać w środku wykonywania z powodu braku jakiejś nowej funkcji.

Drugi wiersz kasuje zawartość tablicy \$Error. Gdy używa się zmiennej globalnej, lepiej mieć do dyspozycji pustą kolekcję na błędy niż zapełnioną starymi danymi.

W końcu w trzecim wierszu ustawiamy tryb z ograniczeniami. To powinno być obowiązkowe ustawienie we wszystkich skryptach (i według mnie powinno być domyślne... ale nie jest).

Dzięki takiemu podejściu można znaleźć wiele błędów w swoich skryptach i zaoszczędzić sobie wielu godzin pracy. Wychwycisz wszystkie literówki, niezdefiniowane zmienne, niepoprawne wywołania, nieistniejące własności itd. Zajrzyj do dokumentacji.

Najlepiej jest utworzyć sobie szablonowy plik zawierający ten kod i na jego podstawie tworzyć każdy nowy skrypt. Można też użyć funkcji definiowania gotowych fragmentów kodu w konsoli Windows PowerShell ISE 4.0. Wystarczy zdefiniować nowy fragment zawierający przedstawiony kod, a potem można go użyć, naciskając kombinację klawiszy *Ctrl+J*. Zobacz też `Get-Help *snippet*`.

Funkcje przyjmujące więcej niż dwa parametry

Funkcja przyjmująca więcej niż dwa parametry moim zdaniem powinna mieć nieco inną budowę. Zmiana ta dotyczy wyglądu i ma na celu zwiększenie czytelności kodu. Poniżej znajduje się podstawowy wzór funkcji przyjmującej trzy parametry. Parametry łańcuchowe mające wartości domyślne mogą być dość długie. Przeniesienie ich do kodu głównego funkcji podkreśla, że są parametrami wejściowymi, i ułatwia ich zrozumienie, odczytanie oraz obsługę.

```
Function Nazwa-Funkcji
{
    Param(
        [int]$Parametr1,
        [String]$Parametr2 = "WartośćDomyślna",
        $Parametr3
    )
    # kod funkcji
} #end Nazwa-Funkcji
```

Przykładem funkcji przyjmującej trzy parametry jest `Get-DirectoryListing`, której kod źródłowy znajduje się poniżej. Ze względu na ograniczenia typów, domyślne wartości oraz nazwy parametrów sygnatura tej funkcji jest zbyt długa, aby wygodnie zmieścić ją całą w jednej linijce, jak widać poniżej:

```
Function Get-DirectoryListing ([String]$Path,[String]$Extension = "txt",[Switch]$Today)
```

Jeśli liczba parametrów zostanie zwiększona do czterech albo zostanie zdefiniowana wartość domyślna dla parametru `-path`, sygnatura może zająć nawet dwie linijki. W bloku `Param` w bloku głównym funkcji można także określić parametry wejściowe funkcji.

Za słowem kluczowym `Function` i nazwą funkcji wpisuje się słowo kluczowe `Param` oznaczające blok definicji parametrów tej funkcji. Parametry oddziela się przecinkami i wszystkie umieszcza się w nawiasie okrągłym. Jeśli któryś z parametrów ma mieć wartość domyślną, jak parametr `Extension` w poniższym przykładzie, można ją normalnie przypisać, jak do zmiennej.

W widocznej poniżej funkcji `Get-DirectoryListing` parametr `Today` jest przełącznikiem. Gdy zostanie przekazany do funkcji, zostaną wyświetlone tylko pliki zapisane po północy dnia, w którym skrypt został uruchomiony. Jeśli parametr ten nie zostanie podany, zostaną wyświetlone wszystkie znajdujące się w folderze pliki o określonym rozszerzeniu. Poniżej znajduje się kompletny kod źródłowy skryptu *Get-DirectoryListing.ps1*.

Get-DirectoryListing.ps1

```
Function Get-DirectoryListing
{
    Param(
        [String]$Path,
        [String]$Extension = "txt",
        [Switch]$Today
    )
    If($Today)
    {
        Get-ChildItem -Path $path\* -include *.$Extension |
        Where-Object { $_.LastWriteTime -ge (Get-Date).Date }
    }
    ELSE
    {
        Get-ChildItem -Path $path\* -include *.$Extension
    }
} #end Get-DirectoryListing

# *** punkt początkowy skryptu ***
Get-DirectoryListing -p c:\fso -t
```

WAŻNE

W miarę możliwości należy wystrzegać się definiowania funkcji o dużej liczbie parametrów wejściowych, ponieważ łatwo można je pomylić. Jeśli będziesz tworzyć funkcję o dużej liczbie parametrów, zastanów się, czy na pewno stosujesz optymalne rozwiązanie. Duża liczba parametrów często wskazuje, że funkcja służy do wykonywania więcej niż jednej czynności. W funkcji `Get-DirectoryListing` zdefiniowałem parametr przełącznikowy filtrujący zwrócone pliki i pozostawiający tylko te, które zostały zapisane dzisiaj. Gdybym pisał skrypt przeznaczony do użycia w środowisku produkcyjnym, a nie tylko służący do zademonstrowania technik tworzenia funkcji przyjmującej kilka parametrów, utworzyłbym inną funkcję, np. o nazwie `Get-FilesByDate`. W funkcji tej znajdowałyby się przełącznik `Today` i parametr `Date`, aby umożliwić wybór daty do filtra. Technika ta, polegająca na użyciu kilku parametrów, pozwala na oddzielenie funkcjonalności gromadzenia danych od filtrowania. Więcej informacji na ten temat znajduje się w podrozdziale „Definiowanie funkcji w celu ułatwienia modyfikacji skryptów”.

Definiowanie logiki biznesowej w funkcjach

Twórcy skryptów muszą pamiętać o dwóch rodzajach logiki. Pierwszy to logika programu, a drugi to logika biznesowa. **Logika programu** to sposób działania skryptu, kolejność wykonywania poszczególnych czynności oraz wymagania kodu. Przykładem logiki programu jest wymóg otwarcia połączenia z bazą danych przed wysłaniem do niej zapytania.

Logika biznesowa to zbiór zasad będących wymogiem dziedziny, ale niekoniecznie programu lub skryptu. Skrypt często działa bez zarzutu niezależnie od szczegółów zasad biznesowych. Jeśli skrypt jest poprawnie zaprojektowany, powinien działać bez względu na reguły biznesowe.

W skrypcie *BusinessLogicDemo.ps1* znajduje się funkcja `Get-Discount` obliczająca upust od sumy końcowej. Definicja reguł biznesowych tego upustu będzie działać, dopóki nie zmieni się kontrakt między funkcją a wywołującym ją kodem. W klamrze tej funkcji można wpisać dowolnie wymyślny algorytm obliczania upustów, zawierający odwołania do bazy danych sprawdzające stan magazynowy, godzinę, tydzień i ogólną wielkość sprzedaży w miesiącu, jak również licząc punktów lojalnościowych klienta czy obliczający pierwiastek kwadratowy jakiejś losowej liczby używanej do wyznaczania wielkości upustu.

Czym jest kontrakt z funkcją? Treść kontraktu z funkcją `Get-Discount` jest następująca: „Jeśli podasz mi stawkę upustu jako wartość typu `System.Double` i kwotę jako liczbę całkowitą, zwrócę ci liczbę reprezentującą kwotę upustu”. Dopóki dotrzymujemy warunków umowy, nie musimy modyfikować kodu.

Definicja funkcji `Get-Discount` zaczyna się od słowa kluczowego `Function`, nazwy funkcji oraz definicji dwóch parametrów wejściowych. Pierwszy z nich nazywa się `-rate` i ma ograniczenie do typu `System.Double`, co oznacza, że przyjmuje tylko ułamki dziesiętne. Drugi parametr nazywa się `-total` i ma ograniczenie do typu `System.Integer`, co oznacza, że nie przyjmuje ułamków dziesiętnych. W bloku skryptu wartość parametru `-total` jest mnożona przez wartość parametru `-rate`. Wartość tego działania jest zwracana do potoku.

Poniżej znajduje się kod opisywanej funkcji `Get-Discount`:

```
Function Get-Discount([double]$rate,[int]$total)
{
    $rate * $total
} #end Get-Discount
```

W punkcie początkowym skryptu przypisywane są wartości zmiennym `$total` i `$rate`, jak pokazano poniżej:

```
$rate = .05
$total = 100
```

W zmiennej `$discount` zapisywany jest wynik obliczeń wykonywanych przez funkcję `Get-Discount`. Przy jej wywołaniu najlepiej używać pełnych nazw parametrów. Dzięki temu kod będzie bardziej czytelny i odporny na problemy w przypadku zmiany sygnatury funkcji.

```
$discount = Get-Discount -rate $rate -total $total
```

WAŻNE

Sygnatura funkcji to uporządkowanie i nazwy parametrów wejściowych. Jeśli wartości parametrów zostaną podane według kolejności, a ta się zmieni w sygnaturze, kod nie zadziała albo, co gorsza, zacznie zwracać niepoprawne wyniki. Jeśli ktoś do wywoływania funkcji używa częściowych nazw parametrów i do sygnatury zostanie dodany nowy parametr, skrypt może nie zadziałać z powodu niejednoznaczności nazw parametrów. Oczywiście trzeba to brać pod uwagę przy pisaniu skryptu i funkcji, ale problem może pojawić się nawet kilka lat później podczas modyfikowania skryptu lub wywoływania funkcji poprzez inny skrypt.

Pozostała część skryptu wyświetla informacje na ekranie. Poniżej znajduje się wynik wykonania tego skryptu:

```
Kwota: 100
Upust: 5
Suma końcowa: 95
```

Poniżej znajduje się kod źródłowy skryptu *BusinessLogicDemo.ps1*:

BusinessLogicDemo.ps1

```
Function Get-Discount
{
    Param ([double]$rate,[int]$total)
    $rate * $total
} #end Get-Discount

$rate = .05
$total = 100
$discount = Get-Discount -rate $rate -total $total
"Kwota: $total"
"Upust: $discount"
"Suma końcowa: $($total-$discount)"
```

Logika biznesowa nie musi wiązać się z celami biznesowymi. Może nią być cokolwiek, co nie wpływa na działanie kodu. W skrypcie *FindLargeDocs.ps1* znajdują się dwie funkcje. Pierwsza nazywa się `Get-Doc` i wyszukuje dokumenty (pliki z rozszerzeniami `.doc`, `.docx` i `.dot`) w folderze przekazanym w wywołaniu. Dzięki dodatkowemu użyciu polecenia cmdlet `Get-ChildItem` i przełącznika `recurse` funkcja `Get-Doc` przeszukuje też podfoldery. Jest to samodzielna funkcja, niezależna od jakiegokolwiek innej funkcji.

`LargeFiles` to filtr. Filtr to specjalny typ funkcji, którą definiuje się przy użyciu słowa kluczowego `Filter`. Poniżej znajduje się kod źródłowy skryptu *FindLargeDocs.ps1*.

FindLargeDocs.ps1

```
Function Get-Doc
{
    Param ($path)
    Get-ChildItem -Path $path -include *.doc,*.docx,*.dot -recurse
} #end Get-Doc

Filter LargeFiles($size)
{
    $_ | Where-Object length -ge $size
} #end LargeFiles

Get-Doc("C:\FS0") | LargeFiles 1000
```

Definiowanie funkcji w celu ułatwienia modyfikacji skryptów

Truizmem jest stwierdzenie, że żaden skrypt nigdy tak naprawdę nie jest dokończony. Zawsze coś można dodać, np. można poprawić lub wzbogacić funkcjonalność na czyjąś prośbę. Gdy skrypt zostanie napisany jednym ciągiem bez użycia funkcji, łatwo o pomyłkę przy jego modyfikowaniu.

Przykład takiego skryptu znajduje się w pliku *InLineGetIPDemo.ps1*. W jego pierwszym wierszu występuje polecenie cmdlet `Get-WmiObject` pobierające egzemplarze klasy WMI `Win32_NetworkAdapterConfiguration` z własnością `IPEnabled` ustawioną na `true`. Wynik tego zapytania zostaje zapisany w zmiennej `$IP`. Poniżej znajduje się opisywany fragment kodu:

```
$IP = Get-WmiObject -class Win32_NetworkAdapterConfiguration -Filter "IPEnabled = $true"
```

Po pobraniu i zapisaniu informacji WMI pozostała część skryptu drukuje je na ekranie. Własności `IPAddress`, `IPSubNet` oraz `DNSServerSearchOrder` są zapisane w tablicy. W tym przykładzie interesuje nas tylko pierwszy adres IP, więc drukujemy element o indeksie 0, który zawsze istnieje, jeśli karta sieciowa ma adres IP. Poniżej znajduje się opisywana część skryptu:

```
"Adres IP: " + $IP.IPAddress[0]
"Podsieć: " + $IP.IPSubNet[0]
"Brama: " + $IP.DefaultIPGateway
"Serwer DNS: " + $IP.DNSServerSearchOrder[0]
"Nazwa FQDN: " + $IP.DNSHostName + "." + $IP.DNSDomain
```

Wynik działania tego skryptu może wyglądać tak:

```
Adres IP: 192.168.2.5
Podsieć: 255.255.255.0
Brama: 192.168.2.1
Serwer DNS: 192.168.2.1
Nazwa FQDN: Windows 8.nwtraders.com
```

Poniżej znajduje się kompletny kod źródłowy skryptu *InLineGetIPDemo.ps1*:

InLineGetIPDemo.ps1

```
$IP = Get-WmiObject -class Win32_NetworkAdapterConfiguration -Filter "IPEnabled = $true"
"Adres IP: " + $IP.IPAddress[0]
"Podsieć: " + $IP.IPSubNet[0]
"Brama: " + $IP.DefaultIPGateway
"Serwer DNS: " + $IP.DNSServerSearchOrder[0]
"Nazwa FQDN: " + $IP.DNSHostName + "." + $IP.DNSDomain
```

Wystarczy tylko parę zmian w tym skrypcie, aby stał się on znacznie bardziej elastyczny. Oczywiście trzeba przenieść kod do funkcji. Zgodnie z najlepszymi praktykami funkcja powinna być ściśle zdefiniowana i wykonywać jedną konkretną czynność. Choć można cały kod tego skryptu umieścić w funkcji, lepiej tego nie robić. Skrypt ten wykonuje dwa zadania. Po pierwsze: pobiera z WMI informacje dotyczące adresu IP, a po drugie: formatuje i wyświetla te informacje. Lepiej oddzielić proces zbierania danych od procesu ich prezentowania, ponieważ są to dwie różne pod względem logicznym czynności.

Aby w skrypcie *InLineGetIPDemo.ps1* zdefiniować funkcję, wystarczy dodać słowo kluczowe *Function*, wymyślić nazwę funkcji oraz otoczyć pierwotny kod klamrą. Tak przekształcony skrypt nazwałem *GetIPDemoSingleFunction.ps1*, a jego kod widać poniżej:

GetIPDemoSingleFunction.ps1

```
Function Get-IPDemo
{
$IP = Get-WmiObject -class Win32_NetworkAdapterConfiguration -Filter "IPEnabled = $true"
"Adres IP: " + $IP.IPAddress[0]
"Podsieć: " + $IP.IPSubNet[0]
"Brama: " + $IP.DefaultIPGateway
"Serwer DNS: " + $IP.DNSServerSearchOrder[0]
"Nazwa FQDN: " + $IP.DNSHostName + "." + $IP.DNSDomain
} #end Get-IPDemo
# *** punkt początkowy skryptu ***
Get-IPDemo
```

Skoro zadajemy sobie tyle trudu, aby zamienić zwykły kod w funkcję, to powinniśmy mieć z tego jakieś korzyści. Dzięki tej zmianie kod jest:

- bardziej czytelny,
- bardziej zrozumiały,
- łatwiejszy w użyciu,
- łatwiejszy w diagnostyce.

Skrypt jest bardziej czytelny, ponieważ nie trzeba czytać całego kodu, aby dowiedzieć się, co robi. Widzimy funkcję pobierającą adres IP. To wszystko, co robi ten skrypt.

Skrypt jest bardziej zrozumiały, ponieważ od razu widzimy w nim funkcję pobierającą adres IP. Jeśli chcemy dowiedzieć się, jak dokładnie to robi, czytamy kod tej funkcji. Jeśli nas to nie interesuje, pomijamy tę część kodu.

Skrypt jest łatwiejszy w użyciu, ponieważ można go dołączać do innych skryptów za pomocą kropki. Gdy skrypt jest dołączony do innego skryptu, zostaje wykonany cały znajdujący się w nim kod wykonywalny. W efekcie pojawia się poniższy wynik, ponieważ każdy skrypt drukuje informacje:

Adres IP: 192.168.2.5
 Podsieć: 255.255.255.0
 Brama: 192.168.2.1
 Serwer DNS: 192.168.2.1
 Nazwa FQDN: Windows 8.nwtraders.com

Wolna przestrzeń w komputerze Windows 8 na dysku C:
 48,767.16 megabajtów

Wersja tego systemu operacyjnego to: 6.0.6001

Poniżej przedstawiono kod źródłowy skryptu *DotSourceScripts.ps1*. Jak widać, skrypt ten umożliwia wybór informacji oraz ich mieszanie zgodnie z potrzebą. Gdyby każdy skrypt pisano w standardowy sposób, a wyniki były standaryzowane, efekty byłyby bardziej imponujące. Na razie te trzy linijki kodu zwracają niezwykle przydatne informacje, których można użyć w wielu sytuacjach.

DotSourceScripts.ps1

```
. C:\BestPracticesBook\GetIPDemoSingleFunction.ps1
. C:\BestPracticesBook\Get-FreeDiskSpace.ps1
. C:\BestPracticesBook\Get-OperatingSystemVersion.ps1
```

Skrypt *GetIPDemoSingleFunction.ps1* jest łatwiejszy do diagnozowania częściowo dzięki temu, że jest czytelniejszy i bardziej zrozumiały. Ponadto jeśli skrypt zawiera kilka funkcji, można je testować pojedynczo, aby wyizolować problematyczny fragment kodu.

Lepszym podejściem do definiowania funkcji jest uprzednie określenie, co funkcja ta ma robić. W skrypcie *FunctionGetIPDemo.ps1* znajdują się dwie funkcje. Pierwsza łączy się z WMI i zwraca obiekt zarządzania. Druga formatuje dane wyjściowe. Są to dwie w żaden sposób niepowiązane ze sobą funkcje. Pierwsza pobiera dane, a druga je wyświetla. Poniżej znajduje się kod źródłowy skryptu *FunctionGetIPDemo.ps1*:

FunctionGetIPDemo.ps1

```
Function Get-IPObject
{
  Get-WmiObject -class Win32_NetworkAdapterConfiguration -Filter "IPEnabled = $true"
} #end Get-IPObject

Function Format-IPOutput($IP)
{
  "Adres IP: " + $IP.IPAddress[0]
  "Podsieć: " + $IP.IPSubNet[0]
  "Brama: " + $IP.DefaultIPGateway
  "Serwer DNS: " + $IP.DNSServerSearchOrder[0]
  "Nazwa FQDN: " + $IP.DNSHostName + "." + $IP.DNSDomain
} #end Format-IPOutput

# *** punkt początkowy skryptu

$ip = Get-IPObject
Format-IPOutput($ip)
```

Dzięki rozdzieleniu czynności gromadzenia danych i ich prezentacji na dwie różne funkcje uzyskaliśmy większą elastyczność. Funkcję `Get-IPObject` można bez problemu tak zmodyfikować, aby szukała kart sieciowych z wyłączonym IP. W tym celu należy zmienić parametr `-filter` polecenia `Get-WmiObject`. Jako że najczęściej potrzebne są tylko informacje o kartach sieciowych z włączonym IP, wydaje się rozsądnym ustawić domyślną wartość parametru wejściowego na `true`. Domyślnie zmodyfikowana funkcja działa tak jak przed modyfikacją. Ale jej zaletą jest teraz to, że można zmieniać zwracane przez nią obiekty. W tym celu przy wywoływaniu należy przekazać wartość `$false`, jak pokazano w skrypcie *Get-IPObjectDefaultEnabled.ps1*:

Get-IPObjectDefaultEnabled.ps1

```
Function Get-IPObject([bool]$IPEnabled = $true)
{
    Get-WmiObject -class Win32_NetworkAdapterConfiguration -Filter "IPEnabled = $IPEnabled"
} #end Get-IPObject
Get-IPObject -IPEnabled $False
```

Dzięki oddzieleniu funkcji zbierającej informacje od funkcji prezentującej informacje zyskaliśmy elastyczność nie tylko pod względem typu pobieranych danych, ale również w odniesieniu do sposobu ich prezentowania. Gdy pobierane są dane karty sieciowej z wyłączoną obsługą IP, wyniki nie są tak imponujące jak w przypadku karty sieciowej z włączoną obsługą IP. Dlatego można utworzyć inną formę prezentacji do przedstawiania tylko istotnych informacji. Jako że funkcja wyświetlająca informacje różni się od funkcji je gromadzącej, opisującą zmianę można bardzo łatwo wprowadzić. Sekcja `Begin` funkcji jest wykonywana raz w każdym uruchomieniu tej funkcji. Jest to więc doskonałe miejsce do utworzenia nagłówka dla danych wyjściowych. Sekcja `Process` jest wykonywana raz dla każdego elementu w potoku. W tym przykładzie jest wykonywana dla każdej karty sieciowej z wyłączoną obsługą IP. Polecenie `Write-Host` zapisuje dane do konsoli Windows PowerShell. Znak `'t` reprezentuje tabulator.

UWAGA

Znak tabulacji `'t` jest łańcuchem, więc można go używać w poleceniach przyjmujących łańcuchy.

Poniżej znajduje się kod źródłowy skryptu *Get-IPObjectDefaultEnabledFormatNonIPOutput.ps1*:

Get-IPObjectDefaultEnabledFormatNonIPOutput.ps1

```
Function Get-IPObject
{
    Param ([bool]$IPEnabled = $true)
    Get-WmiObject -class Win32_NetworkAdapterConfiguration -Filter "IPEnabled = $IPEnabled"
} #end Get-IPObject

Function Format-NonIPOutput
{
    Param ($IP)
    Begin { "Index # Description" }
    Process {
        ForEach ($i in $ip)
        {
```

```

    Write-Host $i.Index 't $i.Description
  } #end ForEach
} #end Process
} #end Format-NonIPOutPut

$ip = Get-IPObject -IPEnabled $False
Format-NonIPOutput($ip)

```

Za pomocą funkcji `Get-IPObject` można pobrać konfigurację karty sieciowej. Następnie za pomocą funkcji `Format-NonIPOutput` i `Format-IPOutput` można sformatować wyświetlane informacje. Jeśli umieścimy te wszystkie funkcje w jednym skrypcie, np. o nazwie *CombinationFormatGetIPDemo.ps1*, efekt będzie taki, jak pokazano poniżej.

CombinationFormatGetIPDemo.ps1

```

Function Get-IPObject
{
    Param ([bool]$IPEnabled = $true)
    Get-WmiObject -class Win32_NetworkAdapterConfiguration -Filter "IPEnabled = $IPEnabled"
} #end Get-IPObject

Function Format-IPOutput
{
    Param ($IP)
    "Adres IP: " + $IP.IPAddress[0]
    "Podsieć: " + $IP.IPSubNet[0]
    "Brama: " + $IP.DefaultIPGateway
    "Serwer DNS: " + $IP.DNSServerSearchOrder[0]
    "Nazwa FQDN: " + $IP.DNSHostName + "." + $IP.DNSDomain
} #end Format-IPOutput

Function Format-NonIPOutput
{
    Param ($IP)
    Begin { "Index # Description" }
    Process {
        ForEach ($i in $ip)
        {
            Write-Host $i.Index 't $i.Description
        } #end ForEach
    } #end Process
} #end Format-NonIPOutPut

# *** punkt początkowy ***
$IPEnabled = $false
$ip = Get-IPObject -IPEnabled $IPEnabled
If($IPEnabled) { Format-IPOutput($ip) }
ELSE { Format-NonIPOutput($ip) }

```

Zaskakujący sposób działania instrukcji return

James Craig Burley, starszy programista testów
Microsoft Corporation

Nasz zespół cały czas uczy się obsługi konsoli Windows PowerShell i niedawno odkryliśmy zaskakującą cechę instrukcji `return` *wyr*;, która nie zwraca wyrażenia *wyr*; do wywołującego, tylko je dołącza na końcu listy innych obiektów, które zostały „zwrócone” wcześniej. Obiekty te są zwracane, ponieważ są nieprzechwyconymi wyrażeniami, które są wykonywane podczas działania funkcji.

Początkowo pomyślałem, że to błąd w projekcie języka, ponieważ każdy inny język programowania, jakiego używałem, zawiera instrukcję `return` zwracającą wartość¹, zastępuje instrukcję `return` nazwą zmiennej² albo domyślnie zwraca wartość ostatniego wyrażenia i opcjonalnie pozwala na użycie instrukcji `return` w celu zwrócenia wartości³. Żaden inny język nie tworzy listy (lub tablicy) obliczonych wyrażień, do której zwrócone wyrażenie jest po prostu dodawane.

Jednak po chwili namysłu doszedłem do wniosku, że taki sposób działania instrukcji `return` może być konieczny, jeśli weźmie się pod uwagę lubiane przeze mnie funkcje-filtry i to, że język Windows PowerShell jest interpretowany, na przykład:

```
function mojafunkcja { 1; 2; 3; invoke-expression $a; }
```

Ten problem z instrukcją `return` będzie lepiej widoczny, gdy przyjrzymy się poniższej funkcji `mojafunkcja2`:

```
function mojafunkcja2 { 1; sleep 1; 2; sleep 1; 3; sleep 1; return
4; }
foo2
```

Po uruchomieniu tej funkcji wyniki będą natychmiast dynamicznie pokazywać się w oknie konsoli: najpierw zostanie wydrukowana liczba 1, potem po sekundzie 2 itd. aż do 4. Następnie funkcja kończy działanie.

Jedynym sposobem na uniknięcie „zwrotu” trzech pierwszych elementów (1, 2 i 3) jest rozpoznanie przez konsolę Windows PowerShell słowa `return` podczas analizowania (przed wykonaniem) funkcji i zapobieżenie zwróceniu tych elementów, dopóki nie pojawi się instrukcja `return` lub nie nastąpi koniec procedury. Wówczas konsola Windows PowerShell odpowiednio je zamieni lub zwróci.

W takim przypadku może dojść do problemów z nieprzewidywalnym przepływem sterowania i nieoczekiwanymi wynikami. W czasie wykonywania, dopóki interpreter „nie wie”, czy instrukcja `return` zostanie wykonana, Windows PowerShell musi zapisywać te

¹ W języku C: `float average(float a, float b) { float c = a * b; return c / 2.0; }`

² W języku FORTRAN 77: `FUNCTION AVERAGE(A,B); C = A*B; AVERAGE=C / 2.0; END`

³ W języku Common Lisp: `(defun average (a b) (setf c (* a b)) (/ c 2.0))`

nieprzechwycone wyniki (ale ich nie okazywać). Gdy wykonywanie dojdzie do punktu, w którym wiadomo, że instrukcja `return` zostanie lub nie zostanie wykonana, można odrzucić wyniki i zatrzymać ich zapisywanie lub okazać wszystkie zapisane wyniki i kontynuować okazywanie nowych, nieprzechwyconych wyników. Jest to zaskakujące, ale większość kodu działa zgodnie z oczekiwaniami.

A co się stanie, jeśli dodamy polecenie `cmdlet Invoke-Expression`, które może (ale nie musi) usiłować w ramach rozwinięcia dołączyć wykonanie instrukcji `return`? Teraz sytuacja staje się nieco trudniejsza! Ogólnie rzecz biorąc, Windows PowerShell „nie wie”, co zostanie wykonane, dopóki nie wykona danej instrukcji. Wyrażenie może (ale nie musi) zawierać instrukcję `return`, ale o tym konsola dowie się dopiero przed samym wykonaniem polecenia `Invoke-Expression`. Dlatego też sama obecność tego polecenia lub innego o podobnym zastosowaniu może powodować nieprzewidywalne przepływy sterowania, w których może być wykonywana instrukcja `return`.

Ciche zapisywanie nieprzechwyconych wyników przez konsolę Windows PowerShell zamiast ich okazywania może się wydawać kuszące, ponieważ spełnia tradycyjne oczekiwania użytkowników języków programowania. Ale czy na pewno to taki dobry pomysł? Pomyśl, co może się stać, gdy kod, który jest wykonywany w efekcie dostarczenia nieprzechwyconych wyników z wiersza poleceń, wpływa na ścieżkę kodu lub zmienne prowadzące do określenia, czy instrukcja `return` zostanie wykonana. Na przykład:

```
function bletch { 1; sleep 1; 2; sleep 1; 3; sleep 1; invoke expression $a; }
$a = "5;"
bletach | %{ if ($? -eq 3) { $a = "return 4;" }; $_; }
```

W tym raczej wydumany przykładzie wywołujący funkcję `bletach` stwierdza dopiero po zobaczeniu trzeciego obiektu wyprodukowanego przez funkcję `bletach`, że na końcu zostanie wykonane wyrażenie `return 4`, a nie `return 5`.

Jako że funkcja `bletach` nie może w sposób niezawodny przewidzieć, czy wywołujący (konsumenci) korzystają z nieprzechwyconych wyników, czy z obiektów strumieniowanych na bieżąco podczas ich tworzenia, nie może po prostu przytrzymać (lub zachować) tych nieprzechwyconych obiektów, aby zmienić stan systemu (nie tylko interpretera Windows PowerShell, ale i plików, ustawień rejestru itd.). Funkcja `bletach` musi wytwarzać/strumieniować nieprzechwycone wyniki do konsumenta natychmiast, zgodnie z oczekiwaniami.

W związku z tym konsola Windows PowerShell nie może też dostarczyć instrukcji o znaczeniu „wyczyść wszystko i dokonaj zwrotu”. Nieprzechwycone wyniki nie są zapisywane, tylko są okazywane (lub przekazywane do wywołującego będącego następnym obiektem w potoku). A zatem zanim konsola Windows PowerShell zorientuje się, że w kodzie jest instrukcja `return`, poprzednie wartości już wyjdą. Zapisywanie nieprzechwyconych wyników do czasu, aż konsola Windows PowerShell dowie się, czy zostaną okazane, czy zrzuczone, niweluje jedną z największych zalet tej konsoli — natychmiastowość pojawiania się wyników. Ponadto zapisywanie nieprzechwyconych wyników uniemożliwia poprawne działanie innego kodu, który potrzebuje ich na bieżąco.

Z tego wynika, że czasami dobra architektura powoduje powstawanie zaskakujących wyników. Windows PowerShell jawi mi się jak mieszanka wyrazistości języka Lisp (języki podobne do języka Lisp z reguły obliczają wartość wyrażenia `return` na końcu funkcji) i błyskawiczności

właściwej powłokom (drukowanie/okazywanie wyników w czasie ich otrzymywania), czego skutkiem ubocznym jest niespodziewane i nieintuicyjne działanie instrukcji `return`.

Jednym z rozwiązań jest rzutowanie każdej nieprzechwyconej wartości (czyli nieprzeznaczonej do zwrotu do wywołującego) na typ `[void]`. To znaczy, że twórca funkcji musi po prostu pamiętać o tym, że funkcja jest w istocie producentem obiektów, a nie tylko maszyną do obliczania jakiegoś wyrażenia, które w końcu zwraca pojedynczą jednorodną wartość.

A jeśli funkcja zawiera dużo kodu, który trudno przerobić? Albo co zrobić, jeśli twórca funkcji chce, aby zwracana była tylko ostateczna wartość, gdy nadal są jakieś nieprzechwycone wyniki? Poniższa funkcja ilustruje możliwe rozwiązanie.

(Nie jestem pewien, czy jest to całkiem poprawne ani czy jest wystarczająco zwięzłe, ale działa przynajmniej w podstawowych przypadkach).

```
function return-last { begin { $rtn = @{}; } process { $rtn = $_; } end { $rtn; } }
```

Kod funkcji można ująć w klamrę poprzedzoną znakiem `&`, a następnie przekazać go do funkcji `Return-Last`, jak pokazano poniżej:

```
PS C:\> function foo2-last { &{ 1; sleep 1; 2; sleep 1; 3; sleep 1; 4; } | return-last }
PS C:\> foo2-last
4
PS C:\>
```

Technika ta powoduje odrzucenie wszystkich wytworzonych wewnątrz funkcji obiektów oprócz ostatniego. Dzięki temu funkcja ta zwraca (okazuje) ostatni wytworzony przez siebie obiekt.

A jak zauważył Louis Clausen, również starszy programista testów, zamiast funkcji `Return-Last` wystarczy prosta referencyjna otoka tablicowa:

```
PS C:\Users\jcburley> function foo2-last-quick { @(&{ 1; sleep 1; 2; sleep 1; 3; sleep 1; 4; })[-1] }
PS C:\Users\jcburley> foo2-last-quick
4
PS C:\Users\jcburley>
```

Oczywiście wywołujący może otoczyć funkcję prostszą otoką, jak pokazano poniżej:

```
PS C:\Users\jcburley> @(foo2)[-1]
4
PS C:\Users\jcburley>.
```

Podstawowe wiadomości o filtrach

Filtr to specjalny rodzaj funkcji działającej na każdym obiekcie w potoku i często używanej do redukcji liczby obiektów przekazywanych przez potok. Z reguły w filtrach nie stosuje się parametrów `-Begin` i `-End`, więc często traktuje się je jak funkcje zawierające tylko blok przetwarzający. Ale z drugiej strony, wiele funkcji nie ma tych parametrów, natomiast można znaleźć filtry, które je mają. Najważniejsza różnica między funkcją a filtrem jest subtelniejsza. Jeśli w potoku zostanie użyta funkcja, to zatrzyma ona przetwarzanie tego potoku do momentu, aż jego pierwszy element dojdzie do końca. Później funkcja ta przyjmie dane wejściowe z pierwszego elementu w potoku i rozpocznie ich przetwarzanie. Po zakończeniu przetwarzania przekaże wyniki do następnego elementu w bloku skryptu.

Funkcja jest wykonywana raz dla danych potokowych. Natomiast filtr jest wykonywany raz dla każdego elementu danych przekazywanego przez potok. Krótko mówiąc, filtr w potoku strumieniuje dane, a funkcja nie, co może mieć wielki wpływ na wydajność. W ramach przykładu przeanalizujemy funkcję i filtr wykonujące to samo zadanie.

W skrypcie *MeasureAddOneFilter.ps1* za pomocą instrukcji `1..50000` tworzona jest tablica 50 000 elementów. (W Windows PowerShell 1.0 jest to maksymalny rozmiar tablicy tworzonej w ten sposób. W Windows PowerShell 2.0 podniesiono pułap do [Int32] 2146483647. Możliwość użycia tak dużej tablicy zależy od pamięci). Poniżej znajduje się opisywany fragment kodu:

```
PS C:\> 1..[Int32]::MaxValue
The '..' operator failed: Zgłoszono wyjątek typu 'System.OutOfMemoryException'..
At line:1 char:4
+ 1.. <<<< [Int32]::MaxValue
    + CategoryInfo          : InvalidOperation: (:) [], RuntimeException
    + FullyQualifiedErrorId : OperatorFailed
```

Następnie tablica zostaje przekazana do filtra `AddOne`. Filtr ten drukuje napis `filtr dodający jeden` i zwraca liczbę 1 do bieżącej liczby w potoku. Następnie zostaje wyświetlona informacja, ile czasu to zajęło. W moim komputerze wykonywanie skryptu *MeasureAddOneFilter.ps1* trwało 2,6 sekundy.

MeasureAddOneFilter.ps1

```
Filter AddOne
{
    "filtr dodający jeden"
    $_ + 1
}

Measure-Command { 1..50000 | addOne }
```

Teraz przyjrzymy się wersji funkcyjnej. Podobnie jak w skrypcie *MeasureAddOneFilter.ps1*, również w tym skrypcie utworzymy tablicę 50 000 elementów i prześlemy wynik do funkcji `AddOne`. Zostanie wyświetlony napis `funkcja dodająca jeden`. Przy przekazywaniu danych do funkcji zostanie utworzona zmienna automatyczna o nazwie `$input`. Jest to wyliczenie, a nie zwykła tablica. Zawiera ono metodę `MoveNext`, za pomocą której można przejść do następnego elementu w kolekcji. Jako że `$input` nie jest zwykłą tablicą, nie można używać normalnych indeksów w rodzaju `$input[0]`. Aby pobrać element, należy użyć własności `$input.current`. W moim komputerze wykonywanie poniższego skryptu *MeasureAddOneFunction.ps1* zajęło 4,3 sekundy, a więc prawie dwa razy dłużej niż wykonywanie filtru.

MeasureAddOneFunction.ps1

```
Function AddOne
{
    "funkcja dodająca jeden"
    While ($input.MoveNext())
    {
        $input.current + 1
    }
}

Measure-Command { 1..50000 | addOne }
```


Dlaczego filtr jest tak dużo szybszy od funkcji? Filtr jest wykonywany raz dla każdego elementu w potoku, jak pokazano poniżej:

```
filtr dodający jeden
2
filtr dodający jeden
3
filtr dodający jeden
4
filtr dodający jeden
5
filtr dodający jeden
6
```

Poniżej znajduje się kod źródłowy filtra *DemoAddOneFilter.ps1*:

DemoAddOneFilter.ps1

```
Filter AddOne
{
    "filtr dodający jeden"
    $_ + 1
}

1..5 | addOne
```

Funkcja *AddOne* działa do końca raz dla wszystkich elementów w potoku. To powoduje zatrzymanie przetwarzania w środku potoku, aż zostaną utworzone wszystkie elementy tablicy. Później wszystkie dane są przekazywane do funkcji jednocześnie poprzez zmienną *\$input*. W metodzie tej nie jest wykorzystywana strumieniowa cecha potoku, która w wielu przypadkach pozwala bardziej efektywnie wykorzystać pamięć.

```
funkcja dodająca jeden
2
3
4
5
6
```

Poniżej znajduje się kod źródłowy skryptu *DemoAddOneFunction.ps1*:

DemoAddOneFunction.ps1

```
Function AddOne
{
    "funkcja dodająca jeden"
    While ($input.MoveNext())
    {
        $input.current + 1
    }
}

1..5 | addOne
```

Aby zlikwidować tę różnicę w wydajności, można napisać funkcję tak, aby działała podobnie do filtra. W tym celu należy jawnie wywołać blok przetwarzający. Używając bloku przetwarzającego, ma się dostęp nie tylko do automatycznej zmiennej *\$input*, ale również do *\$_*. Zmieniony skrypt znajduje się w pliku *DemoAddOneR2Function.ps1*.

```
funkcja dodająca jeden, wersja 2
2
funkcja dodająca jeden, wersja 2
3
funkcja dodająca jeden, wersja 2
4
funkcja dodająca jeden, wersja 2
5
funkcja dodająca jeden, wersja 2
6
```

Poniżej znajduje się kod źródłowy skryptu *DemoAddOneR2Function.ps1*:

DemoAddOneR2Function.ps1

```
Function AddOneR2
{
    Process {
        "funkcja dodająca jeden, wersja 2"
        $_ + 1
    }
} #end AddOneR2

1..5 | addOneR2
```

Jaki wpływ na wydajność wywiera jawne użycie bloku przetwarzającego? W moim komputerze nowa wersja funkcji działa 2,6 sekundy, a więc tyle samo co filtr. Poniżej znajduje się kod źródłowy skryptu *MeasureAddOneR2Function.ps1*:

MeasureAddOneR2Function.ps1

```
Function AddOneR2
{
    Process {
        "funkcja dodająca jeden, wersja 2"
        $_ + 1
    }
} #end AddOneR2

Measure-Command {1..50000 | addOneR2 }
```

Innym powodem do używania filtrów jest to, że wyróżniają się wizualnie, dzięki czemu zwiększają czytelność skryptu. Poniżej znajduje się typowy wzór, według którego pisze się filtry:

```
Filter NazwaFiltru
{
    # kod źródłowy
}
```

W skrypcie *FilterHasMessage.ps1* znajduje się filtr `HasMessage`, którego definicja zaczyna się od słowa kluczowego `Filter` i nazwy `HasMessage`. W bloku skryptu (w klamrze) za pośrednictwem zmiennej automatycznej `$_` wykorzystywany jest potok. Zmienna ta jest wysyłana do polecenia `Where-Object`, które wykonuje filtr. W skrypcie wywołującym wyniki filtru `HasMessage` zostają przekazane do polecenia `Measure-Object`, które informuje użytkownika, ile zdarzeń w dzienniku aplikacji ma dołączoną wiadomość. Poniżej znajduje się kod źródłowy skryptu *FilterHasMessage.ps1*.

FilterHasMessage.ps1

```
Filter HasMessage
{
    $_ |
    Where-Object { $_.message }
} #end HasMessage

Get-WinEvent -LogName Application | HasMessage | Measure-Object
```

To, że filtr ma niejawną blokadę przetwarzającą, nie znaczy, że nie można jawnie użyć bloków *Begin*, *Process* i *End*. W skrypcie *FilterToday.ps1* znajduje się definicja filtra o nazwie *IsToday*. Aby filtr ten był samodzielną jednostką bez jakichkolwiek zewnętrznych zależności, takich jak przekazywanie do niego obiektu daty i czasu, musi sam sprawdzać aktualną datę i godzinę. Ale jeśli polecenie *Get-Date* zostanie wykonane w bloku przetwarzania, filtr wprawdzie będzie działał, lecz polecenie to zostanie wykonane dla każdego obiektu znajdującego się w folderze wejściowym. Jeśli w folderze będzie się znajdować 25 elementów, polecenie *Get-Date* zostanie wykonane 25 razy. Jeśli jakąś procedurę trzeba wykonać tylko raz w bloku przetwarzania filtra, można ją umieścić w sekcji *Begin*, która jest wykonywana tylko raz. Blok przetwarzania jest wywoływany raz dla każdego elementu w potoku. Jeśli trzeba wykonać przetwarzanie końcowe (np. wydrukować napis informujący, ile plików zostało dziś znalezionych), odpowiednie procedury należy umieścić w bloku *End*. Poniżej znajduje się kod źródłowy skryptu *FilterToday.ps1*.

FilterToday.ps1

```
Filter IsToday
{
    Begin {$dte = (Get-Date).Date}
    Process { $_ |
        Where-Object { $_.LastWriteTime -ge $dte }
    }
}

Get-ChildItem -Path C:\fso | IsToday
```

Zapiski praktyka

dr Mark Tabdilio, specjalista od wyszukiwania danych
MVP Microsoft SQL Server

Ogólnie rzecz biorąc, odkryłem, że bardzo pomaga mi użycie najpierw interaktywnego wiersza poleceń Windows PowerShell, a następnie stworzenie potencjalnych przypadków do ewentualnego dalszego oskryptowania. Wolę mieć mniej niż więcej programów, zgodnie z zasadą, że prostsze rozwiązania są najskuteczniejsze. Niniejszy przypadek ilustruje przykład wprowadzenia w życie tej zasady na podstawie trójfazowego procesu kodowania przy użyciu analitycznego języka SAS.

W historii, którą chcę opowiedzieć, badałem trzy niezależne grupy konsultantów SAS, którzy w sumie utworzyli kilka tysięcy plików programu SAS (są to pliki tekstowe z rozszerzeniem .SAS). Klient miał uzasadnione obawy co do nadmiernej ilości kodu produkcyjnego, więc zaproponowałem mu rozwiązanie systemowe. W fazie pierwszej za pomocą wiersza poleceń Windows PowerShell (zaczynając od polecenia `get-childitem`) sporządziłem audyt całej struktury i zapisałem wyniki w tabelach Excela. Na podstawie tych wyników zorganizowaliśmy korpus kodu od wysokiego poziomu i wyznaczyliśmy dalsze cele.

W fazie drugiej napisałem serię poleceń Windows PowerShell (np. `select-string`), aby znaleźć podobne programy SAS. Mogłem zatrzymać wyjście lub po prostu odczytać kod SAS. Okazało się, że wiele z tych tysięcy plików było tylko różnymi wersjami tego samego kodu, luźniejszą wersją kontroli źródła („luźniejszą”, ponieważ brakowało zorganizowanych metadanych). Nie przeczytałem zawartości tysięcy plików, tylko skoncentrowałem swoją energię.

Faza trzecia polegała na uporządkowaniu dużych grup kodu w archiwach i zidentyfikowaniu kilkudziesięciu programów SAS, których logika mogła być potrzebna w aktualnych pracach. W fazie tej pracowałem głównie w SAS, aby poprawić czytelność, skonsolidować podobne funkcje oraz poprawić wydajność. W fazie tej uzasadniłem, że warto używać czegoś więcej niż tylko jednowierszowych poleceń Windows PowerShell. Stosowałem takie techniki jak zmiana dat plików oraz wyszukiwanie i zamienianie tekstu w plikach. Użyłem nawet języka SAS do utworzenia paru skryptów Windows PowerShell (wysyłanych do plików tekstowych z SAS). Użytkownicy SAS mogą generować pliki skryptów Windows PowerShell za pomocą języka SAS Macro i treści ze zbiorów danych lub katalogów SAS. Oczywiście większość pracy w tej trzeciej fazie polegała na ręcznym edytowaniu plików, ale konsola Windows PowerShell pomogła mi zautomatyzować np. standaryzację referencji `libname SAS` i aktualizowanie łańcuchów połączeń z serwerem SQL Server.

W tych trzech fazach pracy wypracowałem kod Windows PowerShell i techniki kodowania, których można użyć w przyszłych projektach. Proste polecenia Windows PowerShell wystarczyły do okiełznania metadanych znajdujących się w tysiącach plików źródłowych zapisanych w setkach katalogów. Bardziej skomplikowane skrypty Windows PowerShell zastosowano wybiórczo do finalnej grupy kilkudziesięciu plików. Z tego wynika następujący wniosek: używaj prostszych poleceń Windows PowerShell w skomplikowanych zewnętrznych środowiskach i bardziej skomplikowanych narzędzi PowerShell w prostszych środowiskach podręcznych.

Dodatkowe źródła informacji

- Centrum skryptów portalu TechNet pod adresem <http://www.microsoft.com/technet/scriptcenter> zawiera wiele przykładów użycia funkcji w Windows PowerShell.
- Na blogu Fix It firmy Microsoft, pod adresem <http://blogs.technet.com/b/fixit4me/>, znajduje się wiele przykładów stron samopomocy.

- W witrynie internetowej Brandona Shella, pod adresem <http://bsonposh.com/>, można znaleźć wiele wskazówek i porad na temat używania konsoli Windows PowerShell oraz dobry opis paru pułapek.
- Wszystkie skrypty opisane w tym rozdziale są dostępne do pobrania w repozytorium centrum skryptów portalu TechNet pod adresem <http://gallery.technet.microsoft.com/scriptcenter/PowerShell-40-Best-d9e16039>.

Rozdział 9

Projektowanie pomocy do skryptów

- Dodawanie dokumentacji pomocniczej do skryptu przy użyciu komentarzy jednowierszowych
- Używanie wielowierszowych znaczników komentarzowych w Windows PowerShell 4.0
- Używanie pomocy komentarzowej
- 13 zasad pisania efektywnych komentarzy
- Dodatkowe źródła informacji

Mimo że dobrze napisany kod jest zrozumiały i łatwy w obsłudze, może stać się jeszcze lepszy dzięki dobrze napisanej dokumentacji. Dobra dokumentacja powinna zawierać listę założeń, jakie przyjęto podczas pisania skryptu, np. istnienie określonego folderu albo konieczność posiadania uprawnień administratora. Ponadto powinna zawierać opis zależności, np. czy potrzebna jest konkretna wersja platformy .NET. Dobra dokumentacja świadczy o profesjonalizmie autora skryptu, ponieważ nie tylko zawiera niezbędne informacje na temat sposobu użycia skryptu, ale również objaśnia sposoby jego modyfikacji, a nawet metody użycia wybranych funkcji w innych skryptach.

Wszystkie skrypty produkcyjne powinny zawierać jakiś rodzaj pomocy. Ale jak najlepiej ją dostarczyć? W rozdziale tym poznasz sprawdzone metody dostarczania pomocy do własnych skryptów Windows PowerShell.

Pisząc dokumentację pomocniczą do skryptu, możesz skorzystać z trzech narzędzi. Pierwsze to zwykły komentarz jednowierszowy, którego można używać od Windows PowerShell 1.0. Drugie narzędzie to wielowierszowy komentarz, jaki wprowadzono w Windows PowerShell 2.0. Trzecie narzędzie to pomoc oparta na komentarzach, którą również wprowadzono w Windows PowerShell 2.0. Najpierw opisuję te trzy narzędzia, a potem przedstawiam 13 zasad pisania efektywnych komentarzy.

Dodawanie dokumentacji w komentarzach jednowierszowych

Komentarze jednowierszowe są doskonałym sposobem na szybkie dodanie dokumentacji do skryptu. Ich zaletą jest prostota i łatwość obsługi. W komentarzach tego typu najczęściej wpisuje się objaśnienia dotyczące niejasnych konstrukcji oraz dodaje notatki na przyszłość. Można ich

używać wyłącznie we własnym środowisku skryptowym. Z tego podrozdziału dowiesz się, jak dodać do skryptu dokumentację w postaci komentarzy jednowierszowych.

W skrypcie *CreateFileNameFromDate.ps1* sekcja nagłówkowa zawiera komentarze objaśniające sposób działania tego skryptu oraz opisujące jego ograniczenia. Poniżej znajduje się zawartość tego skryptu.

CreateFileNameFromDate.ps1

```
# -----
# SKRYPT: CreateFileNameFromDate.ps1
# AUTOR: Ed Wilson, Microsoft
# DATA: 12.15.2008
# SŁOWA KLUCZOWE: platforma .NET, io.path, get-date, plik, new-item, standardowe łańcuchy
#   formatu daty i godziny, wyrażenia regularne, referencje, przekazywanie przez referencję
# UWAGI: Skrypt ten tworzy pusty plik tekstowy na podstawie znacznika czasu. Używa łańcucha
#   formatowania
#   w celu określenia sortowalnej daty. Za pomocą metody GetInvalidFileNameChars pobiera wszystkie
#   znaki, które nie mogą występować w nazwach plików. Przyjęto założenie, że istnieje folder o
#   nazwie fso na dysku C:\. Jeśli nie ma tego folderu, skrypt nie zadziała.
#
Function GetFileName([ref]$fileName)
{
    $invalidChars = [io.path]::GetInvalidFileNamechars()
    $date = Get-Date -format s
    $fileName.value = ($date.ToString() -replace "[$invalidChars]","-") + ".txt"
}

$fileName = $null
GetFileName([ref]$fileName)
new-item -path c:\fso -name $filename -itemtype file
```

Ogólnie rzecz biorąc, do funkcji zawsze powinno się dołączać informację na temat sposobu ich użycia. Należy opisać wszystkie parametry i zależności. Ponadto twórca skryptu powinien dodać informacje potrzebne osobom, które będą zajmowały się obsługą serwisową skryptu. Należy zawsze zakładać, że programista serwisujący skrypt nie będzie wiedział, jak ten skrypt działa, więc trzeba mu to dokładnie wyjaśnić. W skrypcie *BackUpFiles.ps1* dodano komentarze — zarówno w nagłówku, jak i do każdej funkcji. Zawierają one objaśnienia logiki i ograniczeń tych funkcji.

BackUpFiles.ps1

```
# -----
# NAZWA: BackUpFiles.ps1
# AUTOR: ed wilson, Microsoft
# DATA: 12.12.2008
#
# SŁOWA KLUCZOWE: system plików, get-childitem, where-object
#   używanie daty, wyrażenia regularne
#
# UWAGI: Skrypt ten tworzy kopię zapasową folderu. Kopiuje
#   pliki, które zostały zmienione w ciągu ostatnich 24 godzin.
#   Można zmienić częstotliwość kopiowania plików, miejsce ich
#   zapisu oraz źródło. Skrypt tworzy folder o nazwie utworzonej z
#   godziny, o której został uruchomiony. Jeśli folder docelowy nie
#   istnieje, zostanie utworzony. Nazwa folderu jest tworzona na podstawie
#   czasu uruchomienia skryptu i wygląda tak: C:\bu\12.12.2008.1.22.51.PM.
#   Odstęp czasu oznacza wiek w dniach plików do skopiowania.
```



```
#
# -----
Function New-BackupFolder($destinationFolder)
{
    # Pobiera ścieżkę do folderu docelowego i tworzy ścieżkę do
    # folderu potomnego na podstawie daty i godziny. Następnie wywołuje
    # funkcję New-Backup, przekazując ścieżkę źródłową, ścieżkę docelową i
    # odstęp czasu w dniach.
    $dte = get-date
    # Poniższe wyrażenie regularne usuwa białe znaki, średniki,
    # i ukośniki / z daty oraz zamienia je na kropki, tworząc w ten
    # sposób nazwę folderu na kopię zapasową.
    $dte = $dte.tostring() -replace "[:\s/]", "."
    $backUpPath = "$destinationFolder" + $dte
    $null = New-Item -path $backUpPath -itemType directory
    New-Backup $dataFolder $backUpPath $backUpInterval
} #end New-BackupFolder

Function New-Backup($dataFolder,$backUpPath,$backUpInterval)
{
    # Rekurencyjnie kopiuje wszystkie pliki znajdujące się w folderze danych i
    # odfiltrowuje wszystkie pliki, które zmieniły się w ciągu określonej liczby dni.
    # Zapisuje skopiowane pliki do folderu docelowego i tworzy go (wliczając ścieżkę nadrzędną),
    # jeśli nie istnieje. Jeśli dany folder już istnieje, zostanie nadpisany. Jest to jednak
    # mało prawdopodobne, chyba że skrypt zostanie uruchomiony dwa razy w jednej minucie.
    "Kopiowanie plików z folderu $dataFolder... Pliki powinny znajdować się w folderze $backUpPath."
    Get-Childitem -path $dataFolder -recurse |
    Where-Object { $_.LastWriteTime -ge (get-date).addDays(-$backUpInterval) } |
    Foreach-Object { copy-item -path $_.FullName -destination $backUpPath -force }
} #end New-Backup

# *** punkt początkowy skryptu ***

$backUpInterval = 1
$dataFolder = "C:\fso"
$destinationFolder = "C:\BU\"
New-BackupFolder $destinationFolder
```

Zapiski praktyka

Dopracowywanie pomocy do poleceń

Dean Tsaltas

Emerytowany specjalista od skryptów z firmy Microsoft

Pisanie pomocy do polecenia cmdlet pod wieloma względami nie różni się od pisania zwykłej dokumentacji pomocniczej. Jeśli ktoś chce naprawdę dobrze wykonać tę pracę, musi wczuć się w rolę użytkownika. Ale oczywiście łatwiej powiedzieć niż zrobić, zwłaszcza gdy jest się projektantem i implementatorem skryptu. Twórca polecenia może tylko zgadywać, do jakich dziwnych i mrocznych celów może zostać wykorzystany jego produkt. Niemniej jednak trzeba robić co w naszej mocy. Wypożycz sobie film „Karate Kid” i obejrzyj go, aby zdobyć inspirację. Dobrze wszystko przemyśl, zanim zasiądziesz do klawiatury. Gdy już napiszesz

tekst zawierający Twój przekaz, zadaj sobie pytanie: „Co jest niejasne w tym tekście? O co po jego lekturze mógłby spytać mnie użytkownik?”. Wyobraź sobie, że wyjaśniasz coś realnym użytkownikom, postaraj się przewidzieć ich pytania i odpowiedz na nie.

Przypuśćmy, że nasze polecenie tworzy jakiś rodzaj plików oraz jako parametr przyjmuje nazwę lub ścieżkę zawierającą nazwę. Zastanów się, jakie pytania użytkownik mógłby mieć w odniesieniu do tego parametru: jaką powinien mieć długość, czy wszystkie znaki są dozwolone, jak traktowane są cudzysłowy, czy utworzony plik będzie miał rozszerzenie, czy też należy je wpisać w wartości parametru? Nie zmuszaj użytkowników do szukania drogą eksperymentalną odpowiedzi na pytania, które łatwo przewidzieć i na które można łatwo odpowiedzieć. Bądź pomocny.

W następnej kolejności pamiętaj, że jeden przykład jest wart tysiąca słów. Pisząc przykłady, zawsze mierz wysoko. Najlepiej wywołać burzę mózgów, aby wyłonić kilka zadań, które najprawdopodobniej będą wykonywane przez użytkowników. Powinieneś dodać przynajmniej po jednym przykładzie wykonania każdego z tych zadań. Następnie napisz przykład z użyciem wszystkich parametrów polecenia. Nawet jeśli Twoje przykłady będą nudne, staraj się przynajmniej dostarczyć użytkownikom jakiś punkt zaczepienia. Jak dobrze wiesz, łatwiej jest poszperać przy działającym poleceniu, aby zmusić je do działania w oczekiwany sposób, niż napisać całkiem nowe polecenie.

Powinno się też przewidzieć, jak użytkownicy będą używać pomocy do polecenia. Będzie ona wyświetlana na ekranie z podziałem na strony. Jako że pierwszy ekran jest jak treść nad zgięciem w gazecie, na nim powinny znaleźć się wszystkie najważniejsze informacje. Jeśli polecenie wymaga zwiększonych uprawnień, to należy o tym poinformować użytkowników w pierwszej kolejności. Jeżeli jest powiązany jakiś dostawca, który może być przydatny, to również poinformuj użytkowników o tym jak najwcześniej.

Nie zaniedbuj też części zawierającej odnośniki do podobnych tematów. Bardzo łatwo tworzy się listę wszystkich poleceń zawierających w nazwie ten sam rzeczownik, zwłaszcza gdy człowiek się spieszy. Ale czy to są jedyne polecenia mające związek z tym, które opisujesz? Na przykład: czy istnieje inne polecenie, którego użytkownicy muszą użyć, aby wygenerować obiekt przyjmowany jako wartość parametru Twojego polecenia? Jeśli tak, to warto o nim wspomnieć w tej sekcji. Jeszcze raz powtórzę: wyobraź sobie, że rozmawiasz z prawdziwymi użytkownikami. Do jakich innych dodatkowych źródeł byś ich odesłał? Dodaj też odnośniki do dodatkowej pomocy, nie tylko do tych materiałów, których związek z Twoim poleceniem jest oczywisty ze względu na nazewnictwo.

I ostatnia rada. Pomoc do polecenia zaczynaj pisać jak najwcześniej i pokaż ją komuś przed przejściem do fazy testowej alfa, aby szybko otrzymać uwagi. Jedynym sposobem na stworzenie doskonałej pomocy do polecenia (czy też dowolnego innego rodzaju dokumentacji technicznej) jest poprawianie jej na podstawie otrzymywanych uwag od testerów. Jak najszybciej dodaj dużo prostych przykładów. Jeśli ktoś będzie testował polecenie bez dołączonej do niego pomocy, to masz małe szanse na zdobycie wiedzy, jakich informacji będą potrzebować użytkownicy, aby sprawnie się nim posługiwać. Jeśli natomiast dodasz trzy przykłady, to z pewnością dostaniesz wskazówki na temat tego, jak powinny wyglądać kolejne przykłady.

Morał

Dodawanie komentarza do zamknięcia klamry

Kiedys spędziłem całą podróż pociągiem z Ratyzbony do Hamburga (prawie pięć godzin jazdy) na rozwiązywaniu problemu ze skryptem, który odkryłem, gdy pociąg opuszczał stację w Ratyzbonie. Skrypt ten miał być użyty w książce *Windows 7 Resource Kit* (Microsoft Press 2009) i miałem wyznaczony termin, którego trzeba było dotrzymać. Problem wystąpił po zmodyfikowaniu skryptu, gdy zapomniałem zamknąć klamrę. Był szczególnie trudny do rozwiązania, ponieważ wskazywał w bardzo długim skrypcie wiersz, który nie miał żadnego związku z tą sprawą. Właśnie wtedy doceniłem, jak ważne jest dodawanie komentarzy do zamknięć klamer. Teraz prawie nigdy o tym nie zapominam.

Poniżej znajduje się zamknięcie klamry z komentarzem. Jeśli wszystkie komentarze będą miały taką samą strukturę (np. `#end`, bez odstępów), to łatwo będzie je znaleźć np. za pomocą specjalnego skryptu.

```
} #end Get-ieStartPage
```

Teraz mamy za zadanie utworzyć funkcję przypisującą nowe wartości do stron startowych Internet Explorera. W tym celu możemy wywołać funkcję `Set-ieStartPage`, jak pokazano poniżej:

```
Function Set-ieStartPage()
{
```

Musimy przypisać pewne wartości dużej liczbie zmiennych. Pierwsze cztery zmienne są takie same jak w poprzedniej funkcji. (Mogliśmy zdefiniować je jako zmienne skryptowe i w ogólnym rozrachunku zaoszczędzić cztery wiersze kodu, ale wówczas funkcje nie byłyby całkowicie samodzielne). Zmienna `$value` służy do przechowywania domyślnej strony głównej, a zmienna `$aryvalues` ma za zadanie przechowywać tablicę dodatkowych adresów URL stron głównych. Poniżej znajduje się opisywany fragment kodu:

```
$hkcu = 2147483649
$key = "Software\Microsoft\Internet Explorer\Main"
$property = "Start Page"
$property2 = "Secondary Start Pages"
$value = "http://www.microsoft.com/technet/scriptcenter/default.mspx"
$aryValues = "http://social.technet.microsoft.com/Forums/en/ITCG/threads/",
"http://www.microsoft.com/technet/scriptcenter/resources/qanda/all.mspx"
```

Po przypisaniu wartości do zmiennych można za pomocą akceleratora wpisywania [WMICLASS] utworzyć egzemplarz klasy `stdRegProv`. Poniższy wiersz kodu jest używany w funkcji `Get-ieStartPage`:

```
$wmi = [wmiclass] "\\$computer\root\default:stdRegProv"
```

Teraz za pomocą metody `SetStringValue` można ustawić wartość łańcucha. Metoda ta pobiera cztery wartości. Pierwsza to liczba reprezentująca gałąź rejestru, z którą chcemy się połączyć. Następna to łańcuch reprezentujący klucz rejestru. Trzecia wartość określa własność do zmodyfikowania, a czwarta jest łańcuchem reprezentującym nową wartość do przypisania:

```
$rtn = $wmi.SetStringValue($hkcu,$key,$property,$value)
```

Następnie za pomocą metody `SetMultiStringValue` można ustawić wartość wielołańcuchowego klucza rejestru. Czwartym parametrem tej metody jest tablica. Sygnatura tej metody jest podobna do sygnatury metody `SetStringValue`. Jedyna różnica dotyczy właśnie czwartego parametru, którego wartością w przypadku metody `SetMultiStringValue` musi być tablica łańcuchów, a nie pojedyncza wartość, jak pokazano poniżej:

```
$rtn2 = $wmi.SetMultiStringValue($hkcu,$key,$property2,$aryValues)
```

Teraz można wydrukować wartość własności `ReturnValue`. Zawiera ona kod błędu z wywołania metody. Zero oznacza, że metoda została wykonana prawidłowo (brak błędów). Wszystkie inne wartości sygnalizują błąd.

```
"Efekt ustawiania własności $property to $($rtn.returnValue)"
"Efekt ustawiania własności $property2 to $($rtn2.returnValue)"
} #end Set-ieStartPage
```

Teraz przechodzimy do punktu początkowego skryptu. Najpierw musimy pobrać wartości początkowe, a następnie zmienić je na nowe wartości, które chcemy skonfigurować. Jeśli chcesz ponownie wysłać zapytanie do rejestru, aby upewnić się, że wartości zostały zmienione, możesz jeszcze raz wywołać funkcję `Get-ieStartPage`, jak pokazano poniżej:

```
if($get) {Get-ieStartpage}
if($set){Set-ieStartPage}
```

Poniżej znajduje się kompletny kod źródłowy skryptu *GetSetieStartPage.ps1*.

GetSetieStartPage.ps1

```
Param([switch]$get,[switch]$set,$computer="localhost")
$Comment = @"
NAZWA: GetSetieStartPage.ps1
AUTOR: ed wilson, Microsoft
DATA: 1.5.2009
```

SŁOWA KLUCZOWE: stdregprov, ie, akcelerator wpisywania [wmiclass], Hey Scripting Guy
 UWAGI: Skrypt ten pobiera strony startowe przeglądarki IE za pomocą akceleratora wpisywania [wmiclass] i klasy stdregprovto oraz ustawia strony startowe tej przeglądarki. Od IE 7 można ustawiać wiele stron startowych.

```
"@ #end comment
```

```
Function Get-ieStartPage()
```

```
{
$Comment = @"
```

```
FUNKCJA: Get-ieStartPage
```

Funkcja ta pobiera aktualne ustawienia przeglądarki Internet Explorer 7 i nowszych. Wartość `$hkcu` jest ustawiona na stałą wartość z SDK reprezentującą gałąź rejestru `Hkey_Current_User`. Do odczytu danych z rejestru używane są dwie metody, ponieważ strona startowa jest pojedynczą wartością, a klucz drugiej strony startowej jest wielowartościowy.

```
"@ #end comment
```

```
$hkcu = 2147483649
$key = "Software\Microsoft\Internet Explorer\Main"
$property = "Start Page"
$property2 = "Secondary Start Pages"
$wmi = [wmiclass]"\\$computer\root\default:stdRegProv"
```

```

($wmi.GetStringValue($hkcu,$key,$property)).sValue
($wmi.GetMultiStringValue($hkcu,$key, $property2)).sValue
} #end Get-ieStartPage

Function Set-ieStartPage()
{
$Comment = @"
FUNKCJA: Set-ieStartPage
Funkcja ta służy do konfigurowania stron domowych w przeglądarce IE 7 i nowszych. Zmienne
$aryValues i $Value zawierają różne strony domowe. Należy wpisać pełny adres, np.
"http://www.ScriptingGuys.Com". Każdy adres powinien znajdować się w cudzysłowie.

"@ #end comment
$hkcu = 2147483649
$key = "Software\Microsoft\Internet Explorer\Main"
$property = "Start Page"
$property2 = "Secondary Start Pages"
$value = "http://www.microsoft.com/technet/scriptcenter/default.mspx"
$aryValues = "http://social.technet.microsoft.com/Forums/en/ITCG/threads/",
"http://www.microsoft.com/technet/scriptcenter/resources/qanda/all.mspx"
$wmi = [wmiclass]"\\$computer\root\default:stdRegProv"
$rtn = $wmi.SetStringValue($hkcu,$key,$property,$value)
$rtn2 = $wmi.SetMultiStringValue($hkcu,$key,$property2,$aryValues)
"Efekt ustawiania własności $property to $($rtn.returnvalue)"
"Efekt ustawiania własności $property2 to $($rtn2.returnvalue)"
} #end Set-ieStartPage
# *** punkt początkowy skryptu
if($get) {Get-ieStartpage}
if($set){Set-ieStartPage}

```

Praca z folderami tymczasowymi

Ścieżkę do folderu tymczasowego w komputerze lokalnym można sprawdzić na kilka sposobów, między innymi przy użyciu środowiskowego dysku PS. W przykładzie opisanym w tym podrozdziale została użyta statyczna metoda `GetTempPath` z klasy `.NET System.IO.Path`. Metoda ta zwraca ścieżkę do folderu tymczasowego, w którym zapisaliśmy nowy plik tekstowy. Ścieżkę do folderu tymczasowego zapisujemy w zmiennej `$outputPath`, jak pokazano poniżej:

```
$outputPath = [io.path]::GetTempPath()
```

Nazwę dla tworzonego pliku tekstowego zbudujemy na podstawie nazwy skryptu. W związku z tym musimy oddzielić nazwę skryptu od ścieżki jego przechowywania. Do tego celu idealna będzie funkcja `Split-Path`. Parametr `-leaf` nakazuje poleceniu zwrócenie nazwy skryptu. Jeśli potrzebna jest ścieżka do katalogu zawierającego skrypt, można użyć parametru `-parent`. Polecenie `Split-Path` umieszczamy w nawiasie, ponieważ chcemy, aby to działanie zostało wykonane na początku. Znak dolara przed nawiasem powoduje utworzenie podwyrażenia wykonującego kod i zwracającego nazwę skryptu. Tworzonemu plikowi tekstowemu można by było nadać rozszerzenie `.ps1`, ale byłoby to mylące, ponieważ jest to rozszerzenie typowe dla skryptów. Dlatego do zwróconej nazwy pliku dodamy rozszerzenie `.txt` i cały łańcuch umieścimy w cudzysłowie.

Do utworzenia nowej ścieżki do pliku wyjściowego można użyć polecenia `Join-Path`. Ścieżka ta będzie się składać ze ścieżki do folderu tymczasowego zapisanej w zmiennej `$outputPath` i nazwy

pliku utworzonej za pomocą polecenia `Split-Path`. Elementy te łączymy przy użyciu polecenia `Join-Path`. Ścieżkę można też utworzyć za pomocą poleceń do pracy na łańcuchach i konkatencji, ale polecenia `Join-Path` i `Split-Path` są w takich zastosowaniach bardziej niezawodne. Poniżej znajduje się opisywana część kodu:

```
Join-Path -path $outputPath -child "$(Split-Path $script -leaf).txt"
} #end Get-FileName
```

Należy też przemyśleć, co robić w przypadku, gdy wystąpią duplikaty plików. W takiej sytuacji można wyświetlić odpowiednią informację dla użytkownika, jak pokazano poniżej:

```
$Response = Read-Host -Prompt "Plik o nazwie $outputFile już istnieje. Czy chcesz go
usunąć <t / n>?"
if($Response -eq "t")
{ Remove-Item $outputFile | Out-Null }
ELSE { "Kończenie pracy." ; exit }
```

Można też zaimplementować algorytm obsługi nazw, który będzie wykonywał kopię zapasową zduplikowanego pliku, dodając rozszerzenie `.old`, jak pokazano poniżej:

```
if(Test-Path -path "$outputFile.old") { Remove-Item -Path "$outputFile.old" }
Rename-Item -path $outputFile -newname "$(Split-Path $outputFile -leaf).old"
```

Można też po prostu usunąć istniejący plik i najczęściej wybieram właśnie to rozwiązanie. Czynności, które należy wykonać, zostaną zaimplementowane w funkcji `Remove-OutputFile`. Definicja tej funkcji zaczyna się od słowa kluczowego `Function`, za którym znajdują się jej nazwa oraz parametr wejściowy w postaci zmiennej `$outputFile`, jak pokazano poniżej:

```
Function Remove-outputFile($outputFile)
{
```

W celu sprawdzenia, czy dany plik istnieje, można przekazać do polecenia `Test-Path` w parametrze `-path` łańcuch zapisany w zmiennej `$outputFile`. Polecenie to zwraca tylko wartości `true` i `false`. Gdy nie znajdzie pliku, zwraca `false`, co znaczy, że do sprawdzania, czy dany plik istnieje, można używać instrukcji `if`. Jeśli plik zostanie znaleziony, można wykonać działania zdefiniowane w bloku skryptu. Jeżeli plik nie zostanie znaleziony, blok skryptu nie jest wykonywany. Jak pokazano poniżej, pierwsze polecenie nie znajduje pliku, więc zostaje zwrócona wartość `false`. W drugim poleceniu blok skryptu nie zostaje wykonany, ponieważ nie udało się znaleźć pliku.

```
PS C:\> Test-Path c:\missingfile.txt
False
PS C:\> if(Test-Path c:\missingfile.txt){"znaleziony plik"}
PS C:\>
```

W funkcji `Remove-OutputFile` można za pomocą instrukcji `if` sprawdzić, czy istnieje plik wskazywany przez zmienną `$outputFile`. Jeśli tak, należy go usunąć za pomocą polecenia `Remove-Item`. Informacje zwracane po usunięciu pliku są przekazywane do polecenia `Out-Null`, dzięki czemu operacja pozostaje niewidoczna. Poniżej znajduje się opisywany fragment kodu:

```
if(Test-Path -path $outputFile) { Remove-Item $outputFile | Out-Null }
} #end Remove-outputFile
```

Po utworzeniu nazwy pliku wyjściowego i usunięciu ewentualnych poprzednich plików można pobrać komentarze ze skryptu. Do tego celu utworzymy funkcję o nazwie `Get-Comments`, której prześlemy zmienne `$Script` i `$OutputFile`, jak pokazano poniżej:

```
Function Get-Comments($Script,$OutputFile)
{
```

Morał

Nie szperaj w sekcji roboczej skryptu

Jeśli podczas pisania skryptu mam zbierać dane do przekazania do funkcji, zazwyczaj zapisuję je w zmiennej, która będzie używana pod tą samą nazwą zarówno wewnątrz, jak i na zewnątrz funkcji. Robię tak, ponieważ jest to zgodne z jedną z moich zasad tworzenia skryptów: „Nie szperaj w sekcji roboczej skryptu”. W funkcji `Get-OutputFile` „wykonywana jest praca”. Aby zmienić ją w przyszłości w innych skryptach, konieczne jest zmodyfikowanie literalnej wartości łańcuchowej, co stwarza ryzyko uszkodzenia kodu, ponieważ wiele metod ma skomplikowane konstruktory. Jeśli dodatkowo spróbujemy przekazać wartości do konstruktorów metod wymagających zastosowania sekwencji specjalnych zastępujących niektóre znaki, ryzyko popełnienia błędu staje się jeszcze większe.

Używanie wielowierszowych znaczników komentarzowych w Windows PowerShell 4.0

Znaczniki komentarzowe Windows PowerShell 4.0 umożliwiają tworzenie komentarzy zajmujących więcej niż jedną linię skryptu. Można je porównać do łańcuchów miejscowych i znaczników HTML, ponieważ podobnie jak one składają się zarówno z części otwierającej, jak i zamykającej. Tak naprawdę komentarze wielowierszowe są dostępne od Windows PowerShell 2.0, więc nie można powiedzieć, że to jakaś nowość.

Tworzenie komentarzy wielowierszowych za pomocą znaczników komentarzowych

Znacznik otwierający składa się z otwarcia nawiasu ostrego i krzyżyka (`<#`), a znacznik zamykający składa się z krzyżyka i zamknięcia nawiasu ostrego (`#>`). Poniżej pokazano wzorcową definicję komentarza wielowierszowego:

```
<# znacznik otwierający komentarza
pierwsza linijka komentarza
druga linijka komentarza
#> znacznik zamykający komentarza
```

Przykład zastosowania tego rodzaju komentarza pokazano w skrypcie *Demo-Multiline-Comment.ps1*.

Demo-MultilineComment.ps1

```
<#
  Get-Command
  Get-Help
#>
"Wszystko powyżej to komentarz wielowierszowy."
```

Po uruchomieniu skryptu *Demo-MultilineComment.ps1* polecenia znajdujące się w komentarzu nie zostaną wykonane. Wykonane zostanie tylko to, co znajduje się poza blokiem komentarza, a więc skrypt wydrukuje tylko następujący napis w konsoli:

Wszystko powyżej to komentarz wielowierszowy.

Znaczniki komentarzy wielowierszowych nie muszą być jedyną treścią w linijce, tzn. bezpośrednio za lub przed nimi może znajdować się tekst komentarza. Poniżej pokazany jest wzór takiego komentarza wielowierszowego:

```
<# Znacznik otwierający komentarza Pierwsza linijka komentarza
Druka linijka komentarza #> Znacznik zamykający komentarza
```

Przykład praktycznego zastosowania tej metody przedstawiono w pokazanym poniżej skrypcie *MultilineDemo2.ps1*.

MultilineDemo2.ps1

```
<# Get-Help
  Get-Command #>
"Wszystko powyżej to komentarz wielowierszowy."
```

UWAGA

Osobiście uważam, że dobrym zwyczajem jest pozostawianie znaczników komentarzy wielowierszowych w osobnych liniach. Dzięki temu kod jest znacznie bardziej czytelny oraz łatwiej znaleźć początek i koniec komentarza.

Tworzenie jednowierszowych komentarzy przy użyciu znaczników komentarzowych

Za pomocą składni komentarzy wielowierszowych można też tworzyć komentarze jednowierszowe. Poniżej znajduje się wzór tego rodzaju komentarza:

```
<# Znacznik otwierający komentarza Pierwsza linijka komentarza #> Znacznik zamykający komentarza
```

Przykład praktycznego zastosowania tej techniki znajduje się w przedstawionym poniżej skrypcie *MultilineDemo3.ps1*.

MultilineDemo3.ps1

```
<# To jest pojedynczy komentarz. #>
"Powyżej znajduje się pojedynczy komentarz."
```


Używając wielowierszowych komentarzy, należy pamiętać, że wszystko, co znajduje się za znacznikiem zamykającym, zostanie zinterpretowane przez Windows PowerShell. Z interpretacji wyłączone jest tylko to, co znajduje się między znacznikami otwierającym i zamykającym. Należy też podkreślić, że komentarze wielowierszowe różnią się od jednowierszowych, tworzonych przy użyciu samej kratki (#). Jest to także nowość dla użytkowników języka VBScript, którzy są przyzwyczajeni do oznaczania komentarzy jednowierszowych za pomocą apostrofu ('). Poniżej przedstawiono typowy błąd popełniany przy definiowaniu komentarzy:

```
<# -----
Ten kod jest błędny.
#> -----
```

Jeśli ktoś chce wyróżnić swoje komentarze w sposób podobny do pokazanego w tym przykładzie, to musi tylko przenieść znacznik zamykający komentarz na koniec linii, jak pokazano poniżej:

```
<# -----
Ten kod jest poprawny.
-----#>
```

UWAGA

Znak # nie musi być niczym oddzielony od następnego znaku. Choć osobiście lubię wstawiać za nim spację, aby zwiększyć czytelność kodu.

Oczywiście nadal można też tworzyć komentarze jednowierszowe za pomocą znaku #. Aby przy jego użyciu utworzyć komentarz wielowierszowy, należy po prostu wpisać kratkę na początku każdej linii komentarza. Zaletą tej techniki jest jej rozpoznawalność i spójność oraz zgodność z konsolą Windows PowerShell 1.0.

```
# pierwsza linijka komentarza
# druga linijka komentarza
# ostatnia linijka komentarza
```

Używanie pomocy komentarzowej

Pracę nad tworzeniem pomocy do funkcji można znacznie uprościć przez skorzystanie z techniki tworzenia pomocy opartej na komentarzach. Metoda ta polega na umieszczeniu znaczników pomocy w bloku komentarza podczas pisania skryptu. Ten rodzaj pomocy jest zintegrowany z poleceniem Get-Help, co znacznie ułatwia posługiwanie się naszymi funkcjami lub skryptami innym użytkownikom. Można nawet zdefiniować pomoc komentarzową dla całego skryptu Windows PowerShell i dodatkowo dla każdej znajdującej się w nim funkcji osobno. Do dobrych praktyk należy definiowanie pomocy komentarzowej dla wszystkich funkcji znajdujących się w modułach Windows PowerShell. Ponadto fakt posiadania znaczników pomocy przez funkcję podnosi ją do takiej samej rangi, jaką mają macierzyste polecenia cmdlet. Pod względem sposobu użycia funkcje takie w niczym nie różnią się od standardowych poleceń i dla użytkownika nie ma znaczenia, czy funkcja została dołączona za pomocą notacji kropkowej, załadowana z modułu, czy też jest poleceniem macierzystym. W tabeli 9.1 znajduje się wykaz znaczników pomocy wraz z opisem.

TABELA 9.1. Znaczniki pomocy funkcji i ich opis

| Znacznik | Opis |
|--------------|--|
| .Synopsis | Bardzo zwięzły opis funkcji. Często zaczyna się od czasownika i zawiera podstawowe informacje na temat tego, co funkcja robi. Nie wpisuje się nazwy funkcji ani nie opisuje się, jak ona działa. Zawartość tego znacznika występuje w polu SYNOPSIS wszystkich widoków pomocy. |
| .Description | Dwa lub trzy zdania zawierające opis wszystkich wykonywanych przez funkcję czynności. Początek z reguły brzmi <i>Funkcja ta....</i> Jeśli funkcja przyjmuje wiele obiektów lub danych wejściowych, w opisie należy stosować rzeczowniki w liczbie mnogiej. Zawartość tego znacznika występuje w polu DESCRIPTION wszystkich widoków pomocy. |
| .Parameter | Zwięzły, ale szczegółowy opis tego, co robi funkcja, gdy zostanie jej przekazany dany parametr, oraz jakie są dopuszczalne wartości tego parametru. Zawartość tego znacznika występuje w polu PARAMETERS tylko w pełnym i szczegółowym widoku pomocy. |
| .Example | Przykład zastosowania funkcji z wszystkimi parametrami. Pierwszy przykład powinien być najprostszy i powinien zawierać tylko parametry obowiązkowe. Ostatni przykład powinien być najbardziej skomplikowany i powinien uwzględniać także potokowe przekazanie danych, jeśli ma to sens w danym przypadku. Zawartość tego znacznika jest wyświetlana w polu EXAMPLES w widokach przykładowym, szczegółowym oraz pełnym. |
| .Inputs | Zawiera listę klas .NET, których obiekty funkcja przyjmuje na wejściu. Liczba klas na liście jest nieograniczona. Zawartość tego znacznika jest wyświetlana w polu INPUTS tylko w pełnym widoku pomocy. |
| .Outputs | Zawiera listę klas .NET, których obiekty funkcja zwraca na wyjściu. Liczba klas na liście jest nieograniczona. Zawartość tego znacznika jest wyświetlana w polu OUTPUT tylko w pełnym widoku pomocy. |
| .Notes | Miejsce na przekazanie informacji, które nie pasują do żadnej innej sekcji. Mogą to być uwagi dotyczące specjalnych wymagań funkcji, autora, tytułu, wersji itp. Zawartość tego znacznika jest wyświetlana w polu NOTES tylko w pełnym widoku pomocy. |
| .Link | Dostarcza odnośników do innych tematów pomocy i powiązanych tematycznie stron internetowych. Jako że odnośniki te są wyświetlane w wierszu poleceń, nie są prawdziwymi hiperłączami. Ich liczba jest nieograniczona. Zawartość tego znacznika jest wyświetlana w polu RELATED LINKS we wszystkich widokach pomocy. |
| .Component | Funkcjonalność lub technologia używana przez funkcję lub skrypt. Odnosi się także do powiązanej technologii lub funkcjonalności. |
| .Role | Rola użytkownika dla tematu pomocy. |

TABELA 9.1. Znaczniki pomocy funkcji i ich opis — ciąg dalszy

| Znacznik | Opis |
|--|--|
| <code>.Functionality</code> | Przeznaczenie skryptu lub funkcji. |
| <code>.ForwardHelp</code> ↳ <code>TargetName</code> | Przekierowuje temat pomocy określonego polecenia. Znacznik ten pozwala na utworzenie przekierowania do dowolnego tematu pomocy, np. dotyczącego funkcji, skryptu, polecenia cmdlet lub dostawcy. |
| <code>.ForwardHelp</code> ↳ <code>Category</code> | Określa kategorię elementu w <code>ForwardHelpTargetName</code> . Dopuszczalne wartości to: <code>Alias</code> , <code>Cmdlet</code> , <code>HelpFile</code> , <code>Function</code> , <code>Provider</code> , <code>General</code> , <code>FAQ</code> , <code>Glossary</code> , <code>ScriptCommand</code> , <code>ExternalScript</code> , <code>Filter</code> oraz <code>All</code> . Znacznik ten pozwala uniknąć konfliktów, gdy wystąpi kilka poleceń o tej samej nazwie. |
| <code>.RemoteHelpRun</code> ↳ <code>Space</code> | Określa sesję zawierającą temat pomocy. Przyjmuje zmienną zawierającą obiekt <code>PSSession</code> . Znacznik ten jest wykorzystywany przez polecenie <code>Export-PSSession</code> . |
| <code>.ExternalHelp</code> | Używany, gdy funkcja lub skrypt korzysta z plików pomocy w formacie XML. |

Nie trzeba definiować wszystkich znaczników pomocy, ale dobrym zwyczajem jest zdefiniowanie przynajmniej znaczników `.synopsis` i `.example`, które powinny zawierać najważniejsze informacje dla kogoś, kto chce się dowiedzieć, jak używać danej funkcji.

Przykład zastosowania znaczników pomocy znajduje się w skrypcie `GetWmiClassesFunction1.ps1`. Polecenie `Get-Help` wyświetli dla niego dokładnie takie same informacje jak dostarczone w skrypcie `GetWmiClassesFunction.ps1`. Różnica między tymi dwoma skryptami dotyczy tylko użycia znaczników pomocy. Dzięki zintegrowaniu kodu z poleceniem `Get-Help` nie trzeba już używać parametru `-help`, a to z kolei sprawia, że nie trzeba też już sprawdzać, czy istnieje zmienna `$help`. To z kolei upraszcza kod skryptu.

Zalety używania specjalnych znaczników pomocy

Stosowanie specjalnych znaczników pomocy ma kilka zalet, na przykład:

- Nazwa funkcji jest automatycznie wyświetlana we wszystkich widokach pomocy.
- Składnia funkcji jest automatycznie derywowana z parametrów i wyświetlana we wszystkich widokach pomocy.
- Automatycznie generowane są szczegółowe informacje o parametrach, gdy zostanie użyty parametr `-full` w poleceniu `Get-Help`.
- Automatycznie wyświetlane są typowe informacje o parametrach, gdy polecenie `Get-Help` zostanie użyte z parametrami `-detailed` i `-full`.

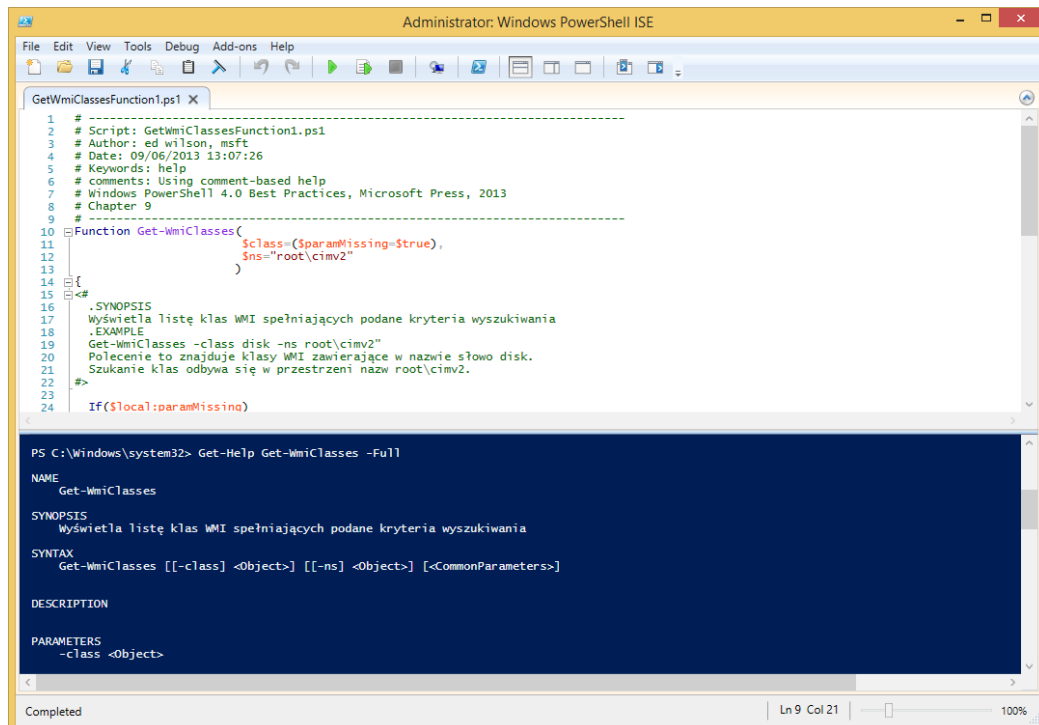
W skrypcie `GetWmiClassesFunction1.ps1` znajduje się funkcja `Get-WmiClasses`, której początek stanowi wielowierszowy blok komentarza zdefiniowany przez znacznik otwierający `<#` i zamykający `#>`. Wszystko, co znajduje się między tymi znacznikami, jest komentarzem. W komentarzu tym zostały zdefiniowane dwa znaczniki pomocy: `.synopsis` i `.example`. W tabeli 9.1 znajduje się opis większej liczby znaczników, ale w tym przypadku zostały zdefiniowane tylko dwa.

```

<#
.SYNOPSIS
Wyświetla listę klas WMI spełniających podane kryteria wyszukiwania
.EXAMPLE
Get-WmiClasses -class disk -ns root\cimv2"
Polecenie to znajduje klasy WMI zawierające w nazwie słowo disk.
Szukanie klas odbywa się w przestrzeni nazw root\cimv2.
#>

```

Po dołączeniu skryptu *GetWmiClassesFunction1.ps1* do konsoli Windows PowerShell za pomocą polecenia *Get-Help* będzie można wyświetlać informacje pomocnicze o funkcji *Get-WmiClasses*. Na rysunku 9.1 pokazano efekt wykonania tego polecenia z parametrem *-full*.



RYСУNEK 9.1. Pełna treść pomocy pobrana z funkcji *Get-WmiClasses*

Poniżej znajduje się kompletny kod źródłowy skryptu *GetWmiClassesFunction1.ps1*.

GetWmiClassesFunction1.ps1

```

Function Get-WmiClasses(
    $class=($paramMissing=$true),
    $ns="root\cimv2"
)
{
<#
.SYNOPSIS
Wyświetla listę klas WMI spełniających podane kryteria wyszukiwania
.EXAMPLE

```

```
Get-WmiClasses -class disk -ns root\cimv2"
```

Polecenie to znajduje klasy WMI zawierające w nazwie słowo disk.

Szukanie klas odbywa się w przestrzeni nazw root\cimv2.

```
#>
```

```
If($local:paramMissing)
{
    throw "Sposób użycia: getwmi2 -class <class type> -ns <wmi namespace>"
} #local:paramMissing
"Klasy w przestrzeni nazw $ns..."
Get-WmiObject -namespace $ns -list |
where-object {
    $_.name -match $class -and '
    $_.name -notlike 'cim*'
}
# mred function
} #end get-wmiclasses
```

Jeśli ktoś planuje dołączać funkcje do roboczego środowiska Windows PowerShell lub modułów przy użyciu składni kropkowej, to może dodać katalog zawierający skrypty do ścieżki. Można dodać katalog z funkcjami na stałe za pomocą narzędzi GUI albo po prostu dodawać ścieżkę przy każdym uruchamianiu Windows PowerShell, wprowadzając odpowiednią zmianę w profilu Windows PowerShell. Jeśli zdecydujesz się na dodanie katalogu z funkcjami za pomocą poleceń Windows PowerShell, możesz używać dysku środowiskowego Windows PowerShell w celu uzyskania dostępu do zmiennej systemowej path i wprowadzenia zmian. Poniższy kod najpierw sprawdza zmienną path, a następnie dołącza na jej końcu folder C:\fso. Każdy katalog dodawany do tej ścieżki wyszukiwania musi być oddzielony od pozostałych średnikiem, dlatego przy dodawaniu katalogu na początku jego ścieżki należy wpisać średnik. W celu dołączenia ścieżki do katalogu na końcu wartości zmiennej path można użyć operatora +=. Ostatnie polecenie sprawdza jeszcze zmienną path, abyśmy mieli pewność, że zmiana rzeczywiście została dokonana.

```
PS C:\> $env:path
C:\Windows\system32;C:\Windows;C:\Windows\System32\Wbem;C:\Windows\System32\Windows System
Resource Manager\bin;C:\Windows\idmu\common;C:\Windows\system32\WindowsPowerShell\v1.0\
PS C:\> $env:path += ";C:\fso"
PS C:\> $env:path
C:\Windows\system32;C:\Windows;C:\Windows\System32\Wbem;C:\Windows\System32\Windows System
Resource Manager\bin;C:\Windows\idmu\common;C:\Windows\system32\WindowsPowerShell\v1.0\;C:\fso
```

Zmiana ścieżki path wprowadzona za pomocą dysku środowiskowego konsoli Windows PowerShell jest tylko tymczasowa i znika po zamknięciu bieżącej sesji konsoli. Efekt wprowadzonej zmiany jest natychmiastowy, więc jest to dobry sposób na szybką zmianę środowiska Windows PowerShell bez trwałego modyfikowania ustawień systemowych.

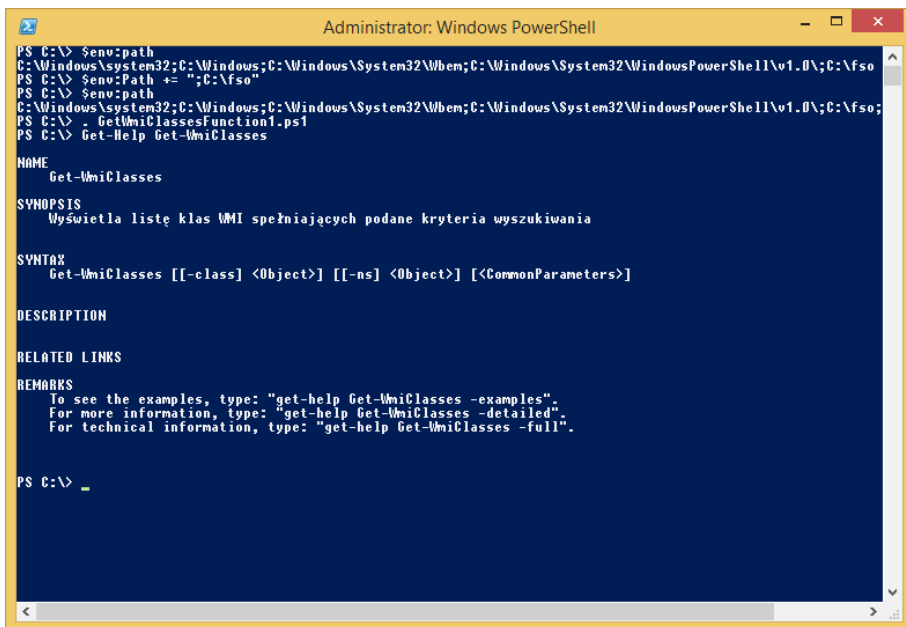
UWAGA

Dla mnie możliwość dostępu do skryptów z poziomu wiersza poleceń jest niezwykle przydatna i dlatego dodaję folder ze skryptami do zmiennej środowiskowej path poprzez profil. Dzięki temu zawsze mam bezpośredni dostęp do wszystkich swoich skryptów poprzez proste dołączanie za pomocą składni kropkowej. Więcej informacji na temat modyfikowania środowiska roboczego Windows PowerShell poprzez profil znajduje się w rozdziale 5.

Wielką zaletą modyfikowania zmiennej path przez dysk środowiskowy Windows PowerShell jest to, że zmiany zostają wprowadzone natychmiast i są od razu widoczne w bieżącej sesji Windows PowerShell. Oznacza to, że można dodać katalog do ścieżki, dołączyć za pomocą kropki skrypt zawierający funkcje i za pomocą polecenia `Get-Help` wyświetlić treść pomocy bez potrzeby zamykania ani otwierania jakichkolwiek okien Windows PowerShell. Gdy jakiś katalog zostanie dodany do ścieżki poszukiwań, to można dołączać za pomocą kropki skrypty z tego katalogu bez podawania pełnej ścieżki do niego. Poniższy kod ilustruje opisaną technikę modyfikacji zmiennej path, dołączenia za pomocą ścieżki katalogu ze skryptami i użycia polecenia `Get-Help`:

```
PS C:\> $env:Path += ";C:\fso"
PS C:\> . GetWmiClassesFunction1.ps1
PS C:\> Get-Help Get-WmiClasses
```

Na rysunku 9.2 pokazano efekt zastosowania tej techniki. Kolejno dodano katalog do zmiennej path, dołączono za pomocą kropki skrypt z tego katalogu oraz wyświetlono informacje na temat jednej z funkcji z tego skryptu za pomocą polecenia `Get-Help`.



RYСУNEK 9.2. Dzięki dodaniu katalogu do zmiennej path można łatwo dołączać funkcje do bieżącego środowiska Windows PowerShell

13 zasad pisania efektywnych komentarzy

Podczas pisania dokumentacji do skryptu należy szczególnie uważać, aby nie popełnić błędu. Jeśli opisy nie zgadzają się z kodem, to istnieje spore prawdopodobieństwo, że oba są błędne. Zawsze gdy zmieniasz coś w skrypcie, wprowadź też odpowiednie zmiany w komentarzach. Dzięki temu informacje zawarte w obu tych miejscach będą ze sobą spójne.

Aktualizuj dokumentację razem ze skryptem

Przy dodawaniu nowych parametrów do funkcji łatwo można zapomnieć o zmodyfikowaniu komentarzy dotyczących parametrów. Podobnie łatwo jest zignorować znajdujące się w nagłówku skryptu informacje odnoszące się do zależności lub założeń przyjętych w tym skrypcie. A skrypt i zawarte w nim komentarze należy traktować z taką samą uwagą. W nagłówku skryptu *FindDisabledUserAccounts.ps* znajdują się komentarze, które wyglądają tak, jakby dotyczyły tego skryptu, ale nie uwzględniono w nich informacji, że w skrypcie tym użyto akceleratora wpisywania [ADSIsearcher]. Tak naprawdę jest to modyfikacja skryptu służącego do tworzenia egzemplarza klasy .NET *DirectoryServices.DirectorySearcher*. Skrypt ten ktoś niedawno zmodyfikował, ale zapomniał o uwzględnieniu wprowadzonych zmian w komentarzach. To przeoczenie może wzbudzać wątpliwości co do tego, czy ten skrypt w ogóle jest przydatny. Poniżej znajduje się zawartość skryptu *FindDisabledUserAccounts.ps1*.

FindDisabledUserAccounts.ps1

```
# -----
# FindDisabledUserAccounts.ps1
# ed wilson, 28.3.2008
#
# Tworzy egzemplarz klasy .NET DirectoryServices.DirectorySearcher
# do przeszukiwania Active Directory.
# Tworzy filtr w składni LDAP, który jest stosowany do obiektu
# wyszukującego. Jeśli szukana jest tylko klasa użytkownika, to otrzymywane są
# również konta komputerowe, ponieważ są one derywowane z klasy użytkownika.
# W związku z tym wykonujemy zapytanie złożone, aby pobrać także osobę.
# Następnie za pomocą metody findall wyszukujemy i pobieramy wszystkich użytkowników.
# Później przy użyciu własności properties wybieramy item, aby otrzymać nazwę
# distinguishedname każdego użytkownika, a potem przy użyciu tej nazwy wykonujemy
# zapytanie i pobieramy atrybut UAC. Następnie w instrukcji warunkowej
# wykonujemy logiczne porównanie z wartością 2, aby dowiedzieć się, które
# konto jest wyłączone.
#
# -----
#Requires -Version 2.0

$filter = "(&(objectClass=user)(objectCategory=person))"
$users = ([adsisearcher]$filter).findall()

foreach($user in $users)
{
    "Sprawdzanie $($user.properties.item("distinguishedname"))"
    $user = [adsisearcher]"LDAP://$($user.properties.item("distinguishedname"))"

    $uac=$user.psbased.InvokeGet("useraccountcontrol")
    if($uac -band 0x2)
    { write-host -foregroundcolor red "Konto 't jest wyłączone." }
    ELSE
    { write-host -foregroundcolor green "Konto 't nie jest wyłączone." }
} #foreach
```

Dodawaj komentarze podczas pisania kodu

Gdy piszesz skrypt, komentarze do kodu dodawaj na bieżąco. Nie czekaj, aż skończysz pracę nad skryptem. Jeśli to zrobisz, jest duże ryzyko, że pominiesz wiele ważnych szczegółów, ponieważ będziesz już tak dobrze znał swój produkt, że nie przyjdzie Ci nawet do głowy, o co ktoś mógłby zapytać. Jeśli będziesz dodawać komentarze w trakcie pisania skryptu, to będziesz mógł się do nich odwoływać, aby mieć pewność, że trzymasz odpowiedni kurs. W ten sposób zapewnisz spójność nazewnictwa zmiennych i stylu kodowania. Problem ze spójnością został pokazany w skrypcie *CheckForPdfAndCreateMarker.ps1*. Patrząc na kod źródłowy, można dojść do wniosku, że skrypt ten sprawdza, czy są pliki PDF, co zresztą sugeruje też nazwa skryptu. Ale czemu skrypt ten proponuje usunięcie plików? Czym jest marker? Jedyna widoczna informacja jest taka, że napisałem ten skrypt w grudniu 2008 roku do artykułu na blogu „Hey, Scripting Guy!”. Na szczęście na blogu tym znajdują się objaśnienia skryptów, więc jakaś dokumentacja jest. Poniżej znajduje się kod źródłowy skryptu *CheckForPdfAndCreateMarker.ps1*.

CheckForPdfAndCreateMarker.ps1

```
# -----
# CheckForPdfAndCreateMarker.ps1
# ed wilson, msft, 12.12.2008
#
# Hey Scripting Guy! 29.12.2008
# -----
$path = "c:\fso"
$include = "*.pdf"
$name = "nopdf.txt"
if(!(Get-ChildItem -path $path -include $include -Recurse))
{
    "No pdf was found in $path. Creating $path\$name marker file."
    New-Item -path $path -name $name -itemtype file -force |
    out-null
} #end if not Get-Childitem
ELSE
{
    $response = Read-Host -prompt "Znaleziono pliki PDF. Czy chcesz je usunąć <t> /<n>?"
    if($response -eq "t")
    {
        "Pliki PDF zostaną usunięte."
        Get-ChildItem -path $path -include $include -recurse |
        Remove-Item
    } #end if response
ELSE
{
    "Pliki PDF nie zostaną usunięte."
} #end else reponse
} #end else not Get-Childitem
```

Pisz z myślą o użytkownikach z różnych krajów

Pisząc komentarze do skryptu, staraj się zwracać do potencjalnych użytkowników z wielu krajów, co oznacza, że najlepiej, aby komentarze te były po angielsku. Staraj się pisać jasno i bez stosowania wyszukanych słów, bo nie każdy użytkownik Twoich skryptów musi biegle posługiwać się

językiem angielskim. Ponadto komentarze po angielsku łatwiej jest zlokalizować za pomocą automatycznych narzędzi. Przy pisaniu dokumentacji przeznaczonej dla użytkowników z różnych krajów przede wszystkim należy pamiętać o stosowaniu prostej składni i standardowej terminologii. Unikaj slangu, akronimów i nadmiernie poufalego języka. Jeśli masz taką możliwość, poproś kogoś innego o przeczytanie napisanej przez Ciebie dokumentacji. W skrypcie *SearchForWordImages.ps1* komentarze objaśniają, co ten skrypt robi i jakie są jego ograniczenia, np. został on przetestowany tylko przy użyciu programu Microsoft Office Word 2007. Zdania są proste i pozbawione żargonu oraz idiomów. Poniżej znajduje się treść skryptu *SearchForWordImages.ps1* — dla wygody polskiego czytelnika dołączone zostało tłumaczenie angielskich komentarzy.

SearchForWordImages.ps1

```
# -----
# NAME: SearchForWordImages.ps1
# AUTHOR: ed wilson, Microsoft
# DATE: 11/4/2008
#
# NAZWA: SearchForWordImages.ps1
# AUTOR: ed wilson, Microsoft
# DATA: 04.11./2008
#
# KEYWORDS: Word.Application, automation, COM
# Get-Childitem -include, Foreach-Object
#
# SŁOWA KLUCZOWE: aplikacja Word, automatyzacja, COM
# Get-Childitem -include, Foreach-Object
#
# COMMENTS: This script searches a folder for doc and
# docx files, opens them with Word and counts the
# number of images embedded in the file.
# It then prints out the name of each file and the
# number of associated images with the file. This script requires
# Word to be installed. It was tested with Word 2007. The folder must
# exist or the script will fail.
#
# UWAGI: skrypt ten wyszukuje w folderze pliki doc i
# docx, otwiera je w programie Word i liczy osadzone w nich
# obrazy. Następnie drukuje nazwę każdego pliku i liczbę
# związanych z nim obrazów. Skrypt ten wymaga do działania
# zainstalowanego programu Word. Został przetestowany z
# programem Word 2007. Folder musi istnieć, aby skrypt ten zadziałał.
#
# -----
#The folder must exist and be followed with a trailing \*
# Folder musi istnieć i za jego nazwą musi znajdować się napis \*.
$folder = "c:\fso\*"
$include = "*.doc", "*.docx"
$word = new-object -comobject word.application
#Makes the Word application invisible. Set to $true to see the application.
# Sprawia, że aplikacja Word staje się niewidoczna. Zmień ustawienie na $true, aby pokazać okno programu.
$word.visible = $false
Get-ChildItem -path $folder -include $include |
Foreach-Object '
{
    $doc = $word.documents.open($_.fullname)
```

```

$_name + " has " + $doc.inlineshapes.count + " images in the file"
}
#If you forget to quit Word, you will end up with multiple copies running at the same time.
#Jeśli zapomnisz zamknąć program Word, to będzie uruchomionych kilka kopii tego programu naraz.
$word.quit()

```

Zamieszczaj informacje w nagłówku

Każdy skrypt powinien zawierać informacje w nagłówku. Powinny one być spójne i zgodne ze standardami przyjętymi w firmie, w której pracujesz. Najczęściej w miejscu tym umieszcza się tytuł skryptu, nazwisko autora, datę napisania skryptu, numer wersji oraz dodatkowe uwagi. Jeśli chodzi o wersję, to nie trzeba się rozdrabniać. Wystarczy numer wersji głównej i pomocniczej. Informacje te wraz z uwagami na temat tego, co dodano w kolejnych wersjach, są przydatne przy kontroli wersji skryptów w środowisku produkcyjnym. Przykład komentarzy w nagłówku jest pokazany w poniższym skrypcie *WriteBiosInfoToWord.ps1*.

WriteBiosInfoToWord.ps1

```

#=====
#
# NAZWA: WriteBiosInfoToWord.ps1
#
# AUTOR: ed wilson , Microsoft
# DATA: 30.10.2008
# E-MAIL: Scripter@Microsoft.com
# Wersja: 1.0
#
# Opis: Tworzy nowy dokument tekstowy przy użyciu obiektu word.application.
# Wysyła zapytanie do WMI za pomocą polecenia get-wmiobject.
# Zamienia zwrócone informacje w łańcuch za pomocą polecenia out-string.
# Zapisuje dane w dokumencie programu Word za pomocą polecenia foreach-object.
#
# Hey Scripting Guy! 11.11.2008
#=====

$class = "Win32 Bios"
$path = "C:\fso\bios"

# Obiekt wdSaveFormat musi być zapisany jako typ referencyjny.
[ref]$SaveFormat = "microsoft.office.interop.word.WdSaveFormat" -as [type]

$word = New-Object -ComObject word.application
$word.visible = $true
$doc = $word.documents.add()
$selection = $word.selection
$selection.typeText("To są informacje z BIOS-u.")
$selection.TypeParagraph()

Get-WmiObject -class $class |
Out-String |
ForEach-Object { $selection.typeText($_) }
$doc.saveas([ref] $path, [ref]$SaveFormat::wdFormatDocument)
$word.quit()

```

Podaj listę warunków używania

Komentarze muszą zawierać informacje na temat warunków, jakie trzeba spełniać, aby móc uruchomić skrypt, oraz o wykorzystywaniu przez niego niestandardowych programów. Jeśli na przykład skrypt wymaga do działania jakiegoś zewnętrznego programu, który nie wchodzi w skład systemu operacyjnego, to w skrypcie tym muszą być zaimplementowane testy sprawdzające dostępność tego programu. Dodatkowo fakt ten powinien być odnotowany w dokumentacji. Jeśli w skrypcie przyjęte są jakieś założenia dotyczące istnienia pewnego katalogu, to również należy zamieścić odpowiednią informację. Oczywiście w samym skrypcie powinno się sprawdzić za pomocą polecenia `Test-Path`, czy dany katalog jest dostępny. Należy to również odnotować jako ważny warunek wstępny.

Kolejną kwestią do rozważenia jest to, czy skrypt powinien utworzyć brakujący katalog, czy nie. Jeśli skrypt potrzebuje jakiegoś pliku wejściowego, należy o tym napisać w komentarzu oraz dodać test sprawdzający, czy ten plik istnieje, zanim się go wywoła. Ponadto dobrym pomysłem jest poinformowanie w komentarzu o formacie pliku wejściowego, ponieważ jedną z najbardziej wrażliwych cech skryptu wczytującego plik jest właśnie format tego pliku. W przedstawionym poniżej skrypcie `ConvertToFahrenheit_include.ps1` pokazano przykład notatki na temat wymogu dostępności dołączanego pliku.

ConvertToFahrenheit_include.ps1

```
# -----
# NAZWA: ConvertToFahrenheit_include.ps1
# AUTOR: ed wilson, Microsoft
# DATA: 24.9.2008
# E-MAIL: Scripter@Microsoft.com
# Wersja: 2.0
# 1.12.2008 Dodano test test-path sprawdzający, czy dołączany plik istnieje.
#      Zmieniono sposób wywoływania dołączanego pliku.
# SŁOWA KLUCZOWE: konwersja stopni Celsjusza na stopnie Fahrenheita
#
# UWAGI: Skrypt zamieniający stopnie Celsjusza na stopnie Fahrenheita.
# Używa parametrów wiersza poleceń i pliku dołączanego.
# Jeśli skrypt ConversionFunctions.ps1 jest niedostępny,
# skrypt ten nie zadziała.
#
# -----
Param($Celsius)
# Zmienna $includeFile zawiera ścieżkę do skryptu ConversionFunctions.ps1.
# Należy ją odpowiednio zmodyfikować.
$includeFile = "c:\data\scriptingGuys\ConversionFunctions.ps1"
if(!(test-path -path $includeFile))
{
    "Nie znaleziono pliku $includeFile."
    Exit
}
. $includeFile
ConvertToFahrenheit($Celsius)
```

Opisuj niedoskonałości

Jeśli skrypt ma jakieś braki, to bezwzględnie należy o nich napisać. Brakiem może być tak prozaiczna sprawa jak to, że skrypt nie został jeszcze ukończony, ale fakt ten koniecznie powinien być wyraźnie odnotowany w nagłówku. Często jest tak, że ktoś zaczyna pisać skrypt, coś mu przerywa pracę, a potem zaczyna pisać nowy skrypt, bo zapomniał o poprzednim. Gdy później ktoś inny znajdzie ten pierwszy skrypt, może zacząć go używać i dziwić się, że nie działa tak, jak powinien. Dlatego niedokończone skrypty zawsze powinny być odpowiednio oznaczone. Jeśli użyje się jakiegoś specjalnego słowa kluczowego, np. **w toku**, to później można znaleźć wszystkie niedokończone skrypty za pomocą specjalnego skryptu. Oprócz tego, że skrypt jest niedokończony, należy też odnotowywać inne braki. Jeśli skrypt działa tylko na komputerze lokalnym, należy o tym wspomnieć w nagłówku. Jeżeli skrypt wyjątkowo długo działa, to o tym również trzeba napisać. Jeśli skrypt powoduje wyświetlanie jakichś błędów, ale mimo to z powodzeniem spełnia swoją funkcję, powinno się o tym poinformować użytkowników. Także jakaś informacja na temat powodu wyświetlania tych błędów postawi twórcę skryptu w lepszym świetle.

Skrypt `CmdLineArgumentsTime.ps1` działa, ale powoduje wyświetlanie informacji o błędach, jeśli nie jest używany w określonych warunkach i nie zostanie wywołany w określony sposób. W komentarzach opisano te specjalne warunki oraz dodano kilka razy informację, że praca nad skryptem jest w toku. Poniżej znajduje się treść skryptu `CmdLineArgumentsTime.ps1`.

CmdLineArgumentsTime.ps1

```
#
=====
#
# NAZWA: CmdLineArgumentsTime.ps1
# AUTOR: Ed Wilson , microsoft
# DATA : 19.2.2009
# E-MAIL: Scripter@Microsoft.com
# Wersja: .0
# SŁOWA KLUCZOWE: Add-PSSnapin, powergadgets, Get-Date
#
# UWAGI: $args[0] to pozbawiony nazwy argument przyjmujący dane z wiersza poleceń.
# C:\cmdLineArgumentsTime.ps1 23 52
# Argumentów nie rozdzielono za pomocą przecinków. Ich użycie spowoduje błąd.
# Wymaga dodatku powergadgets.
# W TOKU: Dodać funkcję pomocniczą do skryptu.
#
=====
# W TOKU: zmienić argumenty bez nazw na bardziej przyjazne dla użytkownika.
[int]$inhour = $args[0]
[int]$intMinute = $args[1]
# W TOKU: znaleźć lepszy sposób na sprawdzanie dostępności dodatku powergadgets.
# Obecna metoda powoduje błędy, które można zignorować.
$erroractionpreference = "SilentlyContinue"
# Ta instrukcja kasuje wszystkie błędy i służy do sprawdzania, czy są obecne błędy.
$error.clear()
# To polecenie spowoduje błąd, jeśli dodatek PowerGadgets nie będzie zainstalowany.
Get-PSSnapin *powergadgets | Out-Null
# W TOKU: zapytać o zgodę przed załadowaniem dodatku powergadgets.
If ($error.count -ne 0)
{Add-PSSnapin powergadgets}

New-TimeSpan -Start (get-date) -end (get-date -Hour $inhour -Minute $intMinute) |
Out-Gauge -Value minutes -Floating -refresh 0:0:30 -mainscale_max 60
```

Nie dodawaj zbędnych informacji

W kodzie skryptu nie należy wpisywać komentarzy zawierających zbędne lub niezwiązane z tematem informacje. Nie zapominaj, że piszesz skrypt i dokumentację do skryptu, a do tego potrzebna jest umiejętność pisania tekstów technicznych, nie literackich. Podczas gdy Ty możesz być zachwycony swoim skryptem, użytkownika nie obchodzi, ile wysiłku włożyłeś w jego napisanie. Chętnie natomiast dowie się, dlaczego użyłeś określonych konstrukcji zamiast innych. Takie objaśnienia mogą być przydatne dla programistów, którzy będą modyfikować skrypt w przyszłości. Dlatego komentarze wewnątrz skryptu powinny zawierać tylko informacje ułatwiające innym programistom zrozumienie sposobu jego działania. Jeśli dany komentarz nie jest cenny, to nie należy go dodawać. Skrypt *DemoConsoleBeep.ps1* zawiera kilka komentarzy w obrębie kodu źródłowego. Ale niektóre z nich zawierają oczywiste informacje, a inne powielają tylko informacje zawarte w nagłówku. Nie ma nic złego w pisaniu dużej ilości komentarzy, jeśli jednak jednowierszowy skrypt, zwłaszcza prosty, ma 20 linijek opisu, to już jest przesada. Poniżej znajduje się treść skryptu *DemoConsoleBeep.ps1*.

DemoConsoleBeep.ps1

```
# -----
# NAZWA: DemoConsoleBeep.ps1
# AUTOR: ed wilson, Microsoft
# DATA: 1.4.2009
#
# SŁOWA KLUCZOWE: Beep
#
# UWAGI: Skrypt ten demonstruje sposób użycia konsolowego polecenia
# beep. Pierwszy parametr określa częstotliwość w przedziale
# 37..32767. Powyżej 7500 prawie nic nie słychać. 37 to najniższy
# odtwarzany ton.
# Drugi parametr określa czas trwania.
#
# -----
# Konstrukcja ta tworzy tablicę liczb od 37 do 3200.
# Znak % jest aliasem polecenia Foreach-Object.
# $ to zmienna automatyczna odnosząca się do bieżącego elementu
# w potoku.
# Średnik oznacza nowy logiczny wiersz.
# Dwa dwukropki służą do odnoszenia się do metod statycznych.
# Zmienna $_ w metodzie określa liczbę znajdującą się w potoku.
# Druga liczba określa czas odtwarzania dźwięku.
37..32000 | % { $_ ; [console]::beep($_ , 1) }
```

Uzasadniaj powody napisania kodu

Wprawdzie dobrze napisany kod jest zawsze czytelny, a dobry programista potrafi się domyślić, do czego służy dany skrypt, to jednak niektórzy mogą nie wiedzieć, dlaczego dany kod został napisany w określony sposób albo dlaczego działa tak, a nie inaczej. W skrypcie *DemoConsoleBeep2.ps1* usunięto niepotrzebne komentarze. Pozostawiono tylko najważniejsze informacje dotyczące przyjmowanego przedziału liczbowego. Ponadto dodano historię wersji, ponieważ w skrypcie wprowadzono znaczne zmiany. Poniżej znajduje się treść skryptu *DemoConsoleBeep2.ps1*.

DemoConsoleBeep2.ps1

```
# -----
# NAZWA: DemoConsoleBeep2.ps1
# AUTOR: ed wilson, Microsoft
# DATA: 1.4.2009
# WERSJA: 2.0
# 4.4.2009: uporządkowano komentarze. Usunięto alias. Zmieniono formatowanie.
#
# SŁOWA KLUCZOWE: beep
#
# UWAGI: Skrypt ten demonstruje sposób użycia konsolowego polecenia
# beep. Pierwszy parametr określa częstotliwość. Przyjmuje wartości z przedziału
# 37..32767. Gdy wartość przekracza 7500, dźwięk staje się ledwie słyszalny.
# Najniższy możliwy ton ma wartość 37.
# Drugi parametr określa czas trwania.
#
# -----
37..32000 |
Foreach-Object { $_ ; [console]::beep($_ , 1) }
```

Zastosowanie komentarzy jednowierszowych

Przed komentowanym kodem powinno się umieszczać jednowierszowe komentarze objaśniające przeznaczenie zmiennych i stałych. Ponadto w komentarzach tego typu należy opisać zastosowane w kodzie sztuczki i wskazać dodatkowe źródła informacji na ich temat. Oczywiście najlepiej jest tak pisać kod, aby nie wymagał dodatkowych wewnętrznych objaśnień. Nie trzeba pisać w komentarzu tego, co w oczywisty sposób wynika z samego kodu źródłowego. Komentarze mają wyjaśniać to, czego bezpośrednio nie widać. W skrypcie *GetServicesInSvchost.ps1* w komentarzach opisano logikę mapowania własności *handle* z klasy *Win32_Process* na własność *ProcessID* z klasy *Win32_Service*, które zastosowano w celu dowiedzenia się, które usługi używają której instancji procesu *\$vchost*. Poniżej znajduje się kod źródłowy skryptu *GetServicesInSvchost.ps1*.

GetServicesInSvchost.ps1

```
# -----
# NAZWA: GetServicesInSvchost.ps1
# AUTOR: ed wilson, Microsoft
# DATA: 21.8.2008
#
# SŁOWA KLUCZOWE: Get-WmiObject, Format-Table,
# Foreach-Object
#
# UWAGI: Skrypt tworzy tablicę obiektów procesów WMI
# i pobiera uchwyt do każdego z tych obiektów.
# Zgodnie z dokumentacją MSDN uchwyt jest identyfikatorem
# procesu. Ponadto jest kluczem klasy Win32_Process. Następnie
# skrypt używa uchwytu, który jest tożsamy z własnością processID
# z klasy Win32_service, w celu określenia dopasowań.
#
# HSG 28.8.2008
# -----
```

```
$aryPid = @(Get-WmiObject win32_process -Filter "name='svchost.exe'") |
    Foreach-Object { $_.Handle }

"Działa " + $arypid.length + " instancji procesu svchost.exe."

foreach ($i in $aryPID)
{
    Write-Host "Usługi działające w ProcessID: $i" ;
    Get-WmiObject win32_service -Filter " processID = $i" |
    Format-Table name, state, startMode
}
```

Unikaj komentarzy na końcu wiersza

Należy unikać stosowania komentarzy na końcu wiersza. Ich dodatek może zaburzać logiczną strukturę kodu i znacznie pogarszać jego czytelność oraz utrudniać konserwację. Niektórzy programiści starają się niwelować szkodliwy wpływ takich komentarzy przez wyrównywanie ich do jednej linii. Ale mimo że w pierwszej chwili może się wydawać, że to bardzo dobry pomysł, w rzeczywistości szybko może stać się koszmarem, bo każda zmiana w kodzie może spowodować, że jakiś wiersz kodu wyjdzie poza linię wyrównania i wtedy trzeba będzie przesunąć wszystkie komentarze. Po kilku takich próbach każdy dostrzeże, że nie ma to sensu. Ponadto w przypadku konsoli Windows PowerShell istnieje jeszcze takie niebezpieczeństwo, że ze względu na możliwość potokowego przetwarzania danych polecenia mogą zajmować po kilka linijek. Każda linijka zakończona znakiem potoku (|) znajduje kontynuację w następnej linijce. Znak komentarza za znakiem potoku stanowi błąd, jak pokazano w poniższym przykładzie, w którym komentarz znajduje się w logicznym środku wiersza kodu. Ten kod nie zadziała:

```
Get-Process | # To polecenie pobiera listę wszystkich procesów działających na komputerze.
Select-Object -property name
```

Podobny problem występuje przy używaniu nazwanych parametrów polecenia `ForEach-Object`, jak pokazano w skrypcie *SearchAllComputersInDomain.ps1*. Znak `'` oznacza kontynuację wiersza, dzięki czemu parametry `-Begin`, `-Process` i `-End` można było wpisać w osobnych linijkach. Dzięki temu kod jest bardziej czytelny i zrozumiały. Jeśli za znakiem `'` znajdzie się komentarz, skrypt nie zadziała. Poniżej znajduje się treść skryptu *SearchAllComputersInDomain.ps1*.

SearchAllComputersInDomain.ps1

```
$Filter = "ObjectCategory=computer"
$Searcher = New-Object System.DirectoryServices.DirectorySearcher($Filter)
$Searcher.FindAll() |
Foreach-Object '
    -Begin { "Wynik zapytania $Filter: " } '
    -Process { $_.properties ; "r" } '
    -End { " Liczba znalezionych wyników $Filter:" + [string]$Searcher.FindAll().Count }
```

Opisuj struktury zagnieżdżone

Poruszone w poprzednim podrozdziale kwestie dotyczące komentarzy na końcu wiersza nie dotyczą komentarzy oznaczających zamknięcia klamer. Zasadniczo powinno się unikać głębokiego zagnieżdżania struktur, ale czasami nie jest to możliwe. W takich przypadkach dodatek komentarzy

za zamknięciami klamer może znacznie poprawić czytelność i ułatwić obsługę skryptu. W przedstawionym poniżej skrypcie *Get-MicrosoftUpdates.ps1* wszystkie zamknięcia klamer są oznaczone komentarzami.

Get-MicrosoftUpdates.ps1

```
# -----
# NAZWA: Get-MicrosoftUpdates.ps1
# AUTOR: ed wilson, Microsoft
# DATA: 25.2.2009
#
# SŁOWA KLUCZOWE: Microsoft.Update.Session, com
#
# UWAGI: Skrypt ten wyświetla listę aktualizacji firmy Microsoft.
# Można wybrać pewną liczbę aktualizacji lub wszystkie
# aktualizacje.
#
# HSG 9.3.2009
# -----

Function Get-MicrosoftUpdates
{
    Param(
        $NumberOfUpdates,
        [switch]$all
    )
    $Session = New-Object -ComObject Microsoft.Update.Session
    $Searcher = $Session.CreateUpdateSearcher()
    if($all)
    {
        $HistoryCount = $Searcher.GetTotalHistoryCount()
        $Searcher.QueryHistory(1,$HistoryCount)
    } #end if all
    Else
    {
        $Searcher.QueryHistory(1,$NumberOfUpdates)
    } #end else
} #end Get-MicrosoftUpdates

# *** punkt początkowy skryptu ***

# Wyświetla najnowszą aktualizację.
# Get-MicrosoftUpdates -NumberOfUpdates 1

# Wyświetla wszystkie aktualizacje.
Get-MicrosoftUpdates -all
```

Używaj standardowego zestawu słów kluczowych

W komentarzach informujących o błędach, usterkach lub elementach do dopracowania używaj standardowego zestawu słów kluczowych. Zasadę tę warto dodać nawet do firmowych wytycznych dotyczących pisania skryptów. Jeśli wszyscy będą jej przestrzegać, będzie można napisać skrypt znajdujący np. elementy do dopracowania. Jeśli dodatkowo zastosuje się kontrolę wersji kodu źródłowego, można dodawać komentarze informujące o wprowadzeniu poprawek. Oczywiście wówczas trzeba też zwiększyć numer wersji skryptu. Skrypt *CheckEventLog.ps1*

przyjmuje dwa parametry z wiersza poleceń. Jeden z nich powinien określać dziennik zdarzeń, do którego ma zostać wysłane zapytanie, a drugi — liczbę zdarzeń, jaka ma zostać zwrócona. Jeżeli użytkownik wybierze dziennik bezpieczeństwa po uruchomieniu skryptu bez uprawnień administratora, zostanie zgłoszony błąd, o czym jest wspomniane w bloku komentarza. Jako że taka sytuacja może być problematyczna, dodano szkic funkcji sprawdzającej, czy skrypt posiada uprawnienia administracyjne, oraz kodu sprawdzającego nazwę dziennika. W skrypcie tym znajduje się kilka znaczników DO ZROBIENIA, oznaczających, co trzeba jeszcze wykonać. Poniżej znajduje się treść skryptu *CheckEventLog.ps1*.

CheckEventLog.ps1

```
# -----
# NAZWA: CheckEventLog.ps1
# AUTOR: ed wilson, Microsoft
# DATA: 4.4.2009
#
# SŁOWA KLUCZOWE: Get-EventLog, Param, Function
#
# UWAGI: Skrypt ten przyjmuje dwa parametry: nazwę dziennika
# i liczbę zdarzeń do pobrania. Jeśli dla parametru -max
# nie zostanie podana żadna liczba, skrypt pobierze najnowszy wpis.
# Jeżeli skrypt zostanie uruchomiony bez uprawnień administratora,
# nie będzie możliwe pobranie danych z dziennika bezpieczeństwa.
# DO ZROBIENIA: Dodać funkcję sprawdzającą, czy skrypt ma uprawnienia
# administracyjne, gdy wybrany zostanie dziennik bezpieczeństwa.
# -----
Param($log,$max)
Function Get-log($log,$max)
{
    Get-EventLog -logname $log -newest $max
} #end Get-Log

# DO ZROBIENIA: dokończyć funkcję Get-AdminRights
Function Get-AdminRights
{
    # DO ZROBIENIA: Dodać kod sprawdzający posiadanie uprawnień administracyjnych.
    # DO ZROBIENIA: Jeśli skrypt nie działa jako administrator,
    #TODO: w miarę możliwości należy dodać kod zdobywający takie uprawnienia.
} #end Get-AdminRights

If(-not $log) { "Podaj nazwę dziennika." ; exit}
if(-not $max) { $max = 1 }
# DO ZROBIENIA: Włączyć test sprawdzający, czy wybrany dziennik jest
# DO ZROBIENIA: dziennikiem bezpieczeństwa.
# If($log -eq "Security") { Get-AdminRights ; exit }
Get-Log -log $log -max $max
```

Opisuj wszelkie dziwne fragmenty kodu

Ostatnia moja uwaga na temat komentarzy dotyczy komentowania wszystkiego, co jest dziwne. Jeśli użyjesz nowego typu konstrukcji, której wcześniej nie używałeś w innych skryptach, napisz o tym w komentarzu. Dobry komentarz powinien zawierać informację o tym, jaka konstrukcja została zastąpiona nową i dlaczego to zrobiono. Ogólnie rzecz biorąc, nie jest dobrym zwyczajem

używanie dziwnych konstrukcji tylko po to, aby popisać się wiedzą lub uzyskać bardziej eleganckie rozwiązanie. Przede wszystkim należy dbać o czytelność kodu źródłowego. Ale jeśli odkryjesz nową konstrukcję, która jest bardziej czytelna i zrozumiała od wcześniej używanej, to używaj jej, tylko opisz to w komentarzu. Jeżeli konstrukcja ta okaże się przydatna, należy ją wyszczególnić w firmowych wytycznych dotyczących pisania skryptów jako wzorzec projektowy.

W skrypcie *GetProcessesDisplayTempFile.ps1* występuje kilka niespodziewanych konstrukcji. Pierwsza z nich to statyczna metoda *GetTempFileName* z klasy *.NET Io.Path*. Mimo że nazwa sugeruje inaczej (pobierz nazwę pliku tymczasowego), metoda ta tworzy nazwę pliku tymczasowego i sam plik tymczasowy. Ale druga technika jest znacznie bardziej nietypowa. Gdy zawartość pliku tymczasowego zostanie wyświetlona w Notatniku, wynik operacji zostaje przekazany przez potok do polecenia *Out-Null*. To powoduje zatrzymanie wykonywania skryptu do zamknięcia Notatnika. Sztuczka ta nie jest typowa, ale przydaje się w sytuacjach, gdy trzeba usunąć pliki tymczasowe, których zawartość została wyświetlona. Oba udziwnienia skryptu *GetProcessesDisplayTempFile.ps1* zostały opisane w komentarzach, jak widać poniżej.

GetProcessesDisplayTempFile.ps1

```
# -----
# NAZWA: GetProcessesDisplayTempFile.ps1
# AUTOR: ed wilson, Microsoft
# DATA: 4/4/2009
# WERSJA: 1.0
#
# SŁOWA KLUCZOWE: [io.path], GetTempFileName, out-null
#
# UWAGI: Skrypt ten tworzy plik tymczasowy,
# pobiera kolekcję danych dotyczących procesu i zapisuje
# ją w pliku tymczasowym. Następnie wyświetla te informacje
# w Notatniku i po zakończeniu usuwa ten plik tymczasowy.
#
# -----
# Funkcja ta tworzy zarówno nazwę pliku, jak i sam plik.
$tempFile = [io.path]::GetTempFileName()
Get-Process >> $tempFile
# Przekazanie przez potok nazwy pliku Notatnika do polecenia Out-Null
# powoduje zatrzymanie wykonywania skryptu.
Notepad $tempFile | Out-Null
# Po zamknięciu pliku zostaje on usunięty.
Remove-Item $tempFile
```

Wiedza tajemna

Jak nauczyć skrypty komunikacji

Peter Costantini

Emerytowany specjalista od skryptów w firmie Microsoft

Gdyby skrypty były czytane tylko przez komputery, to można by pisać kod w postaci samych zer i jedynek. Programiści szybko by od tego oślepli i zwariowali, chociaż z drugiej strony, co roku jest napływ nowych absolwentów informatyki. Oczywiście kod musi nadawać się także do czytania przez ludzi, dlatego języki programowania zaprojektowano tak, aby odpowiadały zarówno ludziom, jak i komputerom.

Gdyby jeden programista mógł napisać, oczyścić z błędów i przetestować każdy rodzaj kodu w aplikacji, to nie miałoby znaczenia, czy używany przez niego język programowania jest zrozumiały także dla innych. Genialny samotnik mógłby wybrać jakiś mało znany dialekt języka Lisp oraz nadawać zmiennym i procedurom nazwy w języku esperanto i wszystko byłoby w porządku, dopóki jego produkty by działały.

Ale po pięciu latach język ten mógłby już nie być tak atrakcyjny, a programista mógłby trochę zapomnieć, jak się go używa. I nagle okazuje się, że niezwykle ważna aplikacja przestaje działać, narażając firmę na miliardowe straty.

Pewnie pomyślisz sobie: „parę miliardów w jedną czy w drugą stronę, cóż to takiego w tych czasach?”. Ale Ty nie jesteś programistą, tylko specjalistą od systemów próbującym zautomatyzować wykonywanie niektórych czynności za pomocą skryptów. Zawsze myślałeś, że skrypty pisze się naprędce, aby szybko coś zrobić.

Tak, to jest jedna z wielkich zalet skryptów. Gdy ktoś pisze pierwszy w życiu skrypt, to nie przejmuje się tym, czy jego kod pięknie wygląda ani nawet czy jest zrozumiały. Zadowolona się tym, że działa on zgodnie z oczekiwaniami i likwiduje problem.

Ale jeśli postanowisz zachować skrypt, aby uruchamiać go w każdy poniedziałek o trzeciej w nocy, sytuacja radykalnie się zmienia. W tym momencie, czy Ci się to podoba, czy nie, stajesz się programistą. Windows PowerShell to język programowania, chociaż dynamiczny, i każdemu fragmentowi kodu, który ma ciągle wpływ na działanie organizacji, należy poświęcać odpowiednio dużo uwagi.

Ponadto niezależnie od tego, jaki masz stosunek do pisanych przez siebie skryptów, na pewno należysz do jakiegoś zespołu. Inni członkowie tej grupy też mogą pisać skrypty. Wielu z nich zapewne musi używać Twoich skryptów i przy okazji zgadywać, co one robią. Pewnie domyślasz się już, do czego zmierzam. Jeśli właśnie doznałeś ośnienia, to dobrze dla Twojej kariery i firmy, w której pracujesz.

Chodzi o to, by skrypty były transparentne. Twój kod i środowisko, w którym działa, powinny dostarczać Twoim kolegom i koleżankom z zespołu wszystkich informacji potrzebnych do jego zrozumienia, skutecznego używania oraz diagnozowania w razie problemów. (W skryptach też obowiązuje prawo Murphy’ego). Klarowność i czytelność to cnoty. Natomiast lapidarność i dwuznaczność — nie. Spójne i opisowe nazewnictwo zmiennych oraz stosowanie białych znaków nie powodują spowolnienia działania kodu, ale sprawiają, że skrypty są bardziej czytelne. Zaczynj traktować transparentność jak polisę ubezpieczeniową na wypadek otrzymania telefonu od przerażonego klienta, gdy opalasz się na plaży na Hawajach, popijając margaritę.

To, o czym piszę, nie jest tylko kwestią techniczną czy dotyczącą wyłącznie współpracy. Ma to również wymiar ekonomiczny. Działy informatyczne bardzo starają się, aby być ważnymi strategicznie jednostkami firmy, a nie jedynie generującymi koszty centrami. Pisane w nich skrypty i inne programy mogą przyczyniać się do dużych zysków, jak również do strat finansowych, a w konsekwencji mogą powodować nadwyżki budżetowe, jak również zwolnienia. W porządku, w tym roku przynajmniej dodaj dobrą dokumentację do skryptów, a może zmniejszysz cięcia budżetowe.

Dodatkowe źródła informacji

- Centrum skryptów portalu TechNet pod adresem <http://www.microsoft.com/technet/scriptcenter> zawiera wiele przykładów skryptów Windows PowerShell oraz parę przykładowych szablonów dokumentacji.
- Warto przeczytać artykuł pt. *How Can I Delete and Manage PDF Files?* na stronie <http://www.microsoft.com/technet/scriptcenter/resources/qanda/dec08/hey1229.mspix>.
- Warto przeczytać artykuł pt. *How Can I Create a Microsoft Word Document from WMI Information?* na stronie <http://www.microsoft.com/technet/scriptcenter/resources/qanda/nov08/hey1111.mspix>.
- Warto przeczytać książkę pt. *Windows Scripting with WMI: Self-Paced Learning Guide* (Microsoft Press 2006).
- Wszystkie skrypty opisane w tym rozdziale są dostępne do pobrania w repozytorium centrum skryptów portalu TechNet pod adresem <http://gallery.technet.microsoft.com/scriptcenter/PowerShell-40-Best-d9e16039>.

Rozdział 10

Projektowanie modułów

- Podstawowe informacje o modułach
- Znajdowanie i ładowanie modułów
- Instalowanie modułów
- Tworzenie modułów
- Dodatkowe źródła informacji

Moduły Windows PowerShell są wygodnym sposobem na tworzenie pakietów kodu nadającego się do wielokrotnego użytku. Można je przenosić między komputerami, przekazywać innym użytkownikom oraz można używać ich w skryptach. Rozwiązują one wiele problemów, które mogłyby nękać początkującego skrypciarza. Na początku tego rozdziału poznasz podstawowe wiadomości na temat modułów i dowiesz się, kiedy warto je tworzyć. Potem przejdziemy do analizy pewnych decyzji, których podjęcie wymuszają moduły, a na koniec utworzymy przykładowy moduł.

Podstawowe wiadomości o modułach

Pojęcie modułu zostało wprowadzone w Windows PowerShell 2.0. Moduł to pakiet mogący zawierać polecenia cmdlet, aliasy, funkcje, zmienne, a nawet dostawców. Krótko mówiąc, moduł Windows PowerShell może zawierać wszystko to, co można umieścić w profilu, a dodatkowo także to, co w Windows PowerShell 1.0 trzeba było dodawać w przystawkach. Moduły w porównaniu z przystawkami mają parę zalet, na przykład:

- Moduł może napisać każdy, kto umie napisać skrypt Windows PowerShell.
- Aby zainstalować moduł, nie trzeba pisać pakietu instalacyjnego instalatora systemu Windows.
- Aby zainstalować moduł, nie trzeba mieć uprawnień administratora.

Wszystkie te zalety są bardzo ważne dla informatyka.

Znajdowanie i ładowanie modułów

Moduły Windows PowerShell są przechowywane w trzech domyślnych miejscach. Pierwsze to katalog główny użytkownika, a drugie to katalog główny Windows PowerShell. Natomiast trzecią lokalizację wprowadzono w Windows PowerShell 4.0 i jest to folder *Program Files\WindowsPowerShell\Modules*. Zaletą tej nowej lokalizacji jest to, że nie trzeba mieć uprawnień administratora, aby w niej instalować (w odróżnieniu od folderu *System32*). Nie jest też przypisana do konkretnego użytkownika (jak katalog główny użytkownika).

Katalog z modułami w katalogu głównym Windows PowerShell jest zawsze dostępny, podobnie jak moduły znajdujące się w katalogu *Program Files\WindowsPowerShell*. Natomiast katalog z modułami znajdujący się w katalogu głównym użytkownika domyślnie nie jest dostępny. Aby był dostępny, najpierw trzeba go utworzyć. Utworzenie tego katalogu następuje dopiero w chwili, gdy ktoś postanowi zapisać w nim moduły. Przydatną cechą tego katalogu jest to, że jeśli istnieje, konsola Windows PowerShell od niego zaczyna szukanie modułów. Jeśli katalog modułów użytkownika nie istnieje, następuje przeszukanie katalogu z modułami w katalogu głównym Windows PowerShell.

Wyświetlanie listy dostępnych modułów

Moduły Windows PowerShell mogą egzystować w dwóch stanach: załadowany i niezaładowany. Aby wyświetlić listę wszystkich załadowanych modułów, należy wykonać polecenie `Get-Module` bez parametrów, jak pokazano poniżej:

```
PS C:\> Get-Module
ModuleType Name
-----
Script ISE
Manifest Microsoft.PowerShell.Management
Manifest Microsoft.PowerShell.Utility
```

| ModuleType | Name | ExportedCommands |
|------------|---------------------------------|--|
| Script | ISE | {Get-IseSnippet, Import-IseSnippet, New-IseSnip... |
| Manifest | Microsoft.PowerShell.Management | {Add-Computer, Add-Content, Checkpoint-Computer... |
| Manifest | Microsoft.PowerShell.Utility | {Add-Member, Add-Type, Clear-Variable, Compare-... |

Jeżeli podczas wykonywania polecenia `Get-Module` jest załadowanych wiele modułów, dla każdego z nich zostaną wyświetlone jego eksportowane polecenia, jak widać poniżej:

```
PS C:\> Get-Module
ModuleType Name
-----
Script GetFreeDiskSpace
Script HelloWorld
Script TextFunctions
Manifest BitsTransfer
Script PSDiagnostics
```

| ModuleType | Name | ExportedCommands |
|------------|------------------|--|
| Script | GetFreeDiskSpace | Get-FreeDiskSpace |
| Script | HelloWorld | {Hello-World, Hello-User} |
| Script | TextFunctions | {New-Line, Get-TextStats} |
| Manifest | BitsTransfer | {Start-BitsTransfer, Remove-BitsTransfe... |
| Script | PSDiagnostics | {Enable-PSTrace, Enable-WManTrace, Sta... |

PS C:\>

Jeśli żaden moduł nie jest załadowany, w konsoli Windows PowerShell nic nie zostanie wyświetlone. Nie pojawi się żadna informacja o błędzie ani żadne potwierdzenie, że polecenie zostało wykonane. W systemie Windows 8 sytuacja taka nie występuje, ponieważ podstawowe polecenia Windows PowerShell znajdują się w dwóch głównych modułach: `Microsoft.PowerShell.Management` i `Microsoft.PowerShell.Utility`. Te dwa moduły są zawsze załadowane, chyba że ktoś uruchomi konsolę z przełącznikiem `-noprofile`. Ale nawet wtedy zostanie załadowany moduł `Microsoft.PowerShell.Management`.

Aby wyświetlić listę wszystkich modułów, które są dostępne w systemie, ale nie są załadowane, można użyć polecenia `Get-Module -ListAvailable`. Spowoduje to wyświetlenie wszystkich dostępnych modułów, zarówno załadowanych, jak i niezaładowanych, do konsoli Windows PowerShell. Poniżej pokazano wynik zarejestrowany w domyślnej instalacji systemu Windows 8.

UWAGA

Konsola Windows PowerShell nadal jest instalowana w katalogu `%windir%\system32\Windows-PowerShell\v1.0` (nawet w systemie Windows 8.1). Powodem tego jest chęć zachowania zgodności ze starszymi aplikacjami, które używają tej lokalizacji. Pytania na ten temat często otrzymuję za pośrednictwem bloga „Hey, Scripting Guy!” (www.scriptingguys.com/blog). Niezawodnym sposobem na sprawdzenie wersji konsoli Windows PowerShell jest użycie zmiennej `$PSVersionTable`.

```
PS C:\> Get-Module -ListAvailable
```

```
Directory: C:\Windows\system32\WindowsPowerShell\v1.0\Modules
```

| ModuleType | Name | ExportedCommands |
|------------|----------------------------------|--|
| ----- | ----- | ----- |
| Manifest | AppLocker | {Get-AppLockerFileInformation, Get... |
| Manifest | Appx | {Add-AppxPackage, Get-AppxPackage,... |
| Manifest | BitLocker | {Unlock-BitLocker, Suspend-BitLock... |
| Manifest | BitsTransfer | {Add-BitsFile, Complete-BitsTransf... |
| Manifest | BranchCache | {Add-BCDataCacheExtension, Clear-B... |
| Manifest | CimCmdlets | {Get-CimAssociatedInstance, Get-Ci... |
| Manifest | DirectAccessClientComponents | {Disable-DAManualEntryPointSelecti... |
| Script | Dism | {Add-AppxProvisionedPackage, Add-W... |
| Manifest | DnsClient | {Resolve-DnsName, Clear-DnsClientC... |
| Manifest | International | {Get-WinDefaultInputMethodOverride... |
| Manifest | iSCSI | {Get-IscsiTargetPortal, New-IscsiT... |
| Script | ISE | {New-IseSnippet, Import-IseSnippet... |
| Manifest | Kds | {Add-KdsRootKey, Get-KdsRootKey, T... |
| Manifest | Microsoft.PowerShell.Diagnostics | {Get-WinEvent, Get-Counter, Import... |
| Manifest | Microsoft.PowerShell.Host | {Start-Transcript, Stop-Transcript} |
| Manifest | Microsoft.PowerShell.Management | {Add-Content, Clear-Content, Clear... |
| Manifest | Microsoft.PowerShell.Security | {Get-Acl, Set-Acl, Get-PfxCertific... |
| Manifest | Microsoft.PowerShell.Utility | {Format-List, Format-Custom, Forma... |
| Manifest | Microsoft.WSMan.Management | {Disable-WSManCredSSP, Enable-WSMa... |
| Manifest | MMAgent | {Disable-MMAgent, Enable-MMAgent, ...} |
| Manifest | MsDtc | {New-DtcDiagnosticTransaction, Com... |
| Manifest | NetAdapter | {Disable-NetAdapter, Disable-NetAd... |
| Manifest | NetConnection | {Get-NetConnectionProfile, Set-Net... |
| Manifest | NetLbfo | {Add-NetLbfoTeamMember, Add-NetLbf... |
| Manifest | NetQos | {Get-NetQosPolicy, Set-NetQosPolic... |
| Manifest | NetSecurity | {Get-DAPolicyChange, New-NetIPsecA... |
| Manifest | NetSwitchTeam | {New-NetSwitchTeam, Remove-NetSwit... |
| Manifest | NetTCPIP | {Get-NetIPAddress, Get-NetIPInterf... |
| Manifest | NetworkConnectivityStatus | {Get-DAConnectionStatus, Get-NCSIP... |
| Manifest | NetworkTransition | {Add-NetIPHttpsCertBinding, Disabl... |
| Manifest | PKI | {Add-CertificateEnrollmentPolicySe... |
| Manifest | PrintManagement | {Add-Printer, Add-PrinterDriver, A... |
| Script | PSDiagnostics | {Disable-PSTrace, Disable-PSWSManC... |
| Binary | PSScheduledJob | {New-JobTrigger, Add-JobTrigger, R... |
| Manifest | PSWorkflow | {New-PSWorkflowExecutionOption, Ne... |

| | | |
|----------|-------------------------|---------------------------------------|
| Manifest | PSWorkflowUtility | {Invoke-AsWorkflow |
| Manifest | ScheduledTasks | {Get-ScheduledTask, Set-ScheduledT... |
| Manifest | SecureBoot | {Confirm-SecureBootUEFI, Set-Secur... |
| Manifest | SmbShare | {Get-SmbShare, Remove-SmbShare, Se... |
| Manifest | SmbWitness | {Get-SmbWitnessClient, Move-SmbWit... |
| Manifest | Storage | {Add-InitiatorIdToMaskingSet, Add-... |
| Manifest | TroubleshootingPack | {Get-TroubleshootingPack, Invoke-T... |
| Manifest | TrustedPlatformModule | {Get-Tpm, Initialize-Tpm, Clear-Tp... |
| Manifest | VpnClient | {Add-VpnConnection, Set-VpnConnect... |
| Manifest | Wdac | {Get-OdbcDriver, Set-OdbcDriver, G... |
| Manifest | WindowsDeveloperLicense | {Get-WindowsDeveloperLicense, Show... |
| Script | WindowsErrorReporting | {Enable-WindowsErrorReporting, Dis... |

Zapiski praktyka

Keith Hill

MVP Microsoft Windows PowerShell

Jedną z moich ulubionych cech konsoli Windows PowerShell są moduły. Programiści mogą w nich pakować swój kod wielokrotnego użytku i przenosić go w inne miejsce. Użytkownicy modułów mają natomiast łatwy sposób instalacji nowych funkcji.

Kiedy przystępuję do pisania skryptu, to z reguły od razu wiem, czy będzie to prosty skrypt, czy cały moduł. Oto kilka wskazówek, dzięki którym można poznać, czy potrzebny jest moduł:

- **Potrzebna funkcjonalność będzie eksponować kilka poleceń.** W takich przypadkach moduł jest doskonałym rozwiązaniem, ponieważ można wyeksportować tylko niektóre funkcje, np. prywatne funkcje pomocnicze.
- **Trzeba zbudować polecenia PowerShell dotyczące źródła nienależącego do Windows PowerShell.** W tym przypadku moduł jest świetnym rozwiązaniem, ponieważ można utworzyć prywatną zmienną do zarządzania zasobem i nie zaśmiecać globalnej sesji użytkownika. A dzięki automatycznemu ładowaniu modułów użytkownik może używać naszych poleceń bez potrzeby uprzedniego dołączania skryptów za pomocą notacji kropkowej.
- **Wiadomo, że budowana funkcjonalność zostanie wykorzystana w wielu skryptach — tzn. wiadomo, że kod znajdzie się w bibliotece.** Moduły doskonale nadają się do tworzenia bibliotek, ponieważ stanowią zamknięte pakiety danych i funkcji, kontrolują, co jest eksportowane do publicznego interfejsu API, oraz pozwalają na łatwe ładowanie i usuwanie funkcji, co jest pożądaną cechą każdej biblioteki.

Jeśli potrzebuję skryptu wykonującego pojedyncze zadania, to tworzę prosty skrypt `.ps1`. Ale od czasu do czasu przekształcam takie proste skrypty w moduły, gdy dodam do nich trochę poleceń. Pod tym względem moje moduły przypominają skupiska poleceń (albo czasowników) zebranych wokół jednego podstawowego rzeczownika, np. `Enable-NetGearSwitchPort`, `Disable-NetGearSwitchPort` lub `Get-NetGearSwitchPort`.

Programistom narzędzia do pracy zdalnej Windows PowerShell okazały się o wiele bardziej przydatne, niż początkowo myślałem. Przeprowadzaliśmy nocne testy regresyjne na wielu różnych komputerach. Dla mnie bardzo przydatna była możliwość zbierania ostrzeżeń

i błędów z dzienników zdarzeń ze wszystkich komputerów za pomocą skryptu. Dzięki niej zaoszczędziłem mnóstwo czasu, ponieważ nie musiałem ręcznie pobierać danych z każdego komputera i przeczesywać jego dziennika zdarzeń.

Jako programista języka C# z dużym doświadczeniem w tworzeniu binarnych poleceń cmdlet dla PSCX dziwię się, że obecnie więcej czasu spędzam na pisaniu zaawansowanych funkcji w Windows PowerShell ISE niż na pisaniu poleceń w języku C#. Jedną z zalet zaawansowanych funkcji w porównaniu z poleceniami binarnymi jest dokumentacja pomocnicza. W funkcjach wystarczy dodać parę komentarzy i Windows PowerShell zajmie się resztą. W PSCX przy tworzeniu poleceń binarnych musimy posługiwać się plikami MAML (XML) i uwierz mi, to nie jest przyjemne. W istocie spora część funkcjonalności PSCX została zaimplementowana przy użyciu funkcji zaawansowanych. Do poleceń binarnych wracam tylko wtedy, gdy zależy mi na wydajności albo gdy podstawowy interfejs API .NET łatwiej jest obsługiwać z poziomu języka C#.

Nadal używam konsoli Windows PowerShell do codziennych czynności, np. do uruchamiania poleceń i eksperymentowania z jednoliniowymi poleceniami. Ale jeśli mam utworzyć, edytować, przetestować lub zdiagnozować skrypt, używam Windows PowerShell ISE.

Jako programista codziennie używam Windows PowerShell jako narzędzia do przyspieszonego wykonywania różnych czynności, np. przeszukiwania kodu źródłowego, zarządzania plikami źródłowymi oraz znajdowania zablokowanych procesów. Ponadto za pomocą Windows PowerShell obsługuję proces kompilacji naszego produktu i nocne testy regresyjne.

Gdybym miał coś poradzić nowicuszowi, powiedziałbym, żeby trzymał się tego narzędzia. Nauka obsługi Windows PowerShell — jak każdego innego „potężnego”, ale skomplikowanego narzędzia — wymaga trochę wysiłku. Ale jeśli Ci się uda przez to przebrnąć, to inwestycja ta zwróci Ci się wielokrotnie. Poza tym nie musisz się szarpać, jeśli czegoś nie wiesz. W portalu StackOverflow udziela się wielu specjalistów od Windows PowerShell, którzy chętnie Ci pomogą (<http://stackoverflow.com/questions/tagged/powershell>). Zanim zadasz pytanie, zawsze przejrzyj wcześniejsze tematy. Możliwe, że ktoś już pytał o to samo i odpowiedź jest już gotowa.

Ładowanie modułów

Po znalezieniu modułu, który chcemy załadować, ładujemy go do bieżącej sesji Windows PowerShell za pomocą polecenia `Import-Module`, jak pokazano poniżej:

```
PS C:\> Import-Module -Name NetConnection
PS C:\>
```

Jeśli wczytywany moduł istnieje, polecenie `Import-Module` załaduje go i nie wyświetli żadnej informacji. Jeżeli moduł jest już załadowany, również nie zostanie wyświetlona żadna informacja o błędzie. Pokazano to poniżej. Za pomocą klawiszy strzałki w górę i *Enter* wykonano trzykrotnie to samo polecenie i nie spowodowało to jakiegokolwiek błędu.

```
PS C:\> Import-Module -Name NetConnection
PS C:\> Import-Module -Name NetConnection
PS C:\> Import-Module -Name NetConnection
PS C:\>
```

Po zaimportowaniu modułu za pomocą polecenia `Get-Module` można wyświetlić znajdujące się w nim funkcje. Nie trzeba podawać całej nazwy modułu. Można użyć symboli wieloznacznych albo skorzystać z funkcji rozwijania nazw za pomocą klawisza *Tab*. Poniżej pokazano zastosowanie techniki z symbolami wieloznacznymi:

```
PS C:\> Get-Module net*
ModuleType Name
-----
Manifest netconnection
ExportedCommands
-----
{Get-NetConnectionProfile, Set-Net...
```

Jak widać w tym przykładzie, moduł `netconnection` eksportuje dwa polecenia: funkcję `Get-NetConnectionProfile` i jakieś inne, zapewne o nazwie `Set-NetConnectionProfile`. Jedną z wad polecenia `Get-Module` jest to, że obcina nazwy w kolumnie *ExportedCommands*. Problem ten można rozwiązać, przekazując potokowo otrzymany obiekt `PSModuleInfo` do polecenia `Select-Object` oraz rozwijając własność `ExportedCommands`. Poniżej przedstawiono przykład zastosowania tej techniki:

```
PS C:\> Get-Module net* | select -expand *comm*
Key Value
---
Get-NetConnectionProfile Get-NetConnectionProfile
Set-NetConnectionProfile Set-NetConnectionProfile
```

Gdy trzeba załadować moduł o długiej nazwie, nie jest konieczne wpisywanie całej tej nazwy. Można użyć symboli wieloznacznych lub funkcji rozwijania nazw za pomocą klawisza *Tab*. W przypadku użycia symboli wieloznacznych powinno się wpisać dużą część nazwy, aby spośród długiej listy dostępnych modułów został wybrany ten, który chcemy. Jeśli do podanego ciągu wieloznacznego będzie pasować więcej niż jeden moduł, wystąpi błąd. W poniższym przykładzie wystąpi taki błąd, ponieważ ciąg wieloznaczny `net*` pasuje do kilku nazw modułów:

```
PS C:\> Import-Module net*
Import-Module : The specified module 'net*' was not loaded because no valid module file was found
in any module directory.
At line:1 char:1
+ Import-Module net*
+ ~~~~~
+ CategoryInfo          : ResourceUnavailable: (net*:String) [Import-Module], FileNotFoundException
+ FullyQualifiedErrorId : Modules_ModuleNotFound,Microsoft.PowerShell.Commands.ImportModuleCommand
```

WAŻNE

W Windows PowerShell 2.0, jeśli ciąg wieloznaczny pasuje do kilku nazw modułów, ładowany jest pierwszy dopasowany moduł, a pozostałe są ignorowane. To powoduje powstawanie niespójnych i nieprzewidywalnych wyników. Dlatego w Windows PowerShell 3.0 zmieniono to tak, aby w tego typu sytuacjach były zwracane błędy.

Jeśli trzeba załadować wszystkie dostępne w systemie moduły, można użyć polecenia `Get-Module` z parametrem `-ListAvailable` i otrzymane w wyniku obiekty `PSModuleInfo` przekazać do polecenia `Import-Module`, jak pokazano poniżej:

```
PS C:\> Get-Module -ListAvailable | Import-Module
PS C:\>
```

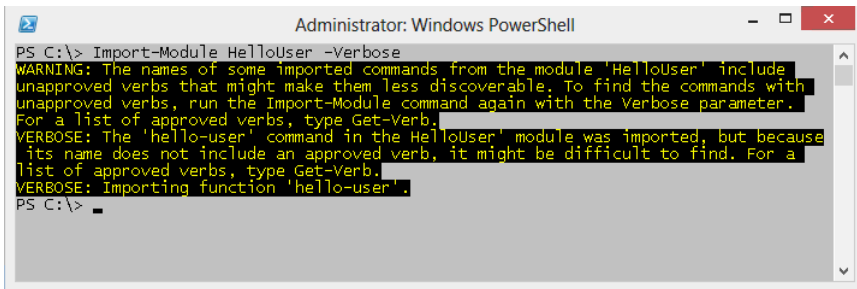
Jeśli trzeba załadować moduł używający czasownika spoza listy zatwierdzonych czasowników, to podczas jego ładowania zostanie wyświetlone ostrzeżenie. Funkcje z tego modułu będą działać i sam moduł też, a ostrzeżenie ma na celu tylko uczulić użytkownika na fakt, że istnieje lista zatwierdzonych czasowników. Pokazano to w poniższym przykładzie:

```
PS C:\> Import-Module HelloUser
WARNING: The names of some imported commands from the module 'HelloUser' include unapproved verbs that might make them less discoverable. To find the commands with unapproved verbs, run the Import-Module command again with the Verbose parameter. For a list of approved verbs, type Get-Verb.
PS C:\> hello-user
hello administrator
```

Aby sprawdzić, które z użytych czasowników są niezatwierdzone, można wykonać polecenie `Import-Module` z parametrem `-Verbose`, jak pokazano poniżej:

```
PS C:\> Import-Module HelloUser -Verbose
```

Wynik wykonania tego polecenia pokazano na rysunku 10.1.



RYSUNEK 10.1. Parametr `-Verbose` polecenia `Import-Module` powoduje wyświetlenie informacji o wszystkich funkcjach oraz niedozwolonych czasownikach użytych w nazwach. Do grupy tej zalicza się czasownik `hello` obecny w nazwie `Hello-User`

W podrozdziale tym dowiedziałeś się, na czym polega wyszukiwanie i ładowanie modułów. Do wyświetlania ich listy służy polecenie `Get-Module` z parametrem `ListAvailable`. Moduły ładuje się za pomocą polecenia `Import-Module`.

Zapiski praktyka

Jim Christopher, MVP Microsoft PowerShell

Niezależny programista, Code Owls LLC

„Właśnie w ciągu jednego popołudnia opracowałeś zastępstwo dla narzędzi programowych, których budowa trwała trzy lata”.

Dokładnie takie słowa powiedział do mnie klient kilka miesięcy temu i jest to chyba najlepszy komplement, jaki może usłyszeć ktoś, kto tworzy narzędzia przy użyciu Windows PowerShell. Programiści lubią opracowywać rozwiązania oparte na dobrze znanych schematach. Tworzą

i analizują przypadki użycia, aby wykryć w nich prawidłowości, na których potem opierają budowę programów. Jest to bardzo dobre podejście i warto je stosować, ponieważ pozwala uzyskać lepszej jakości programy.

Oprzrządowanie aplikacji — elementy wykorzystujące dane instrumentacji, fragmenty używane przez inżynierów pomocy technicznej do rozwiązywania problemów — musi spełniać wymagania całkiem innej klasy użytkowników, którą lubię nazywać **umysłem nadrzędnym**. Podczas gdy sztywne schematy są dobre dla użytkownika końcowego, umysł nadrzędny do wykonania pracy potrzebuje dużej elastyczności. Użytkownicy końcowi podążają utartymi szlakami. Umysły nadrzędne muszą mieć możliwość przecierania własnych szlaków.

U mojego klienta wszystkie narzędzia działały z założeniem, że umysły nadrzędne powinny używać internetowej witryny zarządzania. To spowalniało cykl narzędziowy: każda ścieżka wsparcia wymagała uwagi programisty stron internetowych. Dodatek warstwy GUI przyczynił się do powstania większej liczby błędów. A infrastruktura potrzebna do wdrożenia i obsługi działającej witryny zarządzania wymagała własnej warstwy narzędzi. To utrudniało spełnianie wymagań klientów, a niezadowoleni klienci przechodzą ze swoimi pieniędzmi do konkurencji.

W związku z tym pewnego popołudnia utworzyłem moduł Windows PowerShell spełniający potrzeby związane z zarządzaniem w tej firmie. Zamiast otwierać zakładkę *Users* na stronie internetowej, inżynier może użyć polecenia `Get-User`. Zamiast wyszukiwać i aktualizować zasoby pojedynczo, inżynier może wykonywać te czynności dla wielu zasobów naraz. Choć przytoczony cytat zawiera prawdę — moduł ten w istocie zastąpił wszystkie wcześniej używane narzędzia — rezultaty tego projektu były znacznie ciekawsze. Po pierwsze: w zespołach ds. wsparcia i programowania znacznie podniosło się morale, ponieważ od tej pory można było aktywnie świadczyć pomoc techniczną. Programiści natomiast ucieszyli się z tego, że mogą się skupić na cechach produktu zamiast na kwestiach dotyczących wsparcia. Po drugie: średni czas rozwiązywania problemów został skrócony z kilku dni do kilku minut. Inżynierowie nie tylko otrzymali odpowiednie narzędzia, aby szybko reagować na problemy, ale również odkryli możliwość zapisywania swoich rozwiązań często powtarzających się problemów w skryptach.

Uważam, że zwrot z inwestycji tych paru godzin pracy jest niesamowity.

Dlatego właśnie konsola Windows PowerShell jest idealnym wyborem do tworzenia oprzrządowania. Przekazuje moc decyzyjną w ręce umysłów nadrzędnych, które muszą wykonywać swoją pracę, i robi to bez zbędnych kłopotów.

Instalowanie modułów

Jedną z cech modułów jest to, że można je instalować, nie posiadając podwyższonych uprawnień. Jako że każdy użytkownik ma folder na moduły w swoim katalogu `%userprofile%`, do którego ma wszelkie prawa dostępu, instalacja modułu w tym folderze nie wymaga zwiększania uprawnień do poziomu administratora. Kolejną ważną cechą modułów jest to, że do ich instalacji nie potrzeba specjalnego instalatora. Pliki związane z modułem można skopiować za pomocą narzędzia `XCOPY` albo poleceń Windows PowerShell.

Tworzenie folderu na moduły

Domyślnie folder użytkownika na moduły nie istnieje. Aby uniknąć nieporozumień, przed instalacją modułu można utworzyć ten folder w profilu użytkownika, ale można też utworzyć skrypt instalacyjny sprawdzający, czy ten folder istnieje, i tworzący go w razie potrzeby, a następnie kopiujący do niego potrzebne pliki. Przy bezpośrednim dostępie do folderu z modułami użytkownika (*Modules*) należy pamiętać, że jego lokalizacja może się różnić w zależności od systemu operacyjnego. W systemach Windows XP i Windows Server 2003 folder na moduły użytkownika znajduje się w folderze *Moje dokumenty*. Natomiast w systemach Windows Vista i nowszych folder ten znajduje się w folderze *Dokumenty*.

UWAGA

Konsoli Windows PowerShell 4.0 nie da się zainstalować w systemach starszych niż Windows Vista. W efekcie w czystym środowisku Windows PowerShell 4.0 można pominąć sprawdzanie wersji systemu operacyjnego i po prostu utworzyć folder na moduły w folderze *Dokumenty*.

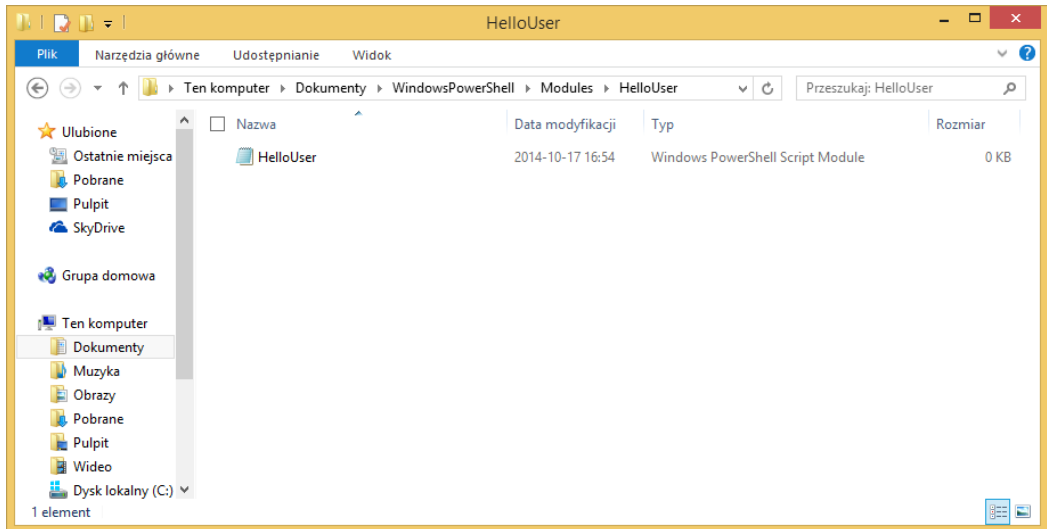
Poniżej znajduje się treść skryptu *Copy-Modules.ps1*, który rozwiązuje problem różnych lokalizacji folderu na moduły przy użyciu funkcji *Get-OperatingSystemVersion*. Funkcja ta pobiera numer wersji głównej systemu operacyjnego. Jej kod widać poniżej:

```
Function Get-OperatingSystemVersion
{
    (Get-WmiObject -Class Win32_OperatingSystem).Version
} #end Get-OperatingSystemVersion
```

Funkcja *Test-ModulePath* używa numeru wersji głównej systemu operacyjnego. Jeśli numer ten jest większy niż 6, to znaczy, że używany jest przynajmniej system Windows Vista, więc na ścieżce do modułów powinien znajdować się folder *Dokumenty*. Jeśli natomiast numer wersji systemu operacyjnego nie jest większy od 6, to skrypt użyje folderu *Moje dokumenty*. Po sprawdzeniu wersji systemu operacyjnego i zatwierdzeniu ścieżki do folderu na moduły, należy sprawdzić, czy folder ten istnieje. Do tego doskonale nadaje się polecenie *Test-Path*, które zwraca wartość logiczną. Jako że nas interesuje tylko brak obecności folderu, możemy użyć operatora negacji *-not*, jak widać w pokazanym poniżej kompletnym kodzie funkcji *Test-ModulePath*:

```
Function Test-ModulePath
{
    $VistaPath = "$env:userProfile\documents\WindowsPowerShell\Modules"
    $XPPath = "$env:Userprofile\my documents\WindowsPowerShell\Modules"
    if ([int](Get-OperatingSystemVersion).substring(0,1) -ge 6)
    {
        if(-not(Test-Path -path $VistaPath))
        {
            New-Item -Path $VistaPath -itemtype directory | Out-Null
        } #end if
    } #end if
    Else
    {
        if(-not(Test-Path -path $XPPath))
        {
            New-Item -path $XPPath -itemtype directory | Out-Null
        } #end if
    } #end else
} #end Test-ModulePath
```

Po utworzeniu folderu na moduły użytkownika należy utworzyć podfolder do przechowywania nowego modułu. Moduł instaluje się w folderze o takiej samej nazwie jak jego. Nazwa modułu jest taka sama jak nazwa zawierającego go pliku, tylko bez rozszerzenia *.psm1*, jak widać na rysunku 10.2.



RYСУNEK 10.2. Moduły umieszcza się w katalogu na moduły użytkownika

W funkcji *Copy-Module* ze skryptu *Copy-Modules.ps1* pierwszą czynnością jest pobranie wartości zmiennej środowiskowej *PSModulePath*. Jako że moduły mogą być zapisywane w dwóch lokalizacjach, zmienna ta zawiera dwie ścieżki. Ale nie jest ona tablicą, tylko łańcuchem. Poniżej pokazano jej wartość:

```
PS C:\> $env:PSModulePath
C:\Users\administrator\Documents\WindowsPowerShell\Modules;C:\Windows\system32\WindowsPowerShell\v1.0\Modules\
```

Za pomocą indeksów ze zmiennej *PSModulePath* można pobierać co najwyżej po jednym znaku, jak pokazano poniżej:

```
PS C:\> $env:psmodulePath[0]
C
PS C:\> $env:psmodulePath[1]
:
PS C:\> $env:psmodulePath[2]
\
PS C:\> $env:psmodulePath[3]
U
```

Pobieranie ścieżki do folderu z modułami użytkownika po jednym znaku byłoby trudne w najlepszym wypadku i ryzykowne w najgorszym. Jako że dane są w formacie łańcuchowym, na ścieżkach można pracować za pomocą metod łańcuchowych. Aby podzielić ten łańcuch na elementy wygodnej w użyciu tablicy, można użyć metody *split* z klasy *System.String*. Metodzie tej należy przekazać tylko jedną wartość — znak, według którego ma zostać podzielony łańcuch.

Jako że wartość znajdująca się w zmiennej `PSModulePath` jest łańcuchem, metody `split` można użyć bezpośrednio. Poniżej przedstawiono przykład zastosowania tej techniki:

```
PS C:\> $env:PSModulePath.Split(";")
C:\Users\administrator\Documents\WindowsPowerShell\Modules
C:\Windows\system32\WindowsPowerShell\v1.0\Modules\
```

Jak widać, pierwszy ze zwróconych łańcuchów reprezentuje ścieżkę do folderu na moduły użytkownika, a drugi — ścieżkę do systemowego folderu na moduły. Jako że metoda `split` przekształca łańcuch w tablicę, teraz można pobrać ścieżkę do folderu na moduły użytkownika za pomocą indeksu `[0]`. Zwróconej tablicy nie trzeba zapisywać w zmiennej pośredniej, jeśli się nie chce tego robić. Można bezpośrednio indeksować zwróconą tablicę. Gdyby jednak trzeba było zapisać tę tablicę w zmiennej pośredniej, kod wyglądałby mniej więcej tak:

```
PS C:\> $aryPaths = $env:PSModulePath.Split(";")
PS C:\> $aryPaths[0]
C:\Users\administrator\Documents\WindowsPowerShell\Modules
```

Ponieważ tablica jest dostępna od razu po wywołaniu metody `split`, ścieżkę do modułów użytkownika można pobrać bezpośrednio, jak pokazano poniżej:

```
PS C:\> $env:PSModulePath.Split(";")[0]
C:\Users\administrator\Documents\WindowsPowerShell\Modules
```

Sposób użycia zmiennej `$modulePath`

Ścieżkę, która zostanie użyta do zapisania modułu, zapiszemy w zmiennej `$modulePath`. Będzie ona zawierała ścieżkę do folderu na moduły użytkownika plus folder podrzędny o takiej samej nazwie jak nazwa modułu. Do tworzenia ścieżek należy używać polecenia `Join-Path`, zamiast łączyć łańcuchy ręcznie. Polecenie to sprawnie połączy dwie części, tworząc nową ścieżkę do folderu, jak pokazano poniżej:

```
$ModulePath = Join-Path -path $userPath '
    -childpath (Get-Item -path $name).basename
```

Konsola Windows PowerShell dodaje do klasy `System.IO.FileInfo` własność skryptową o nazwie `basename`. Ułatwia ona pobieranie nazwy pliku bez rozszerzenia. Przed pojawieniem się konsoli Windows PowerShell 2.0 do oddzielania rozszerzenia od nazwy pliku używano się metody `split` lub innych technik pracy na łańcuchach. Poniżej znajduje się przykład użycia własności `basename`:

```
PS C:\> (Get-Item -Path C:\fso\HelloWorld.psml).basename
HelloWorld
```

Ostatnim krokiem jest utworzenie podkatalogu, w którym zostanie zapisany moduł, i skopiowanie do niego plików tego modułu. Aby nie zapełnić całego okna konsoli informacjami zwróconymi przez polecenia `New-Item` i `Copy-Item`, wyniki są przekazywane do polecenia `Out-Null`, jak pokazano poniżej:

```
New-Item -path $modulePath -itemtype directory | Out-Null
Copy-Item -path $name -destination $ModulePath | Out-Null
```

W punkcie początkowym skryptu *Copy-Modules.ps1* wywoływana jest funkcja *Test-ModulePath* sprawdzająca, czy istnieje folder na moduły użytkownika. Następnie za pomocą polecenia *Get-ChildItem* z parametrem *-Recurse* pobierana jest lista wszystkich plików modułów w określonym folderze. Otrzymane obiekty *FileInfo* zostają przekazane do polecenia *ForEach-Object*. Następnie właściwość *FullName* każdego z tych obiektów zostaje przekazana do funkcji *Copy-Module*, jak pokazano poniżej:

```
Test-ModulePath
Get-ChildItem -Path C:\fso -Include *.psm1,*.psd1 -Recurse |
ForEach-Object { Copy-Module -name $_.FullName }
```

Poniżej znajduje się kompletny kod źródłowy skryptu *Copy-Modules.ps1*:

Copy-Modules.ps1

```
Function Get-OperatingSystemVersion
{
    (Get-WmiObject -Class Win32_OperatingSystem).Version
} #end Get-OperatingSystemVersion

Function Test-ModulePath
{
    $VistaPath = "$env:userProfile\documents\WindowsPowerShell\Modules"
    $XPPPath = "$env:Userprofile\my documents\WindowsPowerShell\Modules"
    if ([int](Get-OperatingSystemVersion).substring(0,1) -ge 6)
    {
        if(-not(Test-Path -path $VistaPath))
        {
            New-Item -Path $VistaPath -itemtype directory | Out-Null
        } #end if
    } #end if
    Else
    {
        if(-not(Test-Path -path $XPPPath))
        {
            New-Item -path $XPPPath -itemtype directory | Out-Null
        } #end if
    } #end else
} #end Test-ModulePath

Function Copy-Module([string]$name)
{
    $UserPath = $env:PSModulePath.split(";")[0]
    $ModulePath = Join-Path -path $UserPath '
        -childpath (Get-Item -path $name).basename
    New-Item -path $modulePath -itemtype directory | Out-Null
    Copy-Item -path $name -destination $ModulePath | Out-Null
}

# *** punkt początkowy skryptu ***
Test-ModulePath
Get-ChildItem -Path C:\fso -Include *.psm1,*.psd1 -Recurse |
ForEach-Object { Copy-Module -name $_.FullName }
```


UWAGA

Aby móc używać modułów utworzonych przez użytkownika, należy włączyć obsługę skryptów. Nie jest to konieczne w przypadku używania modułów systemowych. Ale żeby skrypt *Copy-Modules.ps1* zainstalował moduły w profilu użytkownika, obsługa skryptów jest niezbędna. Można ją włączyć za pomocą polecenia `Set-ExecutionPolicy`.

Tworzenie dysku modułu

Aby ułatwić sobie pracę z modułami, można utworzyć parę dysków Windows PowerShell przy użyciu dostawcy `filesystem`. Jako że folder z modułami znajduje się w miejscu trudno dostępnym z poziomu konsoli oraz zmienna `$PSModulePath` zwraca łańcuch zawierający ścieżkę zarówno do folderu z modułami użytkownika, jak i systemowego, dobrym pomysłem jest uproszczenie pracy z lokalizacją modułów.

Aby utworzyć dysk Windows PowerShell dla lokalizacji modułów użytkownika, należy użyć polecenia `New-PSDrive` i przekazać mu nazwę, np. `mymods`, użyć dostawcy `filesystem` oraz pobrać lokalizację główną ze zmiennej środowiskowej `$PSModulePath` za pomocą metody `split` z klasy łańcuchów platformy .NET. Ścieżka do folderu z modułami użytkownika jest pierwszym elementem zwróconej tablicy, jak widać poniżej:

```
PS C:\> New-PSDrive -Name mymods -PSProvider filesystem -Root ($env:PSModulePath
.Split(";")[0])
```

| Name | Used (GB) | Free (GB) | Provider | Root |
|--------|-----------|-----------|------------|---------------------------------|
| mymods | | 116.50 | FileSystem | C:\Users\administrator\Docum... |

Polecenie służące do utworzenia dysku Windows PowerShell dla lokalizacji modułów systemowych jest prawie takie samo jak dla lokalizacji modułów użytkownika. Różni się tylko nazwą, np. `sysmods`, oraz tym, że z tablicy zwróconej przez metodę `split` ze zmiennej `$PSModulePath` trzeba wybrać drugi element. Poniżej pokazano to polecenie:

```
PS C:\> New-PSDrive -Name sysmods -PSProvider filesystem -Root ($env:PSModulePath.Split(";")[1])
```

| Name | Used (GB) | Free (GB) | Provider | Root |
|---------|-----------|-----------|------------|---------------------------------|
| sysmods | | 116.50 | FileSystem | C:\Windows\system32\WindowsP... |

Można też napisać skrypt tworzący dyski dla obu tych lokalizacji naraz. Najpierw należy utworzyć tablicę nazw planowanych dysków Windows PowerShell. Następnie należy ją przejrzeć za pomocą instrukcji `for` i wywołać polecenie `New-PSDrive`. Jako że polecenia są wywoływane w skrypcie, nowo utworzone dyski Windows PowerShell będą dostępne w zakresie skryptowym. Po zakończeniu skryptu wszystko, co było z nim związane, znika. To by znaczyło, że dyski Windows PowerShell staną się niedostępne, gdy skrypt skończy działanie — co podważa w ogóle sens ich tworzenia. Rozwiązaniem tego problemu jest utworzenie dysków Windows PowerShell w zakresie globalnym, tak aby były dostępne w konsoli także po zakończeniu działania skryptu. Aby wyeliminować niepotrzebne potwierdzenia podczas procesu, wyniki należy przekazać do polecenia `Out-Null`.

W skrypcie *New-ModulesDrive.ps1* tworzymy kolejną funkcję, która wyświetla globalne dyski Windows PowerShell dostawcy `filesystem`. Po uruchomieniu skrypt ten wywołuje funkcję `New-ModuleDrives`, a po niej funkcję `Get-FileSystemDrives`. Poniżej znajduje się kompletny kod źródłowy funkcji `New-ModulesDrive`.

New-ModulesDrive.ps1

```

Function New-ModuleDrives
{
<#
    .SYNOPSIS
    Tworzy dwa dyski PSDrive: myMods i sysMods
    .EXAMPLE
    New-ModuleDrives
    Tworzy dwa dyski PSDrive: myMods i sysMods. Reprezentują one
    odpowiednio foldery z modułami użytkownika i systemowy.
#>
    $driveNames = "myMods", "sysMods"

    For($i = 0 ; $i -le 1 ; $i++)
    {
        New-PSDrive -name $driveNames[$i] -PSProvider filesystem '
        -Root ($env:PSModulePath.split(";")[$i]) -scope Global |
        Out-Null
    } #end For
} #end New-ModuleDrives

Function Get-FileSystemDrives
{
<#
    .SYNOPSIS
    Wyświetla globalne dyski PowerShell używające dostawcy Filesystem.
    .EXAMPLE
    Get-FileSystemDrives
    Wyświetla globalne dyski PowerShell używające dostawcy Filesystem.
#>
    Get-PSDrive -PSProvider FileSystem -scope Global
} #end Get-FileSystemDrives

# *** punkt początkowy skryptu ***
New-ModuleDrives
Get-FileSystemDrives

```

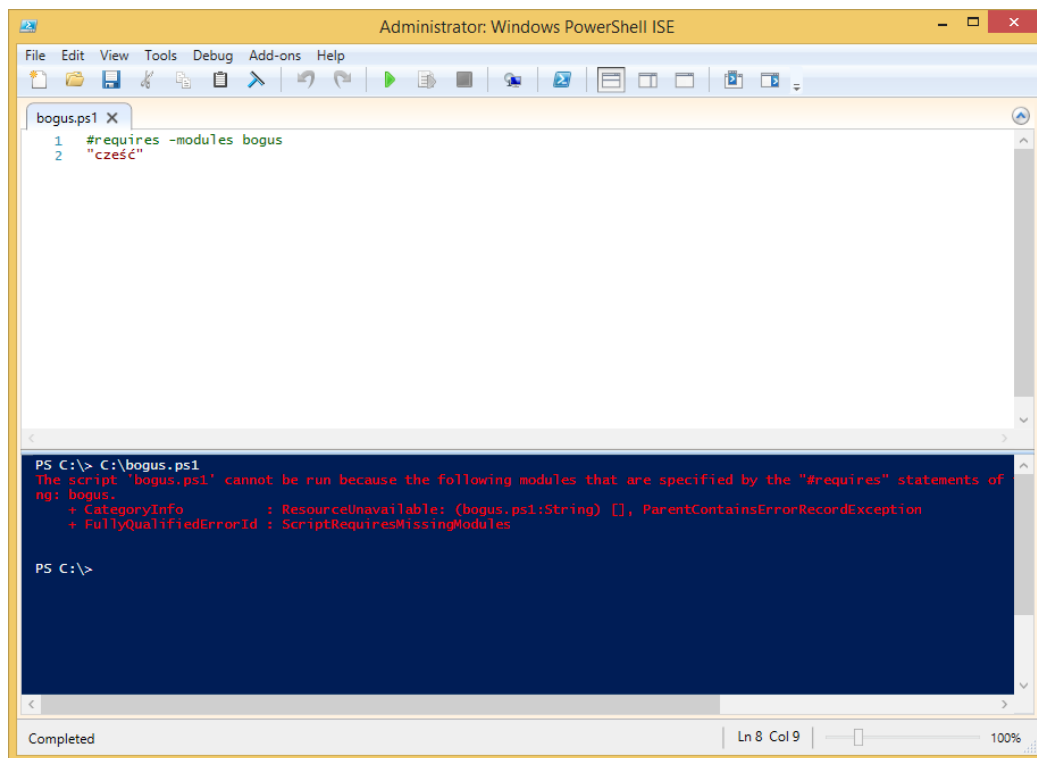
W podrozdziale tym zostały opisane techniki instalowania modułów. Zanim zainstaluje się moduł, należy utworzyć specjalny folder na moduły w profilu użytkownika. Utworzyliśmy skrypt wykonujący tę czynność. Poznaliśmy też zmienną `$modulePath`. Na końcu podrozdziału przeanalizowaliśmy skrypt tworzący dysk Windows PowerShell ułatwiający dostęp do zainstalowanych modułów.

Sprawdzanie zależności modułów

Jedną z wad używania modułów jest to, że tworzy się zależność od zewnętrznego kodu, co znaczy, że dany moduł musi być obecny w systemie, aby można było wykonać używające go skrypty. Jeśli ma się kontrolę nad środowiskiem, taka dodatkowa zewnętrzna niezależność nie jest niczym złym. Ale jeśli nie ma się takiej kontroli, to sytuacja staje się bardzo trudna.

Ze względu na ryzyko występowania problemów w konsoli Windows PowerShell 3.0 dodano instrukcję `#requires`. Sprawdza ona wersję konsoli, moduły, przystawki, a nawet ich wersje. Niestety instrukcji tej można używać tylko w skryptach. Jest niedostępna dla funkcji, poleceń

cmdlet i przystawek. Na rysunku 10.3 pokazano przykład użycia instrukcji `#requires` do sprawdzenia przed wykonaniem skryptu, czy dostępny jest określony moduł. Skrypt ten wymaga obecności modułu o nazwie `bogus`, którego nie ma w tym systemie. A skoro go nie ma, występuje błąd.



RYSUNEK 10.3. Użycie instrukcji `#requires` w celu uniemożliwienia wykonania skryptu, gdy brakuje potrzebnego modułu

Jako że instrukcji `#requires` nie można używać w funkcjach, to do sprawdzenia, czy dany moduł istnieje lub jest już załadowany, można użyć funkcji `Get-MyModule`. Poniżej znajduje się jej kod źródłowy:

Get-MyModule.ps1

```
Function Get-MyModule
{
    Param([string]$name)
    if(-not(Get-Module -name $name))
    {
        if(Get-Module -ListAvailable |
            Where-Object { $_.name -eq $name })
        {
            Import-Module -Name $name
            $true
        } #end jeśli moduł jest dostępny, to następuje jego zaimportowanie
    } else { $false } #moduł jest niedostępny
    } #end jeśli nie moduł
} else { $true } #moduł już jest załadowany
```

```

} #end function get-MyModule

get-mymodule -name "bitsTransfer"

```

Funkcja `Get-MyModule` przyjmuje pojedynczy łańcuch reprezentujący nazwę modułu, który ma zostać sprawdzony. Za pomocą instrukcji `if` sprawdzamy, czy moduł jest już załadowany. Jeśli nie, to za pomocą polecenia `Get-Module` sprawdzamy, czy interesujący nas moduł jest w ogóle dostępny w systemie. Jeśli tak jest, to go ładujemy.

Jeżeli moduł jest już załadowany w bieżącej sesji Windows PowerShell, funkcja `Get-MyModule` zwraca `$true` do kodu, który ją wywołał. Przyjrzymy się trochę dokładniej tej funkcji, aby sprawdzić, jak działa.

Po pierwsze: za pomocą instrukcji `if` sprawdzamy, czy moduł nie jest załadowany w bieżącej sesji. W tym celu używamy operatora `-not`. Za pomocą polecenia `Get-Module` szukamy potrzebnego modułu według nazwy. Poniżej znajduje się opisywany fragment kodu:

```

Function Get-MyModule
{
    Param([string]$name)
    if(-not(Get-Module -name $name))
    {

```

Do pobrania listy modułów zainstalowanych w systemie można użyć polecenia `Get-Module` z parametrem `-ListAvailable`. Niestety nie da się przefiltrować wyników, co zmusza nas do przekazania ich do polecenia `Where-Object`, za pomocą którego sprawdzamy, czy interesujący nas moduł znajduje się w systemie. Jeśli tak, to funkcja importuje go za pomocą polecenia `Import-Module` i zwraca `$true` do kodu, który ją wywołał. Poniżej znajduje się opisywana część skryptu:

```

if(Get-Module -ListAvailable |
    Where-Object { $_.name -eq $name })
{
    Import-Module -Name $name
    $true
} #end jeśli moduł jest dostępny, to następuje jego zaimportowanie

```

Dwie ostatnie czynności w tej funkcji dotyczą postępowania w dwóch pozostałych przypadkach. Jeśli moduł jest niedostępny, to polecenie `Where-Object` nic nie znajdzie. Wówczas następuje wykonanie klauzuli `else` i zwrócenie `$false` do kodu wywołującego. Jeżeli moduł jest już załadowany, to druga klauzula `else` zwraca `$true`. Poniżej znajduje się opisywana część skryptu:

```

else { $false } #moduł jest niedostępny
    } #end jeśli nie moduł
    else { $true } #moduł już jest załadowany

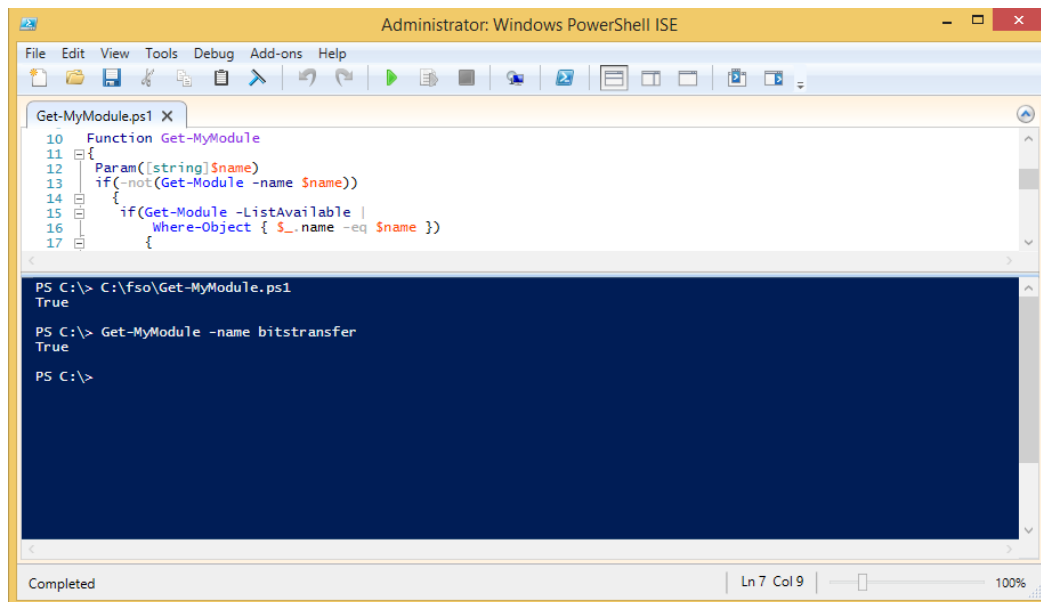
} #end function get-MyModule

```

Najprostszym sposobem użycia funkcji `Get-MyModule` jest wywołanie jej z nazwą modułu. Przykład tego pokazano w ostatnim wierszu skryptu *Get-MyModule.ps1*:

```
get-mymodule -name "bitsTransfer"
```

Tak wywołana funkcja `Get-MyModule` załaduje moduł `bitstransfer`, pod warunkiem że będzie dostępny w systemie i nie będzie jeszcze załadowany. Jeżeli moduł będzie już załadowany lub zostanie załadowany przez funkcję, skrypt zwróci `$true`. Jeżeli moduł będzie niedostępny, nastąpi zwrot wartości `$false`. Na rysunku 10.4 pokazano przykład użycia funkcji `Get-MyModule`.



RYСУNEK 10.4. Użycie funkcji `Get-MyModule` w celu sprawdzenia, czy wymagany moduł istnieje, przed podjęciem próby jego załadowania

Lepszym zastosowaniem funkcji `Get-MyModule` jest użycie jej do sprawdzania, czy funkcja używająca określonego modułu spełnia warunki wstępne. Może to wyglądać mniej więcej tak:

```

If(Get-MyModule -name "bitsTransfer") { kod dotyczący potrzebnego modułu }
ELSE { "Moduł nie jest zainstalowany w tym systemie." ; exit }
  
```

Zapiski praktyka

Marc Carter, informatyk

Prezes grupy Corpus Christi PowerShell Users Group

Większość tego, co robię w konsoli Windows PowerShell, wcześniej czy później kończy jako skrypt. Początkowo tworzę tylko jednolinijkowe skrypty, aby coś sobie rozjaśnić, a z czasem przekształcam je w kombinacje zapytań, instrukcji i komentarzy, które zapisuję gdzieś na wypadek, gdybym został wezwany do pomocy przy likwidowaniu głodu na świecie, co zdarza się średnio raz na tydzień. Moim największym przyjacielem jest zintegrowane środowisko skryptowe (ang. *Integrated Scripting Environment* — ISE). Mogę w nim kompilować i testować fragmenty kodu podczas pracy, wykonywać jednowierszowe skrypty oraz wygodnie

kopiować wyniki. Jeśli chodzi o pisanie modułów... każdy powinien mieć jakieś aspiracje i dla mnie takim celem jest nauczenie się pisania zaawansowanych funkcji i modułów. Gdy mamy chwile wytchnienia w pracy, doskonale swoje umiejętności skryptowe, przekształcając najbardziej przydatne skrypty w taki sposób, aby móc przekazać je do używania swoim współpracownikom. W międzyczasie zdobywam cenną wiedzę na temat pisania lepszych i bardziej efektywnych skryptów. Podczas rozwiązywania problemu głodu na świecie nie przejmuję się takimi sprawami jak obsługa czy przechwytywanie błędów, ale gdy w wolnym czasie doskonale swoje skrypty, zamieniając je w zaawansowane funkcje i moduły, przy okazji wiele uczę się na te i inne tematy. Z czasem coraz częściej stosuję te techniki w swoich skryptach.

Używanie modułów pochodzących z udziałów

Modułów pochodzących z udziałów używa się tak samo jak modułów zapisanych w lokalizacjach domyślnych. Jeśli moduł znajduje się w folderze `%windir%\System32\WindowsPowerShell\v1.0\Modules`, to jest dostępny dla wszystkich użytkowników. Natomiast moduły znajdujące się w folderze `%ProfilUzytkownika%\My Documents\WindowsPowerShell\Modules` są dostępne tylko dla jednego użytkownika. Zaletą umieszczania modułów w profilu użytkownika jest to, że użytkownik ten automatycznie otrzymuje uprawnienia do przeprowadzania instalacji, podczas gdy do pracy w lokalizacji systemowej od systemu Windows 7 potrzebne są uprawnienia administratora.

Skoro mowa o instalowaniu modułów Windows PowerShell, w wielu przypadkach sprowadza się ono do wklejenia pliku `*.psm1` w domyślnym folderze użytkownika. Ważne jest tylko, aby w folderze `\Modules` utworzyć folder o takiej samej nazwie jak nazwa pliku samego modułu. Do instalacji modułu w lokalnym komputerze można używać skryptu `Copy-Modules.ps1`, który upraszcza proces tworzenia i nazywania folderów.

Przy kopiowaniu modułu Windows PowerShell do współdzielonej lokalizacji sieciowej obowiązują takie same zasady, tzn. folder zawierający moduł musi mieć taką samą nazwę, jaką ma ten moduł.

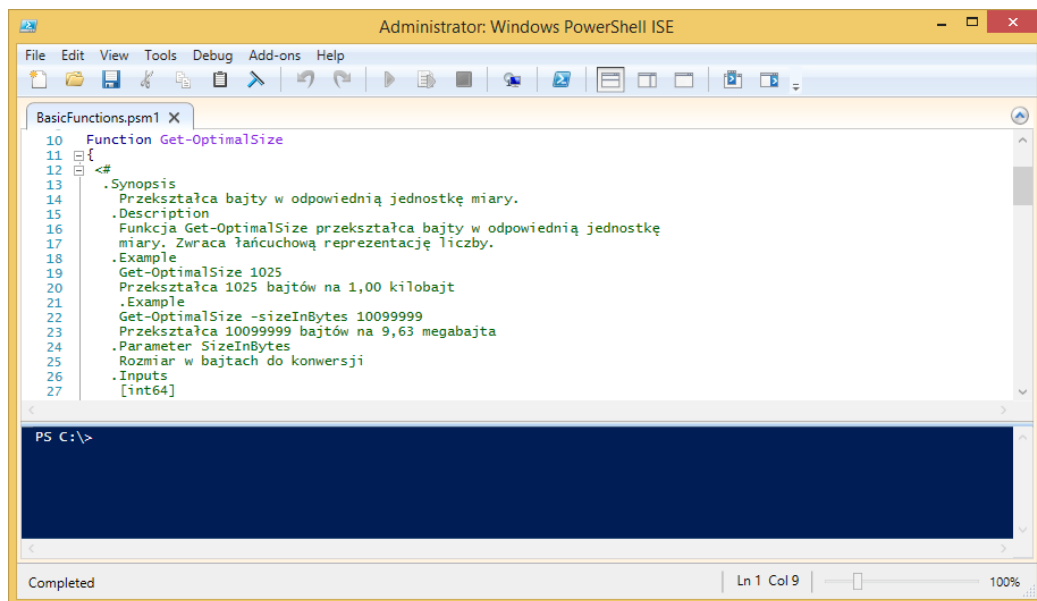
Należy pamiętać o kilku rzeczach. Po pierwsze: moduł Windows PowerShell jest w istocie skryptem (w naszym konkretnym przypadku), więc aby można go było uruchomić, musi być odpowiednio ustawiona reguła wykonywania skryptów. Jeśli reguła ta będzie ustawiona na poziom ograniczony (`restricted`), to zamiast wykonania skryptu zostanie zgłoszony błąd (nawet jeśli użytkownik będzie administratorem). Na szczęście powiadomienie o błędzie informuje o tym. Nawet jeśli reguła wykonywania jest ustawiona na `restricted`, zawsze można uruchomić skrypt (lub moduł), uruchamiając konsolę Windows PowerShell z opcją `bypass`. Poniżej znajduje się odpowiednie polecenie:

```
powershell -executionpolicy bypass
```

Jedną z największych zalet używania współdzielonych modułów jest możliwość scentralizowania profili Windows PowerShell dla użytkowników podłączonych do sieci. W tym celu profil na lokalnym komputerze mógłby po prostu importować współdzielony moduł. Dzięki temu aktualizacja modułu w wielu profilach wymagałaby zmiany tylko w jednym miejscu.

Tworzenie modułu

Pewnie chcesz w końcu utworzyć własny moduł. Możesz do tego użyć konsoli Windows PowerShell ISE. Najłatwiej użyć już wcześniej napisanych funkcji. Tworzenie modułu najlepiej zacząć od znalezienia funkcji, które chce się w nim zapisać. Można je skopiować wprost do okna Windows PowerShell ISE, jak pokazano na rysunku 10.5.



RYСУNEK 10.5. W konsoli Windows PowerShell ISE moduły można tworzyć przez skopiowanie i wklejenie istniejących funkcji do nowego pliku

Po skopiowaniu funkcji do nowego modułu należy go zapisać jako plik z rozszerzeniem *.psm1*. Poniżej znajduje się treść modułu *BasicFunctions.psm1*.

BasicFunctions.psm1

```

Function Get-OptimalSize
{
    <#
    .Synopsis
    Przekształca bajty w odpowiednią jednostkę miary.
    .Description
    Funkcja Get-OptimalSize przekształca bajty w odpowiednią jednostkę
    miary. Zwraca łańcuchową reprezentację liczby.
    .Example
    Get-OptimalSize 1025
    Przekształca 1025 bajtów na 1,00 kilobajt
    .Example
    Get-OptimalSize -sizeInBytes 10099999
    Przekształca 10099999 bajtów na 9,63 megabajta
    .Parameter SizeInBytes

```

```

Rozmiar w bajtach do konwersji
.Inputs
[int64]
.OutPuts
[string]
.Notes
NAZWA: Get-OptimalSize
AUTOR: Ed Wilson
OSTATNIA EDYCJA: 30.6.2012
SŁOWA KLUCZOWE: techniki pisania skryptów, moduły
.Link
Http://www.ScriptingGuys.com
#Requires -Version 2.0
#>
[CmdletBinding()]
param(
    [Parameter(Mandatory = $true, Position = 0, valueFromPipeline=$true)]
    [int64]
    $sizeInBytes
) #end param
Switch ($sizeInBytes)
{
    {$sizeInBytes -ge 1TB} {"{0:n2}" -f ($sizeInBytes/1TB) + " terabajtów";break}
    {$sizeInBytes -ge 1GB} {"{0:n2}" -f ($sizeInBytes/1GB) + " gigabajtów";break}
    {$sizeInBytes -ge 1MB} {"{0:n2}" -f ($sizeInBytes/1MB) + " megabajtów";break}
    {$sizeInBytes -ge 1KB} {"{0:n2}" -f ($sizeInBytes/1KB) + " kilobajtów";break}
    Default { "{0:n2}" -f $sizeInBytes + " bajtów" }
} #end switch
$sizeInBytes = $null
} #end Function Get-OptimalSize

Function Get-ComputerInfo
{
    <#
    .Synopsis
    Pobiera podstawowe informacje o komputerze.
    .Description
    Polecenie Get-ComputerInfo pobiera podstawowe informacje, takie jak nazwa komputera, nazwa domeny oraz aktualnie zalogowany użytkownik na lokalnym lub zdalnym komputerze.
    .Example
    Get-ComputerInfo
    Zwraca nazwę komputera, nazwę domeny i aktualnie zalogowanego użytkownika z komputera lokalnego.
    .Example
    Get-ComputerInfo -computer berlin
    Zwraca nazwę komputera, nazwę domeny oraz aktualnie zalogowanego użytkownika ze zdalnego komputera o nazwie berlin.
    .Parameter Computer
    Nazwa zdalnego komputera, z którego mają zostać pobrane informacje
    .Inputs
    [string]
    .OutPuts
    [object]
    .Notes
    NAZWA: Get-ComputerInfo
    AUTOR: Ed Wilson

```


OSTATNIA EDYCJA: 30.6.2012

SŁOWA KLUCZOWE: zarządzanie pulpitem, podstawowe informacje

.Link

[Http://www.ScriptingGuys.com](http://www.ScriptingGuys.com)

#Requires -Version 2.0

#>

```
Param([string]$computer=$env:COMPUTERNAME)
$wmi = Get-WmiObject -Class win32_computersystem -ComputerName $computer
$pcinfo = New-Object psobject -Property @{ "host" = $wmi.DNSHostname
    "domain" = $wmi.Domain
    "user" = $wmi.Username }
$pcInfo
} #end function Get-ComputerInfo
```

Jeśli chcemy kontrolować, co może być eksportowane z modułu, możemy utworzyć manifest. Jeśli ktoś umieści w module funkcję, których na pewno będzie trzeba używać razem, to nie musi tworzyć manifestu. W module *BasicFunctions.psm1* znajdują się dwie funkcje: jedna konwertuje liczby bajtów na bardziej zrozumiałe jednostki miary, a druga zwraca podstawowe informacje o komputerze.

Funkcja *Get-ComputerInfo* zwraca niestandardowy obiekt zawierający informacje dotyczące użytkownika, komputera oraz domeny komputera. Po utworzeniu i zapisaniu modułu należy go zainstalować. Można to zrobić ręcznie, przechodząc do folderu modułów, tworząc folder na moduł i wklejając do niego kopię pliku tego modułu. Ale ja wolę do tego celu używać opisanego wcześniej skryptu *Copy-Modules.ps1*.

Po skopiowaniu modułu do odpowiedniego folderu (czyli jego zainstalowaniu) można go zaimportować do bieżącej sesji Windows PowerShell za pomocą polecenia *Import-Module*. Jeśli nie pamiętasz nazwy modułu, możesz użyć polecenia *Get-Module* z przełącznikiem *-ListAvailable*, jak pokazano poniżej:

```
PS C:\> Get-Module -ListAvailable
```

Directory: C:\Users\administrator\Documents\WindowsPowerShell\Modules

| ModuleType | Name | ExportedCommands |
|------------|--------------------|---------------------------------------|
| Script | BasicFunctions | {Get-OptimalSize, Get-ComputerInfo} |
| Script | ConversionModuleV6 | {ConvertTo-MetersPerSecond, Conver... |
| Script | HelloUser | hello-user |

Directory: C:\Windows\system32\WindowsPowerShell\v1.0\Modules

| ModuleType | Name | ExportedCommands |
|------------|------------------------------|---------------------------------------|
| Manifest | AppLocker | {Get-AppLockerFileInformation, Get... |
| Manifest | Appx | {Add-AppxPackage, Get-AppxPackage,... |
| Manifest | BitLocker | {Unlock-BitLocker, Suspend-BitLock... |
| Manifest | BitsTransfer | {Add-BitsFile, Complete-BitsTransf... |
| Manifest | BranchCache | {Add-BCDataCacheExtension, Clear-B... |
| Manifest | CimCmdlets | {Get-CimAssociatedInstance, Get-Ci... |
| Manifest | DirectAccessClientComponents | {Disable-DAManualEntryPointSelecti... |
| Script | Dism | {Add-AppxProvisionedPackage, Add-W... |
| Manifest | DnsClient | {Resolve-DnsName, Clear-DnsClientC... |
| Manifest | International | {Get-WinDefaultInputMethodOverride... |
| Manifest | iSCSI | {Get-IscsiTargetPortal, New-IscsiT... |

| | | |
|----------|----------------------------------|---------------------------------------|
| Script | ISE | {New-IseSnippet, Import-IseSnippet... |
| Manifest | Kds | {Add-KdsRootKey, Get-KdsRootKey, T... |
| Manifest | Microsoft.PowerShell.Diagnostics | {Get-WinEvent, Get-Counter, Import... |
| Manifest | Microsoft.PowerShell.Host | {Start-Transcript, Stop-Transcript} |
| Manifest | Microsoft.PowerShell.Management | {Add-Content, Clear-Content, Clear... |
| Manifest | Microsoft.PowerShell.Security | {Get-Acl, Set-Acl, Get-PfxCertific... |
| Manifest | Microsoft.PowerShell.Utility | {Format-List, Format-Custom, Forma... |
| Manifest | Microsoft.WSMan.Management | {Disable-WSManCredSSP, Enable-WSMa... |
| Manifest | MMAgent | {Disable-MMAgent, Enable-MMAgent, ... |
| Manifest | MsDtc | {New-DtcDiagnosticTransaction, Com... |
| Manifest | NetAdapter | {Disable-NetAdapter, Disable-NetAd... |
| Manifest | NetConnection | {Get-NetConnectionProfile, Set-Net... |
| Manifest | NetLbfo | {Add-NetLbfoTeamMember, Add-NetLbF... |
| Manifest | NetQos | {Get-NetQosPolicy, Set-NetQosPolic... |
| Manifest | NetSecurity | {Get-DAPolicyChange, New-NetIPsecA... |
| Manifest | NetSwitchTeam | {New-NetSwitchTeam, Remove-NetSwit... |
| Manifest | NetTCPIP | {Get-NetIPAddress, Get-NetIPInterf... |
| Manifest | NetworkConnectivityStatus | {Get-DAConnectionStatus, Get-NCSIP... |
| Manifest | NetworkTransition | {Add-NetIPHttpsCertBinding, Disab... |
| Manifest | PKI | {Add-CertificateEnrollmentPolicySe... |
| Manifest | PrintManagement | {Add-Printer, Add-PrinterDriver, A... |
| Script | PSDiagnostics | {Disable-PSTrace, Disable-PSWSManC... |
| Binary | PSScheduledJob | {New-JobTrigger, Add-JobTrigger, R... |
| Manifest | PSWorkflow | {New-PSWorkflowExecutionOption, Ne... |
| Manifest | PSWorkflowUtility | {Invoke-AsWorkflow} |
| Manifest | ScheduledTasks | {Get-ScheduledTask, Set-ScheduledT... |
| Manifest | SecureBoot | {Confirm-SecureBootUEFI, Set-Secur... |
| Manifest | SmbShare | {Get-SmbShare, Remove-SmbShare, Se... |
| Manifest | SmbWitness | {Get-SmbWitnessClient, Move-SmbWit... |
| Manifest | Storage | {Add-InitiatorIdToMaskingSet, Add... |
| Manifest | TroubleshootingPack | {Get-TroubleshootingPack, Invoke-T... |
| Manifest | TrustedPlatformModule | {Get-Tpm, Initialize-Tpm, Clear-Tp... |
| Manifest | VpnClient | {Add-VpnConnection, Set-VpnConnect... |
| Manifest | Wdac | {Get-OdbcDriver, Set-OdbcDriver, G... |
| Manifest | WindowsDeveloperLicense | {Get-WindowsDeveloperLicense, Show... |
| Script | WindowsErrorReporting | {Enable-WindowsErrorReporting, Dis... |

Po zaimportowaniu modułu za pomocą polecenia `Get-Command` z parametrem `-module` można sprawdzić, jakie polecenia eksportuje, jak pokazano poniżej:

```

PS C:\> Import-Module basicfunctions
PS C:\> Get-Command -Module basic*

```

| CommandType | Name | ModuleName |
|-------------|------------------|----------------|
| ----- | ---- | ----- |
| Function | Get-ComputerInfo | basicfunctions |
| Function | Get-OptimalSize | basicfunctions |

Funkcji dodanych z modułu można używać bezpośrednio w wierszu poleceń Windows PowerShell. Poniżej pokazano przykład użycia funkcji `Get-ComputerInfo`:

```

PS C:\> Get-ComputerInfo
host                                domain                                user
----                                -
mred1                              NWTraders.Com
NWTRADERS\ed
PS C:\> (Get-ComputerInfo).user

```

```

NWTRADERS\ed
PS C:\> (Get-ComputerInfo).host
mred1
PS C:\> Get-ComputerInfo -computer win8-pc | Format-Table -AutoSize
host      domain      user
----      -
win8-PC   NWTraders.Com  NWTRADERS\Administrator
PS C:\>

```

Dzięki zastosowaniu znaczników pomocy w definicjach funkcji teraz można wyświetlić informacje pomocnicze za pomocą polecenia `Get-Help`. Pod tym względem funkcje z modułów nie różnią się od poleceń cmdlet konsoli Windows PowerShell. Dotyczy to także rozwijania za pomocą klawisza *Tab*.

```
PS C:\> Get-Help Get-ComputerInfo
```

NAME

Get-ComputerInfo

SYNOPSIS

Pobiera podstawowe informacje o komputerze.

SYNTAX

```
Get-ComputerInfo [[-computer] <String>] [<CommonParameters>]
```

DESCRIPTION

Polecenie `Get-ComputerInfo` pobiera podstawowe informacje, takie jak nazwa komputera, nazwa domeny oraz aktualnie zalogowany użytkownik na lokalnym lub zdalnym komputerze.

RELATED LINKS

[Http://www.ScriptingGuys.com](http://www.ScriptingGuys.com)
#Requires -Version 2.0

REMARKS

To see the examples, type: "get-help Get-ComputerInfo -examples".
For more information, type: "get-help Get-ComputerInfo -detailed".
For technical information, type: "get-help Get-ComputerInfo -full".
For online help, type: "get-help Get-ComputerInfo -online"

```
PS C:\> Get-Help Get-ComputerInfo -Examples
```

NAME

Get-ComputerInfo

SYNOPSIS

Pobiera podstawowe informacje o komputerze.

----- EXAMPLE 1 -----

```
C:\PS>Get-ComputerInfo
```

Zwraca nazwę komputera, nazwę domeny i aktualnie zalogowanego użytkownika z komputera lokalnego.

----- EXAMPLE 2 -----

```
C:\PS>Get-ComputerInfo -computer berlin
```

Zwraca nazwę komputera, nazwę domeny oraz aktualnie zalogowanego użytkownika ze zdalnego komputera o nazwie berlin.

Funkcja `Get-OptimalSize` przyjmuje nawet dane z potoku, jak pokazano poniżej:

```
PS C:\> (Get-WmiObject win32_volume -Filter "driveletter = 'c:']").freespace
26513960960
PS C:\> (Get-WmiObject win32_volume -Filter "driveletter = 'c:']").freespace | Get OptimalSize
24,69 gigabajtów
PS C:\>
```

Zapiski praktyka

MVP Windows PowerShell

Boe Prox

Starszy administrator systemów Windows

W większości organizacji rutynowe czynności związane z instalacją poprawek muszą być wykonywane w ściśle określonych ramach czasowych. W zależności od rozmiaru środowiska może to zająć kilka godzin i wymagać pracy kilku osób. Najczęściej praca ta polega na logowaniu się do każdego serwera osobno i ręcznym instalowaniu poprawek.

Ale od czego jest Windows PowerShell? Zastosowania tej konsoli nie kończą się na podstawowych poleceniach i jednowierszowych skryptach. Można w niej wykorzystywać zawartość biblioteki .Net i tworzyć narzędzia (wiersza poleceń lub z graficznym interfejsem użytkownika), z których mogą korzystać bez wysiłku także inni użytkownicy.

Narzędzie *PoshPAIG* (ang. *PowerShell Patch Audit/Install GUI*), dostępne pod adresem <https://PoshPAIG.codeplex.com>, utworzyłem po to, by skrócić ilość czasu potrzebnego administratorom mającym ograniczone zasoby czasu na zainstalowanie poprawek w ich środowiskach. Jako że niektórych mógłby odstraszyć wiersz poleceń, zaimplementowałem graficzny interfejs użytkownika, który jest łatwy w obsłudze dla każdego. Kod nie był łatwy do napisania, chociaż cele były proste, aczkolwiek ciągle się zmieniają zgodnie z zapotrzebowaniem społeczności.

Implementacja narzędzia jest mieszanką kodu XAML (dotyczącego interfejsu użytkownika), .Net (do budowy przestrzeni roboczych do obsługi wielowątkowości), obiektów COM (do przeprowadzania audytów poprawek), zewnętrznych plików wykonywalnych (zdalna instalacja poprawek) i oczywiście kodu Windows PowerShell, który pozwolił to wszystko połączyć i wykonać różne zadania za pomocą poleceń.

Praca nad tym projektem była prawdziwym wyzwaniem pod wieloma względami. Trzeba było na przykład rozwiązać problem wykonywania asynchronicznych operacji, aby można było aktualizować wiele systemów naraz. Kłopoty sprawiały nawet niektóre — zdawałoby się

oczywiste — czynności, takie jak sortowanie kolumn. Tę na pozór prostą operację trzeba było zakodować w interfejsie użytkownika, aby użytkownik mógł sortować wiersze, klikając wybrany nagłówek kolumny.

W portalach takich jak MSDN znalazłem szczegółowe informacje na temat klas .NET i języka XAML, które były pomocne w pisaniu wydajnego kodu.

Wynik końcowy składa się z 3000 wierszy kodu i wielu skryptów obsługujących różne funkcje narzędzia, takie jak na przykład:

- audyt poprawek w systemach,
- instalowanie poprawek w systemach,
- ponowne uruchamianie systemów,
- generowanie raportów na temat poprawek (skontrolowanych i zainstalowanych).

W efekcie powstało narzędzie z graficznym interfejsem użytkownika, do budowy którego użyto rozmaitych technik, języków i elementów interfejsu użytkownika. Za jego pomocą każdy może wykonać aktualizację wielu systemów, zebrać informacje o poprawkach oraz zainstalować poprawki. Wszystkie te czynności nie tylko da się wykonać, ale wykonuje się szybko i sprawnie, co jest ważne zwłaszcza w dużych środowiskach.

Dodatkowe źródła informacji

- Centrum skryptów portalu TechNet pod adresem <http://www.microsoft.com/technet/scriptcenter> zawiera wiele przykładów skryptów Windows PowerShell.
- Wszystkie skrypty opisane w tym rozdziale są dostępne do pobrania w repozytorium centrum skryptów portalu TechNet pod adresem <http://gallery.technet.microsoft.com/scriptcenter/PowerShell-40-Best-d9e16039>.

Rozdział 11

Obsługa wejścia i wyjścia

- Wybór najlepszej metody pobierania danych
- Monitowanie o informacje
- Wybór najlepszej metody zwracania danych
- Dodatkowe źródła informacji

Niewiele jest skryptów, które nie pobierają ani nie zwracają danych. Są to najczęściej skrypty wykonujące serie odpowiednio skonfigurowanych poleceń. Wiele skryptów pisanych przez informatyków wymaga podania danych na wejściu lub zwraca dane na wyjściu, a większość robi jedno i drugie. Aby skrypt był przydatny, zazwyczaj musi zwracać jakieś informacje.

Sposób pobierania danych wejściowych i zwracania wyników to kwestia, którą należy rozwiązać na etapie projektowania skryptu, a więc leżąca w gestii programisty. Najczęściej dane są pobierane w następujące sposoby:

- odczyt z wiersza poleceń,
- odczyt z pliku tekstowego,
- odczyt z bazy danych,
- odczyt z arkusza kalkulacyjnego,
- odczyt z rejestru,
- odczyt z usług domenowych Active Directory.

Często sposób zwracania danych jest podobny do sposobu ich pobierania, chociaż nie jest to obowiązująca reguła. Programista nie powinien ograniczać się tylko do jednego modelu projektowania. Zastanówmy się nad następującymi przypadkami:

- Skrypt pobiera dane z pliku tekstowego i wyświetla je na ekranie.
- Skrypt pobiera dane z bazy danych i wyniki również zapisuje w bazie danych, ale dodatkowo wyświetla potwierdzenie wykonania operacji na ekranie.
- Skrypt pobiera dane z wiersza poleceń i zapisuje wyniki w rejestrze.
- Skrypt pobiera dane z arkusza kalkulacyjnego, zapisuje wyniki w bazie danych, rejestruje informacje diagnostyczne w pliku tekstowym oraz tworzy zdarzenie w dzienniku zdarzeń zawierające kod zakończenia działania.
- Skrypt pobiera dane z wiersza poleceń i zapisuje wyniki w pliku tekstowym, a dodatkowo wyświetla te same dane na ekranie.

Jest wiele możliwości i kombinacji technik wejścia i wyjścia. Wybór najlepszej metody pobierania i zwracania danych nie zawsze jest oczywisty i często zależy od czynników zewnętrznych, takich jak ograniczenia sieci, łatwość obsługi, szybkość itd. Zawsze powinno się wybierać taką metodę pobierania danych, która nie przeszkadza w używaniu skryptu zgodnie z przeznaczeniem. W następnym podrozdziale znajduje się opis zalet i wad różnych metod pobierania danych i zwracania wyników.

Wybór najlepszej metody pobierania danych

Wybór najlepszej metody pobierania danych nie zawsze jest prosty. Jeśli chodzi o najlepsze praktyki, to można odnieść wrażenie, że ostateczna decyzja zawsze jest jakimś kompromisem. Mogłoby się wydawać, że wybór arkusza kalkulacyjnego Microsoft Office Excel to najlepszy wybór, ponieważ jest on dostępny i łatwy w użyciu, ale ta łatwość użycia idzie w parze ze złożonością skryptu. Ktoś może powiedzieć, że najlepszy jest plik tekstowy, ponieważ jego zawartość można wygodnie wczytać za pomocą polecenia `Get-Content`. A jednak użycie plików tekstowych powoduje trudności konserwacyjne, których niektórzy woleliby uniknąć. Jeśli łatwość konserwacji jest priorytetem, można zdecydować się na wczytywanie danych z Active Directory, ponieważ dzięki temu wiadomo, że wczytywane dane zawsze będą aktualne. Jednak ta metoda komplikuje skrypt i nie działa bez dostępu do Active Directory. Dlatego właśnie bez względu na to, na jaką metodę się zdecydujemy, nasza decyzja zawsze będzie kompromisem między użytecznością, zrozumiałością, łatwością konserwacji a łatwością zarządzania. Na początek przeanalizujemy najprostszą technikę pobierania danych do skryptu — z wiersza poleceń.

Wczytywanie danych z wiersza poleceń

Wiersz poleceń jest klasycznym źródłem danych wejściowych dla skryptów. Jego zaletą jest prostota, co skraca czas potrzebny na implementację. Jeśli potrzebna jest możliwość zmiany sposobu działania uruchomionego skryptu i planowane jest jego uruchamianie w interaktywny sposób, to najlepszym rozwiązaniem rzeczywiście może być pobieranie danych z wiersza poleceń.

Implementacja funkcji pobierania danych z wiersza poleceń jest łatwa. Największą wadą tej techniki jest to, że angażuje użytkownika. Można to obejść, przypisując parametrom wejściowym wartości domyślne i określając domyślny sposób działania skryptu.

Sposób użycia zmiennej automatycznej \$args

Dane z wiersza poleceń do skryptu można pobierać na kilka sposobów. Najłatwiej jest użyć argumentów wiersza poleceń. Po uruchomieniu skryptu Windows PowerShell następuje utworzenie automatycznej zmiennej `$args`, w której są zapisywane wartości przekazane do skryptu podczas jego uruchamiania.

```
Get-Bios.ps1
```

```
Get-WmiObject -Class Win32_Bios -computername $args
```


Skrypt *Get-Bios.ps1* uruchamia się przez wywołanie go i przekazanie mu nazwy komputera docelowego. Jako że zmienna `$args` automatycznie przyjmuje łańcuch z wejścia, nie trzeba wpisywać nazwy komputera docelowego w cudzysłowie.

```
PS bp:\> .\Get-Bios.ps1 localhost
```

Podczas działania skryptu wartość przekazana przez wiersz poleceń jest obecna na dysku zmiennych Windows PowerShell. Wartość przekazaną do skryptu można sprawdzić, wysyłając zapytanie o zmienną `$args` do tego dysku, jak pokazano poniżej:

```
Get-Item -path variable:args
```

Poniżej pokazano wynik zwrócony przez powyższe zapytanie:

```
PSPath      : Microsoft.PowerShell.Core\Variable::args
PSDrive     : Variable
PSProvider  : Microsoft.PowerShell.Core\Variable
PSIsContainer : False
Name        : args
Description  :
Value       : {localhost}
Visibility   : Public
Module      :
ModuleName  :
Options     : None
Attributes  : {}
```

Mimo że pobierana przez dysk zmiennych Windows PowerShell zmienna `$args` dostarcza wielu cennych informacji, łatwiej jest użyć polecenia `Get-Variable`:

```
Get-Variable args
```

Morał

Znak dolara nie należy do nazwy zmiennej

W poleceniu `Get-Variable` nie należy wpisywać znaku dolara przed nazwą zmiennej. Jest to bardzo mylące dla początkujących użytkowników konsoli, którzy myślą, że przed nazwą zmiennej zawsze powinien znajdować się ten znak. Podczas gdy rzeczywiście nazwy zmiennych poprzedza się znakiem dolara, znak ten z technicznego punktu widzenia nie należy do nazwy zmiennej. Jest tylko znacznikiem informującym, że określonego łańcucha należy używać jako zmiennej, lecz nie należy on do jej nazwy. Dlatego polecenie `Get-Variable` nie działa, gdy przed nazwą zmiennej wpisze się znak dolara. Poniżej pokazana jest informacja o błędzie zwracana w takim przypadku:

```
PS bp:\> Get-Variable $args
Get-Variable : Cannot find a variable with name 'localhost'.
At C:\Users\edwils.NORTHAMERICA\AppData\Local\Temp\tmp994A.tmp.ps1:17
char:13
+ Get-Variable <<<< $args
+ ~~~~~
+ CategoryInfo          : ObjectNotFound: (localhost:String) [Get-Variable], ItemNotFoundException
+ FullyQualifiedErrorId : VariableNotFound,Microsoft.PowerShell.Commands.GetVariableCommand
```

Jeśli dobrze przyjrzyysz się tym informacjom, to dowiesz się, że nie udało się znaleźć zmiennej o nazwie localhost. To stanowi wskazówkę na temat tego, co dzieje się za kulisami polecenia Get-Variable. Przekształca ono zmienną \$args w znajdującą się w niej wartość, a następnie szuka zmiennej o takiej nazwie. Takie zastępowanie zmiennej \$args zamiast szukania samej tej zmiennej może mieć nieoczekiwane skutki i zmusić użytkownika do straty kilku godzin na poszukiwanie źródła błędu. Przypuśćmy, że mamy następujący kod:

```
$localhost = "my computer"
Get-WmiObject -Class Win32_Bios -computername $args
Get-Variable $args
```

Jeśli skrypt zostanie uruchomiony w sposób pokazany tutaj, to nie wystąpi żaden błąd. Zamiast tego na ekranie zobaczymy następujące dane:

```
PS bp:\> .\Get-Bios.ps1 localhost
SMBIOSBIOSVersion : 7LETB7WW (2.17 )
Manufacturer      : LENOVO
Name               : Ver 1.00PARTTLx
SerialNumber       : L3L4518
Version            : LENOVO - 2170
Name               : localhost
Description        :
Value              : my computer
Visibility          : Public
Module              :
ModuleName          :
Options             : None
Attributes          : {}
```

Oczywiście w większości przypadków zapytania do dysku zmiennych wysłała się tylko w celach diagnostycznych — czyli w czasie, gdy nad naszym czołem gromadzą się czarne chmury zdenerwowania.

Przekazywanie kilku wartości do zmiennej \$args

Jeśli przez wiersz poleceń trzeba przekazać kilka wartości i spróbuj się to zrobić przy użyciu automatycznej zmiennej \$args, to konsola wyświetli pokazaną poniżej informację o błędzie ostrzegającą o niepasującym typie. W wiadomości tej napisano, że próbowano przekazać tablicę obiektów do łańcucha oraz że parametr -computername wymaga łańcucha na wejściu.

```
Get-WmiObject : Cannot convert 'System.Object[]' to the type 'System.String' required by
parameter 'ComputerName'. Specified method is not supported.
At C:\Users\edwils.NORTHAMERICA\AppData\Local\Temp\tmp774.tmp.ps1:18 char:47
+ Get-WmiObject -Class win32_bios -computername <<<< $args
    + CategoryInfo          : InvalidArgument: (:) [Get-WmiObject], ParameterBindingException
    + FullyQualifiedErrorId : CannotConvertArgument,Microsoft.PowerShell.Commands.GetWmiObjectCommand
```

Błąd ten nie został spowodowany przez tablicę, tylko jest wynikiem tego, że automatyczna zmienna \$args jest tablicą typu System.Object. Polecenie Get-WmiObject w parametrze -computername przyjmuje tablicę nazw komputerów. Pokazano to w poniższym skrypcie, w którym przekazano tablicę nazw komputerów bezpośrednio do parametru -computername i pobrano informacje BIOS przy użyciu klasy WMI Win32_Bios:

```
PS C:\> Get-WmiObject -Class Win32_Bios -computername localhost,loopback
SMBIOSBIOSVersion : 7LETB7WW (2.17 )
Manufacturer      : LENOVO
Name              : Ver 1.00PARTTBLx
SerialNumber      : L3L4518
Version           : LENOVO - 2170

SMBIOSBIOSVersion : 7LETB7WW (2.17 )
Manufacturer      : LENOVO
Name              : Ver 1.00PARTTBLx
SerialNumber      : L3L4518
Version           : LENOVO - 2170
```

Jest kilka sposobów na rozwiązanie tego problemu. Pierwszy polega na indeksowaniu tablicy i wymuszeniu pobrania nazw komputerów w sposób pokazany w skrypcie *Get-BiosArray1.ps1*.

Get-BiosArray1.ps1

```
Get-WmiObject -Class Win32_Bios -computername $args[0]
```

Technika bezpośredniego indeksowania automatycznej zmiennej `$args` bardzo dobrze się sprawdza. Podczas gdy wygląda to tak, jakby był pobierany tylko pierwszy element tablicy, w rzeczywistości zmienna `$args` pobiera oba elementy. Jako że konsola Windows PowerShell automatycznie obsługuje zmianę między tablicą jedno- i wieloelementową, technika indeksowania pierwszego elementu tablicy działa niezależnie od tego, czy dostarczony jest jeden element, czy więcej elementów. Sposób obsługi tablic informacji przez zmienną automatyczną Windows PowerShell `$args` pokazano w skrypcie *StringArgs.ps1*.

StringArgs.ps1

```
'Wartość arg0: ' + $args[0] + ', wartość arg1: ' + $args[1]
```

Jeśli skrypt *StringArgs1.ps1* zostanie uruchomiony z przekazaniem tablicy "string1", "String2" przez wiersz poleceń, to dla elementu `$args[0]` zostanie wyświetlona cała ta tablica, a dla elementu `$args[1]` nic nie zostanie wyświetlone.

```
PS C:\> StringArgs.ps1 "string1","String2"
Wartość arg0: string1 String2, wartość arg1
PS C:\>
```

Lepszym sposobem na obsługę tablicy przekazanej do zmiennej automatycznej `$args` jest użycie polecenia `Foreach-Object` i przekazanie tablicy do polecenia `Get-WmiObject`, jak pokazano poniżej:

Get-Biosarray2.ps1

```
$args | Foreach-Object {
Get-WmiObject -Class Win32_Bios -computername $_
}
```

Jeśli skrypt *Get-Biosarray2.ps1* zostanie uruchomiony z tablicą nazw komputerów w wierszu poleceń Windows PowerShell, wyświetli następujące informacje:

```
PS C:\> Get-BiosArray2.ps1 localhost,loopback
SMBIOSBIOSVersion : 7LETB7WW (2.17 )
Manufacturer      : LENOVO
```

```
Name           : Ver 1.00PARTTBLx
SerialNumber    : L3L4518
Version         : LENOVO - 2170
```

```
SMBIOSBIOSVersion : 7LETB7WW (2.17 )
Manufacturer    : LENOVO
Name           : Ver 1.00PARTTBLx
SerialNumber    : L3L4518
Version         : LENOVO - 2170
```

Polecenie `Foreach-Object` ma dwie zalety. Po pierwsze: użycie go poprawia czytelność kodu, ponieważ nie naraża czytelnika na szok związany z użyciem nieznannej techniki. Każdy, kto patrzy na kod źródłowy i widzi, że skrypt przyjmuje tablicę poprzez zmienną `$args`, spodziewa się znaleźć również polecenie `Foreach-Object` służące do przeglądania tej tablicy. Druga zaleta polega na tym, że skrypt zadziała także wtedy, gdy zostanie przekazana tylko jedna wartość.

Niestety gdyby zastosować to samo podejście w skrypcie *StringArgsArray.ps1*, wartość tablicy `$args` zostałaby powtórzona dwa razy. Poniżej znajduje się kod źródłowy skryptu *StringArgsArray.ps1*.

StringArgsArray.ps1

```
$args | Foreach-Object {
    'Wartość arg0: ' + $_ + ', wartość arg1: ' + $_
}
```

Poniżej znajduje się przykład użycia skryptu *StringArgsArray.ps1* i zwracanego wyniku:

```
PS C:\> StringArgsArray1.ps1 "string1","String2"
Wartość arg0: string1 String2, wartość arg1: string1 String2
PS C:\>
```

Jak widać, wyświetlane są oba elementy tablicy `$args`. Gdybyśmy zmienili ten skrypt tak, aby indeksować tablicę wskazywaną przez zmienną automatyczną `$_` (reprezentującą aktualny element w potoku), to otrzymalibyśmy oba elementy z tej tablicy. Poniżej znajduje się zmodyfikowana wersja skryptu o nazwie *StringArgsArray2.ps1*.

StringArgsArray2.ps1

```
$args | Foreach-Object {
    'Wartość arg0: ' + $_[0] + ', wartość arg1: ' + $_[1]
}
```

Ten skrypt zwraca poprawne informacje.

```
PS C:\> StringArgsArray1.ps1 "string1","String2"
Wartość arg0: string1, wartość arg1: String2
PS C:\>
```

Jednak najczęściej występującym problemem ze zmienną automatyczną `$args` nie jest konieczność obsługi wielu elementów pochodzących z wiersza poleceń, tylko konieczność poradzenia sobie w sytuacji, gdy użytkownik nie przekaze żadnego argumentu przez wiersz poleceń. Jeśli uruchomimy skrypt *Get-Bios.ps1* i nie prześlemy żadnej wartości, polecenie `Get-WmiObject` zgłosi błąd:

Get-WmiObject : Cannot validate argument on parameter 'ComputerName'. The argument is null, empty, or an element of the argument collection contains a null value. Supply a collection that does not contain any null values and then try the command again.

At C:\Users\edwils.NORTHAMERICA\AppData\Local\Temp\tmpF8E3.tmp.ps1:16 char:46

```
+ Get-WmiObject -Class Win32_Bios -computername <<<< $args
    + CategoryInfo          : InvalidData: (:) [Get-WmiObject], ParameterBindingValidationException
    + FullyQualifiedErrorId : ParameterArgumentValidationError,Microsoft.PowerShell.Commands.GetWmiObjectCommand
```

Są dwa sposoby na poradzenie sobie w przypadku braku argumentów wiersza poleceń i w obu należy użyć własności `count` zmiennej `$args`. W pierwszym przykładzie, jeśli wartość własności `count` jest równa 0, wyświetlamy stosowną informację i zamykamy skrypt, jak pokazano poniżej:

Get-BiosArgsCheck1.ps1

```
If($args.count -eq 0)
{
    Write-Host -foregroundcolor Cyan "Podaj nazwę komputera."
    Exit
} #end if
Get-WmiObject -Class Win32_Bios -computername $args
```

Ilość tekstu związanego z tworzeniem własnej wiadomości o błędzie można zmniejszyć przez zgłoszenie błędu za pomocą instrukcji `Throw`, która automatycznie wyświetla dane na czerwono. Dzięki temu można pozbyć się polecenia `Write-Host` umożliwiającego wyświetlenie tekstu w innym kolorze niż biały. W skrypcie *Get-BiosArgsCheck2.ps1* do zgłaszania błędu użyto właśnie instrukcji `Throw`. Wiadomością o błędzie do wyświetlenia na ekranie jest łańcuch znajdujący się za tą instrukcją. Ponadto zastosowano jeszcze jedną optymalizację, polegającą na sprawdzeniu, czy zmienna `$args` ma własność `count`, za pomocą operatora negacji `!`, który traktuje wyrażenie `$args.count` tak, jakby było wyrażeniem logicznym. Jeśli własność `count` ma wartość 0, to wyrażenie `(!$args.count)` ma wartość `false` i zostaje wykonana instrukcja `Throw`. Przykład użycia tej instrukcji jest przedstawiony w skrypcie *Get-BiosArgsCheck2.ps1*, którego treść widać poniżej.

Get-BiosArgsCheck2.ps1

```
If(!$args.count)
{
    Throw "Podaj nazwę komputera."
} #end if
Get-WmiObject -Class Win32_Bios -computername $args
```

Należy pamiętać, że instrukcja `Throw` generuje błąd. Błąd ten jest obiektem klasy `RuntimeException`, który zostaje zapisany w zmiennej `$error`. Dobrym zwyczajem jest unikanie stosowania instrukcji `Trow`, chyba że jakaś czynność rzeczywiście powoduje błąd. Opuszczenie parametru przez użytkownika w istocie nie powoduje zwrócenia błędu. My już przechwyciliśmy błąd, który powstałby w wyniku niesprawdzenia własności `count` zmiennej `$args`.

PS C:\Program Files\MrEdSoftware\MrEdScriptEditor> \$error

Podaj nazwę komputera.

At C:\Users\edwils.NORTHAMERICA\AppData\Local\Temp\tmp72FE.tmp.ps1:17 char:9

```
+ Throw <<<< "Podaj nazwę komputera."
    + CategoryInfo          : OperationStopped: (Please supply computer name:String) [], RuntimeException
    + FullyQualifiedErrorId : Podaj nazwę komputera.
```

Za pomocą instrukcji `Trap` można przechwycić błąd dotyczący wiązania parametrów. Jeśli użytkownik przy wywoływaniu skryptu nie poda nazwy komputera w argumencie wiersza poleceń, nastąpi zgłoszenie błędu. Błąd ten będzie egzemplarzem klasy `Microsoft .NET ParameterBindingException`, która znajduje się w przestrzeni nazw `System.Management.Automation`. Ten rodzaj błędu jest zgłaszany w przypadku wystąpienia problemu z wiązaniem parametrów przekazanych do skryptu. Inne parametry dotyczące WMI, np. użycie niewłaściwej nazwy klasy WMI, nie są związane z wiązaniem parametrów, więc nie powodują zgłoszenia wyjątku `ParameterBindingException`.

Zaletą wychwytywania konkretnego rodzaju błędów za pomocą instrukcji `Trap` jest to, że można dostarczyć precyzyjną informację o zaistniałej sytuacji użytkownikowi. Zamiast tylko informować, że wystąpił jakiś problem ze skryptem, można napisać, co konkretnie się stało, i zasugerować możliwe rozwiązanie. W razie potrzeby w skrypcie można użyć kilku instrukcji `Trap`. Ich działanie polega na tym, że przechwytyują błąd, wyświetlają wiadomość i elegancko zamykają skrypt. Wprawdzie w zmiennej `$error` zostaje zapisany obiekt błędu, ale nie jest on prezentowany użytkownikowi. W poniższym skrypcie *Get-BiosArgsTrap1.ps1* znajduje się przykład użycia instrukcji `Trap` do wyświetlenia wiadomości o błędzie w przypadku, gdy skrypt zostanie uruchomiony bez podania wartości dla zmiennej `$args`.

Get-BiosArgsTrap1.ps1

```
Trap [System.Management.Automation.ParameterBindingException]
{
    Write-Host -foregroundcolor cyan "Podaj nazwę komputera."
    Exit
}

Get-WmiObject -Class Win32_Bios -computername $args
```

Jeśli po uruchomieniu skryptu *Get-BiosArgsTrap1.ps1* wystąpi błąd `ParameterBindingException`, to zostanie on przechwycony. Poniżej pokazano efekt tego przechwycenia:

```
PS C:\> Get-BiosArgsTrap1.ps1
Podaj nazwę komputera.
PS C:\>
```

Jeśli błąd wystąpi podczas działania skryptu, zostanie wyświetlona informacja o błędzie związanym z zaistniałą sytuacją.

Można też użyć konstrukcji `Try-Catch-Finally`. W klauzuli `Try` próbuje się wykonać pewną czynność. Błąd przechwytuje się w klauzuli `Catch`, a czynności wykonywane po zakończeniu wszystkich działań definiuje się w klauzuli `Finally`.

Przykład użycia konstrukcji `Try-Catch-Finally` znajduje się w skrypcie *GetBiosTryCatchFinally.ps1*. W sekcji `Try` za pomocą polecenia `Get-WmiObject` pobierane są informacje dotyczące BIOS-u z klasy WMI `Win32_Bios`. Nazwa komputera docelowego jest przekazywana do zmiennej automatycznej `$args` przez wiersz poleceń. Jeśli zostanie zgłoszony błąd `System.Management.Automation.ParameterBindingException`, zostanie on przechwycony przez klauzulę `catch`. Gdy zostanie zgłoszony wyjątek dotyczący parametru, nastąpi wykonanie polecenia `Write-Host` i wyświetlenie czerwonego napisu `Podaj nazwę komputera.` Kod znajdujący się w klauzuli `Finally` jest wykonywany zawsze, więc napis `Kasowanie obiektu $error.` zostanie wyświetlony bez względu na to, czy błąd wystąpi, czy nie. Ponadto dzięki temu zawsze będzie kasowany obiekt błędu. Poniżej znajduje się kompletny kod źródłowy skryptu *GetBiosTryCatchFinally.ps1*.

GetBiosTryCatchFinally.ps1

```
Try
{ Get-WmiObject -class Win32_Bios -computer $args }
Catch [System.Management.Automation.ParameterBindingException]
{ Write-Host -foregroundcolor cyan "Podaj nazwę komputera." }
Finally
{ 'Kasowanie obiektu $error.' ; $error.clear() }
```

Sposób użycia instrukcji Param

Zmienna automatyczna \$args umożliwia szybkie i łatwe pobieranie danych do skryptu z poziomu wiersza poleceń. Ale jak wykazałem w podrozdziale „Sposób użycia zmiennej automatycznej \$args”, ceną za tę prostotę jest elastyczność. Podczas gdy zmienna \$args doskonale nadaje się do pobierania pojedynczych wartości z wiersza poleceń, w przypadku wielu wartości sprawdza się znacznie gorzej. Ponadto gdy używana jest zmienna automatyczna \$args, nie ma możliwości tworzenia przełączników.

Przy użyciu instrukcji Param można tworzyć zarówno argumenty posiadające nazwy, jak i przełączniki. Konstrukcja ta składa się ze słowa kluczowego Param i nawiasu okrągłego, jak pokazano poniżej:

```
Param($computer)
```

Jeśli trzeba zdefiniować parametr z wartością domyślną, należy przypisać mu wartość za pomocą znaku równości, jak widać poniżej:

```
Param($computer = "localhost")
```

Instrukcja Param musi znajdować się w pierwszym niebędącym komentarzem wierszu skryptu. Jeśli zostanie wpisana gdziekolwiek indziej, wystąpi błąd. W poniższym przykładzie wprowadzicie błąd, ale skrypt nie przestaje działać.

BadParam.ps1

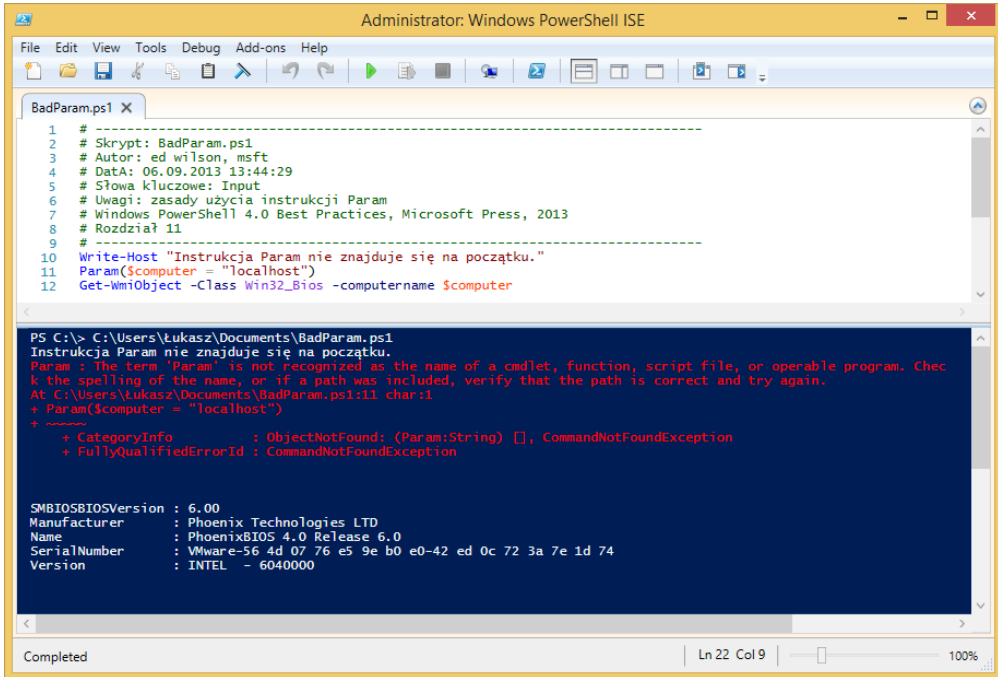
```
Write-Host "Instrukcja Param nie znajduje się na początku."
Param($computer = "localhost")
Get-WmiObject -Class Win32_Bios -computername $computer
```

Zwrócony komunikat o błędzie informuje, że Param nie jest poleceniem, funkcją, skryptem ani programem. Błąd ten widać na rysunku 11.1.

Skrypt *Get-BiosParam.ps1* ilustruje sposób użycia słowa kluczowego Param do utworzenia nazwanego argumentu oraz przypisania wartości domyślnej zmiennej \$computer. Polecenie Get-WmiObject pobiera przy użyciu klasy WMI Win32_Bios informacje dotyczące BIOS-u z komputera, którego nazwa została podana w zmiennej \$computer. Jest to nazwa wpisana przy uruchamianiu skryptu lub nazwa komputera lokalnego.

Parametr -computer można przekazać przez wiersz poleceń na trzy sposoby:

- Przez wpisanie całej nazwy parametru.
- Przez wpisanie części nazwy parametru. Należy wpisać taką jej część, aby nie dało się jej pomylić z czymkolwiek innym.
- Przez pominięcie nazwy parametru i zdefiniowanie go w sposób pozycyjny.



RYСУNEK 11.1. Gdy instrukcja `Param` nie znajduje się w pierwszym wierszu skryptu, jest to błąd

Poniżej przedstawione są te trzy metody przekazywania parametrów w wierszu poleceń na przykładzie skryptu *Get-BiosParam.ps1*:

```
PS C:\> Get-BiosParam.ps1 -computer loopback
SMBIOSBIOSVersion : 7LETB7WW (2.17 )
Manufacturer : LENOVO
Name : Ver 1.00PARTTBLx
SerialNumber : L3L4518
Version : LENOVO - 2170
```

```
PS C:\> Get-BiosParam.ps1 -c loopback
SMBIOSBIOSVersion : 7LETB7WW (2.17 )
Manufacturer : LENOVO
Name : Ver 1.00PARTTBLx
SerialNumber : L3L4518
Version : LENOVO - 2170
```

```
PS C:\> Get-BiosParam.ps1 loopback
SMBIOSBIOSVersion : 7LETB7WW (2.17 )
Manufacturer : LENOVO
Name : Ver 1.00PARTTBLx
SerialNumber : L3L4518
Version : LENOVO - 2170
```

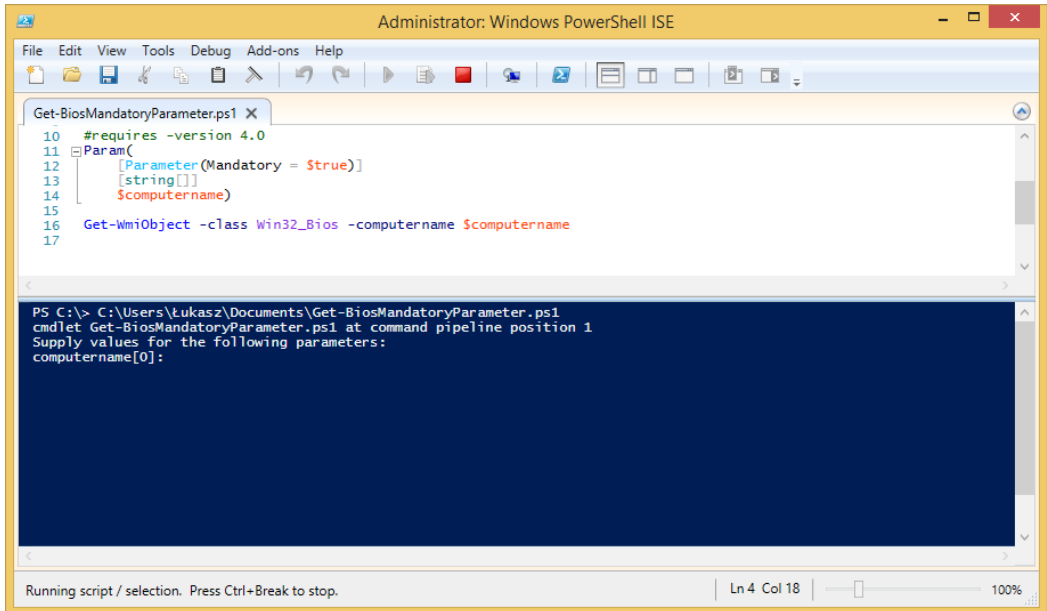
Poniżej znajduje się kompletny kod źródłowy skryptu *Get-BiosParam.ps1*.

Get-BiosParam.ps1

```
Param($computer = "localhost")
Get-WmiObject -Class Win32_Bios -computername $computer
```


Tworzenie obowiązkowego parametru

Aby utworzyć parametr, którego przekazanie jest obowiązkowe, należy użyć znacznika wiązania parametru oraz ustawić wartość atrybutu `mandatory` na `$true`. Atrybut ten powoduje wyświetlenie prośby o podanie wartości dla parametru, jeśli nie zostanie ona podana przy uruchamianiu skryptu. Widać to na rysunku 11.2, na którym użytkownik może uruchomić skrypt, nie powodując błędu.



RYСУNEK 11.2. Konsola Windows PowerShell monitoruje o podanie brakujących wartości dla parametrów wiersza poleceń z atrybutem `mandatory`

Znaczniki parametrów należą do bardzo przydatnych składników Windows PowerShell i wspomagają współpracę ze starszymi wersjami konsoli. W skrypcie *Get-BiosMandatoryParameter.ps1* znajduje się instrukcja `#requires -version 4.0`, która uniemożliwia uruchomienie skryptu w starszych środowiskach Windows PowerShell. Instrukcja `Param` służy do tworzenia parametrów wiersza poleceń. Instrukcja `[Parameter(Mandatory = $true)]` sprawia, że parametr `-computername` jest obowiązkowy. Instrukcja `[string[]]` przekształca parametr `-computername` w tablicę. Jeśli przy uruchamianiu skryptu *GetBiosMandatoryParameter.ps1* nie zostanie podany żaden parametr, skrypt ten będzie wyświetlał monity o podanie kolejnych wartości dla parametru `-computername`, aż użytkownik dwa razy naciśnie klawisz `Enter`. Jeśli dla parametru `-computername` powinna być przyjmowana tylko pojedyncza wartość, należy opuścić operator `[]`, jak pokazano poniżej:

```

Param(
    [Parameter(Mandatory = $true)]
    [string]
    $computername)
  
```

Wartość parametru `-computername` przekazana przez wiersz poleceń w razie możliwości jest przekształcana w łańcuch, ponieważ w definicji tego parametru znajduje się ogranicznik typu `[string]`. Dokładnie tak samo jest zbudowany parametr `-computername` polecenia `Get-WmiObject`,

który również przyjmuje tablicę łańcuchów. Jeśli na przykład ktoś spróbuje ograniczyć typ danych wejściowych do liczb całkowitych, zostanie zgłoszony błąd.

```
PS bp:\> .\Get-BiosMandatoryParameter.ps1 [int]12
Get-WmiObject : Invalid parameter
At C:\data\BookDocs\PowerShellBestPractices\Scripts\chapter12\Get-BiosMandatory
Parameter.ps1:20 char:14
+ Get-WmiObject <<<< -class Win32_Bios -computername $computername
+ CategoryInfo          : InvalidOperation: (:) [Get-WmiObject], ManagementException
+ FullyQualifiedErrorId : GetWMIManagementException,Microsoft.
PowerShell.Commands.GetWmiObjectCommand
```

Poniżej znajduje się kompletny kod źródłowy skryptu *Get-BiosMandatoryParameter.ps1*.

Get-BiosMandatoryParameter.ps1

```
#requires -version 4.0
Param(
    [Parameter(Mandatory = $true)]
    [string[]]
    $computername)

Get-WmiObject -class Win32_Bios -computername $computername
```

Wiedza tajemna

Gary Siepser, Premier Field Engineer Microsoft Corporation

Kilka lat temu, gdy dopiero zaczynałem pisać kod wielokrotnego użytku i zaawansowane funkcje, starałem się naśladować sposób działania moich ulubionych poleceń cmdlet, takich jak Stop-Process. W poleceniu tym podoba mi się to, że można przekazywać obiekty procesów bezpośrednio do parametru -InputObject. Podoba mi się elastyczność tego polecenia uzyskana przez wysokiej jakości projekt parametrów.

Gdy zacząłem pisać własną funkcję o nazwie Export-Excel (wiem, że już kilka takich jest, ale to było parę lat temu), musiałem dokładnie się wczytać, jak implementuje się takie parametry. Stało się dla mnie oczywiste, że muszę oprzeć strukturę funkcji na konstrukcji Begin-Process-End, ale i tak samodzielnie musiałem rozwiązać pewne kwestie projektowe.

W bloku Process dostęp do obiektu aktualnie znajdującego się w potoku można uzyskać poprzez zmienną \$_, ale przy użyciu atrybutu [Parameter] można także dokonać mapowania danych z potoku bezpośrednio na parametry. Zdecydowanie bardziej odpowiadało mi rozwiązanie z mapowaniem parametrów, ponieważ zapewniało więcej możliwości obsługi wielu parametrów przez potok.

Kolejnym wyborem dotyczącym projektu parametru -InputObject był wybór typu obiektu do ścisłego typowania. Miał to być główny parametr do odbierania obiektów do wyeksportowania do pliku Excel. Po paru testach doszedłem do wniosku, że w potoku powinien móc pojawić się

każdy rodzaj obiektu oraz że powinienem eksportować wszystkie własności obiektów. Postanowiłem więc użyć ogranicznika typu `[Object]`, który zezwala na używanie wszystkich typów obiektów. Ponadto ostatecznie zdecydowałem się na zastosowanie ogranicznika `[Object[]]`, aby umożliwić obsługę wielu egzemplarzy przekazywanych obiektów i pobieranie danych z potoku.

Ostatnią trudnością do pokonania był rozdźwięk między przekazywaniem obiektów potokowo i jako parametru nazwanego. Gdy obiekty są przekazywane przez potok, blok `Process` działa jak pętla `Foreach`, tzn. jest wykonywany raz dla każdego pojawiającego się w potoku obiektu. Jeśli jednak te same obiekty przekaże się jako argument parametru, blok `Process` zostanie wykonany tylko raz. Gdy w potoku przekaże się tablicę, zostaje ona rozwinięta, dzięki czemu parametr otrzymuje wiele obiektów, co powoduje odpowiednie wykonanie bloku `Process`. Gdy tablica zostanie przekazana jako argument, parametr otrzymuje pojedynczy obiekt tablicowy zamiast wielu obiektów.

Problem ten rozwiązałem w ten sposób, że opakowałem treść bloku `Process` w pętlę `Foreach`. Jeśli obiekty są przekazywane przez potok, pętla enumeryje tylko pierwszy obiekt i wszystko jest w porządku. Jeśli obiekty zostają przekazane jako argument, czyli pojedynczy obiekt tablicowy, pętla `Foreach` rozwija i wielokrotnie wykonuje blok kodu, tak jak to robi blok `Process` dla potoku. Podczas gdy początkowo definiowanie pętli w połączeniu z pętlowym blokiem `Process` wydawało się dziwne, ostatecznie rozwiązanie to sprawdziło się doskonale.

Sposób użycia atrybutów parametrów

W skrypcie *Get-BiosMandatoryParameter.ps1* użyto atrybutu parametru `mandatory`, aby uniemożliwić wykonanie tego skryptu bez podania wartości dla parametru `-computername`. Istnieje jeszcze kilka innych atrybutów, za pomocą których można zmieniać domyślne działanie parametrów w bloku `Param`. W tabeli 11.1 znajduje się ich zestawienie.

TABELA 11.1. Atrybuty parametrów

| Nazwa atrybutu | Opis |
|----------------|--|
| Mandatory | <p>Atrybut <code>Mandatory</code> oznacza, że dany parametr trzeba obowiązkowo zdefiniować przy wywoływaniu funkcji. Bez tego atrybutu parametr jest opcjonalny.</p> <p>Przykład:</p> <pre>[parameter(Mandatory=\$true)]</pre> |
| Position | <p>Atrybut <code>Position</code> określa pozycję parametru. Jeśli nie jest zdefiniowany, przy ustawianiu parametru należy podać jego nazwę lub alias. Ponadto jeśli żaden z parametrów funkcji nie ma określonej pozycji, środowisko Windows PowerShell automatycznie przypisuje im pozycje na podstawie kolejności ich otrzymywania.</p> <p>Przykład:</p> <pre>[parameter(Position=0)]</pre> |

TABELA 11.1. Atrybuty parametrów — ciąg dalszy

| Nazwa atrybutu | Opis |
|--------------------------------------|---|
| ParameterSetName | <p>Atrybut <code>ParameterSetName</code> określa zestaw, do którego należy dany parametr. Jeśli żaden zestaw parametrów nie jest określony, parametr należy do wszystkich zestawów parametrów zdefiniowanych przez funkcję. Oznacza to, że każdy zestaw parametrów musi zawierać jeden niepowtarzalny parametr nienależący do żadnego innego zestawu.</p> <p>Przykład:</p> <pre>[parameter(Mandatory=\$true, ParameterSetName = "CN")]</pre> |
| ValueFromPipeline | <p>Atrybut <code>ValueFromPipeline</code> sprawia, że parametr przyjmuje dane z obiektu z potoku. Należy go zdefiniować, gdy polecenie pobiera kompletny obiekt, a nie tylko jakąś jego własność.</p> <p>Przykład:</p> <pre>[parameter(Mandatory=\$true, ValueFromPipeline=\$true)]</pre> |
| ValueFromPipelineBy ↳PropertyName | <p>Atrybut <code>ValueFromPipelineByPropertyName</code> sprawia, że parametr przyjmuje dane z własności obiektu z potoku.</p> <p>Przykład:</p> <pre>[parameter(Mandatory=\$true, ValueFromPipelineByPropertyName=\$true)]</pre> |
| ValueFromRemaining ↳Arguments | <p>Atrybut <code>ValueFromRemainingArguments</code> sprawia, że parametr przyjmuje wszystkie argumenty, które nie są związane z parametrami funkcji.</p> <p>Przykład:</p> <pre>[parameter(Mandatory=\$true, ValueFromRemainingArguments=\$true)]</pre> |
| HelpMessage | <p>Atrybut <code>HelpMessage</code> służy do definiowania krótkiego opisu parametru.</p> <p>Przykład:</p> <pre>[parameter(Mandatory=\$true, HelpMessage="Tablica nazw komputerów.")]</pre> |
| Alias | <p>Atrybut <code>Alias</code> służy do tworzenia dodatkowej nazwy dla parametru. Może być przydatny, gdy nazwa jakiegoś parametru jest bardzo długa.</p> <p>Przykład:</p> <pre>[parameter(Mandatory=\$true)] [alias("CN", "MachineName")] [String[]] \$ComputerName</pre> |

Tworzenie aliasu parametru

Atrybut `alias` instrukcji `Param` może ułatwić pracę w wierszu poleceń. Najczęściej wpisuje się go za atrybutem `parameter` w celu utworzenia krótszej nazwy parametru, którą będzie można posługiwać się w wierszu poleceń. Wprawdzie można używać funkcji uzupełniania częściowych nazw parametrów, ale wówczas trzeba wpisać taką część nazwy parametru, która nie pozostawi wątpliwości co do tego, którego parametru chcemy użyć. Spójrz na poniższą instrukcję `Param`, w której tworzone są dwa parametry:

```
Param($computername, $computerIpAddress)
```

W tym przypadku, aby rozróżnić te dwa parametry, należy wpisać `computern` i `computeri`. To dość dużo znaków, więc przydałby się alias. Przykład użycia aliasu znajduje się w pokazanym poniżej skrypcie *Get-BiosMandatoryParameterWithAlias.ps1*.

Get-BiosMandatoryParameterWithAlias.ps1

```
#requires -version 4.0
Param(
    [Parameter(Mandatory = $true)]
    [alias("CN")]
    [string[]]
    $computername
)

Get-WmiObject -class Win32_Bios -computername $computername
```

Zapiski praktyka

Jaap Brasser

Konsultant techniczny

Dla mnie wielką zaletą konsoli Windows PowerShell są dodawane do każdej nowej wersji stopniowe uaktualnienia. Do moich faworytów w Windows PowerShell 4.0 należy dodatek parametru `PipelineVariable`. W poprzednich wersjach konsoli do obsługi obiektów przez potok można było używać tylko zmiennej automatycznej `$_` (albo `$PSItem` w PowerShell 3.0). Przy użyciu parametru `PipelineVariable` można przypisać konkretną zmienną, co jest moim zdaniem bardzo przydatnym dodatkiem do Windows PowerShell 4.0.

Przydatność parametru `PipelineVariable` polega na tym, że dzięki niemu mogę bez problemu zwiększyć czytelność kodu źródłowego moich skryptów. Przede wszystkim można go używać z dowolnym poleceniem i można posługiwać się zarówno jego pełną nazwą `-PipelineVariable`, jak i skróconym aliasem `-pv`.

Poniżej znajduje się przykład wykorzystania tego w skrypcie. Skrypt ten przegląda foldery znajdujące się w katalogu `C:\Users` i przypisuje każdemu z nich zmienną `UserFolder`:

```
Get-ChildItem -Directory -Path C:\Users -PipelineVariable UserFolder
```

Warto zwrócić uwagę na brak znaku dolara przed nazwą zmiennej. Nie dodaje się go także przy tworzeniu zmiennej za pomocą polecenia `New-Variable`. Wynik wykonania tego polecenia byłby dokładnie taki sam, gdyby wykonano je bez użycia parametru `PipelineVariable`. Korzyści z jego zastosowania stają się widoczne, gdy do kodu doda się dwie instrukcje `ForEach`. Spójrz na poniższy kod:

```
Get-ChildItem -Directory -Path C:\Users -PipelineVariable UserFolder |

ForEach-Object -PipelineVariable Access -Process {

    $UserFolder.GetAccessControl().Access

} |

ForEach-Object -Process {

    [pscustomobject]@{

        Folder = $UserFolder.FullName

        User = $Access.IdentityReference

        AccessRights = $Access.FileSystemRights,$Access.AccessControlType

    }
}
```

W tym przykładzie wynik polecenia `Get-ChildItem` jest przekazywany do pętli `ForEach-Object` oraz pobierane są uprawnienia dostępu do każdego folderu, które zostają zapisane w zmiennej `$Access`. Następnie wynik tych działań zostaje przekazany do kolejnej pętli `ForEach`, która z kolei tworzy obiekty zawierające wartości, które chcielibyśmy zdobyć. Dostęp do potrzebnych nam własności możemy uzyskać za pomocą zmiennych zdefiniowanych przy użyciu parametru `-PipelineVariable`.

Kolejną rzeczą, którą warto odnotować, jest to, że zmienne te są dostępne tylko w zakresie tego potoku. Po jego zamknięciu zostają usunięte. Można to sprawdzić za pomocą polecenia `Test-Path`:

```
Test-Path -Path Variable:UserFolder
```

Gdy używam konsoli Windows PowerShell, zawsze mam wrażenie, że zbieram dodatkowe narzędzia do mojej skrzynki. Parametr `PipelineVariable` jest moim najnowszym ulubieńcem, ponieważ ułatwia przypisywanie wartości zmiennym w potoku. Przy jego użyciu można łatwo sprawdzić, co dana zmienna zawiera, i łatwiej tworzy się zagnieżdżone konstrukcje `ForEach-Object`.

Sprawdzanie poprawności danych wejściowych parametru

Wychwytywanie problemów w skrypcie poprzez inspekcję parametrów jest bardziej efektywnym rozwiązaniem niż czekanie ze sprawdzeniem parametrów do czasu uruchomienia skryptu.

W Windows PowerShell 4.0 dodano kilka atrybutów weryfikacyjnych, przy użyciu których można sprawdzać, czy parametry wiersza poleceń spełniają określone wymagania. W starszych wersjach

konsoli, aby dowiedzieć się, czy wartość wybranego parametru wiersza poleceń mieści się w określonym przedziale, najczęściej pisało się funkcję wywoływaną przy rozpoczynaniu wykonywania skryptu. Przykład zastosowania tej metody znajduje się w skrypcie *CheckNumberRange.ps1*.

Znajdująca się w skrypcie *CheckNumberRange.ps1* funkcja *Check-Number* sprawdza, czy wartość parametru *number* jest większa od 1 i mniejsza lub równa 5. Jeśli wartość ta mieści się w dozwolonym przedziale, funkcja zwraca do skryptu wartość *true*, a jeśli nie — *false*. Funkcja *Set-Number* mnoży wartość parametru *number* przez 2. W punkcie początkowym skryptu znajduje się instrukcja *if* wywołująca funkcję *Check-Number*. Jeśli funkcja ta zwróci wartość *true*, następuje wywołanie funkcji *Set-Number*. W przeciwnym razie zostaje wyświetlona informacja, że wartość zmiennej *\$number* nie mieści się w dozwolonym przedziale. Poniżej znajduje się kod źródłowy omawianego skryptu *CheckNumberRange.ps1*.

CheckNumberRange.ps1

```
Param($number)

Function Check-Number($number)
{
    if($number -ge 1 -And $number -le 5)
    { $true }
    Else
    { $false }
} #end check-number

Function Set-Number($number)
{
    $number * 2
} #end Set-Number

# *** początek skryptu ***
If(Check-Number($number))
{ Set-Number($number) }
Else
{ 'Liczba $number nie mieści się w dozwolonym przedziale.' }
```

Niektórzy pewnie wolą pozostać przy pisaniu własnych funkcji weryfikacyjnych, co może być też konieczne, gdy trzeba sprawdzić zgodność wartości z jakimiś bardziej skomplikowanymi warunkami.

Podstawowej weryfikacji przynależności liczby do określonego przedziału, takiej jak przeprowadzana przez funkcję *Check-Number*, w Windows PowerShell 4.0 można dokonać przy użyciu jednego z atrybutów weryfikacyjnych wymienionych w tabeli 11.2 w dalszej części tego rozdziału. Parametr weryfikacyjny służący do sprawdzania przedziału wartości parametru nazywa się *ValidateRange* i przykład jego użycia można znaleźć w skrypcie *ValidateRange.ps1*. W instrukcji *Param* został użyty atrybut *[ValidateRange(1,5)]*, za pomocą którego dopilnowujemy, aby wartość parametru *number* nie wykraczała poza dozwolony przedział od 1 do 5. Jeśli wartość ta mieści się w wyznaczonym przedziale, skrypt rozpoczyna wykonywanie od punktu początkowego, czyli wywołania funkcji *Set-Number*. Skrypty *ValidateRange.ps1* i *CheckNumberRange.ps1* robią to samo, czyli mnożą podaną na wejściu liczbę przez dwa, jeśli liczba ta mieści się w przedziale od 1 do 5. Poniżej znajduje się kod źródłowy skryptu *ValidateRange.ps1*.

ValidateRange.ps1

```
#requires -version 4.0
Param(
    [ValidateRange(1,5)]
    $number
)

Function Set-Number($number)
{
    $number * 2
} #end Set-Number

# *** punkt początkowy skryptu ***
Set-Number($number)
```

Generalnie lepiej sprawdzać wartości parametrów za pomocą atrybutów weryfikacyjnych niż własnoręcznie napisanych funkcji. Oto kilka powodów, dlaczego parametry weryfikacyjne są lepsze:

- redukują poziom złożoności kodu;
- sprawiają, że skrypt zachowuje się jak podstawowe polecenia Windows PowerShell;
- ułatwiają obsługę skryptów;
- ułatwiają sprawdzanie składni za pomocą polecenia `Get-Help`.

Najbardziej wszechstronnym parametrem weryfikacyjnym jest `ValidatePattern`, za pomocą którego można sprawdzać wartości parametrów według wyrażeń regularnych. Wyrażenie regularne może być prostym wzorcem określonej kombinacji znaków w nazwie komputera, jak również bardziej skomplikowanym wyrażeniem. Prosty przykład użycia wyrażeń regularnych pokazano w skrypcie *PingComputers.ps1*.

W skrypcie tym użyto parametru `ValidatePattern` do sprawdzenia, czy gdzieś w łańcuchu przekazanym jako wartość parametru `-computername` znajdują się litery DC. Warunek ten spełniają na przykład takie nazwy jak DC, DCcomputer oraz myDCcomputer. Dopasowanie następuje wtedy, gdy w łańcuchu znajdują się litery DC w dokładnie takiej kolejności. W celu przyjęcia parametrów z wiersza poleceń użyto instrukcji `Param`. Atrybut weryfikacyjny `ValidatePattern` zawiera definicję wyrażenia regularnego, za pomocą którego będzie sprawdzana wartość przekazana przez wiersz poleceń. Użyto też atrybutu `alias` do zdefiniowania alternatywnej nazwy dla parametru `-computername`. Poniżej znajduje się opisywana instrukcja `Param`:

```
Param(
    [ValidatePattern("DC")]
    [alias("CN")]
    $computername
)
```

Funkcja `New-TestConnection` wysyła za pomocą polecenia `Test-Connection` spreparowany pakiet ping do komputera docelowego określonego w parametrze `-computername`. Rozmiar bufora dla tego pakietu został zmniejszony z domyślnych 32 do 16 bajtów, a liczbę pakietów zmniejszono z 4 do 2. Dzięki temu funkcja `New-TestConnection` szybciej zwróci status komputera docelowego, zużyje mniej transferu sieciowego oraz szybciej zakończy działanie niż standardowe polecenie `Test-Connection`. Poniżej znajduje się kompletny kod źródłowy opisywanego skryptu *PingComputers.ps1*.

PingComputers.ps1

```
#requires -version 4.0
Param(
    [ValidatePattern("DC")]
    [alias("CN")]
    $computername
)

Function New-TestConnection($computername)
{
    Test-connection -computername $computername -buffersize 16 -count 2
} #end new-testconnection

# *** punkt początkowy skryptu
New-TestConnection($computername)
```

W atrybucie `ValidatePattern` można też definiować bardziej skomplikowane wyrażenia regularne. Na przykład w skrypcie *PingIpAddress.ps1* za pomocą wyrażenia regularnego sprawdzane jest, czy podany został łańcuch reprezentujący adres IP. Wzorzec ten pozwala na wpisanie czterech grup zawierających od jednej do trzech cyfr rozdzielanych kropkami. Warunki te spełniają np. łańcuchy 127.0.0.1 i 999.999.999.999 (choć ten ostatni nie jest poprawnym adresem IP). Opisywane wyrażenie regularne jest pokazane poniżej:

```
"\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}"
```

W skrypcie *PingIpAddress.ps1* znajduje się instrukcja `Param` tworząca parametry wiersza poleceń. W atrybucie `parameter` zaznaczono, że podanie wartości dla parametru jest obowiązkowe, oraz zdefiniowano informację pomocniczą, która zostanie wyświetlona, gdy użytkownik uruchomi skrypt bez podania wartości parametru. Atrybut `ValidatePattern` zawiera wyrażenie regularne do sprawdzania poprawności danych przekazanych do skryptu przez parametr `-computername`. Ponadto utworzono alias `IP`, aby nie trzeba było wpisywać długiej nazwy `-computername`. Poniżej znajduje się kompletny kod źródłowy skryptu *PingIpAddress.ps1*.

PingIpAddress.ps1

```
#requires -version 4.0
Param(
    [Parameter(Mandatory=$true,
        HelpMessage="Podaj poprawny adres IP.")]
    [ValidatePattern("\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}")]
    [alias("IP")]
    $computername
)

Function New-TestConnection($computername)
{
    Test-connection -computername $computername -buffersize 16 -count 2
} #end new-testconnection

# *** punkt początkowy skryptu
New-TestConnection($computername)
```

W tabeli 11.2 znajduje się wykaz wszystkich atrybutów weryfikacyjnych.

TABELA 11.2. Atrybuty weryfikacji wartości parametrów

| Atrybut | Opis |
|----------------------|--|
| AllowNull | <p>Atrybut AllowNull umożliwia ustawienie argumentu obowiązkowego polecenia na null.</p> <p>Przykład:</p> <pre>[AllowNull()]</pre> |
| AllowEmptyString | <p>Atrybut AllowEmptyString umożliwia ustawienie argumentu obowiązkowego polecenia na pusty łańcuch.</p> <p>Przykład:</p> <pre>[AllowEmptyString()]</pre> |
| AllowEmptyCollection | <p>Atrybut AllowEmptyCollection umożliwia użycie jako argumentu obowiązkowego polecenia pustej kolekcji.</p> <p>Przykład:</p> <pre>[AllowEmptyCollection()]</pre> |
| ValidateCount | <p>Atrybut ValidateCount określa minimalną i maksymalną liczbę argumentów przyjmowaną przez parametr.</p> <p>Przykład:</p> <pre>[ValidateCount(1,5)]</pre> |
| ValidateLength | <p>Atrybut ValidateLength określa minimalną i maksymalną długość argumentu parametru.</p> <p>Przykład:</p> <pre>[ValidateLength(1,10)]</pre> |
| ValidatePattern | <p>Atrybut ValidatePattern określa wyrażenie regularne sprawdzające poprawność struktury argumentu parametru.</p> <p>Przykład:</p> <pre>[ValidatePattern("[0-9][0-9][0-9]")]</pre> |
| ValidateRange | <p>Atrybut ValidateRange określa maksymalną i minimalną wartość argumentu parametru.</p> <p>Przykład:</p> <pre>[ValidateRange(0,10)]</pre> |
| ValidateScript | <p>Atrybut ValidateScript wyznacza skrypt sprawdzający poprawność argumentu parametru. Jeśli skrypt ten zwróci wartość false albo zgłosi wyjątek, system wykonawczy Windows PowerShell wygeneruje błąd.</p> <p>Przykład:</p> <pre>[ValidateScript({\$_ -lt 4})]</pre> |

TABELA 11.2. Atrybuty weryfikacji wartości parametrów — ciąg dalszy

| Atrybut | Opis |
|-------------------------------------|--|
| <code>ValidateSet</code> | Atrybut <code>ValidateSet</code> określa zbiór poprawnych wartości argumentu parametru. Jeśli argument parametru nie pasuje do żadnej wartości w tym zbiorze, system wykonawczy Windows PowerShell wygeneruje błąd. Przykład: <code>[ValidateSet("Tomek", "Maria", "Radek")]</code> |
| <code>ValidateNotNull</code> | Atrybut <code>ValidateNotNull</code> oznacza, że argument parametru nie może mieć wartości <code>null</code> . Przykład: <code>[ValidateNotNull()]</code> |
| <code>ValidateNotNullOrEmpty</code> | Atrybut <code>ValidateNotNullOrEmpty</code> oznacza, że argument parametru nie może mieć wartości <code>null</code> ani być pusty. Przykład: <code>[ValidateNotNullOrEmpty()]</code> |

Sposób użycia wielu argumentów parametrów

Aby użyć kilku argumentów parametrów, atrybutu `alias` oraz atrybutów weryfikacyjnych wraz z instrukcją `Param`, należy zastosować się do następujących kilku zasad:

- Argumenty parametrów powinny znajdować się w nawiasie i modyfikować atrybut `parameter` instrukcji `Param`.
- Argumenty parametrów powinny być rozdzielone przecinkami.
- Atrybut `parameter` powinien znajdować się w nawiasie kwadratowym (tak jak wszystkie inne atrybuty parametrów).
- Każdy atrybut `parameter` powinien być zdefiniowany w osobnej linii.
- Atrybutów parametrów nie rozdziela się przecinkami.
- Parametr wiersza poleceń zaczyna się od znaku dolara i kończy przecinkiem, chyba że jest na końcu wiersza, w którym to przypadku zamiast przecinka powinno znajdować się zamknięcie nawiasu instrukcji `Param`.
- Można użyć dowolnej liczby atrybutów `parameter` i weryfikacyjnych.

Przykład użycia wielu atrybutów parametrów przedstawiono w skrypcie *MultiplyNumbersCheckParameters.ps1*. Na początku znajduje się znacznik `#requires -version 4.0` uniemożliwiający uruchomienie skryptu w komputerze z zainstalowaną konsolą Windows PowerShell 1.0. Instrukcja `Param` tworzy parametry wiersza poleceń dla tego skryptu. Atrybut `parameter` oznacza parametr `FirstNumber` jako obowiązkowy, przywiązuje go do pierwszej pozycji oraz definiuje informację pomocniczą dotyczącą sposobu jego użycia. Atrybut `alias` tworzy alias nazwy `FirstNumber`. Atrybut weryfikacyjny `ValidateRange` ogranicza dopuszczalny zakres wartości parametru `FirstNumber` do przedziału od 1 do 10.

Za nazwą parametru `FirstNumber` znajdują się przecinek i atrybut `parameter` oznaczający parametr `lastnumber` jako obowiązkowy, wiążący go z pozycją 1 oraz przypisujący informację dotyczącą sposobu jego użycia. Atrybut `alias` tworzy alias `ln` dla nazwy `lastnumber`. Dodatkowo użyto ogranicznika typu `[int16]` ograniczającego zakres wartości parametru `lastnumber` do przedziału 16-bitowych liczb całkowitych, w którym maksymalna wartość wynosi 32767. Za pomocą atrybutu weryfikacyjnego `ValidateNotNullOrEmpty` zaznaczono, że parametr `lastnumber` nie może być pusty ani mieć wartości `null`. Na końcu sekcji `param` znajduje się definicja parametru `lastnumber` i zamknięcie nawiasu.

UWAGA

Pojemność wybranych typów systemowych można łatwo sprawdzić za pomocą statycznej właściwości `MaxValue`. Aby sprawdzić maksymalną wartość typu `int32`, należy pobrać jej wartość za pomocą podwójnego dwukropka, np. `[int32]::MaxValue`.

Po tych wszystkich skomplikowanych definicjach parametrów kod roboczy wydaje się prymitywny: tylko mnoży wartości podanych parametrów. Poniżej znajduje się kompletny kod źródłowy opisanego skryptu *MultiplyNumbersCheckParameters.ps1*.

MultiplyNumbersCheckParameters.ps1

```
#requires -version 4.0
Param(
    [Parameter(mandatory=$true,
               Position=0,
               HelpMessage="Podaj liczbę z przedziału od 1 do 10.")]
    [alias("fn")]
    [ValidateRange(1,10)]
    $FirstNumber,
    [Parameter(mandatory=$true,
               Position=1,
               HelpMessage="Wartość nie może być null ani pusta.")]
    [alias("ln")]
    [int16]
    [ValidateNotNullOrEmpty()]
    $LastNumber
)

$FirstNumber*$LastNumber
```

Pobieranie haseł na wejściu

W idealnym świecie nigdy nie trzeba by było wpisywać jakichkolwiek haseł. Skrypty działałyby przy użyciu personifikacji i wykrywałyby, czy użytkownik ma uprawnienia dostępu do danych. Jeśli użytkownik miałby takie uprawnienia, otrzymywałby dostęp. Gdyby ich nie miał, nie mógłby nawiązać połączenia. W pewnym sensie dokładnie to się dzieje podczas pracy ze skryptem. Kwestie dotyczące używania haseł występują w następujących przypadkach:

- Próba użycia danych z niezaufanej domeny.
- Próba użycia danych ze starej bazy danych, która nie używa zintegrowanych zabezpieczeń.

- Próba dostępu do danych z samodzielnej stacji roboczej lub serwerów, które nie są włączone do domeny.
- Zezwolenie użytkownikowi nieposiadającemu pewnych uprawnień na użycie innych danych poświadczających, aby mógł wykonać czynności, których normalnie nie mógłby wykonać.

Kwestię hasła można rozwiązać na kilka sposobów, na przykład:

- Zapisanie hasła w skrypcie.
- Zapisanie hasła w pliku tekstowym.
- Zapisanie hasła w rejestrze.
- Zapisanie hasła w Active Directory.
- Poproszenie o podanie hasła.

Zapisanie hasła w skrypcie

Najprostszym rozwiązaniem problemu obsługi hasła jest zapisanie go w skrypcie, chociaż ma ono wiele oczywistych wad. Przede wszystkim w skrypcie hasło jest przechowywane w postaci tekstowej i każdy, kto ma dostęp do tego skryptu, może je przeczytać.

Hasło można zabezpieczyć na dwa sposoby. Jednym z nich jest użycie uprawnień systemu plików NTFS do ochrony pliku przed osobami, które nie powinny poznać hasła. Drugi polega na zaszyfrowaniu skryptu za pomocą systemu szyfrowania plików (ang. *Encrypting File System* — EFS). Jako że Windows PowerShell to aplikacja .NET, może używać klas zabezpieczeń w celu automatycznego rozszyfrowywania skryptu przy użyciu certyfikatu EFS, czego nie da się zrobić przy użyciu języka VBScript. Zaszyfrowane skrypty VBScript nie zadziałają. W skrypcie *QueryComputersUseCredentials.ps1* użyto klasy ADO w celu wysłania zapytania do domeny zasobów o nazwie *nwtraders.com*. Użytkownik wykonujący to zapytanie nazywa się *LondonAdmin* i używa hasła *Password1*. Wartości te są zapisane w zmiennych i przekazywane do obiektu połączenia ADO poprzez własności *password* i *user ID*. Następnie skrypt pobiera wszystkie obiekty komputerów z domeny *nwtraders.com*. Poniżej znajduje się kod źródłowy opisywanego skryptu.

QueryComputersUseCredentials.ps1

```
$strBase = "<LDAP://dc=nwtraders,dc=msft>"
$strFilter = "(objectCategory=computer)"
$strAttributes = "name"
$strScope = "subtree"
$strQuery = "$strBase;$strFilter;$strAttributes;$strScope"
$strUser = "nwtraders\LondonAdmin"
$strPwd = "Password1"

$objConnection = New-Object -comObject "ADODB.Connection"
$objConnection.provider = "ADsDSOObject"
$objConnection.properties.item("user ID") = $strUser
$objConnection.properties.item("Password") = $strPwd
$objConnection.open("modifiedConnection")
$objCommand = New-Object -comObject "ADODB.Command"

$objCommand.ActiveConnection = $objConnection
$objCommand.CommandText = $strQuery
```

```

$ObjRecordSet = $ObjCommand.Execute()

Do
{
    $ObjRecordSet.Fields.item("name") |Select-Object Name,Value
    $ObjRecordSet.MoveNext()
}
Until ($ObjRecordSet.eof)

$ObjConnection.Close()

```

Zapisywanie hasła w pliku tekstowym

Nieco lepszym rozwiązaniem od zapisywania hasła w skrypcie jest zapisanie go w pliku tekstowym. Zaletą tego rozwiązania jest to, że hasło nie jest dostępne bezpośrednio w skrypcie. Dzięki zapisaniu go w innym pliku można zdefiniować inne zabezpieczenia dla pliku z hasłem niż dla pliku skryptu. Rozwiązanie to może być dobre dla tych, którzy potrzebują możliwości odczytania skryptu, ale nie muszą go uruchamiać. Inną zaletą tego podejścia jest to, że umożliwia używanie tego samego skryptu w różnych kontekstach bezpieczeństwa. Przykładem może być sytuacja, gdy skrypt zostanie napisany przez administratora sieci z jednej domeny, a następnie udostępniony administratorom sieci w kontekście innej domeny. Sytuacja taka często się zdarza w firmach składających się z kilku jednostek organizacyjnych, z których każda ma własną osobną infrastrukturę.

QueryComputersUseCredentials.ps1

```

$strBase = "<LDAP://dc=nwtraders,dc=msft>"
$strFilter = "(objectCategory=computer)"
$strAttributes = "name"
$strScope = "subtree"
$strQuery = "$strBase;$strFilter;$strAttributes;$strScope"
$strUser = "nwtraders\LondonAdmin"
$strPwd = Get-Content -path "C:\fso\password.txt"

$ObjConnection = New-Object -comObject "ADODB.Connection"
$ObjConnection.provider = "ADsDSOObject"
$ObjConnection.properties.item("user ID") = $strUser
$ObjConnection.properties.item("Password") = $strPwd
$ObjConnection.open("modifiedConnection")
$ObjCommand = New-Object -comObject "ADODB.Command"

$ObjCommand.ActiveConnection = $ObjConnection
$ObjCommand.CommandText = $strQuery
$ObjRecordSet = $ObjCommand.Execute()

Do
{
    $ObjRecordSet.Fields.item("name") |Select-Object Name,Value
    $ObjRecordSet.MoveNext()
}
Until ($ObjRecordSet.eof)

$ObjConnection.Close()

```

Zapisywanie hasła w rejestrze

Jako że w Windows PowerShell bardzo wygodnie pracuje się z rejestrem, dobrym wyborem może się wydawać zapisanie hasła właśnie w rejestrze. Administrator może nadać wybranemu kluczowi rejestru pewne zabezpieczenia, co jest dodatkową zaletą tego rozwiązania. Klucz rejestru można utworzyć w osobnym procesie. Po uruchomieniu skrypt może pobrać z rejestru hasło potrzebne do nawiązania zdalnego połączenia.

QueryComputersUseCredentialsFromText.ps1

```
$strBase = "<LDAP://dc=nwtraders,dc=msft>"
$strFilter = "(objectCategory=computer)"
$strAttributes = "name"
$strScope = "subtree"
$strQuery = "$strBase;$strFilter;$strAttributes;$strScope"
$strUser = "nwtraders\administrator"
$strPwd = (Get-ItemProperty HKCU:\Software\ForScripting\CompatPassword).password

$objConnection = New-Object -comObject "ADODB.Connection"
$objConnection.provider = "ADsDSOObject"
$objConnection.properties.item("user ID") = $strUser
$objConnection.properties.item("Password") = $strPwd
$objConnection.open("modifiedConnection")
$objCommand = New-Object -comObject "ADODB.Command"

$objCommand.ActiveConnection = $objConnection
$objCommand.CommandText = $strQuery
$objRecordSet = $objCommand.Execute()

Do
{
    $objRecordSet.Fields.item("name") |Select-Object Name,Value
    $objRecordSet.MoveNext()
}
Until ($objRecordSet.eof)

$objConnection.Close()
```

Zapisywanie hasła w usługach domenowych Active Directory

W miarę łatwo można rozszerzyć schemat, aby utworzyć atrybut do przechowywania hasła dla wybranych skryptów. Polega to na dostarczeniu centralnej lokalizacji, która jest dostępna z dowolnego miejsca w domenie.

UWAGA

Jeśli nie masz uprawy w dodawaniu atrybutów do schematów Active Directory, to zamiast tworzyć własne atrybuty, możesz użyć jednego z atrybutów konfigurowalnych. W każdym razie nie zapomnij użyć poprawnego identyfikatora obiektu (OID) i przetestować zmian w środowisku testowym, zanim wdrożysz je do środowiska produkcyjnego.

QueryComputersUdeCredentialsFromADDS.ps1

```
$strBase = "<LDAP://dc=nwtraders,dc=msft>"
$strFilter = "(objectCategory=computer)"
$strAttributes = "name"
$strScope = "subtree"
```

```

$strQuery = "$strBase;$strFilter;$strAttributes;$strScope"
$strUser = "nwtraders\testUser"
$strPwd = ([adsisearcher]"LDAP://cn=testUser,ou=myusers,dc=nwtraders,dc=com").compatPassword

$objConnection = New-Object -comObject "ADODB.Connection"
$objConnection.provider = "ADsDSOObject"
$objConnection.properties.item("user ID") = $strUser
$objConnection.properties.item("Password") = $strPwd
$objConnection.open("modifiedConnection")
$objCommand = New-Object -comObject "ADODB.Command"

$objCommand.ActiveConnection = $objConnection
$objCommand.CommandText = $strQuery
$objRecordSet = $objCommand.Execute()

Do
{
    $objRecordSet.Fields.item("name") | Select-Object Name, Value
    $objRecordSet.MoveNext()
}
Until ($objRecordSet.eof)

$objConnection.Close()

```

Wyświetlanie monitu o podanie hasła

Najlepszym rozwiązaniem jest monitowanie o podanie hasła, gdy jest potrzebne. Metoda ta ma kilka zalet. Największą z nich jest to, że całkowicie eliminuje problem z przechowywaniem hasła w skrypcie, pliku tekstowym, rejestrze lub jeszcze jakimś innym miejscu. Druga zaleta jest taka, że nie trzeba obsługiwać zapisanego hasła, tzn. jeśli hasło od czasu do czasu się zmienia i jest ono przechowywane w pliku, to trzeba ten plik modyfikować. Ponadto dzięki przyjmowaniu hasła z wiersza poleceń łatwiej jest rozwiązywać problemy ze skryptem. Gdy skrypt pobiera hasło ze zdalnej lokalizacji i z jakiegoś powodu ulegnie awarii, powstają problemy z łącznością i uprawnieniami do zdalnych zasobów. Oczywiście skrypt powinien też zawierać solidny mechanizm obsługi błędów, ale to wprowadza dodatkowe komplikacje, które mogą jeszcze bardziej utrudnić diagnostykę. Najłatwiejszym sposobem na wyświetlenie monitu o podanie hasła jest użycie polecenia Read-Host.

QueryComputersPromptForPassword.ps1

```

$strBase = "<LDAP://dc=nwtraders,dc=com>"
$strFilter = "(objectCategory=computer)"
$strAttributes = "name"
$strScope = "subtree"
$strQuery = "$strBase;$strFilter;$strAttributes;$strScope"
$strUser = "nwtraders\administrator"
$strPwd = Read-Host -prompt "Enter password to Connect to AD"

$objConnection = New-Object -comObject "ADODB.Connection"
$objConnection.provider = "ADsDSOObject"
$objConnection.properties.item("user ID") = $strUser
$objConnection.properties.item("Password") = $strPwd
$objConnection.open("modifiedConnection")
$objCommand = New-Object -comObject "ADODB.Command"
$objCommand.ActiveConnection = $objConnection

```



```

$objCommand.CommandText = $strQuery
$objRecordSet = $objCommand.Execute()

Do
{
    $objRecordSet.Fields.item("name") |Select-Object Name,Value
    $objRecordSet.MoveNext()
}
Until ($objRecordSet.eof)

$objConnection.Close()

```

Jeżeli polecenie obsługuje obiekt `PSCredential`, to można użyć parametru `AsSecureString` polecenia `Read-Host`. Bezpiecznych łańcuchów używa się do przekazywania poufnego tekstu. Podczas używania tekst taki jest zaszyfrowany, a po zakończeniu pracy zostaje usunięty. Hasło nigdy nie zostanie ujawnione w postaci zwykłego tekstu. Klasa `.NET System.Security.SecureString` jest niewidoczna dla modelu COM, przez co nie można jej używać poprzez klasyczne interfejsy COM.

ReadHostSecureStringQueryWmi.ps1

```

$user = "Nwtraders\administrator"
$password = Read-Host -prompt "Wpisz hasło" -asSecureString
$credential = new-object system.management.automation.PSCredential $user,$password
Get-WmiObject -class Win32_Bios -computername berlin -credential $credential

```

Ponadto istnieje możliwość zapisania skrótu (ang. *hash*) hasła w pliku tekstowym. Zaletą tej techniki jest to, że umożliwia używanie hasła w połączeniu z obiektem `PSCredential`, dzięki czemu mamy ten komfort, że nie musimy ręcznie wprowadzać hasła przy każdym uruchamianiu skryptu. Ponadto możemy przekazać skrypt innemu użytkownikowi, który nie zna hasła do konta.

Aby zrealizować ten pomysł, należy użyć polecenia `Read-Host` z parametrem `AsSecureString`. W poniższym przykładzie zaszyfrowane hasło zostaje zapisane w zmiennej `$pwd`:

```

PS C:\> $pwd = Read-Host -Prompt "Podaj hasło" -AsSecureString
Podaj hasło: *****

```

Jeśli zostanie użyta metoda `ToString` z obiektu `SecureString`, to do konsoli Windows PowerShell zostanie przekazany jedynie egzemplarz klasy `System.Security.SecureString`, jak pokazano poniżej. Jeżeli ktoś spróbuje zapisać wynik metody `ToString` w pliku tekstowym, to ujrzy w nim tylko napis `System.Security.SecureString`.

```

PS C:\> $PWD.ToString()
System.Security.SecureString

```

Aby zapisać bezpieczny łańcuch w pliku tekstowym, należy użyć polecenia `ConvertFrom-SecureString`. Jak widać poniżej, polecenie to ujawnia skrót hasła:

```

PS C:\> $PWD | ConvertFrom-SecureString
01000000d08c9ddf0115d118c7a00c04fc297eb01000000151046ea8f869541a129ff10c91b850e0000000020000000
00003660000a800000010000000aa2caba61452ffd5f973901a5dbd0e81000000004800000a0000000100000003172c7
49434dfac3262616d15dea4d1018000000916d60d590d381bff1225663c6b4dcab536fca5920077cb414000000e92d30f
80b9bf337c1a8e5d99f50f118fae2d3b

```

Skrót ten można zapisać w pliku tekstowym za pomocą pary strzałek.

```

PS C:\> $PWD | ConvertFrom-SecureString >> C:\fso\passwordHash.txt

```

Teraz plik *passwordHash.txt* zawiera dokładnie te informacje, które wcześniej zostały wyświetlone w oknie konsoli, tzn. nie napis `System.Security.SecureString`, tylko łańcuch skrótu reprezentujący klasę `SecureString`. Skróć ten można przekształcić z powrotem na bezpieczny łańcuch za pomocą polecenia `ConvertTo-SecureString`.

```
PS C:\> ConvertTo-SecureString (Get-Content C:\fso\passwordHash.txt)
System.Security.SecureString
```

UWAGA

Pamiętaj, że chcesz przekształcić w bezpieczny łańcuch zawartość pliku *passwordHash.txt*, a nie ścieżkę do niego. Gdy pierwszy raz wykonywałem tę czynność, użyłem następującego niepoprawnego polecenia: `ConvertTo-SecureString C:\fso\passwordHash.txt`. Później dopiero zdałem sobie sprawę, że próbowałem zaszyfrować ścieżkę do pliku, a nie jego zawartość.

Efektywniejszym sposobem na utworzenie pliku ze skrótem hasła jest użycie potoku, co pozwala pominąć pośrednią zmienną. Metodę tę zastosowano w poniższym poleceniu, które monituje o podanie hasła.

```
PS C:\> Read-Host -Prompt "Podaj hasło:" -AsSecureString |
>> ConvertFrom-SecureString >> C:\fso\passwordHash.txt
```

Poniżej znajduje się przykładowy skrypt, w którym użyto tego pliku tekstowego ze skrótem hasła.

UsePasswordhashFile.ps1

```
$user = "Nwtraders\administrator"
$password = ConvertTo-SecureString -String (Get-content C:\fso\passwordHash.txt)
$credential = new-object system.management.automation.PSCredential $user,$password
Get-WmiObject -class Win32_Bios -computername berlin -credential $credential
```

Wiedza tajemna

Importowanie i eksportowanie poświadczeń

**Lee Holmes, starszy programista i autor książki *Windows PowerShell Cookbook*
Microsoft Corporation**

Jedną z kwestii często podnoszonych przez użytkowników skryptów `Windows PowerShell` jest sposób poprawnego posługiwania się nazwami użytkownika i hasłami. Odpowiedzią jest polecenie `Get-Credential`, za pomocą którego tworzy się obiekty klasy `PSCredential`. Obiekt taki w odróżnieniu od poleceń przyjmujących hasła i nazwy użytkownika gwarantuje ochronę hasła w pamięci.

Jeżeli jakiś parametr przyjmuje obiekty typu `PSCredential`, to dane do konsoli `Windows PowerShell` można wprowadzać na kilka sposobów:

- **Puste:** jeśli dla obowiązkowego parametru `-credential` nie zostaną przekazane żadne informacje, konsola Windows PowerShell poprosi o podanie hasła i nazwy użytkownika.
- **Łańcuch:** jeśli do obowiązkowego parametru `-credential` zostanie przekazany łańcuch, konsola Windows PowerShell potraktuje go jako nazwę użytkownika, a następnie poprosi o podanie hasła.
- **Obiekt poświadczeń:** jeśli do obowiązkowego parametru `-credential` zostanie przekazany obiekt poświadczeń, konsola Windows PowerShell przyjmie go w niezmienionej postaci.

Wszystko jest pięknie, jeśli chcemy pracować interaktywnie, ale co zrobić, gdy trzeba napisać automatyczny skrypt przyjmujący parametr `-credential` dla polecenia? Rozwiązaniem jest przekazanie wcześniej utworzonego obiektu `PSCredential`. Jego opis znajduje się w przytoczonej poniżej recepturze 18.12 z książki *Windows PowerShell Cookbook*.

Aby zapisać hasło na dysku, pierwszą czynność z reguły trzeba wykonać ręcznie. Jeśli dane poświadczające zostały zapisane w zmiennej `$credential`, to można bezpiecznie wyeksportować z nich hasło do pliku *password.txt* za pomocą poniższych poleceń.

- W Windows PowerShell 4 do importowania i eksportowania poświadczeń służą polecenia `Export-CliXml` i `Import-CliXml`.
- W Windows PowerShell należy użyć poleceń `ConvertFrom-SecureString` i `ConvertTo-SecureString`.

Pierwsza czynność zazwyczaj musi być wykonana ręcznie. Wprawdzie nie ma obowiązkowych zasad dotyczących nazwy pliku, ale zastosujemy konwencjonalną nazwę *BieżącySkrypt.ps1.credential*. Jeśli dane poświadczające zostały zapisane w zmiennej `$credential`, można je bezpiecznie zapisać na dysku za pomocą polecenia `Export-CliXml`. Ciąg *BieżącySkrypt* należy zastąpić nazwą skryptu:

```
PS > $credPath = Join-Path (Split-Path $profile) CurrentScript.ps1.credential
PS > $credential | Export-CliXml $credPath
```

W Windows PowerShell 2 należy użyć polecenia `ConvertFrom-SecureString`:

```
PS > $credPath = Join-Path (Split-Path $profile) BieżącySkrypt.ps1.credential
PS > $credential.Password | ConvertFrom-SecureString | Set-Content
$credPath
```

W skrypcie, który ma być uruchamiany automatycznie, dodaj poniższe polecenia dla konsoli Windows PowerShell 4:

```
$credPath = Join-Path (Split-Path $profile) BieżącySkrypt.ps1.credential
$credential = Import-CliXml $credPath
```

Natomiast w konsoli Windows PowerShell 2 należy ręcznie utworzyć obiekt `PSCredential` przy użyciu hasła zaimportowanego przez polecenie `ConvertTo-SecureString`. Podczas gdy polecenie `Export-CliXml` automatycznie rejestruje nazwę użytkownika, w tym rozwiązaniu trzeba to robić ręcznie:

```
$credPath = Join-Path (Split-Path $profile) BieżącySkrypt.ps1.credential
$password = Get-Content $credPath | ConvertTo-SecureString
$credential = New-Object System.Management.Automation.PSCredential '
    "CachedUser",$password
```

Polecenia te tworzą nowy obiekt poświadczeń (dla użytkownika `CachedUser`) i zapisują go w zmiennej `$variable`.

Pierwszą myślą po przeczytaniu tego rozwiązania mogą być wątpliwości dotyczące zapisywania hasła na dysku. Niechcąc do rozsiewania poufnych informacji po dysku jest naturalna (i uzasadniona), jednak polecenie `Export-CliXml` szyfruje obiekty poświadczeń przy użyciu standardowego interfejsu API ochrony danych systemu Windows. Dzięki temu mamy pewność, że deszyfracji danych można dokonać tylko przy użyciu naszego konta. Podobnie szyfruje podane hasła polecenie `ConvertFrom-SecureString`.

Choć zabezpieczenie hasła to bardzo ważna kwestia, czasami trzeba je (lub inne poufne informacje) zapisać na dysku po to, by udostępnić je innym kontom. Często jest tak w przypadku skryptów wykonywanych przez konta usługowe i skryptów z góry przeznaczonych do przenoszenia między komputerami. Pomocne w tym są polecenia `ConvertFrom-SecureString` i `ConvertTo-SecureString`, w których można podać klucz szyfrowania.

Jeśli klucz szyfrowania wpisze się bezpośrednio w kod, technika ta traci swoje właściwości ochronne. Jeżeli użytkownik ma dostęp do treści naszego automatycznego skryptu, to może też odczytać klucz szyfrowania. A jeżeli użytkownik ma dostęp do klucza szyfrowania, to ma też dostęp do danych, które chcieliśmy chronić.

Mimo że w przedstawionym rozwiązaniu hasło jest zapisywane w katalogu zawierającym nasz profil, można je także wczytywać z tej samej lokalizacji co skrypt.

Pobieranie łańcuchów połączenia

Podczas pracy z różnymi typami źródeł danych, takich jak np. bazy danych Microsoft Office Access, zabezpieczone hasłem dokumenty Microsoft Office Word lub arkusze kalkulacyjne Office Excel, zazwyczaj należy umożliwić podanie informacji poświadczających uprawnienia dostępu do danego zasobu, ponieważ te typy źródeł nie obsługują personifikacji. Metody przekazywania hasła zostały już opisane i można je łatwo przystosować do tych przypadków. Ponieważ dostęp do tych źródeł danych odbywa się przy użyciu COM, do przekazywania poświadczeń nie można używać klasy `.NET System.Security.SecureString`. Ponadto oznacza to, że nie można używać polecenia `Get-Credential`, ponieważ szyfruje ono hasło przy użyciu klasy `SecureString`.

Dobrym zwyczajem przy pracy z łańcuchami połączeń do źródeł danych jest przechowywanie każdej części łańcucha połączenia w osobnej zmiennej. To ułatwia aktualizowanie poszczególnych składników łańcucha oraz ułatwia zmianę metody wprowadzania danych. Poniżej znajduje się przykład zastosowania tej techniki.

OpenPasswordProtectedExcel.ps1

```
$filename = "C:\fso\TestNumbersProtected.xls"
$updateLinks = 3
$readOnly = $false
$format = 5
$password = "password"
$excel = New-Object -comobject Excel.Application
$excel.visible = $true
$excel.workbooks.open($fileName,$updateLinks,$readOnly,$format,$password) |
Out-Null
```

Kiedy parametr `password` jest dostępny przez inny parametr, to bardzo łatwo można przerobić ten skrypt tak, aby przyjmował dane wejściowe przez wiersz poleceń. W skrypcie *OpenPasswordProtectedWord.ps1* zarówno nazwa pliku, jak i hasło zostały zamienione na parametry wiersza poleceń. Kod odpowiedzialny za otwieranie zabezpieczonego hasłem pliku Office Word zdefiniowano w funkcji i obie wartości przyjmowane przez wiersz poleceń są obowiązkowe. Obecność znacznika `Parameter` sprawia, że skrypt wymaga przynajmniej konsoli Windows PowerShell 2.0 — dlatego dodano instrukcję `#requires -version 2.0`.

OpenPasswordProtectedWord.ps1

```
#requires -version 2.0
Param(
    [Parameter(Mandatory=$true)]
    [string]$fileName,
    [Parameter(Mandatory=$true)]
    [string]$password
)
Function Open-PasswordProtectedDocument($filename,$password)
{
    $Conversion= $false
    $readOnly = $false
    $addRecentFiles = $false
    $doc = New-Object -Comobject Word.Application
    $doc.visible = $true
    $doc.documents.open($filename,$Conversion,$readOnly,$addRecentFiles,$password) |
    out-null
} #end function Open-PasswordProtectedDocument

# *** punkt początkowy skryptu ***

Open-PasswordProtectedDocument -filename $filename -password $password
```

Monitowanie o informacje

Pewnie domyślasz się, że skoro za pomocą polecenia `Read-Host` można prosić o podanie hasła, to można go też używać przed wykonaniem pewnych czynności. Można wyróżnić dwie podstawowe sytuacje, w których potrzebne są dane od użytkownika. W pierwszej użytkownik musi podać jakieś informacje potrzebne skryptowi do zakończenia działania. Technikę tę często stosuje się w skryptach wykonujących kilka różnych czynności i wówczas o podjęciu określonej ścieżki wykonywania decyduje to, co wpisze użytkownik.

Druga sytuacja jest prostsza i polega na tym, że skrypt wyświetla prośbę o potwierdzenie chęci wykonania danej czynności. Monit taki można wyświetlić na przykład przed usunięciem pliku albo rozpoczęciem operacji, która może zająć zasoby komputera na dłuższy czas. Oba te scenariusze są opisane w tym podrozdziale.

Informacje od użytkownika są często potrzebne skryptom po to, by dostosować zwracany wynik. W skrypcie *ReadHostQueryDrive.ps1* polecenie `Read-Host` wyświetla monit o podanie litery dysku, dla którego mają zostać pobrane informacje z WMI. Do obsługi wartości przekazanej przez użytkownika użyto instrukcji `Switch`. Poniżej znajduje się kod źródłowy skryptu *ReadHostQueryDrive.ps1*.

ReadHostQueryDrive.ps1

```
$response = Read-Host "Wpisz literę dysku <c: / d:>"
Switch -regex($response) {
    "C" { Get-WmiObject -class Win32_Volume -filter "driveletter = 'c:'" }
    "D" { Get-WmiObject -class Win32_Volume -filter "driveletter = 'd:'" }
} #end switch
```

Bardziej eleganckim rozwiązaniem, jeśli chodzi o pobieranie danych od użytkownika, jest użycie klasy `$host.ui.PromptForChoice`. Klasa `PromptForChoice` używa opcji wyboru utworzonych przez klasę `System.Management.Automation.Host.ChoiceDescription`. Opcje wyboru definiuje się w postaci tablicy łańcuchów zawierających na początku znak `&`. A ponieważ jest to tablica, każda wartość ma numer w numeracji zaczynającej się od zera, jak widać w poniższym skrypcie *PromptForChoice.ps1*.

PromptForChoice.ps1

```
$caption = "Brak dysku"
$message = "Brak dysku w napędzie. Włóż dysk do napędu D:"
$choices = [System.Management.Automation.Host.ChoiceDescription[]] '
@("&Anuluj", "&Spróbuj ponownie", "&Zignoruj")
[int]$defaultChoice = 2
$choiceRTN = $host.ui.PromptForChoice($caption,$message, $choices,$defaultChoice)

switch($choiceRTN)
{
    0 { "Anulowanie..." }
    1 { "Spróbuj ponownie..." }
    2 { "Ignorowanie..." }
}
```

Wybór najlepszej metody zwracania danych

Technik zwracania danych wyjściowych ze skryptów w konsoli Windows PowerShell 4.0 jest przynajmniej tyle co technik przyjmowania danych do skryptów. Gdyby zsumowano wszystkie możliwości przekazania wyników obliczeń, to ich lista byłaby pewnie nawet dłuższa od listy metod pobierania informacji. W tym podrozdziale znajduje się opis technik wysyłania wyników na ekran, do plików oraz na adres e-mail.

Zapiski praktyka

Shane Hoey

MVP Microsoft PowerShell

Naukę obsługi konsoli Windows PowerShell zacząłem kilka lat temu i dla mnie największą korzyścią z używania tego narzędzia jest to, że znacznie lepiej rozumiem produkt, z którym pracuję.

Aktualnie jestem specjalistą od usługi Microsoft Lync i wszystkie umiejętności dotyczące posługiwania się konsolą Windows PowerShell nabyte kiedyś do dziś nic nie straciły na ważności. Jedyna różnica między przeszłością a teraźniejszością polega na tym, że niektóre czynności stały się moimi nawykami.

Jedna z moich ulubionych sztuczek dotyczących Windows PowerShell jest zarazem jedną z najprostszych czynności, jakie można wykonać przy użyciu tego narzędzia. Jestem integratorem systemów, więc co parę tygodni odwiedzam siedziby różnych klientów, a przypomnienie sobie np. konfiguracji planu wybierania numerów w usłudze Lync każdego klienta po kilku miesiącach nieobecności jest nie lada wyzwaniem. Jednak za pomocą polecenia `Export-Clixml` mogę wykonać kilka poleceń Windows PowerShell, aby wyeksportować całą konfigurację serwera Lync do pliku XML.

Przyjrzyjmy się temu bliżej. Zamiast Lync użyjemy polecenia `Get-Service`. Najpierw przekazujemy obiekt zwrócony przez polecenie `Get-Service` do polecenia `Export-Clixml`, które zapisze go w pliku XML.

```
Get-Service | Export-Clixml -Path c:\scripts\services.xml
```

Po wyeksportowaniu obiektu lub obiektów do pliku XML później będę mógł w razie potrzeby użyć polecenia `Import-Clixml`. Jest to bardzo przydatne ze względu na porównywanie i dokumentację. Ale problem polega na tym, że klienci nie są tak zachwyceni plikami XML jak ja. Dlatego postanowiłem napisać parę skryptów Windows PowerShell pomocnych w procesie sporządzania dokumentacji.

Napisałem skrypt zapisujący bieżący stan środowiska Lync i tworzący dokument Word. Wielką zaletą tego skryptu jest to, że dzięki eksportowi obiektów za pomocą polecenia `Export-Clixml` mogę aktualizować dokumentację w dowolnym czasie oraz mogę porównywać zmiany w konfiguracji przez porównanie plików XML.

A teraz będzie zabawa. Najpierw tworzymy nowy dokument Word. Musimy to zrobić tylko raz, ponieważ będziemy używać tego samego dokumentu przy każdym użyciu skryptu. Oto lista czynności, jakie należy wykonać:

1. Utwórz nowy dokument programu Word.
2. Utwórz nagłówek Moje usługi.
3. Utwórz akapit zawierający opis.
4. Utwórz formant zawartości w formie tekstu sformatowanego, a następnie zmień własności i ustaw tytuł formantu na `RichTextContent1`. Jeśli nie możesz znaleźć formantów zawartości w formie tekstu sformatowanego, sprawdź, czy w programie Word włączony jest pasek narzędzi *Developer*.
5. Zapisz dokument jako `c:\Scripts\PoshDoc.docx`.

Teraz utworzymy dokument PDF, którego zawartość będzie stanowić treść wyeksportowanego wcześniej pliku XML:

```
$service = import-clixml -path .\services.xml
```

```
$word = new-object -ComObject Word.Application
```

```
$worddoc = $Word.Documents.Add("C:\Scripts\PoshDoc.docx")

$word.Visible = $true

($worddoc.Content.ContentControls | where title -eq "RichTextContent1").range.text = foreach
($item in ($service | where {$_.status -match "running"})){$item.DisplayName + "r"}

$wdFormatPDF = [Ref]17

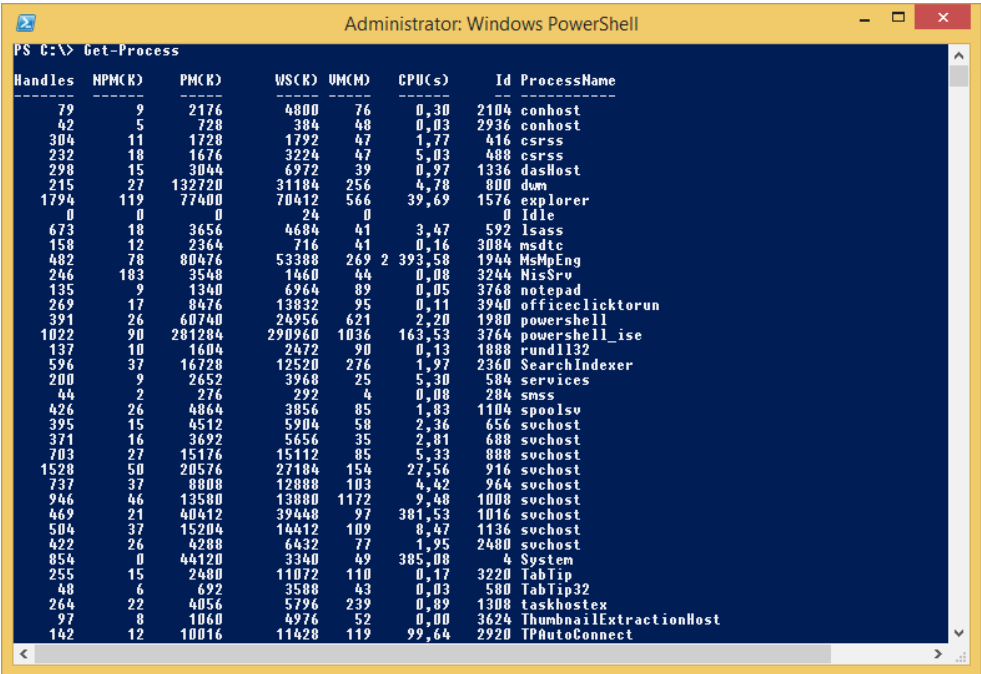
$worddoc.SaveAs([ref]"C:\Scripts\PoshDoc.pdf",$wdFormatPDF)
```

Dokument PDF jest już utworzony. Nic prostszego. Oczywiście przydałoby się poprawić formatowanie i dodać parę formantów, ale na podstawie tego skryptu można utworzyć własne rozwiązanie dokumentacyjne oparte na konsoli Windows PowerShell i programie Word.

Wysyłanie informacji na ekran

Polecenia cmdlet automatycznie wysyłają wyniki na ekran. Jest to jedna z cech, dzięki którym praca z wierszem poleceń konsoli Windows PowerShell jest taka łatwa i wygodna. Jeśli wywołamy polecenie `Get-Process`, automatycznie wyświetli ono wyniki na ekranie.

Do wykonania wielu czynności wystarczy proste wywołanie polecenia cmdlet. Wówczas automatycznie otrzymujemy ładnie sformatowany wynik, który zostaje zaprezentowany na ekranie, jak widać na rysunku 11.3.



RYSUNEK 11.3. Wynik polecenia `Get-Process` jest automatycznie wyświetlany w oknie konsoli Windows PowerShell

To, że dane są tak ładnie formatowane, zawdzięczamy kilku plikom *format.ps1xml* utworzonym przez programistów konsoli Windows PowerShell. Pliki te decydują o sposobie formatowania różnych obiektów na ekranie i można je znaleźć w katalogu instalacyjnym konsoli. Istnieje też zmienna automatyczna o nazwie `$psHOME`, za pomocą której można odwoływać się do katalogu instalacyjnego Windows PowerShell. Aby wyświetlić listę wszystkich plików *format.ps1xml* dostępnych w komputerze, można wywołać polecenie `Get-ChildItem` z parametrem ścieżki reprezentującej każdy plik o nazwie zawierającej słowo `format`. Potem można przekazać otrzymane obiekty `FileInfo` do polecenia `Select-Object` i wybrać własność `name`.

```
PS C:\> Get-ChildItem -Path $psHOME/*format* | Select-Object -Property name
```

```
Name
```

```
----
```

```
certificate.format.ps1xml
dotnettypes.format.ps1xml
filesystem.format.ps1xml
help.format.ps1xml
powershellcore.format.ps1xml
powershelltrace.format.ps1xml
registry.format.ps1xml
```

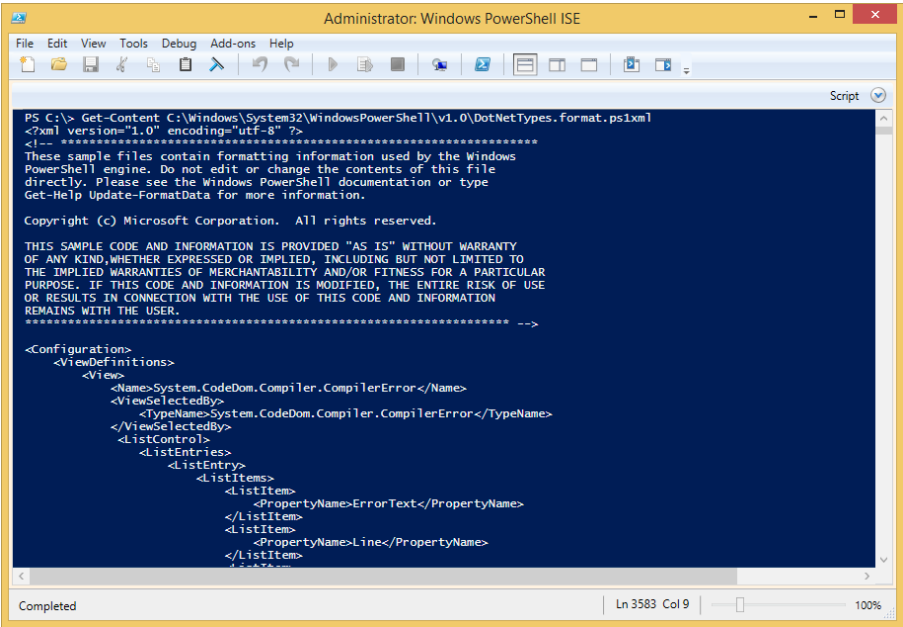
Pliki te są wykorzystywane przez system Windows PowerShell Extended Type System do formatowania prezentacji obiektów na ekranie. Jest to konieczne, ponieważ większość obiektów nie potrafi wyświetlać się samodzielnie. Skoro pliki są w formacie XML, to bez problemu można zmieniać ich zawartość, aby zmodyfikować sposób ich działania. Ale zanim się to zrobi, należy się dobrze zastanowić, ponieważ składnia tych plików jest skomplikowana. Dlatego zanim cokolwiek zrobisz, wykonaj kopię zapasową wszystkich plików. Ponadto pamiętaj, że bezpośrednia ingerencja w te pliki może mieć nieoczekiwane skutki. Istnieje też możliwość napisania własnego pliku *format.ps1xml*, ale to bardzo skomplikowane zadanie.

Do formatowania danych zwracanych przez wiele poleceń zwracających obiekty .NET (np. `Get-Process`, `Get-Service`, `Get-EventLog`) używany jest plik *dotnettypes.format.ps1xml*. Na rysunku 11.4 pokazano fragment jego zawartości. Jest to sekcja odpowiedzialna za sposób prezentacji danych zwracanych przez polecenie `Get-Process`. W sekcji `<TableHeaders>` znajdują się definicje nagłówków kolumn w znacznikach `<TableColumnHeader>`. Każdy z tych znaczników zawiera trzy dodatkowe węzły.

Aby wyświetlić informacje w konsoli, nie trzeba przejmować się formatowaniem plików XML. Można polegać na ustawieniach domyślnych i zdać się na konsolę Windows PowerShell. Aby na przykład wyświetlić w konsoli łańcuch, wystarczy wpisać go w cudzysłowie, jak pokazano poniżej:

```
PS C:\> "to jest łańcuch do wyświetlenia w konsoli"
to jest łańcuch do wyświetlenia w konsoli
PS C:\>
```

Należy tylko pamiętać, że łańcuch przesłany do konsoli zachowuje swój typ, tzn. nadal jest łańcuchem.



RYSUNEK 11.4. Plik dotnettypes.format.ps1xml kontroluje sposób prezentacji informacji zwracanych przez niektóre polecenia cmdlet

PS C:\> "to jest łańcuch do wyświetlenia w konsoli" | Get-Member

TypeName: System.String

| Name | MemberType | Definition |
|----------------|------------|--|
| ----- | ----- | ----- |
| Clone | Method | System.Object Clone(), System.Object ... |
| CompareTo | Method | int CompareTo(System.Object value), i... |
| Contains | Method | bool Contains(string value) |
| CopyTo | Method | void CopyTo(int sourceIndex, char[] d... |
| EndsWith | Method | bool EndsWith(string value), bool End... |
| Equals | Method | bool Equals(System.Object obj), bool ... |
| GetEnumerator | Method | System.CharEnumerator GetEnumerator()... |
| GetHashCode | Method | int GetHashCode() |
| GetType | Method | type GetType() |
| GetTypeCode | Method | System.TypeCode GetTypeCode(), System... |
| IndexOf | Method | int IndexOf(char value), int IndexOf(... |
| IndexOfAny | Method | int IndexOfAny(char[] anyOf), int Ind... |
| Insert | Method | string Insert(int startIndex, string ... |
| IsNormalized | Method | bool IsNormalized(), bool IsNormalize... |
| LastIndexOf | Method | int LastIndexOf(char value), int Last... |
| LastIndexOfAny | Method | int LastIndexOfAny(char[] anyOf), int... |
| Normalize | Method | string Normalize(), string Normalize(... |
| PadLeft | Method | string PadLeft(int totalWidth), strin... |
| PadRight | Method | string PadRight(int totalWidth), stri... |
| Remove | Method | string Remove(int startIndex, int cou... |
| Replace | Method | string Replace(char oldChar, char new... |
| Split | Method | string[] Split(Params char[] separato... |
| StartsWith | Method | bool StartsWith(string value), bool S... |
| Substring | Method | string Substring(int startIndex), str... |

| | | |
|------------------|-----------------------|---|
| ToBoolean | Method | bool IConvertible.ToBoolean(System.IF... |
| ToByte | Method | byte IConvertible.ToByte(System.IForm... |
| ToChar | Method | char IConvertible.ToChar(System.IForm... |
| ToCharArray | Method | char[] ToCharArray(), char[] ToCharAr... |
| ToDateTime | Method | datetime IConvertible.ToDateTime(Syst... |
| ToDecimal | Method | decimal IConvertible.ToDecimal(System... |
| ToDouble | Method | double IConvertible.ToDouble(System.I... |
| ToInt16 | Method | int16 IConvertible.ToInt16(System.IFo... |
| ToInt32 | Method | int IConvertible.ToInt32(System.IForm... |
| ToInt64 | Method | long IConvertible.ToInt64(System.IFor... |
| ToLower | Method | string ToLower(), string ToLower(cult... |
| ToLowerInvariant | Method | string ToLowerInvariant() |
| ToSByte | Method | sbyte IConvertible.ToSByte(System.IFo... |
| ToSingle | Method | float IConvertible.ToSingle(System.IF... |
| ToString | Method | string ToString(), string ToString(Sy... |
| GetType | Method | System.Object IConvertible.GetType(typ... |
| ToUInt16 | Method | uint16 IConvertible.ToUInt16(System.I... |
| ToUInt32 | Method | uint32 IConvertible.ToUInt32(System.I... |
| ToUInt64 | Method | uint64 IConvertible.ToUInt64(System.I... |
| ToUpper | Method | string ToUpper(), string ToUpper(cult... |
| ToUpperInvariant | Method | string ToUpperInvariant() |
| Trim | Method | string Trim(Params char[] trimChars),... |
| TrimEnd | Method | string TrimEnd(Params char[] trimChars) |
| TrimStart | Method | string TrimStart(Params char[] trimCh... |
| Chars | ParameterizedProperty | char Chars(int index) {get;} |
| Length | Property | int Length {get;} |

Użycie jednego z poleceń `out-*`, np. `Out-Host` lub `Out-Default`, sprawia, że łańcuch traci swoją obiektową naturę. To znaczy, że wynik przestaje być egzemplarzem klasy `.NET System.String`.

```
PS C:\> "to jest łańcuch do wyświetlenia w konsoli" | Out-Host | Get-Member
to jest łańcuch do wyświetlenia w konsoli
Get-Member : No object has been specified to the get-member cmdlet.
At line:1 char:66
+ "to jest łańcuch do wyświetlenia w konsoli" | Out-Host | Get-Member <<<<
+ CategoryInfo          : CloseError: (:) [Get-Member], InvalidOperationException
+ FullyQualifiedErrorId : NoObjectInGetMember,Microsoft.PowerShell.Commands.GetMemberCommand
PS C:\>
```

Lepiej unikać używania poleceń `Out-Host` i `Out-Default`, jeśli nie ma dobrego powodu, aby ich użyć, ponieważ powodują stratę obiektu. Jedynym powodem, dla którego można użyć polecenia `Out-Host`, jest jego parametr `-paging`.

```
PS C:\> Get-WmiObject -Class Win32_process | Out-Host -Paging

__GENUS                : 2
__CLASS                : Win32_Process
__SUPERCLASS           : CIM_Process
__DYNASTY              : CIM_ManagedSystemElement
__RELPATH              : Win32_Process.Handle="0"
__PROPERTY_COUNT       : 45
__DERIVATION           : {CIM_Process, CIM_LogicalElement, CIM_ManagedSystemElement}
__SERVER               : VISTA
__NAMESPACE            : root\cimv2
__PATH                 : \\VISTA\root\cimv2:Win32_Process.Handle="0"
Caption                : System Idle Process
CommandLine            :
CreationClassName       : Win32_Process
```

```

CreationDate           :
CSCreationClassName    : Win32_ComputerSystem
CSName                 : VISTA
Description            : System Idle Process
ExecutablePath         :
ExecutionState         :
Handle                 : 0
HandleCount            : 0
InstallDate           :
KernelModeTime        : 151488730096
MaximumWorkingSetSize  :
MinimumWorkingSetSize  :
Name                  : System Idle Process
OSCreationClassName    : Win32_OperatingSystem
OSName                 : Microsoft Windows Vista Business |C:\Windows|\Device
                        \Harddisk0\Partition1
OtherOperationCount    : 0
OtherTransferCount     : 0
PageFaults            : 0
PageFileUsage          : 0
ParentProcessId        : 0
PeakPageFileUsage      : 0
PeakVirtualSize        : 0
PeakWorkingSetSize     : 0
Priority               : 0
PrivatePageCount       : 0
<SPACE> next page; <CR> next line; Q quit

```

Jeśli ktoś nie używa parametru `-paging`, to nie ma żadnej korzyści z używania polecenia `Out-Host`. Z punktu widzenia tego, co zostanie wyświetlone na ekranie, poniższe dwa polecenia są równoważne:

```

Get-Process
Get-Process | Out-Host
Get-Process | Out-Default

```

W istocie polecenia `Out-Host` i `Out-Default` w większości systemów działają tak samo, ponieważ polecenie `Out-Host` jest często domyślnym mechanizmem wyświetlania danych. Jedynym powodem, dla którego ktoś mógłby użyć polecenia `Out-Default`, jest przewidywanie, że w przyszłości może zmienić się domyślny mechanizm wyświetlania danych, a w związku z tym chęć uniknięcia konieczności modyfikowania skryptu. Polecenie `Out-Default` zawsze używa domyślnego mechanizmu, którym może, ale nie musi być `host`.

Zapiski praktyka

Mark Minasi

MVP Microsoft Directory Services

Dwadzieścia jeden lat temu firma Microsoft wydała swój pierwszy stos TCP/IP o nazwie kodowej *Wolverine*. Była to implementacja beta działająca w systemie Windows for Workgroups 3.11.

Po zainstalowaniu odkryłem, że zawiera niezwykle przydatne i zwarte narzędzie wiersza poleceń o nazwie `ipconfig`. Do dziś nie ma tygodnia, abym nie użył do czegoś tego starego, dobrego znajomego.

Hm.. W poprzednim zdaniu chyba jednak trochę przesadziłem pod względem temperatury naszych relacji w ciągu ostatnich mniej więcej pięciu lat. Jak niektórzy starzy przyjaciele, również narzędzie `ipconfig` **zmieniło** się z wiekiem i niestety niekoniecznie na lepsze. Przybrało trochę na wadze i prawdę mówiąc, zaczęło fiksować. Zwięzłość, która cechowała je w młodości, gdzieś się rozmyła, ponieważ nie wiem i nie chcę wiedzieć, co to jest DHCP DUID i DHCP IAID. A te wszystkie karty tunelowe? Nie chcę się rozwodzić, ale dobrze pamiętam, jak po raz pierwszy w 2006 roku wpisałem polecenie `ipconfig/all` w wersji beta systemu Windows Vista. Z przerażeniem patrzyłem, jak mój stary przyjaciel `ipconfig` produkuje tyle danych, że nie da się ich zmieścić na ekranie monitora. Był to dzień, w którym wykryto wielkie zakłócenie mocy, tak jakby miliony informatyków jednocześnie wydało jęk zawodu.

Przez wiele lat żartobliwie sugerowałem, aby firma Microsoft dodała do narzędzia `ipconfig` parametr `/noEXTRAS`, ale jej przedstawiciele pozostali głusi na moje wołania. Od kilku lat obiecuję sobie, że przypomnę sobie wiadomości z programowania w języku C++, poczytam książki na temat sieciowych interfejsów API Win32 i napiszę nowe narzędzie `ipconfig`, ale chyba nie znajdę na to czasu.

Wtedy pojawił się system Windows 8 wraz z ponad tysiącem poleceń Windows PowerShell, z których kilkadziesiąt dotyczy konfiguracji sieci i IP. W związku z tym pewnej chłodnej jesiennej niedzieli w 2012 roku usiadłem wygodnie, aby sprawdzić, jak działają te wszystkie nowe polecenia dotyczące sieci i jakie informacje można przy ich użyciu uzyskać. Kwakałem niczym Sknerus McKwacz w swoim skarbcu, gdy kolejne polecenia zwracały coraz to więcej informacji.

Ale moje upojenie tym bogactwem szybko minęło, gdy przypominałem sobie o starym przyjacielu `ipconfig`. Głos w mojej głowie podpowiadał mi: „Mamy odpowiednią technologię. Możemy ją poprawić... na lepsze”. Oczywiście tak naprawdę to słyszałem odgłosy z telewizji, którą oglądał mój współlokator, ale to wystarczyło, abym poświęcił kilka godzin na znalezienie poleceń, które dostarczają takich samych informacji dotyczących sieci co `ipconfig`. Potem jeszcze sprawdziłem, jak się tworzy własne polecenia. Tak powstało polecenie `Get-Ipinfo`. (Można je znaleźć na stronie <http://www.minasi.com/newsletters/nws1210.htm>). Może nie jest najelegantszym i najbardziej profesjonalnie napisanym poleceniem świata, ale działa, a jego budowa zajęła mi kilka godzin, nie miesięcy.

Jak powiedziałby Robert Kennedy, gdyby był specjalistą od Windows PowerShell: „Są tacy, którzy szukają słabych stron narzędzi administracyjnych systemu Windows i pytają, skąd się wzięły. Mnie marzy się szybkie budowanie lepszych narzędzi i pytam: Czemu nie?”.

Ty też tak powinienes. Naucz się obsługi konsoli Windows PowerShell i zbieraj plony.

Wysyłanie wyników do pliku

Jeśli wynik ma zostać wyświetlony na ekranie, wystarczy wykonać polecenie, aby domyślnie wysłać wynik do okna konsoli, jak pokazano poniżej:

```
PS C:\> Get-WmiObject -Class Win32_Bios

SMBIOSBIOSVersion : A01
Manufacturer       : Dell Computer Corporation
Name               : Default System BIOS
SerialNumber       : 9HQ1S21
Version            : DELL - 6
```

```
PS C:\>
```

Jeśli informacje trzeba zapisać w pliku tekstowym, można użyć strzałki przekierowującej.

```
PS C:\> Get-WmiObject -Class Win32_Bios >c:\fso\bios.txt
PS C:\>
```

Wadą tego rozwiązania jest brak jakiegokolwiek potwierdzenia wykonania operacji oraz informacji na temat zawartości pliku. Wprawdzie można użyć polecenia `Out-File`, jak pokazano poniżej, ale to i tak nie zmienia faktu, że konsola nie wyświetli żadnego potwierdzenia.

```
PS C:\> Get-WmiObject -Class Win32_Bios | Out-File -FilePath C:\fso\bios.txt
PS C:\>
```

Aby upewnić się, że w pliku zostały zapisane odpowiednie informacje, można sprawdzić jego zawartość za pomocą polecenia `Get-Content`. Należy podkreślić, że w tym przypadku nie przekazujemy potokowo informacji z polecenia `Out-File` do polecenia `Get-Content` — średnik oznacza początek nowego polecenia i jego użycie jest równoznaczne z rozpoczęciem nowej linii.

```
PS C:\> Get-WmiObject -Class Win32_Bios | Out-File -FilePath C:\fso\bios.txt ;
Get-Content -Path C:\fso\bios.txt
```

```
SMBIOSBIOSVersion : A01
Manufacturer       : Dell Computer Corporation
Name               : Default System BIOS
SerialNumber       : 9HQ1S21
Version            : DELL - 6
```

Ponieważ wiesz już, że strzałka przekierowania jest równoznaczna z poleceniem `Out-File`, na potrzeby tego przykładu możesz nią zastąpić to polecenie. Ponadto polecenie można jeszcze bardziej skrócić, wpisując alias `cat` zamiast długawej nazwy `Get-Content`.

```
PS C:\> Get-WmiObject -Class Win32_Bios > C:\fso\bios.txt ; cat C:\fso\bios.txt
SMBIOSBIOSVersion : A01
Manufacturer       : Dell Computer Corporation
Name               : Default System BIOS
SerialNumber       : 9HQ1S21
Version            : DELL - 6
```

Zamieniając nazwę polecenia `Get-WmiObject` na alias i opuszczając nazwę parametru `-class`, można jeszcze bardziej skrócić polecenie.

```
PS C:\> gwmi Win32_Bios > C:\fso\bios.txt ; cat C:\fso\bios.txt
SMBIOSBIOSVersion : A01
Manufacturer      : Dell Computer Corporation
Name              : Default System BIOS
SerialNumber      : 9HQ1S21
Version           : DELL - 6
```

Wysyłanie danych równocześnie na ekran i do pliku

Mamy już związane polecenie służące do wysłania danych z polecenia do pliku tekstowego, a następnie wyświetlenia ich w oknie konsoli. Jest to wprowadzić jakieś rozwiązanie, ale wygodniej byłoby użyć pojedynczego polecenia, które robi to samo. Takie polecenie nawet istnieje. Nazywa się Tee-Object i rozdzwaja wynik innego polecenia na dwa kanały — ekran i plik. Najczęściej dane trzeba wysłać do pliku i konsoli. Aby to zrobić, należy w parametrze -FilePath podać pełną ścieżkę do pliku. Jak pokazano poniżej, polecenie Tee-Object ma jeszcze wiele innych przełączników i parametrów oraz trzy różne zestawy parametrów:

```
Tee-Object [-FilePath] <string> [-InputObject <psoobject>] [-Append] [<CommonParameters>]
```

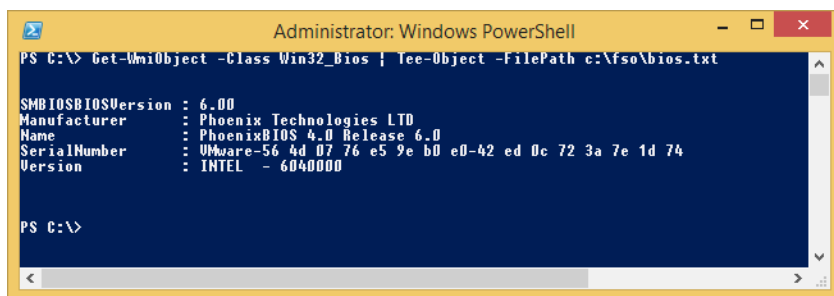
```
Tee-Object -LiteralPath <string> [-InputObject <psoobject>] [<CommonParameters>]
```

```
Tee-Object -Variable <string> [-InputObject <psoobject>] [<CommonParameters>]
```

Wróćmy do przykładu. Strzałkę przekierowania (lub polecenie Out-File) i polecenie Get-Content (którego aliasem jest cat) można zastąpić poleceniem Tee-Object. Poniżej znajduje się odpowiednio zmieniony kod:

```
Get-WmiObject -Class Win32_Bios | Tee-Object -FilePath c:\fso\bios.txt
```

Efekt wykonania tego polecenia jest taki, jak widać na rysunku 11.5.



RYСУNEK 11.5. Polecenie Tee-Object rozdziela dane wyjściowe na plik tekstowy i konsolę Windows PowerShell

Używając polecenia Tee-Object, należy tylko pamiętać, że jeśli wskazany plik tekstowy istnieje, to zostanie nadpisany. A jeśli wskazanego pliku nie ma, to zostanie utworzony — nie dotyczy to jednak folderu. Jeśli ktoś spróbuje zapisać za pomocą polecenia Tee-Object dane w nieistniejącym folderze, zostanie wyświetlona informacja o nieistniejącej ścieżce.

```
PS C:\> Get-WmiObject -class Win32_Bios | Tee-Object -FilePath C:\fso5\bios.txt
out-file : Nie można odnaleźć części ścieżki „C:\fso5\bios.txt”.
PS C:\>
```

Ponadto za pomocą polecenia `Tee-Object` można wysłać wyniki innego polecenia do zmiennej. Może to być wygodny sposób na zapisanie danych do późniejszego użytku w skrypcie. Poniżej znajduje się przykład zapisania wyników polecenia w zmiennej, a następnie wyświetlenia ich bez użycia polecenia `Tee-Object`.

```
PS C:\> $bios = Get-WmiObject -class Win32_Bios
PS C:\> $bios
```

```
SMBIOSBIOSVersion : A01
Manufacturer      : Dell Computer Corporation
Name              : Default System BIOS
SerialNumber      : 9HQ1S21
Version           : DELL - 6
```

```
PS C:\>
```

Poniżej natomiast pokazano składnię potrzebną do zapisania w zmiennej wyników z potoku za pomocą polecenia `Tee-Object`.

```
Tee-Object [-InputObject <PSObject>] -Variable <String> [-Verbose] [-Debug] [-ErrorAction
<ActionPreference>] [-ErrorVariable <String>] [-OutVariable <String>] [-OutBuffer <Int32>]
```

Aby zapisać wynik polecenia `Get-WmiObject -Class Win32_Bios` w zmiennej o nazwie `$bios`, można użyć poniższego polecenia:

```
PS C:\> Get-WmiObject -class Win32_Bios | Tee-Object -Variable bios
```

```
SMBIOSBIOSVersion : A01
Manufacturer      : Dell Computer Corporation
Name              : Default System BIOS
SerialNumber      : 9HQ1S21
Version           : DELL - 6
```

Używając parametru `variable` z poleceniem `Tee-Object`, należy pamiętać, że przed nazwą zmiennej nie trzeba wpisywać znaku dolara. Pod tym względem polecenie to jest podobne do polecenia `New-Variable`.

Aby wyświetlić zawartość zmiennej `$bios`, należy wpisać `$bios` w wierszu poleceń konsoli Windows PowerShell, jak pokazano poniżej:

```
PS C:\> $bios
```

```
SMBIOSBIOSVersion : A01
Manufacturer      : Dell Computer Corporation
Name              : Default System BIOS
SerialNumber      : 9HQ1S21
Version           : DELL - 6
PS C:\>
```

Jedną z największych zalet polecenia `Tee-Object` jest to, że przekazuje też obiekty przez potok. To znaczy, że nie jesteśmy skazani na domyślną prezentację informacji zwróconych przez poprzednie polecenie, np. `Get-WmiObject`. Możemy zapisać obiekt w zmiennej `$bios`, a następnie wyświetlić z niego tylko wartość własności `name`.

```
PS C:\> Get-WmiObject -class Win32_Bios | Tee-Object -Variable bios |
select name
```



```
name
----
Default System BIOS
```

Aby pobrać obiekt ze zmiennej, należy jeszcze raz wpisać w wierszu poleceń \$bios lub użyć tej zmiennej w dowolnym miejscu skryptu.

```
PS C:\> $bios
```

```
SMBIOSBIOSVersion : A01
Manufacturer       : Dell Computer Corporation
Name               : Default System BIOS
SerialNumber       : 9HQ1S21
Version            : DELL - 6
```

Nie jest też nigdzie powiedziane, że polecenia Tee-Object można używać tylko z poleceniami Windows PowerShell. Równie dobrze działa ono w połączeniu ze zwykłymi narzędziami wiersza poleceń, czego dowodem jest poniższy przykład, w którym wynik polecenia ping jest wyświetlany w konsoli i zapisywany w zmiennej \$ping.

```
PS C:\> ping berlin | Tee-Object -Variable ping
Pinging Berlin.nwtraders.com [192.168.2.1] with 32 bytes of data:
Reply from 192.168.2.1: bytes=32 time=11ms TTL=128
Reply from 192.168.2.1: bytes=32 time=1ms TTL=128
Reply from 192.168.2.1: bytes=32 time=1ms TTL=128
Reply from 192.168.2.1: bytes=32 time=1ms TTL=128
Ping statistics for 192.168.2.1:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
    Approximate round trip times in milli-seconds:
        Minimum = 1ms, Maximum = 11ms, Average = 3ms
PS C:\>
```

Zaletą tej techniki jest to, że teraz za pomocą polecenia Select-String można szybko przeszukać zawartość zmiennej, aby znaleźć potrzebne informacje. Jeśli kogoś interesuje tylko to, ile z wysłanych pakietów zostało odebranych, to może przesłać dane ze zmiennej \$ping do polecenia Select-String.

```
PS C:\> $ping | Select-String packet
Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
PS C:\>
```

Zapiski praktyka

Praca z wyjściem

Dave Schwinn, starszy konsultant

Full Service Networking

W konsoli Windows PowerShell 4.0 dane wyjściowe skryptów i powłoki można obsługiwać na wiele różnych sposobów. Na przykład bardzo łatwo można wysłać informacje do pliku tekstowego, bazy danych, na ekran, do pliku HTML lub XML albo CSV. Wybór należy do mnie i zależy od tego, co chcę zrobić z otrzymanymi informacjami.

Jedną z moich ulubionych czynności jest eksportowanie wyników do formatu XML. Najpierw wykonuję polecenie, potem jego wynik eksportuję do pliku XML, a następnie wyświetlam zawartość tego pliku na ekranie. Podczas gdy dla niektórych format ten może być nieczytelny, ja go lubię, bo wyraźnie w nim widać relacje między różnymi elementami danych.

W typowy dzień w pracy dane na ekranie wyświetlam zazwyczaj w formie tabeli. Dzięki temu szybko znajduję powiązane ze sobą elementy w długich listach danych. Ten sposób wyświetlania warto rozważyć, gdy używa się takich poleceń, jak np. `Get-Process` czy `Get-Service`, które wyświetlają tabele domyślnie. Weźmy na przykład klasę WMI `Win32_LogicalDisk`. Przy użyciu poniższego polecenia szybko znajdę potrzebne mi informacje:

```
PS C:\> Get-WmiObject Win32_LogicalDisk |
Format-Table name, size, freespace -AutoSize
name                size    freespace
----                -
C:                  158391595008 15872155648
E:
S:                  1647308800 1554030592
```

Do wysyłania informacji do drukarki lubię używać polecenia `ConvertTo-Html`, w którym mogę zdefiniować różne szczegóły sprawiające, że raport wygląda profesjonalnie. Ale często też wysyłam dane do innego polecenia i kontynuuję pracę w wierszu poleceń.

Czasami używam też poleceń Windows PowerShell w środowisku Microsoft Visual Studio. Często piszę polecenia w projektach Microsoft Visual Studio, które wywołują Windows PowerShell w celu pobrania danych i zwrócenia ich do mojej aplikacji. Łatwiej jest mi pobrać obiekty WMI za pomocą Windows PowerShell i użyć danych zwróconych przez PowerShell w aplikacji, niż bezpośrednio wywołać klasę WMI z platformy .NET. Później przesyłam wyniki w postaci zbioru danych i przetwarzam kolumny dla mojej aplikacji.

Dość często używam polecenia `Export-CSV`, ponieważ mogę bez problemu otwierać pliki w programie Excel i wykonywać różne zaawansowane działania na danych oraz tworzyć wykresy i raporty. Jako że pracuję dla dostawcy rozwiązań firmy Microsoft, wielu moich klientów używa modelu licencjonowania Microsoftu polegającego w zasadzie na wynajmowaniu oprogramowania od tej firmy. Klienci ci zawsze mają prawo do uaktualniania swoich programów w dowolnym momencie i łatwo mogą przewidzieć w budżecie wydatki na oprogramowanie. Problem w tym, że co miesiąc muszą ujawniać firmie Microsoft liczbę stanowisk, na których używany jest dany produkt. Za pomocą polecenia `Get-Mailbox` konsoli Windows PowerShell wysyłam zapytanie do serwera Microsoft Exchange Server klienta, aby pobrać listę wszystkich skrzynek pocztowych używanych na tym serwerze. Następnie eksportuję tę listę do pliku CSV i przekazuję ją potokowo do polecenia `Send-MailMessage`. Raport idzie bezpośrednio do pracownika z działu zakupów, który otwiera plik w Excelu i sprawdza, za ile stanowisk należy zapłacić w tym miesiącu. W czasach gdy nie było konsoli Windows PowerShell, taka automatyzacja była po prostu niemożliwa.

Wysyłanie wyników na adres e-mail

Często wynik skryptu trzeba przesłać bezpośrednio na określony adres e-mail. Kiedyś, aby to zrobić, należałoby napisać skomplikowaną funkcję i liczyć, że wszystko się uda. Praca była łatwiejsza, zanim spamerzy nie spowodowali zaostreżenia warunków wysyłania wiadomości e-mail ze skryptów. Wirusy rozsyłane pocztą przyczyniły się do powstania dodatkowej warstwy uwierzytelniania i znacznie wszystko utrudniły.

W konsoli Windows PowerShell 2.0 wprowadzono polecenie `Send-MailMessage` służące do wysyłania wiadomości e-mail ze skryptów. Czasami działa ono bez definiowania jakichkolwiek ustawień, a czasami trzeba nadać kontu użytkownika używanemu do uruchamiania skryptów uprawnienia do wysyłania wiadomości e-mail ze skryptów.

Wysyłanie wyników z funkcji

Gdy wywołana funkcja kończy działanie, zwraca wynik do tego kodu, który ją wywołał. Wielu programistów innych języków skryptowych rozpoczynających naukę Windows PowerShell nie rozumie tego. Jeśli uruchomisz skrypt *AddOne.ps1*, w oknie konsoli pojawi się liczba 6. Problemy wynikają z tego, że dane są zwracane z wiersza kodu, który wywołał funkcję, a nie z samej funkcji, czego wielu programistów się nie spodziewa. W większości przypadków wynik dodawania dwóch liczb jest zwracany z wiersza, który wykonał to działanie.

```
PS C:\> $int = 5
PS C:\> $int + 1
6
PS C:\>
```

Można więc oczekiwać, że liczba 6 pochodzi z wnętrza funkcji *AddOne*, a nie spoza niej. Poniżej znajduje się kod źródłowy skryptu *AddOne.ps1* zawierającego funkcję *AddOne*.

AddOne.ps1

```
Function AddOne($int)
{
    $int + 1
}

AddOne(5)
```

Aby pokazać, skąd przychodzą dane, można zmodyfikować skrypt tak, aby zapisywał wynik wywołania funkcji w zmiennej. Potem za pomocą polecenia `Get-Member` wyświetlimy zwróconą informację, jak pokazano w skrypcie *AddOne1.ps1*.

AddOne1.ps1

```
Function AddOne($int)
{
    $int + 1
}

$number = AddOne(5)
$number | get-member
'Wartość zmiennej $number: ' + $number
```

Po uruchomieniu skryptu *AddOne1.ps1* zauważysz, że informacje są zwracane do kodu, który wywołał funkcję. W pierwszej linijce po wywołaniu funkcji obiekt zapisany w zmiennej \$number jest typu System.Int32. Natomiast za poleceniem Get-Member wartość zmiennej \$number wynosi 6. Wartość 5 z wnętrza funkcji AddOne nie została wyświetlona.

TypeName: System.Int32

| Name | MemberType | Definition |
|------------------------------|------------|--|
| ---- | ----- | ----- |
| CompareTo | Method | System.Int32 CompareTo(Object value), System.Int32 Co... |
| Equals | Method | System.Boolean Equals(Object obj), System.Boolean Equ... |
| GetHashCode | Method | System.Int32 GetHashCode() |
| GetType | Method | System.Type GetType() |
| GetTypeCode | Method | System.TypeCode GetTypeCode() |
| ToString | Method | System.String ToString(), System.String ToString(Stri... |
| Wartość zmiennej \$number: 6 | | |

Gdy używa się poleceń typu Write-Host wewnątrz funkcji, omija się proces zwrotu wartości będący podstawową cechą funkcji. Poniżej pokazano przykład użycia tego polecenia wewnątrz funkcji w skrypcie *AddOne2.ps1*.

```
addOne2.ps1

Function AddOne($int)
{
    Write-Host $int + 1
}

$number = AddOne(5)
$number | get-member
'Wartość zmiennej $number: ' + $number
```

Po uruchomieniu tego skryptu zauważysz, że z wnętrza funkcji nic nie zostanie zwrócone. Zmienna \$number nie zawiera teraz obiektu.

```
5 + 1
Get-Member : No object has been specified to get-member.
At C:\Documents and Settings\ed\Local Settings\Temp\tmp6.tmp.ps1:9 char:21
+ $number | get-member <<<<
Wartość zmiennej $number:
```

Unikanie wprowadzania danych do zmiennej globalnej

Oprócz omijania mechanizmu zwrotu z funkcji za pomocą polecenia Write-Host można też zapisywać wyniki funkcji w zmiennych. Problem z zapisywaniem wyników funkcji w zmiennej znajdującej się w tej funkcji polega na tym, że zmienna taka jest niedostępna poza funkcją, co zilustrowano w pokazanym poniżej skrypcie *AddOne3.ps1*.

```
AddOne3.ps1

Function AddOne($int)
{
    $number = $int + 1
}
```

```
$number = AddOne(5)
$number | get-member
'Wartość zmiennej $number: ' + $number
```

Gdy skrypt *AddOne3.ps1* zostanie uruchomiony, w zmiennej `$number` nie będzie żadnego obiektu, ponieważ zmienna ta jest niedostępna poza funkcją `AddOne`.

```
Get-Member : No object has been specified to get-member.
At C:\Documents and Settings\ed\Local Settings\Temp\tmp9.tmp.ps1:9 char:21
+ $number | get-member <<<<
Wartość zmiennej $number:
```

Jedną z technik, które czasami stosuje się, aby dostarczyć wartość zmiennej z wnętrza funkcji do skryptu wywołującego, jest określenie zakresu zmiennej, jak pokazano w poniższym skrypcie *AddOne4.ps1*.

AddOne4.ps1

```
Function AddOne($int)
{
    $global:number = $int + 1
}

AddOne(5)
$global:number | get-member
'Wartość zmiennej $global:number: ' + $global:number
```

Ze zmiennymi dodawanymi do zakresu globalnego jest jednak jeden problem — nie przestają istnieć po zakończeniu działania skryptu. Dopóki konsola Windows PowerShell jest otwarta i programista jawnie nie usunie zmiennej globalnej, zmienna ta będzie cały czas dostępna. To oznacza, że będzie można jej używać także w innych skryptach i ogólnie w konsoli. To nie zawsze musi być problem, ale może wprowadzać w błąd inne skrypty, w których używana jest już zmienna o takiej samej nazwie. Jednym ze sposobów na dowiedzenie się, czy zmienna przetrwała, jest sprawdzenie dysku zmiennych.

```
TypeName: System.Int32
```

| Name | MemberType | Definition |
|-------------|------------|--|
| CompareTo | Method | System.Int32 CompareTo(Object value), System.Int32 Co... |
| Equals | Method | System.Boolean Equals(Object obj), System.Boolean Equ... |
| GetHashCode | Method | System.Int32 GetHashCode() |
| GetType | Method | System.Type GetType() |
| GetTypeCode | Method | System.TypeCode GetTypeCode() |
| ToString | Method | System.String ToString(), System.String ToString(Stri... |

Wartość zmiennej `$global:number`: 6

```
PS C:\data\PowerShellBestPractices\Scripts\Chapter12> Get-Item Variable:\number
Name                                     Value
----                                     -
number                                 6
```

Zmienną globalną można usunąć w ostatnim wierszu skryptu za pomocą polecenia `Remove-Variable`, ale lepszym rozwiązaniem jest zdefiniowanie zakresu skryptowego zamiast globalnego.

Zmienna skryptowa jest dostępna na zewnątrz funkcji podczas działania skryptu. Gdy skrypt kończy działanie, zmienna ta zostaje usunięta. W przedstawionym poniżej skrypcie *AddOne5.ps1* pokazano przykład użycia zmiennej o skryptowym zakresie dostępności.

```
AddOne5.ps1

Function AddOne($int)
{
    $script:number = $int + 1
}

AddOne(5)
$script:number | get-member
'Wartość zmiennej $script:number: ' + $script:number
```

Po uruchomieniu skryptu *AddOne5.ps1* wartość zmiennej *\$number* staje się dostępna na zewnątrz funkcji. Gdy po zakończeniu działania skryptu ktoś spróbuje odczytać wartość tej zmiennej za pomocą polecenia *Get-Item*, konsola zgłosi błąd.

```
TypeName: System.Int32
```

```
Name      MemberType Definition
----      -
CompareTo  Method      System.Int32 CompareTo(Object value), System.Int32 Co...
Equals     Method      System.Boolean Equals(Object obj), System.Boolean Equ...
GetHashCode Method      System.Int32 GetHashCode()
GetType    Method      System.Type GetType()
GetTypeCode Method      System.TypeCode GetTypeCode()
ToString   Method      System.String ToString(), System.String ToString(Stri...
Wartość zmiennej $script:number: 6

PS C:\data\PowerShell\BestPractices\Scripts\Chapter12> Get-Item variable:number
Get-Item : Cannot find path 'number' because it does not exist.
At line:1 char:9
+ Get-Item <<<< variable:number
```

Przypisywanie zmiennych globalnych do przestrzeni nazw

Jednym ze sposobów na ochronę konsoli Windows PowerShell przed zarzuceniem globalnymi zmiennymi tworzonymi w skryptach jest użycie przestrzeni nazw. Dzięki temu zmienna jest cały czas dostępna, a jednocześnie nie wchodzi w konflikty nazewnicze z innymi zmiennymi o takiej samej nazwie. Aby przypisać zmienną globalną do osobnej przestrzeni nazw, należy użyć znaku dolara, klamry i słowa kluczowego *Global* z dwukropkiem, po którym należy wpisać nazwę przestrzeni nazw. Natomiast nazwę zmiennej oddziela się od nazwy przestrzeni nazw kropką.

```
${Global:AddOne6.number} = $int + 1
```

Aby odwołać się do zmiennej globalnej przypisanej do przestrzeni nazw, należy użyć znaku dolara, klamry oraz kropki do rozdzielenia nazwy przestrzeni nazw i nazwy zmiennej. Nie trzeba już dodawać słowa kluczowego *Global*.

```
${AddOne6.number}
```

Przykład użycia globalnej zmiennej przypisanej do przestrzeni nazw znajduje się w skrypcie *AddOne6.ps1*, którego kod pokazano poniżej.

AddOne6.ps1

```
Function AddOne($int)
{
    ${Global:AddOne6.number} = $int + 1
}

AddOne(5)
${AddOne6.number} | get-member
'Wartość zmiennej ${AddOne6.number}: ' + ${AddOne6.number}
```

Po uruchomieniu skryptu *AddOne6.ps* można odwoływać się do zmiennej globalnej za pomocą kropki.

TypeName: System.Int32

| Name | MemberType | Definition |
|-------------|------------|--|
| CompareTo | Method | System.Int32 CompareTo(Object value), System.Int32 Co... |
| Equals | Method | System.Boolean Equals(Object obj), System.Boolean Equ... |
| GetHashCode | Method | System.Int32 GetHashCode() |
| GetType | Method | System.Type GetType() |
| GetTypeCode | Method | System.TypeCode GetTypeCode() |
| ToString | Method | System.String ToString(), System.String ToString(Stri... |

Wartość zmiennej \${AddOne6.number}: 6

```
PS C:\data\PowerShellBestPractices\Scripts\Chapter12> Get-Item Variable:\AddOne6.number
Name                                     Value
----                                     -
AddOne6.number                          6
```

Zapiski praktyka

Konsola Windows PowerShell zmusza do zmiany sposobu myślenia

Richard Norman, Senior Premier Field Engineer

Microsoft Corporation

Jako inżynier zajmujący się kluczowymi technologiami oraz produktami firmy Microsoft dużo rozmawiam z klientami na tematy związane z posługiwaniem się konsolą Windows PowerShell. Zawsze powtarzam, że podstawą jest zmiana sposobu myślenia. Dotyczy to szczególnie osób, które wcześniej programowały w języku VBScript.

Konsola Windows PowerShell stwarza wiele nowych możliwości, ale nie zrywa z przeszłością. Używając jej, należy pamiętać, że w istocie praca polega na posługiwaniu się obiektami. Obiekty mają własności i metody, których można używać w sposoby niedostępne w języku VBScript. Kiedyś wyniki zawsze były zwracane w formie tekstowej. Aby zrobić z nimi cokolwiek innego, trzeba było je przetwarzać za pomocą dodatkowych narzędzi. W konsoli Windows PowerShell

otrzymuje się coś więcej niż zwykły tekst — obiekty, którymi można posługiwać się w całkiem nowe sposoby. Jeśli ktoś jest przyzwyczajony do pracy z tekstem, to nadal może tak pracować, ale wtedy pozbawia się niektórych możliwości. Kolejnym krokiem w procesie zmiany sposobu myślenia jest nauczenie się wykorzystywania własności obiektów. Za ich pomocą można na przykład zdobyć listę plików, które zostały zmodyfikowane w ciągu ostatniego tygodnia:

```
dir | where-object {$_.lastwritetime -ge (get-date).adddays(-7)}
```

Powinno się też porzucić przekonanie, że potok działa na listach (zwanych tablicami) obiektów z wiersza poleceń. Obiekty te przesyła się kolejno z jednego polecenia do drugiego, potem do następnego itd. Ta lista obiektów jest kluczem do zrozumienia sposobu działania konsoli Windows PowerShell. Wielkie możliwości, jakie dają polecenia cmdlet, można docenić, używając obiektów w skryptach. Sposób działania konsoli sprawia, że możliwe jest leniwe przetwarzanie skryptów. Oznacza to, że podczas gdy jedno polecenie wykonuje obliczenia, konsola może rozpocząć przetwarzanie wyników, zanim to pierwsze polecenie skończy działanie. Zazwyczaj dzieje się to tak szybko, że użytkownik tego nie zauważa, ale w przypadku dużych plików i list może to być bardzo wygodne rozwiązanie.

W końcu warto wiedzieć, jak wykorzystać cechy interpretowanych łańcuchów. W konsoli Windows PowerShell zmienne znajdujące się w łańcuchach oznaczonych cudzysłowami podwójnymi są normalnie interpretowane. (Przed nazwą zmiennej powinien znajdować się znak dolara). Dzięki temu można wyeliminować starą technikę konkatencji łańcuchów oraz używać zmiennych bezpośrednio w łańcuchach.

Jeśli ktoś założy, że Windows PowerShell jest jak każdy inny język skryptowy, to może mieć kłopoty. Wprawdzie kod źródłowy skryptów można pisać w podobny sposób jak w języku VBScript, ale występują też ważne różnice. Na przykład: jeśli trzeba przefiltrować dane na podstawie daty lub jakiegoś innego kryterium, w języku VBScript trzeba napisać sporo linijek kodu. Natomiast w Windows PowerShell filtry mieszczą się w jednym wierszu kodu:

```
dir | where-object {$_.lastwritetime -ge (get-date).adddays(-14)}
```

Innym przykładem może być tak prosta czynność, jak zamiana znacznika czasu z systemu DNS na datę i godzinę. W języku VBScript trzeba w tym celu wykonać trochę obliczeń i napisać parę funkcji. Natomiast w Windows PowerShell dzięki obiektom odpowiedni kod mieści się w jednym wierszu:

```
get-date "1/1/1601 12:00 am GMT").addhours($timestamp)
```

Obiektowa natura języka Windows PowerShell sprawia, że niektóre rzeczy są w nim ułatwione, a inne utrudnione. Niektóre funkcje języka VBScript mają bezpośrednie odpowiedniki, podczas gdy inne w świecie Windows PowerShell są bardziej skomplikowane. Dotyczy to na przykład polecenia `DMD.exe DIR /a:d`. Zwraca ono listę folderów znajdujących się w bieżącym katalogu. W Windows PowerShell analogiczne polecenie wyglądałoby tak: `DIR | Where-Object {$_.PSISContainer}`. Jest trochę dłuższe, ale korzyści uzyskiwane w innych obszarach rekompensują tę niedogodność z nawiązką.

Ponadto niektóre często używane procedury, np. polecenie CLS narzędzia `CMD.exe`, mają w Windows PowerShell swoje odpowiedniki w postaci nowych funkcji — w tym przypadku `Clear-Host`. Znalezienie informacji, do czego służy to polecenie, pozostawiam jako zadanie domowe. Chociaż implementacja tej funkcji jest znacznie bardziej skomplikowana niż starszego polecenia CLS.

Często spotykam się z twierdzeniem, że Windows PowerShell udostępnia potęgę narzędzi programistycznej platformy .NET w wierszu poleceń używanym najczęściej przez administratorów. Jest to skomplikowana technologia i do niektórych rzeczy trzeba po prostu się przyzwyczaić. Na przykład przy użyciu platformy .NET bardzo łatwo wykonuje się wyszukiwanie DNS: `[system.net.dns]::Resolve($address)`. Najważniejsze w tym przykładzie jest jednak to, że wynik nie jest zwykłym tekstem, tylko obiektem.

Do pracy na zwykłych łańcuchach mamy do dyspozycji wiele metod, np. "łańcuch" | `Get-Member`. Ponadto klas i metod .NET można używać z wieloma innymi typami obiektów, np. datami, adresami IP, identyfikatorami URI itd. W konsoli Windows PowerShell można pracować z tymi wszystkimi typami danych, które niegdyś były dostępne tylko na platformie .NET. Można nawet używać bibliotek .NET do budowy interfejsów użytkownika i sieciowych.

Gdy pracuję z danymi w formacie XML, mogę je po prostu pobrać w formie łańcucha i przekształcić w dokument XML. Na przykład poniższe polecenie tworzy z łańcucha dokument XML, który można zapisać w pliku, poddać analizie składniowej albo przeszukać przy użyciu instrukcji XPath lub XQuery.

```
$dom=[xml]"<doc><item1>value1</item1><item1>value2</item1><item1>value3</item2>subvalue1</item2></item1></doc>"
$dom.doc
$dom | get-member
```

Mogę napisać wyrażenie regularne i przy użyciu dwulinijkowego skryptu Windows PowerShell sprawdzić, czy pasuje do niego wybrany łańcuch. Jeśli dorzucę jeszcze jeden wiersz kodu, to będę mógł wyświetlić wszystkie znalezione dopasowania. Możliwość tworzenia i używania wyrażeń regularnych w Windows PowerShell jest niezwykle przydatna.

```
$regex=[regex]"^((6\.(1\.(98\.(10|[0-9]))|((9[0-7]|1[1-8]?[0-9])\..*))|(0\..*))|([0-5]\..*))$"
$regex.IsMatch("6.0.84.18")
```

Dodatkowe źródła informacji

- Centrum skryptów portalu TechNet pod adresem <http://www.microsoft.com/technet/scriptcenter> zawiera wiele przykładów skryptów Windows PowerShell pobierających i obsługujących dane wejściowe.
- Wszystkie skrypty opisane w tym rozdziale są dostępne do pobrania w repozytorium centrum skryptów portalu TechNet pod adresem <http://gallery.technet.microsoft.com/scriptcenter/PowerShell-40-Best-d9e16039>.

Rozdział 12

Obsługa błędów

- Obsługa brakujących parametrów
- Ograniczanie możliwości wyboru
- Postępowanie w przypadku braku uprawnień
- Postępowanie w przypadku braku dostawców WMI
- Niepoprawne typy danych
- Błędy zakresu
- Dodatkowe źródła informacji

Aby dobrze zaimplementować obsługę błędów w skrypcie, należy zastanowić się, jak ten skrypt będzie używany. Przewidywany sposób używania skryptu czasami nazywa się **scenariuszem używania** (ang. *use case scenario*). Pojęcie to dotyczy opisu sposobu interakcji użytkownika ze skryptem.

W prostych przypadkach zadaniem użytkownika może być jedynie wpisanie nawy skryptu w konsoli Windows PowerShell. Skrypt taki jak *Get-Bios.ps1* może obyć się bez jakiegokolwiek obsługi błędów, ponieważ nie przyjmuje żadnych danych wejściowych. Jego jedynym zadaniem jest wyświetlenie informacji, które zawsze powinny być dostępne, ponieważ klasa WMI `Win32_Bios` znajduje się we wszystkich systemach Windows od wersji Windows 2000.

Get-Bios.ps1

```
Get-WmiObject -class Win32_Bios
```

Jeśli jednak zasady obsługi skryptu są bardziej skomplikowane, obsługa błędów może być nieodzowna. Większość skryptów używanych w środowiskach korporacyjnych przyjmuje jakieś parametry od użytkownika poprzez wiersz poleceń. Bardzo niewiele z nich trzeba otwierać w edytorze, aby zmienić przypisania zmiennych. Najczęściej wartości podaje się w wierszu poleceń, w którym można popełnić wszystkie możliwe rodzaje błędów. Najczęstszym błędem jest niepodanie jakiejś podstawowej informacji, np. nazwy komputera docelowego. Co się dzieje, gdy użytkownik wpisze nazwę komputera wyłączzonego albo niedostępnego w danej sieci? Załóżmy, że mamy skrypt monitorujący wydajność zdalnego komputera, w którym użytkownik może wybrać odstęp czasowy między sprawdzeniami. Co się stanie, gdy użytkownik wybierze sprawdzanie liczników wydajności co jedną dziesiątą sekundy? Może to obniżyć wydajność badanego komputera. Albo wyobraź sobie skrypt próbujący odczytać dane z klasy WMI, której nie ma na zdalnym komputerze. Jak taki skrypt ma zadziałać w przypadku wystąpienia błędu? Istnieją wypróbowane metody radzenia sobie w takich sytuacjach i w tym rozdziale je poznasz.

Obsługa brakujących parametrów

Jak już wiemy, skrypt *Get-Bios.ps1* nie przyjmuje żadnych danych od użytkownika za pośrednictwem wiersza poleceń. Jest to dobry sposób na wyeliminowanie błędów, jednak nie zawsze tak się da. Ale gdy umożliwiamy użytkownikom wprowadzanie danych do naszego skryptu, otwieramy furtkę wszelkim możliwym rodzajom błędów. W zależności od sposobu przyjmowania danych z wiersza poleceń czasami konieczne jest sprawdzenie, czy przekazane przez użytkownika informacje są odpowiedniego typu. Jak widać, dzięki nieprzyjmowaniu parametrów z wiersza poleceń w skrypcie *Get-Bios.ps1* uniknięto większości potencjalnych błędów.

Przypisywanie parametrowi wartości domyślnej

Wartość domyślną można przypisać parametrowi wiersza poleceń na dwa sposoby. Można to zrobić w instrukcji *Param* albo w samym skrypcie. Z tych dwóch rozwiązań wolę przypisywanie wartości domyślnej w bloku instrukcji *Param*, ponieważ dzięki temu skrypt jest bardziej czytelny.

Wykrywanie braku wartości i przypisywanie domyślnej wartości w skrypcie

W skrypcie *Get-BiosInformation.ps1* znajduje się parametr wiersza poleceń o nazwie *-computername*, umożliwiający skryptowi pobieranie informacji zarówno z komputera lokalnego, jak i zdalnego. Jeśli skrypt ten zostanie uruchomiony bez podania wartości dla parametru *-computername*, polecenie *Get-WmiObject* nie zostanie wykonane, ponieważ potrzebuje tej informacji do działania. Problem ten rozwiązano przez sprawdzenie obecności zmiennej *\$computerName*. Jeśli jej nie ma, to znaczy, że nie została utworzona przy użyciu parametru wiersza poleceń i skrypt przypisuje jej wartość domyślną. Poniżej znajduje się odpowiedzialny za to wiersz kodu:

```
If(-not($computerName)) { $computerName = $env:computerName }
```

Poniżej znajduje się kompletny kod źródłowy skryptu *Get-BiosInformation.ps1*:

Get-BiosInformation.ps1

```
Param(
    [string]$computerName
) #end param

Function Get-BiosInformation($computerName)
{
    Get-WmiObject -class Win32_Bios -computername $computername
} #end function Get-BiosName

# *** punkt początkowy skryptu ***
If(-not($computerName)) { $computerName = $env:computerName }
Get-BiosInformation -computername $computername
```

Przypisywanie domyślnej wartości zmiennej w instrukcji Param

Aby przypisać wartość domyślną zmiennej w instrukcji Param, należy wpisać nazwę parametru, znaku równości oraz wartość domyślną tego parametru, jak pokazano poniżej:

```
Param(
    [string]$computerName = $env:computername
) #end param
```

Zaletą tej techniki jest czytelność kodu. Deklaracje parametrów i ich wartości domyślne znajdują się w jednym miejscu, dzięki czemu wystarczy rzut oka, aby dowiedzieć się, które parametry mają wartości domyślne. Ponadto przypisywanie domyślnych wartości parametrom w instrukcji Param ułatwia pisanie skryptu. Zwróć uwagę na brak instrukcji If do sprawdzania, czy istnieje zmienna \$computerName. Poniżej znajduje się kompletny kod źródłowy skryptu *Get-BiosInformationDefaultParam.ps1*:

Get-BiosInformationDefaultParam.ps1

```
Param(
    [string]$computerName = $env:computername
) #end param

Function Get-BiosInformation($computerName)
{
    Get-WmiObject -class Win32_Bios -computername $computername
} #end function Get-BiosName

# *** punkt początkowy skryptu ***

Get-BiosInformation -computername $computername
```

Tworzenie parametrów obowiązkowych

Najlepszym sposobem obrony przed błędami jest w ogóle uniemożliwienie ich wystąpienia. W Windows PowerShell od wersji 2.0 istnieje możliwość oznaczania parametrów jako obowiązkowych. Zaletą tego rozwiązania jest to, że uniemożliwia uruchomienie skryptu, dopóki użytkownik nie poda odpowiednich informacji. Jeśli więc nie chcesz, aby Twój skrypt był uruchamiany bez podania określonych opcji, najlepiej zdefiniuj wybrane parametry jako obowiązkowe. Do tworzenia parametrów obowiązkowych służy słowo kluczowe Mandatory:

```
Param(
    [Parameter(Mandatory=$true)]
    [string]$drive,
    [string]$computerName = $env:computerName
) #end param
```

Poniżej znajduje się kompletny kod źródłowy skryptu *MandatoryParameter.ps1*:

MandatoryParameter.ps1

```
#Requires -version 4.0
Param(
    [Parameter(Mandatory=$true)]
    [string]$drive,
```

```
[string]$computerName = $env:computerName
) #end param

Function Get-DiskInformation($computerName,$drive)
{
    Get-WmiObject -class Win32_volume -computername $computername '
    -filter "DriveLetter = '$drive'"
} #end function Get-BiosName

# *** punkt początkowy skryptu ***

Get-DiskInformation -computername $computerName -drive $drive
```

Jeśli powyższy skrypt zostanie uruchomiony bez podania wartości dla parametru, to nie wystąpi błąd, tylko konsola Windows PowerShell poprosi o wpisanie tej informacji:

```
PS C:\bp> .\MandatoryParameter.ps1
```

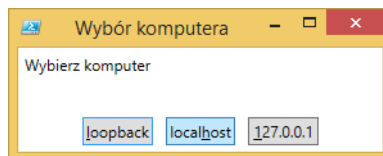
```
cmdlet MandatoryParameter.ps1 at command pipeline position 1
Supply values for the following parameters:
drive:
```

Ograniczanie możliwości wyboru

Ilość konstrukcji sprawdzających błędy można zredukować, chociaż zależy to w dużym stopniu od budowy skryptu. Jeśli użytkownik może wybierać tylko opcje z określonego zbioru, można użyć metody `PromptForChoice`. Jeżeli trzeba ograniczyć możliwości wyboru do komputerów, które są w danej chwili uruchomione, można przed próbą nawiązania połączenia wysłać do nich polecenie ping. A jeśli trzeba ograniczyć możliwości wyboru do podzbioru komputerów lub własności, można pobrać dane z pliku tekstowego i użyć operatora `-contains`. Podrozdział ten zawiera opis wszystkich tych technik ograniczania dopuszczalnych wartości wejściowych do skryptów.

Ograniczanie liczby opcji za pomocą metody `PromptForChoice`

Metoda `PromptForChoice` służy do wyświetlania użytkownikowi tylko ściśle określonego zestawu opcji do wyboru. W ten sposób całkowicie eliminuje się problem niepoprawnych danych wejściowych. Przykład zastosowania tej metody pokazano na rysunku 12.1.



RYСУNEK 12.1. Metoda `PromptForChoice` wyświetla menu z opcjami do wyboru

Przykład użycia metody `PromptForChoice` przedstawiono w skrypcie *Get-ChoiceFunction.ps1*. Znajduje się w nim funkcja o nazwie `Get-Choice` zawierająca zmienne `$caption` i `$message` przechowujące tytuł i tekst okna wyświetlanego przez metodę `PromptForChoice`. Opcje do wyboru są egzemplarzami klasy `.NET ChoiceDescription`. Przy tworzeniu obiektu tej klasy podaje się tablicę opcji do wyboru, jak pokazano poniżej:

```
$choices = [System.Management.Automation.Host.ChoiceDescription[]] `
@("&loopback", "local&host", "&127.0.0.1")
```

Następnie należy wybrać, który numer będzie reprezentował domyślny wybór. Należy przy tym pamiętać, że `ChoiceDescription` to tablica, więc pierwsza opcja ma numer 0. Następnie wywołujemy metodę `PromptForChoice` i wyświetlamy opcje:

```
[int]$defaultChoice = 0
$choiceRTN = $host.ui.PromptForChoice($caption,$message,$choices,$defaultChoice)
```

Ponieważ metoda `PromptForChoice` zwraca liczbę całkowitą, wartość zmiennej `$choiceRTN` można przetestować za pomocą instrukcji `if`. Ale instrukcja `Switch` jest bardziej zwięzła i w tym przypadku będzie lepszym rozwiązaniem. Poniżej znajduje się instrukcja `Switch` z funkcji `Get-Choice`:

```
switch($choiceRTN)
{
    0 { "loopback" }
    1 { "localhost" }
    2 { "127.0.0.1" }
}
```

Funkcja `Get-Choice` zwraca komputer wybrany za pośrednictwem metody `PromptForChoice`. Wywołanie funkcji `Get-Choice` umieszczamy w nawiasie, aby wymusić jej wykonanie przed pozostałą częścią polecenia.

```
Get-WmiObject -class win32_bios -computername (Get-Choice)
```

Takie rozwiązanie jest doskonałe w przypadkach, gdy pracownicy pomocy technicznej pracują z ograniczoną liczbą komputerów. Jego wadą jest natomiast to, że gdyby często trzeba było zmieniać listę komputerów, stałoby się to bardzo czasochłonnym zajęciem. Poniżej znajduje się kompletny kod źródłowy skryptu *Get-ChoiceFunction.ps1*:

Get-ChoiceFunction.ps1

```
Function Get-Choice
{
    $caption = "Wybór komputera"
    $message = "Wybierz komputer"
    $choices = [System.Management.Automation.Host.ChoiceDescription[]] `
@("&loopback", "local&host", "&127.0.0.1")
    [int]$defaultChoice = 0
    $choiceRTN = $host.ui.PromptForChoice($caption,$message, $choices,$defaultChoice)

    switch($choiceRTN)
    {
        0 { "loopback" }
        1 { "localhost" }
        2 { "127.0.0.1" }
    }
}
```

```

}
} #end Get-Choice function

Get-WmiObject -class win32_bios -computername (Get-Choice)

```

Znajdowanie dostępnych komputerów za pomocą polecenia ping

Jeśli komputerów jest dużo albo ich lista nie jest stała, to rozwiązaniem problemu z nawiązywaniem połączeń z nieistniejącymi komputerami jest wysyłanie polecenia ping przed podjęciem próby nawiązania połączenia WMI.

Do wysyłania polecenia ping do komputerów można użyć klasy WMI Win32_PingStatus, najlepiej w funkcji wysyłającej to polecenie do komputera docelowego. Jako że chcemy szybko otrzymać odpowiedź, funkcja Test-ComputerPath wysyła tylko jeden ping. Przyjmuje też tylko jedną informację na wejściu — nazwę lub adres IP komputera docelowego. W ramach kontrolowania wartości przekazywanych do funkcji parametr \$computer jest ograniczony do typu string, dzięki czemu nikt nie wprowadzi do funkcji danych innego typu niż łańcuch. Poniżej znajduje się kod źródłowy funkcji Test-ComputerPath:

```

Function Test-ComputerPath([string]$computer)
{
    Get-WmiObject -class win32_pingstatus -filter "address = '$computer'"
} #end Test-ComputerPath

```

Do kodu wywołującego zostaje zwrócony podzbiór obiektu klasy Win32_PingStatus:

| Source | Destination | IPv4Address | IPv6Address |
|--------|-------------|---------------|-------------|
| ----- | ----- | ----- | ----- |
| EDLT | dc1 | 192.168.0.101 | |

W skrypcie *Test-ComputerPath.ps1* używana jest własność statusCode z obiektu Win32_PingStatus. Wartość 0 oznacza pomyślne wysłanie polecenia ping. Natomiast wartość null lub jakakolwiek inna różna od zera oznacza niepowodzenie operacji. Jako że do skryptu zwracany jest obiekt klasy Win32_PingStatus, własność statusCode można pobrać bezpośrednio i sprawdzić ją za pomocą operatora równości.

```
if( (Test-ComputerPath -computer $computer).statusCode -eq 0 )
```

Jeśli własność statusCode ma wartość 0, skrypt *Test-ComputerPath.ps1* pobiera za pomocą polecenia Get-WmiObject informacje dotyczące BIOS-u z klasy WMI Win32_Bios.

```
Get-WmiObject -class Win32_Bios -computer $computer
```

Jeśli komputer docelowy jest niedostępny, skrypt *Test-ComputerPath.ps1* wyświetla w konsoli Windows PowerShell stosowną informację.

```

Else
{
    "Komputer $computer jest niedostępny."
}

```

Poniżej znajduje się kompletny kod źródłowy skryptu *Test-ComputerPath.ps1*:

Test-ComputerPath.ps1

```

Param([string]$computer = "localhost")

Function Test-ComputerPath([string]$computer)
{
    Get-WmiObject -class win32_pingstatus -filter "address = '$computer'"
} #end Test-ComputerPath

# *** punkt początkowy skryptu ***

if( (Test-ComputerPath -computer $computer).statusCode -eq 0 )
{
    Get-WmiObject -class Win32_Bios -computer $computer
}
Else
{
    "Komputer $computer jest niedostępny."
}

```

Sprawdzanie zawartości tablicy za pomocą operatora -contains

Aby zweryfikować dane otrzymane z wiersza poleceń, można użyć operatora `-contains`, który umożliwia przejrzenie zawartości tablicy dopuszczalnych wartości. Najlepiej technikę tę przedstawić na konkretnym przykładzie. Mamy tablicę trzech wartości zapisaną w zmiennej `$noun`. Za pomocą operatora `-contains` sprawdzamy, czy tablica ta zawiera napis `wombat australijski`. Jako że żaden element tej tablicy nie zawiera takiego tekstu, operator `-contains` zwraca wartość `False`.

```

PS C:\> $noun = "kot","pies","królik"
PS C:\> $noun -contains "wombat australijski"
False
PS C:\>

```

Jeżeli tablica zawiera szukany tekst, operator `-contains` zwraca wartość `True`.

```

PS C:\> $noun = "kot","pies","królik"
PS C:\> $noun -contains "królik"
True
PS C:\>

```

Operator `-contains` zwraca wartość `True` tylko wtedy, gdy znajdzie dokładne dopasowanie. Częściowe dopasowania się nie liczą.

```

PS C:\> $noun = "kot","pies","królik"
PS C:\> $noun -contains "ólik"
False
PS C:\>

```

Operator `-contains` nie rozróżnia wielkości liter, więc zwraca wartość `True`, nawet jeśli wielkość liter w szukanym i przeszukiwanym ciągu znaków jest różna.

```

PS C:\> $noun = "kot","pies","królik"
PS C:\> $noun -contains "Królik"

```

```
True
PS C:\>
```

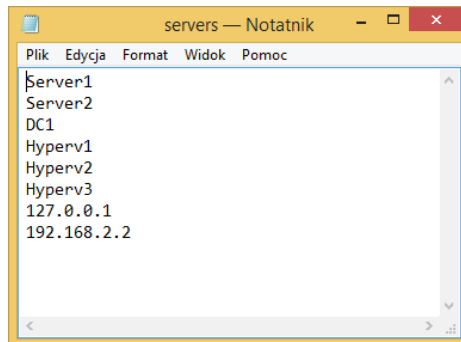
Jeśli trzeba przeprowadzić wyszukiwanie z rozróżnianiem wielkości liter, należy użyć specjalnej wersji operatora `-contains` w postaci `-ccontains`. Ta wersja zwraca wartość `True` tylko wtedy, gdy pasują także wielkości liter:

```
PS C:\> $noun = "kot","pies","królik"
PS C:\> $noun -contains "Królik"
False
PS C:\> $noun -ccontains "królik"
True
PS C:\>
```

W skrypcie *Get-AllowedComputer.ps1* utworzono jeden parametr wiersza poleceń do przechowywania nazwy komputera docelowego dla zapytania WMI. Parametr ten nazywa się `-computer` i jest łańcuchem oraz ma przypisaną wartość domyślną z dysku środowiskowego. Jest to dobre podejście, ponieważ dzięki temu skrypt otrzyma nazwę komputera lokalnego, której później można użyć do sporządzenia raportu. Jeśli wartość tego parametru ustawiono by na `localhost`, to nie byłoby wiadomo, do którego komputera należą wyniki.

```
Param([string]$computer = $env:computername)
```

Funkcja `Get-AllowedComputer` tworzy tablicę dozwolonych nazw komputerów i sprawdza wartość zmiennej `$computer`, aby dowiedzieć się, czy ona istnieje. Jeżeli wartość zmiennej `$computer` znajduje się w tablicy, funkcja `Get-AllowedComputer` zwraca wartość `True`. Jeżeli nie ma tej wartości, funkcja zwraca wartość `False`. Tablica nazw komputerów jest tworzona przez odczytanie pliku tekstowego zawierającego nazwy komputerów za pomocą polecenia `Get-Content`. Plik ten nazywa się *servers.txt* i jest w formacie ASCII. Każda nazwa znajduje się w osobnej linii, jak widać na rysunku 12.2.



RYСУNEK 12.2. Plik tekstowy z nazwami komputerów i adresami umożliwia określenie listy dopuszczalnych komputerów

Plik tekstowy jest wygodniejszy w obsłudze niż wpisana bezpośrednio w kod źródłowy tablica. Ponadto plik taki można umieścić w centralnym udziale, aby mogły go używać także inne skrypty. Poniżej znajduje się kod źródłowy funkcji `Get-AllowedComputer`:

```
Function Get-AllowedComputer([string]$computer)
{
    $servers = Get-Content -path c:\fso\servers.txt
    $servers -contains $computer
} #end funkcja Get-AllowedComputer
```

Jako że funkcja `Get-AllowedComputer` zwraca wartość logiczną (prawda lub fałsz), można jej użyć bezpośrednio w instrukcji `if`, aby dowiedzieć się, czy wartość przekazana dla zmiennej `$computer` znajduje się na liście dozwolonych wartości. Jeżeli funkcja `Get-AllowedComputer` zwróci wartość `True`, następuje pobranie informacji o BIOS-ie docelowego komputera za pomocą polecenia `Get-WmiObject`.

```
if(Get-AllowedComputer -computer $computer)
{
    Get-WmiObject -class Win32_Bios -Computer $computer
}
```

A jeśli wartość zmiennej `$computer` nie zostanie znaleziona w tablicy `$servers`, następuje wyświetlenie informacji, że dany komputer jest niedozwolony.

```
Else
{
    "Komputer $computer jest niedozwolony."
}
```

Poniżej znajduje się kompletny kod źródłowy skryptu *Get-AllowedComputer.ps1*:

Get-AllowedComputer.ps1

```
Param([string]$computer = $env:computername)

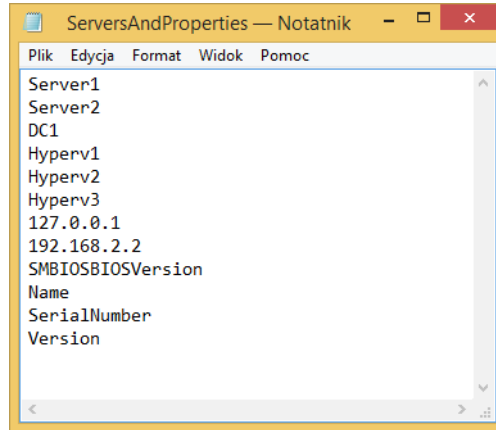
Function Get-AllowedComputer([string]$computer)
{
    $servers = Get-Content -path c:\fso\servers.txt
    $servers -contains $computer
} #end Get-AllowedComputer function

# *** punkt początkowy skryptu ***

if(Get-AllowedComputer -computer $computer)
{
    Get-WmiObject -class Win32_Bios -computer $computer
}
Else
{
    "Komputer $computer jest niedozwolony."
}
```

Testowanie własności za pomocą operatora `-contains`

W funkcji `Get-AllowedComputer` nie musimy ograniczać się tylko do sprawdzania nazw komputerów. Wystarczy do pliku tekstowego dodać parę innych informacji (rysunek 12.3) i można testować także inne własności.



RYSUNEK 12.3. Plik tekstowy zawierający nazwy serwerów i własności rozszerza funkcjonalność skryptu

Aby zamienić skrypt *Get-AllowedComputer.ps1* w skrypt *Get-AllowedComputerAndProperty.ps1*, wystarczy tylko parę zmian. Pierwsza polega na dodaniu jeszcze jednego parametru wiersza poleceń, aby umożliwić użytkownikowi wybór własności do wyświetlenia.

```
Param([string]$computer = $env:computername, [string]$property="name")
```

Następnie należy zmienić sygnaturę funkcji *Get-AllowedComputer*, aby umożliwić przekazywanie nazwy własności. Wyniki operatora `-contains` nie są zwracane bezpośrednio, tylko są zapisywane w zmiennych. Funkcja *Get-AllowedComputer* najpierw sprawdza, czy tablica `$servers` zawiera podaną nazwę komputera. Następnie sprawdza, czy tablica `$servers` zawiera własność `name`. Każda z otrzymanych wartości zostaje zapisana w zmiennej. Następnie zmienne te zostają dodane do siebie i następuje zwrócenie wyniku do kodu wywołującego. W przypadku sumowania wartości logicznych wartość `True` powstaje wyłącznie wtedy, gdy obie sumowane wartości reprezentują prawdę.

```
PS C:\> $true -and $false
False
PS C:\> $true -and $true
True
PS C:\> $false -and $false
False
PS C:\>
```

Poniżej znajduje się kod źródłowy zmodyfikowanej funkcji *Get-AllowedComputer*:

```
Function Get-AllowedComputer([string]$computer, [string]$property)
{
    $servers = Get-Content -path c:\fso\serversAndProperties.txt
    $s = $servers -contains $computer
    $p = $servers -contains $property
    Return $s -and $p
} #end funkcja Get-AllowedComputer
```

Za pomocą instrukcji `if` skrypt sprawdza, czy zarówno nazwa komputera, jak i własności znajdują się na listach dozwolonych wartości. Jeśli funkcja *Get-AllowedComputer* zwraca wartość `True`, polecenie *Get-WmiObject* wyświetla wartość wybranej własności z wybranego komputera.

```
if(Get-AllowedComputer -computer $computer -property $property)
{
    Get-WmiObject -class Win32_Bios -Computer $computer |
    Select-Object -property $property
}
```

Jeżeli komputera i własności nie ma na liście dozwolonych, skrypt *Get-AllowedComputerAnd-Property.ps1* wyświetla informację, że użyto niedozwolonej wartości.

```
Else
{
    "Podano niedozwoloną nazwę komputera $computer albo własności, 'r' $property."
}
```

Poniżej znajduje się kompletny kod źródłowy skryptu *Get-AllowedComputerAndProperty.ps1*:

Get-AllowedComputerAndProperty.ps1

```
Param([string]$computer = $env:computername,[string]$property="name")

Function Get-AllowedComputer([string]$computer, [string]$property)
{
    $servers = Get-Content -path c:\fso\serversAndProperties.txt
    $s = $servers -contains $computer
    $p = $servers -contains $property
    Return $s -and $p
} #end Get-AllowedComputer function

# *** punkt początkowy skryptu ***

if(Get-AllowedComputer -computer $computer -property $property)
{
    Get-WmiObject -class Win32_Bios -computer $computer |
    Select-Object -property $property
}
Else
{
    "Podano niedozwoloną nazwę komputera $computer albo własności $property."
}
```

Postępowanie w przypadku braku uprawnień

Kolejnym źródłem potencjalnych błędów jest podniesienie uprawnień skryptu. Od systemu Windows Vista znacznie ułatwiono działanie i pracę bez ciągłego posiadania uprawnień administracyjnych. W efekcie coraz więcej użytkowników i administratorów sieci korzysta z kont nie należących do grupy administracyjnej. Funkcja Kontrola konta użytkownika (ang. *User Account Control* — UAC) ułatwia zwiększanie uprawnień programów w trybie interaktywnym, ale konsola Windows PowerShell i języki skryptowe jej nie znają, dlatego nie wyświetlają monitu o podanie hasła, gdy trzeba zwiększyć poziom uprawnień w celu wykonania określonej czynności. W związku z tym programista skryptu podczas pracy musi uwzględnić kwestię uprawnień i odpowiednio o nią zadbać. Skrypt *Get-Bios.ps1* nie używa klasy WMI i nie potrzebuje zwiększonych uprawnień do działania. W obecnej postaci może go uruchomić każdy użytkownik komputera należący do

grupy lokalnej — obejmuje ona wszystkich interaktywnie zalogowanych użytkowników. Oznacza to, że przed pobraniem informacji z klasy `Win32_Bios` nie trzeba sprawdzać, jakimi uprawnieniami dysponuje skrypt.

Wiedza tajemna

**Gary Siepser, Senior Premier Field Engineer i kierownik ds. technologii PowerShell
Microsoft Corporation**

Razem z kolegą pracujemy nad fajnym projektem polegającym na integracji technologii Kinect for Windows z Windows PowerShell. W projekcie tym bardzo dużo nauczyłem się na temat posługiwania się interfejsami API, pisania kodu bardziej jak programista oraz przekształcania kodu źródłowego w języku C# w skrypty Windows PowerShell (to akurat nie jest takie fajne).

Jedną z dziedzin wiedzy, w których miałem luki, było zarządzanie zasobami, a dokładniej mówiąc, nie wiedziałem, jak zwalniać zasoby, gdy są już niepotrzebne. Podczas używania interfejsu API Kinect odkryłem, że po uruchomieniu czujnika Kinect i włączeniu różnych strumieni danych 20% procesora było wykorzystywane przez oprogramowanie wykonujące na komputerze obliczenia związane z interpretowaniem i przygotowywaniem danych kamery Kinect do użytku poprzez API. Także magistrala USB, do której podłączone było urządzenie Kinect, była bardzo zajęta, a po włączeniu Kinecta następowało jego zablokowanie i stawał się on niedostępny dla innych aplikacji Kinect (podobnie jak skrypty Windows PowerShell w innych oknach). Pracując z Windows PowerShell, często otwieram jedno okno i używam go przez długi czas. Od czasu do czasu wykonam jakieś polecenie, jakąś funkcję itd. Szybko się zorientowałem, że po zakończeniu przetwarzania danych z Kinecta przez skrypt lub funkcję natychmiast za wszelką cenę należy wyłączyć czujnik, aby zwolnić i odblokować zasoby.

Zatrzymanie Kinecta w celu zwolnienia zasobów przy użyciu interfejsu API nie było trudne i polegało na wywołaniu jednej metody. Początkowo po prostu dodawałem to wywołanie na końcu każdego kodu. Ale prawdziwe problemy pojawiły się dopiero w fazie programowania. Ciągłe pojawiały się błędy i musiałem wyłączać skrypty, zanim zakończyły działanie (naciskając kombinację klawiszy `Ctrl+C` lub klikając przycisk `Stop` w PowerShell ISE). W efekcie następowało wykonanie kodu porządkującego, który zatrzymywał Kinecta, ponieważ działanie skryptu zostało przedwcześnie zakończone. Był to duży problem, z powodu którego dochodziło do zablokowania zasobów i marnowania czasu pracy procesora.

Wkrótce dowiedziałem się o strukturze do obsługi błędów `Try-Catch-Finally`, która była zbawieniem w mojej sytuacji. Konsola Windows PowerShell rozpoznaje blok `Finally` i wykonuje go, nawet gdy działanie skryptu zostanie przerwane w bloku `Try` lub `Catch`. Włączenie metody `Stop Kinecta` do bloku `Finally` okazało się strzałem w dziesiątkę. Po zakończeniu pracy nad kodem i podczas jego używania było to wygodne, ale podczas programowania rozwiązanie to było niezastąpione.

Podczas gdy konstrukcja Try-Catch-Finally ułatwia obsługę błędów powodujących zamknięcie programu i jest powszechnie wykorzystywana przez programistów, osobiście uważam, że najcenniejsza jest gwarancja wykonania bloku Finally. Gdy wcieliłem tę konstrukcję do swojego kodu, zauważyłem, że metoda Kinecta Stop była wykonywana zawsze, niezależnie od sposobu zakończenia działania przez skrypt — samoczynnie czy też z powodu przerwania ręcznego za pomocą kombinacji klawiszy *Ctrl+C*. Dzięki temu w końcu wyeliminowałem problem marnowania zasobów.

Podejmowanie nieudanych prób

Jednym ze sposobów postępowania w przypadku braku odpowiednich uprawnień jest podjęcie próby wykonania czynności i poniesienie porażki. Powoduje to wygenerowanie błędu. W konsoli Windows PowerShell wyróżnia się dwa rodzaje błędów: powodujące zamknięcie programu i niepowodujące zamknięcia programu. Błędy powodujące zamknięcie programu to oczywiście takie, przez które skrypt nagle przestaje działać i nie da się z tym już nic zrobić. Natomiast błędy niepowodujące zamknięcia programu są wyświetlane w konsoli, ale skrypt nadal działa po ich wystąpieniu. Błędy powodujące zamknięcie programu są z reguły poważniejsze. Występują na przykład przy próbie użycia klasy .NET albo obiektu COM w konsoli Windows PowerShell, gdy programista użyje nieistniejącego polecenia albo gdy użytkownik nie poda wymaganego parametru w wierszu poleceń. Dobrze napisany skrypt obsługuje wszystkie przewidziane przez programistę błędy, a nieprzewidziane zwraca do informacji użytkownika. Jako że każdy przyzwoity język skryptowy musi zawierać mechanizmy obsługi błędów, w Windows PowerShell również dostępnych jest kilka metod podejścia do tego problemu. Kiedyś używało się instrukcji Trap, która jednak potrafi sprawiać problemy. Ale jest też nowa technika w postaci konstrukcji Try-Catch-Finally.

Wiedza tajemna

Chwywanie błędów

James Brundage, prezes

Start Automating

W Windows PowerShell 1.0 błędy powodujące zamknięcie programu można było obsługiwać tylko w jeden sposób: za pomocą instrukcji Trap. Umieszcza się ją na końcu skryptu, aby przechwyciła wszystkie błędy (lub wszystkie błędy określonego typu). Dlatego większość skryptów Windows PowerShell 1.0 zawierających mechanizm obsługi błędów wygląda mniej więcej tak:

```
Rób-Coś
... Rób-CośJeszcze
trap {
    "Przytrafiło się coś złego."
}
```

Niestety instrukcja `Trap` jest trochę dziwna. Po pierwsze: zna ją niewielu programistów, i to zarówno języków skryptowych, jak i kompilowanych. Po drugie: instrukcja `Trap` tak naprawdę nie służy do łatwego przechwytywania błędów występujących w konkretnych paru liniach kodu. Jeśli na przykład użyje się jej w jednym skrypcie, bo przewiduje się możliwość wystąpienia pewnych błędów, i wywoła się jakiś inny skrypt, w którym również wystąpi błąd, to instrukcja `Trap` może przechwycić oba zestawy błędów, a Ty będziesz długo dumał nad tym, dlaczego Twój skrypt nie działa.

Programiści znający język C# lub JavaScript znają też konstrukcję `Try-Catch-Finally`. Została ona wprowadzona także w Windows PowerShell 2.0, aby rozwiązać niektóre niedogodności związane z obsługiwaniem błędów we wcześniejszej wersji narzędzia.

Blok `Try` to sekcja kodu, która może obsługiwać błędy. Próbuje on wykonać znajdujący się w nim kod i jeśli w tym czasie wystąpi jakiś błąd powodujący zamknięcie programu, następuje przechwycenie tego błędu przez najbliższy blok `Catch`. Bloki `Try` i `Catch` muszą występować parami (każdemu blokowi `Try` może, a zarazem musi odpowiadać tylko jeden blok `Catch`), ale dodatkowo można zdefiniować blok `Finally`. Kod znajdujący się w tym bloku jest niezależny od tego, czy wystąpią błędy, czy nie, więc jest on doskonałym miejscem na umieszczenie kodu porządkującego.

Poniżej znajduje się kompletny przykład użycia opisywanej konstrukcji:

```
try {
    throw "Houston, mamy problem."
}
catch {
    Write-Error $_
    try {
        Test-System
    }
    catch [Management.Automation.CommandNotFoundException] {
        "Gdzie jest polecenie $($_.TargetObject)?"
    }
}
finally {
    "pa, pa"
}
```

W przykładzie tym pierwszy błąd ("Houston, mamy problem.") zostaje przechwycony przez blok `Catch` i wydrukowany za pomocą polecenia `Write-Error`. To powoduje przemianę błędu powodującego zamknięcie programu w błąd niepowodujący zamknięcia programu, dzięki czemu skrypt może kontynuować działanie. We wspomnianym bloku `catch` znajduje się inny blok `try-catch` wykonujący polecenie diagnostyczne (`Test-System`). Jeśli polecenie to wydrukuje jakiegokolwiek błąd, to chcę je zobaczyć. Ale pytanie do użytkownika o lokalizację polecenia chcę wyświetlić tylko wówczas, gdy polecenie nie zostanie znalezione. Dlatego utworzyłem blok `Catch` przechwytyjący wyjątki typu `CommandNotFoundException` (wyjątki tego typu są zgłaszane, gdy brakuje polecenia). Blok `Finally` jest wykonywany niezależnie od tego, czy wystąpił błąd, czy nie, więc po zakończeniu działania skryptu wyświetlam napis `pa, pa`.

Cicha reinterpretacja błędów to jedna z najbardziej przydatnych czynności, jakie można wykonać przy użyciu bloków `Try-Catch`. Osobiście lubię, gdy w oknie konsoli wyświetlane są

informacje o wszystkich występujących w czasie działania skryptu błędach, ale nie lubię, gdy użytkownicy moich skryptów oglądają czerwone alarmy. (To zaostża im chorobę wrzodową). Dlatego w swoich skryptach często umieszczam następujący kod:

```
try {
}
catch {
    Write-Debug ($_|Out-String)
}
```

Powyższy blok przesyła błędy do strumienia Debug (który domyślnie jest ukryty, a który mogę włączyć za pomocą instrukcji `$DebugPreference = "Continue"`). W efekcie użytkownicy moich skryptów prawie nigdy nie widzą informacji o błędach, ja natomiast mam wgląd do wszystkich błędów.

Sprawdzanie, czy skrypt posiada potrzebne uprawnienia, i eleganckie kończenie pracy

Najlepszym rozwiązaniem w przypadku, gdy skrypt nie ma wystarczających uprawnień, jest eleganckie zakończenie działania. Jakie problemy mogą wystąpić z wykonywaniem takiego prostego skryptu jak opisany wcześniej *Get-Bios.ps1*? Jego wykonanie może się nie udać, gdy w konsoli Windows PowerShell będzie ustawiona zaostzona reguła zabezpieczeń, co uniemożliwia wykonywanie jakichkolwiek skryptów. Problem z tą regułą polega na tym, że ze względu na brak możliwości wykonywania skryptów nie można za pomocą skryptu sprawdzić, czy jest ona włączona. Ale ponieważ reguła ta jest przechowywana w rejestrze, można napisać w języku VBScript skrypt sprawdzający i ustawiający regułę zabezpieczeń przed uruchomieniem skryptu Windows PowerShell, chociaż nie jest to idealne rozwiązanie. Najlepszym sposobem jest ustawienie odpowiedniej reguły zabezpieczeń za pomocą reguły grupowej. W pojedynczym komputerze można to zrobić poprzez uruchomienie konsoli Windows PowerShell z uprawnieniami administratora i wykonanie polecenia `Set-ExecutionPolicy`. W większości przypadków wystarczające jest ustawienie `RemoteSigned`, które pokazano poniżej:

```
PS C:\> Set-ExecutionPolicy remotesigned
PS C:\>
```

Regułę zabezpieczeń zazwyczaj ustawia się raz i można o niej zapomnieć. Poza tym wiadomość jej dotycząca jest w miarę jasna i zawiera informację, że nie można wykonać skryptu z powodu całkowitego braku możliwości wykonywania skryptów w tym systemie. Ponadto w wiadomości tej znajduje się odnośnik do artykułu na temat różnych ustawień zabezpieczeń.

```
File C:\Documents and Settings\ed\Local Settings\Temp\tmp2A7.tmp.ps1 cannot be loaded because the
execution of scripts is disabled on this system. Please see "get-help about_signing" for more details.
At line:1 char:66
+ C:\Documents' and' Settings\ed\Local' Settings\Temp\tmp2A7.tmp.ps1 <<<<
```

Sposób użycia instrukcji #Requires

W Windows PowerShell 4.0 rozszerzono funkcjonalność instrukcji `#Requires` o możliwość definiowania paru dodatkowych testów wykonywanych przed uruchomieniem skryptu.

Należy pamiętać, że instrukcji tej nie używa się w funkcjach, poleceniach cmdlet ani przystawkach, ale można jej używać w skryptach. Oto zasady używania instrukcji `#Requires`:

- Instrukcja `#Requires` musi znajdować się na początku wiersza kodu.
- Instrukcja `#Requires` może znajdować się w dowolnym wierszu kodu w skrypcie.
- Skrypt może zawierać kilka instrukcji `#Requires`.
- Jeżeli skrypt zawiera więcej niż jedną instrukcję `#Requires`, każda z nich musi znajdować się w osobnym wierszu.

W tabeli 12.1 znajduje się lista dostępnych parametrów instrukcji `#Requires`.

TABELA 12.1. Wartości parametrów instrukcji `#Requires`

| Parametr | Opis i przykład użycia |
|--------------------|---|
| Version | Najstarsza obsługiwana wersja Windows PowerShell. Przykład: <code>#Requires -Version 4.0</code> |
| PSSnapin | Nazwa wymaganej przystawki. Przykład: <code>#Requires -PSSnapin mojabrzystawka</code> |
| Modules | Moduły wymagane przez skrypt. Przykład: <code>#Requires -Modules ActiveDirectory</code> |
| ShellID | Nazwa wymaganej powłoki. Przykład: <code>#Requires -ShellID Microsoft.PowerShell</code> |
| RunAsAdministrator | Skrypt może działać tylko z uprawnieniami administratora. Przykład: <code>#Requires -RunAsAdministrator</code> |

Wymaganie uprawnień administratora

Aby w starszych wersjach konsoli Windows PowerShell zdefiniować wymóg posiadania przez konsolę uprawnień administratora — tak by można było wykonać skrypt — należało napisać specjalną funkcję, np. taką jak `Test-IsAdmin` z pokazanego poniżej skryptu *Test-IsAdminFunction.ps1*.

Test-IsAdminFunction.ps1

```
Function Test-IsAdmin
{
    <#
        .Synopsis
            Sprawdza, czy użytkownik jest administratorem.
        .Description
```

```

        Zwraca prawdę, jeżeli użytkownik jest administratorem, lub fałsz w przeciwnym przypadku
    .Example
        Test-IsAdmin
    #>
    $identity = [Security.Principal.WindowsIdentity]::GetCurrent()
    $principal = New-Object Security.Principal.WindowsPrincipal $identity
    $principal.IsInRole([Security.Principal.WindowsBuiltinRole]::Administrator)
}

```

Sposób użycia funkcji `Test-IsAdmin` polega na załadowaniu jej do pamięci (przez skrypt) oraz użyciu w instrukcji warunkowej `if`. Poniżej znajduje się przykład takiej instrukcji:

```
if(-not (Test-IsAdmin)) {"Aby wykonać ten skrypt, trzeba mieć uprawnienia administratora."}
```

W Windows PowerShell 4.0 zadanie to jest znacznie prostsze dzięki konstrukcji `#Requires -RunAsAdministrator`. Dobrym przykładem jej zastosowania jest użycie poleceń z modułu `Hyper-V` (potrzebne są do tego uprawnienia administratora). Jeśli polecenia te zostaną wykonane bez uprawnień administracyjnych, to nie zwrócą błędów — nie zwrócą niczego. To powoduje konsternację i jeśli padnie na niedoświadczonego administratora, może on nawet podczas debugowania zniszczyć bardzo dobry skrypt. Poniżej znajduje się przykład zastosowania tej techniki — skrypt `get-VM.ps1`.

get-VM.ps1

```

#Requires -Version 4.0
#Requires -RunAsAdministrator
#Requires -Modules Hyper-V

Import-Module Hyper-V
Get-VM

```

Gdy ktoś spróbuje wykonać skrypt zawierający konstrukcję `#Requires -RunAsAdministrator`, nie mając uprawnień administracyjnych, to w konsoli zostanie wyświetlony szczegółowy komunikat o błędzie.

Wymaganie określonych modułów

W Windows PowerShell 4.0 łatwo można zdefiniować warunek, że do wykonania skryptu musi być dostępny określony moduł. Można nawet podać, która wersja tego modułu jest potrzebna. Wymagane moduły definiuje się w tablicy skrótów z kluczami `ModuleName` (nazwa modułu) i `ModuleVersion` (wersja modułu). Poniżej znajduje się kod skryptu `RequireModuleVersion.ps1` ilustrującego zastosowanie tej techniki.

RequireModuleVersion.ps1

```

#Requires -version 4.0
#Requires -RunAsAdministrator
#Requires -modules ScheduledTasks, @{ModuleName='StartScreen';ModuleVersion='1.0.0.0'}
Import-Module StartScreen
Get-StartApps
Get-ScheduledTask

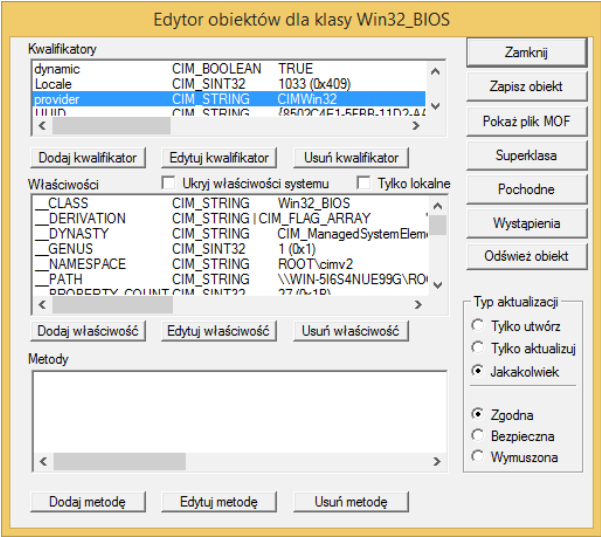
```

Postępowanie w przypadku braku dostawców WMI

Jedyny problem, jaki może wystąpić przy wykonywaniu skryptu *Get-Bios.ps1*, to brak lub uszkodzenie dostawcy WMI klasy *Win32_Bios*. Aby sprawdzić dostępność wybranego dostawcy WMI, trzeba znać jego nazwę dla danej klasy WMI. Do sprawdzenia nazwy można użyć Testera oprzyrządowania Instrumentacji zarządzania Windows (ang. *Windows Management Instrumentation Tester* — *WbemTest*) należącego do instalacji WMI. Jeśli w komputerze jest zainstalowana usługa WMI, to znajduje się w nim też plik *Wbemtest.exe*. Jako że program ten jest zlokalizowany w folderach systemowych, można go uruchomić bezpośrednio w konsoli Windows PowerShell, wpisując jego nazwę.

```
PS C:\> wbemtest
PS C:\>
```

Gdy pojawi się okno Testera oprzyrządowania Instrumentacji zarządzania Windows, należy połączyć się z odpowiednią przestrzenią nazw, klikając przycisk *Połącz*. W większości przypadków należy połączyć się z przestrzenią *root\cimv2*. Od systemu Windows Vista jest to domyślna przestrzeń nazw WMI w narzędziu *WbemTest*. W starszych wersjach systemu Windows domyślną przestrzenią nazw jest *root\default*. Zmień lub zatwierdź domyślną przestrzeń nazw i kliknij przycisk *Połącz*. Na ekranie pojawi się zestaw aktywnych przycisków, z których część będzie miała tajemniczo wyglądające nazwy. Aby zdobyć informacje o dostawcy WMI dla wybranej klasy, należy otworzyć tę klasę. Kliknij przycisk *Otwórz klasę* i w oknie dialogowym *Pobieranie nazwy klasy* wpisz nazwę klasy WMI. Interesuje nas nazwa dostawcy dla klasy *Win32_Bios*, a więc tę nazwę wpisujemy w polu tekstowym podpisanym *Wprowadź nazwę klasy docelowej*. Następnie kliknij przycisk *OK*, aby wyświetlić *Edytor obiektów dla klasy Win32_BIOS* widoczny na rysunku 12.4. Na górze tego okna znajduje się lista kwalifikatorów. Dostawca jest jednym z nich. Jak widać, dostawcą dla klasy *Win32_Bios* jest *CIMWin32*.



RYSUNEK 12.4. Okno Testera oprzyrządowania Instrumentacji zarządzania Windows z wyświetlonymi informacjami o dostawcy klasy

Mając nazwę dostawcy WMI, można za pomocą polecenia `Get-WmiObject` sprawdzić, czy dostawca ten jest zainstalowany w komputerze. Można w tym celu wysłać zapytanie o egzemplarze klasy `__provider`. W WMI od dwóch znaków podkreślenia zaczynają się nazwy klas systemowych. Klasa `__provider` stanowi bazę wszystkich dostawców. Ograniczając wyniki do dostawców o nazwie `CIMWin32`, można się dowiedzieć, czy dostawca ten jest zainstalowany w komputerze.

```
PS C:\> Get-WmiObject -Class __provider -filter "name = 'cimwin32'"
```

```
__GENUS                : 2
__CLASS                : __Win32Provider
__SUPERCLASS           : __Provider
__DYNASTY               : __SystemClass
__RELPATH               : __Win32Provider.Name="CIMWin32"
__PROPERTY_COUNT       : 24
__DERIVATION            : { __Provider, __SystemClass }
__SERVER                : WIN-5I6S4NUE99G
__NAMESPACE            : ROOT\cimv2
__PATH                 : \\WIN-5I6S4NUE99G\ROOT\cimv2:__Win32Provider.Name="CIMWin32"
ClientLoadableCLSID    :
CLSID                  : {d63a5850-8f16-11cf-9f47-00aa00bf345c}
Concurrency            :
DefaultMachineName     :
Enabled               :
HostingModel           : NetworkServiceHost
ImpersonationLevel     : 1
InitializationReentrancy : 0
InitializationTimeoutInterval :
InitializeAsAdminFirst :
Name                   : CIMWin32
OperationTimeoutInterval :
PerLocaleInitialization : False
PerUserInitialization  : False
Pure                   : True
SecurityDescriptor     :
SupportsExplicitShutdown :
SupportsExtendedStatus :
SupportsQuotas         :
SupportsSendStatus     :
SupportsShutdown       :
SupportsThrottling     :
UnloadTimeout          :
Version                :
```

```
PS C:\>
```

Aby sprawdzić, czy w komputerze znajduje się wybrany dostawca, nie trzeba pobierać tak dużo informacji. Lepiej potraktować zapytanie tak, jakby zwracało wartość logiczną w instrukcji `if`. Jeśli dostawca istnieje, zapytanie da się wykonać.

```
If(Get-WmiObject -Class __provider -filter "name = 'cimwin32'")
{
    Get-WmiObject -class Win32_bios
}
```

Jeśli dostawca `CIMWin32` nie istnieje, należy wyświetlić stosowną informację.

```
Else
{
    "Nie można wysłać zapytania do klasy Win32_Bios z powodu braku dostawcy."
}
```

Poniżej znajduje się kompletny kod źródłowy skryptu *CheckProviderThenQuery.ps1*:

```
CheckProviderThenQuery.ps1

If(Get-WmiObject -Class __provider -filter "name = 'cimwin32'")
{
    Get-WmiObject -class Win32_bios
}
Else
{
    "Nie można wysłać zapytania do klasy Win32_Bios z powodu braku dostawcy."
}
```

Lepszym sposobem na dowiedzenie się, czy istnieje określona klasa WMI, jest sprawdzenie istnienia dostawcy. Dostawcą klasy Win32_Product jest MSIProv. Napiszemy funkcję o nazwie *Get-WmiProvider* służącą do sprawdzania, czy w komputerze jest zainstalowany dowolny dostawca.

Funkcja *Get-WmiProvider* zawiera jeden parametr, reprezentujący nazwę dostawcy. Jako że funkcja ta przy wywołaniu z parametrem *-verbose* używa atrybutu *[cmdletbinding()]*, w konsoli zostają wyświetlone szczegółowe informacje, które mogą być przydatne podczas diagnostyki skryptu.

Po zadeklarowaniu funkcji należy poszukać dostawcy WMI. W tym celu za pomocą polecenia *Get-WmiObject* wysyłamy zapytanie o wszystkie egzemplarze systemowej klasy WMI *__provider*. Nie są one szczególnie interesujące dla administratorów, ale programista, który zada sobie trud, by je poznać, może mieć z nich spore korzyści. Wszyscy dostawcy WMI pochodzą od klasy *__provider*. Podobnie wszystkie przestrzenie nazw WMI pochodzą od klasy WMI *__Namespace*. W tabeli 12.2 znajduje się wykaz własności klasy *__provider*.

TABELA 12.2. Własności klasy *__provider*

| Nazwa własności | Typ własności |
|-------------------------------|----------------|
| ClientLoadableCLSID | System.String |
| CLSID | System.String |
| Concurrency | System.Int32 |
| DefaultMachineName | System.String |
| Enabled | System.Boolean |
| HostingModel | System.String |
| ImpersonationLevel | System.Int32 |
| InitializationReentrancy | System.Int32 |
| InitializationTimeoutInterval | System.String |
| InitializeAsAdminFirst | System.Boolean |
| Name | System.String |
| OperationTimeoutInterval | System.String |

TABELA 12.2. Własności klasy __provider — ciąg dalszy

| Nazwa własności | Typ własności |
|--------------------------|-----------------|
| PerLocaleInitialization | System.Boolean |
| PerUserInitialization | System.Boolean |
| Pure | System.Boolean |
| SecurityDescriptor | System.String |
| SupportsExplicitShutdown | System.Boolean |
| SupportsExtendedStatus | System.Boolean |
| SupportsQuotas | System.Boolean |
| SupportsSendStatus | System.Boolean |
| SupportsShutdown | System.Boolean |
| SupportsThrottling | System.Boolean |
| UnLoadTimeout | System.String |
| Version | System.UInt32 |
| __CLASS | System.String |
| __DERIVATION | System.String[] |
| __DYNASTY | System.String |
| __GENUS | System.Int32 |
| __NAMESPACE | System.String |
| __PATH | System.String |
| __PROPERTY_COUNT | System.Int32 |
| __RELPATH | System.String |
| __SERVER | System.String |
| __SUPERCLASS | System.String |

Za pomocą parametru `-filter` polecenia `Get-WmiObject` wybierany jest dostawca podany w zmiennej `$providerName`. Jeśli nazwa dostawcy jest nieznana, należy ją znaleźć za pomocą programu `WbemTest`. Aby go uruchomić, wystarczy wpisać jego nazwę w konsoli Windows PowerShell i nacisnąć klawisz `Enter`.

Pierwszą czynnością, jaką należy wykonać w programie `WbemTest`, jest połączenie się z odpowiednią przestrzenią nazw za pomocą przycisku *Połącz*. W większości przypadków odpowiednia przestrzeń nazw to `root\cimv2`. W związku z tym zmień lub zaakceptuj domyślną przestrzeń nazw i kliknij przycisk *Połącz*. Następnie kliknij przycisk *Otwórz klasę* i wpisz nazwę klasy w oknie dialogowym *Pobieranie nazwy klasy*, w polu tekstowym *Wprowadź nazwę klasy docelowej*. Interesuje nas nazwa dostawcy dla klasy `Win32_Product`, a więc tę nazwę wpisz w polu tekstowym *Wprowadź nazwę klasy docelowej*. Następnie kliknij przycisk *OK*, aby wyświetlić *Edytor obiektów dla klasy Win32_Product*. Na górze tego okna znajduje się lista kwalifikatorów. Dostawca jest jednym z nich. Jak widać, dostawcą dla klasy `Win32_Product` jest `MSIProv`.

Odkrytą nazwę dostawcy WMI przypisujemy do zmiennej `$providerName`.

```
$providerName = "MSIProv"
```

Otrzymany obiekt zapisujemy w zmiennej `$provider`.

```
$provider = Get-WmiObject -Class __provider -filter "name = '$providerName'"
```

Jeśli dostawca nie zostanie znaleziony, w zmiennej `$provider` nie zostanie zapisany żaden obiekt. Można więc to wykorzystać i sprawdzać, czy wartość tej zmiennej jest `null`. Jeśli nie, to można pobrać własność `CLSID` dostawcy. We własności tej przechowywany jest identyfikator klasy dostawcy WMI.

```
If($provider -ne $null)
{
    $clsID = $provider.CLSID
```

Gdy funkcja jest wywoływana z parametrem `-verbose`, zmienna `$verbosePreference` zostaje ustawiona na wartość `Continue`, co powoduje wyświetlenie na ekranie informacji przez polecenie `Write-Verbose`. Jeśli natomiast zmienna ta zostanie ustawiona na `SilentlyContinue`, polecenie `Write-Verbose` niczego nie wyświetli. Atrybut `[cmdletbinding()]` robi to automatycznie, dzięki czemu ułatwia implementację mechanizmów śledzenia bez pisania skomplikowanych konstrukcji warunkowych. Gdy funkcja zostanie wywołana z parametrem `-verbose`, zostaje wyświetlony identyfikator klasy dostawcy.

```
Write-Verbose "Znaleziono dostawcę WMI $providerName. Jego CLSID to $($CLSID)"
}
```

Jeśli dostawca nie zostanie znaleziony, funkcja zwraca fałsz do kodu wywołującego.

```
Else
{
    Return $false
}
```

Następnie funkcja sprawdza w rejestrze, czy dostawca WMI jest poprawnie zarejestrowany w DCOM. I ponownie polecenie `Write-Verbose` dostarcza szczegółowych informacji.

```
Write-Verbose "Sprawdzanie poprawności rejestracji w rejestrze..."
```

Do szukania informacji w rejestrze o rejestracji dostawcy WMI używa się dostawcy rejestru Windows PowerShell. Domyślnie nie istnieje dysk Windows PowerShell dla gałęzi rejestru `HKEY_CLASSES_ROOT`. Ale nie można zakładać, że nikt go nie utworzy w swoim profilu. Aby uniknąć potencjalnego błędu, który może wyniknąć podczas tworzenia dysku Windows PowerShell dla gałęzi `HKEY_CLASSES_ROOT`, za pomocą polecenia `Test-Path` sprawdzamy, czy istnieje dysk `HKCR`. Jeśli tak, to zostanie użyty i polecenie `Write-Verbose` wydrukuje dane dotyczące statusu, informujące o tym, że dysk `HKCR` został znaleziony i że rozpoczyna się szukanie identyfikatora klasy dostawcy WMI.

```
If(Test-Path -path HKCR:)
{
    Write-Verbose "HKCR: znaleziono dysk. Szukanie $clsID."
```

Aby dowiedzieć się, czy dostawca WMI jest zarejestrowany w DCOM, należy sprawdzić, czy identyfikator klasy tego dostawcy znajduje się w sekcji `CLSID` gałęzi `HKEY_CLASSES_ROOT`. Istnienie klucza rejestru najłatwiej jest sprawdzić za pomocą polecenia `Test-Path`.

```
Test-path -path (Join-Path -path HKCR:\CLSID -childpath $clsID)
}
```


Jeśli w komputerze nie ma dysku HKCR, można go utworzyć. Można sprawdzić, czy istnieje dysk o katalogu głównym w HKEY_Classes_Root, i jeśli uda się go znaleźć, użyć go w zapytaniu. Aby sprawdzić, czy istnieją jakiekolwiek dyski Windows PowerShell zakorzenione w HKEY_Classes_Root, można użyć polecenia Get-PSDrive.

```
Get-PSDrive | Where-Object { $_.root -match "classes" } |
Select-Object name
```

Szczerze mówiąc, z poleceniem Get-PSDrive jest sto pociech i tysięcy utrapień. Nie ma nic złego w zmapowaniu kilku dysków Windows PowerShell do tego samego źródła. Jeśli więc nie istnieje dysk HKCR, polecenie Write-Verbose drukuje informację, że dysk nie istnieje i że zostanie utworzony.

```
Else
{
    Write-Verbose "HKCR: nie znaleziono dysku. Zostanie on utworzony."
```

Do utworzenia nowego dysku Windows PowerShell służy polecenie New-PSDrive, w którym należy podać nazwę i lokalizację główną dysku. Jako że ma to być dysk rejestru, można użyć dostawcy rejestru. Po utworzeniu dysku w konsoli zostają wyświetlone informacje dotyczące przebiegu procesu.

```
PS C:\AutoDoc> New-PSDrive -Name HKCR -PSProvider registry -Root HKEYClasses_Root
Name      Provider      Root                      CurrentLocation
-----
HKCR      Registry      Hkey_Classes_Root
```

Informacje te mogą rozpraszać użytkownika, więc lepiej się ich pozbyć, przekazując wyniki do polecenia Out-Null.

```
New-PSDrive -Name HKCR -PSProvider registry -Root HKEY_Classes_Root | Out-Null
```

Po utworzeniu dysku rejestru Windows PowerShell można zacząć szukać identyfikatora klasy dostawcy WMI. Ale najpierw dobrze byłoby poinformować użytkownika, co się dzieje, za pomocą polecenia Write-Verbose.

```
Write-Verbose "Szukanie identyfikatora $clsID"
```

Do sprawdzania, czy istnieje identyfikator klasy dostawcy WMI, służy polecenie Test-Path. Do utworzenia ścieżki do klucza rejestru użyjemy polecenia Join-Path. Nadrzędna część ścieżki to gałąź CLSID dysku rejestru HKCR, a ścieżką podrzędną jest identyfikator klasy dostawcy WMI zapisany w zmiennej \$clsID.

```
Test-path -path (Join-Path -path HKCR:\CLSID -childpath $clsID)
```

Po sprawdzeniu za pomocą polecenia Test-Path, czy istnieje identyfikator klasy dostawcy WMI, wyświetlamy informację o zakończeniu operacji za pomocą polecenia Write-Verbose.

```
Write-Verbose "Test zakończony."
```

Nie jest dobrym zwyczajem wprowadzanie trwałych zmian w środowisku Windows PowerShell za pomocą skryptów. Jeśli więc skrypt utworzył dysk, to należy go potem usunąć. Za pomocą polecenia Write-Verbose wyświetlamy informację o aktualizacji statusu, a za pomocą polecenia

Remove-PSDrive usuwamy dysk rejestru HKCR. Żeby nie zająć całego okna konsoli, wynik usuwania dysku przekazujemy do polecenia Out-Null.

```
Write-Verbose "Usuwanie dysku HKCR"
Remove-PSDrive -Name HKCR | Out-Null
}
```

W punkcie początkowym skryptu przypisujemy wartość do zmiennej \$providerName.

```
$providerName = "MSIProv"
```

Następuje wywołanie funkcji Get-WmiProvider z przekazaniem jej nazwy dostawcy WMI zapisanej w zmiennej \$providerName oraz parametru -verbose. Instrukcja If została użyta dlatego, ponieważ funkcja Get-WmiProvider zwraca wartość logiczną.

```
if(Get-WmiProvider -providerName $providerName -verbose )
```

Jeżeli funkcja Get-WmiProvider zwróci prawdę, następuje wysłanie zapytania za pomocą polecenia Get-WmiObject do klasy WMI obsługiwanej przez dostawcę WMI.

```
{
  Get-WmiObject -class win32_product
}
```

Jeśli dostawca WMI nie zostanie znaleziony, w konsoli zostaje wyświetlona stosowna informacja.

```
else
{
  "Nie znaleziono dostawcy $providerName."
}
```

Poniżej znajduje się kompletny kod źródłowy skryptu *Get-WmiProviderFunction.ps1*:

Get-WmiProviderFunction.ps1

```
Function Get-WmiProvider
{
  [cmdletbinding()]
  Param ([string]$providerName)
  $provider = Get-WmiObject -Class __provider -filter "name = '$providerName'"
  If($provider -ne $null)
  {
    $clsID = $provider.clsID
    Write-Verbose "Znaleziono dostawcę WMI $providerName. Jego CLSID to $($CLSID)"
  }
  Else
  {
    Return $false
  }
  Write-Verbose "Checking for proper registry registration ..."
  If(Test-Path -path HKCR:)
  {
    Write-Verbose "HKCR: znaleziono dysk. Szukanie $clsID."
    Test-path -path (Join-Path -path HKCR:\CLSID -childpath $CLSID)
  }
  Else
  {
    Write-Verbose "Nie znaleziono dysku HKCR. Zostanie on utworzony."
```

```

New-PSDrive -Name HKCR -PSProvider registry -Root HKEY_Classes_Root | Out-Null
Write-Verbose "Szukanie identyfikatora $clsID"
Test-path -path (Join-Path -path HKCR:\CLSID -childpath $CLSID)
Write-Verbose "Test zakończony."
Write-Verbose "Usuwanie dysku HKCR."
Remove-PSDrive -Name HKCR | Out-Null
}
} #end funkcja Get-WmiProvider

# *** punkt początkowy skryptu ***
$providerName = "msiprov"

if(Get-WmiProvider -providerName $providerName -verbose )
{
    Get-WmiObject -class win32_product
}
else
{
    "Nie znaleziono dostawcy $providerName."
}

```

Niepoprawne typy danych

Są dwa sposoby na to, by umożliwić przekazywanie do skryptów wyłącznie danych odpowiedniego typu. Pierwszy polega na dostarczeniu ograniczonej liczby możliwości do wyboru. Natomiast druga metoda pozwala użytkownikowi wpisać dowolną wartość, którą następnie się sprawdza i przekazuje do dalszej części skryptu, jeśli spełnia określone warunki. W skrypcie *Get-ValidWmi-ClassFunction.ps1* znajduje się funkcja o nazwie *Get-ValidWmiClass*, sprawdzająca, czy wartość przekazana do skryptu jest poprawną nazwą klasy WMI. Mówiąc dokładniej, funkcja ta sprawdza, czy łańcuch przekazany w parametrze *-class* da się rzutować na poprawny egzemplarz klasy *.NET System.Management.ManagementClass*. Do konwersji łańcuchów na egzemplarze klasy *System.Management.ManagementClass* służy akcelerator typu *[WMICLASS]*. Jak widać poniżej, jeśli do zmiennej przypisze się łańcuch, zmienna ta staje się egzemplarzem klasy *System.String*. Za pomocą metody *GetType* wyświetlono informacje o typie obiektu zapisanego w zmiennej.

```

PS C:\> $class = "win32_bio"
PS C:\> $class.GetType()
IsPublic IsSerial Name                                     BaseType
-----
True     True     String                                     System.Object

```

Aby przekonwertować łańcuch na klasę WMI, można użyć akceleratora typu *[WMICLASS]*. Oczywiście łańcuch ten musi reprezentować poprawną nazwę klasy WMI. Jeżeli klasa, którą usiłujemy utworzyć na komputerze, nie istnieje, zostaje zgłoszony błąd.

```

PS C:\> $class = "win32_bio"
PS C:\> [wmiclass]$class
Cannot convert value "win32_bio" to type "System.Management.ManagementClass".
Error: "Not found "
At line:1 char:16
+ [wmiclass]$class <<<<

```

Skrypt *Get-ValidWmiClassFunction.ps1* zaczyna się od utworzenia dwóch parametrów wiersza poleceń. Pierwszy z nich nazywa się `-computer` i dzięki niemu skrypt może działać zarówno na komputerze zdalnym, jak i lokalnym. Drugi parametr to `-class` i służy on do przekazywania nazwy klasy WMI, do której skrypt ma wysłać zapytanie. Trzeci parametr umożliwia skryptowi przeglądanie innych przestrzeni nazw WMI. Wszystkie trzy parametry są łańcuchami.

```
Param (
    [string]$computer = $env:computername,
    [string]$class,
    [string]$namespace = "root\cimv2"
) #end param
```

Funkcja *Get-ValidWmiClass* służy do sprawdzania, czy wartość przekazana przez parametr `-class` jest poprawną nazwą klasy WMI w danym komputerze. Jest to ważne, ponieważ różne wersje systemu operacyjnego zawierają charakterystyczne dla nich klasy. Na przykład system Windows XP zawiera klasę WMI o nazwie *NetDiagnostics*, której nie znajdzie się w żadnej innej wersji tego systemu operacyjnego. Natomiast Windows XP nie zawiera klasy *Win32_Volume*, która znajduje się w systemie Windows Server 2003 i nowszych. Dlatego sprawdzenie, czy dana klasa WMI jest dostępna w zdalnym komputerze, jest dobrym sposobem na zapewnienie poprawnego działania skryptu.

Funkcja *Get-ValidWmiClass* zaczyna od pobrania bieżącej wartości zmiennej `$ErrorActionPreference`. Zmienna ta może mieć jedną z czterech wartości wyliczeniowych: *SilentlyContinue*, *Stop*, *Continue* oraz *Inquire*. Wartości te rządzą obsługą błędów przez konsolę Windows PowerShell. Jeżeli zmienna `$ErrorActionPreference` ma wartość *SilentlyContinue*, wszelkie błędy będą pomijane i skrypt będzie próbował wykonywać kolejne wiersze kodu. Ten sposób działania jest podobny do instrukcji *On Error Resume Next* w języku VBScript. Normalnie nie należy stosować tego ustawienia, ponieważ bardzo utrudnia diagnozowanie skryptów. Poza tym może spowodować nieprzewidywalne działanie skryptu, a nawet mieć poważne konsekwencje.

Wyobraź sobie, że piszesz skrypt, którego pierwszą czynnością jest utworzenie nowego katalogu na zdalnym serwerze. Później skrypt ten kopiuje na ów serwer wszystkie pliki z lokalnego katalogu i usuwa ten katalog. Teraz włączamy ustawienie `$ErrorActionPreference = SilentlyContinue` i uruchamiamy ten skrypt. Zdalny serwer jest niedostępny, więc pierwsze polecenie nie zostaje poprawnie wykonane. Drugie polecenie również nie może się udać, ponieważ nie może skopiować plików, natomiast trzecie polecenie zostaje wykonane poprawnie, w efekcie czego usunęliśmy wszystkie pliki, nie wykonując ich kopii zapasowej. Lepiej, żebyśmy mieli jakąś inną kopię zapasową najważniejszych danych. Jeśli zmienna `$ErrorActionPreference` zostanie ustawiona na wartość *SilentlyContinue*, należy samodzielnie obsługiwać potencjalne błędy.

Funkcja *Get-ValidWmiClass* sprawdza ustawienie zmiennej `$ErrorActionPreference` i zapisuje je w zmiennej `$oldErrorActionPreference`. Następnie ustawia zmienną `$ErrorActionPreference` na *SilentlyContinue*, ponieważ w czasie sprawdzania poprawności nazwy klasy WMI mogą wystąpić błędy. Później kasuje zawartość stosu błędów. Opisany proces zawiera się w trzech poniższych wierszach kodu:

```
$oldErrorActionPreference = $ErrorActionPreference
>ErrorActionPreference = "SilentlyContinue"
>Error.Clear()
```

Wartość zmiennej `$class` zostaje użyta z akceleratorem typu `[WMICLASS]` do utworzenia obiektu `System.Management.ManagementClass`. Jako że musi być możliwość wykonywania skryptu zarówno na zdalnym, jak i lokalnym komputerze, do utworzenia kompletnej ścieżki do potencjalnego obiektu zarządzania używana jest wartość zmiennej `$computer`. Przy łączeniu zmiennych w celu utworzenia ścieżki do klasy WMI przeszkadza dwukropek za zmienną `$namespace`. Rozwiązaniem jest napisanie wyrażenia wymuszającego ewaluację tej zmiennej przed wykonaniem konkatencji. Wyrażenie to jest oznaczone znakiem dolara i otoczone nawiasem.

```
[WMICLASS]"\\$computer\$($namespace):$class" | out-null
```

W celu dowiedzenia się, czy konwersja łańcucha na klasę `ManagementClass` przebiegła pomyślnie, należy sprawdzić rejestr błędów. Jako że zawartość tego rejestru została wcześniej skasowana, znalezienie w nim jakiegokolwiek błędu oznacza, że wykonanie polecenia się nie udało. Jeżeli zaistnieje błąd, funkcja `Get-ValidWmiClass` zwraca fałsz do kodu wywołującego. Jeśli nie ma błędu, funkcja zwraca prawdę.

```
If($Error.count) { Return $false } Else { Return $true }
```

Ostatnią czynnością w funkcji `Get-ValidWmiClass` jest oczyszczenie środowiska błędów. Najpierw kasujemy zawartość rejestru błędów, a następnie przywracamy oryginalną wartość zmiennej `$ErrorActionPreference`.

```
$Error.Clear()
$ErrorActionPreference = $oldErrorActionPreference
```

Ponadto w skrypcie *Get-ValidWmiClassFunction.ps1* znajduje się funkcja `Get-WmiInformation`. Przyjmuje ona wartości ze zmiennych `$computer`, `$class` i `$namespace` oraz przekazuje je do polecenia `Get-WmiObject`. Otrzymany obiekt klasy `ManagementObject` zostaje przekazany do polecenia `Format-List` w celu wyświetlenia wszystkich właściwości, których nazwy zaczynają się od liter od a do z.

```
Function Get-WmiInformation ([string]$computer, [string]$class, [string]$namespace)
{
    Get-WmiObject -class $class -computername $computer -namespace $namespace |
    Format-List -property [a-z]*
} #end funkcja Get-WmiInformation
```

W punkcie początkowym skryptu wywoływana jest funkcja `Get-ValidWmiClass`. Jeśli zwróci ona prawdę, to następuje wywołanie funkcji `Get-WmiInformation`. W przeciwnym razie następuje wyświetlenie nazw klasy, przestrzeni nazw oraz komputera. Informacje te można wykorzystać przy rozwiązywaniu problemów z pobieraniem danych WMI.

```
If(Get-ValidWmiClass -computer $computer -class $class -namespace $namespace)
{
    Get-WmiInformation -computer $computer -class $class -namespace $namespace
}
Else
{
    "W przestrzeni nazw $namespace na komputerze $computer nie ma klasy o nazwie $class."
}
```

Poniżej znajduje się kompletny kod źródłowy skryptu *Get-ValidWmiClassFunction.ps1*:

Get-ValidWmiClassFunction.ps1

```

Param (
    [string]$computer = $env:computername,
    [string]$class,
    [string]$namespace = "root\cimv2"
) #end param

Function Get-ValidWmiClass([string]$computer, [string]$class, [string]$namespace)
{
    $oldErrorActionPreference = $ErrorActionPreference
    $ErrorActionPreference = "SilentlyContinue"
    $Error.Clear()
    [wmi]$obj = ([wmi]"\\$computer\$($namespace):$class" | out-null)
    If($obj) { Return $true } Else { Return $false }
    $Error.Clear()
    $ErrorActionPreference = $oldErrorActionPreference
} #end funkcja Get-ValidWmiClass

Function Get-WmiInformation ([string]$computer, [string]$class, [string]$namespace)
{
    Get-WmiObject -class $class -computername $computer -namespace $namespace |
    Format-List -property [a-z]*
} #end funkcja Get-WmiInformation

# *** punkt początkowy skryptu ***

If(Get-ValidWmiClass -computer $computer -class $class -namespace $namespace)
{
    Get-WmiInformation -computer $computer -class $class -namespace $namespace
}
Else
{
    "W przestrzeni nazw $namespace na komputerze $computer nie ma klasy o nazwie $class."
}

```

Zapiski praktyka**Nauka technik obsługi błędów Windows PowerShell****Bill Stewart, administrator sieci****Moderator oficjalnego forum Scripting Guys**

W swojej karierze napisałem wiele skryptów Hosta skryptów systemu Windows (ang. *Windows Script Host* — WSH) w języku VBScript i mogę powiedzieć, że jedną z najsłabszych stron tego języka jest obsługa błędów. Na przykład: jeśli jakiś wiersz kodu spowoduje błąd, to zawsze następuje zakończenie wykonywania tego skryptu, chyba że wyłączy się domyślny mechanizm obsługi błędów za pomocą instrukcji `On Error Resume Next`. Jednak wyłączenie tego mechanizmu może mieć nieprzewidywalne skutki, ponieważ powoduje, że interpreter skryptów VBScript

pomija wszystkie kolejne wiersze kodu zawierające błędy. Fora internetowe są pełne pytań dotyczących problemów ze skryptami VBScript wynikających z użycia instrukcji `On Error Resume Next` przez programistów nierozumiejących, jak dokładnie działa mechanizm obsługi błędów w tym języku.

Dla porównania obsługa błędów w Windows PowerShell jest znacznie elastyczniejsza i bardziej rozbudowana niż w języku VBScript. W Windows PowerShell rozróżniane są błędy powodujące zamknięcie programu i niepowodujące zamknięcia programu, dzięki czemu można tworzyć bardziej zaawansowane konstrukcje obsługi błędów. Zrozumienie różnicy między tymi dwoma rodzajami błędów bardzo pomaga w pisaniu kodu obsługi błędów w skryptach.

Błędy niepowodujące zamknięcia programu zazwyczaj obsługują przez ustawienie zmiennej `$ErrorActionPreference` (lub parametru `-ErrorAction` polecenia) na wartość `SilentlyContinue`, a następnie sprawdzenie zmiennej `$?`.

```
get-item "C:\FileDoesNotExist.txt" -erroraction SilentlyContinue
if (-not $?) {
    write-host ("Wyjątek: " + $Error[0].Exception.GetType().FullName)
    write-host $Error[0].Exception.Message
}
```

Błędy powodujące zamknięcie programu obsługują za pomocą instrukcji `Try` i `Catch`.

```
try {
    $searcher = [WMISearcher] "select * from Win32_NonExistentClass"
    $searcher.Get()
}
catch [System.Management.Automation.RuntimeException] {
    write-host ("Wyjątek: " + $_.Exception.GetType().FullName)
    write-host $_.Exception.Message
}
```

W Windows PowerShell 1.0 błędy powodujące zamknięcie programu można było obsługiwać tylko za pomocą instrukcji `Trap`, ale konstrukcje `Try-Catch` są o wiele czytelniejsze i łatwiejsze w obsłudze.

Początkowo dziwiłem się, dlaczego w bloku `Catch` można obsługiwać tylko błędy powodujące zamknięcie programu. W istocie w bloku tym można obsługiwać błędy niepowodujące zamknięcia programu, ale pod warunkiem ustawienia zmiennej `$ErrorActionPreference` (lub parametru `-ErrorAction`) na `Stop`.

```
try {
    get-item "C:\FileDoesNotExist.txt" -ErrorAction Stop
}
catch {
    write-host ("Wyjątek: " + $_.Exception.GetType().FullName)
    write-host $_.Exception.Message
}
```

Gdyby z przykładu tego usunąć parametr `-ErrorAction`, polecenie `Get-Item` zgłosiłoby błąd niepowodujący zamknięcia programu i blok `Catch` zostałby zignorowany.

Ale z obsługą błędów niepowodujących zamknięcia programu za pomocą konstrukcji `Try-Catch` wiąże się jedna niedogodność. Jeśli zmienna `$ErrorActionPreference` zostanie ustawiona na `Stop` i błąd zostanie obsłużony w bloku `Catch`, wiadomość obiektu wyjątku będzie zawierała

następujący tekst: „Command execution stopped because the preference variable \"errorActionPreference\" or common parameter is set to Stop.” (Wykonywanie polecenia zostało zatrzymane, ponieważ zmienną preferencji `errorActionPreference` lub wspólny parametr ustawiono na wartość `Stop`). Jeśli nie przeszkadza Ci taki napis na początku powiadomienia o wyjątku (jeśli na przykład nie nadpisujesz powiadomienia o wyjątku gdzie indziej), to nie ma problemu. Ale ponieważ zazwyczaj wyświetlam powiadomienia o wyjątkach, wolę ustawiać zmienną `$errorActionPreference` na wartość `SilentlyContinue` i testować zmienną `$?`.

Błędy zakresu

Zakres dozwolonych wartości przyjmowanych od użytkownika jest ograniczony do pewnego przedziału. Jeśli przedział ten jest niewielki, to najlepiej przedstawić listę opcji do wyboru, taką jak pokazana wcześniej w podrozdziale „Ograniczanie możliwości wyboru”. Ale jeżeli przedział dozwolonych wartości jest szeroki, menu jest niepraktyczne. W takim przypadku lepiej zastosować sprawdzanie wartości granicznych.

Sposób użycia funkcji sprawdzającej wartości graniczne

Jedna z technik sprawdzania wartości granicznych polega na zdefiniowaniu funkcji sprawdzającej, czy dana wartość mieści się w dopuszczalnym przedziale. Implementacja takiej funkcji może bazować na tablicy skrótów zawierającej dopuszczalne wartości. Następnie za pomocą metody `-contains` można sprawdzać, czy wartość przekazana przez wiersz poleceń znajduje się na białej liście, czy nie. Jeśli tak, metoda `-contains` zwraca prawdę, a jeśli nie — fałsz. Funkcja `Check-AllowedValue` tworzy tablicę skrótów woluminów docelowego komputera. Następnie tablica ta jest używana do sprawdzania, czy wolumin przekazany w parametrze wiersza poleceń `-drive` rzeczywiście znajduje się w danym komputerze. Funkcja ta zwraca do kodu wywołującego logiczną prawdę lub fałsz. Poniżej znajduje się jej kod źródłowy:

```
Function Check-AllowedValue($drive, $computerName)
{
    Get-WmiObject -class Win32_Volume -computername $computerName |
    ForEach-Object { $drives += @{ $_.DriveLetter = $_.DriveLetter } }
    $drives.contains($drive)
} #end funkcja Check-AllowedValue
```

Jako że funkcja `Check-AllowedValue` zwraca wartość logiczną, do sprawdzenia, czy wartość przekazana w parametrze `-drive` jest dopuszczalna, można użyć instrukcji `If`. Jeżeli dana litera dysku zostanie znaleziona w tablicy `$drives` utworzonej w funkcji `Check-AllowedValue`, następuje wywołanie funkcji `Get-DiskInformation`. Jeżeli wartość parametru `-drive` nie zostanie znaleziona w tablicy skrótów, następuje wyświetlenie ostrzeżenia w konsoli Windows PowerShell i skrypt kończy działanie. Poniżej znajduje się kompletny kod źródłowy skryptu `GetDrivesCheckAllowedValue.ps1`:

GetDrivesCheckAllowedValue.ps1

```

Param(
    [Parameter(Mandatory=$true)]
    [string]$drive,
    [string]$computerName = $env:computerName
) #end param

Function Check-AllowedValue($drive, $computerName)
{
    Get-WmiObject -class Win32_Volume -computername $computerName|
    ForEach-Object { $drives += @{ $_.DriveLetter = $_.DriveLetter } }
    $drives.contains($drive)
} #end funkcja Check-AllowedValue

Function Get-DiskInformation($computerName,$drive)
{
    Get-WmiObject -class Win32_volume -computername $computername -filter "DriveLetter = '$drive'"
} #end funkcja Get-BiosName

# *** punkt początkowy skryptu ***

if(Check-AllowedValue -drive $drive -computername $computerName)
{
    Get-DiskInformation -computername $computerName -drive $drive
}
else
{
    Write-Host -foregroundcolor yellow "Wartość $drive jest niedozwolona:"
}

```

Ograniczanie dopuszczalnych wartości parametru

Od wersji 2.0 konsoli Windows PowerShell zakres dopuszczalnych wartości parametrów można definiować bezpośrednio przy definicjach parametrów w sekcji Param. Technika ta jest przydatna, gdy zbiór możliwych wartości jest nieduży. Atrybut parametru `ValidateRange` tworzy przedział dozwolonych liczb, ale można go wykorzystać także do utworzenia zakresu liter. Przy użyciu tej techniki można bardzo uprościć skrypt *GetDrivesCheckAllowedValue.ps1*, w którym trzeba utworzyć przedział dozwolonych liter dysków. Poniżej znajduje się odpowiednia instrukcja Param:

```

Param(
    [Parameter(Mandatory=$true)]
    [ValidateRange("c","f")]
    [string]$drive,
    [string]$computerName = $env:computerName
) #end param

```

Dzięki możliwości kontrolowania liter dysków, jakie można wpisać w wierszu poleceń, skrypt staje się prostszy i bardziej czytelny, ponieważ można się pozbyć funkcji weryfikującej wartości parametru. Ponadto w skrypcie *GetDrivesValidRange.ps1* potrzebna jest jeszcze jedna zmiana, polegająca na dołączeniu dwukropka za literą dysku. W skrypcie *GetDrivesCheckAllowedValue.ps1* dwukropek mógł być podawany wraz z literą dysku przez wiersz poleceń, ale z atrybutem `ValidateRange` tak się nie da zrobić. Sztuka z dołączaniem tego dwukropka polega na zastosowaniu sekwencji specjalnej.

```
-filter "DriveLetter = '$drive':"
```

Poniżej znajduje się kompletny kod źródłowy skryptu *GetDrivesValidRange.ps1*:

GetDrivesValidRange.ps1

```
Param(
    [Parameter(Mandatory=$true)]
    [ValidateRange("c","f")]
    [string]$drive,
    [string]$computerName = $env:computerName
) #end param

Function Get-DiskInformation($computerName,$drive)
{
    Get-WmiObject -class Win32_volume -computername $computername '
    -filter "DriveLetter = '$drive':"
} #end funkcja Get-BiosName

# *** punkt początkowy skryptu ***

Get-DiskInformation -computername $computerName -drive $drive
```

Dodatkowe źródła informacji

- Centrum skryptów portalu TechNet pod adresem <http://www.microsoft.com/technet/scriptcenter> zawiera wiele przykładów skryptów Windows PowerShell z obsługą błędów.
- Wszystkie skrypty opisane w tym rozdziale są dostępne do pobrania w repozytorium centrum skryptów portalu TechNet pod adresem <http://gallery.technet.microsoft.com/scriptcenter/PowerShell-40-Best-d9e16039>.

Rozdział 13

Testowanie skryptów

- Techniki testowania podstawowej składni
- Testowanie wydajności skryptów
- Sposób użycia parametrów standardowych
- Tworzenie dzienników za pomocą funkcji Start-Transcript
- Zaawansowane techniki testowania skryptów
- Dodatkowe źródła informacji

Jeśli poświęcisz pewną ilość czasu na napisanie skryptu, to powinieneś też znaleźć parę minut na jego przetestowanie. Ale skąd wiadomo, które części skryptu powinno się przetestować? Dla wielu specjalistów IT testowanie skryptów polega po prostu na ich uruchamianiu i obserwowaniu pojawiających się błędów. Jeżeli skrypt działa bezbłędnie, to uznają, że jest dobry. Ale z tego rozdziału dowiesz się, że testowanie skryptu to coś więcej niż tylko sprawdzanie, czy podczas działania nie zwraca on błędów. Dobrym zwyczajem przy testowaniu skryptów jest sprawdzanie podstawowej składni. Ponadto powinno się zmierzyć wydajność, aby dowiedzieć się, czy skrypt będzie spełniał wymagania środowiska, w którym będzie uruchamiany. Jeżeli skrypt przyjmuje parametry z wiersza poleceń, to dodatkowo powinno się przetestować obsługę tych parametrów.

Techniki testowania podstawowej składni

Podstawową składnię można przetestować, po prostu uruchamiając skrypt i obserwując, czy nie występują jakieś błędy. Ale jeśli trzeba sprawdzić więcej skryptów, to można napisać skrypt, który automatycznie wykona te testy. Skrypt *Test-ScriptHarness.ps1* przeszukuje folder w celu znalezienia wszystkich plików z rozszerzeniem *.ps1*, wykonuje każdy z nich, obserwując błędy, i rejestruje czas wykonywania każdego skryptu. Następnie zapisuje wyniki w pliku tekstowym i wyświetla raport.

Najpierw skrypt *Test-ScriptHarness.ps1* sprawdza, czy został uruchomiony w maszynie wirtualnej. Skrypt wykonujący dużą liczbę innych skryptów może spowodować duże szkody w stacji roboczej, w zależności od tego, co konkretnie robią uruchamiane przez niego skrypty. Jeśli na przykład jeden z nich uruchamia automatyczną instalację systemu Windows 8, to można przypadkowo skasować wszystkie dane i niechcący dostać świeżą instalację systemu. Jeśli skrypt uruchomi się w maszynie wirtualnej Hyper-V z włączonym cofaniem dysków, to ryzyko spowodowania szkód staje się minimalne.

Jako że klasa `WMI Win32_ComputerSystem` zwraca pojedynczy egzemplarz, można uzyskać bezpośrednio dostęp do własności tej klasy. To znaczy, że nie trzeba przekopywać się przez kolekcję egzemplarzy tej klasy, aby zdobyć wartość własności `model`. W Hyper-V własność `model` ma wartość `virtual machine`. Jeżeli więc własność ta ma jakąkolwiek inną wartość, skrypt wyświetla pytanie, czy na pewno chcemy go uruchomić. Implementacja tego monitu bazuje na poleceniu `Read-Host`. Jeśli w odpowiedzi na pytanie wpisujemy literę *n*, skrypt zostanie zamknięty. Każda inna odpowiedź oznacza zgodę na uruchomienie skryptu.

```
if((Get-WmiObject Win32_ComputerSystem).model -ne "virtual machine")
{
    $response = Read-Host -prompt "Ten skrypt najlepiej jest uruchamiać w maszynie wirtualnej.
    Czy chcesz kontynuować? <t / n>"
    if ($response -eq "n") { exit }
}
```

Ścieżka do katalogu ze skryptami Windows PowerShell jest zapisana w zmiennej `$path`. W zależności od planowanego sposobu uruchamiania skryptu *Test-ScriptHarness.ps1* zmienną tę można zamienić na parametr wiersza poleceń.

```
$path = "C:\bp"
```

Za pomocą statycznej metody `GetTempFileName` z klasy `.NET System.IO.Path` tworzymy tymczasowy plik w tymczasowym katalogu użytkownika. Ścieżkę do tego pliku zapisujemy w zmiennej `$report`. Poniżej znajduje się przykład takiej ścieżki do tymczasowego pliku:

```
C:\Users\administrator.NWTRADERS.000\AppData\Local\Temp\tmpC484.tmp
```

Jako że nazwa pliku jest losowo tworzona w każdym wywołaniu metody `GetTempFileName`, zachowujemy ją na później w zmiennej `$report`.

```
$report = [io.path]::GetTempFileName()
```

Użyte w poniższym kodzie polecenie `Get-ChildItem` tworzy listę wszystkich plików z rozszerzeniem *ps1* znajdujących się w folderze wskazywanym przez zmienną `$path`. Parametr `-recurse` jest niezbędny, aby polecenie to znalazło wszystkie pliki skryptów znajdujące się w danym folderze. Wyniki polecenia `Get-ChildItem` są następnie przekazywane do polecenia `ForEach-Object`.

```
Get-ChildItem -Path $path -Include *.ps1 -Recurse |
```

Dzięki użyciu parametru `-Begin` polecenie `ForEach-Object` jednokrotnie wykonuje działania dla każdego elementu pojawiającego się w potoku. W tym przykładzie rejestrowany jest czas rozpoczęcia wykonywania skryptu testującego, który zapisujemy w zmiennej `$stime`. Wartość automatycznej zmiennej `$ErrorActionPreference` zostaje ustawiona na `SilentlyContinue`, dzięki czemu żadne informacje o błędach nie będą wyświetlane, a jeśli wystąpi jakiś błąd, to nie spowoduje on zakończenia działania skryptu. W zmiennej `$report` zostaje zapisana wiadomość statusowa zawierająca informację o czasie rozpoczęcia testowania skryptu i ile czasu to zajęło.

```
ForEach-Object -Begin '
{
    $stime = Get-Date
    $ErrorActionPreference = "SilentlyContinue"
    "Testowanie skryptów w folderze $path $stime" |
    Out-File -append -FilePath $report
```

Parametr `-Process` występuje po razie dla każdego obiektu przechodzącego przez potok. Pierwszą czynnością w jego bloku jest skasowanie wszystkich błędów ze stosu błędów, aby było pewne, że wszelkie znalezione błędy dotyczą aktualnie testowanego skryptu. W zmiennej `$startTime` zostaje zapisana nowa wartość czasu, która zostanie wykorzystana do obliczenia, ile trwało wykonywanie danego skryptu. Do raportu zostaje dodany wpis zawierający nazwę skryptu oraz czas jego uruchomienia ze zmiennej `$startTime`. Nazwa skryptu zostaje pobrana ze zmiennej automatycznej `$_`, która odnosi się do bieżącego obiektu w potoku. Następnie wszystkie dane z tej sekcji zostają przekazane do polecenia `Out-File` z parametrem `-append`, który powoduje dołączenie informacji na końcu pliku `$report`, zamiast zastąpienia nimi już znajdujących się tam informacji.

```
} -Process '
{
$error.Clear()
$startTime = Get-Date
" Początek testowania skryptu $_ o godzinie $startTime" |
Out-File -append -FilePath $report
```

Teraz należy uruchomić skrypt znajdujący się aktualnie w potoku. W tym celu użyjemy polecenia `Invoke-Expression` z parametrem `-command` ustawionym na wartość zmiennej automatycznej `$_`.

```
Invoke-Expression -Command $_
```

Po zakończeniu działania skryptu należy zapisać, o której godzinie miało to miejsce. Czas zakończenia działania skryptu przekazujemy do polecenia `Out-File` z parametrem `-append`.

```
$endTime = Get-Date
" Testowanie skryptu $_ zakończono o godzinie $endTime." |
Out-File -append -FilePath $report
```

W następnej kolejności pobieramy liczbę błędów ze stosu błędów i zapisujemy ją w pliku `$report`. Jako że zmienna automatyczna `$error` zawiera obiekt, trzeba zastosować podwyrażenie (znak dolara i nawias), aby wymusić ewaluację własności `count` tego obiektu. Później wartość tę przekazujemy potokiem do polecenia `Out-File`.

```
" Liczba błędów wygenerowanych przez skrypt: $($error.Count)" |
Out-File -append -FilePath $report
```

Obiekt `DateTime` zapisany w zmiennej `$startTime` odejmujemy od obiektu `DateTime` zapisanego w zmiennej `$endTime`. Ponownie tworzymy podwyrażenie, aby spowodować ewaluację tej operacji. Gdybyśmy nie użyli podwyrażenia, to w celu połączenia łańcucha z obiektami `DateTime` konieczne byłoby zastosowanie konkatenacji. Wartość czasu otrzymana przez odjęcie czasu rozpoczęcia od czasu zakończenia zostaje przekazana do polecenia `Out-File` z zamiarem dodania jej do raportu. Po tej czynności blok przetwarzania polecenia `ForEach-Object` jest zakończony.

```
" Czas testowania tego skryptu: $($endTime - $startTime)" |
Out-File -append -FilePath $report
} -end '
```

Po zakończeniu wykonywania ostatniego skryptu następuje zapisanie czasu zakończenia testowania w zmiennej `$etime`. Potem ustawiamy zmienną `$errorActionPreference` z powrotem na wartość domyślną `Continue` i zapisujemy czas zakończenia w raporcie.

```
{
$etime = Get-Date
$ErrorActionPreference = "Continue"
"Zakończono testowanie wszystkich skryptów z folderu $path $etime" |
Out-File -append -FilePath $report
}
```

Na koniec trzeba zarejestrować sumaryczny czas trwania całego testu. W tym celu należy odjąć czas rozpoczęcia zapisany w zmiennej `$stime` od czasu zakończenia zapisanego w zmiennej `$etime`. Do wymuszenia ewaluacji tego czasu konieczne jest użycie podwyrażenia. Następnie otrzymana wartość zostaje przekazana do polecenia `Out-File` i następuje wyświetlenie raportu w Notatniku.

```
"Czas trwania testu: $($etime - $stime)" |
Out-File -append -FilePath $report
}
Notepad $report
```

Poniżej znajduje się kompletny kod źródłowy skryptu *Test-ScriptHarness.ps1*:

Test-ScriptHarness.ps1

```
if((Get-WmiObject win32_computersystem).model -ne "virtual machine")
{
    $response = Read-Host -prompt "Ten skrypt najlepiej jest uruchamiać w maszynie wirtualnej. Czy chcesz kontynuować? <t / n>"
    if ($response -eq "n") { exit }
}
$path = "C:\ScriptFolder"
$report = [io.path]::GetTempFileName()
Get-ChildItem -Path $path -Include *.ps1 -Recurse |
ForEach-Object -Begin '
{
    $stime = Get-Date
    $ErrorActionPreference = "SilentlyContinue"
    "Testowanie skryptów w folderze $path $stime" |
    Out-File -append -FilePath $report
} -Process '
{
    $error.Clear()
    $startTime = Get-Date
    " Początek testowania skryptu $_ o godzinie $startTime" |
    Out-File -append -FilePath $report
    Invoke-Expression -Command $_
    $endTime = Get-Date
    " Testowanie skryptu $_ zakończono o godzinie $endTime." |
    Out-File -append -FilePath $report
    " Liczba błędów wygenerowanych przez skrypt: $($error.Count)" |
    Out-File -append -FilePath $report
    " Czas testowania tego skryptu: $($endTime - $startTime)" |
    Out-File -append -FilePath $report
} -end '
{
    $etime = Get-Date
    $ErrorActionPreference = "Continue"
    "Zakończono testowanie wszystkich skryptów z folderu $path $etime" |
    Out-File -append -FilePath $report
    " Czas trwania testu: $($etime - $stime)" |
    Out-File -append -FilePath $report
}
```

}

Notepad \$report

Na rysunku 13.1 widać raport wygenerowany za pomocą tego skryptu.

```

Testowanie skryptów w folderze C:\ScriptFolder 10/24/2014 08:40:14
Początek testowania skryptu C:\ScriptFolder\Get-CommandLineOptions.ps1 o godzinie 10/24/2014 08:40:14
Testowanie skryptu C:\ScriptFolder\Get-CommandLineOptions.ps1 zakończono o godzinie 10/24/2014 08:40:14
Liczba błędów wygenerowanych przez skrypt: 0
Czas testowania tego skryptu: 00:00:00
Początek testowania skryptu C:\ScriptFolder\Get-ComputerWmiInformation.ps1 o godzinie 10/24/2014 08:40:14
Testowanie skryptu C:\ScriptFolder\Get-ComputerWmiInformation.ps1 zakończono o godzinie 10/24/2014 08:40:14
Liczba błędów wygenerowanych przez skrypt: 0
Czas testowania tego skryptu: 00:00:00
Początek testowania skryptu C:\ScriptFolder\Get-ModifiedFiles.ps1 o godzinie 10/24/2014 08:40:14
Testowanie skryptu C:\ScriptFolder\Get-ModifiedFiles.ps1 zakończono o godzinie 10/24/2014 08:40:14
Liczba błędów wygenerowanych przez skrypt: 1
Czas testowania tego skryptu: 00:00:00.0156590
Początek testowania skryptu C:\ScriptFolder\Get-ModifiedFilesUsePipeline.ps1 o godzinie 10/24/2014 08:40:14
Testowanie skryptu C:\ScriptFolder\Get-ModifiedFilesUsePipeline.ps1 zakończono o godzinie 10/24/2014 08:40:14
Liczba błędów wygenerowanych przez skrypt: 1
Czas testowania tego skryptu: 00:00:00
Początek testowania skryptu C:\ScriptFolder\Get-ModifiedFilesUsePipeline2.ps1 o godzinie 10/24/2014 08:40:14
Testowanie skryptu C:\ScriptFolder\Get-ModifiedFilesUsePipeline2.ps1 zakończono o godzinie 10/24/2014 08:40:14
Liczba błędów wygenerowanych przez skrypt: 2
Czas testowania tego skryptu: 00:00:21.3120303
Początek testowania skryptu C:\ScriptFolder\New-LocalGroupFunction.ps1 o godzinie 10/24/2014 08:40:36
Testowanie skryptu C:\ScriptFolder\New-LocalGroupFunction.ps1 zakończono o godzinie 10/24/2014 08:40:36
Liczba błędów wygenerowanych przez skrypt: 0
  
```

RYSUNEK 13.1. Raport wygenerowany przez skrypt Test-ScriptHarness.ps1

Szukanie błędów

Systematyczne testowanie skryptów z pewnością bezpośrednio nie skróci czasu pracy i prawie na pewno wydłuży czas tworzenia skryptu. Ale dzięki zredukowaniu liczby trudnych do wyjaśnienia błędów, które są cechą słabo napisanych skryptów, zapewne uda się oszczędzić sporo czasu w trakcie eksploatacji programu.

Najprostszą metodą testowania skryptów jest uruchamianie. Ale zanim się to zrobi, powinno się przejrzeć, czy w kodzie nie ma oczywistych błędów i usterek dotyczących funkcjonalności. Należy przyrzeć się każdej części skryptu. Poniżej znajduje się spis poszczególnych sekcji skryptu ze wskazówkami, na co należy zwracać uwagę. Czynności te należy wykonać zawsze przed uruchomieniem skryptu.

W sekcji Param skoncentruj się na następujących kwestiach:

- Parametry wiersza poleceń znajdują się w bloku zaczynającym się od słowa kluczowego Param.
- Każdy parametr jest oddzielony od poprzedniego przecinkiem.
- Za ostatnim parametrem nie ma przecinka.
- Nawias powinien mieć otwarcie i zamknięcie.

- Instrukcja Param powinna zajmować pierwszy wiersz w skrypcie niebędący komentarzem.
- Sprawdź obowiązkowe parametry i domyślne wartości parametrów.

Poniżej znajduje się poprawnie zbudowana instrukcja Param:

```
Param(
    [string]$computer=$env:computerName,
    [switch]$disk,
    [switch]$processor,
    [switch]$memory,
    [switch]$network,
    [switch]$video,
    [switch]$all
) #end param
```

W sekcji Function należy zwrócić uwagę na wiele szczegółów, między innymi:

- Definicja funkcji powinna zaczynać się od słowa kluczowego Function i nazwy funkcji.
- Nazwa funkcji powinna składać się z czasownika i rzeczownika, podobnie jak nazwy standardowych poleceń Windows PowerShell.
- Parametry wejściowe funkcji powinny być objęte nawiasem.
- Każdy parametr powinien być oddzielony od poprzedniego przecinkiem. Za ostatnim parametrem nie powinno być przecinka.
- Każdemu otwarciu klamry musi towarzyszyć zamknięcie.
- Jak funkcje są wywoływane w skrypcie? Jeśli jakaś funkcja nie jest używana w skrypcie, to nie powinna się w nim znajdować.
- Szczególną uwagę zwróć na parametry funkcji. Jakich typów parametrów wymaga funkcja?

Poniżej znajduje się przykład poprawnie zbudowanej funkcji:

```
Function Get-Disk($computer)
{
    Get-WmiObject -class Win32_LogicalDisk -computername $computer
} #end Get-Disk
```

Teraz przechodzimy do punktu początkowego skryptu. Jest to kod, który zostanie wykonany jako pierwszy po uruchomieniu programu (po instrukcji Param). Jest on bardzo ważny, ponieważ decyduje o istocie działania skryptu. Rozważ poniższe pytania.

- Co tak naprawdę robi kod początkowy?
- Jakie zmienne są inicjowane? Czy na końcu zmienne zostają zwolnione?
- Jakie stałe są deklarowane? W jakim zakresie te stałe są tworzone?
- Jakie obiekty są tworzone? Jakie metody i własności udostępniają?
- Co skrypt robi domyślnie? Co się stanie, gdy zostanie uruchomiony bez żadnych parametrów?
- Czy skrypt zawiera jakąś pomoc?
- Czy w pomocy tej znajdują się jakieś przykłady użycia skryptu?
- Jakiego typu dane zwraca skrypt? Czy przedstawia je na ekranie, wysyła do pliku tekstowego, bazy danych, na adres e-mail lub w jeszcze jakieś inne miejsce?

- Czy lokalizacja wyjściowa ze skryptu jest dostępna na stacji roboczej, na której jest on uruchamiany?

Wiedza tajemna

Testowanie skryptów w konsoli Windows PowerShell

James Brundage i Ibrahim Abdul Rahim, programiści

Microsoft Corporation

Najważniejszą rzeczą do zapamiętania przy testowaniu programów w Windows PowerShell jest to, że ogólnie rzecz biorąc, stanowi ono taką samą czynność jak automatyzacja systemu. Aby testować programy, należy wprowadzić system operacyjny w odpowiedni stan. (Na przykład należy uruchomić jakieś programy albo zmienić coś w rejestrze). Potem można zautomatyzować proces i sprawdzać, co się dzieje z systemem.

Jako że do automatycznego testowania programów i automatyzacji systemów używa się bardzo podobnych narzędzi, tester powinien umieć wykorzystywać przykłady dostarczane przez twórców skryptów i programistów C#. Konsola Windows PowerShell bezproblemowo działa z wszystkimi obiektami języków C# (poprzez polecenie `New-Object`) i VBScript (poprzez polecenie `New-Object` z parametrem `-ComObject`).

Kolejną ważną rzeczą, o jakiej nie należy zapominać przy testowaniu skryptów Windows PowerShell, jest to, że potrzebny jest system szkieletowy. Testy najczęściej automatyzuje się w obrębie jakiegoś ogólnego systemu. System ten wykonuje fragment kodu z pewnymi parametrami i zapisuje wyniki w dzienniku. Pisząc funkcję Windows PowerShell, mamy właśnie fragment kodu z parametrami, a w tym kodzie funkcję mogącą zapisywać dane w kilku dziennikach (wyjściowym, błędów, szczegółowym, diagnostycznym, ostrzeżeń, postępu, a nawet zdarzeń). Jako że Windows PowerShell ma wszystko, co jest potrzebne dobremu systemowi szkieletowemu, ze zintegrowanym środowiskiem skryptowym (ISE) włącznie, zazwyczaj jako testowego systemu szkieletowego używam tej konsoli.

Ponadto testując w Windows PowerShell, należy pamiętać o tym, jak ważna jest interaktywność. Kluczem do testowania w Windows PowerShell jest interakcja z interfejsami API, stronami internetowymi i interfejsami użytkownika, które wypróbowuje się w krótkich skryptach, a potem skrypty te włącza się do ogólnych bibliotek i testów automatycznych. Do obsługi interakcji zalecam używanie okienka poleceń konsoli Windows PowerShell ISE. Potem działające polecenia można przenieść do okienka skryptowego w celu utworzenia z nich testów.

Uruchamianie skryptu

Po szczegółowym przeanalizowaniu skryptu można go uruchomić. Zanim jednak zrobi się to po raz pierwszy, należy przemyśleć, jaki będzie to miało wpływ na komputer. (Skrypty najlepiej jest testować w maszynie wirtualnej z włączonym cofaniem dysków). Odpowiedz na poniższe pytania:

- Czy masz jakieś niezapisane prace?
- Czy zamknąłeś wszystkie niepotrzebne programy?
- Czy skrypt, nad którym pracujesz, jest zapisany?
- Czy masz kopię zapasową na zewnętrznym dysku skryptu, nad którym pracujesz? (Jeżeli skrypt skasuje zawartość komputera, to przynajmniej sprawdzisz, jak to się stało).
- Czy masz poprzednią działającą wersję skryptu? (Jeśli wprowadziłeś wiele zmian w skrypcie, to możesz nie być w stanie przywrócić poprzedniej wersji).
- Czy masz świeżą kopię zapasową danych z całej stacji roboczej?

Gdy uruchomisz skrypt *Get-ComputerWmiInformation.ps1*, zauważysz, że nie wyświetla on żadnych informacji: żadnych błędów, uwag, jakiegokolwiek pomocy. Aby dowiedzieć się, jak ten skrypt działa, trzeba przeanalizować parametry wiersza poleceń i jego punkt początkowy. Prawie wszystkie jego parametry są przełącznikami, a ich nazwy to *-disk*, *-processor*, *-memory* oraz *-all*. Na tej podstawie można przypuszczać, że skrypt pobiera informacje na temat sprzętu komputerowego. Z nazwy *Get-ComputerWmiInformation.ps1* można wywnioskować, że pobiera informacje przy użyciu WMI.

Punkt początkowy tego skryptu wywołuje funkcję *Get-CommandLineOptions*. Rzut oka na jej kod pozwala odkryć, że sprawdza wszystkie parametry wiersza poleceń i wywołuje odpowiednią inną funkcję. Jako że brakuje domyślnych działań, po uruchomieniu bez parametrów skrypt nie daje żadnego znaku. Poniżej znajduje się kod funkcji *Get-CommandLineOptions*:

```
Function Get-CommandLineOptions
{
    if($all)
    {
        Get-Disk($computer)
        Get-Processor($computer)
        Get-Memory($computer)
        Get-Network($computer)
        Get-Video($computer)
        exit
    } #end all

    if($disk)
    {
        Get-Disk($computer)
    } #end disk

    if($processor)
    {
        Get-Processor($computer)
    } #end processor

    if($memory)
    {
        Get-Memory($computer)
    } #end memory

    if($network)
    {
        Get-Network($computer)
    }
}
```

```

    } #end network

if($video)
{
    Get-Video($computer)
} #end video
} #end funkcja Get-CommandLineOptions

```

Wiele ścieżek testowych

Aby prawidłowo przetestować skrypt `Get-ComputerWmiInformation.ps1`, każdy z jego parametrów wiersza poleceń należałoby przetestować osobno, a potem wszystkie razem. Ponadto skrypt powinno uruchomić się zarówno zdalnie, jak i lokalnie w różnych systemach operacyjnych. Dobrym pomysłem może też być przetestowanie skryptu w Windows PowerShell 4.0 i 3.0. Jeśli okaże się, że skrypt działa tylko w konsoli Windows PowerShell 3.0, to powinno się dodać do niego instrukcję `#requires -version 3.0`. Jest to jedyna instrukcja, która może znajdować się nad instrukcją `Param`. Ale ma to związek z tym, że zaczyna się od znaku `#`, który oznacza komentarz.

Dokumentowanie pracy

Przebieg testowania skryptu powinno się dokumentować. Na działanie skryptu ma wpływ wiele czynników, np. wersja systemu operacyjnego, numer dodatku serwisowego, zainstalowane poprawki itp. Zanotuj wszystkie zainstalowane w komputerze programy, wliczając klientów zarządzania (np. Pakiet administracyjny programu Operations Manager). Zapisz też, czy skrypt uruchamiasz w edytorze skryptów, czy bezpośrednio w wierszu poleceń Windows PowerShell. Jeśli używasz 64-bitowego systemu operacyjnego, to przetestuj skrypt zarówno w 32-, jak i 64-bitowej wersji konsoli Windows PowerShell. Ponadto należy sprawdzić działanie skryptu zarówno z normalnymi, jak i administracyjnymi uprawnieniami.

Poniżej znajduje się kompletny kod źródłowy skryptu *Get-ComputerWmiInformation.ps1*:

Get-ComputerWmiInformation.ps1

```

Param(
    [string]$computer=$env:computerName,
    [switch]$disk,
    [switch]$processor,
    [switch]$memory,
    [switch]$network,
    [switch]$video,
    [switch]$all
) #end param

Function Get-Disk($computer)
{
    Get-WmiObject -class Win32_LogicalDisk -computername $computer
} #end Get-Disk

```

```

Function Get-Processor($computer)
{
    Get-WmiObject -class Win32_Processor -computername $computer
} #end Get-Processor

Function Get-Memory($computer)
{
    Get-WmiObject -class Win32_PhysicalMemory -computername $computer
} #end Get-Processor

Function Get-Network($computer)
{
    Get-WmiObject -class Win32_NetworkAdapter -computername $computer
} #end Get-Processor

Function Get-Video($computer)
{
    Get-WmiObject -class Win32_VideoController -computername $computer
} #end Get-Processor

Function Get-CommandLineOptions
{
    if($all)
    {
        Get-Disk($computer)
        Get-Processor($computer)
        Get-Memory($computer)
        Get-Network($computer)
        Get-Video($computer)
        exit
    } #end all

    if($disk)
    {
        Get-Disk($computer)
    } #end disk
    if($processor)
    {
        Get-Processor($computer)
    } #end processor

    if($memory)
    {
        Get-Memory($computer)
    } #end memory

    if($network)
    {
        Get-Network($computer)
    } #end network

    if($video)
    {
        Get-Video($computer)
    } #end video
}

```

```

} #end funkcja Get-CommandLineOptions

# *** punkt początkowy skryptu ***

Get-CommandLineOptions

```

Testowanie wydajności skryptów

Wiele osób popełnia błąd polegający na traktowaniu Windows PowerShell jak zwykłego języka skryptowego. Jeśli skrypt zawiera pewne konstrukcje, takie jak wczytujące zawartość plików i zapisujące wyniki w zmiennych, a potem iterujące przez te wyniki za pomocą instrukcji `ForEach`, jego wydajność może być niska. Poniżej znajduje się przykład takiej mało wydajnej konstrukcji:

```

$a = Get-Content -Path c:\fso\myfile.txt
Foreach ($i in $a)
{
    Write-Host $i
}

```

Taki kod równie dobrze można napisać w języku Microsoft Visual Basic, VBScript czy jakimkolwiek innym, ponieważ jest to typowy wzorzec. Ale w ten sposób nie wykorzystuje się maksymalnie możliwości konsoli Windows PowerShell. Poniżej znajduje się polecenie, którym można zastąpić omawiany przykład:

```
Get-Content -Path c:\fso\myfile.txt
```

Jednym z najbardziej przydatnych narzędzi konsoli Windows PowerShell jest potok i jeśli ktoś go nie wykorzystuje, to nie osiągnie optymalnych wyników. Potok nie musi wczytywać całej zawartości pliku, aby rozpocząć jej przetwarzanie. To pozwala zmniejszyć zużycie pamięci przy wczytywaniu dużych plików, ponieważ nie trzeba zapisywać ich zawartości w zmiennej. Asynchroniczny sposób działania i możliwość zredukowania ilości zużytej pamięci to cechy decydujące o tym, że potoku powinno używać się zawsze, gdy jest to możliwe.

Skoro wiadomo, że potok w Windows PowerShell zapewnia najlepszą wydajność, wydaje się oczywiste, że należy go używać we wszystkich skryptach. Ale to nie tak. Niektóre operacje, jak choćby przetwarzanie niewielkich plików i działania niewymagające użycia dużej ilości pamięci, można szybciej wykonać, stosując przedstawioną wcześniej technikę polegającą na zapisywaniu i przeglądaniu danych. Najlepiej przetestować dwie wersje skryptu i sprawdzić, która jest szybsza. W tym podrozdziale przyjrzymy się właśnie dwóm różnym wersjom jednego skryptu i wyłonimy zwycięzcę.

Zapisywanie i przeglądanie danych

Skrypt *Get-ModifiedFiles.ps1* oblicza, ile plików w wybranym folderze zostało zmodyfikowanych w określonym okresie. W bloku `Param` utworzono dwa parametry wiersza poleceń. Pierwszy z nich nazywa się `-path` i określa folder do przeszukania. Natomiast drugi parametr to `-days` i służy do określania daty, od której ma się zacząć liczenie zmodyfikowanych plików.

```
Param(
```

```
$path = "C:\data",
$days = 30
) #end param
```

Data początkowa musi być podana w postaci obiektu klasy `DateTime`. Obiekt ten, który udostępnia metodę `AddDays`, jest tworzony za pomocą polecenia `Get-Date`. Przekazując ujemną liczbę dni do dodania do obiektu `DateTime`, można utworzyć reprezentację momentu z przeszłości. Domyślnie skrypt cofa się o 30 dni.

```
$dteModified = (Get-Date).AddDays(-$days)
```

Polecenie `Get-ChildItem` tworzy kolekcję wszystkich plików i folderów znajdujących się w folderze określonym przez zmienną `$path`. Przełącznik `-recurse` sprawia, że polecenie przejrzysz także wszystkie podfoldery. Otrzymana kolekcja zostaje zapisana w zmiennej `$files`.

```
$files = Get-ChildItem -path $path -recurse
```

Kolekcja zapisana w pliku `$files` jest przeglądana za pomocą instrukcji `Foreach`. Jako enumerator wskazujący bieżącą pozycję w kolekcji używana jest zmienna `$file`. W pętli znajduje się instrukcja `If` porównująca własność `LastWriteTime` obiektu `DateTime` ze zmienną `$file` z wartością obiektu `DateTime` zapisanego w zmiennej `$dteModified`. Jeśli wartość własności `LastWriteTime` jest większa lub równa wartości obiektu `$dteModified`, następuje zwiększenie o jeden wartości zmiennej `$changedFiles`.

```
Foreach($file in $files)
{
    if($file.LastWriteTime -ge $dteModified)
    { $changedFiles ++ }
}
```

Ostatnią czynnością skryptu *Get-ModifidFiles.ps1* jest wyświetlenie w oknie konsoli liczby znalezionych zmodyfikowanych plików. Służy do tego poniższe polecenie:

```
"Od dnia $dteModified w folderze $path zmodyfikowano $changedFiles plików."
```

Poniżej znajduje się kompletny kod źródłowy skryptu *Get-ModifidFiles.ps1*:

Get-ModifidFiles.ps1

```
Param(
    $path = "D:",
    $days = 30
) #end param
$dteModified= (Get-Date).AddDays(-$days)
$files = Get-ChildItem -path $path -recurse

Foreach($file in $files)
{
    if($file.LastWriteTime -ge $dteModified)
    { $changedFiles ++ }
}

"Od dnia $dteModified w folderze $path zmodyfikowano $changedFiles plików."
```

W moim komputerze skrypt *Get-ModifidFiles.ps1* działa dość długo, ale jest to zrozumiałe, ponieważ dysk *D* ma pojemność około 60 GB i zawiera prawie 30 000 plików w 4000 folderów. Biorąc pod uwagę ilość pracy, jaką skrypt musi wykonać, można stwierdzić, że nie jest taki powolny.

Użycie potoku Windows PowerShell

W skrypcie *Get-ModifidFiles.ps1* można też wykorzystać potok Windows PowerShell. Instrukcja *Param* i kod tworzący obiekt *DateTime* w zmiennej *\$dteModified* pozostają bez zmian. Pierwsza modyfikacja polega na przekazaniu wyników polecenia *Get-ChildItem* do następnego polecenia zamiast do zmiennej *\$files*. Efekt zwiększenia wydajności jest dwójaki. Po pierwsze: teraz dalsze sekcje skryptu mogą rozpocząć pracę prawie natychmiast. Gdy wyniki polecenia *Get-ChildItem* były zapisywane w zmiennej, trzeba było w niej zapisać po kolei wszystkie 30 000 plików i 4000 folderów przed rozpoczęciem przetwarzania. Ponadto zmienna ta jest przechowywana w pamięci, co oznacza, że istnieje duże ryzyko wyczerpania pamięci, zanim zapisze się wszystkie pliki z dużego dysku. Zmiana na potok jest przedstawiona poniżej:

```
Get-ChildItem -path $path -recurse |
```

Zamiast instrukcji *Foreach* w skrypcie *Get-ModifidFilesUsePipeline.ps1* użyto polecenia *ForEach-Object*. Przyjmuje ono dane z potoku i jest bardziej elastyczne od instrukcji *Foreach*. Domyślnym parametrem polecenia *ForEach-Object* jest *-Process*. Każdy obiekt przechodzący przez potok jest dostępny za pośrednictwem zmiennej automatycznej *\$_*. W tym skrypcie pełni ona podobną rolę do zmiennej *\$file* ze skryptu *Get-ModifidFiles.ps1*. W instrukcji *If* potrzebna jest tylko jedna zmiana — zmienną *\$file* należy zamienić na *\$_*. Poniżej znajduje się sekcja polecenia *ForEach-Object* ze skryptu *Get-ModifidFilesUsePipeline.ps1*:

```
ForEach-Object {
    if($_.LastWriteTime -ge $dteModified)
    { $changedFiles ++ }
}
```

Informacja dla użytkownika jest taka sama jak w skrypcie *Get-ModifidFiles.ps1*. Poniżej znajduje się kompletny kod źródłowy skryptu *Get-ModifidFilesUsePipeline.ps1*:

Get-ModifidFilesUsePipeline.ps1

```
Param(
    $path = "D:",
    $days = 30
) #end param

$dteModified= (Get-Date).AddDays(-$days)
Get-ChildItem -path $path -recurse |
ForEach-Object {
    if($_.LastWriteTime -ge $dteModified)
    { $changedFiles ++ }
}

"Od dnia $dteModified w folderze $path zmodyfikowano $changedFiles plików."
```

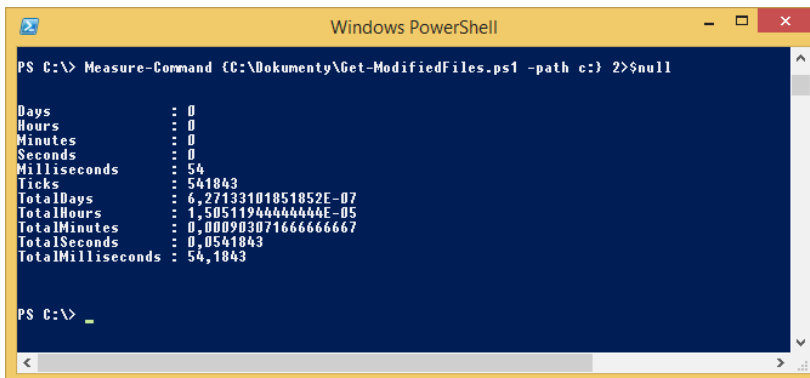
Porównanie szybkości działania dwóch skryptów

Gdy uruchomiłem skrypt *Get-ModifidFilesUsePipeline.ps1*, to sprawiał wrażenie, jakby działał trochę szybciej, ale trudno powiedzieć, czy na pewno. Czy wprowadzone zmiany były warte zachodu? Aby się o tym przekonać, można zmierzyć czas działania skryptu za pomocą polecenia *Measure-Command*. Najpierw sprawdzimy oryginalny skrypt, a potem jego zmodyfikowaną wersję. Aby zmierzyć czas wykonywania oryginalnego skryptu, należy przekazać ścieżkę do niego jako parametr *Expression* polecenia *Measure-Command*.

W przedstawionym poniżej poleceniu przekierowano strumień błędów za pomocą operatora *2>*. Wiem, że rekurencyjne przeglądanie katalogu *C* generuje błędy, ale nie chcę ich oglądać i dlatego skierowałem błędy do zmiennej *\$null*. Dzięki temu otrzymałem czytelne wyniki.

```
Measure-Command {C:\Dokumenty\Get-ModifiedFiles.ps1 -path c;} 2>$null
```

Polecenie *Measure-Command* zwraca obiekt klasy *.NET System.TimeSpan* służący do obliczania różnicy między wartościami dwóch obiektów klasy *System.DateTime*. Obiekt ten zawiera różne właściwości określające liczbę dni, godzin, minut, sekund i milisekund, których wartości są wyświetlane w oknie konsoli. Na rysunku 13.2 widać, że skryptowi *Get-ModifidFiles.ps1* praca zajęła 54 milisekundy albo dokładniej 54,1843 milisekundy, czyli 0,0541843 sekundy.



RYСУNEK 13.2. Wynik polecenia *Measure-Command*

Osoby nieobeznane z obiektami klasy *System.TimeSpan* pewnie zastanawiają się, dlaczego wszystkie wartości są przedstawione dwa razy. Ogólnie przy testowaniu skryptów najlepiej jest posługiwać się wartością *TotalSeconds*.

Teraz sprawdzimy, czy użycie potoku coś zmienia pod względem wydajności. Aby zmierzyć czas działania skryptu *Get-ModifidFilesUsePipeline.ps1*, należy przekazać ścieżkę do niego jako parametr *Expression* polecenia *Measure-Command*, jak pokazano poniżej:

```
PS C:\> Measure-Command {C:\Dokumenty\Get-ModifiedFilesUsePipeline.ps1 -path c;} 2>$null
```

Na rysunku 13.3 pokazano wynik tego testu.


```

Windows PowerShell

PS C:\> Measure-Command {C:\Dokumenty\Get-ModifiedFilesUsePipeline.ps1 -path c:\ 2>$null}

Days           : 0
Hours          : 0
Minutes        : 0
Seconds        : 0
Milliseconds    : 60
Ticks          : 603287
TotalDays      : 6.90248842592593E-07
TotalHours     : 1.67579722222222E-05
TotalMinutes   : 0.00100547033333333
TotalSeconds   : 0.00603287
TotalMilliseconds : 60,3287

PS C:\>

```

RYСУNEK 13.3. Wynik testu wydajności zmodyfikowanego skryptu

Jak widać na powyższym rysunku, wykonywanie skryptu *Get-ModifiedFilesUsePipeline.ps1* zajęło 60,3287 milisekundy, a więc więcej niż skryptu *Get-ModifiedFiles.ps1*. Przykład ten stanowi dowód na to, jak ważne jest testowanie wydajności. Czasami wyniki są niezgodne z oczekiwaniami i może się okazać, że czas poświęcony na modyfikacje jest stracony.

Upraszczenie kodu

Skrypt *Get-ModifiedFilesUsePipeline.ps1* można jeszcze bardziej zmodyfikować. Tym razem będzie to bardziej radykalna ingerencja w kod, ponieważ usuniemy polecenie `ForEach-Object` i instrukcję `If`. Pozbędziemy się poniższego fragmentu:

```

ForEach-Object {
    if($_.LastWriteTime -ge $dteModified)
    { $changedFiles ++ }
}

```

Usuwaając konstrukcje `ForEach-Object` i `If`, pozbędziemy się instrukcji `$changedFiles++` i wykorzystamy fakt, że konsola Windows PowerShell automatycznie zwraca obiekty z poleceń. Pojedyncze polecenie `Where-Object` powinno być szybsze niż bardziej skomplikowana konstrukcja `ForEach-Object` połączona z `If`. Ale efektywność zmian będzie można ocenić dopiero po przeprowadzeniu testu za pomocą polecenia `Measure-Object`. Redukując rozwiązanie do pojedynczego polecenia `Where-Object`, otrzymujemy następujący kod:

```
where-object { $_.LastWriteTime -ge $dteModified }
```

Wynik działania potoku jest zapisywany w zmiennej `$changedFiles`, która ma własność `count`. Bezpośredni odczyt własności `count` powinien być szybszy niż zwiększanie zmiennej `$changedFiles`, jak to robiliśmy w skrypcie *Get-ModifiedFilesUsePipeline.ps1*. Poniżej znajduje się kompletny kod źródłowy skryptu *Get-ModifiedFilesUsePipeline2.ps1*:

Get-ModifiedFilesUsePipeline2.ps1

```

Param(
    $path = "D:\",
    $days = 30
) #end param

```

```
$changedFiles = $null
$dteModified= (Get-Date).AddDays(-$days)
$changedFiles = Get-ChildItem -path $path -recurse |
where-object { $_.LastWriteTime -ge $dteModified }

"Od dnia $dteModified w folderze $path zmodyfikowano $changedFiles plików."
```

Wykonanie skryptu *Get-ModifiedFilesUsePipeline2.ps1* zajęło 73,0299 milisekundy, co jest jeszcze gorszym wynikiem niż poprzednio. Na rysunku 13.4 pokazano wynik pomiarów wydajności tego skryptu w konsoli.

```
Windows PowerShell

PS C:\> Measure-Command {C:\Dokumenty\Get-ModifiedFilesUsePipeline2.ps1 -path c:\ 2>$null}

Days           : 0
Hours          : 0
Minutes        : 0
Seconds        : 0
Milliseconds    : 73
Ticks          : 730299
TotalDays      : 8,45253472222222E-07
TotalHours     : 2,02608333333333E-05
TotalMinutes   : 0,001217165
TotalSeconds   : 0,00730299
TotalMilliseconds : 73,0299

PS C:\>
```

RYSUNEK 13.4. Wyniki pomiarów wskazujące, że wprowadzone zmiany jeszcze pogorszyły wydajność skryptu

Szacowanie wydajności różnych wersji skryptu

Za pomocą polecenia *Measure-Command* można w łatwy sposób przetestować wydajność skryptu i monitorować efekty niewielkich zmian. Jeśli jednak zmiany są szersze, powinno się tworzyć kolejne wersje skryptu. Aby ułatwić testowanie, można utworzyć skrypt testujący wydajność dwóch różnych skryptów. Aby uwzględnić różne czynniki wpływające na czas wykonywania programów, takie jak ładowanie, dostępność zasobów itp., powinno się wykonać testy kilka razy i sporządzić raport z wartości średnich.

Skrypt *Test-TwoScripts.ps1* testuje wydajność dwóch skryptów wiele razy. Następnie sporządza raport przedstawiający czas każdego wykonania i podsumowanie.

Parametry wiersza poleceń

W skrypcie *Test-TwoScripts.ps1* zdefiniowane są następujące parametry wiersza poleceń:

- Pierwszy parametr nazywa się *baseLineScript* i reprezentuje ścieżkę do skryptu, który będzie służył jako podstawa porównywania. Najczęściej jest to skrypt w stanie sprzed modyfikacji.
- Drugi parametr nazywa się *modifiedScript* i wskazuje ścieżkę do zmodyfikowanej wersji skryptu. Skrypty te nie muszą być w żaden sposób ze sobą powiązane.

- Trzeci parametr nazywa się `numberOfTests` i określa liczbę uruchomień skryptów. Dzięki wielokrotnemu wykonaniu skryptów i wyciągnięciu uśrednionych wartości można uzyskać bardziej wiarygodne wyniki.

Testowany skrypt za każdym razem może działać szybciej lub wolniej. Najczęściej ma to związek z buforowaniem plików i działaniem innych mechanizmów optymalizacji wydajności systemu operacyjnego, ale może również być spowodowane zablokowaniem zasobów lub innymi problemami.

- Ostatni parametr to przełącznik o nazwie `-log`. Jeśli zostanie podany, skrypt zapisze dane dotyczące wydajności w tymczasowym pliku tekstowym, którego zawartość wyświetli po zakończeniu pracy.

Poniżej znajduje się sekcja `Param` omawianego skryptu:

```
Param(
    [string]$baseLineScript,
    [string]$modifiedScript,
    [int]$numberOfTests = 20,
    [switch]$log
) #end param
```

Funkcje

W skrypcie znajduje się funkcja `Test-Scripts` wywołująca polecenie `Measure-Command` dla każdego z dwóch testowanych skryptów. W sekcji `Param` znajdują się definicje dwóch parametrów wejściowych: `baseLineScript` i `modifiedScript`. Zostały one skopiowane z analogicznej sekcji skryptu, bo tak było łatwiej, niż wpisywać wszystko od nowa. Poza tym dzięki skopiowaniu kodu mamy pewność, że nie popełnimy w nim żadnej literówki.

```
Function Test-Scripts
{
    Param(
        [string]$baseLineScript,
        [string]$modifiedScript
    ) #end param
```

Po definicjach parametrów znajdują się dwa wywołania polecenia `Measure-Command`.

W pierwszym wywołaniu do parametru `Expression` przekazywany jest skrypt bazowy. Łańcuch przekazywany do parametru `$baseLineScript` zawiera pełną ścieżkę do skryptu oraz wszystkie wymagane przez niego parametry. Drugie polecenie `Measure-Command` testuje wydajność zmodyfikowanej wersji skryptu. Do parametru `Expression` przekazywane są ścieżka do skryptu i wszystkie potrzebne mu do działania parametry.

```
Measure-Command -Expression { $baseLineScript }
Measure-Command -Expression { $modifiedScript }
} #end funkcja Test-Scripts
```

Funkcja `Get-Change` oblicza, o ile procent wydłużył lub skrócił się czas wykonywania zmodyfikowanego skryptu w porównaniu ze skrypcem bazowym. Parametr `baseLine` zawiera liczbę sekund, jaką zajęło wykonanie skryptu podstawowego. Parametr `modified` zawiera liczbę

sekund, jaką zajęło wykonanie zmodyfikowanego skryptu. Jeśli funkcja `Test-Scripts` zostanie wywołana kilka razy (z powodu wykonywania przez skrypt kilku testów), zmienne `$baseLine` i `$modified` będą zawierały sumę sekund wszystkich uruchomień każdego ze skryptów. Procentowa zmiana czasu wykonywania jest obliczana przez odjęcie wartości zmiennej `$modified` od wartości zmiennej `$baseLine` i podzielenie otrzymanej liczby przez wartość zmiennej `$baseLine`. Następnie wynik tego działania zostaje pomnożony przez sto. Poniżej znajduje się kod źródłowy funkcji `Get-Change`:

```
Function Get-Change($baseLine, $modified)
{
    (($baseLine - $modified)/$baseLine)*100
} #end funkcja Get-Change
```

Po funkcji `Get-Change` należy zdefiniować funkcję `Get-TempFile`. Będzie ona wywoływać statyczną metodę `GetTempFileName` z klasy `.NET IO.Path`. Poniżej znajduje się kod źródłowy funkcji `Get-TempFile`:

```
Function Get-TempFile
{
    [io.path]::GetTempFileName()
} #end funkcja Get-TempFile
```

Po utworzeniu wszystkich funkcji można przejść do punktu początkowego skryptu. Najpierw musimy sprawdzić, czy skrypt *Test-TwoScripts.ps1* został uruchomiony z przełącznikiem `-log`. Jeśli tak, to znaczy, że zmienna `$log` istnieje. A skoro zmienna ta istnieje, zostaje wywołana funkcja `Get-TempFile` i nazwa tymczasowego pliku zostaje zapisana w zmiennej `$logFile`.

```
if($log) { $logFile = Get-TempFile }
```

Do liczenia testów do wykonania na skryptach służy pętla `for`. Liczba testów jest zapisana w zmiennej `$numberOfTests`. W konsoli wyświetlany jest numer aktualnie wykonywanego testu. Poniżej znajduje się odpowiedzialna za to część kodu.

```
For($i = 0 ; $i -le $numberOfTests ; $i++)
{
    "Test $i z $numberOfTests" ; start-sleep -m 50 ; cls
```

Po wyświetleniu informacji o postępie testów następuje wywołanie funkcji `Test-Scripts`. Funkcja ta zwraca dwa obiekty klasy `System.TimeSpan`, które zostają zapisane w zmiennej `$results`.

```
$results= Test-Scripts -baseLineScript $baseLineScript -modifiedScript $modifiedScript
```

Jako że zmienna `$results` zawiera tablicę z dwoma obiektami klasy `TimeSpan`, można z niej wydobyć wartość zmiennej `$TotalSeconds` za pomocą indeksowania. Indeks `[0]` reprezentuje pierwszy obiekt, a indeks `[1]` — drugi. Zsumowaną liczbę sekund trwania bieżącego testu dodajemy do liczb sekund zapisanych w zmiennych `$baseLine` i `$modified`.

```
$baseLine += $results[0].TotalSeconds
$modified += $results[1].TotalSeconds
```

Jeśli skrypt zostanie uruchomiony z przełącznikiem `-log`, w pliku tekstowym zostaną zapisane nazwa skryptu, numer testu oraz wyniki. Poniżej znajduje się odpowiedzialny za to kod:

```
If($log)
{
    "Skrypt {$baseLineScript}: test $i z $numberOfTests $(get-date)" >> $logFile
    $results[0] >> $logFile
    "Skrypt {$modifiedScript}: test $i z $numberOfTests $(get-date)" >> $logFile
    $results[1] >> $logFile
} #if $log
} #for $i
```

Poniżej znajduje się kompletny kod źródłowy skryptu *Test-TwoScripts.ps1*:

Test-TwoScripts.ps1

```
Param(
    [string]$baseLineScript,
    [string]$modifiedScript,
    [int]$numberOfTests = 20,
    [switch]$log
) #end param

Function Test-Scripts
{
    Param(
        [string]$baseLineScript,
        [string]$modifiedScript,
        [int]$numberOfTests,
        [switch]$log
    ) #end param
    Measure-Command -Expression { $baseLineScript }
    Measure-Command -Expression { $modifiedScript }
} #end funkcja Test-Scripts

Function Get-Change($baseLine, $modified)
{
    (($baseLine - $modified)/$baseLine)*100
} #end funkcja Get-Change

Function Get-TempFile
{
    [io.path]::GetTempFileName()
} #end funkcja Get-TempFile
# *** punkt początkowy skryptu ***

if($log) { $logFile = Get-TempFile }
For($i = 0 ; $i -le $numberOfTests ; $i++)
{
    "Test $i z $numberOfTests" ; start-sleep -m 50 ; cls
    $results= Test-Scripts -baseLineScript $baseLineScript -modifiedScript $modifiedScript
    $baseLine += $results[0].TotalSeconds
    $modified += $results[1].TotalSeconds
    If($log)
    {
        "Skrypt {$baseLineScript}: test $i z $numberOfTests $(get-date)" >> $logFile
        $results[0] >> $logFile
        "Skrypt {$modifiedScript}: test $i z $numberOfTests $(get-date)" >> $logFile
    }
}
```

```

    $results[1] >> $logFile
} #if $log
} #for $i
"$Średnia zmiana dla $numberOfTests testów"
"$Średnia liczba sekund wykonywania skryptu {$baseLineScript}: $($baseLine/$numberOfTests)"
"$Średnia liczba sekund wykonywania skryptu {$modifiedScript}: $($modified/$numberOfTests)"
"$Zmiana procentowa: " + "{0:N2}" -f (Get-Change -baseLine $baseLine -modified $modified)
if($log)
{
    "$Średnia zmiana dla $numberOfTests testów" >> $logFile
    "$Średnia liczba sekund wykonywania skryptu {$baseLineScript}: $($baseLine/$numberOfTests)"
>> $logFile
    "$Średnia liczba sekund wykonywania skryptu {$modifiedScript}: $($modified/$numberOfTests)"
>> $logFile
    "$Zmiana procentowa: " + "{0:N2}" -f (Get-Change -baseLine $baseLine -modified $modified)
>> $logFile
} #if $log
if($log) { Notepad $logFile }

```

Wiedza tajemna

Testowanie interfejsów API oraz usług sieciowych, SOAP i REST przy użyciu konsoli Windows PowerShell

James Brundage i Ibrahim Abdul Rahim, programiści
Microsoft Corporation

Testowanie interfejsów API w Windows PowerShell jest chyba najłatwiejsze, ponieważ ma się możliwość interakcji z testowanym interfejsem. Testowanie API zazwyczaj polega na sprawdzeniu zwróconych wyników oraz zmian spowodowanych w systemie. Można łatwo tworzyć obiekty, wywoływać ich metody oraz pobierać wyniki do Windows PowerShell. Aby utworzyć obiekt klasy .NET, wystarczy użyć polecenia `New-Object`. Aby wczytać jakiś typ z dysku, należy załadować złożenie za pomocą instrukcji `[Reflection.Assembly]::LoadFrom($PełnaŚcieżka)`. Aby wykonać metodę statyczną lub pobrać wartość statycznej własności, można użyć instrukcji `[Typ]::WłasnośćLubMetoda`.

W konsoli Windows PowerShell 2.0 bardzo łatwe stało się testowanie usług sieciowych. Testuje się je podobnie jak interfejsy API w tym sensie, że najczęściej po prostu uruchamia się jakąś metodę i sprawdza się wynik. Oczywiście w testach usług sieciowych większą rolę odgrywa czas. Aby zmierzyć czas wykonywania testów, można użyć polecenia `Measure-Command`. A jeśli polecenie jest wykonywane za długo, można awaryjnie zakończyć test za pomocą instrukcji `Throw` lub polecenia `Write-Error`.

Także testowanie usług SOAP od Windows PowerShell 2.0 stało się bardzo łatwe dzięki możliwości użycia autowykrywania SOAP do utworzenia typu w celu użycia usługi sieciowej w PowerShell za pomocą polecenia `New-WebServiceProxy`. Jako że usługi sieciowe SOAP używają wielu różnych typów, zalecam używanie polecenia `New-WebServiceProxy` w konstrukcjach podobnych do poniższej:

```
$webService = New-WebServiceProxy -Uri $url -Namespace NazwaUsługiSieciowej
```

Po utworzeniu pośrednika usługi sieciowej można ją interaktywnie testować tak jak każdy inny interfejs API, tzn. wykonując różne czynności i sprawdzając wyniki oraz skutki uboczne.

Usługi REST nie są automatycznie wykrywalne, przez co ich testowanie jest trudniejsze niż w przypadku usług SOAP. Ale ponieważ łatwo jest wysłać zapytania do tych usług, można po prostu wysłać zapytanie i sprawdzić zwrócony wynik w formacie XML. Większość usług sieciowych typu REST pobiera parametry w postaci długich zapytań GET (takich jak widoczne w pasku adresu przeglądarki podczas wyszukiwania w wyszukiwarce internetowej).

Aby przetestować usługę typu REST w konsoli Windows PowerShell, należy najpierw napisać funkcję opakowującą tę usługę. Jeśli na przykład usługa przyjmuje łańcuch reprezentujący temat, liczbę artykułów do pobrania oraz offset, to należy napisać funkcję o sygnaturze pasującej do tej usługi (czyli `function GetRestService([string]$Topic, [int]$count = 20, [int]$offset = 0) {}`).

Funkcja opakowująca usługę REST jest prosta. Najczęściej trzeba tylko za pomocą polecenia `New-Object` utworzyć klienta sieciowego, pobrać wyniki za pomocą metody `DownloadString` i rzutować wyniki na obiekt typu XML.

```
($client = New-Object NET.Webclient; $client.DownloadString($url) -as [xml])
```

Następnie za pomocą polecenia `Select-Xml` można przeszukiwać dane XML zwrócone przez usługę sieciową, aby sprawdzić, czy są poprawne.

Wyświetlanie wyników i tworzenie dziennika

Po dokonaniu zapisu w dzienniku w konsoli Windows PowerShell należy wyświetlić liczbę testów i średni czas każdego testu zarówno dla skryptu podstawowego, jak i zmienionego. Poniżej znajduje się odpowiadająca za to część kodu:

```
"Średnia zmiana dla $numberOfTests testów"
"Średnia liczba sekund wykonywania skryptu {$baseLineScript}: $($baseLine/$numberOfTests)"
"Średnia liczba sekund wykonywania skryptu {$modifiedScript}: $($modified/$numberOfTests)"
```

Procentowa różnica czasu wykonywania skryptów jest obliczana przez funkcję `Get-Change`. Za pomocą specyfikatora formatu `{0:N2}` .NET ograniczono liczbę cyfr po przecinku do dwóch.

```
"Zmiana procentowa: " + "{0:N2}" -f (Get-Change -baseLine $baseLine -modified $modified)
```

Te same informacje co w konsoli zostają zapisane także w dzienniku, jeśli skrypt jest uruchomiony z przełącznikiem `-log`.

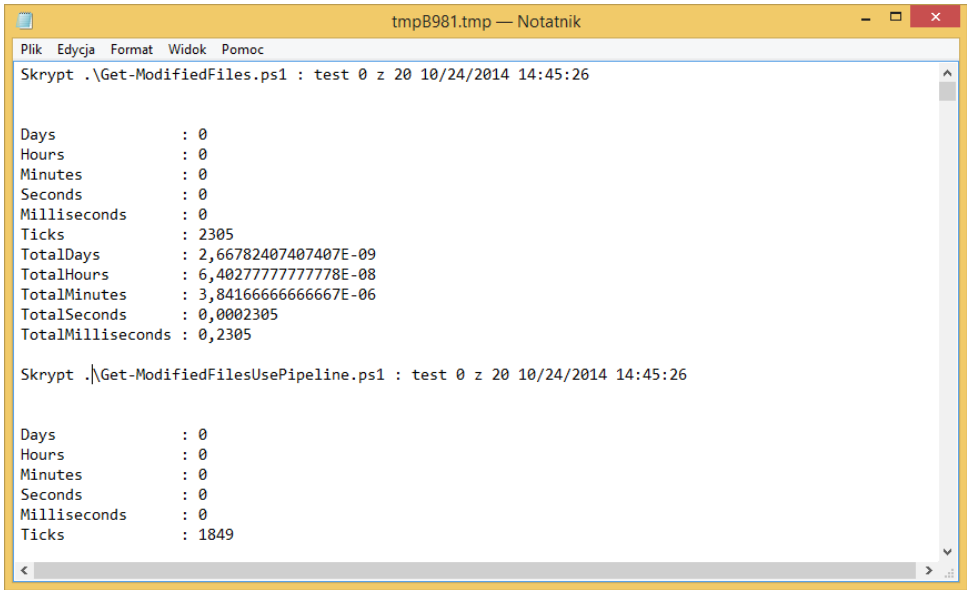
```
if($log)
{
    "Średnia zmiana dla $numberOfTests testów" >> $logFile
    "Średnia liczba sekund wykonywania skryptu $baseLineScript: $($baseLine/$numberOfTests)" >> $logFile
}
```

```
"Średnia liczba sekund wykonywania skryptu {$modifiedScript}: $($modified/$numberOfTests)" >> $logFile
"Zmiana procentowa: " + "{0:N2}" -f (Get-Change -baseLine $baseLine -modified $modified) >> $logFile
} #if $log
```

Po zapisaniu danych w dzienniku wyświetlamy je za pomocą Notatnika. Poniżej znajduje się odpowiedzialny za to wiersz kodu.

```
if($log) { Notepad $logFile }
```

Na rysunku 13.5 pokazano zawartość takiego dziennika.



RYSUNEK 13.5. Raport dotyczący wydajności dwóch skryptów dla 20 uruchomień

Wiedza tajemna

Testowanie aplikacji graficznych przy użyciu środowiska Windows PowerShell

James Brundage i Ibrahim Abdul Rahim, programiści

Microsoft Corporation

Najłatwiejszym sposobem na przetestowanie strony internetowej w Windows PowerShell jest napisanie skryptów obsługi przeglądarki Windows Internet Explorer. Ilość danych, jakie można zdobyć na temat strony, jest tak duża, że bardzo łatwo można określić, czy treść jest poprawna, czy nie. Przy automatyzacji tego procesu centralnym obiektem jest obiekt COM `Shell.Application` (`$Shell = New-Object -ComObject Shell.Application`). Za pomocą metody `Shell.Execute` tego obiektu można tworzyć nowe okna, a za pomocą metody `Shell.Windows()`

można uzyskać dostęp do wszystkich działających okien. Narzędzia programistyczne przeglądarki Internet Explorer (uruchamiane za pomocą klawisza *F12*) także są bardzo pomocne w sprawdzaniu, czy treść, którą chcemy zautomatyzować, znajduje się na stronie.

Podczas testowania stron internetowych należy zwrócić szczególną uwagę na dwie sprawy, z których jedną mogę trochę wyjaśnić. Każdą stronę internetową najpierw trzeba wczytać, a następnie ją wyrenderować, co oznacza, że proces wyświetlania strony przebiega etapami o określonym czasie trwania. W poniższym przykładzie najpierw czekamy na wczytanie strony *Bing.com*, a następnie na to, aż pole wyszukiwania będzie gotowe na przyjmowanie danych. Innym problemem, jaki twórca strony internetowej może nam sprawić, jest utworzenie strony w technologii Adobe Flash lub Microsoft Silverlight. W takim przypadku lepiej potraktować stronę jak zwykły interfejs użytkownika, który trzeba przetestować.

Poniżej znajduje się krótki test sprawdzający funkcję wyszukiwania w wyszukiwarce *Bing.com*:

```
$shell.ShellExecute("http://www.bing.com")
$timeout = New-TimeSpan -Seconds 15
$startTime = Get-Date
do {
    $window = $shell.Windows() | ? { $_.LocationUrl -eq "http://www.bing.com/" }
} while (-not $window -and
($startTime + $timeout) -gt (Get-Date))
if (-not $window) {
    throw "Minął czas oczekiwania na załadowanie okna."
}
$timeout = New-TimeSpan -Seconds 5
$startTime = Get-Date
do {
    $searchQuery = $window.Document.getElementById("sb_form_q")
} while (-not $searchQuery -and
($startTime + $timeout) -gt (Get-Date))
if (-not $searchQuery) {
    throw "Minął czas oczekiwania na pole wyszukiwania."
}
$searchQuery.InnerText = "foobar"
$window.Document.getElementById("sb_form_go").Click()
```

Konsola Windows PowerShell daje łatwy dostęp do aplikacji konsolowych i interfejsów API. Ale jeśli pracujemy z graficznym interfejsem użytkownika, Windows PowerShell musi przejść kilka warstw, aby zdobyć informacje o nim (np. tekst pola tekstowego albo czy pole wyboru jest zaznaczone) i wykonać określone polecenia, np. wysłać naciśnięcia klawiszy lub kliknięcia przycisków.

Podobnie jak jest w przypadku testowania stron internetowych, wszystko sprowadza się do znalezienia potrzebnych elementów i wykonania na nich odpowiednich czynności. Jako że technologie budowy interfejsu użytkownika mogą mieć różne implementacje, sztuczki zastosowane w jednym przypadku mogą nie zadziałać w innym. Do dyspozycji mamy wiele interfejsów API dających dostęp do stron, np. Microsoft Active Accessibility (MSAA), User Interface Automation (UIA) oraz API C systemu. API C Windows wymagają międzyoperacyjności (Add-Type i P/Invoke), natomiast MSAA i UIA mają API .NET i COM. Na początek do testowania interfejsów użytkownika można użyć bazującej na API C Windows przystawki Windows Automation for Windows PowerShell (<http://www.codeplex.com/WASP>).

Sztuka automatyzacji większości testów interfejsów użytkownika polega na posługiwaniu się klawiaturą zawsze, gdy to możliwe. Używając klawiatury, można skorzystać z `API SendKeys` (znajduje się w obiekcie `New-Object -ComObject WScript.Shell`), aby wysyłać sekwencje czynności jako naciśnięcia klawiszy. Później można zweryfikować zmiany w interfejsie użytkownika przez sprawdzenie pozycji ekranu i tekstu na platformie usług aplikacji sieciowej (ang. *Web Application Services Platform* — WASP).

Jeśli nie chcesz automatyzować testów, to inną techniką testowania stron internetowych jest utworzenie magazynu przypadków testowych do ręcznego wykonania. W tym przypadku konsola Windows PowerShell również ma wszystko, czego potrzeba do przeprowadzania testów. Są to przede wszystkim te dwa polecenia: `Read-Host` i `Write-Host`. Polecenie `Read-Host` pobiera dane od użytkownika, a `Write-Host` wyświetla informacje w oknie konsoli. Drukując kroki przypadku testowego za pomocą polecenia `Write-Host` i pobierając dane za pomocą polecenia `Read-Host`, można utworzyć ręcznie obsługiwane przypadki testowe do wykonania dla innych użytkowników.

Sposób użycia parametrów standardowych

W konsoli Windows PowerShell dostępne są dwa standardowe parametry, które mogą być przydatne podczas testowania skryptów: `debug` i `-whatif`. Parametr `debug` powoduje wyświetlanie w oknie konsoli szczegółowych informacji diagnostycznych przez polecenie `Write-Debug`. Natomiast parametr `-whatif` powoduje, że w konsoli wyświetlane są parametry przekazane do funkcji. Oba te parametry mogą dostarczyć cennych informacji diagnostycznych.

Sposób użycia parametru `debug`

W Windows PowerShell 4.0 parametr `debug` jest automatycznie dostępny, ponieważ jest standardowy. Aby go użyć, wystarczy do funkcji dodać instrukcję `[cmdletbinding()]`. Nie trzeba nic więcej robić.

Jeśli zmienna `$DebugPreference` jest ustawiona na wartość `Continue`, to polecenie `Write-Debug` powoduje wyświetlenie łańcucha w oknie konsoli. Jeśli natomiast jest ustawiona na wartość `SilentlyContinue` (jest to ustawienie domyślne), polecenie `Write-Debug` jest ignorowane, chyba że zostanie zastosowane wiązanie polecenia i funkcja będzie wywołana z parametrem `-Debug`. Ta różnorodność umożliwia wyświetlanie szczegółowych informacji na różne sposoby bez potrzeby tworzenia diagnostycznej i ostatecznej wersji skryptu. Dobrym zwyczajem jest wywoływanie polecenia `Write-Debug` przed każdym wywołaniem metody, podczas przypisywania wartości oraz przed wywołaniami funkcji.

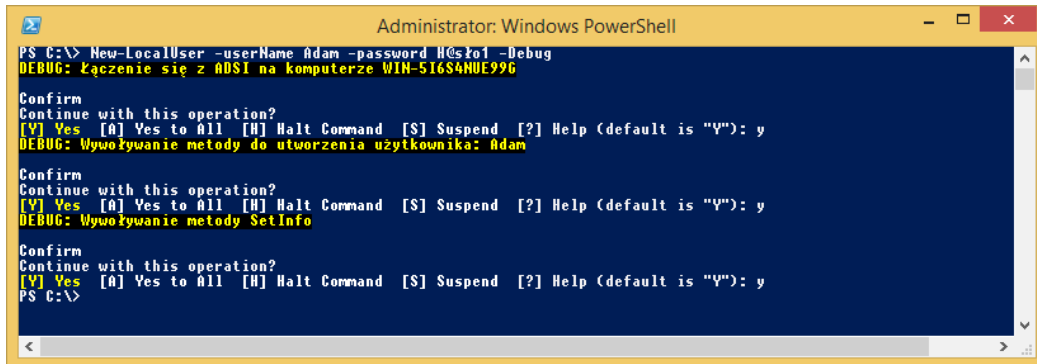
New-LocalUserFunction.ps1

```
Function New-LocalUser
{
    <#
        .Synopsis
        Funkcja tworząca lokalnego użytkownika.
```

```
.Description
Funkcja ta tworzy lokalnego użytkownika.
.Example
New-LocalUser -userName "ed" -description "świetny skrypciarz" '
    -password "hasło"
    Tworzy nowego użytkownika lokalnego o nazwie ed, opisie świetny skrypciarz i z hasłem hasło.
.Parameter ComputerName
Nazwa komputera, na którym ma zostać utworzony użytkownik
.Parameter UserName
Nazwa użytkownika, który ma zostać utworzony
.Parameter password
Hasło tworzonego użytkownika
.Parameter description
Opis utworzonego użytkownika
.Notes
NAZWA: New-LocalUser
AUTOR: ed wilson, msft
DATA MODYFIKACJI: 6.9.2013 10:07:42
SŁOWA KLUCZOWE: zarządzanie lokalnymi kontami, użytkownicy
HSG: oparte na HSG-06-30-11
Wymaga uprawnień administratora
.Link
Http://www.ScriptingGuys.com/blog
#>

[CmdletBinding()]
Param(
    [Parameter(Position=0,
        Mandatory=$True,
        ValueFromPipeline=$True)]
    [string]$userName,
    [Parameter(Position=1,
        Mandatory=$True,
        ValueFromPipeline=$True)]
    [string]$password,
    [string]$computerName = $env:ComputerName,
    [string]$description = "Utworzone przez PowerShell"
)
Write-Debug "Łączenie się z ADSI na komputerze $computerName"
$computer = [ADSI]"WinNT://$computerName"
Write-Debug "Wywoływanie metody do utworzenia użytkownika: $userName"
$user = $computer.Create("User", $userName)
$user.setpassword($password)
$user.put("description",$description)
Write-Debug "Wywoływanie metody SetInfo"
$user.SetInfo()
} #end funkcja New-LocalUser
```

Testując funkcję `New-LocalUser`, należy ją wywołać z parametrem debug. Wyświetlone zostaną informacje diagnostyczne i dodatkowo użytkownik będzie proszony o zatwierdzenie czynności, dzięki czemu będzie mógł pominąć problematyczne części kodu lub je wykonywać, jak widać na rysunku 13.6.



```

Administrator: Windows PowerShell
PS C:\> New-LocalUser -userName Adam -password H0s101 -Debug
DEBUG: Łączenie się z ADSI na komputerze WIN-S16S4NUE996

Confirm
Continue with this operation?
[Y] Yes [A] Yes to All [N] Halt Command [S] Suspend [?] Help (default is "Y"): y
DEBUG: Wywoływanie metody do utworzenia użytkownika: Adam

Confirm
Continue with this operation?
[Y] Yes [A] Yes to All [N] Halt Command [S] Suspend [?] Help (default is "Y"): y
DEBUG: Wywoływanie metody SetInfo

Confirm
Continue with this operation?
[Y] Yes [A] Yes to All [N] Halt Command [S] Suspend [?] Help (default is "Y"): y
PS C:\>

```

RYSUNEK 13.6. Informacje diagnostyczne wyświetlane po uruchomieniu skryptu z przełącznikiem debug

Sposób użycia parametru Verbose

Aby dowiedzieć się szczegółowo, co robi funkcja (lub skrypt), można użyć parametru Verbose. Różnica między nim a parametrem -Debug polega na tym, do czego służą zwracane informacje. Dane otrzymywane dzięki parametrowi -Debug służą do diagnostyki, natomiast dane otrzymywane dzięki parametrowi -Verbose mają na celu szczegółowo objaśnić użytkownikowi wszystkie wykonywane czynności. W istocie oba te zestawy informacji są do siebie podobne, a największa różnica polega na tym, że parametr -Debug umożliwia użytkownikowi podjęcie decyzji, czy wykonać, czy pominąć wybrane polecenia. Oznacza to, że każde polecenie Write-Debug wymaga od użytkownika podjęcia jakiejś czynności. Jeśli więc potrzebne są tylko szczegółowe informacje o działaniu funkcji, należy użyć polecenia Write-Verbose i wywołać funkcję z parametrem -Verbose.

Poniżej znajduje się kod źródłowy przykładowego skryptu o nazwie *Set-LocalGroupFunction.ps1*:

Set-LocalGroupFunction.ps1

```

Function Set-LocalGroup
{
    <#
    .Synopsis
    Funkcja dodająca i usuwająca lokalnego użytkownika z lokalnej grupy
    .Description
    Funkcja ta dodaje lub usuwa lokalnego użytkownika z lokalnej grupy
    .Example
    Set-LocalGroup -username "ed" -groupname "administrators" -add
    Przypisuje lokalnego użytkownika o nazwie ed do lokalnej grupy administratorów
    .Example
    Set-LocalGroup -username "ed" -groupname "administrators" -remove
    Usuwa lokalnego użytkownika o nazwie ed z lokalnej grupy administratorów
    .Parameter username
    Nazwa lokalnego użytkownika
    .Parameter groupname
    Nazwa grupy lokalnej
    .Parameter ComputerName
    Nazwa komputera
    .Parameter add
    Powoduje dodanie użytkownika przez funkcję
    .Parameter remove

```

```

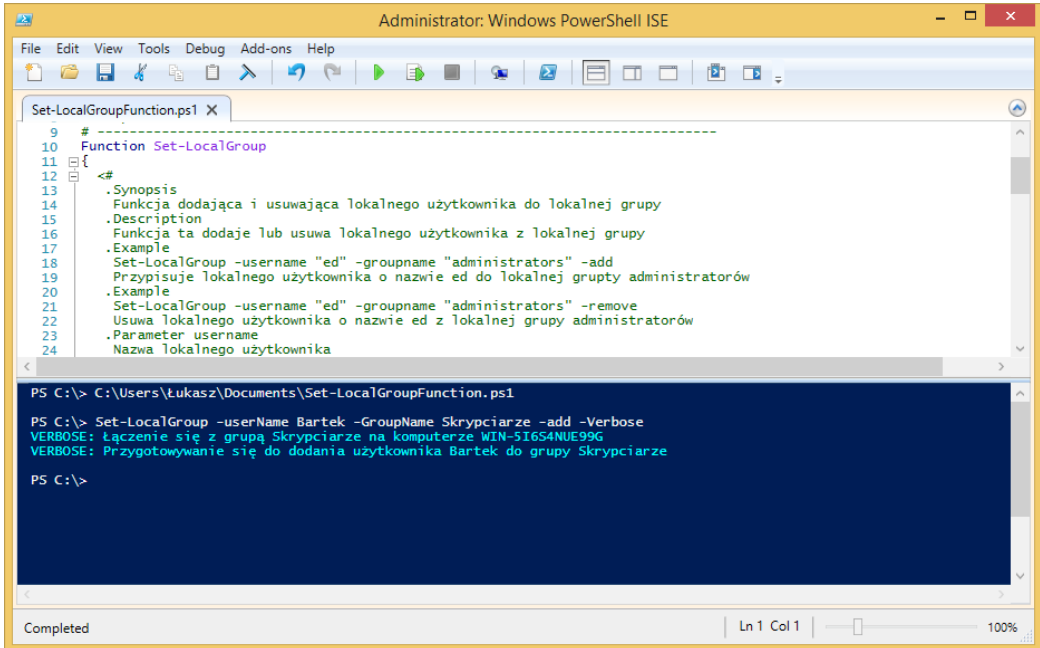
    Powoduje usunięcie użytkownika przez funkcję
.Notes
NAZWA: Set-LocalGroup
AUTOR: ed wilson, msft
DATA EDYCJI: 6.9.2013 10:23:53
WYMAGANIA: uprawnienia administratora
SŁOWA KLUCZOWE: zarządzanie kontami lokalnymi, użytkownicy, grupy
HSG: HSG-06-30-11
.Link
    Http://www.ScriptingGuys.com/blog
#Requires -Version 2.0
#>
[CmdletBinding()]
Param(
    [Parameter(Position=0,
        Mandatory=$True,
        ValueFromPipeline=$True)]
    [string]$UserName,
    [Parameter(Position=1,
        Mandatory=$True,
        ValueFromPipeline=$True)]
    [string]$GroupName,
    [string]$computerName = $env:ComputerName,
    [Parameter(ParameterSetName='addUser')]
    [switch]$add,
    [Parameter(ParameterSetName='removeuser')]
    [switch]$remove
)
Write-Verbose "Łączenie się z grupą $GroupName na komputerze $computerName"
$group = [ADSI]"WinNT://$ComputerName/$GroupName,group"
if($add)
{
    Write-Debug "Przygotowywanie się do dodania użytkownika $UserName do grupy $groupName"
    Write-Verbose "Przygotowywanie się do dodania użytkownika $UserName do grupy $groupName"
    $group.add("WinNT://$ComputerName/$UserName")
}
if($remove)
{
    Write-Debug "Przygotowywanie się do usunięcia użytkownika $UserName z grupy $groupName"
    Write-Verbose "Przygotowywanie się do usunięcia użytkownika $UserName z grupy $groupName"
    $group.remove("WinNT://$ComputerName/$UserName")
}
} #end funkcja Set-LocalGroup

```

Gdy funkcja Set-LocalGroup zostanie wywołana z parametrem -Verbose, podczas jej działania zostaną wykonane wszystkie instrukcje Write-Verbose, jak widać na rysunku 13.7.

Sposób użycia parametru whatif

Parametr -whatif nie jest tworzony automatycznie. Jeśli trzeba go zaimplementować w skrypcie, należy utworzyć przełącznik o takiej nazwie oraz funkcję wyświetlającą wszystkie parametry, które zostały przekazane do danej funkcji. Parametr -whatif powinno się implementować w każdym skrypcie, który zmienia stan systemu, czyli np. usuwa pliki lub foldery, tworzy pliki lub foldery albo zapisuje wartości w rejestrze. Wyświetlanie informacji, np. o pozycji kursora na ekranie, nie liczy się jako zmiana stanu systemu, więc nie wymaga implementacji parametru -whatif.



RYSUNEK 13.7. Szczegółowe informacje o wykonywaniu funkcji wyświetlone dzięki implementacji polecenia `Write-Verbose`

Pierwszą czynnością w implementacji parametru `-whatif` jest utworzenie przełącznika o nazwie `whatif`.

```

[CmdletBinding()]
Param(
    [Parameter(Position=0,
        Mandatory=$True,
        ValueFromPipeline=$True)]
    [string]$GroupName,
    [string]$computerName = $env:ComputerName,
    [string]$description = "Utworzono za pomocą PowerShell",
    [switch]$whatif)

```

Dodatkowo należy utworzyć instrukcję przyjmującą wszystkie parametry przekazane do funkcji. W prostej funkcji najłatwiej jest to zrealizować za pomocą instrukcji `if`. W łańcuchu wyjściowym należy wstawić wszystkie odpowiednie parametry. W instrukcji `if` można użyć instrukcji `exit`, ale to spowoduje zamknięcie okna konsoli, przez co nie da się przeczytać wyniku polecenia. Dlatego lepiej jest użyć instrukcji `Return`, jak pokazano poniżej:

```

If($whatif)
{
    "WHATIF: Tworzenie nowej lokalnej grupy $groupName z opisem $description na komputerze
    $computername."
    Return
} #end Whatif

```

Poniżej znajduje się kompletny kod źródłowy skryptu *New-LocalGroupFunction.ps1*:

New-LocalGroupFunction.ps1

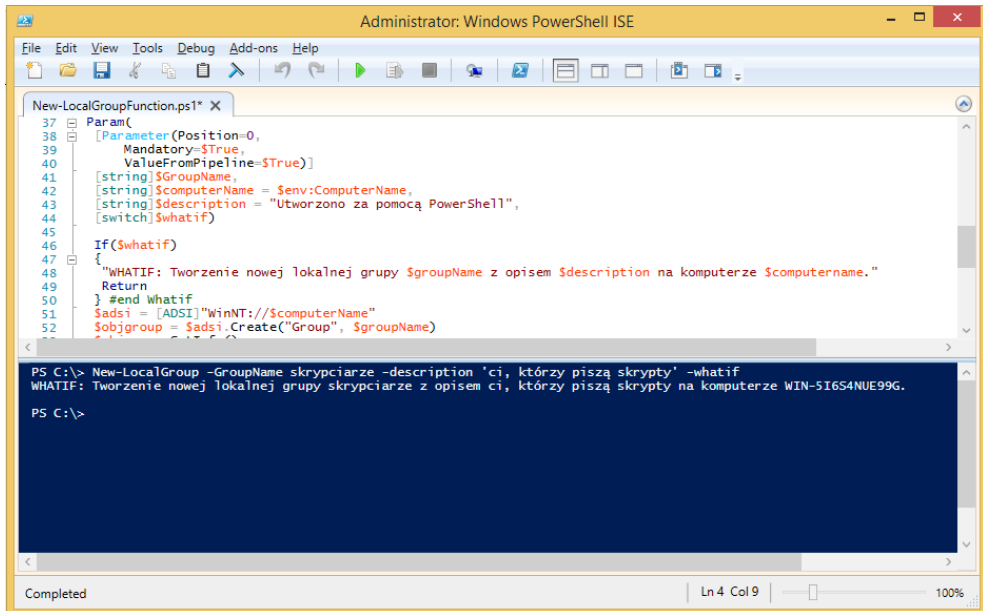
```

Function New-LocalGroup
{
    <#
    .Synopsis
        Funkcja tworząca grupę lokalną
    .Description
        Funkcja ta tworzy grupę lokalną
    .Example
        New-LocalGroup -GroupName "mojagrupa" -description "fajni lokalni użytkownicy"
        Tworzy nową grupę lokalną o nazwie mojagrupa z opisem fajni lokalni użytkownicy.
    .Parameter ComputerName
        Nazwa komputera, na którym ma zostać utworzona grupa
    .Parameter GroupName
        Nazwa grupy, która ma zostać utworzona
    .Parameter description
        Opis nowo utworzonej grupy
    .Notes
        NAZWA: New-LocalGroup
        AUTOR: ed wilson, msft
        DATA EDYCJI: 6.9.2013 10:07:42
        WYMAGANIA: uprawnienia administratora
        SŁOWA KLUCZOWE: zarządzanie kontami lokalnymi, grupy
        HSG: na bazie HSG-06-30-11
    .Link
        Http://www.ScriptingGuys.com/blog
    #>
    [CmdletBinding()]
    Param(
        [Parameter(Position=0,
            Mandatory=$True,
            ValueFromPipeline=$True)]
        [string]$GroupName,
        [string]$computerName = $env:ComputerName,
        [string]$description = "Utworzono za pomocą PowerShell",
        [switch]$whatif)

    If($whatif)
    {
        "WHATIF: Tworzenie nowej lokalnej grupy $groupName z opisem $description na komputerze
        $computername."
        Return
    } #end Whatif
    $adsi = [ADSI]"WinNT://$computerName"
    $objgroup = $adsi.Create("Group", $groupName)
    $objgroup.SetInfo()
    $objgroup.description = $description
    $objgroup.SetInfo()
} #end funkcja New-LocalGroup

```

Uruchomienie powyższego skryptu z parametrem `-whatif` powoduje wyświetlenie wyniku widocznego na rysunku 13.8.



RYSUNEK 13.8. Parametr `-whatif` powoduje wyświetlenie informacji o sposobie działania skryptu, ale go nie wykonuje

Zapiski praktyka

Testowanie skryptów

Enrique Cedeno, MCSE

Starszy administrator sieci

Jako starszy administrator sieci w dużej firmie świadczącej usługi dużo czasu poświęcam na pisanie skryptów, które są używane w serwerowych procesach kompilacji. Testowanie skryptu zawsze zaczynam od podzielenia go na funkcje i przetestowania każdej z nich indywidualnie.

Jeśli na przykład skrypt łączy się z bazą danych, to korzystam z testowej bazy danych zawierającej znane mi informacje. Każda funkcja jest traktowana tak, jakby była osobnym skryptem. Dzięki temu można łatwiej wykryć potencjalne problemy i usterki.

Jeden z członków naszego zespołu napisał kiedyś skrypt łączący się z bazą danych, ale zrobił błąd w instrukcji SQL. Mimo że w skrypcie nie było mechanizmu obsługi błędów, skrypt ten pozornie działał bezbłędnie. Ale gdy tylko uruchomiłem go z testową bazą danych, od razu zauważyłem, że zwraca bezsensowne dane. Skrypt ten był wzorowym przykładem kodu spaghetti i jego diagnozowanie zajęło nam trzy dni. To dowodzi, jak ważne jest posiadanie testowej bazy danych zawierającej znane informacje. Gdy wiadomo, jakiego typu dane powinno zwrócić zapytanie, łatwo można wykryć ewentualne nieprawidłowości w działaniu skryptu.

W skryptach produkcyjnych przechwytyjemy dane wyjściowe i zapisujemy je w dzienniku. Pobieramy szczegółowe dane diagnostyczne za pomocą polecenia `Write-Debug`. Jeśli trzeba przeprowadzić diagnostykę skryptu, używamy przełącznika zmieniającego wartość zmiennej `$DebugPreference` tak, aby włączyć polecenie `Write-Debug`.

Gdy mamy do czynienia z błędem logicznym w skrypcie, uruchamiamy go w znanym środowisku testowym i uważnie obserwujemy, czy zwracane wyniki są sensowne. Dzięki znajomości środowiska testowego wiemy, czy skrypt zwraca poprawne informacje, czy bzdury. Jeśli w grę wchodzi kilka funkcji i przeprowadzamy testy jednostkowe, to za pomocą polecenia `Write-Debug` informujemy w konsoli, kiedy następuje rozpoczęcie i zakończenie wykonywania każdej funkcji.

Jedną ze złotych zasad pisania skryptów jest hojne operowanie komentarzami. Są one potrzebne, bo po paru miesiącach po napisaniu skryptu niewiele już z niego pamiętamy. W komentarzach należy opisać wszystko, co jest nietypowe, a nawet logikę niektórych fragmentów kodu. Nie należy natomiast opisywać rzeczy oczywistych dla każdego, kto posługuje się konsolą Windows PowerShell. Na przykład nie ma sensu opisywać sposobu działania poleceń cmdlet. Natomiast zawsze powinno się wyjaśnić odkryte błędy i zastosowane rozwiązania.

Jeśli chodzi o rejestrowanie błędów skryptów, umieszczamy kod w skrypcie zapisującym wszystkie informacje diagnostyczne w dzienniku. Skryptu takiego używamy podczas prac, ponieważ nie zawsze można polegać na wyświetlaniu wiadomości o błędach na ekranie. Dodatkową korzyścią z przesyłania błędów do dziennika jest to, że metoda ta ułatwia automatyzację testów. Przed przekazaniem skryptu do produkcji wyłączamy szczegółowe rejestrowanie błędów, ale pozostawiamy kod, aby w razie potrzeby móc go włączyć za pomocą przełącznika.

Tworzenie dzienników za pomocą funkcji Start-Transcript

Jednym z najłatwiejszych sposobów na udokumentowanie wyników działania skryptu jest wywołanie funkcji `Start-Transcript`. Wprawdzie w ten sposób można uzyskać tylko ograniczoną ilość informacji, ale jest to łatwa metoda testowania i dokumentowania skryptów. Aby z niej skorzystać, należy za pomocą polecenia `Start-Transcript` utworzyć wykaz danych. Domyślnie polecenie to nadpisuje wszelkie pliki dzienników o takiej samej nazwie jak podana w parametrze `-path`. Ale można to zmienić za pomocą parametru `NoClobber`. Poniżej znajduje się wywołanie polecenia `Start-Transcript` ze skryptu *TranscriptBios.ps1*:

```
Start-Transcript -path $path
```

Jako że w dzienniku nie jest standardowo zapisywana nazwa skryptu, należy ją pobrać ze zmiennej `$myInvocation.InvocationName`. Polecenie `Start-Transcript` kopiuje do dziennika wszystko, co pojawia się w konsoli Windows PowerShell, więc łatwym sposobem na dodanie nazwy jest po prostu jej wyświetlenie. Jako że ważny może być też czas uruchomienia skryptu,

można dodać polecenie `Get-Date`, aby wyświetlić znacznik czasu, który również zostanie dodany do dziennika. Zmienną `$myInvocation.InvocationName` i polecenie `Get-Date` umieszczono w podwyrażeniach, aby wymusić ich ewaluację i zwrócenie wartości do łańcucha.

```
"Uruchamianie skryptu $($myInvocation.InvocationName) dnia $(Get-Date)"
```

Następnie wywołujemy funkcję `Get-Bios` będącą standardowym poleceniem WMI wykorzystującym polecenie `Get-WmiObject`. Ale do wyświetlenia nazwy funkcji użyta została zmienna `$myInvocation.InvocationName`. Technika pobierania nazwy wywołanej funkcji dostarcza cennych informacji o wynikach skryptu.

```
Function Get-Bios($computer)
{
    "Wywoływanie funkcji $($myInvocation.InvocationName)"
    Get-WmiObject -class win32_bios -computer $computer
}#end funkcja Get-Bios
```

Na koniec należy wyłączyć rejestrowanie danych za pomocą polecenia `Stop-Transcript` bez żadnych parametrów.

```
Stop-Transcript
```

Poniżej znajduje się kompletny kod źródłowy skryptu *TranscriptBios.ps1*:

TranscriptBios.ps1

```
Param(
    [Parameter(Mandatory=$true)]
    [string]$path,
    [string]$computer = $env:computername
)#end param

# *** funkcje ***

Function Get-Bios($computer)
{
    "Wywoływanie funkcji $($myInvocation.InvocationName)"
    Get-WmiObject -class win32_bios -computer $computer
}#end funkcja Get-Bios

# *** punkt początkowy skryptu ***

Start-Transcript -path $path
"Uruchamianie skryptu $($myInvocation.InvocationName) dnia $(Get-Date)"

Get-Bios -computer $computer
Stop-Transcript
```

Gdy uruchomimy powyższy skrypt, utworzy on w miejscu określonym w parametrze wywołania plik dziennika, którego zawartość pokazano na rysunku 13.9.

```

*****
Windows PowerShell transcript start
Start time: 20141027112228
Username : WIN-SI6S4NUE99G\tukasz
Machine : WIN-SI6S4NUE99G (Microsoft Windows NT 6.3.9600.0)
*****
Transcript started, output file is C:\fso\bios.txt
Uruchamianie skryptu .\TranscriptBios.ps1 dnia 10/27/2014 11:22:28
Wywoływanie funkcji Get-Bios

SMBIOSBIOSVersion : 6.00
Manufacturer : Phoenix Technologies LTD
Name : PhoenixBIOS 4.0 Release 6.0
SerialNumber : VMware-56 4d 07 76 e5 9e b0 e0-42 ed 0c 72 3a 7e 1d 74
Version : INTEL - 6040000

*****
Windows PowerShell transcript end
End time: 20141027112228
*****

```

RYSUNEK 13.9. Dziennik dokumentujący wyniki wykonania skryptu TranscriptBios.ps1

Zaawansowane techniki testowania skryptów

Podczas gdy uruchamiając skrypt i przyglądając się efektom, można wykryć błędy składniowe, np. braki zamknięcia klamer albo problemy z uprawnieniami, nie da się stwierdzić, czy skrypt będzie poprawnie spełniał swoje zadanie po przeniesieniu do środowiska produkcyjnego. Aby upewnić się, że nie wystąpią żadne problemy, skrypt należy przetestować w testowym środowisku o zbliżonej konfiguracji do środowiska produkcyjnego. Najlepiej, żeby środowisko testowe było wierną kopią środowiska produkcyjnego — nawet pod względem fizycznej infrastruktury i modeli serwerów o takich samych wersjach BIOS-u. Niektóre firmy utrzymują kopię infrastruktury na wypadek wystąpienia problemów. Często można ją wykorzystać do celów testowych.

Jeśli jednak ktoś nie ma luksusu w postaci duplikatu całego systemu, może go utworzyć przy użyciu maszyn wirtualnych.

Kopia całej sieci nie zawsze jest potrzebna, a przynajmniej nie wszystkich maszyn klienckich. W zależności od tego, co robi skrypt, często wystarczy odtworzyć kontroler domeny i parę serwerów.

Morał

Testowanie skryptów na znanych zbiorach danych

Enrique Cedeno, MCSE

Starszy administrator systemu

Kiedyś napisaliśmy skrypt sprawdzający, czy dana osoba należy do określonej grupy. Jeśli tak, to osoba ta musiała zmieniać hasło co 30 dni. Ale podczas prac ciągle otrzymywaliśmy od skryptu bezsensowne nazwy użytkowników. Po przyjrzeniu się problemowi dostrzegliśmy, że musimy zmodyfikować sposób zmieniania daty i najpierw ją odejmować. Gdybyśmy nie znali nazw użytkowników przypisanych do badanej grupy, to nie wykrylibyśmy błędu związanego z przetwarzaniem dat, co spowodowałoby wiele problemów i postawiłoby nas w bardzo złym świetle.

Istnieje możliwość napisania skryptu tworzącego użytkowników, grupy i jednostki organizacyjne składające się na typową implementację Active Directory, ale nie jest to konieczne. W każdej firmowej sieci regularnie tworzy się kopię zapasową kontrolerów domeny. Można więc przywrócić tę domenę na maszynie testowej. Dopóki maszyny testowe będą oddzielone od środowiska produkcyjnego, nie będzie żadnych problemów. Oczywiście w zależności od sposobu wykonania kopii zapasowych i sposobu przechowywania plików mogą występować trudności ze zgodnością sprzętową. Ale problemy te z reguły można rozwiązać. Zaletą tej techniki jest to, że hasła, identyfikatory zabezpieczeń (SID) itd. będą identyczne jak w środowisku produkcyjnym. Wadami są wymagania sprzętowe i ilość czasu potrzebnego na wykonanie i przywrócenie kopii zapasowej.

Szybszym rozwiązaniem jest użycie wbudowanych narzędzi do eksportu tylko interesującej części Active Directory. Są dwa takie narzędzia: CSVDE i LDIFDE. (Więcej informacji można znaleźć w artykule na stronie <http://support.microsoft.com/kb/237677>). Osobiście wolę narzędzie CSVDE, ponieważ tworzy plik wartości oddzielanych przecinkami, którego zawartość można łatwo wyczyścić w programie Microsoft Office Excel.

Kwestia czyszczenia zawartości pliku w przypadku używania tych dwóch narzędzi ma znaczenie, ponieważ dane nie są eksportowane do formatu wymaganego później w imporcie. W związku z tym operacje czyszczenia stają się koniecznością, a najlepsze jest to narzędzie, które jest najłatwiejsze w użyciu.

Za pomocą narzędzia CSVDE można wyeksportować wybraną jednostkę organizacyjną.

```
PS C:\fso> csvde -f testou.csv -d "ou=testou,dc=nwtraders,dc=com"
Connecting to "(null)"
Logging in as current user using SSPI
Exporting directory to file testou.csv
Searching for entries...
Writing out entries
...
Export Completed. Post-processing in progress...
3 entries exported
```

The command has completed successfully

Wyeksportowane dane przed zaimportowaniem na inny serwer należy wyczyścić, ponieważ niektóre z wyeksportowanych własności są własnościami tylko do odczytu kontrolowanymi przez serwer.

Wyczyszczony plik CSV można zaimportować do środowiska testowego. Aby zaimportować plik, należy użyć parametru `i`. (Eksport jest domyślną operacją narzędzia CSVDE, więc nie ma parametru eksportu).

```
PS C:\fso> csvde -f testou.csv -i
Connecting to "(null)"
Logging in as current user using SSPI
Importing directory from file "testou.csv"
Loading entries....
3 entries modified successfully.
```

The command has completed successfully

Morał

Obsługa haseł w maszynie wirtualnej

Ani narzędzie CSVDE, ani LDIFDE nie umożliwia wyeksportowania haseł z Active Directory. Jeśli więc skrypt testuje hasła albo mechanizmy uwierzytelniania, należy zastosować opisaną wcześniej technikę tworzenia i przywracania kopii zapasowej. Jeśli podczas testu rejestrowane jest, co się dzieje z wybranymi użytkownikami i grupami, to technika polegająca na użyciu narzędzia CSVDE powinna być odpowiednim rozwiązaniem. Jako że użytkownicy są eksportowani bez haseł, mamy dwie możliwości do wyboru. Pierwsza polega na zaimportowaniu wszystkich użytkowników w wyłączonym stanie, a następnie włączeniu ich za pomocą innego skryptu i ustawieniu wszystkim tego samego hasła. Ale jest to dość czasochłonne. Lepszym rozwiązaniem jest zmienienie reguły obsługi haseł w maszynie wirtualnej tak, aby można było używać pustych haseł, co umożliwi poprawne działanie skryptom CSVDE.

Dodatkowe źródła informacji

- Centrum skryptów portalu TechNet pod adresem <http://www.microsoft.com/technet/scriptcenter> zawiera wiele przykładów skryptów Windows PowerShell.
- Więcej informacji na temat narzędzi CSVDE i LDIFDE znajduje się na stronie <http://support.microsoft.com/kb/237677>.
- Wszystkie skrypty opisane w tym rozdziale są dostępne do pobrania w repozytorium centrum skryptów portalu TechNet pod adresem <http://gallery.technet.microsoft.com/scriptcenter/PowerShell-40-Best-d9e16039>.

Rozdział 14

Dokumentowanie skryptów

- Pobieranie dokumentacji z pomocy
- Pobieranie dokumentacji z komentarzy
- Sposób użycia parsera AST
- Dodatkowe źródła informacji

Jedną z wielkich zalet konsoli Windows PowerShell są metody sporządzania dokumentacji skryptów. W rozdziale tym poznasz różne techniki dokumentowania swoich programów. Najpierw przyjrzymy się technikom pobierania dokumentacji z pomocy, a następnie omówię różne sposoby wydobywania informacji wprost ze skryptów.

Pobieranie dokumentacji z pomocy

Skrypty dokumentuje się z wielu powodów. Robi się to w celu spełnienia postawionych wymagań, ułatwienia obsługi skryptu albo kontroli wersji. Najłatwiej można pobrać dokumentację wprost ze skryptu. Podczas gdy sposobów na umieszczenie informacji w skrypcie jest kilka, najbardziej niezawodna metoda polega na wykorzystaniu komentarzy. Jeśli skrypty mają zaimplementowaną pomoc komentarzową bazującą na standardowych znacznikach, to spełnienie wymagań dokumentacyjnych powinno być łatwe. Oczywiście warunkiem jest stosowanie jednolitej pomocy komentarzowej we wszystkich skryptach produkcyjnych.

Skrypt *Get-Scripthelp.ps1* analizuje katalog zawierający skrypty, na każdym skrypcie wywołuje funkcję *Get-Help*, a następnie zapisuje wyniki w dzienniku. Poniżej znajduje się kompletny kod źródłowy skryptu *Get-ScriptHelp.ps1*:

Get-ScriptHelp.ps1

```
function New-Underline
{
    <#
    .Synopsis
    Podkreśla podany na wejściu łańcuch
    .Example
    New-Underline -strIN "Witaj, świecie"
    .Example
    New-Underline -strIn "Morgen welt" -char "-" -sColor "blue" -uColor "yellow"
    .Example
```

```

"to jest łańcuch" | New-Underline
.Notes
NAZWA: New-Underline
AUTOR: Ed Wilson
DATA EDYCJI: 9.9.2013
SŁOWA KLUCZOWE: narzędzie pomocnicze
.Link
Http://www.ScriptingGuys.com
#>
[CmdletBinding()]
param(
    [Parameter(Mandatory = $true, Position = 0, valueFromPipeline=$true)]
    [string]
    $strIN,
    [string]
    $char = "-",
    [string]
    $sColor = "Green",
    [string]
    $uColor = "darkGreen",
    [switch]
    $pipe
) #end param
$strLine= $char * $strIn.length
if(-not $pipe)
{
    Write-Host -ForegroundColor $sColor $strIN
    Write-Host -ForegroundColor $uColor $strLine
}
Else
{
    $strIn
    $strLine
}
} #end funkcja New-Underline

Function Get-ScriptHelp
{
    [cmdletbinding()]
    Param ($scriptPath,$filePath)
    Get-ChildItem -Path $scriptPath -filter *.ps1 -Recurse |
    ForEach-Object {
        If(-not($_.psIsContainer))
        {
            New-Underline "$_" -pipe | Out-File -FilePath $filepath -Append
            Write-Verbose "Pobieranie pomocy dla $_.fullName"
            Get-Help $_.fullname -detailed | Out-File -FilePath $filepath -Append
        } #end if
    } #end foreach-object
} #end funkcja get-Scripthelp

# *** punkt początkowy skryptu ***
$ErrorActionPreference = "continue"
$filepath = "C:\fso\BestPracticeScripts.txt"
$Scriptpath = "C:\ScriptFolder"
if(Test-Path $filepath) {Remove-Item $filepath}
Get-ScriptHelp -filepath $filepath -scriptpath $scriptpath -Verbose
Notepad $filepath

```


Skrypt *Get-ScriptHelp.ps1* analizuje folder zawierający skrypty. Znajdujący się w nim filtr ogranicza wyszukiwanie do plików z rozszerzeniem *ps1*. Jeśli podczas działania skryptu wystąpi błąd, to domyślnie zostanie on zignorowany i skrypt będzie kontynuował działanie. W przypadku, gdy dokumentacja ma krytyczne znaczenie, można zmienić wartość zmiennej `$ErrorActionPreference` na `stop`. Dla uproszczenia istniejący plik dziennika jest automatycznie kasowany, dzięki czemu skrypt można uruchomić wiele razy bez obawy, że nowe informacje zostaną dodane na końcu.

Przedstawione podejście bazuje na założeniu, że pomoc komentarzowa jest jednym z pierwszych elementów pliku skryptu. Jeżeli przed pomocą komentarzową znajduje się blok komentarza, np. taki jaki można znaleźć w większości skryptów dołączonych do tej książki, to metoda ta nie zadziała. Jeżeli zmiana metody dokumentacyjnej nie wchodzi w grę, może być konieczne zmienienie wszystkich tradycyjnych nagłówkowych bloków komentarzy w pomoc komentarzową. Tradycyjny nagłówek komentarzowy wygląda tak:

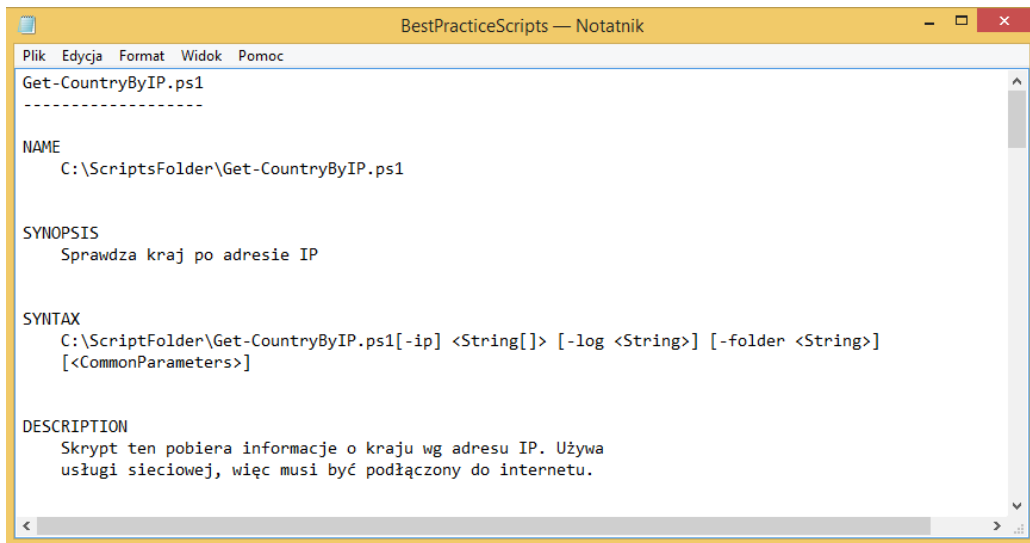
```
# -----
# Skrypt: Untitled1.ps1
# Autor: ed wilson, msft
# Data: 8.9.2013 13:47:33
# Słowa kluczowe: dokumentacja
# komentarze: Get-Help
# Windows PowerShell 4.0 Best Practices, Microsoft Press, 2013
# Rozdział 14
# -----
```

Przejsie na implementację pomocy na bazie komentarzy w nagłówkach skryptów wymaga wprowadzenia drobnej modyfikacji — zamiast licznych znaków `#` należy użyć symboli `<#` i `#>`. Wystarczy użyć paru standardowych znaczników, aby stworzyć dokumentację, która będzie łatwo dostępna za pomocą polecenia `Get-Help`. Poniżej znajduje się taki komentarzowy blok nagłówkowy:

```
<#
.Synopsis
To jest blok komentarzowy
.Description
To ilustruje użycie pomocy komentarzowej jako nagłówka skryptu
.Notes
NAZWA: Comment-BasedHeader.ps1
AUTOR: ed wilson, msft
DATA EDYCJI: 8.9.2013 13:49:40
SŁOWA KLUCZOWE: dokumentacja, pomoc
Książka: Windows PowerShell 4.0 Best Practices, Microsoft Press, 2013
Rozdział: 14
.Link
Http://www.ScriptingGuys.com
#Requires -Version 4.0
#>
```

Gdy uruchomi się skrypt *Get-ScriptHelp.ps1*, wyświetli on raport pokazany na rysunku 14.1.

Dokumentowanie skryptów ma znaczenie nie tylko ze względu na zapewnianie zgodności. W dokumentacji można też sprawdzić, do czego służy dany skrypt, zanim się go uruchomi. Jest to podstawowa czynność, zwłaszcza w przypadku skryptów pobieranych z takich miejsc jak repozytorium skryptów Scripting Guys. Więcej informacji na temat bezpieczeństwa pod tym względem można znaleźć w ramce zawierającej wypowiedź MVP Windows PowerShell Jeffery'ego Hicksa.



RYSUNEK 14.1. Dzięki zastosowaniu komentarzowych bloków nagłówkowych generowanie dokumentacji jest bardzo łatwe

Zapiski praktyka

Jeffery Hicks, MVP Windows PowerShell

Autor książki PowerShell in Depth: An administrator's guide

Spółeczność skupiona wokół konsoli Windows PowerShell cały czas szybko się rozwija i tworzy coraz więcej skryptów, które można pobierać z takich serwisów jak PosCode.org, TechNet i wielu blogów. Ale nie wszystkie skrypty są takiej samej wysokiej jakości. Jak sprawdzić, co dany skrypt robi? Czy można go bezpiecznie uruchomić? A może napisałeś skrypt do użytku przez kogoś innego? Czy masz pewność, że ten ktoś będzie się nim posługiwać w prawidłowy sposób? Oto kilka wątpliwości, jakie może mieć użytkownik skryptu:

- Co robi skrypt w przypadku, gdy nie przekaże się jakiegoś parametru?
- Jak skrypt działa w różnych systemach operacyjnych?
- Jak skrypt obsługuje różne wartości na wejściu?
- Jak skrypt działa dla różnych kombinacji parametrów?

Niektóre z tych kwestii można rozwiązać podczas prac nad skryptem. Na początku skryptu lub funkcji dodaj poniższy wiersz kodu:

```
Set-StrictMode -version latest
```

W ten sposób uniemożliwisz skryptowi „łamanie reguł”. Na przykład jeśli użyjesz niezdefiniowanej zmiennej, konsola poinformuje Cię o tym. Jest to doskonały sposób na wyeliminowanie literówek. Ponadto zawsze zalecam definiowanie domyślnych wartości dla

parametrów oraz rzutowanie wartości parametrów na odpowiednie typy danych. Dzięki temu konsola Windows PowerShell poinformuje nas, jeśli użytkownik przekaże w parametrze obiekt niewłaściwego typu. Ponadto warto korzystać z atrybutów weryfikacyjnych, które jeszcze bardziej uszczelniają system zabezpieczeń typów. Poniżej znajduje się przykład, który powinien działać już w konsoli Windows PowerShell 3.0.

```
Param (
[Parameter(Position=0,ValueFromPipeline,
    HelpMessage="Jaka jest nazwa komputera do skopiowania?")]
[ValidateScript({ Test-Connection $_ -quiet -count 1})]
[ValidatePattern("^CHI-\w{2}\d{2}$")]
[string]$Computername=$env:computername,
)
```

Administrator musi wpisać nazwę komputera zaczynającą się od znaków CHI, po których znajdują się łącznik, dwa znaki i dwie cyfry, np. CHI-DC02. Ponadto komputer musi odpowiedzieć na pojedyncze polecenie ping. Jeśli którykolwiek z tych warunków nie zostanie spełniony, Windows PowerShell zgłosi wyjątek i polecenie nie zostanie wykonane, co jest lepsze niż przerwanie pracy w pół drogi.

Oczywiście **wszystko** trzeba przetestować w środowisku testowym. Skrypt powinien przejść przez taki sam proces jak każda inna wewnętrzna aplikacja. Zawsze staram się wymyślić najgłupszy możliwy sposób użycia skryptu i przetestować, co się wtedy stanie. Robię to, bo chcę, aby moje skrypty były jak najsolidniejsze. Założę się, że masz podobne priorytety.

A co zrobić ze skryptem pobranym z internetu? Jak sprawdzić, czy jest bezpieczny? Wprawdzie można go po prostu uruchomić w środowisku testowym, np. w maszynie wirtualnej, ale najpierw lepiej przejrzeć jego kod źródłowy. Wiem, że nie każdy jest specjalistą od Windows PowerShell i nie musi wszystkiego rozumieć. Dlatego poniżej daję parę wskazówek.

W poprzednim wydaniu tej książki opisałem skrypt Windows PowerShell służący do „profilowania” skryptów. Za jego pomocą można się dowiedzieć, jakie polecenia dany skrypt wywołuje i czy któreś z nich mogą być niebezpieczne. Ale szczerze mówiąc, skrypt ten nie był najlepiej napisany. Na szczęście w Windows PowerShell 3.0 wprowadzono nowy sposób przeprowadzania analizy składniowej skryptów, polegający na użyciu abstrakcyjnych drzew składni (ang. *abstract syntax tree* — AST). Nie musisz dokładnie wiedzieć, co to jest. Wystarczy, że wśród plików z przykładowymi skryptami znajdziesz mój skrypt *Get-ASTScriptProfile.ps1*.

Skrypt ten pobiera nazwę skryptu (*.ps1* lub *.psm1*) do zbadania. Przy użyciu AST przygotowuje raport tekstowy zawierający informacje o wymaganiach skryptu, jego parametrach, użytych w nim poleceniach oraz nazwach typów. W raporcie są uwzględnione nazwy wszystkich użytych poleceń, wliczając nieznane i takie, które według mnie mogą być niebezpieczne, czyli np. o nazwach zaczynających się od słów Remove i Stop. Jako że niektórzy mogą wywoływać metody bezpośrednio z klas .NET, postanowiłem też sprawdzić wszystkie nazwy typów. Większość z nich będzie pewnie dotyczyć parametrów, ale przynajmniej będzie wiadomo, czego szukać. Cały raport ma postać tematu pomocy. Zostaje zapisany w folderze *Dokumenty* i można go wyświetlić za pomocą polecenia `Get-Help` z parametrem `-ShowWindow`.

W raporcie nie są uwzględniane parametry z zagnieżdżonych funkcji, ale przynajmniej pokazane są użyte w nich polecenia. Do wykrywania poleceń mogących wczytywać moduły użyto polecenia `Get-Command`. Nie powinno to być problemem, ale i tak profilowanie lepiej przeprowadzać w testowym środowisku wirtualnym. Każde wyszczególnione w raporcie nieznane polecenie pochodzi z jakiegoś modułu, którego nie udało się załadować, albo jest zdefiniowane wewnętrznie. Gdy wiadomo, czego szukać, można otworzyć skrypt w dowolnym edytorze i przejrzeć kod źródłowy.

Bezpieczna obsługa konsoli Windows PowerShell to nawyk, który trzeba sobie wyrobić nie tylko podczas używania własnych skryptów i narzędzi, ale także gdy używa się skryptów napisanych przez innych.

Pobieranie dokumentacji z komentarzy

Jeśli skrypt nie zawiera typowej pomocy komentarzowej, to i tak można z niego wydobyć dobrą dokumentację, pobierając treść zwykłych komentarzy. Skrypt *GetCommentsFromScript.ps1* przyjmuje ścieżkę do skryptu i zwraca wszystkie znajdujące się w nim komentarze zaczynające linijki kodu. Czyli skrypt ten nie pobiera komentarzy znajdujących się wewnątrz wierszy kodu, a jedynie te komentarze, które znajdują się na początku wierszy kodu. Uwzględnia więc tradycyjne bloki komentarzy opisane w poprzednim podrozdziale, jak również komentarze sekcyjne. Poniżej znajduje się kompletny kod źródłowy skryptu *GetCommentsFromScript.ps1*:

GetCommentsFromScript.ps1

```
Function Get-FileName
{
    Param ($Script)
    $OutPutPath = [io.path]::GetTempPath()
    Join-Path -path $OutPutPath -child "$(Split-Path $script -leaf).txt"
} #end Get-FileName

Function Remove-OutPutFile($OutPutFile)
{
    if(Test-Path -path $OutPutFile)
    {
        $Response = Read-Host -Prompt "Plik $OutPutFile już istnieje. Czy chcesz go usunąć <t / n>?"
        if($Response -eq "t")
        { Remove-Item $OutPutFile | Out-Null }
        ELSE
        {
            if(Test-Path -path "$OutPutFile.old") { Remove-Item -Path "$OutPutFile.old" }
            Rename-Item -path $OutPutFile -newname "$(Split-Path $OutPutFile -leaf).old" -Force
        }
    }
} #end Remove-OutPutFile
```

```
Function Get-Comments
{
    Param ($Script, $OutPutFile)
    Get-Content -path $Script |
    Foreach-Object {
        If($_ -match '^\#')
        { $_ |
            Out-File -FilePath $OutPutFile -append }
        } #end Foreach
    } #end Get-Comments

Function Get-OutPutFile($OutPutFile)
{
    Notepad $OutPutFile
} #end Get-OutPutFile

# *** punkt początkowy skryptu ***

$Script = 'C:\scriptfolder\Get-ModifiedFilesUsePipeline.ps1'
$OutPutFile = Get-FileName($Script)
Remove-OutPutFile($OutPutFile)
Get-Comments -script $Script -outfile $OutPutFile
Get-OutPutFile($OutPutFile)
```

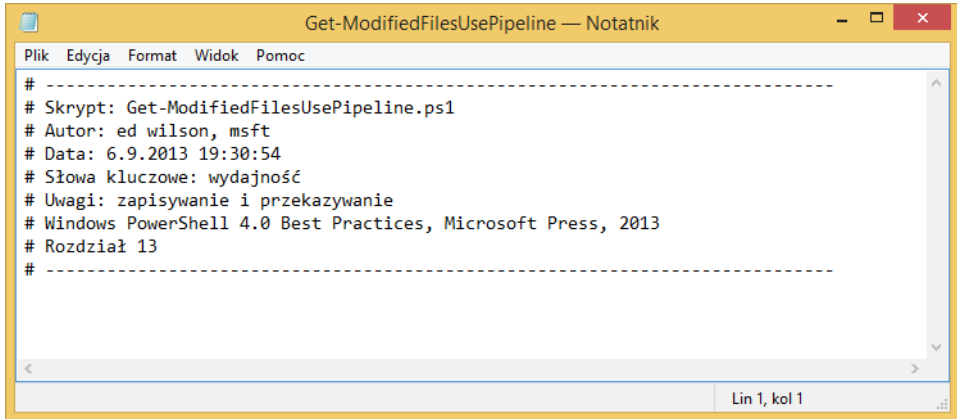
Powyższy skrypt tworzy dla określonego skryptu plik dokumentacyjny o nazwie takiej samej jak nazwa badanego skryptu z dołączonym rozszerzeniem *txt*. Ponadto plik ten tworzy w tymczasowym katalogu roboczym użytkownika, który go uruchomił. Ta ścieżka zawsze jest dostępna w systemie. Opisywane czynności wykonuje przedstawiona poniżej funkcja `Get-FileName`:

```
Function Get-FileName
{
    Param ($Script)
    $OutPutPath = [io.path]::GetTempPath()
    Join-Path -path $OutPutPath -child "$(Split-Path $Script -leaf).txt"
} #end Get-FileName
```

Funkcja `Get-Comments` wczytuje skrypt i za pomocą prostego wyrażenia regularnego wyciąga z niego po jednej linijce kodu. Poniżej znajduje się kod źródłowy tej funkcji:

```
Function Get-Comments
{
    Param ($Script, $OutPutFile)
    Get-Content -path $Script |
    Foreach-Object {
        If($_ -match '^\#')
        { $_ |
            Out-File -FilePath $OutPutFile -append }
        } #end Foreach
    } #end Get-Comments
```

Po zakończeniu działania skrypt zwraca plik podobny do pokazanego na rysunku 14.2.



RYSUNEK 14.2. Efekt wyciągnięcia komentarzy ze skryptu za pomocą skryptu GetCommentsFromScript.ps1

Wiedza tajemna

Chris Bellee, Premier Field Engineer
Microsoft Corporation, Australia

Polecenia cmdlet to w konsoli Windows PowerShell najważniejsze narzędzie administracyjne. Są one skompilowanymi programami zawierającymi klasy .NET. W każdej kolejnej wersji konsoli dodawano nowe polecenia, dzięki którym narzędzie stawało się coraz bardziej wszechstronne i przydatne. Na przykład w Windows PowerShell 1.0 było tylko 126 standardowych poleceń, a w Windows PowerShell 4.0 (w systemie Windows 8) jest ich już 510.

Jednak mimo tak dużej liczby nowych poleceń za ich pośrednictwem mamy dostęp tylko do niewielkiej części zasobów platformy .NET. Dlatego niezmiernie przydatne jest polecenie `New-Object` służące do tworzenia egzemplarzy klas .NET (zwanymi **obiektami**), których później można używać w konsoli.

Powiedzmy na przykład, że chcę sprawdzić, czy jakiś zdalny komputer nasłuchuje na porcie 80. Jakiego polecenia powinienem użyć? Jeśli używam Windows PowerShell 4.0, to jest jedno takie polecenie, ale mogę też podłączyć się bezpośrednio do platformy .NET i utworzyć specjalny obiekt. Do utworzenia takiego obiektu służy polecenie `New-Object`, w którego parametrze `-TypeName` podaje się pełną ścieżkę do wybranej klasy. Składnia tego polecenia jest bardzo prosta i tak naprawdę przy jego używaniu największym problemem jest wybór odpowiedniej klasy.

```
$tcpClient = New-Object -TypeName System.Net.Sockets.TcpClient
```

Z utworzonego obiektu można wydobyć informacje za pomocą dobrze znanego polecenia `Get-Member`.

```
$tcpClient | Get-Member
```

W obiekcie tym znajduje się kilka składowych (własności i metod). Można podejrzewać, że metoda `Connect()` służy do nawiązywania połączenia ze zdalnym komputerem, ale jak jej użyć? Można zastosować sztuczkę polegającą na wpisaniu samej nazwy metody, bez nawiasu. To spowoduje, że konsola wyświetli różne wersje tej metody z informacjami o liczbie i typach argumentów.

```
$tcpClient.Connect
OverloadDefinitions
-----
void Connect(string hostname, int port)
void Connect(ipaddress address, int port)
void Connect(System.Net.IPEndPoint remoteEP)
void Connect(ipaddress[] ipAddresses, int port)
```

Jak widać, metoda `Connect()` dysponuje czterema zestawami argumentów wejściowych. W tym przykładzie użyjemy pierwszej wersji, która przyjmuje łańcuch reprezentujący nazwę komputera zdalnego oraz liczbę całkowitą reprezentującą numer zdalnego portu, z którym ma zostać nawiązane połączenie.

```
$tcpClient.Connect("Server-01",80)
```

Aby dowiedzieć się, czy połączenie zostało nawiązane, można sprawdzić wartość logicznej własności `Connected` obiektu, którą powinno być `true` lub `false`.

```
$tcpClient.Connected
```

Wystarczy powyższy kod wstawić do funkcji albo pętli i już ma się prostą metodę do sprawdzania dostępności portów i diagnozowania problemów z łącznością.

Sposób użycia parsera AST

Bardziej zaawansowana technika dokumentowania skryptów polega na użyciu klasy `PSParser` z przestrzeni nazw `.NET System.Management.Automation`. Za pomocą statycznej metody `Tokenize` z tej klasy można korzystać z tokenizatora. Tokenizator to program rozbijający kod skryptu Windows PowerShell na części zwane **tokenami**. Za pomocą tokenizatora można znaleźć polecenia i zmienne w skryptach. Największą zaletą tego narzędzia jest to, że nie ma znaczenia, w którym miejscu skryptu znajduje się polecenie. Ponadto jest ono łatwiejsze w obsłudze od skomplikowanych wyrażeń regularnych.

Przykład analizy składniowej skryptu przedstawiono na bazie skryptu *ParseScriptCommands.ps1*.

ParseScriptCommands.ps1

```
$errors = $null
$logpath = "C:\fso\commandlog.txt"
$path = 'C:\ScriptFolder'
Get-ChildItem -Path $path -Include *.ps1 -Recurse |
ForEach-Object {
    $script = $_.fullname
    $scriptText = get-content -Path $script
    [system.management.automation.psparser]::Tokenize($scriptText, [ref]$errors) |
    Foreach-object -Begin {
        "Przetwarzanie skryptu $script" | Out-File -FilePath $logPath -Append } '
```

```
-process { if($_.type -eq "command")
  { "t $($_.content)" | Out-File -FilePath $logpath -Append } }
}
notepad $logpath
```

Na początku inicjowane są trzy zmienne. Pierwsza służy do przechowywania błędów wygenerowanych przez tokenizator. Druga określa ścieżkę do pliku dziennika, a trzecia — katalog zawierający skrypty Windows PowerShell, które trzeba przeanalizować. Definicje te widać poniżej:

```
$errors = $null
$logpath = "C:\logs\commandlog.txt"
$path = "C:\data\PSExtras"
```

Następnie wywołujemy cztery standardowe polecenia Windows PowerShell. Polecenie `Get-ChildItem` pobiera tylko skrypty Windows PowerShell (z rozszerzeniem *ps1*) z określonego wcześniej katalogu ze skryptami. Przy pobieraniu plików z folderu potrzebny jest parametr `-recurse`. Otrzymane obiekty `fileinfo` zostają przekazane do polecenia `ForEach-Object`, w którym następuje zapisanie pełnej ścieżki do każdego skryptu w zmiennej `$script`. Następnie polecenie `Get-Content` wczytuje każdy skrypt Windows PowerShell i zapisuje jego treść w zmiennej `$scriptText`. Poniżej znajduje się opisywana część skryptu:

```
Get-ChildItem -Path $path -Include *.ps1 -Recurse |
ForEach-Object {
  $script = $_.fullname
  $scriptText = get-content -Path $script
```

Klasa `.NET PSParser` z przestrzeni nazw `System.Management.Automation` zawiera metodę statyczną `Tokenize`. Pierwszy parametr jest zmienną zawierającą treść skryptu Windows PowerShell, a drugi jest zmienną referencyjną do przechowywania błędów. W wywołaniu metody `Tokenize` konieczne jest podanie drugiego parametru. Następnie tokeny zostają przekazane do kolejnej części skryptu. Poniżej znajduje się opisywane polecenie:

```
[system.management.automation.psparser]::Tokenize($scriptText, [ref]$errors)
```

Do przetwarzania tokenów zostało użyte polecenie `ForEach-Object`. Jego blok zaczyna się od zapisania w dzienniku pełnej ścieżki do skryptu. Potem sprawdzane jest, czy własność `type` obiektu tokenu jest poleceniem — jeśli tak, to polecenie to również zostaje zapisane w dzienniku. Po zakończeniu przetwarzania wszystkich skryptów w folderze następuje wyświetlenie raportu. Poniżej znajduje się opisywana część skryptu:

```
ForEach-object -Begin {
  "Przetwarzanie skryptu $script" | Out-File -FilePath $logPath -Append } '
-process { if($_.type -eq "command")
  { "t $($_.content)" | Out-File -FilePath $logpath -Append } }
}
notepad $logpath
```

Wynik działania skryptu *ParseScriptCommands.ps1* powinien wyglądać mniej więcej jak na rysunku 14.3.

Jeśli ktoś umie posługiwać się tokenizatorem Windows PowerShell, to może napisać narzędzie do analizy skryptów z uwzględnieniem różnych aspektów.


```

commandlog — Notatnik
Plik  Edycja  Format  Widok  Pomoc
Przetwarzanie skryptu C:\ScriptFolder\Get-CommandLineOptions.ps1
Get-Disk
Get-Processor
Get-Memory
Get-Network
Get-Video
Get-Disk
Get-Processor
Get-Memory
Get-Network
Get-Video
Przetwarzanie skryptu C:\ScriptFolder\Get-ComputerWmiInformation.ps1
Get-WmiObject
Get-WmiObject
Get-WmiObject
Get-WmiObject
Get-WmiObject
Get-Disk
Get-Processor
Get-Memory

```

RYСУNEK 14.3. Raport utworzony przez skrypt ParseScriptCommands.ps1

Dodatkowe źródła informacji

- Centrum skryptów portalu TechNet pod adresem <http://www.microsoft.com/technet/scriptcenter> zawiera wiele przykładów skryptów Windows PowerShell.
- Wszystkie skrypty opisane w tym rozdziale są dostępne do pobrania w repozytorium centrum skryptów portalu TechNet pod adresem <http://gallery.technet.microsoft.com/scriptcenter/PowerShell-40-Best-d9e16039>.
- Dokumentacja klasy PSParser znajduje się w portalu MSDN pod adresem [http://msdn.microsoft.com/en-us/library/windows/desktop/system.management.automation.psparser\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/desktop/system.management.automation.psparser(v=vs.85).aspx).
- Dokumentacja klasy CommandAST znajduje się w portalu MSDN pod adresem [http://msdn.microsoft.com/en-us/library/windows/desktop/system.management.automation.language.commandast\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/desktop/system.management.automation.language.commandast(v=vs.85).aspx).
- Więcej informacji na temat atrybutów weryfikacyjnych można znaleźć w pomocy konsoli Windows PowerShell i module pomocy napisanym przez Jeffery'ego Hicksa i dostępnym na stronie <http://jdhitsolutions.com/blog/2012/05/introducing-the-scriptinghelp-powershell-module/>.

CZĘŚĆ IV

Wdrażanie skryptu

Rozdział 15. Ustawianie zasad wykonywania skryptów

Rozdział 16. Uruchamianie skryptów

Rozdział 17. Kontrola wersji skryptów

Rozdział 18. Rejestrowanie wyników

Rozdział 19. Rozwiązywanie problemów ze skryptami

Rozdział 20. Praca w środowisku Windows PowerShell ISE

Rozdział 21. Narzędzia do pracy zdalnej i zadania Windows PowerShell

Rozdział 22. Przepływy pracy w Windows PowerShell

Rozdział 23. Usługa konfiguracji żądanego stanu programu PowerShell

Rozdział 15

Ustawianie zasad wykonywania skryptów

- Wybór zasady wykonywania dla skryptu
- Wdrażanie zasady wykonywania skryptu
- Podpisywanie kodu
- Dodatkowe źródła informacji

Przed uruchomieniem skryptu Windows PowerShell należy wybrać zasadę jego wykonywania. Potem należy jeszcze wybrać sposób wdrożenia tej zasady na komputerach w sieci. Jako że w przypadku niektórych zasad wykonywania skryptów może wystąpić problem z podpisem kodu, w rozdziale tym opisuję dodatkowo techniki podpisywania skryptów. Potem przechodzę do opisu specjalnych typów skryptów i różnych metod kontroli wersji.

Wybór zasady wykonywania dla skryptu

Wybór poziomu wsparcia dla skryptu ze strony środowiska jest pierwszą zasadniczą czynnością w procesie wdrażania skryptu do środowiska produkcyjnego. Na początku należy zdecydować, czy na wybranym komputerze lub serwerze w ogóle zezwolić na uruchamianie skryptów Windows PowerShell. Domyślnie jest to zabronione. Ustawienie to stanowi dodatkową warstwę zabezpieczeń nie tylko przed szkodliwymi programami, ale również przed bezmyślnością użytkowników i administratorami sieci nieprzeszkolonymi z zakresu obsługi konsoli Windows PowerShell. Aby umożliwić wykonywanie skryptów, należy włączyć ich obsługę. Daje to kilka korzyści, na przykład:

- Można wykonywać skrypty Windows PowerShell jako skrypty logowania (wymagany jest kontroler domeny Windows Server 2008 R2 lub nowszy).
- Można zdalnie administrować serwerami i zwykłymi komputerami przy użyciu technik jeden do wielu i wiele do jednego.
- Można szybko zastosować jednakowe zmiany konfiguracji na zwykłych komputerach i serwerach.
- Można zapisać szereg poleceń do użytku kiedy indziej.
- Można dostosować serię poleceń do własnych potrzeb i zoptymalizować ich wydajność.

- Można używać profilu Windows PowerShell.
- Można używać modułów.

Przeznaczenie zasad wykonywania skryptów

Zasady wykonywania skryptów Windows PowerShell nie są elementem systemu zabezpieczeń, a jedynie funkcją pomocniczą. Mają za zadanie uświadomić użytkownikom kwestie bezpieczeństwa związane z uruchamianiem skryptów Windows PowerShell. Nawet jeśli ktoś podpisze wszystkie swoje skrypty, nie może mieć pewności, że któryś z nich w jakiejś sieci nie narobi szkód. Jedyne, co można zagwarantować, to że nikt obcy w naszym skrypcie nie grzebał od czasu jego podpisania. Certyfikat do podpisywania kodu może mieć każdy, nawet ludzie piszący szkodliwe programy. Dlatego też certyfikatu takiego nie należy traktować jak rozwiązania wszystkich problemów z zabezpieczeniami. I tak trzeba uczulić pracowników z działu IT na kwestie związane z bezpieczeństwem środowiska informatycznego. Poza tym nawet jeśli uruchamianie skryptów jest wyłączone przez zasadę wykonywania skryptów, to i tak istnieje możliwość obejścia tego ograniczenia za pomocą przełącznika `bypass`.

UWAGA

W systemie Windows Server 2012 R2 zasada wykonywania skryptów jest automatycznie ustawiana na `RemoteSigned`. Natomiast w systemie Windows 8.1 jest ona ustawiona na `Restricted`.

Różne zasady wykonywania skryptów

W konsoli Windows PowerShell istnieje kilka ustawień dotyczących sposobu obsługi skryptów. Każde z nich daje różne uprawnienia dotyczące wykonywania skryptów lokalnie i na komputerach zdalnych. Ponadto niektóre mogą mieć też wpływ na sposób pisania i testowania skryptów. Istnieje pięć takich ustawień poziomu obsługi skryptów; ich opis zamieszczono w tabeli 15.1.

TABELA 15.1. Ustawienia zasad wykonywania skryptów

| Ustawienie | Opis |
|------------|---|
| Restricted | <ul style="list-style-type: none">◆ Domyślna zasada wykonywania skryptów◆ Pozwala na interaktywne wykonywanie poleceń w konsoli Windows PowerShell◆ Pozwala na przekazywanie wyników poleceń przez potok◆ Pozwala na tworzenie funkcji w konsoli Windows PowerShell◆ Pozwala na używanie bloków skryptów w poleceniach◆ Blokuje wykonywanie zawartości plików z rozszerzeniem <code>ps1</code>, wliczając wszystkie sześć profili Windows PowerShell◆ Blokuje moduły (pliki z rozszerzeniem <code>psm1</code>)◆ Blokuje pliki konfiguracyjne Windows PowerShell (pliki <code>.ps1</code> w formacie XML) |

TABELA 15.1. Ustawienia zasad wykonywania skryptów — ciąg dalszy

| Ustawienie | Opis |
|--------------|---|
| AllSigned | <ul style="list-style-type: none"> ◆ Pozwala na uruchamianie skryptów, profili, modułów i plików konfiguracyjnych mających podpis zaufanego wydawcy ◆ Wymaga, aby podpisane były wszystkie skrypty, również skrypty napisane na lokalnym komputerze ◆ Przed uruchomieniem skryptów podpisanych przez niezaufanego wydawcę prosi o potwierdzenie ◆ Przed pierwszym uruchomieniem skryptu podpisanego przez zaufanego wydawcę prosi o potwierdzenie |
| RemoteSigned | <ul style="list-style-type: none"> ◆ Pozwala na uruchamianie lokalnych skryptów, profili, modułów i plików konfiguracyjnych bez podpisu ◆ Pozwala na uruchamianie tylko podpisanych przez zaufanego wydawcę skryptów ze strefy internetowej ◆ Przed wykonaniem skryptów ze strefy internetowej podpisanych przez zaufanego wydawcę prosi o potwierdzenie |
| Unrestricted | <ul style="list-style-type: none"> ◆ Pozwala na uruchamianie wszystkich niepodpisanych skryptów, profili, modułów i plików konfiguracyjnych ◆ Przed wykonaniem skryptów ze strefy internetowej prosi o potwierdzenie |
| Bypass | <ul style="list-style-type: none"> ◆ Nic nie jest blokowane i nie są wyświetlane żadne monity o potwierdzenie |

Co to jest strefa internetowa

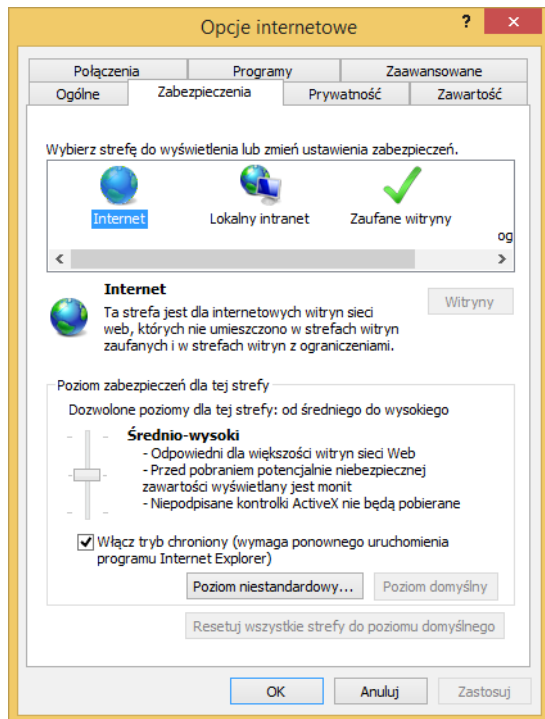
Zasady wykonywania skryptów Windows PowerShell bazują na ustawieniach strefy internetowej przeglądarki Windows Internet Explorer. Korzystają z nich też niektóre inne aplikacje, jak np. Microsoft Office Outlook. Na rysunku 15.1 widać ustawienia stref internetowych przeglądarki Internet Explorer.

Internet Explorer dodaje znacznik do alternatywnego strumienia plikowego pliku skryptu. Jeśli skrypt pochodzi z programu Office Outlook (przy założeniu, że nie usunie go program antywirusowy), to również dodawany jest znacznik do jego alternatywnego strumienia danych. Każda aplikacja może respektować definicję stref internetowych przeglądarki Internet Explorer i dodawać znacznik lub go odczytywać z odbieranych plików. Aby zobaczyć alternatywny strumień danych pliku, można użyć narzędzia Windows SysInternals *Streams.exe*, które może zarówno odczytywać, jak i usuwać znacznik strefy internetowej. Aby wyszukać pliki z alternatywnymi strumieniami danych, można użyć przełącznika `-s` oraz podać ścieżkę do folderu. Spowoduje to zwrócenie wszystkich plików z określonego folderu zawierających alternatywne strumienie danych.

```
PS C:\data\streams> .\streams.exe -s c:\fso
```

```
Streams v1.5 - Enumerate alternate NTFS data streams
Copyright (C) 1999-2003 Mark Russinovich
Sysinternals - www.sysinternals.com
```

```
c:\fso\InternetScript.ps1:
:Zone.Identifier:$DATA          26
```



RYSUNEK 15.1. Konsola Windows PowerShell do określania, czy skrypt pochodzi z internetu, używa ustawień stref zabezpieczeń przeglądarki Internet Explorer

Znacznik Zone.Identifier oznacza, że plik został pobrany z internetu, więc próba uruchomienia skryptu *InternetScript.ps1* zostanie zablokowana. Ale za pomocą przełącznika `-d` narzędzia *Streams.exe* znacznik ten można usunąć. Należy w tym celu podać ścieżkę do skryptu.

```
PS C:\data\streams> .\streams.exe -d C:\fso\InternetScript.ps1
```

```
Streams v1.5 - Enumerate alternate NTFS data streams
Copyright (C) 1999-2003 Mark Russinovich
Sysinternals - www.sysinternals.com
```

```
C:\fso\InternetScript.ps1:
:Zone.Identifier:$DATA 26
PS C:\data\streams> .\streams.exe -s c:\fso
```

```
Streams v1.5 - Enumerate alternate NTFS data streams
Copyright (C) 1999-2003 Mark Russinovich
Sysinternals - www.sysinternals.com
```

Po usunięciu alternatywnego strumienia danych z pliku plik ten będzie traktowany tak, jakby pochodził z komputera lokalnego, i przestanie być blokowany. Technika tą można usunąć informację o internetowym pochodzeniu ze skompilowanych plików pomocy (CHM) oraz skryptów i innych plików, które są standardowo blokowane z powodu pobrania ich ze strefy internetowej.

Jeśli skrypt zostanie skopiowany ze strony internetowej i zapisany w pliku z rozszerzeniem *ps1* (np. w Windows PowerShell ISE albo Notatniku), to zostanie uznany za pochodzący z zaufanej strefy lokalnej, ponieważ został utworzony lokalnie. Skrypt może zostać oznaczony jako zdalny

wyłącznie wtedy, gdy zostanie pobrany z internetu w całości — a nie wtedy, gdy ktoś skopiuje jego zawartość ze strony internetowej i wklei ją w pliku lokalnym. Może to mieć wpływ na inne pliki niż Windows PowerShell. Gdy skrypty Windows PowerShell (skompresowane przez pewne narzędzia do kompresji plików) są pobierane ze strefy internetowej jako pliki skompresowane, to pozostają oznaczone jako internetowe także po dekompresji. Zawdzięczamy to temu, że określone narzędzia kompresujące uznają ustawienia stref internetowych przeglądarki Internet Explorer. Gdyby skrypty Windows PowerShell spakowano za pomocą wykonywalnych instalatorów, to nie zostałyby oznaczone jako zdalne, ponieważ zainstalowane skrypty uważa się za pliki lokalne.

Jeśli zasada wykonywania skryptów Windows PowerShell zostanie ustawiona na RemoteSigned, skrypty oznaczone jako pochodzące z internetu będą musiały mieć podpis, aby można je było uruchomić. Przeglądarka Internet Explorer bardzo agresywnie określa granice strefy internetowej. Udziały UNC (ang. *Universal Naming Convention*) są domyślnie zaliczane do strefy internetowej, co oznacza, że skrypty pobrane z wewnętrznego udziału da się uruchomić dopiero po ich podpisaniu. Jako że wiele firm przechowuje magazyny skryptów w wewnętrznych udziałach plikowych, jest to poważny problem. Rozwiązaniem jest dodanie udziału ze skryptami do strefy zaufanych stron przeglądarki Internet Explorer. Można to zrobić bezpośrednio w przeglądarce, dodając lokalizację w rejestrze, albo przy użyciu zasady grupowej. W środowiskach korporacyjnych oczywiście najlepszym rozwiązaniem jest użycie zasady grupowej.

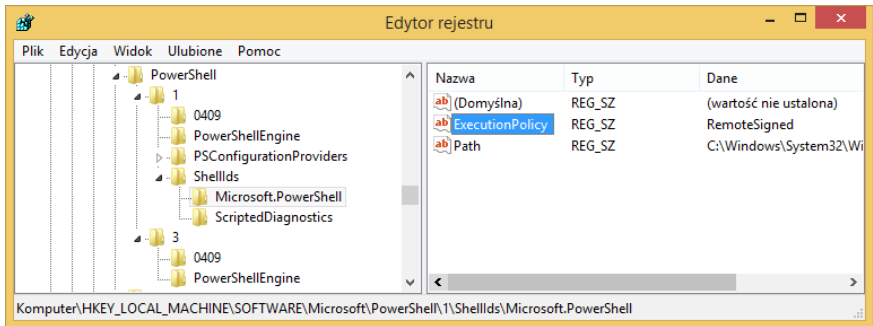
Wdrażanie zasady wykonywania skryptu

Jest kilka możliwości wdrożenia zasady wykonywania skryptu. Można na przykład zmodyfikować ją w rejestrze, ale nie jest to najlepsze rozwiązanie, ponieważ istnieje ryzyko popełnienia błędu i uszkodzenia tej bazy danych. Poza tym w wielu przypadkach jest to po prostu zbyt czasochłonne. Jeśli trzeba zmienić zasadę wykonywania skryptów na komputerze lokalnym, najlepiej to zrobić za pomocą polecenia `Set-ExecutionPolicy`, które jest łatwe w obsłudze i daje gwarancję, że zmiana zostanie wprowadzona prawidłowo.

Jeśli komputerów jest więcej, najlepiej będzie użyć zasady grupowej. Zaletą tego rozwiązanie jest to, że operację można łatwo cofnąć oraz kontrolować z jednego centralnego miejsca. Jeśli ktoś nie używa zasady grupowej, może wprowadzić zmiany za pomocą skryptu logowania.

Modyfikowanie rejestru

Zasadę wykonywania skryptów można włączyć i wyłączyć przez wprowadzenie odpowiednich zmian w rejestrze. Technikę tę najlepiej stosować w środowiskach, w których brak zasady grupowej. Jeśli zasada wykonywania skryptów Windows PowerShell nie jest ustawiona na opcję umożliwiającą wykonywanie skryptów Windows PowerShell, to można pomyśleć, że jedynym sposobem na wprowadzenie zmian w rejestrze jest użycie skryptu VBScript lub pliku wsadowego (.bat). Ale konsola Windows PowerShell zawiera przełącznik `bypass`, dzięki któremu można uruchamiać skrypty w trybie pominięcia procedur. Na rysunku 15.2 widać klucz rejestru `HKEY_LocalMachine\Software\Microsoft\PowerShell\1\ShellIDs\Microsoft.PowerShell\ExecutionPolicy`.



RYSUNEK 15.2. Zasada wykonywania skryptów Windows PowerShell w rejestrze systemowym

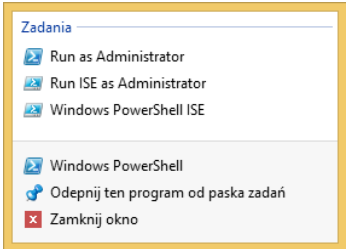
Jeśli ktoś tworzy plik instalatora Microsoft (MSI) instalujący standardowy profil na komputerach, to może też zmodyfikować rejestr za pomocą pakietu MSI. Ale to są wyjątkowe sytuacje i z reguły powinno się unikać bezpośredniego modyfikowania rejestru.

Użycie polecenia Set-ExecutionPolicy

Zasadę wykonywania skryptów Windows PowerShell można też ustawić za pomocą polecenia Set-ExecutionPolicy. Do jego wykonania w konsoli Windows PowerShell potrzebne są podwyższone uprawnienia, ponieważ polecenie to wprowadza zmiany w rejestrze. W istocie polecenie to robi dokładnie to samo, co robi się w przypadku ręcznego modyfikowania rejestru. Bezpośrednio po instalacji systemu Windows klucze rejestru dotyczące zasad wykonywania skryptów Windows PowerShell nie istnieją. Domyślną wartością zasady wykonywania skryptów jest Restricted, ale nie jest ona widoczna w rejestrze, dopóki nie zostanie zmieniona. W związku z tym modyfikacja rejestru wymaga sprawdzenia, czy istnieje odpowiedni klucz, i utworzenia go w razie potrzeby lub zmodyfikowania jego wartości. Sprawdzanie i modyfikowanie klucza rejestru jest dość kłopotliwe i pracochłonne. O wiele lepiej jest użyć polecenia Set-ExecutionPolicy, aby nie musieć się w to bawić.

Sposób użycia polecenia Set-ExecutionPolicy na komputerze lokalnym

Aby zmienić zasadę wykonywania skryptów Windows PowerShell na komputerze lokalnym, w systemie Windows Vista i nowszym należy uruchomić konsolę PowerShell jako administrator — rysunek 15.3.



RYSUNEK 15.3. Aby można było używać polecenia Set-ExecutionPolicy, konsola musi być uruchomiona z uprawnieniami administratora

Jeżeli użytkownik nie ma uprawnień administratora, próba wykonania polecenia `Set-ExecutionPolicy` zakończy się błędem.

```
PS C:\Users\edwils> Set-ExecutionPolicy -ExecutionPolicy remotesigned
```

Execution Policy Change

The execution policy helps protect you from scripts that you do not trust. Changing the execution policy might expose you to the security risks described in the `about_Execution_Policies` help topic. Do you want to change the execution policy?

[Y] Yes [N] No [S] Suspend [?] Help (default is "Y"): y

Set-ExecutionPolicy : Access to the registry key

'HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\PowerShell\1\ShellIds\Microsoft.PowerShell' is denied. At line:1 char:20

```
+ Set-ExecutionPolicy <<<< -ExecutionPolicy remotesigned
```

```
+ CategoryInfo : NotSpecified: (:) [Set-ExecutionPolicy], UnauthorizedAccessException
```

```
+ FullyQualifiedErrorId : System.UnauthorizedAccessException,
Microsoft.PowerShell.Commands.SetExecutionPolicyCommand
```

Jeśli polecenie zostanie wykonane bez przeszkód, to w konsoli nie pojawi się żadna informacja.

```
PS C:\> Set-ExecutionPolicy -ExecutionPolicy unrestricted
```

```
PS C:\>
```

Efekt zmian jest natychmiastowy. Ustawienia zasady wykonywania skryptów można wypróbować przez uruchomienie skryptu lub za pomocą polecenia `Get-ExecutionPolicy`. W skrypcie *Test-Script.ps1* zastosowano oba te rozwiązania.

Test-Script.ps1

```
"Ten skrypt testowy wyświetla zasadę wykonywania skryptów."
Get-ScriptExecutionPolicy
```

Użycie polecenia `Set-ExecutionPolicy` w skrypcie logowania

Jeśli w sieci nie ma zasady grupowej i nie uśmiecha Ci się pomysł edytowania rejestru, to masz jeszcze jedno wyjście. Polega ono na dodaniu odpowiedniego polecenia do skryptu logowania. Skrypt ten i tak jest wykonywany, więc stanowi gotową infrastrukturę do konfiguracji zasady wykonywania skryptów. Aby ustawić zasadę wykonywania skryptów na `RemoteSigned` w skrypcie logowania, należy użyć poniższego polecenia:

```
powershell -command &{Set-ExecutionPolicy remotesigned}
```

Jeśli skrypt logowania jest plikiem wsadowym, powyższe polecenie zadziała bezpośrednio. Ale jeśli będzie nim skrypt VBScript, trzeba będzie trochę więcej popracować. Aby za pomocą polecenia `Set-ExecutionPolicy` ustawić zasadę wykonywania skryptów Windows PowerShell na wszystkich stacjach roboczych, można użyć metody `run` obiektu `WshShell`. Przedstawiony poniżej skrypt *SetScriptExecutionPolicy.vbs* ustawia zasadę wykonywania skryptów na `RemoteSigned`, ale łatwo można to zmienić na dowolną inną wartość. Pamiętaj tylko, że skrypt ten musi zostać uruchomiony z uprawnieniami administratora, ponieważ polecenie `Set-ExecutionPolicy` modyfikuje rejestr.

SetScriptExecutionPolicy.vbs

```
Set WshShell = CreateObject("WScript.Shell")
WshShell.Run("powershell -Noninteractive -command &{Set-ExecutionPolicy remotesigned}")
```

Zapiski praktyka

Ustawianie zabezpieczeń Windows PowerShell

Richard Siddaway, MVP Microsoft PowerShell

Prezes brytyjskiej grupy użytkowników PowerShell

„Konsola Windows PowerShell nie działa. Nie da się uruchamiać skryptów”.

Nie wiem, ile już razy widziałem podobne zdania na rozmaitych forach internetowych. Kwestia ta jest tak często podnoszona, że omawiam ją za każdym razem, gdy przemawiam publicznie. A rozwiązanie jest proste. Konsola działa, tylko trzeba zmienić zasadę wykonywania skryptów.

W niektórych kręgach skrypty mają złą opinię od czasów niesławnego wirusa I Love You z 2000 roku. Wirus ten nakłaniał do otwarcia załącznika do wiadomości e-mail zawierającego szkodliwy program wysyłający za pomocą kodu VBScript kopię wirusa do wszystkich adresatów znajdujących się na liście adresowej użytkownika oraz powodujący różne inne szkody. Ścisłe rzecz biorąc, jest to problem dotyczący samych użytkowników, nie skryptów, ale konsola Windows PowerShell ma wbudowane zabezpieczenie także przed tego rodzaju sytuacjami.

Podstawowa instalacja konsoli Windows PowerShell obsługuje standardowe polecenia, ale nie wykonuje skryptów, ponieważ ma ustawioną ograniczoną zasadę wykonywania skryptów. Ustawienie to można sprawdzić za pomocą polecenia `Get-ExecutionPolicy`. Z poziomu kont administracyjnych można zmieniać tę zasadę za pomocą poleceń podobnych do poniższego.

```
Set-ExecutionPolicy -ExecutionPolicy RemoteSigned
```

Takie ustawienie jest dobrym rozwiązaniem, ponieważ umożliwia wykonywanie skryptów lokalnych, a jednocześnie blokuje skrypty pochodzące z internetu i dysków mapowanych UNC, które nie mają podpisu certyfikatem akceptowanym przez system. Inne możliwości to `AllSigned` (wszystkie skrypty muszą być podpisane) i `Unrestricted` (można uruchamiać wszystkie skrypty, ale dla skryptów z internetu wyświetlane jest ostrzeżenie). Ustawienia `AllSigned` należy używać, gdy skonfigurowana jest infrastruktura podpisów kodu. Ustawienie `Unrestricted` jest niezalecane. W konsoli Windows PowerShell 2.0 wprowadzono przełącznik `bypass`, który nie blokuje żadnych skryptów i nie wyświetla jakichkolwiek ostrzeżeń. Przełącznika tego można używać, gdy konsola Windows PowerShell jest wbudowana w inną aplikację albo zastosowano inny model zabezpieczeń.

Inną często powtarzaną skargą jest to, że skryptów Windows PowerShell nie można uruchamiać za pomocą dwukrotnego kliknięcia przyciskiem myszy, które powoduje otwarcie skryptu w domyślnym edytorze. To również jest celowo zastosowane zabezpieczenie mające uchronić użytkownika przed przypadkowym uruchomieniem nieprzyjaznych skryptów.

Kolejnym hamulcem dla wykonywania skryptów jest brak bieżącego folderu na ścieżce. Jeśli wpisze się nazwę skryptu, Windows PowerShell nie znajdzie go w bieżącym folderze. Ale można to zmienić w sposób pokazany poniżej:

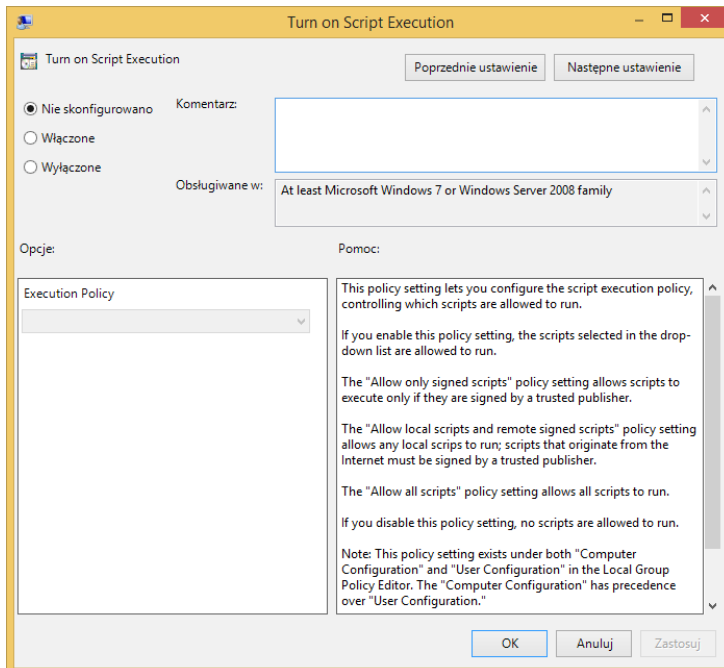
```
./mójskrypt.ps1
```

Czy dzięki tym wszystkim zabiegom konsola Windows PowerShell jest całkowicie bezpieczna? Oczywiście nie. Zawsze ktoś może zmienić ustawienia. Na forach czasami pojawia się pytanie, jak sprawić, aby można było uruchamiać skrypty Windows PowerShell za pomocą dwukrotnego kliknięcia przyciskiem myszy. Ale to, że coś **można** zrobić, wcale nie znaczy, że powinno się to robić! Dlatego zalecam ustawienie zasady wykonywania skryptów dostosowanej do potrzeb organizacji — najlepiej AllSigned albo ewentualnie RemoteSigned. Nie powinno się zmieniać pozostałych z wymienionych ustawień.

Na Twój system czyha wielu złoczyńców. Nie pomagaj im. Pozostaw domyślne ustawienia. Konfiguracja konsoli Windows PowerShell została gruntownie przemyślana i wszystkie jej ustawienia mają jakiś cel. Używaj konsoli Windows PowerShell, ale rób to bezpiecznie.

Wdrażanie zasady wykonywania skryptów za pomocą zasady grupowej

Najlepszym sposobem na zdefiniowanie zasady wykonywania skryptów Windows PowerShell jest użycie zasady grupowej. W systemie Windows Server 2008 R2 obiekt zasady grupowej (GPO) dla Windows PowerShell ma ustawienie o nazwie *Turn On Script Execution*. Obiekt ten można zastosować do komputera lub użytkownika. Jak widać na rysunku 15.4, jest tylko jedna opcja: *Execution Policy* (zasada wykonywania). Do wyboru są trzy wartości, które odpowiadają ustawieniom Signed, RemoteSigned i Unrestricted polecenia Set-ExecutionPolicy. W zasadzie grupowej nie można natomiast stosować ustawienia Bypass.



RYSUNEK 15.4. Ustawienia zasady grupowej dotyczącej wykonywania skryptów Windows PowerShell

Ustawienia zasady grupowej nie da się przesłonić. Dotyczy to także ustawiania bardziej restrykcyjnej zasady, jak również uruchomienia konsoli z uprawnieniami administratora. Jeśli spróbujesz zmienić zasadę wykonywania skryptów, gdy ustawiony jest obiekt GPO, konsola wyświetli błąd. Błąd ten jest nieco mylący, ponieważ na początku informuje, że polecenie zostało wykonane pomyślnie — rysunek 15.5.

```

Administrator: Windows PowerShell

PS C:\> Set-ExecutionPolicy -ExecutionPolicy Restricted -Force
Set-ExecutionPolicy : Windows PowerShell updated your execution policy successfully, but the setting is overridden by a
policy defined at a more specific scope. Due to the override, your shell will retain its current effective execution
policy of RemoteSigned. Type "Get-ExecutionPolicy -List" to view your execution policy settings. For more information p
lease see "Get-Help Set-ExecutionPolicy".
At line:1 char:1
+ Set-ExecutionPolicy -ExecutionPolicy Restricted -Force
+ ~~~~~
+ CategoryInfo          : PermissionDenied: (:) [Set-ExecutionPolicy], SecurityException
+ FullyQualifiedErrorId : ExecutionPolicyOverride,Microsoft.PowerShell.Commands.SetExecutionPolicyCommand

PS C:\>
  
```

RYСУNEK 15.5. Próba zmiany zasady wykonywania skryptów ustawionej przez obiekt GPO kończy się błędem

Wiedza tajemna

Posługiwanie się skryptami

Daniele Muscetta, menedżer ds. programów

Microsoft Corporation

Skrypty są niezwykle przydatnym narzędziem, przy użyciu którego można naprawdę wiele zdziałać, np. zintegrować i zautomatyzować różne zadania. Ale duże możliwości oznaczają też dużą odpowiedzialność.

Kiedyś już przerabialiśmy nadużywanie skryptów — mam na myśli wirusy w języku VBScript. Ktoś wysłał nam wiadomość e-mail, użytkownik ją kliknął i już jest za późno. Firma Microsoft postanowiła nie popełniać tego samego błędu dwa razy. Dlatego właśnie pliki z rozszerzeniem *.ps1*, w odróżnieniu od plików z rozszerzeniem *.vbs*, nie są wykonywane automatycznie.

Ponadto domyślnie skrypty Windows PowerShell nie są wykonywane nawet wtedy, gdy celowo je wywołamy w konsoli. Odpowiada za to tzw. zasada wykonywania skryptów. Zasady tej nie należy jednak traktować jako zabezpieczenia chroniącego przed wszelkim złem. Administrator może ją nawet wyłączyć. Można ją raczej porównać do pasów bezpieczeństwa w samochodzie — lepiej mieć je zapięte, ale zawsze można je odpiąć.

Instalator Windows PowerShell domyślnie stosuje bezpieczne ustawienie *Restricted*, ponieważ większość użytkowników konsoli i tak nigdy nie używa skryptów. Ustawienie to uniemożliwia uruchamianie jakichkolwiek skryptów, więc mimo iż jest ono bezpieczne, dla kogoś, kto jednak chce uruchamiać skrypty, jest też nieprzydatne (a Ty chcesz, bo po co inaczej czytałybyś tę książkę).

W związku z tym można ustawić jedną z innych często używanych opcji:

- **RemoteSigned** — skrypty i pliki konfiguracyjne pobrane z internetu muszą być podpisane przez zaufanego wydawcę.
- **Unrestricted** — konsola może wczytywać wszystkie pliki konfiguracyjne i uruchamiać wszystkie skrypty. Jeśli zostanie pobrany niepodpisany skrypt z internetu, konsola poprosi o potwierdzenie chęci jego wykonania.

Twórcy skryptów i administratorzy z reguły podejrzliwie traktują wszystkie nienapisane przez siebie skrypty i wykonują je tylko wtedy, gdy głęboko ufają autorowi albo dokładnie przejrzą kod źródłowy. Dlatego najczęściej zmieniają zasadę wykonywania skryptów na mniej restrykcyjną, np. **RemoteSigned** lub **Unrestricted**. Może to być uzasadnione na komputerach testowych, w których sprawdza się, czy dany skrypt nie jest szkodliwy. Ale co robi się w środowisku produkcyjnym, czyli na serwerach i komputerach klientów?

W systemie Windows XP konsola Windows PowerShell może być niedostępna, jeśli ktoś jej ręcznie nie zainstaluje, przez co możliwości administratora mogą być mocno ograniczone. W systemie Windows Server 2008 dostępna jest standardowo konsola Windows PowerShell 1.0. W systemach Windows 8.1 i Windows Server 2012 R2 standardowo dostępna jest już nawet konsola Windows PowerShell 4.0. Konsola ta daje duże możliwości w zakresie administrowania i automatyzacji, ale to oznacza, że trzeba zadbać o bezpieczny mechanizm uniemożliwiający wykonywanie skryptów pochodzących z niezaufanych źródeł.

Zasada **AllSigned** jest uważana przez większość osób za bezpieczne ustawienie. Pozwala ona na wykonywanie tylko tych skryptów i plików konfiguracyjnych, które są podpisane przez zaufanego wydawcę. Dotyczy to nawet skryptów pisanych na komputerze lokalnym. Jeśli jesteś administratorem systemu, możesz ustawić tę zasadę dla wszystkich użytkowników niemających wiedzy technicznej, aby mogli uruchamiać tylko określony zestaw bezpiecznych, podpisanych przez Ciebie skryptów (to tak, jakby zapiąć im pasy bezpieczeństwa w samochodzie). Sobie natomiast możesz ustawić mniej restrykcyjną zasadę wykonywania skryptów dla celów testowych.

Administrator w swoim własnym komputerze może ustawić dowolną zasadę wykonywania skryptów za pomocą polecenia `Set-ExecutionPolicy`, które zapisuje wybraną opcję w wartości klucza rejestru `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\PowerShell\1\ShellIds\Microsoft.PowerShell.Executionpolicy`. Dzięki temu, że w rejestrze pojawiło się ustawienie zasady wykonywania skryptów, można je również zastosować centralnie do klienckich stacji roboczych za pomocą poręcznych szablonów zasad grupowych (plików ADM) masowo ustawiających wartości rejestru.

Pliki ADM zostały napisane przez programistów systemu Windows i można je pobrać z Centrum pobierania Microsoft.

W konsoli Windows PowerShell 3.0 wprowadzono jeszcze inne opcje ustawień zasad wykonywania skryptów. Najpierw w Windows PowerShell 2.0 wprowadzono pojęcie zakresu, dzięki któremu można ustawić zasadę wykonywania obowiązującą tylko w wybranym zakresie. Dostępne zakresy to `Process`, `CurrentUser` oraz `LocalMachine`.

LocalMachine jest domyślnym zakresem przy ustawianiu zasady wykonywania skryptów. Ale administrator może chcieć zmienić tylko własną zasadę do celów testowych, a zasadę dla pozostałych użytkowników pozostawić bez zmian. W takim przypadku wystarczy wprowadzić zmianę obowiązującą tylko dla własnego zakresu w bieżącym procesie (bieżącej sesji w bieżącym procesie Windows PowerShell). Ustawienie to jest nietrwale i traci ważność po zamknięciu sesji. Ewentualnie za pomocą ustawienia `CurrentUser` administrator może ustawić zasadę dotyczącą tylko własnego profilu i zapisać to ustawienie w gałęzi rejestru `HKEY_CURRENT_USER`. Za pomocą zakresów można nawet ustawić różne zasady dla różnych użytkowników na tym samym komputerze.

W Windows PowerShell 2.0 i nowszych wersjach konsoli wprowadzono dodatkowo nowe ustawienia zasad wykonywania skryptów:

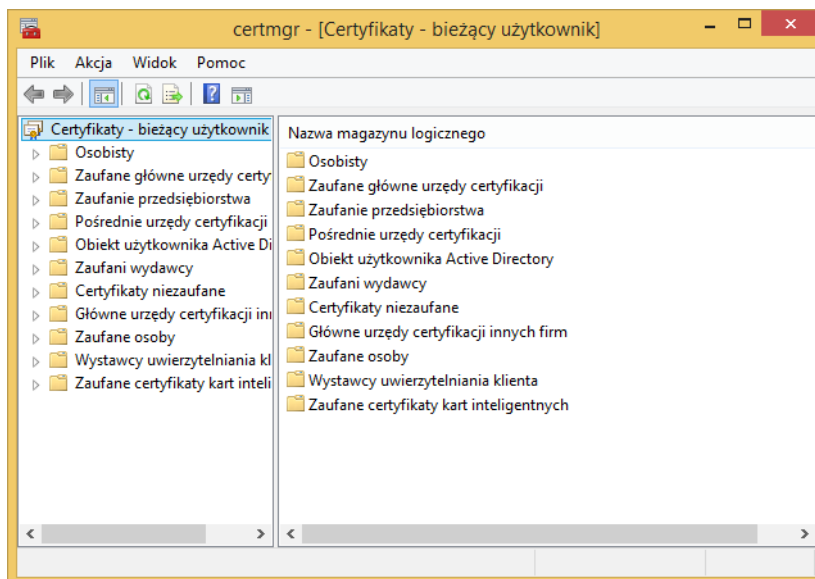
- **Undefined** — oznacza, że w przynajmniej jednym zakresie zasada wykonywania skryptów w ogóle nie jest ustawiona. W takim przypadku obowiązuje zasada ustawiona w jednym z innych zakresów. (Na przykład: jeśli dla bieżącego użytkownika nie jest ustawiona żadna zasada wykonywania skryptów, ale jest ustawiona dla lokalnej maszyny, to używana jest zasada lokalnej maszyny). Oczywiście jeśli w żadnym zakresie nie zostanie ustawiona jakakolwiek zasada wykonywania skryptów, to domyślnie stosowane jest ustawienie `Restricted`.
- **Bypass** — jest to ustawienie jeszcze mniej restrykcyjne od ustawienia `Restricted`, ponieważ nie blokuje niczego i nie powoduje wyświetlania jakichkolwiek ostrzeżeń ani monitów. Zasada ta jest przeznaczona dla konfiguracji, w których skrypty Windows PowerShell są wbudowane w większe aplikacje, lub dla konfiguracji, w których Windows PowerShell stanowi podstawę programu mającego własny model zabezpieczeń.

Przy takiej różnorodności ustawień administrator nie powinien mieć problemu ze zdefiniowaniem bezpiecznej i wygodnej konfiguracji dla swoich klientów i użytkowników przy jednoczesnym zachowaniu możliwości testowania i diagnozowania własnych skryptów.

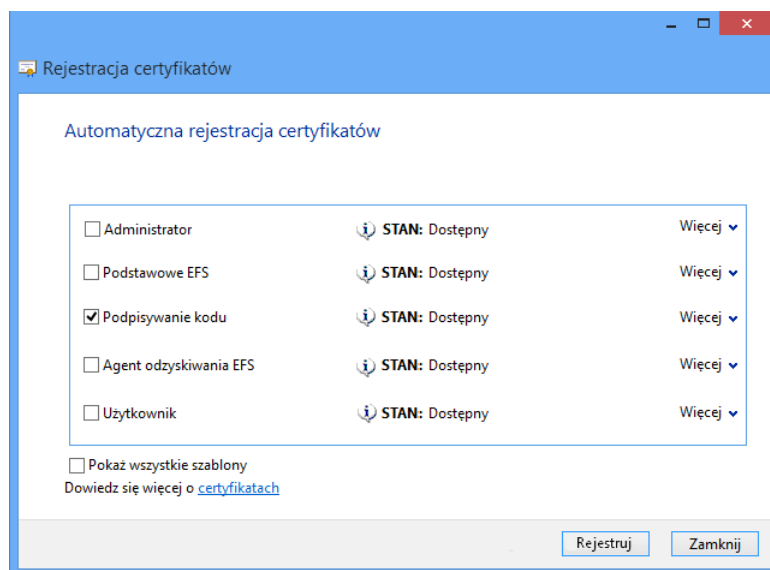
Podpisywanie kodu

Praca z podpisanymi skryptami w konsoli Windows PowerShell jest stosunkowo łatwa i wygodna, ponieważ dostępne są dwa polecenia, za pomocą których można podpisywać skrypty i weryfikować istniejące podpisy: `Get-AuthenticodeSignature` i `Set-AuthenticodeSignature`. Aby móc skorzystać z polecenia `Set-AuthenticodeSignature`, należy mieć certyfikat do podpisywania kodu. Aby mieć pewność, że używa się poprawnego certyfikatu, można skorzystać z Menedżera certyfikatów widocznego na rysunku 15.6.

Za pomocą Menedżera certyfikatów można wysłać do urzędu certyfikacji przedsiębiorstwa (CA) żądanie certyfikatu do podpisywania kodu. Na rysunku 15.7 widać okno kreatora rejestracji certyfikatów Menedżera certyfikatów.



RYSUNEK 15.6. Menedżer certyfikatów pozwala na wgląd do magazynów certyfikatów użytkownika



RYSUNEK 15.7. Za pomocą Menedżera certyfikatów można zażądać certyfikatu, jeśli dla domeny nie włączono rejestracji automatycznej

Dodatkowe źródła informacji

- Centrum skryptów portalu TechNet pod adresem <http://www.microsoft.com/technet/scriptcenter> zawiera wiele przykładów skryptów Windows PowerShell.
- Na stronie <http://blogs.msdn.com/b/dougste/archive/2007/09/06/version-history-of-the-clr-2-0.aspx> znajduje się opis historii wersji konsoli Windows PowerShell.
- Na stronie <http://msdn.microsoft.com/en-us/library/lh925568.aspx> znajdują się informacje na temat sprawdzania, która wersja platformy .NET jest zainstalowana w komputerze.
- Strona główna portalu MSDN znajduje się pod adresem <http://msdn.microsoft.com>.

Rozdział 16

Uruchamianie skryptów

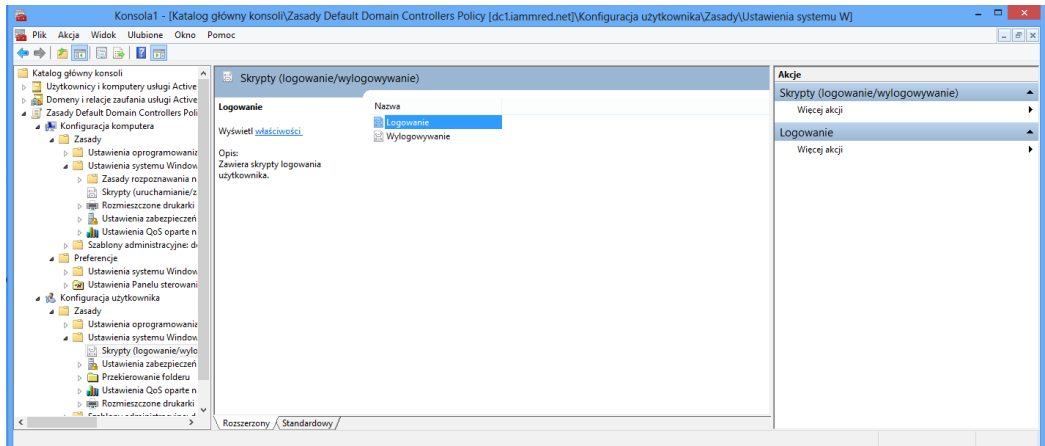
- Skrypty logowania
- Folder skryptów
- Samodzielne skrypty
- Skrypty pomocy technicznej
- Dodatkowe źródła informacji

Jeśli chodzi o uruchamianie skryptów w konsoli Windows PowerShell, to nie wszystkie skrypty są równe. W środowisku może znajdować się wiele różnych rodzajów skryptów. Niektóre z nich mogą być tylko skleconymi naprędce zbiorami poleceń, które czasami trzeba wpisać w oknie konsoli, a inne mogą być bliższe prawdziwym aplikacjom i tak też należy je traktować. Takie zaawansowane skrypty zazwyczaj pełnią ważną rolę w konfiguracji i automatyzacji systemu oraz są często używane przez pracowników pomocy technicznej.

Skrypty logowania

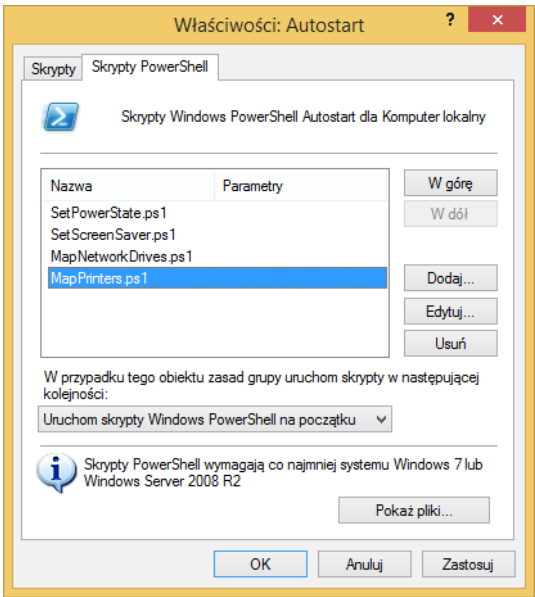
W Windows PowerShell 4.0 jako skryptu logowania można używać skryptu Windows PowerShell, pod warunkiem że kontroler domeny działa w systemie nie starszym niż Windows Server 2008 R2. W istocie skryptów Windows PowerShell można używać jako skryptów logowania, wylogowywania, uruchamiania oraz zamykania. Kiedyś skrypty logowania definiowało się za pomocą obiektu użytkownika w narzędziu Użytkownicy i komputery usługi Active Directory. Aby zastosować skrypt logowania Windows PowerShell, należy użyć zasady grupowej.

Aby zdefiniować skrypt logowania lub wylogowywania, należy zdefiniować ustawienie obiektu zasad grupy (ang. *Group Policy Object* — GPO) w węźle *Konfiguracja użytkownika/Zasady/Ustawienia systemu Windows/Skrypty (logowanie/wylogowywanie)*. Skrypty uruchamiania i zamykania definiuje się w węźle *Konfiguracja komputera (Zasady/Ustawienia systemu Windows/Skrypty (logowanie/wylogowywanie))*. Istnieje możliwość skonfigurowania wielu obiektów zasad grupy z różnymi skryptami i ustawieniami i dla optymalizacji wydajności można wyłączyć przetwarzanie niezdefiniowanych ustawień konfiguracyjnych. Posiadane obiekty GPO można połączyć z domeną lub jednostką organizacyjną, a nawet można je filtrować na podstawie przynależności do grup. Wszystkie te opcje dotyczące skryptów można ustawiać za pomocą zasad grupy w domenie Active Directory, jak pokazano na rysunku 16.1, albo za pomocą lokalnego edytora zasad grupy.



RYСУNEK 16.1. Za pomocą zasady grupy można zarządzać skryptami uruchamiania, zamykania, logowania oraz wylogowywania

Domyślnie skrypty logowania są przechowywane w folderze `NetLogon\Sysvol\Domain\Policies`. Edytor skryptów automatycznie znajduje skrypty znajdujące się w tym katalogu, jak widać na rysunku 16.2.



RYСУNEK 16.2. Skrypty są uruchamiane w kolejności występowania w oknie

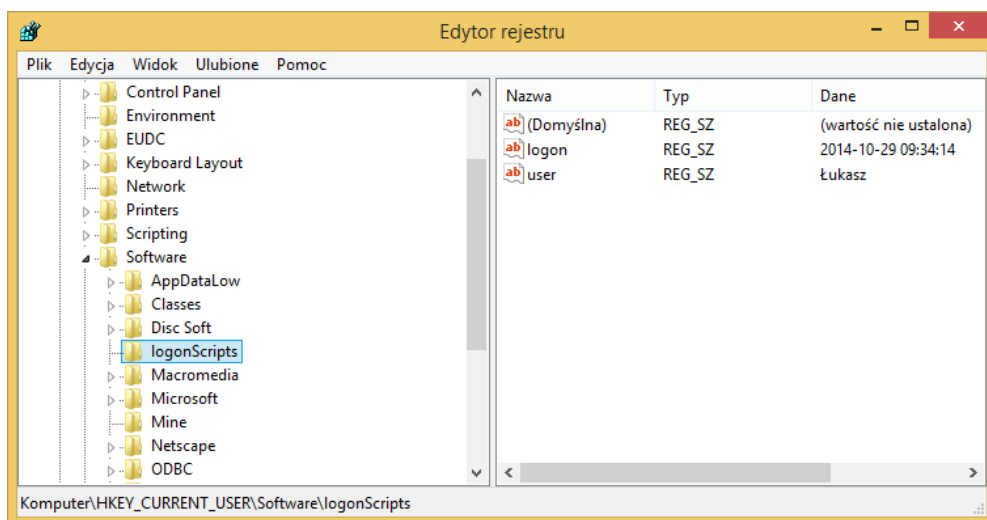
Co uwzględnić w skryptach logowania

Jako że wiele elementów konfiguracyjnych, które kiedyś umieszczano w skryptach logowania, przeniesiono do preferencji zasad grupy, wielu administratorów sieci w ogóle nie korzysta ze skryptów logowania. Jeśli masz możliwość użycia zasady grupy i uniknięcia kłopotów związanych

z tworzeniem i obsługą skryptów logowania, to eliminujesz jedno źródło potencjalnych błędów. Jednak większość sieci jest na tyle skomplikowana i zawiera tak dużo starych aplikacji, że nie da się uniknąć skryptów logowania. Kiedyś skryptów logowania używano do następujących celów:

- mapowanie dysków użytkowników,
- ustawianie domyślnych drukarek.

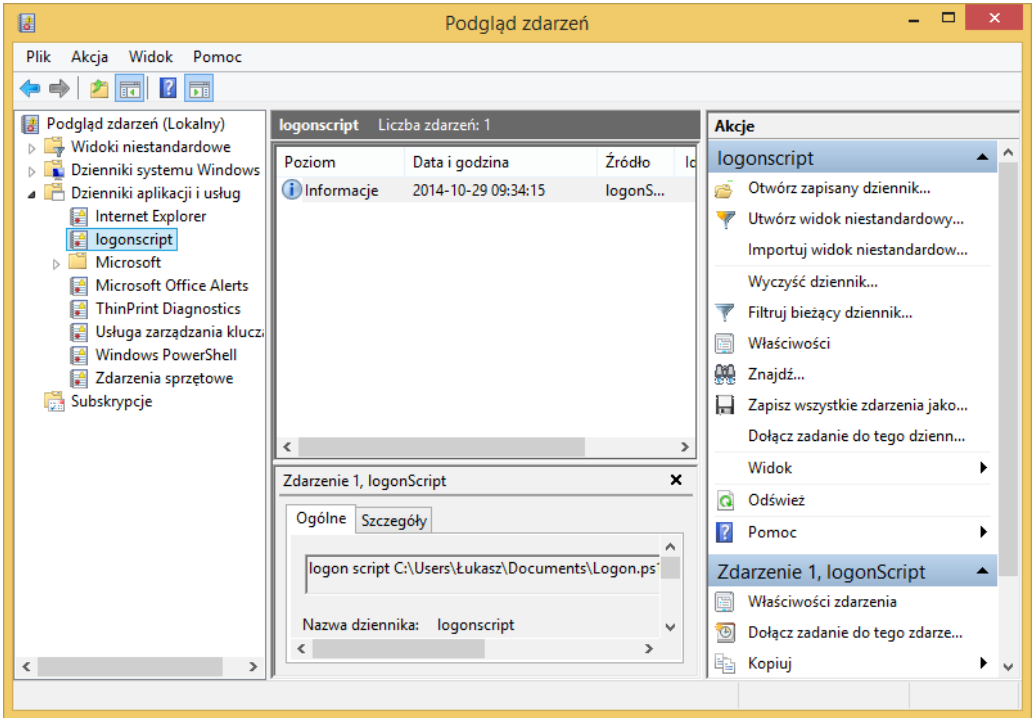
Ale wraz ze skryptami logowania Windows PowerShell pojawiły się nowe możliwości. Administrator może przeprowadzić prosty audyt albo zaimplementować mechanizm rejestrujący błędy. Przykład takiego skryptu logowania znajduje się w pliku o nazwie *Logon.ps1*. Skrypt ten tworzy nowy klucz rejestru zawierający nazwę użytkownika i godzinę zalogowania na komputerze. Ten nowy klucz jest pokazany na rysunku 16.3.



RYSUNEK 16.3. Skrypty logowania Windows PowerShell mogą zapisywać w rejestrze wyniki audytu

Inną cenną rzeczą, jaką wykonuje skrypt *Logon.ps1*, jest nowy dziennik zdarzeń o nazwie *Logonscripts*. Zapisuje w nim pełną ścieżkę do skryptu logowania oraz czas jego uruchomienia, co jest niezwykle ważną informacją podczas usuwania usterek. Jednym z kluczowych problemów jest określenie, który skrypt logowania jest wykonywany. Na rysunku 16.4 widać dziennik zdarzeń *Logonscripts*. Została w nim zapisana pełna ścieżka UNC w postaci hiperłącza. Klikając to hiperłącze w okienku szczegółów, można otworzyć skrypt logowania i w razie potrzeby go zmodyfikować. Możliwość wyświetlenia tekstu skryptu logowania z poziomu dziennika zdarzeń ułatwia usuwanie usterek.

Jedną z wielkich zalet zapisywania danych we własnym dzienniku zdarzeń za pomocą skryptu logowania jest to, że dziennik taki można bardzo łatwo przeszukiwać, i to zarówno lokalnie, jak i zdalnie. Aby pobrać dane, wystarczy użyć polecenia `Invoke-Command` i zapytania `Get-EventLog` w bloku skryptu. Poniżej znajduje się przykład połączenia się ze zdalnym komputerem o nazwie *client1*:



RYSUNEK 16.4. Skrypty logowania Windows PowerShell mogą zapisywać informacje we własnych dziennikach zdarzeń

```
PS C:\> invoke-command -ComputerName client1 -ScriptBlock {Get-EventLog -LogName logonscript} |
format-table timewritten, message -wrap
Timewritten                                     Message
-----
9/7/2013 12:06:29 PM                             logon script ran at 09/07/2013 12:06:29
9/7/2013 12:06:11 PM                             logon script \\tsclient\C\data\BookD0cs\P
S4_BestPractices\Scripts\scripts_ch16\Log
on.ps1 ran at 09/07/2013 12:06:11
9/7/2013 12:05:30 PM                             logon script \\tsclient\C\data\BookD0cs\P
S4_BestPractices\Scripts\scripts_ch16\Log
on.ps1 ran at 09/07/2013 12:05:30
```

Poniżej znajduje się kompletny kod źródłowy skryptu *Logon.ps1*:

```
Logon.ps1
$ErrorActionPreference = "SilentlyContinue"
if(-not(Test-path -path HKCU:\Software\logonScripts))
{
    new-Item -path HKCU:\Software\logonScripts
    new-ItemProperty -path HKCU:\Software\logonScripts -name logon '
    -Value $(get-date).tostring() -Force
    new-ItemProperty -path HKCU:\Software\logonScripts -name user '
    -Value $env:USERNAME -Force
}
else
{
}
```

```

set-Itemproperty -path HKCU:\Software\logonScripts -name logon '
-Value $(get-date).tostring() -Force
set-Itemproperty -path HKCU:\Software\logonScripts -name user '
-Value $env:USERNAME -Force
}

try
{
    New-EventLog -source logonscript -logname logonscript
}
Catch{ [System.Exception] }
Finally
{
    Write-EventLog -LogName logonscript -Source logonScript '
-EntryType information '
-EventId 1 '
-Message "Skrypt logowania $($myinvocation.invocationName) wykonany dnia $(get-date)"
}
$ErrorActionPreference = "Continue"

```

Zapiski praktyka

Wes Stahler

Prezes grupy użytkowników PowerShell z Central Ohio

Poproszono mnie o wykonanie pomiarów na jednym z naszych dużych współdzielonych dysków. Konkretnie chodziło o wyświetlenie rozmiaru każdego z katalogów znajdujących się o jeden poziom poniżej podanego katalogu głównego. Nic trudnego, ale dysk, o którym mowa, był gigantyczny, przez co wykonywanie na nim polecenia `Get-ChildItem` mogłoby trwać kilka godzin.

Jest kilka możliwości do wyboru:

- Napisanie skryptu wykonującego polecenie `Get-ChildItem` na ścieżce do katalogu głównego.
- Wykorzystanie przepływu pracy w PowerShell.
- Wykonanie zadań w tle.

Postanowiłem utworzyć po jednym zadaniu w tle dla każdego podfolderu. Dodatkowo zabezpieczyłem się przed wyczerpaniem pamięci, ustalając maksymalną liczbę zadań, jakie mogą działać naraz. Poniżej znajduje się kod ilustrujący ten proces:

```

# folder główny
$Path = '\\dfs-p01\dfs\Shared\COM\'

# Pobiera foldery znajdujące się o jeden poziom niżej od głównego.
$Folders = Get-ChildItem -Path $path -Directory | Sort-Object Name

# Blok pętli pobierający rozmiar każdego podfolderu
subfolder

```

```
$sb = {
    param([string]$path)
    $sum = (Get-ChildItem -Path $path -Recurse -Force -ErrorAction SilentlyContinue |
    Measure-Object -Property Length -Sum).Sum
    $dt = Get-Date -Format yyyyMMdd
    "{0},{1},{2:##.##}" -f $dt,$path,([long]$sum/1GB)
}

<# Przegląda iteracyjnie podfoldery, tworząc zadanie dla każdego z nich, dopóki nie dojdzie
do liczby maksymalnej. Po osiągnięciu liczby maksymalnej odczekuje 10 sekund i ponawia próbę. #>
Foreach ($Folder in $Folders) {
    While ($(Get-Job -state running).count -ge 5){
        "{0}: czeka 10 skund..." -f $folder
        Start-Sleep -Seconds 10
    }

    $name = $Folder.FullName
    Start-Job -ScriptBlock $sb -ArgumentList $name
}

# Zapisuje wyniki w zmiennej do dalszej analizy
$jobs = Get-Job | Wait-Job | Receive-Job
```

Dzięki ograniczeniu maksymalnej liczby zadań w tle skrypt działa znacznie szybciej, niż gdyby sekwencyjnie iterował przez wszystkie foldery za pomocą polecenia `Get-ChildItem`.

Metody wywoływania skryptów logowania

Skrypty logowania są oczywiście wywoływane podczas logowania użytkownika. Kiedyś skrypty takie przypisywano użytkownikom przy użyciu narzędzia Użytkownicy i komputery usługi Active Directory, ale niestety w takich skryptach nie ma możliwości uruchamiania skryptów Windows PowerShell.

Dlatego skrypty logowania, a także wylogowywania, uruchamiania i zamykania, powinno się przypisywać w zasadach grup. Wielką zaletą skryptów wylogowywania i zamykania jest to, że za ich pomocą można usuwać mapowania dysków i drukarek, co znacznie przyspiesza późniejsze uruchamianie komputera, gdy domyślna drukarka lub współdzielony udział będą niedostępne.

Zapiski praktyka

Niklas Goude, MVP Windows PowerShell

Właściciel produktu ZervicePoint, Enfo Zipper

„O, widzę, że w Windows PowerShell umiesz zrobić praktycznie wszystko przy użyciu prostych jednowierszowych poleceń. Czy to jest bezpieczne?”.

Takie pytanie bardzo często słyszę od swoich klientów. Staram się im wyjaśniać, że użytkownicy mają ograniczone uprawnienia i że użytkownik o wysokich uprawnieniach może wykonywać zadania administracyjne niezależnie od tego, czy używa konsoli Windows PowerShell, czy nie. Tłumaczę też, że dobrym zwyczajem jest nadawanie użytkownikom systemu jak najniższych uprawnień.

Oczywiście czasami użytkownik potrzebuje większych uprawnień, aby wykonać niektóre czynności. Dotyczy to na przykład pracowników pomocy technicznej. Jednym ze sposobów na umożliwienie tym pracownikom wykonywania ich codziennej pracy bez nadawania im uprawnień administracyjnych jest tworzenie ograniczonych punktów końcowych.

Ograniczony punkt końcowy pozwala na określenie zestawu poleceń, funkcji i skryptów dostępnych dla danego użytkownika. Poza tym punkt końcowy można tak skonfigurować, aby działał z uprawnieniami innego konta, np. serwisowego.

Powiedzmy, że chcemy, aby nasi pracownicy z działu pomocy technicznej mogli zarządzać użytkownikami, komputerami i grupami Active Directory za pomocą konsoli Windows PowerShell.

Najpierw zobaczymy, jak się konfiguruje ograniczony punkt końcowy. Konsola Windows PowerShell zawiera polecenie `New-PSSessionConfigurationFile`. Generuje ono plik zawierający ustawienia definiujące konfigurację sesji. Jako że chcemy ograniczyć możliwości naszych pracowników do zarządzania tylko użytkownikami, grupami i komputerami w Active Directory, możemy wygenerować plik konfiguracyjny za pomocą poniższego polecenia:

```
New-PSSessionConfigurationFile -Path C:\HelpDesk.pssc -SessionType RestrictedRemoteServer
-ModulesToImport ActiveDirectory -VisibleCmdlets Get-ADUser,Set-ADUser,Get-ADComputer,
Set-ADComputer,Get-ADGroup, Set-ADGroup
```

Polecenie to tworzy nowy plik konfiguracji ograniczającej możliwości użytkownika do uruchamiania tylko następujących poleceń: `Get-ADUser`, `Set-ADUser`, `Get-ADComputer`, `Set-ADComputer`, `Get-ADGroup` oraz `Set-ADGroup`. Ponadto parametr `ModulesToImport` został ustawiony na `ActiveDirectory`.

Jako że pracownicy pomocy technicznej nie mają uprawnień do zarządzania użytkownikami, komputerami i grupami Active Directory, należy tak skonfigurować punkt końcowy, aby miał uprawnienia konta posiadającego odpowiednie uprawnienia.

Za pomocą polecenia `Get-Credential` można zapisać poświadczenia konta w obiekcie `PSCredential`:

```
$runAsCred = Get-Credential domain\serviceaccount
```

Po utworzeniu pliku konfiguracji i zapisaniu obiektu poświadczeń można utworzyć nową konfigurację sesji za pomocą polecenia `Register-PSSessionConfiguration`. Poniżej znajduje się przykład, jak to zrobić:

```
Register-PSSessionConfiguration -Name HelpDesk -Path C:\HelpDesk.pssc -Force -RunAsCredential
$runAsCred
```

W kodzie tym użyto obiektu `PSCredential` jako wartości parametru `RunAsCredential`. Ponadto dodano ścieżkę do wygenerowanego wcześniej pliku konfiguracji.

W standardowej konfiguracji sesji trzeba być członkiem grupy `BUILTIN\Administrators` lub `BUILTIN\Remote Management Users`. Jako że nie chcemy dodawać naszych pracowników pomocy technicznej do żadnej z tych grup, za pomocą polecenia `Set-PSSessionConfiguration` zezwolimy dodatkowym grupom na dostęp do konfiguracji sesji, jak pokazano poniżej:

```
Set-PSSessionConfiguration -Name HelpDesk -ShowSecurityDescriptorUI
```

Polecenie to przyznaje dodatkowej grupie uprawnienia do używania konfiguracji sesji.

Dzięki temu każdy użytkownik należący do grupy określonej w powyższym poleceniu może używać ograniczonego punktu końcowego do wykonywania udostępnianych przez niego poleceń.

```
Enter-PSSession -ComputerName SRV01 -ConfigurationName HelpDesk
```

Powyższe polecenie ilustruje sposób połączenia się z ograniczonym punktem dostępowym przez pracownika pomocy technicznej.

Folder skryptów

Skrypty Windows PowerShell można przechowywać w dowolnym folderze, zarówno lokalnie, jak i na zdalnym udziale. Jeśli skrypty przechowywane są na zdalnym udziale, udział ten należy dodać do zaufanej strefy internetowej, aby można je było wykonywać bez przeszkód ze strony środowiska wykonawczego. Jeśli udział nie jest dodany do zaufanej strefy, użytkownik zostanie poproszony o zrobienie tego przy pierwszym uruchamianiu skryptu. Jeśli zaznaczymy, że ufamy danemu skryptowi, będziemy mogli go używać. Można też ustawić zasadę wykonywania skryptów na Bypass, która wyłącza wszelkie ostrzeżenia dotyczące strefy internetowej.

Wdrażanie lokalne

Jednym ze sposobów na uniknięcie problemu ze strefami internetowymi jest zapisanie skryptów w folderze lokalnym na lokalnych stacjach roboczych. Jest to także ważna kwestia do uwzględnienia, jeśli używane są skrypty logowania i wylogowywania, które mogą stać się niedostępne, jeśli będą przechowywane zdalnie. Kolejnym argumentem przemawiającym na korzyść lokalnego przechowywania skryptów jest brak narzutu spowodowanego kopiowaniem skryptu uruchamianego przez sieć. Natomiast wadą lokalnego przechowywania plików jest to, że trudniej jest zapewnić ich spójność ze skryptami przechowywanymi w innych miejscach. Można to rozwiązać przez oznaczanie numerów wersji kolekcji skryptów. Numer taki można zapisać w rejestrze i sprawdzać go podczas logowania. Jeśli pojawi się nowa wersja, na stację roboczą można skopiować nowe skrypty.

Lokalne wdrażanie pakietu MSI

Jeśli skrypty są kopiowane z udziału sieciowego, który nie jest dodany do zaufanej strefy internetowej, to można je uruchamiać tylko po ustawieniu zasady wykonywania skryptów Windows PowerShell na Bypass lub Unrestricted. Jeżeli skrypty zostaną zainstalowane lokalnie, system umieści je w lokalnej strefie internetowej. Łatwym sposobem na wdrożenie skryptów jest utworzenie pakietu MSI tworzącego folder skryptów i kopiującego do niego skrypty. Potem można wdrożyć taki pakiet za pomocą zasady grupy.

Samodzielne skrypty

Samodzielne skrypty nie mają żadnych zewnętrznych zależności. Działają zawsze, ponieważ nie potrzebują żadnych modułów, które mogą, ale nie muszą być zainstalowane. Nie dołączają żadnych plików, bo te również mogą być niedostępne. Samodzielne skrypty są z reguły dłuższe od innych rodzajów skryptów, ponieważ zawierają wszystkie funkcje, stałe, zmienne i aliasy, które są im potrzebne do działania.

W środowisku firmowym, w którym mamy całkowitą kontrolę nad komputerem, możemy zapewnić dostępność potrzebnych modułów, stałych, zmiennych i aliasów. Zawsze można sprawdzić, czy dany moduł jest dostępny, a jeśli nie jest, to można go skopiować z udziału sieciowego i zainstalować albo zapisać błąd w dzienniku. Ponadto można wysłać wiadomość do pomocy technicznej z informacją o brakujących zależnościach.

Diagnostyka

Skrypty diagnostyczne mają za zadanie rozwiązywanie konkretnych problemów. W skryptach takich często używa się klas WMI do mierzenia wydajności i odpowiednich poleceń Windows PowerShell. Tego rodzaju skrypty zazwyczaj wykonuje się na żądanie — w zależności od potrzeby w sposób zdalny lub lokalnie.

Raportowanie i kontrolowanie

Skrypty raportujące zbierają informacje z określonego komputera. Często wykonuje się je jednocześnie na wielu komputerach, aby zgromadzić w jednym miejscu dane dotyczące wielu maszyn. Skrypty te można wywoływać w skryptach logowania lub wylogowywania lub bezpośrednio w konsoli Windows PowerShell.

Zapiski praktyka

Don Jones

Prezes PowerShell.Org

To, że przepływ pracy (ang. *workflow*) jest fascynującą (i wciąż dość nową) funkcją, nie znaczy, że nadaje się do użycia w każdej sytuacji. Jest to zewnętrzna technologia, a nasz kod Windows PowerShell jest na nią tłumaczony. To oznacza, że trzeba nieco zmienić styl kodowania, a niektóre czynności, jak choćby debugowanie, są znacznie trudniejsze do wykonania. Innymi słowy: użycie przepływu pracy zmusza nas do intensywniejszego myślenia. To prawda, że przepływy pracy mają wiele zalet, ale trzeba mieć pewność, że się je wykorzysta, zanim się zdecyduje ponieść dodatkowe koszty. Za pomocą przepływu pracy można na przykład pracować na zdalnych komputerach, ale to samo można robić w o wiele prostszy sposób za pomocą narzędzi do pracy zdalnej Windows PowerShell. Przepływ pracy pozwala na równoległe wykonywanie zadań, ale w niektórych przypadkach to samo można uzyskać za pomocą zadań.

Jeśli postanowisz używać przepływów pracy, to napotkasz mniej problemów, jeśli zaczniesz od budowy normalnej funkcji, bez użycia przepływu pracy. Stosuj najlepsze praktyki programowania, takie jak wpisywanie całych nazw poleceń i funkcji, ponieważ w przepływach pracy jest to obowiązkowe (zwłaszcza w Windows PowerShell 3.0). Przetestuj swoją funkcję na komputerze lokalnym, na którym masz dostęp do wszystkich funkcji diagnostycznych. Potem możesz zacząć przerabiać swoją funkcję. Jeśli natkniesz się na jakiś problem, ponownie uruchom skrypt jako zwykłą funkcję, aby dowiedzieć się, czy problem dotyczy samego kodu, czy też środowiska przepływów pracy.

Skrypty pomocy technicznej

Skrypty pomocy technicznej to specjalna klasa samodzielnych skryptów, które często wykonują wiele czynności naraz i zazwyczaj muszą umożliwiać pracę na kilku komputerach. Podczas gdy wiele skryptów zapisuje informacje w bazach danych, plikach tekstowych, plikach CSV lub na stronach internetowych, większość skryptów pomocy technicznej wyświetla informacje w konsoli Windows PowerShell, ponieważ są one potrzebne od razu do rozwiązania problemu zgłoszonego przez dzwoniącego klienta. Dane takie nie są w żaden sposób utrwalane.

Unikaj edytowania

Dobrze zaprojektowany skrypt pomocy technicznej powinien być obsługiwany za pomocą parametrów wiersza poleceń. Skryptów tego rodzaju nie powinno się edytować, ponieważ łatwo można wprowadzić błąd albo spowodować, że skrypt zacznie działać w inny sposób, niż powinien. Skrypty pomocy technicznej należy traktować jak narzędzia diagnostyczne do wykrywania i usuwania określonych problemów. Jednym ze sposobów na zapewnienie ich integralności jest podpisanie plików. Jakakolwiek ingerencja w podpisany skrypt skutkuje unieważnieniem podpisu, a więc po wprowadzeniu zmian należy ponownie podpisać skrypt.

Funkcja rozwiązywania problemów na zdalnych komputerach w skryptach pomocy technicznej powinna być dostępna przez parametr `-computer` oraz zawierać różne inne parametry pomocnicze.

Dostarcz dobrej jakości pomoc

Jako że skrypty pomocy technicznej mogą mieć wiele parametrów wiersza poleceń, konieczne jest dostarczenie opisu wszystkich parametrów, dozwolonych zakresów wartości oraz przykładów składni. W skrypcie *DisplayProcessor.ps1* użyto znaczników pomocy do zdefiniowania abstraktu, opisu, przykładów i innych informacji dotyczących sposobu użycia skryptu. Skrypt ten jest w pełni zintegrowany z poleceniem `Get-Help` i obsługuje standardowe parametry, jak pokazano poniżej:

```
Get-Help DisplayProcessor.ps1
Get-Help DisplayProcessor.ps1 -full
Get-Help DisplayProcessor.ps1 -detailed
Get-Help DisplayProcessor.ps1 -examples
```

Poniżej znajduje się kompletny kod źródłowy skryptu *DisplayProcessor.ps1*:

DisplayProcessor.ps1

```
<#
.Synopsis
Wyświetla informacje o procesorze komputera.
.Description
Skrypt ten wyświetla informacje o procesorze lokalnego lub zdalnego komputera. Zwraca
następujące informacje: użycie procesora, szybkość procesora, rozmiar bufora L2, liczba
rdzeni oraz architektura.
.Example
DisplayProcessor.ps1
Wyświetla informacje o procesorze komputera lokalnego.
.Example
DisplayProcessor.ps1 -computer berlin
Wyświetla informacje o procesorze zdalnego komputera o nazwie berlin.
.Inputs
[string]
.Outputs
[string]
.Notes
NAZWA: Windows PowerShell Best Practices
AUTOR: Ed Wilson
DATA EDYCJI: 7.9.2013
WERSJA: 1.0.1
SŁOWA KLUCZOWE:
.Link
Http://www.ScriptingGuys.com
#Requires -Version 2.0
#>
param(
    [Parameter(position=0)]
    [string]
    [alias("CN")]
    $computer=$env:computername
) #end param

# Początek sekcji z funkcjami
function New-Underline
{
    <#
    .Synopsis
    Tworzy podkreślenie o długości łańcucha wejściowego.
    .Example
    New-Underline -strIN "Witaj, świecie"
    .Example
    New-Underline -strIn "Morgen welt" -char "-" -sColor "blue" -uColor "yellow"
    .Example
    "to jest łańcuch" | New-Underline
    .Notes
    NAZWA:
    AUTOR: Ed Wilson
    DATA EDYCJI: 5/20/2009
    WERSJA: 1.0.0
    SŁOWA KLUCZOWE:
    .Link
    Http://www.ScriptingGuys.com
```

```

#>
[CmdletBinding()]
param(
    [Parameter(Mandatory = $true, Position = 0, valueFromPipeline=$true)]
    [string]
    $strIN,
    [string]
    $char = "=",
    [string]
    $sColor = "Green",
    [string]
    $uColor = "darkGreen",
    [switch]
    $pipe
) #end param
$strLine= $char * $strIN.length
if(-not $pipe)
{
    Write-Host -ForegroundColor $sColor $strIN
    Write-Host -ForegroundColor $uColor $strLine
}
Else
{
    $strIN
    $strLine
}
} #end funkcja New-Underline
Function Get-Processor
{
    Param ([string]$computer)
    get-wmiobject -class win32_processor -computername $computer |
    foreach-object '
    {
        New-Underline("Dane procesora komputera $computer")
        $_.psobject.properties |
        foreach-object '
        {
            If($_.value)
            {
                if ($_.name -match "__"){}
                ELSE
                {
                    $Processor +=@{ $_.name = $_.value }
                } #end else
            } #end if
        } #end foreach property
        $Processor ; $Processor.clear()
    } #end foreach Processor
    Return
} #end Get-Processor
# punkt początkowy skryptu

Get-Processor -computer $computer

```

Dodatkowe źródła informacji

- Centrum skryptów portalu TechNet pod adresem <http://www.microsoft.com/technet/scriptcenter> zawiera wiele przykładów skryptów logowania i pomocy technicznej.
- Wszystkie skrypty opisane w tym rozdziale są dostępne do pobrania w repozytorium centrum skryptów portalu TechNet pod adresem <http://gallery.technet.microsoft.com/scriptcenter/PowerShell-40-Best-d9e16039>.

Rozdział 17

Kontrola wersji skryptów

- Dlaczego warto stosować kontrolę wersji
- Programy do kontroli wersji
- Dodatkowe źródła informacji

W kwestii kontroli wersji skryptów Windows PowerShell opinie są podzielone. Na każdej konferencji TechEd w Północnej Ameryce przez cztery lata z rzędu kwestia wersjonowania Windows PowerShell była poruszana w sesji Birds of a Feather.

Dlaczego warto stosować kontrolę wersji

Kontrola wersji polega na śledzeniu zmian wprowadzanych w skryptach produkcyjnych. Jest kilka powodów do robienia tego; niektóre przedstawiam poniżej.

- Unikanie wprowadzania błędów do istniejących skryptów produkcyjnych.
- Możliwość precyzyjnego usuwania usterek ze skryptów produkcyjnych.
- Śledzenie zmian w skryptach produkcyjnych.
- Utrzymywanie listy wszystkich skryptów produkcyjnych.
- Utrzymywanie zgodności z innymi skryptami.

Zapiski praktyka

Kontroluj swoje źródła

Don Jones, MVP Microsoft PowerShell
ConcentratedTech.com

Niestety niektórzy nie traktują swoich skryptów zbyt poważnie. Dla mnie skrypt jest efektem długiej pracy w wierszu poleceń, który chciałbym zachować na przyszłość. Wolałbym go nie stracić dlatego, że któryś z moich współpracowników zacznie w nim grzebać albo zginie mi

jedyna kopia. Programiści już dawno znaleźli rozwiązanie tego typu problemów i nazwali je **kontrolą wersji**. Jeżeli poważnie traktujesz swoje skrypty, to powinieneś zacząć korzystać z systemu kontroli wersji. A jeśli nie bierzesz tego na tyle poważnie, aby chronić swoje skrypty, to po co w ogóle je pisziesz?

W repozytoriach kontroli wersji oprogramowania przechowywane są **wszystkie** poprzednie wersje skryptów, dzięki czemu w każdej chwili można wrócić do wybranej wersji. W większości repozytoriów, aby pobrać skrypt do modyfikacji, należy go najpierw wyewidencjonować, chociaż niektóre zachowują wszystkie zapisywane wersje, eliminując kłopotliwe czynności związane z ewidencjonowaniem skryptów i usuwaniem ich z ewidencji. Najlepiej znaleźć sobie dobrej jakości edytor — a **dobrej jakości** edytor skryptów zawiera funkcje do łączenia się z systemem kontroli wersji, czyli inaczej mówiąc, współpracuje z popularnymi systemami kontroli wersji.

Jeśli w Twojej firmie używany jest już system kontroli wersji, to świetnie. Zapewne jest to jakieś rozwiązanie oparte na narzędziu Microsoft Visual SourceSafe, Microsoft Team Server albo CSV/Subversion, które są programami o otwartym kodzie źródłowym. Używaj firmowego systemu kontroli wersji — po prostu utwórz projekt Windows PowerShell i wprowadź do niego wszystkie swoje skrypty. Jeśli jednak w Twojej firmie nie używa się systemu kontroli wersji, pomyśl nad czymś prostszym niż te kombajny przeznaczone dla programistów. Na przykład firma SAPIEN Technologies oferuje program ChangeValue, a jeśli w wyszukiwarce internetowej poszukasz frazy „programy do kontroli wersji”, znajdziesz kilka innych dobrych narzędzi, niektóre o zabawnych nazwach, jak FileHamster, Git czy History Explorer. Istnieją też internetowe usługi kontroli wersji, np. *Beanstalkapp.com* i *Unfuddle.com*, i setki innych, do obsługi których potrzebny jest tylko klient Subversion (który może być dostępny w lepszych środowiskach skryptowych).

Jeśli Twoje dane są na tyle ważne, że chcesz je zapisać w pliku *.ps1*, to znaczy, że plik ten jest wystarczająco ważny, aby go zapisać w repozytorium systemu kontroli wersji. Pisanie skryptów Windows PowerShell **bez** kontroli wersji jest jak jazda bez zapiętych pasów bezpieczeństwa. Da się — i wiele osób tego nie żałuje — ale jeśli już przyjdzie żałować, to ten żal jest straszny.

Unikanie wprowadzania błędów

Jest mało prawdopodobne, że ktoś zmieniający coś w skrypcie nie wprowadzi przy okazji jakiegoś błędu. Proces pisania skryptów często sprowadza się do nanoszenia zmian w kodzie i właśnie szukania błędów. Nieważne, czy wprowadzane zmiany są drobne, czy duże, ryzyko uszkodzenia skryptu produkcyjnego zawsze jest duże. Jeżeli zmiana jest ważna, a błąd poważny, to możliwe, że już nigdy nie uda się przywrócić skryptu do działającego stanu. Ale dzięki systemowi kontroli wersji pracuje się tylko na kopii skryptu. Dopiero po zakończeniu wprowadzania zmian i przetestowania programu zostaje on wprowadzony jako nowy model produkcyjny do systemu kontroli wersji. Jeśli w kolejnej wersji pojawi się jakiś nieoczekiwany problem, zawsze można przywrócić poprzednią wersję. Produkcyjna wersja skryptu nie jest nigdy modyfikowana. Wszystkie zmiany są śledzone i wykonywane na kopiach.

Precyzyjne usuwanie usterek

Gdyby skrypty rejestrowano tylko według nazw, to nie dałoby się odróżniać od siebie ich poszczególnych wersji. Jeśli w skrypcie zostanie znaleziony błąd, a nie jest używany system kontroli wersji, należy uważnie przeczytać kod źródłowy jednej wersji i porównać go z kodem źródłowym innej wersji. Nie ma pewności, który skrypt jest nowszy ani który powinno się ostatecznie wdrożyć. Dzięki zapisywaniu wersji plików można szybko odróżnić poszczególne wersje skryptów.

Jeżeli użytkownik skryptów zgłasza problem ze skryptem, a używany jest system kontroli wersji, to wystarczy spytać tego użytkownika, której wersji skryptu używa, aby dowiedzieć się, czy nie posługuje się przestarzałym plikiem, czy też może po prostu odkrył nowy błąd w kodzie.

Śledzenie zmian

Niestety nie wszystkie zmiany dokonywane w skryptach produkcyjnych poprawiają niezawodność, wydajność, bezpieczeństwo i komfort użytkowania tych skryptów. Prawda jest taka, że każda zmiana może spowodować błąd, pogorszyć wydajność oraz skomplikować dotychczas prosty program. Jeżeli dokonana zmiana była poważna, to należy ją wycofać z produkcji.

Jeśli używany jest system kontroli wersji, to wystarczy wrócić do ostatniej działającej wersji skryptu. W przeciwnym razie rozwiązaniem jest otwarcie skryptu produkcyjnego i ręczne usunięcie z niego wszystkich ostatnio wprowadzonych zmian. Jeśli modyfikacje nie zostały odpowiednio opisane w komentarzach, to pozostaje już tylko poszukanie poprzedniej wersji skryptu w kopiach zapasowych.

Lista skryptów

Jeśli prowadzona jest kontrola wersji wszystkich skryptów produkcyjnych, to można sporządzić raport zawierający szczegółowe informacje o tym, które skrypty zostały przekazane do produkcji, a które są jeszcze niedokończone. Jeśli znajdzie się jakiś skrypt, którego nie ma na liście gotowych do produkcji, to znaczy, że skrypt ten nie został jeszcze autoryzowany do przekazania do użytku.

Zachowanie zgodności z innymi skryptami

W miarę jak biblioteka skryptów się rozrasta, w kolejnych skryptach mogą pojawiać się zależności od innych skryptów. Jest to spowodowane tym, że w niektórych skryptach używa się funkcji z innych skryptów, albo tym, że wyniki zwracane przez jeden skrypt są używane przez inne skrypty. W każdym razie, jeśli jakiś skrypt jest używany czy to pośrednio, czy to bezpośrednio przez inne skrypty, należy bardzo skrupulatnie notować wprowadzane w nim zmiany i dokładnie go testować, aby nie uniemożliwić działania innych skryptów.

Wiedza tajemna

Przechowywanie skryptów w repozytorium

Ian Farr, Premier Field Engineer

Microsoft Corporation

Wraz z powiększaniem się zbioru skryptów zaczynają się problemy z ich obsługą i przechowywaniem. A możliwość szybkiego znalezienia kodu, który napisało się kilka lat temu, pozwala zaoszczędzić sporo czasu i nerwów. Dlatego też zalecam dobrze przemyśleć, gdzie i jak przechowywać się magazyn skryptów.

Jeśli w firmie nie ma możliwości zainstalowania aplikacji do obsługi magazynu skryptów, skrypty powinno się przechowywać w udziale sieciowym. Możliwość przywracania danych oraz ich bezpieczeństwo i integralność to bardzo ważne kwestie. Zastanówmy się nad każdą z nich:

- Przywracanie danych — jeśli magazyn skryptów zostanie umieszczony w sieciowym udziale plikowym, to jest duża szansa, że znajdujące się w nim pliki będą automatycznie kopiowane i replikowane.
- Bezpieczeństwo danych — jako że niektóre skrypty w nieodpowiednich rękach mogą spowodować wiele szkód w środowisku informatycznym, udział sieciowy powinien chronić dane za pomocą uprawnień NTFS i wyliczania opartego na dostępie.
- Integralność — czasami można użyć opcji Enterprise Certificate Authority lub Commercial Certificate Authority w celu zdobycia certyfikatu do podpisywania swoich skryptów. To uniemożliwi konsoli Windows PowerShell uruchamianie zmienionych lub niepodpisanych skryptów.

Do przechowywania osobistych skryptów można użyć dysku we własnym komputerze, zewnętrznego dysku albo chmury, np. w usłudze SkyDrive. Niezależnie od wybranej opcji nadal nie można zapominać o przywracaniu danych oraz ich bezpieczeństwie i integralności:

- Przywracanie danych — istnieje narzędzie Kopia zapasowa systemu Windows, więc można napisać skrypt wykonujący kopię zapasową skryptów. Usługa przechowywania plików w chmurze zapewnia niezawodność, ale niezależnie od miejsca przechowywania danych zawsze warto mieć ich kilka kopii.
- Bezpieczeństwo danych — może nie obchodzi Cię, co się stanie z Twoimi skryptami, ale podejrzewam, że jest inaczej, więc rozważ kwestie fizycznego bezpieczeństwa, stosuj skomplikowane hasła i używaj rozwiązania BitLocker.
- Integralność — możesz utworzyć certyfikat z podpisem własnym do użytku na swoim komputerze. Niektórzy mogą powiedzieć, że we własnym komputerze jest to zbędne, ale wybór należy do Ciebie.

Najlepsze repozytoria, bez względu na to, czy są używane w firmie, czy do celów prywatnych, mają logiczną strukturę, jak poniższa:

- Typ skryptów (np. PS1, VBS).
- Technologia (np. AD, DHCP).
- Temat (np. Replikacja, Zarządzanie zakresem).
- Funkcja skryptu (np. Sprawdzanie replikacji, Dodanie opcji zakresu).

W moim repozytorium nazwy skryptów odzwierciedlają ich przeznaczenie. Ponadto dodaje słowa kluczowe lub znaczniki w łańcuchach miejscowych zawierające opis skryptu. Dzięki temu każdy skrypt można łatwo znaleźć za pomocą funkcji wyszukiwania. Można nawet posunąć się o krok dalej — jeden z moich kolegów z pracy napisał skrypt parsujący całe repozytorium, bo lubił mieć dane w określonej formie.

Przy tworzeniu i obsłudze repozytorium skryptów należy przemyśleć wiele spraw. Z czasem odkryjesz, co jest najlepsze dla Ciebie lub Twojej firmy.

Wewnętrzny numer wersji w komentarzach

Jednym z prostych sposobów na kontrolowanie wersji jest dodawanie numerów w komentarzach, z których bardzo łatwo można je w razie potrzeby odczytać. Ale technika ta wymaga ręcznego zmieniania numeru po każdej modyfikacji skryptu przez autora.

Z podejściem tym wiążą się dwie trudności. Po pierwsze: wymagana jest ręczna ingerencja, przez co osoba dokonująca zmian kodu musi pamiętać, aby zmienić też numer wersji. Gdy wprowadzane są drobne zmiany, takie jak modyfikacja komentarza, istnieje silna pokusa, aby nie zmieniać numeru wersji.

Po drugie: jeśli tworzona jest nowa wersja skryptu, to starą wersję trzeba zapisać pod zmienioną nazwą, aby nie kolidowała z nazwą nowej wersji. Problem ten można rozwiązać przez zapisywanie każdej wersji w osobnym folderze. Najnowsza wersja znajduje się w najnowszym folderze. Skrypt *Get-ScriptVersion.ps1* pobiera wersję skryptu i datę jego ostatniej modyfikacji. Wykorzystuje informacje dotyczące wersji i daty ostatniej modyfikacji z nagłówka skryptu, które widać poniżej:

```
.Notes
NAME: Windows PowerShell Best Practices
AUTHOR: Ed Wilson
LASTEDIT: 5/20/2009
VERSION: 1.0.0
KEYWORDS:
.Link
Http://www.ScriptingGuys.com
```

Poniżej znajduje się kompletny kod źródłowy skryptu *Get-ScriptVersion.ps1*:

Get-ScriptVersion.ps1

```
function get-ScriptVersion ([string]$path)
{
    $scripts = Get-ChildItem -Path $path -recurse
    ForEach($script in $scripts)
    {
        $info = New-Object psobject
        $scriptText = Get-Content $script.fullname
```

```

$info |
Add-Member -Name "name" -Value $script.name -MemberType noteproperty
$lastedit = $scriptText |
Select-String -Pattern "\s\d{1,1}/\d{1,2}/\d{1,4}"

if($lastedit.count -gt 1)
{
    $info |
    Add-Member -Name "LastEdit" -Value $lastedit[0].matches[0].value '
    -membertype noteproperty
}
if($lastedit.matches.count -gt 0)
{
    $info |
    Add-Member -Name "LastEdit" -Value $lastedit.matches[0].value '
    -membertype noteproperty -Force
}
$version = $scriptText |
Select-String -Pattern "\s\d\.\d\.\d"

if($version.count -gt 1)
{
    $info |
    Add-Member -Name version -Value $version[0].matches[0].value '
    -membertype noteproperty -Force
}
if($version.matches.count -gt 0)
{
    $info |
    Add-Member -Name version -Value $version.matches[0].value '
    -membertype noteproperty -Force
}
$info
$version = $lastedit = $scriptText = $null
} #end foreach
} #end funkcja get-ScriptVersion

# *** punkt początkowy skryptu ***

Get-ScriptVersion -path C:\data\BookD0cs\PS4_BestPractices\Scripts |
Format-Table -Property * -AutoSize -Wrap

```

Zwiększanie numerów wersji

Numeracja wersji skryptów raczej nie powinna zawierać więcej niż trzech elementów. Pierwsza liczba reprezentuje numer wersji głównej. A zatem numer 1.0.0 oznacza pierwsze wydanie skryptu bez jakichkolwiek zmian czy poprawek. Numer wersji głównej zmienia się tylko po wprowadzeniu dużych zmian w skrypcie. Zmiany te mogą polegać na przykład na dodaniu nowych funkcji wymagających przepisania znacznej części kodu skryptu od nowa. Zmiana numeru wersji pobocznej na 1.1.0 oznacza mniejsze zmiany w skrypcie i różne drobne poprawki, np. polepszające wydajność albo usprawniające jego działanie. Jeśli natomiast poprawi się literówkę, usunie jakiś błąd lub udoskonali obsługę błędów, należy zmienić numer wersji na 1.1.1.

Morał

Usuwanie niepoprawnej wersji skryptu

Pamiętaj, że każda zmiana wprowadzona w skrypcie powinna być odnotowana w postaci zmiany numeru wersji. Ciągle napotykam różne wersje tych samych skryptów bez możliwości łatwego ich rozróżnienia. Powinno się zachować działającą wersję skryptu i tak zmienić nazwy poprzednich wersji, aby łatwo było je zidentyfikować. W skrypcie powinno się zapisać tabelę wersji z listą zmian w poszczególnych wersjach. Dzięki temu unikniesz przypadkowego usunięcia nie tej co trzeba wersji skryptu.

Śledzenie zmian

Przy zmianie numeru wersji skryptu należy dodać komentarz zawierający informacje o tej zmianie. Komentarz ten można umieścić np. w notatkach w nagłówku. W skrypcie *Get-WindowsEdition.ps1* zostały wymienione wszystkie wersje, podano datę ostatniej edycji oraz napisano, która zmiana spowodowała obecną zmianę wersji. Poniżej znajduje się opisywana część skryptu:

.Notes

NAZWA: Get-WindowsEdition.ps1

AUTOR: Ed Wilson

DATA EDYCJI: 20.5.2009

WERSJA: 1.2.0 Dodano znaczniki pomocy

1.1.1 4/2/1009 Dodano odnośnik do strony <http://www.ScriptingGuys.com>

1.1.0 4/1/2009 Zastosowano wyrażenie regularne

SŁOWA KLUCZOWE: najlepsze zwyczaje w pracy z konsolą Windows PowerShell

Poniżej znajduje się kompletny kod źródłowy skryptu *Get-WindowsEdition.ps1*:

Get-WindowsEdition.ps1

<#

.Synopsis

Sprawdza wersję systemu Windows zainstalowanego na lokalnym komputerze.

.Description

Sprawdza wersję systemu Windows zainstalowanego na lokalnym komputerze.

Zwraca informację w rodzaju Windows 7 Enterprise.

.Example

Get-WindowsEdition.ps1

Wyświetla wersję systemu Windows lokalnego komputera.

.Inputs

brak

.Outputs

[string]

.Notes

NAZWA: *Get-WindowsEdition.ps1*

AUTOR: *Ed Wilson*

DATA EDYCJI: *20.9.2013*

WERSJA: *1.2.0 Dodano znaczniki pomocy*

1.1.1 4/2/1009 Dodano odnośnik do strony <http://www.ScriptingGuys.com>

1.1.0 4/1/2009 Zastosowano wyrażenie regularne

```

SŁOWA KLUCZOWE: najlepsze zwyczaje w pracy z konsolą Windows PowerShell
.Link
Http://www.ScriptingGuys.com
#Requires -Version 4.0
#>

$strPattern = "version"
$text = net config workstation

switch -regex ($text)
{
    $strPattern { Write-Host $switch.current }
}

```

Programy do kontroli wersji

Najłatwiej jest prowadzić kontrolę wersji skryptów Windows PowerShell przy użyciu specjalnego programu. W starszych wersjach środowiska Microsoft Visual Studio dostępny był pakiet o nazwie Visual SourceSafe (VSS). Ale do zastosowań skryptowych był on zbyt skomplikowany. Poza tym nie ma go już w nowszych wersjach Microsoft Visual Studio i nie da się go opcjonalnie zainstalować.

Istnieje wiele innych programów do kontroli wersji, chociaż większość z nich to produkty komercyjne słabo nadające się dla firmowych skrypciarzy. Ewentualnie można utworzyć repozytorium skryptów na serwerze Microsoft SharePoint Server, który też ma funkcje ewidencjonowania i wersjonowania. Ale trzeba go zmodyfikować, aby móc przechowywać skrypty PowerShell i VBScript. Najlepsze rozwiązanie powinno łączyć w jednym edytor i automatyczną kontrolę wersji.

Zapiski praktyka

Obsługa programów do kontroli wersji

Alexander Riedel, wiceprezes

SAPIEN Technologies

Od kiedy firma Microsoft wprowadziła narzędzie Script Encoder dla Hosta skryptów systemu Windows (ang. *Windows Script Host* — WSH), a firma SAPIEN Technologies dodała w środowisku PrimalScript możliwość pakowania skryptów w wykonywalne pliki, na naszych forach zaczęło powtarzać się jedno pytanie: „Jak mogę pobrać mój skrypt?”.

Choć bywa, że przyczyną powstawania takich nieprzyjemnych dla użytkownika sytuacji jest wypadkowa braku kopii zapasowej i awarii dysku twardego, eksplozji akumulatora w laptopie lub nieuwagi nastolatka, który zainfekował wirusem komputer tatusia, najczęściej spotyka się jednak następujące historie:

- Poprzedni pracownik na tym stanowisku napisał skrypt, a ja nie wiem, gdzie znajduje się oryginał.
- Kiedyś działał, a teraz nie chce. Działa tylko wersja znajdująca się w pliku *.exe*.
- Ktoś go zmienił i nie wiem, co dokładnie zrobił.

Oczywiście w takich przypadkach zwykła kopia zapasowa na niewiele się zda, bo trzeba sprawdzić, co dokładnie się zmieniło. Dla programisty żadna z wymienionych sytuacji nie jest wielkim problemem. W branży powstało wiele narzędzi i wypracowano wiele dobrych praktyk, dzięki którym nawet jeśli coś takiego się zdarzy, to nie ma paniki. Ale ponieważ programiści skryptów bardzo często nie uważają się za „programistów”, łamią też wiele zasad dobrego programowania.

Wiadomo z doświadczenia, że nawet najlepsze intencje czasami nie pomogą. Odpowiedz sobie na przykład na następujące pytanie: „Kiedy ostatni raz **robiłeś** kopię zapasową?”.

Firma SAPIEN dostrzegła te problemy i w odpowiedzi na nie wyprodukowała program o nawie VersionRecall. Jest to produkt dla samotnie pracujących administratorów, którzy potrzebują systemu wykonywania kopii zapasowych i kontroli wersji.

Program ten nie wymaga konfigurowania, jak typowe systemy kontroli wersji, dzięki czemu można go zainstalować i zacząć używać w ciągu paru minut.

Bardzo ważną cechą programu VersionRecall jest to, że nie potrzebuje do działania żadnego konkretnego interfejsu API, który wiązałby go z jednym środowiskiem IDE. Jest to bardzo ważne, ponieważ typowy administrator każdego dnia korzysta z wielu różnych narzędzi, np. Windows PowerShell, Windows PowerShell ISE, PowerShell Studio, Notatnika itd.

Zmiany można zatwierdzać ręcznie w głównej aplikacji albo za pomocą narzędzia wiersza poleceń. Ale najczęściej robi to automatyczna usługa programu VersionRecall, która wykrywa zmienione pliki w określonym folderze i zatwierdza je bez względu na to, za pomocą jakiego programu dokonano tych zmian.

W powyższym akapicie bardzo ważne jest słowo „automatyczny”. Wystarczy tak skonfigurować środowisko, aby automatycznie robiło kopie zapasowe i śledziło zmiany w najważniejszych plikach, a nie będzie trzeba nikogo prosić o pomoc z paniką w głosie.

Dodatkowe źródła informacji

- Centrum skryptów portalu TechNet pod adresem <http://www.microsoft.com/technet/scriptcenter> zawiera wiele przykładów skryptów logowania i pomocy technicznej.
- Wszystkie skrypty opisane w tym rozdziale są dostępne do pobrania w repozytorium centrum skryptów portalu TechNet pod adresem <http://gallery.technet.microsoft.com/scriptcenter/PowerShell-40-Best-d9e16039>.

Rozdział 18

Rejestrowanie wyników

- Zapisywanie danych w pliku tekstowym
- Zapisywanie danych w dzienniku zdarzeń
- Zapisywanie danych w rejestrze
- Dodatkowe źródła informacji

Po napisaniu, wdrożeniu i wykonaniu skryptów dobrze by było sprawdzić, czy ich wykonywanie przebiegło zgodnie z planem. Najlepiej w tym celu zarejestrować wyniki. Sposobów rejestracji wyników jest kilka i w tym rozdziale przyjrzymy się im dokładniej.

Rejestrowanie wyników skryptów to podstawowa i często wykonywana czynność. Podczas gdy sposobów przechowywania danych jest wiele, od bazy danych po tworzenie stron internetowych, trzy techniki są stosowane tak często, że każdy administrator obowiązkowo powinien je znać. Są one tak ważne, że w Windows PowerShell 4.0 dodano nawet specjalne polecenia do ich obsługi. Trzy najważniejsze metody rejestrowania wyników skryptów to zapis danych w pliku tekstowym, dzienniku zdarzeń oraz rejestrze. W rozdziale tym opisuję najlepsze praktyki przy korzystaniu z tych opcji.

Zapisywanie danych w pliku tekstowym

Mimo zalet dokumentów XML, HTML, Microsoft Office i innych sposobów przechowywania danych pliki tekstowe wciąż są często wybieraną opcją przechowywania wyników skryptów. Wśród ich zalet można wymienić łatwość w użyciu, kompaktowość, przenośność oraz brak problemów ze zgodnością. Najłatwiejszym sposobem na zapisanie informacji w pliku tekstowym jest użycie jednego z dwóch operatorów przekierowania — pojedynczego i podwójnego. Pojedynczy operator przekierowania zapisuje dane w pliku tekstowym. Jeśli plik ten nie istnieje, to go tworzy, a jeśli istnieje, to go nadpisuje.

```
PS C:\> Get-Process > C:\data\FS0\process.txt
```

Podwójny operator przekierowania tworzy plik, jeśli ten nie istnieje. Ale jeśli wyznaczony plik istnieje, to go nie nadpisuje, tylko dodaje dane na końcu.

```
PS C:\> Get-Process >> C:\data\FS0\process.txt
```

Projektowanie metody rejestrowania wyników skryptu

Jedną z decyzji, jakie trzeba podjąć przy projektowaniu mechanizmu rejestracji danych w pliku tekstowym, jest to, czy nadpisywać stary plik, czy dołączać nowe informacje na jego końcu. Decyzję należy podjąć po dokładnym przeanalizowaniu jej ewentualnych skutków. Pomocna w tym może być tabela 18.1.

TABELA 18.1. Przewodnik pomocny w podjęciu decyzji na temat sposobu rejestrowania wyników

| Tryb | Potrzeba | Przykład |
|--------------|---|---|
| Dołączanie | Przechowywanie historii | Rejestrowanie danych zwracanych przez skrypt dokonujący wielu zmian |
| Dołączanie | Przechowywanie danych do kontroli | Skrypt logowania zapisujący godziny logowania użytkownika |
| Dołączanie | Przechowywanie danych do śledzenia | Skrypt zapisujący w pliku informacje o błędach każdej wykonywanej operacji |
| Nadpisywanie | Zapisywanie kodu zwrotnego | Skrypt zapisujący w pliku kod zwrotny powodzenia lub błędu |
| Nadpisywanie | Wyświetlenie informacji, które nie mieszczą się w oknie konsoli Windows PowerShell | Skrypt wyświetlający składowe obiektu |
| Nadpisywanie | Wyświetlenie informacji, które użytkownik może chcieć przeszukiwać lub które trzeba przewijać | Skrypt wyświetlający szczegółowe informacje, które użytkownik może chcieć wyświetlić w Notatniku, aby je przeszukać |

Nadpisywanie dziennika

Jeśli chcemy tylko wiedzieć, czy wykonanie skryptu powiodło się, czy nie, to możemy za każdym razem nadpisywać zawartość dziennika. Takie jednorazowe informacje są przydatne podczas diagnostyki, w której dane przekrojowe nie są istotne, a dziennik konserwacji jest niepotrzebny.

Typowym zastosowaniem dla jednorazowego dziennika jest skrypt logowania. Gdy użytkownik zaloguje się do systemu, nie ma sensu przechowywać informującego o tym dziennika. Ale jeśli użytkownik ma problemy z systemem i nie może wydrukować czegoś za pomocą drukarki sieciowej albo pobrać plików z udziału sieciowego, dziennik staje się bardzo ważnym przyrządem diagnostycznym.

Przykładem skryptu logowania z wbudowaną rejestracją wyników jest plik *LogonScriptWithLogging.ps1*. Jego pierwszą czynnością jest wyłączenie za pomocą odpowiedniego ustawienia zmiennej `$errorActionPreference` służącej do wyświetlania błędów w konsoli Windows PowerShell. Ukrywanie błędów przed użytkownikiem podczas logowania jest dobrym zwyczajem, ponieważ unika się w ten sposób denerwowania użytkownika i otrzymywania przez pomoc techniczną niepotrzebnych telefonów. Następnie skrypt kasuje zawartość obiektu błędów, aby było wiadomo, że wszystkie ewentualne znajdujące się w nim informacje dotyczą tylko bieżącego

skryptu logowania. Następnie kilka zmiennych zostaje ustawionych na wartość `null` dla pewności, że nie przyjmą jakichś niepożądanych wartości ze środowiska.

```
$ErrorActionPreference = "SilentlyContinue"
$error.Clear()
$startTime, $endTime, $message, $logResults = $null
```

Za pomocą polecenia `Test-Path` skrypt sprawdza, czy folder dzienników istnieje. Jeśli nie, tworzy go za pomocą polecenia `New-Item`. Kompletna ścieżka do pliku dziennika tworzona jest przez polecenie `Join-Path`.

```
$logDir = "c:\fso"
if(-not(Test-Path -path $logdir))
{ New-Item -Path $logdir -ItemType directory | Out-Null }
$logonLog = Join-Path -Path $logDir -ChildPath "logonlog.txt"
```

Ważnym elementem każdego dziennika jest znacznik czasu informujący o tym, kiedy dana operacja została wykonana. Zasadniczo najlepiej jest rejestrować czas rozpoczęcia i zakończenia wykonywania skryptu, aby wiedzieć, jak długo to trwało. Jeśli skrypt, który normalnie działał trzy sekundy, nagle zajmie 35 sekund, można podejrzewać, że wystąpił jakiś problem. Skrypt *LogonScriptWithLogging.ps1* mapuje dyski sieciowe i domyślną drukarkę za pomocą obiektu `WshNetwork`. Po zakończeniu każdej operacji skrypt zapisuje w zmiennej `$message` tę operację i wszystkie napotkane błędy.

```
$startTime = (Get-Date).tostring()
$WshNetwork = New-Object -ComObject wscript.network
$WshNetwork.MapNetworkDrive("f:", "\\berlin\studentShare")
$message += "'r'nMapowanie dysku f na \\berlin\student share 'r'n$($error[0])"
$WshNetwork.SetDefaultPrinter("berlinPrinter")
$message += "'r'nUstawianie domyślnej drukarki na berlinPrinter 'r'n$($error[0])"
```

Gdy skrypt wykona wszystkie swoje zadania, pobiera za pomocą polecenia `Get-Date` czas i formatuje wiadomość wyjściową. Jako że wszystkie błędy, operacje i znaczniki czasu są zapisane w zmiennych, utworzenie wiadomości wyjściowej jest bardzo łatwe. Gromadzenie części wyniku w zmiennych jest bardzo dobrym sposobem, ponieważ pozwala na utworzenie dziennika za pomocą pojedynczej operacji wejścia-wyjścia. Jest to o wiele bardziej efektywne rozwiązanie niż wielokrotny zapis w dzienniku podczas działania skryptu. Do utworzenia raportu z działania skryptu użyto łańcucha miejscowego, a do zapisu danych w pliku — pojedynczego operatora przekierowania.

```
$endTime = (Get-Date).tostring()
$logResults = @"
**Rozpoczęcie wykonywania skryptu: $($MyInvocation.InvocationName) $startTime.
$message
**Koniec skryptu logowania: $endTime.
**Czas wykonywania skryptu w sekundach: $((New-TimeSpan -Start $startTime '
-End $endTime).totalSeconds).
"@
$logResults > $logonLog
```

Poniżej znajduje się kompletny kod źródłowy skryptu *LogonScriptWithLogging.ps1*:

LogonScriptWithLogging.ps1

```
$ErrorActionPreference = "SilentlyContinue"
$error.Clear()
$startTime = $endTime = $message = $logResults = $null
```

```

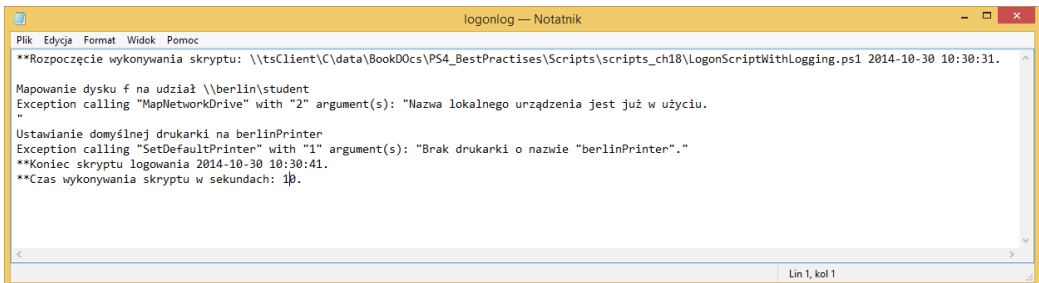
$logDir = "c:\fso"
if(-not(Test-Path -path $logdir))
{ New-Item -Path $logdir -ItemType directory | Out-Null }
$logonLog = Join-Path -Path $logDir -ChildPath "logonlog.txt"

$startTime = (Get-Date).tostring()
$WshNetwork = New-Object -ComObject wscript.network
$WshNetwork.MapNetworkDrive("f:", "\\berlin\studentShare")
$message += "r'nMapowanie dysku f na udział \\berlin\student 'r'n$(Error[0])"
$WshNetwork.SetDefaultPrinter("berlinPrinter")
$message += "r'nUstawianie domyślnej drukarki na berlinPrinter 'r'n$(Error[0])"

$endTime = (Get-Date).tostring()
$logResults = @"
**Rozpoczęcie wykonywania skryptu: $($MyInvocation.InvocationName) $startTime.
$message
**Koniec skryptu logowania $endTime.
**Czas wykonywania skryptu w sekundach: $((New-TimeSpan -Start $startTime '
-End $endTime).totalSeconds).
"@
$logResults > $logonLog

```

Skrypt ten tworzy w folderze C:\fso dziennik podobny do widocznego na rysunku 18.1.



RYSUNEK 18.1. Dziennik zawierający informacje o czasie logowania i statusie operacji

Zapiski praktyka

Wykorzystanie plików CSV

Mike Pfeiffer, Premier Field Engineer

Microsoft Corporation

Jedną z największych zalet konsoli Windows PowerShell jest łatwość tworzenia szczegółowych raportów. Do moich ulubionych poleceń od zawsze zaliczają się polecenia `Export-Csv` i `ConvertTo-Html`, których używam praktycznie cały czas. Dzięki obiektowej budowie konsoli i przy użyciu wymienionych poleceń nawet początkujący użytkownik może bez trudu wyeksportować strukturalne dane do plików zewnętrznych. Ale czasami trzeba coś dostroić, aby otrzymać dokładnie takie informacje, jakich się potrzebuje.

Wyjaśnię teraz, jak rozwiązać typowy problem dotyczący generowania raportów. Przypuśćmy, że chcemy wyeksportować listę usług do pliku CSV, ale interesują nas tylko usługi mające jakieś zależności. Pracę zaczniemy od napisania poniższego polecenia:

```
Get-Service |
  Where-Object DependentServices |
    Select-Object DisplayName,DependentServices
```

W konsoli Windows PowerShell wygląda to całkiem dobrze. Dla każdej usługi zgodnie z oczekiwaniami są dwie kolumny. Oczywiście dodamy jeszcze jeden potok, aby wyeksportować raport do pliku CSV.

```
Get-Service |
  Where-Object DependentServices |
    Select-Object DisplayName,DependentServices |
      Export-Csv c:\myservices.csv -NoTypeInfoation
```

Teraz otwieramy otrzymany plik CSV w Excelu, dodajemy ozdoby komórek w rodzaju kolorowego obramowania itp. i możemy zapisać plik jako arkusz kalkulacyjny. Szef i współpracownicy będą z nas dumni. Ale jeszcze się nie ciesz. Kolumna *DependentServices* nie wygląda tak, jak powinna.

| DisplayName | DependentServices |
|---|---|
| Konstruktor punktów końcowych audio systemu Windows | System.ServiceProcess.ServiceController[] |
| Podstawowy aparat filtrowania | System.ServiceProcess.ServiceController[] |
| Usługi kryptograficzne | System.ServiceProcess.ServiceController[] |

W kolumnie *DependentServices* zamiast wartości znajdują się tylko nazwy typów .NET własności. Przyczyną jest to, że wartość tej kolumny stanowi kolekcja usług zależnych od usługi nadrzędnej.

Natomiast polecenie `Export-CSV` pobiera „płaskie” wartości własności. Można je porównać do zmiennej `Path` systemu Windows, która jest pojedynczym elementem, ale każda zapisana w niej ścieżka jest oddzielona od poprzednich średnikiem. To samo trzeba zrobić z naszym raportem. Wartości kolumny *DependentServices* powinny być pojedynczym łańcuchem złożonym z wartości oddzielanych średnikami. Łatwo to osiągnąć przy użyciu własnej własności, która umożliwi nam uporządkowanie wartości własności za pomocą własnego wyrażenia. Poniżej znajduje się rozwiązanie:

```
Get-Service |
  Where-Object DependentServices |
    Select-Object DisplayName,@{n='DependentServices';e={$ _DependentServices -join ';' }} |
      Export-Csv c:\myservices.csv -NoTypeInfoation
```

Jedyna różnica w porównaniu z poprzednim kodem polega na dostosowaniu własności *DependentServices*. Zwróć uwagę, że wewnątrz wyrażenia użyto operatora `-Join`, którego zadaniem jest łączenie łańcuchów. W wyrażeniu mówimy, że wszystkie obiekty usług z własności *DependentServices* powinny zostać połączone. Na szczęście na każdej zależnej usłudze automatycznie zostanie wywołana metoda `.ToString()` i cała lista zostanie połączona w jeden łańcuch wartości oddzielanych średnikami.

Jest to klasyczny problem, z którym boryka się większość twórców skryptów. W tym tygodniu byłem o to pytany dwa razy, więc postanowiłem napisać tę wskazówkę. Zapamiętaj tę technikę, ponieważ na pewno wcześniej czy później natkniesz się na ten problem.

Dopisywanie danych na końcu dziennika

Jeśli ze względu bezpieczeństwa lub do celów diagnostycznych potrzebne są też stare dane z dziennika, nowe informacje należy dopisywać na końcu. Opcja ta jest też przydatna, gdy skrypt wykonuje kilka różnych działań, których wyniki chcemy zarejestrować. Dopisywanie danych może mieć duże znaczenie, gdy istnieje ryzyko wystąpienia awarii w środku wykonywania operacji.

Używanie dzienników do rozwiązywania problemów

Kiedy pisałem 200 skryptów do książki *Windows 7 Resource Kit*, do każdego z nich musiałem napisać dokumentację opisującą ogólne przeznaczenie i parametry wraz z przykładami składni. Jako że do każdego skryptu pisałem pomoc, napisałem skrypt tworzący listę wszystkich skryptów i dla każdego z nich wywołałem polecenie `Get-Help`. Następnie użyłem podwójnego operatora przekierowania, aby dołączyć dane do pliku *Windows7_Script_Documentation.txt*. Gdy zajrzałem do tego pliku, zauważyłem, że ostatnia pozycja nie dotyczy skryptu z ostatniego rozdziału książki. Zaciekawilem się, dlaczego kolejność skryptów jest przemieszana. Poza tym dostrzegłem brak dokumentacji niektórych skryptów. Obejrzałem ostatni skrypt na liście i nie znalazłem w nim żadnych błędów. Ale kiedy otworzyłem następny skrypt w folderze, zauważyłem, że jego dokumentacji nie ma w pliku tekstowym. Gdy wywołałem polecenie `Get-Help` na tym skrypcie, okazało się, że zostały zwrócone błędy. Po ich usunięciu i ponownym wygenerowaniu dokumentacji wszystko było w porządku. Dołączając na końcu pliku tekstowego wyniki wielu operacji, utworzyłem nie tylko dokumentację, ale również narzędzie diagnostyczne.

Jeśli skrypt jest skomplikowany, można wyposażyć go w mechanizm dostarczania szczegółowych informacji do dziennika. Przykładem takiego skryptu jest plik *LogChartProcessWorkingSet.ps1*. W sekcji `Param` utworzony został przełącznik o nazwie `trace`. Gdy skrypt zostanie uruchomiony z tym przełącznikiem, zapisuje instrukcje w pliku *Tracelog.txt* w katalogu `C:\fso`. Zmienna `$errorActionPreference` jest ustawiona na `SilentlyContinue` oraz kasowana jest zawartość obiektu błędów. Zmienne `$startTime` i `$endTime` są ustawione na `null`. Poniżej znajduje się sekcja inicjacyjna opisywanego skryptu:

```
Param([switch]$trace)
$trace=$true
$errorActionPreference = "SilentlyContinue"
$error.Clear()
$startTime = $endTime = $null
```

Skrypt sprawdza, czy folder dziennika istnieje i w razie potrzeby go tworzy. Następnie tworzy ścieżkę do pliku *Tracelog.txt* i rejestruje czas rozpoczęcia wykonywania skryptu w postaci łańcucha w zmiennej `$startTime`.

```
$logDir = "c:\fso"
if(-not(Test-Path -path $logdir))
{ New-Item -Path $logdir -ItemType directory | Out-Null }
$traceLog = Join-Path -Path $logDir -ChildPath "Tracelog.txt"
$startTime = (Get-Date).tostring()
```


Skrypt szuka zmiennej \$trace. Jeśli ją znajdzie, zapisuje dane w pliku *Tracelog.txt*. Jeśli nie znajdzie zmiennej \$trace, nic nie rejestruje.

```
If($trace)
{ "***Początek wykonywania skryptu: $($MyInvocation.InvocationName) $startTime" >> $traceLog }
```

Gdy skrypt kończy tworzenie wykresu, zapisuje w dzienniku liczbę wygenerowanych błędów. Potem za pomocą instrukcji Foreach przegląda kolekcję błędów i zapisuje w dzienniku każdy błąd.

```
*** Liczba błędów: $($error.count) ***" >> $traceLog
Foreach ($e in $error) { $e >> $tracelog }
```

UWAGA

Pamiętaj, że obiekt COM MSGraph.Application domyślnie nie istnieje w stacji roboczej. Jest instalowany wraz z pakietem Microsoft Office.

Poniżej znajduje się kompletny kod źródłowy skryptu *LogChartProcessWorkingSet.ps1*:

LogChartProcessWorkingSet.ps1

```
Param([switch]$trace = $true)
$errorActionPreference = "SilentlyContinue"
$error.Clear()
$startTime = $endTime = $null

$logDir = "c:\fso"
if(-not(Test-Path -path $logdir))
{ New-Item -Path $logdir -ItemType directory | Out-Null }
$traceLog = Join-Path -Path $logDir -ChildPath "Tracelog.txt"
$startTime = (Get-Date).tostring()

If($trace)
{ "***Początek wykonywania skryptu: $($MyInvocation.InvocationName) $startTime" >> $traceLog }
If($trace)
{ "Tworzenie obiektu msgraph.application" >> $traceLog }
$chart = New-Object -ComObject msgraph.application
$chart.visible = $true
If($trace)
{ "Dodawanie podpisów kolumn wykresu" >> $traceLog }
$chart.datasheet.cells.item(1,1) = "Nazwa procesu"
$chart.datasheet.cells.item(1,2) = "Zbiór roboczy"
If($trace)
{ "Dodawanie danych do wykresu" >> $traceLog }
$r = 2
If($trace)
{ "Pobieranie informacji procesu" >> $traceLog }

Get-Process |
Foreach-Object {
    $chart.datasheet.cells.item($r,1) = $_.name
    $chart.datasheet.cells.item($r,2) = $_.workingSet
    $r++
} #end foreach process

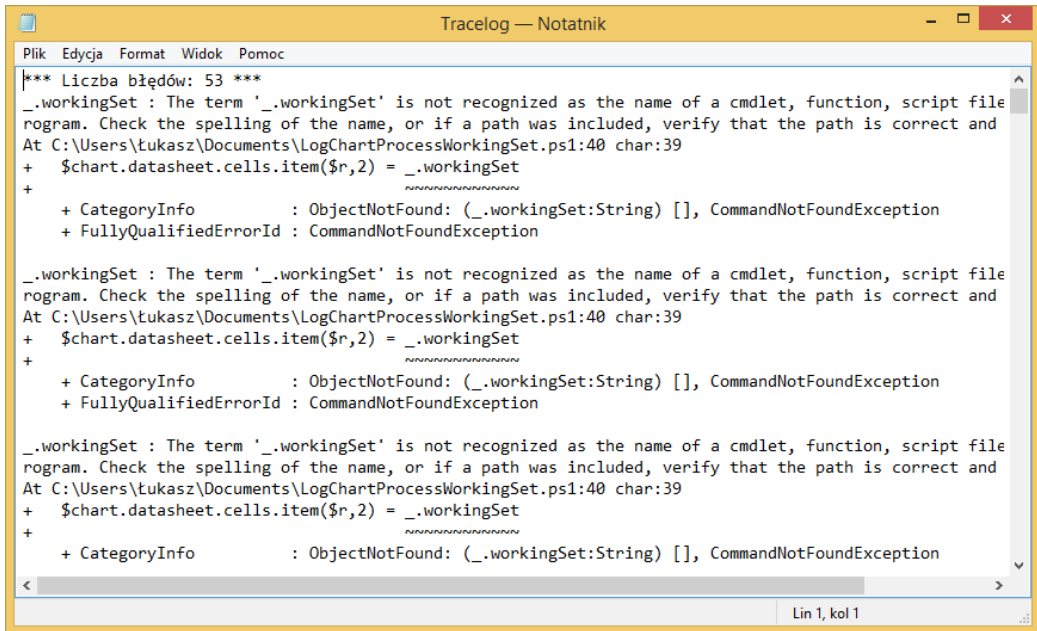
$endTime = (Get-Date).tostring()
```

```

If($trace)
{
    "***Koniec wykonywania skryptu: $endTime. " >> $traceLog}
If($trace)
{
    "***Czas działania skryptu w sekundach: $((New-TimeSpan -Start $startTime '
-End $endTime).totalSeconds) 'r'n" >> $traceLog}
"*** Liczba błędów: $($error.count) ***" >> $traceLog
Foreach ($e in $error) { $e >> $traceLog }

```

Skrypt ten sporządza raport podobny do pokazanego na rysunku 18.2.



RYSUnek 18.2. Dziennik śledzenia utworzony dzięki uruchomieniu skryptu z przełącznikiem trace

Użycie polecenia Out-File

Oprócz operatorów przekierowania do tworzenia plików tekstowych można też używać polecenia `Out-File`. Obie metody są dobre i działają podobnie, ponieważ tak naprawdę konsola Windows PowerShell wywołania operatorów przekierowania mapuje właśnie na polecenie `Out-File`. Różnica między nimi polega na tym, że polecenie `Out-File` ma opcje konfiguracyjne, a operatory przekierowania nie. Domyślne ustawienia tych operatorów są następujące:

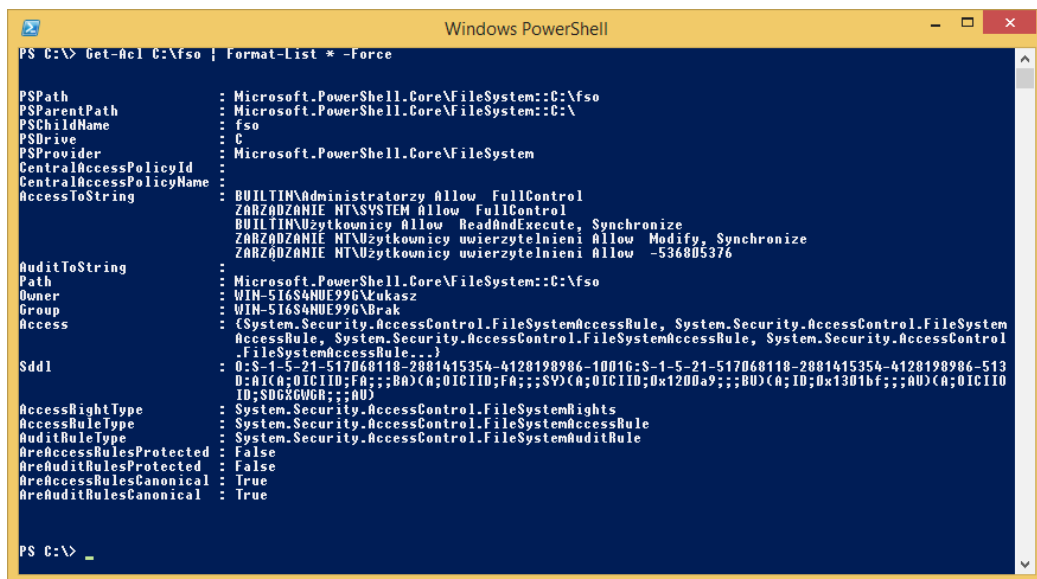
- używają znaków Unicode,
- przy zapisie danych w pliku używają wymiarów konsoli Windows PowerShell.

Zarówno operatory przekierowania, jak i polecenie `Out-File` wysyłają dane do plików przez formater Windows PowerShell. Formater ten może czasami coś dodać lub zmienić w taki sposób, że może to spowodować uszkodzenie pewnych binarnych typów danych.

Aby zmienić kodowanie pliku wyjściowego, można użyć parametru `Encoding`.

```
PS C:\> (Get-Acl -Path C:\fso\access.txt).AccessToString |
Out-File -FilePath C:\fso\outFile.Txt -Encoding ASCII
```

Jeśli dane w konsoli Windows PowerShell są obcięte, jak widać na rysunku 18.3, to można je zapisać w pliku tekstowym i za pomocą parametru `width` spowodować zapisanie wszystkich informacji.



```
Windows PowerShell

PS C:\> Get-Acl C:\fso | Format-List * -force

PSPath                : Microsoft.PowerShell.Core\FileSystem::C:\fso
PSParentPath          : Microsoft.PowerShell.Core\FileSystem::C:\
PSChildName           : fso
PSDrive               : C
PSProvider            : Microsoft.PowerShell.Core\FileSystem
CentralAccessPolicyId  :
CentralAccessPolicyName :
AccessToString        : BUILTIN\Administrators Allow FullControl
                        ZARZĄDZANIE NT\SYSTEM Allow FullControl
                        BUILTIN\Użytkownicy Allow ReadAndExecute, Synchronize
                        ZARZĄDZANIE NT\Użytkownicy wierzytelnieni Allow Modify, Synchronize
                        ZARZĄDZANIE NT\Użytkownicy wierzytelnieni Allow -536805376
AuditToString         :
Path                  : Microsoft.PowerShell.Core\FileSystem::C:\fso
Owner                 : WIN-51684HUE996\Łukasz
Group                 : WIN-51684HUE996\Brak
Access                : (System.Security.AccessControl.FileSystemAccessRule, System.Security.AccessControl.FileSystem
                        AccessRule, System.Security.AccessControl.FileSystemAccessRule, System.Security.AccessControl
                        FileSystemAccessRule...)
Sddl                  : O:S-1-5-21-517068110-2001415354-4120190906-10016;S-1-5-21-517068110-2001415354-4120190906-513
                        0;A(A;0IC11D;FA;;;BA)(A;0IC11D;FA;;;SY)(A;0IC11D;0x1200a9;;;BU)(A;ID;0x1301bf;;;AU)(A;0IC11D
                        ID;SDG;GWB;G;;AU)
AccessRightType       : System.Security.AccessControl.FileSystemRights
AccessRuleType        : System.Security.AccessControl.FileSystemAccessRule
AuditRuleType        : System.Security.AccessControl.FileSystemAuditRule
AreAccessRulesProtected : False
AreAuditRulesProtected : False
AreAccessRulesCanonical : True
AreAuditRulesCanonical : True

PS C:\> _
```

RYSUNEK 18.3. Obcięcie części wyniku w konsoli Windows PowerShell (na rysunku w sekcji `Access`) jest oznaczane trzema kropkami

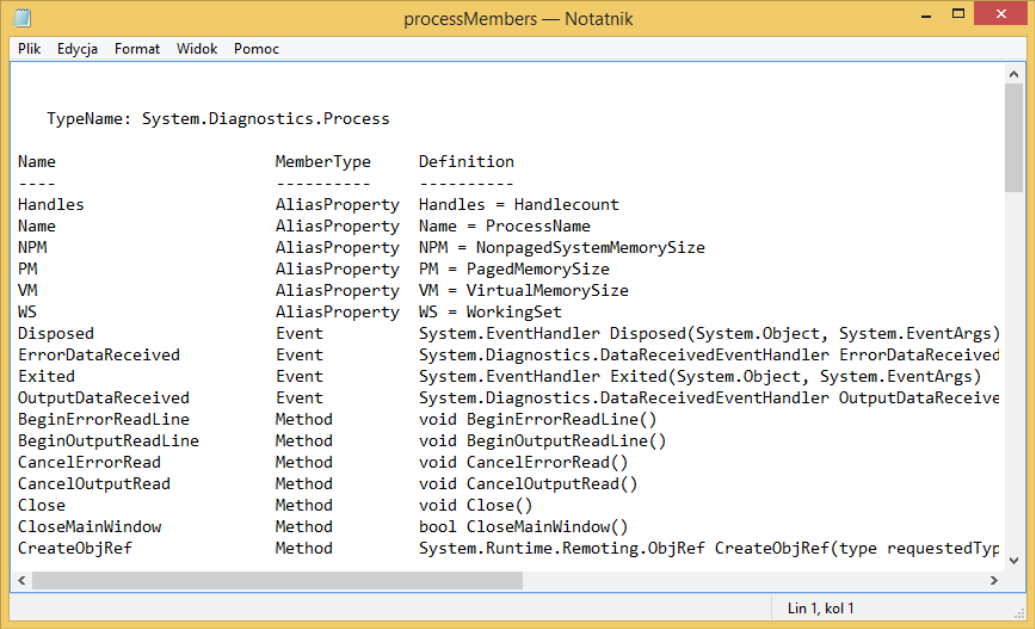
Poniżej znajduje się przykładowe polecenie zmieniające szerokość danych w pliku tekstowym za pomocą parametru `width`:

```
PS C:\> Get-Acl -Path C:\fso\access.txt |
Out-File -FilePath C:\fso\outFile.Txt -Encoding ASCII -Width 1500
```

Jeśli polecenia w konsoli lub skrypcie Windows PowerShell oddzieli się średnikiem, to można wywołać Notatnik w momencie tworzenia pliku. Potem w programie tym można wyświetlić wszystkie informacje i wygodnie je przeglądać.

Przekazując wyniki do polecenia `Get-Member`, można wyświetlić składowe obiektów. Treść ta jest za szeroka do wyświetlenia w oknie konsoli Windows PowerShell, więc zostaje obcięta. Ale wystarczy określić szerokość 200 w poleceniu `Out-File`, aby pozbyć się tego problemu w pliku.

```
PS C:\> Get-Process |
Get-Member |
Out-File -FilePath C:\fso\processMembers.txt -Width 200 ;
notepad C:\fso\processMembers.txt
```



The screenshot shows a Notepad window with the title 'processMembers — Notatnik'. The menu bar includes 'Plik', 'Edycja', 'Format', 'Widok', and 'Pomoc'. The text content is as follows:

```
TypeName: System.Diagnostics.Process
```

| Name | MemberType | Definition |
|---------------------|---------------|---|
| Handles | AliasProperty | Handles = Handlecount |
| Name | AliasProperty | Name = ProcessName |
| NPM | AliasProperty | NPM = NonpagedSystemMemorySize |
| PM | AliasProperty | PM = PagedMemorySize |
| VM | AliasProperty | VM = VirtualMemorySize |
| WS | AliasProperty | WS = WorkingSet |
| Disposed | Event | System.EventHandler Disposed(System.Object, System.EventArgs) |
| ErrorDataReceived | Event | System.Diagnostics.DataReceivedEventHandler ErrorDataReceived |
| Exited | Event | System.EventHandler Exited(System.Object, System.EventArgs) |
| OutputDataReceived | Event | System.Diagnostics.DataReceivedEventHandler OutputDataReceive |
| BeginErrorReadLine | Method | void BeginErrorReadLine() |
| BeginOutputReadLine | Method | void BeginOutputReadLine() |
| CancelErrorRead | Method | void CancelErrorRead() |
| CancelOutputRead | Method | void CancelOutputRead() |
| Close | Method | void Close() |
| CloseMainWindow | Method | bool CloseMainWindow() |
| CreateObjRef | Method | System.Runtime.Remoting.ObjRef CreateObjRef(type requestedType) |

The status bar at the bottom right indicates 'Lin 1, kol 1'.

RYSUNEK 18.4. W Notatniku można wyświetlać długie linijki tekstu

Wiedza tajemna

Rejestrowanie danych ze skryptów

Ian Farr, Premier Field Engineer
Microsoft Corporation, Zjednoczone Królestwo

Dlaczego rejestrowanie wyników skryptów jest ważne? Choćby dlatego, że ułatwia to usuwanie usterek — piszę skrypty używane w przedsiębiorstwach już od paru lat i widziałem zaledwie parę skryptów, które działały idealnie od samego początku. Ale zazwyczaj skrypt wymaga dopracowania na etapie tworzenia, po przekazaniu do produkcji i czasami także wówczas, gdy ktoś inny zacznie z niego korzystać. Aby efektywnie rozwiązywać problemy, trzeba wiedzieć, co dokładnie robi skrypt, a w zdobywaniu tych informacji bardzo pomocna jest rejestracja danych w dzienniku. Jest oczywiste, że łatwiej naprawić skrypt, który zapisuje ważne dane w pliku tekstowym. A nawet jeśli nie występują żadne problemy, z dziennika można dowiedzieć się, jakie udane zmiany zostały wprowadzone, i utworzyć cenny dziennik inspekcji.

Jakie informacje powinno się zapisywać w dzienniku? To zależy od tego, co dany skrypt robi. Na początek zalecam zapisać następujące dane: nazwa użytkownika, nazwa komputera, czas rozpoczęcia działania skryptu, czas zakończenia działania skryptu, wykonane zadania oraz błędy. W jakim formacie zapisać te informacje? Kolejną wielką zaletą konsoli Windows PowerShell

jest to, że za jej pomocą można łatwo zapisywać dane w plikach tekstowych, CSV, HTML oraz XML, chociaż to jeszcze nie wszystkie opcje. Mój kolega utworzył centralną bazę danych na dane skryptów, dzięki czemu administratorzy mogą analizować różne dzienniki — użyj wyobraźni.

W jaki sposób konsola Windows PowerShell może mi pomóc? Jak się pewnie spodziewasz, jest wiele narzędzi pomocnych przy rejestrowaniu danych dotyczących działania skryptów, na przykład:

- Wśród przydatnych poleceń do pracy z dziennikami znajdują się: `Add-Content`, `Set-Content`, `Out-File`, `Tee-Object`, `Export-CSV`, `ConvertTo-HTML` oraz `Export-CLIXML`.
- Za pomocą operatorów przekierowania można wysyłać informacje ze strumieni `Success`, `Error`, `Warning`, `Debug` oraz `Verbose` do plików. Można nawet wysyłać do nich własne wiadomości za pomocą poleceń `Write-Error`, `Write-Warning` oraz `Write-Debug` i następnie przekierowywać je do dziennika.
- Jeśli trzeba sprawdzić, czy wykonywanie operacji zakończyło się pomyślnie, niezastąpione są zmienne `$LastExitCode`, `$Error` i `$?` . Weźmy na przykład zmienną `$Error`. Przechowuje ona ostatnie błędy w tablicy obiektów i wybierając własności najnowszego obiektu, można dostosować informacje zapisywane w dzienniku.
- Jeśli podczas pracy lub diagnostyki trzeba szybko coś zarejestrować, można na początku skryptu skasować zawartość zmiennej `$Error`, a następnie wyeksportować jej treść do pliku po zakończeniu działania skryptu. Można zmienić domyślną wartość 256 bufora `$Error` na dowolną inną liczbę za pomocą zmiennej automatycznej `$MaximumErrorCount`.

Na pewno doświadczysz różnych problemów ze swoimi skryptami, więc poświęć chwilę na napisanie funkcji rejestrującej dane w dzienniku — będzie Ci często potrzebna.

Miejsce przechowywania tekstu

Pracując z plikami tekstowymi, należy wybrać miejsce na ich przechowywanie. Problem jest rozwiązany, jeśli mamy przeznaczony do tego folder. Podejście to zastosowano w skrypcie *LogChartProcessWorkingSet.ps1*, który tworzy folder `C:\fso` i zapisuje w nim dzienniki. Jeżeli użytkownik nie ma uprawnień do utworzenia tego folderu lub zapisywania w nim danych, operacja nie powiedzie się. Innym problemem, jaki można napotkać przy tworzeniu specjalnego folderu na dzienniki, jest to, że użytkownik nie zawsze musi być dostępny. Jako że folder docelowy może być czasami niedostępny, zawsze należy sprawdzać, czy istnieje, i tworzyć go, jeżeli nie istnieje. Kolejną trudnością jest to, że w niektórych przypadkach dysk systemowy ma inną literę niż `C`. Dla bezpieczeństwa lepiej więc zawsze sprawdzać, który dysk jest systemowy, i używać wykrytej lokalizacji w skrypcie.

Wszystkich wymienionych problemów można uniknąć, wybierając folder, który zawsze znajduje się w komputerze, a najlepiej taki, do którego użytkownik ma zawsze prawo dostępu. W komputerze tworzonych jest wiele standardowych folderów, w których można przechowywać tworzone przez skrypty pliki dzienników. Ścieżka do folderów specjalnych jest automatycznie rozwiązywana przez system, więc zawsze jest poprawna bez względu na nazwę użytkownika i literę dysku systemowego. Ale pobieranie jej ręcznie jest kłopotliwe i dlatego lepiej do tego celu używać klasy platformy .NET.

W skrypcie *Get-CountryByIP.ps1* polecenie *Tee-Object* wyświetla wynik w oknie konsoli i zapisuje informacje w pliku tekstowym. Skrypt ten za pomocą usługi sieciowej i polecenia *Get-WebServiceProxy* wykrywa kraj na podstawie adresu IP. Skryptu tego można używać nie tylko do zabawy, ale też do automatycznego wykrywania i konfigurowania ustawień dotyczących lokalizacji.

Na początku skryptu *Get-CountryByIP.ps1* znajduje się sekcja znaczników pomocy zawierająca definicję pomocy do użytku z poziomu wiersza poleceń. W sekcji tej znajdują się znaczniki *synopsis*, *description*, *example*, *inputs*, *outputs* oraz *notes*, w których wpisano szczegółowe informacje przydatne dla użytkownika skryptu. Opisywana sekcja skryptu jest pokazana poniżej:

```
<#
.Synopsis
  Sprawdza kraj po adresie IP
.Description
  Skrypt ten pobiera informacje o kraju wg adresu IP. Używa
  usługi sieciowej, więc musi być podłączony do internetu.
.Example
  Get-CountryByIP.ps1 -ip 10.1.1.1, 192.168.1.1 -log iplog.txt
  Zapisuje nazwę kraju w pliku %mydocuments%\iplog.txt i drukuje ją na ekranie.
.Inputs
  [string]
.Outputs
  [PSObject]
.Notes
  NAZWA: Get-CountryByIP.ps1
  AUTOR: Ed Wilson
  WERSJA: 1.0.0
  DATA EDYCJI 20.8.2009
  SŁOWA KLUCZOWE: New-WebServiceProxy, IP, New-Object, PSObject
.Link
  Http://www.ScriptingGuys.com
#requires -version 2.0
#>
```

Na początku skryptu znajduje się wywołanie metody *CmdletBinding()* i zdefiniowano parę parametrów wiersza poleceń. Parametr *ip* jest tablicą łańcuchów i służy do przekazywania adresu IP do sprawdzenia kraju. Parametr *-log* służy do przekazywania nazwy skryptu. Parametr *folder* określa folder specjalny. Poniżej znajduje się opisywana część skryptu:

```
[CmdletBinding()]
Param(
    [Parameter(Mandatory = $true, Position = 0, ValueFromPipeline = $true)]
    [string[]]$ip,
    [string]$log = "ipLogFile.txt",
    [string]$folder = "Personal"
)#end param
```

Najważniejszy kod znajduje się w funkcji *Get-CountryByIP*. Na początku zdefiniowano identyfikator URI wskazujący usługę sieciową WSDL. Następnie za pomocą polecenia *New-WebServiceProxy* tworzony jest pośrednik do tej usługi. Utworzony obiekt zostaje zapisany w zmiennej *\$proxy*. Następnie wywołana zostaje metoda *GetGeoIP* tego obiektu, a jej wynik zostaje zapisany w zmiennej *\$rtn*.

```
Function Get-CountryByIP($IP)
{
    $URI = "http://www.webservices.net/geoip/service.asmx?wsdl"
    $Proxy = New-WebServiceProxy -uri $URI -namespace WebServiceProxy -class IP
    $RTN = $proxy.GetGeoIP($IP)
```

Dla ułatwienia pracy ze zwróconymi danymi za pomocą polecenia `New-Object` tworzony jest egzemplarz klasy `PSObject`. Obiekt ten zostaje zapisany w zmiennej `$ipReturn`. Po utworzeniu egzemplarza klasy `PSObject` polecenie `Add-Member` dodaje do niego adres IP, nazwę kraju i kod kraju. Później obiekt ten zostaje zwrócony do kodu wywołującego.

```
$ipReturn = New-Object PSObject
$ipReturn | Add-Member -MemberType noteproperty -Name ip -Value $rtn.ip
$ipReturn | Add-Member -MemberType noteproperty -Name countryName -Value $rtn.CountryName
$ipReturn | Add-Member -MemberType noteproperty -Name countryCode -Value $rtn.CountryCode
$ipReturn
} #end Get-CountryByIP
```

Folder wyjściowy do przechowywania nowo utworzonego pliku tekstowego jest określany przez statyczną metodę `GetFolderPath` z klasy `.NET Environment`. Metoda ta musi otrzymać wartość wyliczenia `Environment.SpecialFolder`. Ścieżkę do określonego folderu specjalnego zwraca funkcja `Get-Folder`.

```
Function Get-Folder($folderName)
{
    [Environment]::GetFolderPath([environment+SpecialFolder]::$folderName)
} #end function Get-Folder
```

Punkt początkowy tego skryptu przekazuje adres IP zapisany w zmiennej `$ip` do polecenia `Foreach-Object`, w którym funkcja `Get-CountryByIP` pobiera bieżący element z potoku przez parametr `ip`. Następnie zwrócony obiekt `PSObject` zostaje przekazany do polecenia `Tee-Object` i otrzymany obiekt zostaje wyświetlony w konsoli Windows PowerShell oraz zapisany w zmiennej `$results`. Zmienna ta zostaje przekazana do polecenia `Out-File` oraz zostaje utworzona ścieżka do pliku za pomocą polecenia `Join-Path` odbierającego łańcuch zwrócony przez funkcję `Get-Folder`. Ścieżka do folderu specjalnego i nazwa pliku zostają połączone w celu utworzenia ścieżki do pliku.

```
$ip |
Foreach-Object { Get-CountryByIP -ip $_ } |
Tee-Object -Variable results
$results |
Out-File -FilePath '
(Join-Path -Path (Get-Folder -folderName $folder) -childPath $log)
```

Poniżej znajduje się kompletny kod źródłowy skryptu *Get-CountryByIP.ps1*:

Get-CountryByIP.ps1

```
<#
.Synopsis
Sprawdza kraj po adresie IP
.Description
Skrypt ten pobiera informacje o kraju wg adresu IP. Używa
usługi sieciowej, więc musi być podłączony do internetu.
.Example
Get-CountryByIP.ps1 -ip 10.1.1.1, 192.168.1.1 -log iplog.txt
```

```

Zapisuje nazwę kraju w pliku %mydocuments%\iplog.txt i drukuje ją na ekranie.
.Inputs
[string]
.OutPuts
[PSObject]
.Notes
NAZWA: Get-CountryByIP.ps1
AUTOR: Ed Wilson
WERSJA: 1.0.0
DATA EDYCJI 20.8.2009
SŁOWA KLUCZOWE: New-WebServiceProxy, IP, New-Object, PSObject
.Link
Http://www.ScriptingGuys.com
#requires -version 2.0
#>
[CmdletBinding()]
Param(
    [Parameter(Mandatory = $true, Position = 0, ValueFromPipeline = $true)]
    [string[]]$ip,
    [string]$log = "ipLogFile.txt",
    [string]$folder = "Personal"
)#end param

# *** funkcja ***

Function Get-CountryByIP($IP)
{
    $URI = "http://www.webservices.net/geoip/service.asmx?wsdl"
    $Proxy = New-WebServiceProxy -uri $URI -namespace WebServiceProxy -class IP
    $RTN = $proxy.GetGeoIP($IP)

    $ipReturn = New-Object PSObject -Property @{
        'ip' = $rtn.ip;
        'CountryName' = $rtn.countryname;
        'CountryCode'=$rtn.countrycode}

    $ipReturn
} #end Get-CountryByIP

Function Get-Folder($folderName)
{
    [Environment]::GetFolderPath([Environment+SpecialFolder]::$folderName)
} #end funkcja Get-Folder

# *** punkt początkowy skryptu ***

$ip |
ForEach-Object { Get-CountryByIP -ip $_ } |
Tee-Object -Variable results

$results |
Out-File -FilePath '
    (Join-Path -Path (Get-Folder -folderName $folder) -childPath $log)

```


Przechowywanie dzienników w lokalizacjach sieciowych

Czasami wygodnym rozwiązaniem jest przechowywanie dzienników w centralnym współdzielonym folderze zamiast na komputerze lokalnym. Rozwiązuje to wiele opisanych wcześniej problemów dotyczących tworzenia i obsługi folderu na każdym komputerze. Sieciowymi dziennikami można się posługiwać na dwa sposoby. Po pierwsze: można dokonywać zapisu bezpośrednio w dzienniku, a po drugie: można najpierw zapisywać dane w pliku tymczasowym na komputerze lokalnym, a następnie kopiować ten plik do lokalizacji sieciowej. Zgodnie z najlepszymi praktykami każdy duży plik lub taki, który może zawierać dużo informacji, powinien być tworzony lokalnie, a następnie kopiowany do lokalizacji sieciowej. Poniżej analizuję oba te podejścia.

Zapis bezpośrednio w pliku

Najprostszym sposobem pracy z sieciowymi dziennikami jest bezpośrednie zapisywanie w nich informacji. Polecenie `Out-File` przyjmuje ścieżki UNC (ang. *Universal Naming Convention*) oraz ścieżki do zmapowanych dysków sieciowych. Najwygodniejszym rozwiązaniem jest użycie ścieżki UNC, ponieważ nie trzeba tworzyć i obsługiwać dysku sieciowego.

```
Get-Process | Out-File -FilePath \\berlin\netshare\processes.txt
```

Jeśli trzeba zapisywać niewielkie ilości danych, to bezpośredni zapis w dobrze działającej sieci jest odpowiednim rozwiązaniem. Ale jeśli danych jest dużo lub sieć nie jest niezawodna albo ma ograniczoną łączność, należy zastosować inne podejście.

Zapis danych w pliku lokalnym i kopiowanie tego pliku do lokalizacji sieciowej

Jako że tworzenie plików w udziale sieciowym i bezpośrednie zapisywanie w nich danych nie jest optymalnym rozwiązaniem, podczas wykonywania tych operacji można napotkać wiele różnych problemów. Zapis w pliku lokalnym jest bardziej niezawodny, podobnie jak kopiowanie gotowego pliku do udziału sieciowego. Ze względu na różne trudności mogące pojawić się podczas pracy z lokalnymi plikami i folderami najlepiej jest najpierw zapisywać dane w tymczasowym pliku w tymczasowym katalogu, a następnie kopiować ten plik do udziału sieciowego. Nie jest to trudne i znacznie ułatwia pracę z zapisem dzienników w sieci.

Najłatwiejszym sposobem na zapisanie informacji w pliku tymczasowym jest użycie metody `GetTempFileName` z klasy `.NET Io.Path`. Metoda ta tworzy w tymczasowym katalogu użytkownika tymczasowy plik podobny do poniższego:

```
C:\Users\edwilson\AppData\Local\Temp\tmpE7C6.tmp
```

Skrypt `New-TempFile.ps1` stanowi przykład zapisu danych w pliku tymczasowym i wyświetlenia wyników operacji w Notatniku. Skrypt ten zawiera funkcję o nazwie `New-TempFile` z metodą `CmdletBinding`. Funkcja ta tworzy jeden parametr wejściowy przyjmujący tablicę obiektów `PSObject` w zmiennej `$inputObject`. Następnie skrypt wywołuje statyczną metodę `GetTempFileName` z klasy `.NET Io.Path`. Nazwa pliku tymczasowego jest przechowywana w zmiennej `$tmpFile`.

Dane znajdujące się w zmiennej `$InputObject` zostają przekazane do polecenia `Out-File`, a następnie do tymczasowego pliku wskazywanego przez zmienną `$tmpFile`. Potem ścieżka do tego pliku zostaje zwrócona do kodu wywołującego:

```
Function New-TempFile
{
    [CmdletBinding()]
    Param(
        [Parameter(Position=0,ValueFromPipeline=$true)]
        [PSObject[]]$inputObject
    )#end param
    $tmpFile = [Io.Path]::getTempFileName()
    $inputObject | Out-File -filepath $tmpFile
    $tmpFile
} #end funkcja New-TempFile
```

Punkt początkowy tego skryptu stanowi przykład sposobu użycia opisanej funkcji. Wywołujemy funkcję `New-TempFile` i przekazujemy jej wynik do polecenia `Get-Service` przez parametr `inputObject`. Zwrócona ścieżka do pliku zostaje zapisana w zmiennej `$rtn`. Po utworzeniu pliku tymczasowego w funkcji `New-TempFile` plik ten zostaje przeniesiony do udziału plikowego na zdalnym serwerze i przemianowany za pomocą polecenia `Move-Item`.

```
$destination = "\\berlin\fileshare\services.txt"
$rtn = New-TempFile -inputObject (Get-Service)
Move-Item -path $rtn -destination $destination
```

Poniżej znajduje się kompletny kod źródłowy skryptu *New-TempFile.ps1*:

New-TempFile.ps1

```
Function New-TempFile
{
    [CmdletBinding()]
    Param(
        [Parameter(Position=0,ValueFromPipeline=$true)]
        [PSObject[]]$inputObject
    )#end param
    $tmpFile = [Io.Path]::getTempFileName()
    $inputObject | Out-File -filepath $tmpFile
    $tmpFile
} #end funkcja New-TempFile

# *** punkt początkowy skryptu ***
$destination = "\\berlin\fileshare\services.txt"
$rtn = New-TempFile -inputObject (Get-Service)
Move-Item -path $rtn -destination $destination
```

Zapiski praktyka

Tworzenie dzienników w konsoli Windows PowerShell

Andrew Willett, architekt systemów

Unitrans Logistics, Steinhoff Group

Napisałeś skrypt Windows PowerShell, przetestowałeś go na swoim komputerze i wdrożyłeś do sieci, ale coś poszło nie tak. Tylko co?

Rejestrowanie — albo jak lubią mawiać programiści **instrumentacja** — jest niezastąpionym narzędziem do testowania i diagnozowania skryptów oraz kluczowym składnikiem cyklu ich istnienia. W zarejestrowanych informacjach można sprawdzić, dlaczego dany skrypt działa lub nie działa zgodnie z oczekiwaniami, co spowodowało wystąpienie wyjątku albo dowiedzieć się bardziej szczegółowych rzeczy, np. ile czasu zajmuje wykonanie skryptu i dlaczego właśnie tyle.

Implementacja podstawowego mechanizmu rejestracji danych jest prosta i wygląda podobnie do poleceń stosowanych w wierszu poleceń. Zamiast wysyłać dane do pliku tekstowego za pomocą operatora `>`, można używać polecenia `Tee-Object`, które wysyła dane jednocześnie do zmiennej i pliku tekstowego.

```
Get-Service | Tee-Object -filepath c:\services.txt
```

Polecenie `Tee-Object` dobrze się sprawdza w przypadku pojedynczych poleceń, ale dla całych skryptów nie jest najwygodniejszym rozwiązaniem. Następnym logicznym krokiem jest używanie polecenia `Tee-Object` wszędzie, gdzie to konieczne, i dołączanie danych na końcu jednego pliku. Niestety polecenie to może tylko nadpisywać pliki, więc dodano polecenie `Start-Transcript`. W najprostszym przypadku użycie tego polecenia wymaga napisania dwóch wierszy kodu — oznaczającego początek i koniec — odpowiednio włączającego i wyłączającego rejestrowanie danych.

```
Start-Transcript -path c:\scriptoutput.txt
```

```
(...)
```

```
Stop-Transcript
```

Wśród najbardziej przydatnych parametrów polecenia `Start-Transcript` można wymienić parametr `-append` dołączający dane do istniejącego pliku i `NoClobber` wyłączający domyślne ustawienie polegające na nadpisywaniu istniejącego pliku. (To może wyglądać znajomo dla administratorów systemów uniksowych). Wywołanie polecenia `Stop-Transcript` jest niejawne, więc jeśli zapomnisz je wpisać albo skrypt zakończy działanie inną ścieżką lub z powodu wyjątku, to nic złego pod tym względem się nie stanie.

Polecenia te są bardzo pomocne w znajdowaniu przyczyny problemów w skryptach działających w produkcji oraz w debugowaniu skryptów w fazie testowej. Ale przekopywanie się przez gąszcz informacji tylko po to, by dowiedzieć się, czy wykonywanie skryptu zakończyło się pomyślnie, nie jest już tak miłe.

Podczas pracy nad skryptem wiadomo, jakie są potencjalne zagrożenia mogące spowodować jego awarię. Może to być na przykład problem z połączeniem, brak zasobów systemowych lub posiadanie niewystarczających uprawnień. Wiadomo też, jak sprawdzić, czy skrypt pomyślnie wykonał swoje zadanie. Ponadto niektóre parametry skrypt może znajdować lub określać w czasie działania, zamiast pobierać na sztywno wprost z kodu. Dotyczy to na przykład dostępnej przepustowości, sprawdzania poziomu uprawnień użytkownika oraz tego, czy komputer jest podłączony do gniazda zasilającego, czy działa na akumulatorze. Diagnozowanie usterki po jej wystąpieniu może być trudne, jeśli można tylko **zakładać**, jakich parametrów używał skrypt. Możliwość przywołania tych kluczowych informacji, np. dodanych do pełnego dziennika, zaoszczędzi Ci wiele czasu — wiem, że wiele osób przeszukujących dziennik *Windowsupdate.log* marzy o czymś takim.

Postrzeganie instrumentacji jako całego zbioru technologii oznacza uwzględnienie zarówno przechowywania, jak i dostarczania informacji — plik tekstowy zagrzebany w jakimś bliżej nieznanym miejscu na dysku twardym nie jest zbyt pomocny. Jeśli wiadomo, które informacje są istotne, należy podjąć decyzję co do tego, co z nimi zrobić, na podstawie efektów, jakie może spowodować awaria, a także zdecydować, czy trzeba podjąć jakieś działania i kto powinien to zrobić oraz jak bardzo ważny jest czas w zaistniałej sytuacji. Jeżeli dane mają być archiwizowane, należy zastanowić się, gdzie je zapisać — np. w systemie plików, dzienniku zdarzeń lub w sieci — uwzględniając uprawnienia posiadane przez użytkownika i okres przechowywania.

Poniżej opisuję kilka przydatnych sztuczek oraz podaję parę przykładów ich zastosowania na dobry początek:

- Wysyłając dane dziennika na adres e-mail administratora, można go szybko poinformować o zaistniałym problemie. W razie potrzeby można nawet wysłać dane diagnostyczne do programu pomocy technicznej.

```
$to = "helpdesk@contoso.com"
$from = "scripts@contoso.com"
$subject = "Błąd dotyczący uprawnień w skrypcie"
$body = "Nie udało się uruchomić skryptu, ponieważ użytkownik " + (Get-Content
env:username) +
" nie należy do odpowiedniej grupy."
$server = "smtp.contoso.com"
$smtp = New-Object Net.Mail.SmtpClient($server)
$smtp.Send($from, $to, $subject, $body)
```

- Wysyłanie plików tekstowych do sieciowego udziału plikowego jest sposobem na stworzenie centralnej bazy danych diagnostycznych, która może być przydatna na przykład do wyświetlenia listy komputerów i sprawdzenia, kiedy dany skrypt był ostatnio wykonywany. Tworząc nazwy plików z nazwy komputera i daty/godziny z sekundami, można mieć pewność, że będą one niepowtarzalne. Ponadto takie pliki można łatwo posortować w Eksploratorze plików, chociaż to dość toporne rozwiązanie.

```
$path = "\\fileserver\logs\script1\" + (Get-Content env:computername)
+
" " + (Get-Date -f "yyyy-MM-dd HHmmss") + ".txt"
Start-Transcript -path $path
```

- Zapisywanie konkretnych błędów w dzienniku zdarzeń jest doskonałym sposobem na ujawnienie danych instrumentacyjnych, a od tego dziennika wielu techników zaczyna diagnozowanie problemów. Ponadto dziennik zdarzeń można monitorować za pomocą powszechnie dostępnych narzędzi, takich jak Microsoft System Center Operations Manager czy Usługa kolektora zdarzeń.

Szczegółowość rejestrowanych w dzienniku informacji można stopniować i np. uwzględniać tylko komunikaty informacyjne, ostrzeżenia albo powiadomienia o błędach. Można też dodawać szczegółowe kody błędów odnoszące się do przyczyn ich występowania. Jedyny haczyk polega na tym, że do ustawienia własnego źródła dla dziennika zdarzeń potrzebne są uprawnienia administracyjne. Jeśli chcesz używać dziennika zdarzeń w wymienionych sytuacjach, musisz zadbać o to, by został utworzony zawczasu, albo musisz zarekwirować jedno z już istniejących źródeł systemu Windows na swoje potrzeby.

```
$source = "MyScript"
$log = "Application"
$message = "Nie udało się uruchomić skryptu, ponieważ użytkownik " + (Get-Content
env:username) +
" nie należy do odpowiedniej grupy."
$type = "Error"
$id = 1

if (![System.Diagnostics.EventLog]::SourceExists($source)) {
[System.Diagnostics.EventLog]::CreateEventSource($source, $log) }

$eventLog = New-Object System.Diagnostics.EventLog
$eventLog.Log = $log
$eventLog.Source = $source
$eventLog.WriteEntry($message, $type, $id)
```

Zapisywanie danych w dzienniku zdarzeń

W dziennikach zdarzeń systemu Windows można wygodnie przechowywać krótkie informacje dotyczące statusu i diagnostyczne. Źródła zdarzeń, dzienniki zdarzeń i wpisy w dziennikach zdarzeń można tworzyć bezpośrednio za pomocą klas .NET lub przy użyciu poleceń cmdlet. Za pomocą polecenia `New-EventLog` można utworzyć nowy dziennik zdarzeń i źródła dziennika zdarzeń. Aby dodać wpis do dziennika zdarzeń, należy podać zarówno jego nazwę, jak i źródło.

Do utworzenia dziennika i źródła zdarzeń potrzebne są uprawnienia administracyjne. Bez nich zostanie wyświetlony poniższy komunikat o błędzie:

```
PS C:\> New-EventLog -LogName scripting -Source processAudit
New-EventLog : Access is denied. Please try with an elevated user permission.
At line:1 char:13
+ New-EventLog <<<< -LogName scripting -Source processAudit
+ CategoryInfo          : InvalidOperation: (:) [New-EventLog], Exception
+ FullyQualifiedErrorId : AccessIsDenied,Microsoft.PowerShell.Commands.NewEventLogCommand
```

Aby uruchomić konsolę Windows PowerShell lub Windows PowerShell ISE, należy kliknąć prawym przyciskiem myszy ikonę programu i w menu, które się pojawi, kliknąć pozycję *Uruchom jako administrator*. W skrypcie powinno się najpierw sprawdzić uprawnienia za pomocą funkcji

w rodzaju `Test-IsAdministrator`. Funkcja taka znajduje się w skrypcie *TestAdminCreateEventLog.ps1*. Na jej początku znajduje się parę podstawowych informacji pomocniczych, takich jak zwięzły opis i przykład użycia.

```
function Test-IsAdministrator
{
    <#
    .Synopsis
        Sprawdza, czy użytkownik jest administratorem
    .Description
        Zwraca prawdę, jeśli użytkownik jest administratorem
        i fałsz w przeciwnym przypadku.
    .Example
        Test-IsAdministrator
    #>
```

W funkcji tej użyta jest statyczna metoda `GetCurrent` z klasy `.NET Security.Principal.WindowsIdentity`, która zwraca obiekt typu `WindowsIdentity` reprezentujący bieżącego użytkownika. Obiekt ten zostaje przekazany do klasy `System.Principal.WindowsPrincipal` w celu wygenerowania obiektu typu `WindowsPrincipal`. Metoda `IsInRole` pobiera wartość wyliczenia `WindowsBuiltinRole`, aby dowiedzieć się, czy użytkownik ma rolę administracyjną.

```
$currentUser = [Security.Principal.WindowsIdentity]::GetCurrent()
(New-Object Security.Principal.WindowsPrincipal $currentUser).IsInRole '
([Security.Principal.WindowsBuiltinRole]::Administrator)
} #end funkcja Test-IsAdministrator
```

Funkcja `Test-IsAdministrator` zwraca wartość logiczną. Jeśli zwraca prawdę, to znaczy, że użytkownik jest administratorem, a jeśli zwraca fałsz, to znaczy, że użytkownik nie jest administratorem i trzeba zamknąć skrypt. Jeżeli użytkownik jest administratorem, skrypt tworzy nowy dziennik zdarzeń i nowe źródło.

```
If(-not (Test-IsAdministrator)) { "Ten skrypt wymaga do działania uprawnień administratora." ; exit }
New-EventLog -LogName scripting -Source processAudit
```

Poniżej znajduje się kompletny kod źródłowy skryptu *TestAdminCreateEventLog.ps1*:

TestAdminCreateEventLog.ps1

```
function Test-IsAdministrator
{
    <#
    .Synopsis
        Tests if the user is an administrator
    .Description
        Returns true if a user is an administrator,
        false if the user is not an administrator
    .Example
        Test-IsAdministrator
    #>

    param()
    $currentUser = [Security.Principal.WindowsIdentity]::GetCurrent()
    (New-Object Security.Principal.WindowsPrincipal $currentUser).IsInRole '
```

```

    ([Security.Principal.WindowsBuiltInRole]::Administrator)
} #end funkcja Test-IsAdministrator

# *** punkt początkowy skryptu ***
If(-not (Test-IsAdministrator)) { "Ten skrypt wymaga do działania uprawnień administratora."
; exit }
New-EventLog -LogName scripting -Source processAudit

```

Sposób użycia dziennika aplikacji

Najłatwiej jest używać dziennika aplikacji, ponieważ jest zawsze dostępny i nie trzeba do niego mieć uprawnień administratora. W dzienniku zdarzeń musi istnieć źródło. Jeśli zostanie wybrane istniejące źródło, ale zostanie użyty identyfikator nieistniejącego zdarzenia, nie nastąpi wygenerowanie jakiegokolwiek błędu, tylko w danych zdarzenia pojawi się informacja o brakującym opisie zdarzenia.

```

PS C:\> Write-EventLog -LogName application -Source certenroll -EntryType information '
-EventId 0 -Message "test"
PS C:\> Get-WinEvent -LogName application -MaxEvents 1 | format-list *

```

```

EventID           : 0
MachineName       : EDWILSON.microsoft.com
Data              : {}
Index             : 6130
Category          : (1)
CategoryNumber    : 1
EntryType         : Information
Message           : The description for Event ID '0' in Source 'certenroll' cannot be found.
                   The local computer may not have the necessary registry information or
                   message DLL files to display the message or you may not have permission to
                   access them. The following information is part of the event:'test'
Source            : certenroll
ReplacementStrings : {test}
InstanceId        : 0
TimeGenerated     : 2009-08-17 00:03:52
TimeWritten       : 2009-08-17 00:03:52
UserName          :
Site              :
Container         :

```

Tworzenie własnego dziennika zdarzeń

Najlepszą metodą rejestrowania zdarzeń jest utworzenie własnego dziennika zdarzeń ze swoimi źródłami. Jako że dziennik aplikacji jest intensywnie eksploatowany przez wiele różnych źródeł, pobieranie z niego zdarzeń wymaga przekopywania się przez tysiące wpisów. Mając własny dziennik zdarzeń, sprawuje się pełną kontrolę nad liczbą wpisów, zdefiniowanymi źródłami oraz szczegółowością rejestrowanych informacji. Z tych powodów znajdowanie wpisów we własnym dzienniku zdarzeń jest z reguły łatwiejsze niż w dzienniku systemowym lub aplikacji. Do tworzenia nowego dziennika zdarzeń służy polecenie `New-EventLog`, któremu należy przekazać nazwę dziennika i źródło zdarzeń.

UWAGA

Jeśli czytasz rozdziały książki po kolei i wykonujesz wszystkie opisane w nich polecenia, to poniższego polecenia nie uda się wykonać, ponieważ wymienione w nim źródło zdarzeń zostało już utworzone wcześniej.

```
New-EventLog -LogName ForScripting -Source scripting
```

Błąd powstanie z tego powodu, że źródło scripting zostało już wcześniej utworzone.

Trzeba więc je usunąć za pomocą poniższego polecenia:

```
Remove-EventLog -Source scripting
```

```
PS C:\> New-EventLog -LogName ForScripting -Source scripting
```

Aby dokonać zapisu w dzienniku zdarzeń, należy użyć polecenia `Write-EventLog`. Należy podać nazwę dziennika, źródło, typ wpisu oraz identyfikator zdarzenia i treść komunikatu. Wszystko to można wpisać w jednym wierszu kodu:

```
PS C:\> Write-EventLog -LogName ForScripting -Source scripting -EntryType information '
-EventId 0 -Message test
```

Do odczytywania wpisów z dziennika zdarzeń służy polecenie `Get-EventLog`, któremu należy przekazać nazwę dziennika.

```
PS C:\> Get-WinEvent -LogName ForScripting
```

```
ProviderName: scripting
```

| TimeCreated | Id | LevelDisplayName | Message |
|---------------------|-----|------------------|---------|
| ----- | --- | ----- | ----- |
| 2014-10-31 10:03:48 | 0 | Informacje | test |

Zapisywanie danych w rejestrze

Rejestr jest idealnym miejscem do przechowywania niewielkich porcji informacji, takich jak kod zakończenia działania skryptu albo znacznik czasu. Ze względu na rodzaj bazy danych, jaką jest rejestr, nie należy w nim przechowywać dużych ilości informacji. Ponadto obiekty powinno się zamienić na łańcuchy za pomocą polecenia `Out-String` lub metody `ToString`.

Najlepszym miejscem w rejestrze do zapisywania informacji jest gałąź `Hkey_Current_User`, ponieważ dotyczy bieżącego użytkownika, który ma prawo zapisu w gałęzi `Current_User`. W ten sposób unika się wszelkich problemów związanych z uprawnieniami. Przykład zastosowania techniki zapisu informacji w rejestrze znajduje się w skrypcie *CreateRegistryKey.ps1*, który tworzy klucz o nazwie `ForScripting` w gałęzi `Hkey_Current_User`. W kluczu tym tworzy własność o nazwie *forscripting* i wartości *test*.

Skrypt *CreateRegistryKey.ps1* zawiera funkcję o nazwie `Add-RegistryValue`, która przyjmuje dwa parametry — zmienne `$key` i `$value`. Można ją też tak zmodyfikować, aby dodatkowo przyjmowała katalog główny rejestru. Wartość zmiennej `$scriptRoot` określa miejsce utworzenia klucza. Jeśli taka ścieżka nie istnieje, to zostanie utworzona, a następnie zostanie dodana własność z wartością. Do sprawdzania, czy ścieżka do klucza `$scriptRoot` istnieje, używane jest polecenie `Test-Path`. Klucz jest tworzony za pomocą polecenia `New-Item`, natomiast własność z wartością

tworzy polecenie `New-ItemProperty`. Ponadto użyto też polecenia `Out-Null`, aby uniemożliwić wyświetlenie wyników operacji w oknie konsoli.

```
Function Add-RegistryValue
{
    Param ($key,$value)
    $scriptRoot = "HKCU:\software\ForScripting"
    if(-not (Test-Path -path $scriptRoot))
    {
        New-Item -Path HKCU:\Software\ForScripting | Out-Null
        New-ItemProperty -Path $scriptRoot -Name $key -Value $value '
        -PropertyType String | Out-Null
    }

    Else
    {
        Set-ItemProperty -Path $scriptRoot -Name $key -Value $value | '
        Out-Null
    }
}
```

Jeżeli dany klucz istnieje, polecenie `Set-ItemProperty` tworzy w nim wartość własności lub zmienia istniejącą wartość własności. Wynik tego działania jest przekazywany do polecenia `Out-Null`.

W punkcie początkowym wywoływana jest funkcja `Add-RegistryValue`, której przekazano nazwę klucza do zmodyfikowania i wartość.

```
Add-RegistryValue -key forscripting -value test
```

Poniżej znajduje się kompletny kod źródłowy skryptu *CreateRegistryKey.ps1*:

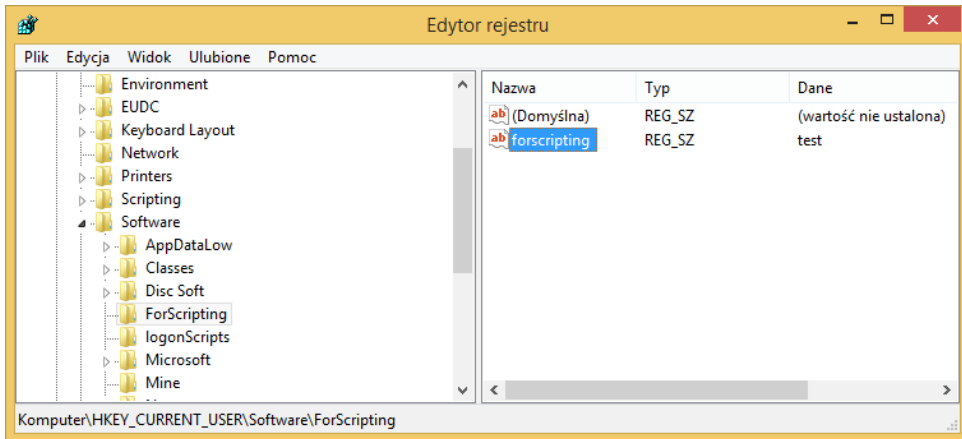
CreateRegistryKey.ps1

```
Function Add-RegistryValue
{
    Param ($key,$value)
    $scriptRoot = "HKCU:\software\ForScripting"
    if(-not (Test-Path -path $scriptRoot))
    {
        New-Item -Path HKCU:\Software\ForScripting | Out-Null
        New-ItemProperty -Path $scriptRoot -Name $key -Value $value '
        -PropertyType String | Out-Null
    }
    Else
    {
        Set-ItemProperty -Path $scriptRoot -Name $key -Value $value | '
        Out-Null
    }
}

#end funkcja Add-RegistryValue

# *** punkt początkowy skryptu ***
Add-RegistryValue -key forscripting -value test
```

Podczas działania skrypt *CreateRegistryKey.ps1* niczego nie wyświetla na ekranie. Tworzy tylko klucz rejestru *ForScripting* z własnością *forscripting* o wartości *test*, jak pokazano na rysunku 18.5.



RYSUNEK 18.5. Gałąź rejestru Current_User to doskonałe miejsce do przechowywania niewielkich porcji informacji

Dodatkowe źródła informacji

- Centrum skryptów portalu TechNet pod adresem <http://www.microsoft.com/technet/scriptcenter> zawiera wiele przykładów skryptów zapisujących dane w plikach tekstowych, dziennikach zdarzeń i rejestrze.
- Więcej informacji na temat rejestrowania wyników skryptów znajduje się w rozdziale 3. książki *Windows PowerShell™ Scripting Guide* (Microsoft Press 2008).
- Wszystkie skrypty opisane w tym rozdziale są dostępne do pobrania w repozytorium centrum skryptów portalu TechNet pod adresem <http://gallery.technet.microsoft.com/scriptcenter/PowerShell-40-Best-d9e16039>.

Rozdział 19

Rozwiązywanie problemów ze skryptami

- Podstawy debugowania w Windows PowerShell
- Sposób użycia polecenia Set-PSDebug
- Debugowanie skryptów
- Dodatkowe źródła informacji

Dobrze zaprojektowany i napisany skrypt wymaga niewiele uwagi. To oczywiście nie znaczy, że wszystkie skrypty są doskonałe czy że wszystkie są bezbłędne już przy pierwszym uruchomieniu, ani nawet przy drugim. Mimo to dobry skrypt powinien być tak skonstruowany, aby jego kod był czytelny i zrozumiały. Jeśli warunki te zostaną spełnione, skrypt będzie łatwiejszy do diagnozowania i obsługi, ponieważ błędy będą po prostu lepiej widoczne. Jeśli jednak już pojawią się jakieś problemy, warto wiedzieć, jak debugować skrypt. W tym rozdziale opisuję polecenia i najlepsze praktyki dotyczące śledzenia wykonywania skryptów, wykonywania poleceń skryptów krok po kroku oraz debugowania skryptów. Konsola Windows PowerShell 4.0 zawiera wiele przydatnych narzędzi, dzięki którym usuwanie usterek, jeśli w ogóle się pojawiają, nie jest trudne.

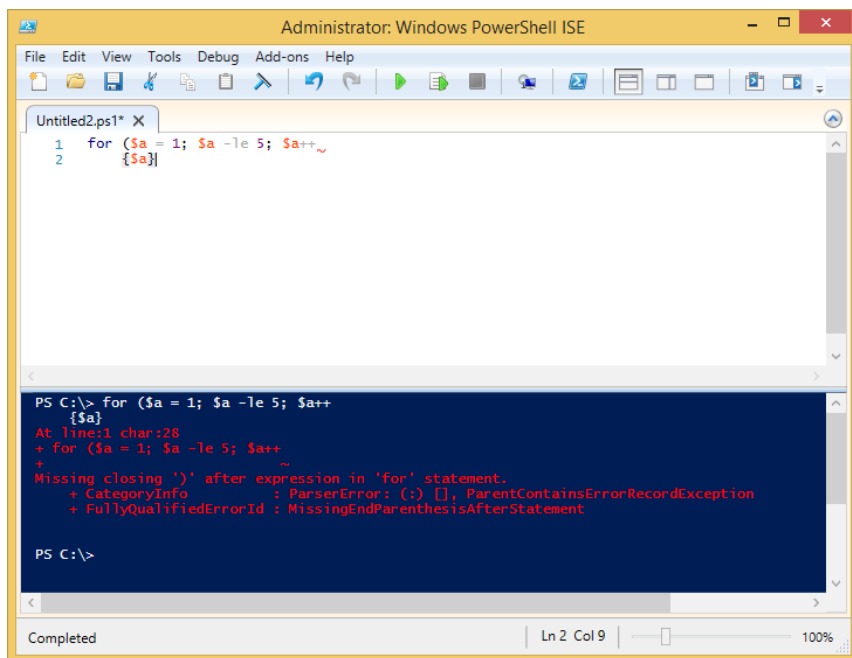
Podstawy debugowania w Windows PowerShell

Jeśli rozumiesz napisany przez siebie kod Windows PowerShell, to jest duża szansa, że nie sprawi Ci on wielu kłopotów. Ale co zrobić, jeżeli jednak trzeba przeprowadzić diagnostykę? Tak jak golfiści godzinami ćwiczą wydobywanie piłki z bunkra, choć mają nadzieję, że nigdy się im ta umiejętność nie przyda, tak kompetentny specjalista od pisania skryptów Windows PowerShell powinien ćwiczyć debugowanie skryptów, nawet jeśli liczy na to, że nie skorzysta z tej wiedzy. Znajomość kolorowania składni w środowisku Windows PowerShell ISE, umiejętność ocenienia, kiedy brakuje cudzysłowu, oraz wiedza, które klamry są związane z danym poleceniem — to wszystko cechy pozwalające znacznie skrócić ewentualny czas debugowania.

Debugowanie to czynność polegająca na znajdowaniu i eliminowaniu błędów z kodu źródłowego skryptów. Wyróżnia się trzy rodzaje błędów: składniowe, logiczne oraz wykonawcze.

Błędy składniowe

Błędy składniowe są najłatwiejsze do wychwycenia i najczęściej usuwa się je już podczas pisania kodu w środowisku Windows PowerShell ISE. Jest to możliwe głównie dzięki temu, że w tle działa parser języka, który wykrywa błędy składniowe i oznacza je falistym podkreśleniem (dzięki któremu wiadomo, że dane polecenie wymaga dodatkowego parametru, jakiegoś znaku itp.). Dla doświadczonych programistów skryptów funkcja ta jest ich dodatkowym okiem. Nie traktują jej jak mechanizm poprawiania błędów, tylko jak narzędzia do uzupełniania poleceń. Najbardziej doświadczeni programiści umieją współpracować z parserem i poprawiają wszelkie podkreślone błędy przed pierwszym uruchomieniem kodu. Jeśli pozostawi się jakiś błąd, to po uruchomieniu skryptu konsola zazwyczaj wyświetla informację naprowadzającą na to, w czym tkwi problem. Na rysunku 19.1 widać błąd składni w skrypcie.

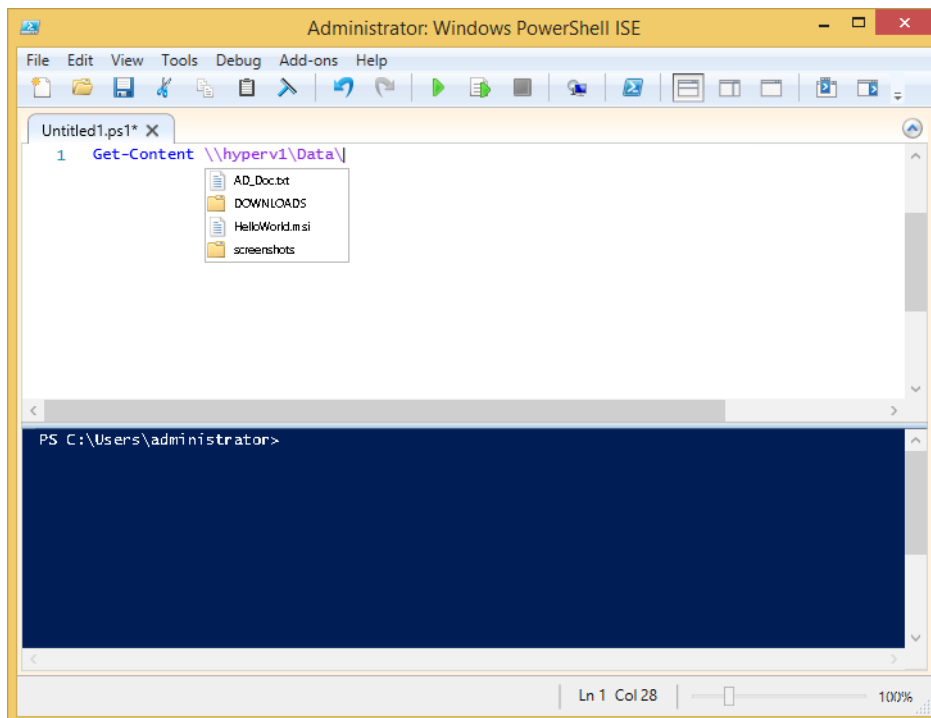


RYСУNEK 19.1. W środowisku Windows PowerShell ISE potencjalnie błędne fragmenty kodu zostają podkreślone czerwoną falowaną linią. W wiadomości o błędzie podawana jest nazwa błędnego polecenia i znajduje się wskazówka, co należy poprawić

Błędy wykonawcze

Analizator składni nie potrafi wykryć wszystkich błędów wykonawczych, które ujawniają się dopiero po uruchomieniu skryptu. Wśród przykładów tego typu błędów można wymienić brak dostępności zasobów (np. dysku albo pliku), problemy z uprawnieniami (np. zwykły użytkownik nie może wykonać pewnych czynności wymagających uprawnień administratora), literówki oraz niespełnione warunki (np. brak dostępu do jakiegoś modułu). Na szczęście wiele z tych błędów w środowisku Windows PowerShell ISE 4.0 można wyeliminować dzięki funkcji rozwijania nazw za pomocą klawisza *Tab*. Korzystając z tej funkcji, można na przykład całkowicie wykluczyć

ryzyko spowodowania błędów związanych z brakiem dostępu do zasobów, ponieważ funkcja ta działa nawet w udziałach UNC. Na rysunku 19.2 widać ją w akcji podczas próby odczytu za pomocą polecenia `Get-Content` treści pliku o nazwie `AD_Doc.txt` z udziału `Data` na serwerze o nazwie `hyperv1`.



RYСУNEK 19.2. Udoskonalona funkcja rozwijania nazw za pomocą klawisza Tab pozwala uniknąć pewnego rodzaju błędów wykonawczych

Niestety funkcja rozwijania nie jest pomocna w przypadku kwestii uprawnień, chociaż czytając uważnie wiadomość o błędzie, można przynajmniej się dowiedzieć, o co chodzi. Z reguły wyświetlana jest informacja o odmowie dostępu (ang. *Access Is Denied*). Pojawia się ona na przykład w poniższym przykładzie, w którym użytkownik `boguser` próbuje wykonać zapytanie WMI na serwerze `DC1`.

```
PS C:\> Get-WmiObject win32_bios -cn dc1 -Credential iammred\boguser
Get-WmiObject : Access is denied. (Exception from HRESULT: 0x80070005 (E_ACCESSDENIED))
At line:1 char:1
+ Get-WmiObject win32_bios -cn dc1 -Credential iammred\boguser
+ ~~~~~
+ CategoryInfo          : NotSpecified: (:) [Get-WmiObject], UnauthorizedAccessException
+ FullyQualifiedErrorId : System.UnauthorizedAccessException,
    Microsoft.PowerShell.Commands.GetWmiObjectCommand
```

Jednym ze sposobów na wykrywanie błędów wykonawczych jest wyświetlanie zawartości podejrzanych zmiennych za pomocą polecenia `Write-Debug`. Jeśli zamieni się jednowierszowe polecenie w krótki skrypt ze zmiennymi i poleceniami `Write-Debug`, to można sprawić, że typowe

techniki usuwania usterek będą działały automatycznie. Na przykład w pokazanym poniżej skrypcie *RemoteWMI_SessionNoDebug.ps1* znajdują się dwa źródła błędów wykonawczych: dostępność komputera docelowego i dane poświadczające do nawiązania połączenia.

RemoteWMI_SessionNoDebug.ps1

```
$credential = Get-Credential
$cn = Read-Host -Prompt "Wpisz nazwę komputera"
Get-WmiObject win32_bios -cn $cn -Credential $credential
```

W okienku bezpośrednim środowiska Windows PowerShell ISE można sprawdzić wartości zmiennych `$cn` i `$credential`. Można też za pomocą polecenia `Test-Connection` sprawdzić status komputera `$cn`. Dzięki przewidzeniu tych typowych problemów skrypt wyświetli precyzyjne informacje i nie będzie trzeba w ogóle przeprowadzać diagnostyki. Poniżej znajduje się kod źródłowy skryptu *DebugRemoteWMI_Session.ps1* z poleceniami `Write-Debug` wyświetlającymi informacje diagnostyczne.

DebugRemoteWMI_Session.ps1

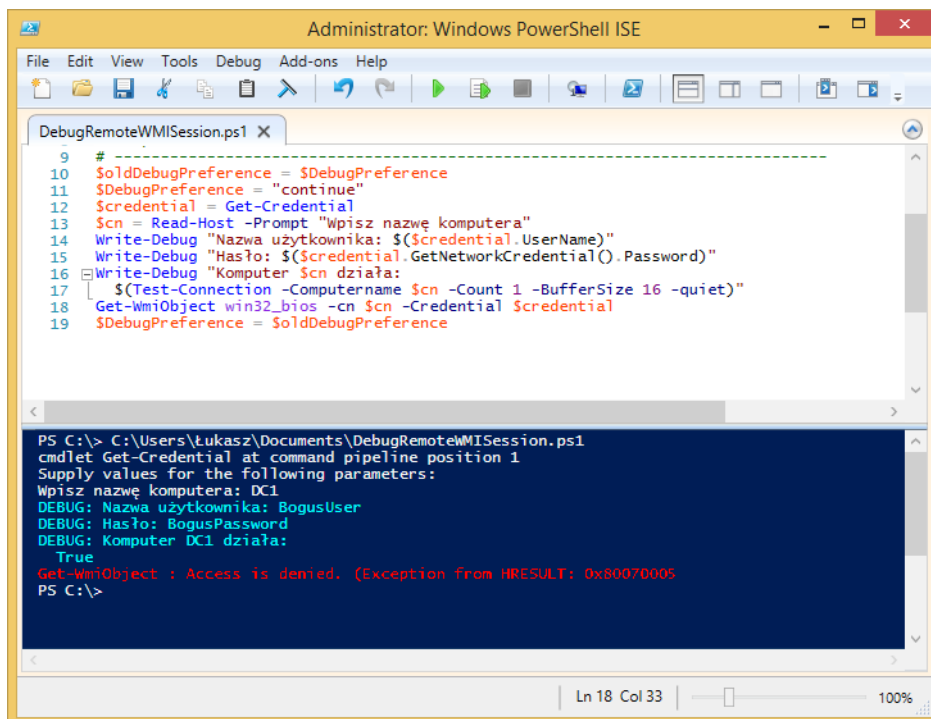
```
$oldDebugPreference = $DebugPreference
$DebugPreference = "continue"
$credential = Get-Credential
$cn = Read-Host -Prompt "Wpisz nazwę komputera"
Write-Debug "Nazwa użytkownika: $($credential.UserName)"
Write-Debug "Hasło: $($credential.GetNetworkCredential().Password)"
Write-Debug "Komputer $cn działa:
    $(Test-Connection -Computername $cn -Count 1 -BufferSize 16 -quiet)"
Get-WmiObject win32_bios -cn $cn -Credential $credential
$DebugPreference = $oldDebugPreference
```

Na rysunku 19.3 pokazano efekt uruchomienia skryptu *DebugRemoteWMI_Session.ps1* w konsoli Windows PowerShell ISE w celu znalezienia przyczyny błędu. Z danych wynika, że zdalny serwer *DC1* jest dostępny, ale użytkownik *Bogus User* posługujący się hasłem *BogusPassword* nie ma prawa dostępu. Możliwe, że użytkownik ten nie ma konta, nie ma odpowiednich uprawnień albo że podał niepoprawne hasło. Sprawę powinny wyjaśnić szczegółowe dane diagnostyczne.

Jeszcze lepszym pomysłem jest użycie polecenia `Write-Debug` wraz z atrybutem `[CmdletBinding()]` na początku skryptu (lub funkcji). Aby atrybut ten zadziałał, musi być spełnionych kilka warunków. Po pierwsze: skrypt lub funkcja musi mieć przynajmniej jeden parametr. To oznacza, że będzie dostępne słowo kluczowe `param`. Po drugie: atrybut `[CmdletBinding()]` musi znajdować się przed słowem kluczowym `param`. Modyfikacje te pozwolą nam na używanie wspólnego parametru `debug`. Parametr ten jest przełącznikiem powodującym wyświetlenie na wyjściu danych ze strumienia `Write-Debug`. Ponadto ta prosta zmiana wyeliminuje potrzebę używania zmiennej `$DebugPreference` oraz konieczność tworzenia własnego przełącznika `debug` i dodawania na początku skryptu kodu podobnego do poniższego:

```
Param([switch]$debug)
If($debug) {$DebugPreference = "continue"}
```

Poniżej znajduje się uproszczona wersja skryptu zapisana pod nazwą *Switch_DebugRemote-WMI_Session.ps1*. Najważniejsze zmiany to wprowadzenie atrybutu `[CmdletBinding()]`, utworzenie parametru o nazwie `cn` oraz ustawienie domyślnej wartości na nazwę komputera lokalnego. Poza tym usunięto zmienną `$DebugPreference`.



RYСУNEK 19.3. Szczegółowa diagnostyka ułatwia usuwania błędów wykonawczych

Switch_DebugRemoteWMIsession.ps1

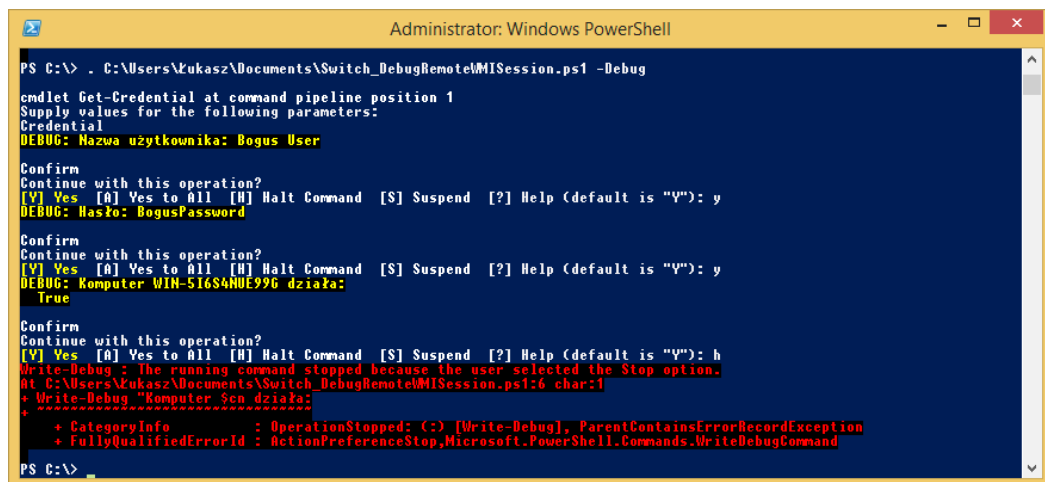
```

[CmdletBinding()]
Param($cn = $env:computername)
$credential = Get-Credential
Write-Debug "Nazwa użytkownika: $($credential.UserName)"
Write-Debug "Hasło: $($credential.GetNetworkCredential().Password)"
Write-Debug "Komputer $cn działa:"
while ($?) {
    $(Test-Connection -Computername $cn -Count 1 -BufferSize 16 -quiet)"
    Get-WmiObject win32_bios -cn $cn -Credential $credential
}
  
```

Gdy skrypt *Switch_DebugRemoteWMIsession.ps1* zostanie uruchomiony z przełącznikiem debug, to oprócz strumienia diagnostycznego wyświetli też monit o potwierdzenie chęci kontynuowania wykonywania. Dzięki temu można zatrzymać dalsze wykonywanie, jeśli zostanie napotkana niespodziewana wartość. Na rysunku 19.4 pokazano sytuację, w której użytkownik o nazwie Bogus User chce się połączyć ze zdalnym serwerem o nazwie DC1, ale odkrywa, że łączy się ze stacją roboczą o nazwie W8Client6.

Błędy logiczne

Błędy logiczne bywają bardzo trudne do wykrycia, ponieważ zawierające je skrypty wyglądają tak, jakby działały poprawnie, i w niektórych przypadkach mogą nawet zwracać poprawne wyniki. Ale jeśli coś się nie uda, to bardzo trudno jest to naprawić. Samo przeanalizowanie wartości zmiennych może nie wystarczyć, ponieważ kod sam w sobie działa bez zarzutu.



RYСУNEK 19.4. Sposób użycia przełącznika debug, aby zatrzymać wykonywanie skryptu na ewentualnych błędach

Problem dotyczy tzw. **zasad biznesowych** skryptu. Są to decyzje podejmowane przez program i niemające nic wspólnego z działaniem np. instrukcji swi t.ch. Czasami może to wyglądać tak, jakby właśnie ta instrukcja działała błędnie, ponieważ na jej końcu zwracana jest niepoprawna wartość. Ale bardzo często przyczyną usterki są właśnie zasady biznesowe skryptu.

Prosty przykład błędu logicznego znajduje się w przedstawionym poniżej skrypcie *My-Function.ps1*.

My-Function.ps1

```
Function my-function
{
  Param(
    [int]$a,
    [int]$b)
    "$a plus $b równa się cztery"
}
```

Funkcja `My-Function` przyjmuje dwa parametry wiersza poleceń, o nazwach `a` i `b`. Ich wartości wstawia do łańcucha i wyświetla informację, że suma tych dwóch liczb wynosi cztery. Tester wykonuje cztery różne testy i za każdym razem funkcja działa zgodnie z oczekiwaniami:

```
PS C:\> S:\psh sbs 4\chapter19\scripts\my-function.ps1
```

```
PS C:\> my-function -a 2 -b 2
2 plus 2 równa się cztery
```

```
PS C:\> my-function -a 1 -b 3
1 plus 3 równa sie cztery
```

```
PS C:\> my-function -a 0 -b 4
0 plus 4 równa sie cztery
```

```
PS C:\> my-function -a 3 -b 1
3 plus 1 równa sie cztery
```


Ale po przekazaniu funkcji do produkcji zaczynają napływać skargi od użytkowników, ponieważ funkcja mało kiedy zwraca poprawny wynik, chociaż nigdy nie ulega awarii. Jak najlepiej rozwiązać taki problem? Samo wyświetlenie wartości zmiennych `a` i `b` za pomocą poleceń `Write-Debug` może nie wystarczyć. Lepszym rozwiązaniem jest wykonanie kodu linijka po linijce i obserwowanie wyników. W tym celu można użyć polecenia `Set-PSDebug`, które jest tematem następnego podrozdziału.

Zapiski praktyka

Luc Dekens, inżynier systemów, MVP Windows PowerShell

Eurocontrol

Czasami piękno tkwi w drobiazgach, czego dowodem jest wprowadzony w Windows PowerShell 3.0 atrybut `[ordered]`.

Wcześniej nie można było kontrolować kolejności wyświetlania własności tablicy skrótów. Wprawdzie można to było ominąć za pomocą polecenia `Select-Object`, ale dla mnie to było zbyt skomplikowane i nieintuicyjne rozwiązanie.

Najlepiej wytłumaczyć to na przykładzie.

Jedno z moich dzieci miało przygotować do szkoły tabelę przedstawiającą twierdzenie Pitagorasa. To było łatwe, ponieważ syn trochę umiał posługiwać się konsolą Windows PowerShell i wiedział, że może wywoływać funkcje `.Net`.

```
$table = @()
for($i = 0; $i -le 90; $i += 10){
    $rad = $i*[Math]::PI/180
    $cos = [math]::Cos($rad)
    $sin = [math]::Sin($rad)
    $row = @{
        Angle = $i
        Cosine = "{0:n2}" -f $cos
        Sine = "{0:n2}" -f $sin
        Pythagoras = [math]::Pow($cos,2) + [math]::Pow($sin,2)
    }
    $table += New-Object PSObject -Property $row
}
$table | Export-Csv pyth.csv -NoTypeInfo -UseCulture
```

Szczegóły działania tego skryptu pozostawiam Czytelnikowi do samodzielnego rozszyfrowania.

Syn utworzył otrzymany plik CSV i to, co zobaczył, trochę go rozczarowało.

Skrypt zadziałał, w pliku znajdowały się odpowiednie wartości dowodzące poprawności twierdzenia Pitagorasa, ale kolejność kolumn była inna, niż się spodziewał. To nie tak miało wyglądać.

W konsoli Windows PowerShell 3.0 pojawiło się rozwiązanie tego problemu niewymagające użycia dodatkowego polecenia `Select-Object`.

W nowej wersji skryptu trzeba było wprowadzić tylko drobną zmianę polegającą na dodaniu znacznika [ordered] w wierszu tworzenia tablicy skrótów.

```
$table = @()
for($i = 0; $i -le 90; $i += 10){
    $rad = $i*[Math]::PI/180
    $cos = [math]::Cos($rad)
    $sin = [math]::Sin($rad)
    $row = [ordered]@{
        Angle = $i
        Cosine = "{0:n2}" -f $cos
        Sine = "{0:n2}" -f $sin
        Pythagoras = [math]::Pow($cos,2) + [math]::Pow($sin,2)
    }
    $table += New-Object PSObject -Property $row
}
$table | Export-Csv pyth-v3.csv -NoTypeInfoInformation -UseCulture
```

W istocie znacznik [ordered] nie wnosi do konsoli Windows PowerShell niczego nowego, bo już od dawna istniały uporządkowane tablice skrótów.

```
$object1 = New-Object System.Collections.Specialized.OrderedDictionary
```

Ale w Windows PowerShell 3.0 znacznie ułatwiono zapisywanie i odczytywanie tablic.

```
$object2 = [ordered]@{}
```

Ponadto wraz ze znacznikiem [ordered] pojawiła się jeszcze nowa klasa [PSCustomObject] do tworzenia własnych obiektów z tablic skrótów. Jej zalety to zachowywanie kolejności i większa szybkość działania od niektórych innych metod tworzenia własnych obiektów.

```
Function Get-Pythagoras {
    param($angle)

    $rad = $angle*[Math]::PI/180
    $cos = [math]::Cos($rad)
    $sin = [math]::Sin($rad)

    [PSCustomObject]@{
        Angle = $angle
        Cosine = "{0:n2}" -f $cos
        Sine = "{0:n2}" -f $sin
        Pythagoras = [math]::Pow($cos,2) + [math]::Pow($sin,2)
    }
}

for($i = 0; $i -le 90; $i += 10){
    Get-Pythagoras -Angle $i
}
```

Sposób użycia polecenia Set-PSDebug

Polecenie Set-PSDebug jest dostępne już w konsoli Windows PowerShell 1.0 i nic się nie zmieniło w Windows PowerShell 4.0. To oczywiście nie znaczy, że je zaniedbano, tylko że po prostu robi to, co powinno. Jeśli trzeba szybko i łatwo przeprowadzić podstawową diagnostykę, to nic nie przebijie dostępnych standardowych narzędzi. Za pomocą polecenia Set-PSDebug można zrobić

trzy rzeczy. Można włączyć automatyczne śledzenie wykonywania skryptu, interaktywnie wykonać skrypt krok po kroku oraz włączyć tryb ścisły. W podrozdziale tym znajduje się opis każdej z tych czynności. Polecenie Set-PSDebug nie służy do prowadzenia skomplikowanych procesów diagnostycznych. Jest to lekkie narzędzie do szybkiego sprawdzenia, jak działa skrypt, lub wykonania skryptu po jednym wierszu kodu.

Śledzenie wykonywania skryptu

Jednym z najłatwiejszych sposobów debugowania skryptu jest włączenie śledzenia, aby każde wykonywane polecenie było wyświetlane w oknie konsoli. W ten sposób można sprawdzić, czy wszystkie linijki kodu są wykonywane. Aby włączyć śledzenie wykonywania skryptu, należy wywołać polecenie Set-PSDebug z jednym z trzech parametrów, których opis znajduje się w tabeli 19.1.

TABELA 19.1. Poziomy śledzenia wykonywania skryptu

| Poziom śledzenia | Opis |
|------------------|--|
| 0 | Wyłącza śledzenie wykonywania skryptu |
| 1 | Śledzi wykonywanie każdej linijki skryptu. Niewykonywane linijki nie są śledzone. Nie są wyświetlane przypisania wartości do zmiennych, wywołania funkcji ani skrypty zewnętrzne |
| 2 | Śledzi wykonywanie każdej linijki skryptu. Wyświetlane są przypisania wartości do zmiennych, wywołania funkcji i skrypty zewnętrzne. Niewykonywane linijki nie są śledzone |

Sposób działania poszczególnych parametrów poziomu śledzenia przeanalizujemy na podstawie przykładowego skryptu o nazwie *CreateRegistryKey.ps1*. Zawiera on tylko funkcję Add-Registry, w której polecenie Test-Path sprawdza, czy istnieje określony klucz rejestru. Jeśli tak, następuje ustawienie jego wartości. Jeśli nie, najpierw tworzony jest ten klucz, a następnie zostaje ustawiona jego wartość. Funkcja ta jest wywoływana w momencie uruchomienia skryptu. Poniżej znajduje się kompletny kod źródłowy skryptu *CreateRegistryKey.ps1*:

CreateRegistryKey.ps1

```
Function Add-RegistryValue($key,$value)
{
    $scriptRoot = "HKCU:\software\ForScripting"
    if(-not (Test-Path -path $scriptRoot))
    {
        New-Item -Path HKCU:\Software\ForScripting | Out-null
        New-ItemProperty -Path $scriptRoot -Name $key -Value $value '
        -PropertyType String | Out-Null
    }
    Else
    {
        Set-ItemProperty -Path $scriptRoot -Name $key -Value $value | '
        Out-Null
    }
} #end funkcja Add-RegistryValue
```

```
# *** punkt początkowy skryptu ***
Add-RegistryValue -key forscripting -value test
```

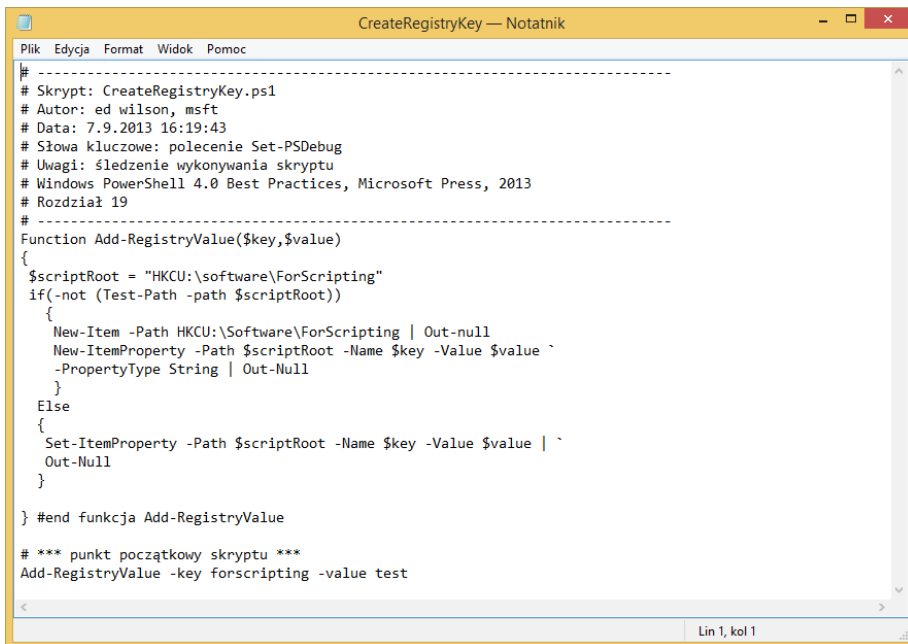
Poziom pierwszy śledzenia

Gdy poziom śledzenia jest ustawiony na 1, w konsoli Windows PowerShell wyświetlane są nazwy wszystkich wykonywanych poleceń. Aby ustawić ten poziom, wystarczy wykonać polecenie `Set-PSDebug`, przekazując wartość 1 do parametru `trace`. Po naciśnięciu klawisza *Enter* od razu nastąpi przejście do nowego wiersza, jak widać poniżej:

```
Windows PowerShell
Copyright (C) 2013 Microsoft Corporation. All rights reserved.

PS C:\Users\ed.IAMMRED> Set-PSDebug -Trace 1
PS C:\Users\ed.IAMMRED>
```

Ustawiony w ten sposób poziom śledzenia ma zastosowanie do wszystkiego, co jest wpisywane w konsoli — poleceń interaktywnych, poleceń cmdlet, skryptów itd. Jeśli podczas uruchamiania skryptu *CreateRegistryKey.ps1* interesujący nas klucz rejestru nie istnieje, w pierwszej linijce zostanie wyświetlona ścieżka do wykonywanego skryptu. Jako że konsola Windows PowerShell przetwarza kod od góry, w następnej kolejności zostaje wykonany wiersz tworzący funkcję `Add-RegistryValue`. Polecenie to znajduje się w dziesiątej linijce skryptu, ponieważ na początku jest dziewięć linijek komentarzy. W Notatniku numery linijek są wyświetlane w prawym dolnym rogu paska stanu (*Widok/Pasek stanu*), jak widać na rysunku 19.5.



RYСУNEK 19.5. Na pasku stanu w prawym dolnym rogu okna Notatnika wyświetlane są numery linijek tekstu

Następny wykonywany wiersz po utworzeniu funkcji ma numer 25. Wiersz ten znajduje się za komentarzem oznaczającym punkt początkowy skryptu i zawiera wywołanie funkcji Add-RegistryValue z parametrami key i value:

```
PS C:\> E:\Data\BookD0cs\PS4_BestPractices\Scripts\scripts_ch19\CreateRegistryKey.ps1
DEBUG: 1+ >>>>
E:\Data\BookD0cs\PS4_BestPractices\Scripts\scripts_ch19\CreateRegistryKey.ps1
DEBUG: 28+ >>>> Add-RegistryValue -key forscripting -value test
DEBUG: 11+ >>>> {
DEBUG: 12+ >>>> $scriptRoot = "HKCU:\software\ForScripting"
DEBUG: 13+ if( >>>> -not (Test-Path -path $scriptRoot))
DEBUG: 15+ >>>> New-Item -Path HKCU:\Software\ForScripting | Out-null
DEBUG: 16+ >>>> New-ItemProperty -Path $scriptRoot -Name $key -Value $value
DEBUG: 25+ >>>> } #end funkcja Add-RegistryValue
```

Wewnątrz funkcji Add-RegistryValue następuje przypisanie łańcucha HKCU:\software\ForScripting do zmiennej \$scriptRoot:

```
DEBUG: 12+ >>>> $scriptRoot = "HKCU:\software\ForScripting"
```

Teraz następuje ewaluacja instrukcji if. Jeżeli polecenie Test-Path nie znajdzie w rejestrze lokalizacji \$scriptRoot, zostanie wykonany kod znajdujący się w bloku instrukcji if. W tym przykładzie nie znaleziono lokalizacji \$scriptRoot, więc nastąpiło wykonanie poleceń z tego bloku:

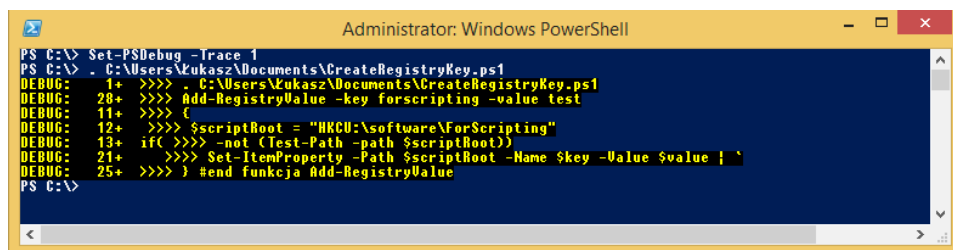
```
DEBUG: 13+ if( >>>> -not (Test-Path -path $scriptRoot))
```

W wierszach 15. i 16. wykonane zostały polecenia New-Item i New-ItemProperty.

```
DEBUG: 15+ >>>> New-Item -Path HKCU:\Software\ForScripting | Out-null
DEBUG: 16+ >>>> New-ItemProperty -Path $scriptRoot -Name $key -Value $value
```

Po zakończeniu wykonywania skryptu parser wraca do normalnego stanu.

Dzięki ustawieniu pierwszego poziomu śledzenia otrzymaliśmy podstawowy plan wykonywania skryptu. Technika ta pozwala szybko sprawdzić wyniki wykonywania instrukcji rozgałęzionych (takich jak np. if). Na rysunku 19.6 pokazano wynik diagnostyki omawianego skryptu.



RYСУNEK 19.6. Poziom pierwszy śledzenia wykonywania skryptu wyświetla wszystkie wykonane polecenia

Poziom drugi śledzenia

Na drugim poziomie śledzenia również wyświetlane są wszystkie wykonywane w konsoli polecenia. Ponadto uwzględniane są przypisania wartości do zmiennych, wywołania funkcji i wywołania skryptów zewnętrznych. Dla ułatwienia znajdowania wszystkie te dodatkowe

szczególne są poprzedzone wykrzyknikiem. Ustawienie parametru `Trace` polecenia `Set-PSDebug` na 2 powoduje pojawienie się dodatkowej linii z przypisaniem wartości do zmiennej:

```
PS C:\> Set-PSDebug -Trace 2
DEBUG: 1+ <<<< Set-PSDebug -Trace 2
DEBUG: 2+ $foundSuggestion = <<<< $false
DEBUG: ! SET $foundSuggestion = 'False'.
DEBUG: 4+ if <<<< ($lastError -and
DEBUG: 15+ $foundSuggestion <<<<
```

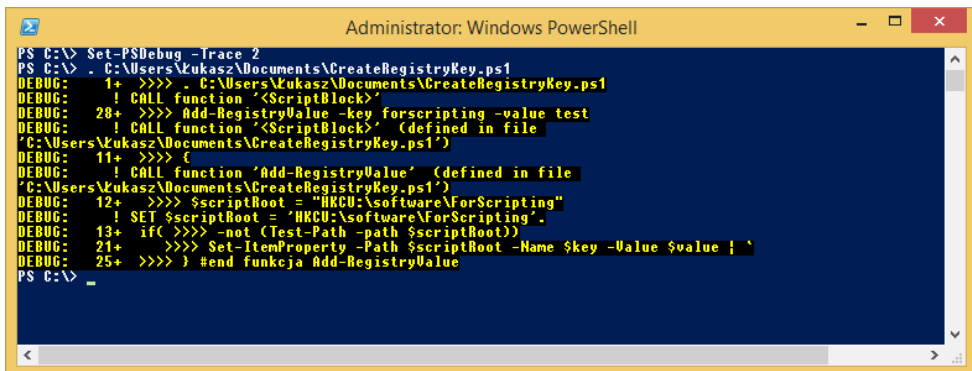
Po uruchomieniu skryptu *CreateRegistryKey.ps1* najpierw zostaje wyświetlona informacja, że wywoływana jest funkcja o nazwie *CreateRegistryKey.ps1*. Wywołania funkcji są poprzedzone przedrostkiem `! CALL`, który ułatwia ich znalezienie w gąszczu informacji. Konsola Windows PowerShell traktuje skrypty jak funkcje. Następnie zostaje wywołana funkcja o nazwie `Add-RegistryValue`. W danych znajduje się też ścieżka do zawierającego ją pliku:

```
PS C:\> y:\CreateRegistryKey.ps1
DEBUG: 1+ <<<< y:\CreateRegistryKey.ps1
DEBUG: ! CALL function 'CreateRegistryKey.ps1' (defined in file 'y:\CreateRegistryKey.ps1')
DEBUG: 7+ Function Add-RegistryValue <<<< ($key,$value)
DEBUG: 25+ <<<< Add-RegistryValue -key forscripting -value test
DEBUG: ! CALL function 'Add-RegistryValue' (defined in file 'y:\CreateRegistryKey.ps1')
```

Przypisania do zmiennych są oznaczane symbolem `! SET`. Pierwsze jest przypisanie do zmiennej `$scriptRoot`.

```
DEBUG: 9+ $scriptRoot = <<<< "HKCU:\software\ForScripting"
DEBUG: ! SET $scriptRoot = 'HKCU:\software\ForScripting'.
DEBUG: 10+ if <<<< (-not (Test-Path -path $scriptRoot))
DEBUG: 18+ <<<< Set-ItemProperty -Path $scriptRoot -Name $key -Value $value | '
DEBUG: 2+ $foundSuggestion = <<<< $false
DEBUG: ! SET $foundSuggestion = 'False'.
DEBUG: 4+ if <<<< ($lastError -and
DEBUG: 15+ $foundSuggestion <<<<
PS C:\>
```

Na rysunku 19.7 pokazano efekt wykonania skryptu *CreateRegistryKey.ps1* przy ustawionym drugim poziomie śledzenia wykonywania skryptów.



RYSUNEK 19.7. Drugi poziom śledzenia wykonywania skryptów dodaje informacje o przypisaniach zmiennych, wywołaniach funkcji i wywołaniach skryptów zewnętrznych

Wykonywanie skryptu krok po kroku

Obserwacja wykonywania kolejnych linii kodu skryptu może dostarczyć cennych informacji pozwalających rozwiązać niejedyn problem. Jeżeli skrypt jest dość skomplikowany i zawiera kilka funkcji, to najprostsza forma śledzenia może być niewystarczająca do jego zdiagnozowania. W takich przypadkach najlepszym rozwiązaniem jest wykonywanie skryptu krok po kroku. W trybie tym przed wykonaniem każdego wiersza kodu konsola wyświetla pytanie, czy go wykonać. Proces ten przeanalizujemy na przykładzie skryptu *BadScript.ps1*, którego kod źródłowy znajduje się poniżej.

BadScript.ps1

```
Function AddOne([int]$num)
{
    $num+1
} #end funkcja AddOne

Function AddTwo([int]$num)
{
    $num+2
} #end funkcja AddTwo

Function SubOne([int]$num)
{
    $num-1
} #end funkcja SubOne

Function TimesOne([int]$num)
{
    $num*2
} #end funkcja TimesOne

Function TimesTwo([int]$num)
{
    $num*2
} #end funkcja TimesTwo

Function DivideNum([int]$num)
{
    12/$num
} #end funkcja DivideNum

# *** punkt początkowy skryptu ***

$num = 0
SubOne($num) | DivideNum($num)
AddOne($num) | AddTwo($num)
```

Skrypt *BadScript.ps1* zawiera kilka funkcji do sumowania, odejmowania, mnożenia i dzielenia liczb. Niestety zawiera parę błędów, przez co nie działa tak, jak powinien. Teoretycznie można by było ustawić drugi poziom śledzenia wykonywania skryptu, ale przy tej liczbie funkcji i takim rodzaju błędów wykrycie istoty problemów mogłoby być trudne. Przy włączaniu trybu wykonywania skryptów krok po kroku domyślnie ustawiony jest też pierwszy poziom śledzenia i w większości przypadków jest to najlepszy możliwy wybór.

Dlatego dobrym rozwiązaniem prowadzącym do zbadania skryptu *BadScript.ps1* może być wykonanie kodu linijka po linijce. Ta metoda diagnostyczna ma dwie zalety. Po pierwsze: pozwala obserwować, co się dzieje podczas wykonywania każdej linijki kodu, co umożliwia dokładne zbadanie całego procesu. Podczas śledzenia można przeoczyć cenne informacje, ponieważ wszystko zostaje wyświetlone naraz. Natomiast w przypadku wykonywania kodu krok po kroku wykonanie każdego wiersza kodu trzeba zatwierdzić. Do wyboru jest kilka możliwości, a domyślna opcja to *Y* — oznaczająca, że chcemy kontynuować wykonywanie operacji. Jeśli wybierzemy domyślną opcję, w konsoli zostaną wyświetlone dane diagnostyczne. Są to takie same informacje, jakie wyświetla tryb śledzenia, i ilość szczegółów również jest zależna od ustawionego poziomu śledzenia. Poniżej znajduje się wynik wykonania opisywanego skryptu krok po kroku:

```
PS C:\> Set-PSDebug -Step
PS C:\> C:\Users\Łukasz\Documents\BadScript.ps1

Continue with this operation?
1+ >>>> C:\Users\Łukasz\Documents\BadScript.ps1
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help (default is "Y"): y
DEBUG: 1+ >>>> C:\Users\Łukasz\Documents\BadScript.ps1

Continue with this operation?
42+ >>>> $num = 0
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help (default is "Y"): y
DEBUG: 42+ >>>> $num = 0

Continue with this operation?
43+ >>>> SubOne($num) | DivideNum($num)
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help (default is "Y"): y
DEBUG: 43+ >>>> SubOne($num) | DivideNum($num)

Continue with this operation?
21+ >>>> {
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help (default is "Y"): y
DEBUG: 21+ >>>> {

Continue with this operation?
22+ >>>> $num-1
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help (default is "Y"): y
DEBUG: 22+ >>>> $num-1

Continue with this operation?
23+ >>>> } #end funkcja SubOne
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help (default is "Y"): y
DEBUG: 23+ >>>> } #end funkcja SubOne

Continue with this operation?
36+ >>>> {
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help (default is "Y"): y
DEBUG: 36+ >>>> {

Continue with this operation?
37+ >>>> 12/$num
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help (default is "Y"): y
DEBUG: 37+ >>>> 12/$num

Continue with this operation?
19+ if ( & >>>> { Set-StrictMode -Version 1;
```



```

$.PSMessageDetails } ) {
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help (default is "Y"): y
DEBUG: 19+ if ( & >>>> { Set-StrictMode -Version 1;
$.PSMessageDetails } ) {

Continue with this operation?
19+ if ( & { >>>> Set-StrictMode -Version 1;
$.PSMessageDetails } ) {
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help (default is "Y"): y
DEBUG: 19+ if ( & { >>>> Set-StrictMode -Version 1;
$.PSMessageDetails } ) {

Continue with this operation?
19+ if ( & { Set-StrictMode -Version 1; >>>>
$.PSMessageDetails } ) {
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help (default is "Y"): y
DEBUG: 19+ if ( & { Set-StrictMode -Version 1; >>>>
$.PSMessageDetails } ) {

Continue with this operation?
1+ & >>>> { Set-StrictMode -Version 1; $this.Exception.InnerException.PSMessageDetails }
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help (default is "Y"): y
DEBUG: 1+ & >>>> { Set-StrictMode -Version 1; $this.Exception.InnerException.PSMessageDetails }

Continue with this operation?
1+ & { >>>> Set-StrictMode -Version 1; $this.Exception.InnerException.PSMessageDetails }
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help (default is "Y"): y
DEBUG: 1+ & { >>>> Set-StrictMode -Version 1; $this.Exception.InnerException.PSMessageDetails }

Continue with this operation?
1+ & { Set-StrictMode -Version 1; >>>> $this.Exception.InnerException.PSMessageDetails }
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help (default is "Y"): y
DEBUG: 1+ & { Set-StrictMode -Version 1; >>>> $this.Exception.InnerException.PSMessageDetails }

Continue with this operation?
1+ & { Set-StrictMode -Version 1; $this.Exception.InnerException.PSMessageDetails >>>> }
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help (default is "Y"): y
DEBUG: 1+ & { Set-StrictMode -Version 1; $this.Exception.InnerException.PSMessageDetails >>>> }

Continue with this operation?
19+ if ( & { Set-StrictMode -Version 1; $.PSMessageDetails
>>>> } ) {
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help (default is "Y"): y
DEBUG: 19+ if ( & { Set-StrictMode -Version 1;
$.PSMessageDetails >>>> } ) {

Continue with this operation?
26+ $errorCategoryMsg = & >>>> { Set-StrictMode -Version 1;
$.ErrorCategory_Message }
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help (default is "Y"): y
DEBUG: 26+ $errorCategoryMsg = & >>>> { Set-StrictMode -
Version
1; $.ErrorCategory_Message }

Continue with this operation?
26+ $errorCategoryMsg = & { >>>> Set-StrictMode -Version 1;
$.ErrorCategory_Message }
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help (default is "Y"): y

```

```

DEBUG: 26+                                     $errorCategoryMsg = & { >>>> Set-StrictMode -
Version
1; $_.ErrorCategory_Message }

Continue with this operation?
26+                                     $errorCategoryMsg = & { Set-StrictMode -Version 1; >>>>
$_.ErrorCategory_Message }
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help (default is "Y"): y
DEBUG: 26+                                     $errorCategoryMsg = & { Set-StrictMode -Version 1;
>>>> $_.ErrorCategory_Message }

Continue with this operation?
26+                                     $errorCategoryMsg = & { Set-StrictMode -Version 1;
$_.ErrorCategory_Message >>>> }
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help (default is "Y"): y
DEBUG: 26+                                     $errorCategoryMsg = & { Set-StrictMode -Version 1;
$_.ErrorCategory_Message >>>> }

Continue with this operation?
42+                                     $originInfo = & >>>> { Set-StrictMode -Version 1;
$_.OriginInfo }
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help (default is "Y"): y
DEBUG: 42+                                     $originInfo = & >>>> { Set-StrictMode -Version 1;
$_.OriginInfo }

Continue with this operation?
42+                                     $originInfo = & { >>>> Set-StrictMode -Version 1;
$_.OriginInfo }
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help (default is "Y"): y
DEBUG: 42+                                     $originInfo = & { >>>> Set-StrictMode -Version 1;
$_.OriginInfo }

Continue with this operation?
42+                                     $originInfo = & { Set-StrictMode -Version 1; >>>>
$_.OriginInfo }
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help (default is "Y"): y
DEBUG: 42+                                     $originInfo = & { Set-StrictMode -Version 1; >>>>
$_.OriginInfo }

Continue with this operation?
42+                                     $originInfo = & { Set-StrictMode -Version 1; $_.OriginInfo
>>>> }
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help (default is "Y"): y
DEBUG: 42+                                     $originInfo = & { Set-StrictMode -Version 1;
$_.OriginInfo >>>> }
Nastąpiła próba podzielenia przez zero.
At C:\Users\Łukasz\Documents\BadScript.ps1:37 char:2
+ 12/$num
+ ~~~~~
+ CategoryInfo          : NotSpecified: (:) [], RuntimeException
+ FullyQualifiedErrorId : RuntimeException

Continue with this operation?
38+ >>>> } #end funkcja DivideNum
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help (default is "Y"): y
DEBUG: 38+ >>>> } #end funkcja DivideNum

Continue with this operation?
44+ >>>> AddOne($num) | AddTwo($num)

```

```
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help (default is "Y"): y
DEBUG: 44+ >>>> AddOne($num) | AddTwo($num)
```

Continue with this operation?

```
11+ >>>> {
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help (default is "Y"): y
DEBUG: 11+ >>>> {
```

Continue with this operation?

```
12+ >>>> $num+1
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help (default is "Y"): y
DEBUG: 12+ >>>> $num+1
```

Continue with this operation?

```
13+ >>>> } #end funkcja AddOne
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help (default is "Y"): y
DEBUG: 13+ >>>> } #end funkcja AddOne
```

Continue with this operation?

```
16+ >>>> {
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help (default is "Y"): y
DEBUG: 16+ >>>> {
```

Continue with this operation?

```
17+ >>>> $num+2
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help (default is "Y"): y
DEBUG: 17+ >>>> $num+2
2
```

Continue with this operation?

```
18+ >>>> } #end funkcja AddTwo
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help (default is "Y"): y
DEBUG: 18+ >>>> } #end funkcja AddTwo
PS C:\>
```

Drugą korzyścią z używania parametru `step` polecenia `Set-PSDebug` jest możliwość zawieszenia wykonywania skryptu, wykonania dodatkowych poleceń Windows PowerShell i wrócenia do dalszego wykonywania tego skryptu. Zwracając wartość zmiennej z konsoli Windows PowerShell, można zdobyć cenne informacje na temat tego, co robi skrypt. Jeśli w wierszu poleceń wpisze się literę *s* (ang. *suspend* — „zawieść”), nastąpi przejście do zagnieżdżonego wiersza poleceń. Można w nim pobrać wartość zmiennej w taki sam sposób jak w normalnej konsoli, tzn. wpisując jej nazwę — można nawet korzystać z opcji rozwijania nazw za pomocą klawisza *Tab*. Po pobraniu wartości zmiennej należy wpisać `exit`, aby wrócić do wykonywania skryptu.

Continue with this operation?

```
1+ >>>> C:\Users\Łukasz\Documents\BadScript.ps1
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help
(default is "Y"):y
DEBUG: 1+ >>>>
C:\Users\Łukasz\Documents\BadScript.ps1
```

Continue with this operation?

```
42+ >>>> $num = 0
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help
(default is "Y"):y
DEBUG: 42+ >>>> $num = 0
```

```
Continue with this operation?
43+ >>>> SubOne($num) | DivideNum($num)
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help
(default is "Y"):s
PS C:\>> $num
0
PS C:\>> exit
```

Jeśli chcesz zobaczyć, co się stanie, jeśli pozwolisz skryptowi na dokończenie działania bez przerw od aktualnego miejsca przerywania, możesz nacisnąć klawisz *A* (tak, dla wszystkich). Najczęściej z opcji tej korzysta się wtedy, gdy znajdzie się błąd. W tym przypadku na przykład można zauważyć, że skrypt próbuje wykonać dzielenie przez zero.

```
Continue with this operation?
50+ >>>> AddOne($num) | AddTwo($num)
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help
(default is "Y"):A
DEBUG: 50+ >>>> AddOne($num) | AddTwo($num)
2
PS C:\>
```

Po znalezieniu błędu można zmienić wartość zmiennej bezpośrednio w zawieszonym konsoli, aby sprawdzić, czy wyeliminuje to wszystkie pozostałe błędy. W tym celu należy ponownie uruchomić skrypt i w wierszu, w którym został wykryty błąd, nacisnąć klawisz *S*. W tym momencie należy bardzo uważnie przeczytać pojawiające się informacje. Gdy w poprzednim przykładzie naciśnięto klawisz *A*, skrypt został wykonany aż do linii 37. Numer wiersza znajduje się za dwukropkiem i nazwą skryptu. Znak *+* oznacza polecenie, którym w tym przypadku jest `12/$num`. Cztery strzałki w lewo oznaczają, że to wartość zmiennej `$num` powoduje problemy, jak pokazano poniżej:

Nastąpiła próba podzielenia przez zero.

```
At C:\Users\Łukasz\Documents\BadScript.ps1:37 char:2
+ 12/$num
+ ~~~~~
+ CategoryInfo          : NotSpecified: (:) [], RuntimeException
+ FullyQualifiedErrorId : RuntimeException
```

Należy wykonać kod krok po kroku aż do wiersza 37. Wiersz poleceń będzie wyglądał tak: `43+ 12/ <<<< $num`. Oznacza to, że jesteśmy w wierszu nr 37 i że teraz ma zostać wykonane działanie (+) dzielenia 12 przez wartość zapisaną w zmiennej `$num`. Teraz należy nacisnąć klawisz *S*, aby zawiesić wykonywanie skryptu i przejść do zagnieżdżonego wiersza poleceń Windows PowerShell. Następnie sprawdzamy wartość zmiennej `$num` i zmieniamy ją np. na 2. Potem wychodzimy z zagnieżdżonego wiersza poleceń i wracamy do wykonywania kodu krok po kroku. Powinniśmy wykonać kod do końca, aby sprawdzić, czy nie występują jeszcze jakieś inne problemy. Jeśli nie, to wiemy, że znaleźliśmy błąd.

```
Continue with this operation?
37+ >>>> 12/$num
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help
(default is "Y"):s
PS C:\>> $num
0
```

```
PS C:\>> $num = 2
PS C:\>> exit
```

Continue with this operation?

```
37+ >>>> 12/$num
```

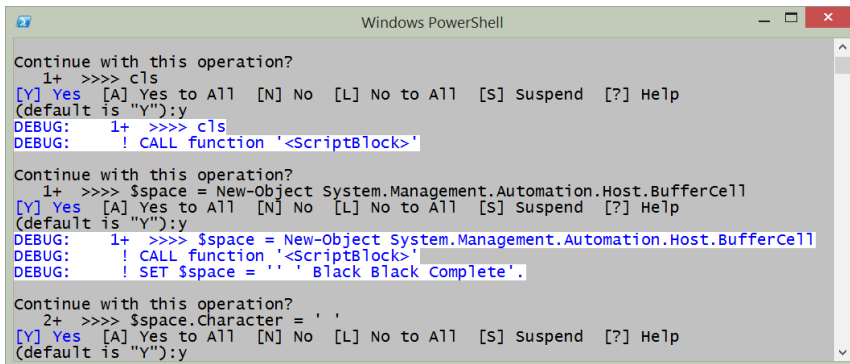
```
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help
(default is "Y"):y
```

```
DEBUG: 37+ >>>> 12/$num
```

```
6
```

Oczywiście znalezienie źródła problemu nie jest równoznaczne z jego rozwiązaniem, ale w poprzednim przykładzie wszystko wskazuje na to, że winna jest wartość zmiennej \$num. Dlatego naszą kolejną czynnością będzie przyjrzenie się sposobowi przypisywania wartości tej zmiennej.

Narzędzia do śledzenia wykonywania kodu polecenia Set-PSDebug mają parę denerwujących cech. Po pierwsze: trzeba przeglądać dodatkowe linijki zwracane przez funkcje debugujące. Komunikaty konsoli zajmują połowę okna. Jeśli wywoła się polecenie Clear-Host w celu skasowania zawartości okna, to przez kilka minut będzie trzeba się przebić przez jego diagnostykę. To samo dotyczy zmieniania poziomu śledzenia w trakcie diagnostyki. Domyślnie poziom śledzenia jest ustawiony na 1 przez parametr step polecenia Set-PSDebug. Drugi problem pojawia się wtedy, gdy chcemy pominąć jakieś polecenie ze skryptu. Po prostu nie da się tego zrobić. Zamiast tego sesja kończy się wyświetleniem błędu w konsoli Windows PowerShell, jak pokazano na rysunku 19.8.



```
Windows PowerShell

Continue with this operation?
1+ >>>> cls
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help
(default is "Y"):y
DEBUG: 1+ >>>> cls
DEBUG: ! CALL function '<ScriptBlock>'

Continue with this operation?
1+ >>>> $space = New-Object System.Management.Automation.Host.BufferCell
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help
(default is "Y"):y
DEBUG: 1+ >>>> $space = New-Object System.Management.Automation.Host.BufferCell
DEBUG: ! CALL function '<ScriptBlock>'
DEBUG: ! SET $space = '' Black Black Complete'.

Continue with this operation?
2+ >>>> $space.Character = ,
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help
(default is "Y"):y
```

RYSunek 19.8. Polecenie Set-PSDebug -step nie pozwala na pomijanie funkcji ani poleceń

Aby wyłączyć krokowe wykonywanie skryptu, należy użyć parametru off. Jednak to polecenie również trzeba będzie wykonać krok po kroku.

```
PS C:\> Set-PSDebug -Off
```

Continue with this operation?

```
1+ Set-PSDebug -Off
```

```
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help
(default is "Y"):y
```

```
DEBUG: 1+ Set-PSDebug -Off
```

```
PS C:\>
```

Włączanie trybu ścisłego

Jednym z łatwych do poprawienia błędów, które mogą zamienić debugowanie w koszmar, są zmienne. Często używa się ich niepoprawnie, gubi się je oraz niepoprawnie się je inicjuje. Łatwo też zrobić zwykłą literówkę przy wpisywaniu nazwy zmiennej. Literówki mogą też wywoływać wiele problemów w bardziej skomplikowanych skryptach.

Praca w trybie Set-PSDebug -strict

Poniżej znajduje się kod źródłowy skryptu *SimpleTypingError.ps1* zawierający prostą literówkę.

SimpleTypingError.ps1

```
$a = 2
$b = 5
$d = $a + $b
'Wartość zmiennej $c wynosi: ' + $c
```

Po uruchomieniu tego skryptu w konsoli pojawi się następujący wynik:

```
PS C:\> y:\SimpleTypingError.ps1
Wartość zmiennej $c wynosi:
PS C:\>
```

Wartość zmiennej \$c nie została wyświetlona. Jeśli ustawimy tryb ścisły za pomocą parametru `strict` polecenia `Set-PSDebug`, to konsola wygeneruje błąd informujący, że wartość zmiennej \$c nie istnieje.

```
PS C:\> Set-PSDebug -Strict
PS C:\> y:\SimpleTypingError.ps1
The variable $c cannot be retrieved because it has not been set yet.
At y:\SimpleTypingError.ps1:4 char:27
+ 'Wartość zmiennej $c wynosi: ' + $c <<<<
PS C:\>
```

Jeśli dobrze przyjrzyś się kodowi źródłowemu skryptu *SimpleTypingError.ps1*, zauważysz, że suma wartości zmiennych \$a i \$b została przypisana do zmiennej \$d, a nie do \$c. Rozwiązaniem problemu jest przypisanie tej sumy do zmiennej \$c (bo taki pewnie był plan). Polecenie `Set-PSDebug -Strict` można także umieścić w skrypcie, aby móc wykrywać niezainicjowane zmienne już w czasie pisania kodu i uniknąć późniejszych problemów.

Jeśli rutynowo wyświetlasz wartości zmiennych w łańcuchach rozwijanych (w podwójnym cudzysłowie), to pamiętaj, że znajdujące się w nich niezainicjowane zmienne nie są zgłaszane jako błąd. W skrypcie *SimpleTypingErrorNotReported.ps1* w łańcuchu takim rozwijana jest wartość zmiennej \$c. Jej pierwsze wystąpienie jest specjalnie oznaczone znakiem `'`, aby została wyświetlona sama nazwa, nie wartość. Drugie wystąpienie jest natomiast rozwijane. Poniżej znajduje się omawiana linijka kodu:

```
"Wartość zmiennej '$c' wynosi: $c"
```

Wynik wykonania skryptu *SimpleTypingErrorNotReported.ps1* w konsoli jest następujący:

```
PS C:\> Set-PSDebug -Strict
PS C:\> y:\SimpleTypingErrorNotReported.ps1
Wartość zmiennej $c wynosi:
PS C:\>
```

Poniżej znajduje się kompletny kod źródłowy skryptu *SimpleTypingErrorNotReported.ps1*:

SimpleTypingErrorNotReported.ps1

```
$a = 2
$b = 5
$d = $a + $b
"Wartość zmiennej '$c' wynosi: $c"
```

Aby wyłączyć tryb ścisły, należy wykonać polecenie Set-PSDebug -off.

Sposób użycia polecenia Set-StrictMode

Do ustawiania trybu ścisłego można też używać polecenia Set-StrictMode. Jego zaletą jest to, że rozpoznaje zakresy, tzn. podczas gdy polecenie Set-PSDebug zawsze działa globalnie, polecenia Set-StrictMode można użyć w funkcji, aby ograniczyć jego zakres działania właśnie do tej funkcji. Przy użyciu polecenia Set-StrictMode można zdefiniować dwa tryby działania. Pierwszy to wersja 1 i jest równoważny z ustawieniem Set-PSDebug -strict (wyjawszy rozpoznawanie zakresu dostępności). Poniżej pokazano przykład jego użycia:

```
PS C:\> Set-StrictMode -Version 1
PS C:\> y:\SimpleTypingError.ps1
The variable '$c' cannot be retrieved because it has not been set.
At y:\SimpleTypingError.ps1:4 char:28
+ 'Wartość zmiennej $c wynosi: ' + $c <<<<
    + CategoryInfo          : InvalidOperation: (c:Token) [], RuntimeException
    + FullyQualifiedErrorId : VariableIsUndefined
PS C:\>
```

Polecenie Set-StrictMode nie wykryje niezainicjowanej zmiennej w rozwijanym łańcuchu w skrypcie *SimpleTypingErrorNotDetected.ps1*.

W drugim trybie wymuszany jest sposób wywoływania funkcji jak metod. Skrypt *AddTwoError.ps1* przekazuje do funkcji add-two dwie wartości za pomocą notacji metodowej. Jako że notacja ta jest dozwolona przy wywoływaniu funkcji, normalnie nie jest zgłaszany żaden błąd. Jednak sposób przekazywania parametrów do funkcji tak jak do metod działa tylko wtedy, gdy do przekazania jest tylko jedna wartość. Aby przekazać dwie wartości, należy zastosować notację funkcyjną, jak pokazano poniżej:

```
add-two 1 2
```

Inny sposobem na poprawne wywołanie funkcji add-two jest dodanie do przekazywanych wartości nazw parametrów, których dotyczą, jak widać poniżej:

```
add-two -a 1 -b 2
```

W obu tych przypadkach otrzymalibyśmy poprawny wynik. Notacja metodowa spowoduje wyświetlenie niepoprawnego wyniku, ale też nie wygeneruje błędu. Jeśli funkcja zwraca niepoprawną wartość i nie powoduje jakiegokolwiek błędu, to może przysporzyć sporo kłopotów. Metodowa notacja wywoływania funkcji add-two jest użyta w skrypcie *AddTwoError.ps1* i została pokazana poniżej:

```
add-two(1,2)
```

Jeśli skrypt zostanie uruchomiony bez włączenia trybu Set-StrictMode -version 2, to w konsoli nie zostanie wyświetlony żaden błąd. Wynik będzie niejasny, bo suma zmiennych \$a i \$b po prostu nie zostanie wyświetlona, jak widać poniżej:

```
PS C:\> y:\AddTwoError.ps1
1
2
PS C:\>
```

Po włączeniu trybu Set-StrictMode -version 2 skrypt *AddTwoError.ps1* spowoduje wygenerowanie błędu informującego, że funkcja została wywołana tak, jakby była metodą. W wiadomości o błędzie zostanie podany numer wiersza, w którym wystąpił błąd, i zostanie pokazane, która funkcja go spowodowała. Przed wywołaniem funkcji będzie znajdował się znak + z nazwą funkcji i czterema strzałkami wskazującymi, co zostało przekazane do funkcji. Poniżej znajduje się cały opisywany komunikat o błędzie:

```
PS C:\> Set-StrictMode -Version 2
PS C:\> y:\AddTwoError.ps1
The function or command was called as if it were a method. Parameters should be separated by spaces. For information about parameters, see the about_Parameters Help topic.
At Y:\AddTwoError.ps1:7 char:8
+ add-two <<<< (1,2)
      + CategoryInfo          : InvalidOperation: (:) [], RuntimeException
      + FullyQualifiedErrorId : StrictModeFunctionCallWithParens
PS C:\>
```

Poniżej znajduje się kompletny kod źródłowy skryptu *AddTwoError.ps1*:

AddTwoError.ps1

```
Function add-two ($a,$b)
{
    $a + $b
}

add-two(1,2)
```

W trybie Set-StrictMode -Version 2 sprawdzane są następujące kwestie:

1. Odniesienia do niezainicjowanych zmiennych, zarówno bezpośrednie, jak i z łańcuchów.
2. Odniesienia do nieistniejących własności obiektów.
3. Wywołania funkcji tak jak metod.
4. Zmienne bez nazw.

W pierwszej wersji trybu ścisłego sprawdzane są tylko odwołania do niezainicjowanych zmiennych.

Jeżeli nie wiesz, czy wybrać wersję 2, 3, czy 4 (nie ma zmian), to najlepszym rozwiązaniem jest wpisanie wartości latest, jak pokazano poniżej:

```
Set-StrictMode -version latest
```

Problemem z tym rozwiązaniem może być to, że nie wiadomo, jaki wpływ ostatnie zmiany mogą mieć na skrypt. Dlatego generalnie najbezpieczniej jest używać wersji 1, 2, 3 lub 4, aby zapewnić konkretny rodzaj ochrony.

Zapiski praktyka

Debugowanie

Don Jones, MVP Microsoft Windows PowerShell

CEO, Concentrated Technologies

Często obserwuję, jak studenci męczą się z debugowaniem. Ale są dwie proste sztuczki, które znacznie to ułatwiają. Po pierwsze: z góry zakładam, że na pewno gdzieś popełnię błąd, więc od samego początku implementuję udogodnienia dotyczące debugowania. Polega to na dodaniu polecenia `Write-Debug` w każdym miejscu skryptu, w którym podejmowane są jakieś decyzje (abym wiedział, jakie działanie skrypt podejmie) lub w którym zmieniana jest zawartość jakiejś zmiennej, oraz wszędzie tam, gdzie używane są wartości jakichś własności. W razie potrzeby zamiast `Write-Debug` można używać polecenia `Write-Verbose`, a niektórzy tworzą nawet specjalne funkcje automatycznie dodające punkty wstrzymania. Bez względu na sposób ich uzyskania dane diagnostyczne powinny przede wszystkim spełniać swoją funkcję: pozwalać weryfikować założenia programisty. Błędy logiczne prawie zawsze wynikają z błędnych założeń co do zawartości zmiennych lub własności. Jeśli potrafisz powiedzieć, co według Ciebie **powinny** one zawierać, to na podstawie danych diagnostycznych będziesz mógł zweryfikować to założenie i ewentualnie je poprawić. Jeśli na przykład ktoś wysyła zapytanie do obiektu klasy `WMI Win32_LogicalDisk` i **zakłada**, że własność `DriveType` zawiera wartość w rodzaju "Fixed" albo "Removable", a otrzymuje wartość typu 2 lub 3, to od razu wie, że jego założenia były błędne. Gdy zaczynam debugowanie skryptu, zwłaszcza nienapisanego przeze mnie, to często najpierw drukuję go na kartce i wykonuję na próbę w myślach. Zapisuję, jakich wartości zmiennych i własności **się spodziewam**, i dodaję polecenia `Write-Debug`, aby zweryfikować te założenia. Jeśli moje założenia są błędne, to zazwyczaj udaje mi się to wykryć.

Debugowanie skryptów

Przy dostępnych w Windows PowerShell 4.0 narzędziach diagnostycznych polecenie `Set-PSDebug` wydaje się bardzo uproszczone czy wręcz nieporęczne. Gdy już je poznasz, to prawdopodobnie nigdy więcej nie spojrzysz na polecenie `Set-PSDebug`. Jest kilka poleceń, za pomocą których można włączyć debugowanie zarówno z poziomu konsoli Windows PowerShell, jak i środowiska Windows PowerShell ISE.

W tabeli 19.2 znajduje się zestawienie poleceń diagnostycznych.

TABELA 19.2. Polecenia diagnostyczne Windows PowerShell

| Nazwa polecenia | Opis |
|----------------------|---|
| Set-PSBreakpoint | Ustawia punkty wstrzymania na wierszach kodu, zmiennych i poleceniach |
| Get-PSBreakpoint | Pobiera punkty wstrzymania z bieżącej sesji |
| Disable-PSBreakpoint | Wyłącza punkty wstrzymania w bieżącej sesji |
| Enable-PSBreakpoint | Ponownie włącza punkty wstrzymania w bieżącej sesji |
| Remove-PSBreakpoint | Usuwa punkty wstrzymania z bieżącej sesji |
| Get-PSCallStack | Wyświetla bieżący stos wywołań |

Zapiski praktyka

Debugowanie w Windows PowerShell

Andy Schneider, inżynier systemów

Właściciel bloga Get-PowerShell

Zawsze ciekawiła mnie etymologia słów. Początek wykorzystywania angielskich słów *bug* (dosł. „robak”) i *debugging* w kontekście błędów i ich usuwania (debugowania) datuje się na lata 1940 i przypisuje admirał Grace Hopper.

Podczas pracy nad komputerem Mark II w Harvard University asystenci zauważyli, że do przekaźnika przykleiła się ćma, która powodowała wadliwe działanie systemu. Grace Hopper zanotowała, że trzeba było „odrobaczyć” („zdebugować”) system. — Wikipedia

Gdy byłem początkującym programistą skryptów, debugowanie wydawało mi się czymś bardzo trudnym i żmudnym. Ale szybko odkryłem, że wystarczy wykonać parę prostych czynności i trochę pomyśleć, aby znaleźć błąd w prawie każdym skrypcie.

W 99% przypadków, aby usunąć błędy ze skryptu, wystarczy podejrzeć wartość jakiejś zmiennej w odpowiednim momencie. Czy kiedykolwiek zdarzyło Ci się, że po napisaniu funkcji myślałeś sobie: „Gdybym tylko wiedział, jaka jest wartość x, zanim zostanie przetworzona przez y...”? Gdy diagnozowałem usterki związane z siecią, mój były szef mówił mi, abym „był bitem”. Trzeba dokładnie wiedzieć, skąd się przyszło i jaki będzie następny krok. Podobnie jest z debugowaniem kodu, tylko trzeba „być zmienną”.

Konsola Windows PowerShell 4.0 zawiera doskonałe narzędzie do podglądania zmiennych — punkty wstrzymania. Umożliwiają one wstrzymanie wykonywania kodu w dowolnym miejscu i zbadanie różnych wartości w tym czasie. W Windows PowerShell ISE obsługa punktów wstrzymania jest bardzo łatwa. Można je ustawiać w dowolnej linijce za pomocą opcji *Toggle Breakpoint* (włącz/wyłącz punkt wstrzymania) z menu *Debug* (diagnostyka) lub przy użyciu klawisza *F9*.

Jedną z cech punktów wstrzymania, które były dla mnie zaskakujące, jest to, że gdy ustawi się punkt wstrzymania w wybranym wierszu kodu w Windows PowerShell ISE, to cały ten wiersz zostaje wyróżniony, ale nie jest on wykonywany. Czyli wykonywanie skryptu zatrzyma się na początku tego wiersza. To znaczy, że punkty wstrzymania należy ustawiać bezpośrednio za miejscem, do którego chce się wykonać kod. Potem resztę kodu można wykonać krok po kroku za pomocą funkcji *Step Into* (wkrocz).

Inną przydatną opcją jest ustawianie punktów wstrzymania na zmiennych, tzn. zamiast ustawiać punkt wstrzymania na początku np. linii nr 45, można utworzyć punkt wstrzymania zatrzymujący wykonywanie skryptu w momencie dojścia do określonej zmiennej. Do ustawiania takich punktów wstrzymania służy polecenie `Set-PSBreakpoint` z parametrem `variable`. Należy tylko pamiętać, że nazwa zmiennej powinna być bez znaku dolara, np. `var` (nie `$var`).

Kolejną kwestią, która początkowo sprawiała mi problemy, było posługiwanie się zagnieżdżonym wierszem poleceń pojawiającym się po dojściu do punktu wstrzymania. Można wpisać `?` lub `h`, aby wyświetlić instrukcję obsługi tej „minipowłoki”. Swoją drogą to ciekawe, że w zagnieżdżonym wierszu poleceń pomoc można wyświetlić za pomocą znaku `?`, który normalnie jest aliasem polecenia `Where-Object`. W zagnieżdżonym wierszu poleceń dostępne są skróty do wszystkich poleceń z menu *Debug* ISE.

Najważniejsze, aby nie przestraszyć się debugowania. Dzięki metodycznemu podejściu do problemu i przy użyciu narzędzi dostępnych w Windows PowerShell 4.0 szybko znajdziesz nękające Cię błędy.

Ustawianie punktów wstrzymania

Funkcje diagnostyczne konsoli Windows PowerShell bazują na punktach wstrzymania, które są bardzo dobrze znane każdemu, kto programował np. w środowisku Microsoft Visual Studio. Ale wielu informatykom bez doświadczenia programistycznego pojęcie punktu wstrzymania jest obce. Punkt wstrzymania to miejsce, w którym powinno zostać wstrzymane wykonywanie tego skryptu. Jest to więc coś podobnego do wykonywania kodu krok po kroku, o którym była mowa wcześniej. Tylko w tym przypadku sami kontrolujemy, w którym miejscu nastąpi przerwa, więc cały proces wykonywania skryptu trwa dłużej. Ponadto punktów wstrzymania można używać na wiele sposobów (nie tylko do wykonywania kodu linijka po linijce) dających szansę na zdobycie potrzebnych informacji.

Ustawianie punktu wstrzymania na wybranym wierszu kodu

Do ustawiania punktów wstrzymania służy polecenie `Set-PSBreakpoint`. Najłatwiej jest ustawić punkt wstrzymania na pierwszym wierszu kodu skryptu. W tym celu należy użyć parametrów `line 1 -script`. Ustawienie punktu wstrzymania powoduje zwrócenie obiektu klasy `.NET System.Management.Automation.LineBreak` i wyświetlenie wartości własności `ID`, `Script` oraz `Line` przypisanych podczas tworzenia punktu wstrzymania.

```
PS C:\> Set-PSBreakpoint -line 1 -script Y:\BadScript.ps1
```

| ID | Script | Line | Command | Variable | Action |
|----|---------------|------|---------|----------|--------|
| -- | ----- | ---- | ----- | ----- | ----- |
| 0 | BadScript.ps1 | 1 | | | |

Taki punkt wstrzymania powoduje natychmiastowe przerwanie wykonywania skryptu. Potem można wykonywać funkcję krok po kroku w taki sam sposób jak przy użyciu polecenia Set-PSDebug z parametrem step. Po uruchomieniu skryptu następuje jego zatrzymanie na punkcie wstrzymania ustawionym w pierwszym wierszu kodu i uruchomienie debugera. Windows PowerShell uruchamia debugger za każdym razem, gdy skrypt *BadScript.ps1* jest uruchamiany z dysku Y. Gdy debugger jest włączony, na początku linijki wiersza poleceń pojawia się napis [DBG]: PS C:\>>>. Aby przejść do następnej linijki skryptu, należy nacisnąć klawisz S. Aby zakończyć sesję debugowania, należy nacisnąć klawisz Q. W poleceniach diagnostycznych wielkość liter nie ma znaczenia.

```
PS C:\> Y:\BadScript.ps1
Hit Line breakpoint on 'Y:\BadScript.ps1:1'
BadScript.ps1:1 #
-----
[DBG]: PS C:\>>> s
BadScript.ps1:16 Function AddOne([int]$num)
[DBG]: PS C:\>>> s
BadScript.ps1:21 Function AddTwo([int]$num)
[DBG]: PS C:\>>> s
BadScript.ps1:26 Function SubOne([int]$num)
[DBG]: PS C:\>>> s
BadScript.ps1:31 Function TimesOne([int]$num)
[DBG]: PS C:\>>> s
BadScript.ps1:36 Function TimesTwo([int]$num)
[DBG]: PS C:\>>> s
BadScript.ps1:41 Function DivideNum([int]$num)
[DBG]: PS C:\>>> s
BadScript.ps1:48 $num = 0
[DBG]: PS C:\>>> s
BadScript.ps1:49 SubOne($num) | DivideNum($num)
[DBG]: PS C:\>>> s
BadScript.ps1:28 $num-1
[DBG]: PS C:\>>> s
BadScript.ps1:43 12/$num
[DBG]: PS C:\>>> s
if ($_.FullyQualifiedErrorId -ne
"NativeCommandErrorMessage" -and $ErrorView -ne "CategoryView")
{[DBG]: PS C:\>>> q
PS C:\>
```

Morał

Ustawiając punkty wstrzymania, należy pamiętać, że są one zależne od lokalizacji skryptów. Gdy ustawia się punkt wstrzymania w konkretnym skrypcie, należy podać miejsce przechowywania tego skryptu. Jako że posługuję się systemem kontroli wersji, niektóre skrypty przechowuję w kilku miejscach. Przez to czasami się mylę, gdy podczas debugowania przez przypadek otworzę nie tę wersję skryptu, co trzeba. I wtedy diagnostyka nie działa. Najgorsze jest to, że jeśli skrypty nie różnią się niczym oprócz lokalizacji, to nie występuje żadna awaria. Jeśli chcesz ustawić jeden wspólny punkt wstrzymania dla skryptu przechowywanego w wielu miejscach, możesz ustawić punkt wstrzymania dla określonych warunków w konsoli Windows PowerShell, tylko wtedy nie używaj parametru -script.

Ustawianie punktu wstrzymania na zmiennej

Ustawienie punktu wstrzymania na pierwszej linii kodu jest dobrym sposobem na włączenie sesji diagnostycznej, ale w wykrywaniu przyczyn problemów często bardziej przydatne są punkty wstrzymania na zmiennych. Oczywiście dotyczy to przypadków, gdy już wiadomo, że problem w ogóle wiąże się ze zmienną. Punkty wstrzymania na zmiennych można definiować w trzech trybach, których opis znajduje się w tabeli 19.3.

TABELA 19.3. Tryby dostępu do punktów wstrzymania na zmiennych

| Tryb dostępu | Opis |
|--------------|--|
| Write | Zatrzymuje wykonywanie bezpośrednio przed zapisaniem nowej wartości w zmiennej |
| Read | Zatrzymuje wykonywanie po odczytaniu zmiennej, tzn. gdy jej wartość zostaje pobrana w celu przypisania, wyświetlenia lub użycia. W trybie tym wykonywanie nie zostaje wstrzymane, gdy zmienia się wartość zmiennej |
| Readwrite | Zatrzymuje wykonywanie w momencie odczytu lub zapisu zmiennej |

Aby dowiedzieć się, w którym momencie skrypt *BadScript.ps1* zapisuje wartość w zmiennej \$num, można użyć trybu write. W wartości parametru variable nie należy używać znaku dolara na początku nazwy zmiennej. Aby ustawić punkt wstrzymania na zmiennej, należy tylko podać ścieżkę do skryptu, nazwę zmiennej oraz tryb dostępu. Po ustawieniu punktu wstrzymania na zmiennej następuje zwrócenie obiektu klasy .NET System.Management.Automation.LineBreak zawierającego wartość trybu dostępu. Jest tak nawet w przypadku bezpośredniego dostępu do punktu wstrzymania za pomocą polecenia Get-PSBreakpoint. Jeśli obiekt System.Management.Automation.LineBreak przekaze się potokowo do polecenia Format-List, to można się przekonać, że zawiera on własność trybu dostępu. W poniższym przykładzie ustawiono punkt wstrzymania na zapisie wartości w zmiennej \$num ze skryptu Y:\BadScript.ps1.

```
PS C:\> Set-PSBreakpoint -Variable num -Mode write -Script Y:\BadScript.ps1
```

| ID | Script | Line | Command | Variable | Action |
|----|---------------|------|---------|----------|--------|
| 3 | BadScript.ps1 | | | num | |

| ID | Script | Line | Command | Variable | Action |
|----|---------------|------|---------|----------|--------|
| 3 | BadScript.ps1 | | | num | |

```
PS C:\> Get-PSBreakpoint | Format-List * -Force
AccessMode : Write
Variable   : num
Action     :
Enabled    : True
HitCount   : 0
Id         : 3
Script     : Y:\BadScript.ps1
```

Jeśli uruchomisz skrypt po ustawieniu punktu wstrzymania (pod warunkiem że inne punkty wstrzymania zostały usunięte lub wyłączone, o czym będzie mowa nieco dalej), to w momencie dotarcia do tego punktu (czyli w chwili zapisu wartości w zmiennej \$num) nastąpi włączenie debugera Windows PowerShell. Następnie za pomocą polecenia s można wykonać kolejne operacje. W tym przykładzie ustawiony jest tylko jeden punkt wstrzymania — znajduje się w 48. wierszu, w którym wartość zmiennej \$num zostaje ustawiona na 0.

```
PS C:\> Y:\BadScript.ps1
Hit Variable breakpoint on 'Y:\BadScript.ps1:$num' (Write access)

BadScript.ps1:48 $num = 0
[DBG]: PS C:\>>> $num
[DBG]: PS C:\>>> Write-Host $num

[DBG]: PS C:\>>> s
BadScript.ps1:49 SubOne($num) | DivideNum($num)
[DBG]: PS C:\>>> $num
0
```

Aby ustawić punkt wstrzymania na operacji odczytu zmiennej, należy w parametrze `variable` przekazać nazwę zmiennej, w parametrze `-script` podać ścieżkę do skryptu oraz w parametrze `-Mode` zdefiniować tryb `read`.

```
PS C:\> Set-PSBreakpoint -Variable num -Script Y:\BadScript.ps1 -Mode read
ID Script          Line Command          Variable Action
--
4 BadScript.ps1    -----
num
```

Po uruchomieniu skryptu każdy przypadek odczytu zmiennej będzie powodował wstrzymanie wykonywania kodu. Każdy punkt wstrzymania zostanie odnotowany w konsoli przez napis `Hit Variable breakpoint` z wyszczególnieniem ścieżki do skryptu i trybu dostępu. W skrypcie *BadScript.ps1* wartość zmiennej \$num jest odczytywana kilka razy, więc poniżej pokazano tylko część wyników.

```
PS C:\> Y:\BadScript.ps1
Hit Variable breakpoint on 'Y:\BadScript.ps1:$num' (Read access)

BadScript.ps1:49 SubOne($num) | DivideNum($num)
[DBG]: PS C:\>>> s
Hit Variable breakpoint on 'Y:\BadScript.ps1:$num' (Read access)

BadScript.ps1:49 SubOne($num) | DivideNum($num)
[DBG]: PS C:\>>> s
BadScript.ps1:28 $num-1
[DBG]: PS C:\>>> s
Hit Variable breakpoint on 'Y:\BadScript.ps1:$num' (Read access)

BadScript.ps1:28 $num-1
[DBG]: PS C:\>>> s
```

Jeśli dla zmiennej \$num ze skryptu *BadScript.ps1* parametr `-Mode` zostanie ustawiony na `readwrite`, konsola wyświetli następującą informację zwrotną:

```
PS C:\> Set-PSBreakpoint -Variable num -Mode readwrite -Script Y:\BadScript.ps1
ID Script          Line Command          Variable Action
--
6 BadScript.ps1    -----
num
```

Gdy uruchomimy teraz skrypt (przy założeniu, że poprzednie punkty wstrzymania zostały wyłączone), jego wykonywanie będzie wstrzymywane przy każdej operacji odczytu i zapisu zmiennej \$num. Jeśli znuży Cię wpisywanie s i naciskanie klawisza *Enter*, możesz naciskać tylko *Enter*, co będzie powodowało powtarzanie poprzednio wpisanych poleceń s. Gdy debugger dojdzie do błędu w skrypcie, należy wpisać q, aby go zamknąć.

```
PS C:\> Y:\BadScript.ps1
Hit Variable breakpoint on 'Y:\BadScript.ps1:$num' (ReadWrite access)
```

```
BadScript.ps1:48 $num = 0
[DBG]: PS C:\>>> s
BadScript.ps1:49 SubOne($num) | DivideNum($num)
[DBG]: PS C:\>>>
Hit Variable breakpoint on 'Y:\BadScript.ps1:$num' (ReadWrite access)
```

```
BadScript.ps1:49 SubOne($num) | DivideNum($num)
[DBG]: PS C:\>>>
Hit Variable breakpoint on 'Y:\BadScript.ps1:$num' (ReadWrite access)
```

```
BadScript.ps1:49 SubOne($num) | DivideNum($num)
[DBG]: PS C:\>>>
BadScript.ps1:28 $num-1
[DBG]: PS C:\>>>
Hit Variable breakpoint on 'Y:\BadScript.ps1:$num' (ReadWrite access)
```

```
BadScript.ps1:28 $num-1
[DBG]: PS C:\>>>
BadScript.ps1:43 12/$num
[DBG]: PS C:\>>>
Hit Variable breakpoint on 'Y:\BadScript.ps1:$num' (ReadWrite access)
```

```
BadScript.ps1:43 12/$num
[DBG]: PS C:\>>>
```

```
if ($_.FullyQualifiedErrorId -ne
"NativeCommandErrorMessage" -and $ErrorView -ne "CategoryView") {
[DBG]: PS C:\>>> q
PS C:\>
```

W trybie readwrite punkt wstrzymania nie informuje o tym, czy operacja dotyczy odczytu, czy zapisu. Trzeba przyjrzeć się wykonywanemu kodowi, aby się tego dowiedzieć.

W wartości parametru -action można wpisać zwykły kod Windows PowerShell, który ma zostać wykonany w momencie napotkania punktu wstrzymania. Jeśli na przykład trzeba śledzić wartość zmiennej i w tym celu chcemy wyświetlać jej wartość w każdym punkcie wstrzymania, to można w parametrze -action zdefiniować polecenie Write-Host wyświetlające wartość tej zmiennej. W razie potrzeby można też dodać napis informujący, która to jest wartość interesującej nas zmiennej. W ten sposób można wykryć przypadki braku inicjacji zmiennych, bo łatwiej jest zauważyć konkretną wartość niż pustą linię. Poniżej znajduje się przykład zastosowania opisywanej techniki.

```
PS C:\> Set-PSBreakpoint -Variable num -Action { write-host "num = $num" ;
Break } -Mode readwrite -script Y:\BadScript.ps1
```

| ID Script | Line Command | Variable | Action |
|-----------|---------------|----------|------------------|
| -- | ---- | ----- | ----- |
| 5 | BadScript.ps1 | num | write-host "..." |

Jeśli po ustawieniu tego punktu wstrzymania uruchomimy skrypt, debugger Windows PowerShell wyświetli następujące informacje:

```
PS C:\> Y:\BadScript.ps1
num =
Hit Variable breakpoint on 'Y:\BadScript.ps1:$num' (ReadWrite access)

BadScript.ps1:48 $num = 0
[DBG]: PS C:\>>> s
BadScript.ps1:49 SubOne($num) | DivideNum($num)
[DBG]: PS C:\>>> s
Set-PSBreakpoint -Variable num -Action { write-host "num = $num" ; break }
-Mode readwrite -script Y:\BadScript.ps1
[DBG]: PS C:\>>> s
num = 0
Set-PSBreakpoint -Variable num -Action { write-host "num = $num" ; break }
-Mode readwrite -script Y:\BadScript.ps1
[DBG]: PS C:\>>> c
Hit Variable breakpoint on 'Y:\BadScript.ps1:$num' (ReadWrite access)

BadScript.ps1:49 SubOne($num) | DivideNum($num)
[DBG]: PS C:\>>>
```

Ustawianie punktu wstrzymania na poleceniu

Aby ustawić punkt wstrzymania na poleceniu, należy użyć parametru `-command`. Wykonywanie skryptu można przerwać na poleceniu cmdlet Windows PowerShell, funkcji lub skrypcie zewnętrznym. Przy ustawianiu punktów wstrzymania można też używać aliasów, ale wtedy wykonywanie skryptu będzie wstrzymywane tylko po napotkaniu przez debugger aliasu, a nie prawdziwej nazwy polecenia. Ponadto nie trzeba określać nazwy skryptu do przerywania działania. Jeśli nie poda się ścieżki do skryptu, debugger będzie włączany dla wszystkich skryptów w sesji Windows PowerShell. Każde wystąpienie polecenia `Foreach` spowoduje przerwanie wykonywania skryptu przez debugger. Jako że `Foreach` jest zarówno instrukcją językową, jak i aliasem polecenia `Foreach-Object`, możesz się zastanawiać, czy debugger będzie przerywał wykonywanie skryptu w obu tych przypadkach — odpowiedź brzmi: nie. Można ustawiać punkty wstrzymania na instrukcjach językowych, ale debugger nie zatrzyma się na ich odpowiednikach w postaci poleceń. Jak widać w poniższym przykładzie, debugger przerywa wykonywanie skryptu na aliasie `Foreach`, natomiast na poleceniu `Foreach-Object` nie:

```
PS C:\> Set-PSBreakpoint -Command foreach
ID Script          Line Command      Variable      Action
--
10          foreach

PS C:\> 1..3 | Foreach-Object { $_ }
1
2
3
PS C:\> 1..3 | foreach { $_ }
Hit Command breakpoint on 'foreach'

1..3 | foreach { $_ }
[DBG]: PS C:\>>> c
1
```


Hit Command breakpoint on 'foreach'

```
1..3 | foreach { $_ }
```

```
[DBG]: PS C:\>>> c
```

2

Hit Command breakpoint on 'foreach'

```
1..3 | foreach { $_ }
```

```
[DBG]: PS C:\>>> c
```

3

UWAGA

W konsoli Windows PowerShell istnieje możliwość tworzenia punktów wstrzymania na skróty przez pominięcie nazwy skryptu. Jeśli przy tworzeniu punktu wstrzymania nie zdefiniuje się parametru `-script`, to debugger będzie przerywał działanie każdego skryptu, w którym napotka określoną funkcję. Jest to dobry sposób na debugowanie różnych skryptów zawierających tę samą funkcję.

Przy tworzeniu punktu wstrzymania dla funkcji `DivideNum` ze skryptu *BadScript.ps1* można opuścić ścieżkę do skryptu, ponieważ funkcja ta występuje tylko w nim. Chociaż trzeba przyznać, że jest to pewne ułatwienie, to nie należy też zapominać, że takie postępowanie może być przyczyną nieporozumień. Gdyby prowadzono diagnostykę kilku skryptów naraz w jednej sesji Windows PowerShell, to po wyświetleniu listy zdefiniowanych punktów wstrzymania nie byłoby wiadomo, do którego skryptu niektóre z nich należą (oczywiście pomijając przypadek debugowania tej samej funkcji w wielu skryptach). Poniżej znajduje się przykład utworzenia punktu wstrzymania na funkcji `DivideNum`:

```
PS C:\> Set-PSBreakpoint -Command DivideNum
```

| ID Script | Line Command | Variable | Action |
|-----------|--------------|----------|--------|
| -- | ---- | ----- | ----- |
| 7 | DivideNum | | |

Gdy uruchomisz skrypt, nastąpi przerwanie jego działania w momencie dotarcia do funkcji `DivideNum`. Gdy skrypt *BadScript.ps1* dociera do funkcji `DivideNum`, zmienna `$num` ma wartość 0. Wkraczamy do tej funkcji i przypisujemy zmiennej `$num` wartość 2. Zostaje wyświetlony wynik 6 oraz następuje wykonanie działania `12/$num`. Następnie zostaje wywołana funkcja `AddOne` i wartość zmiennej `$num` z powrotem zmienia się na 0. To samo dotyczy wywołania funkcji `AddTwo`.

```
PS C:\> Y:\BadScript.ps1
```

Hit Command breakpoint on 'DivideNum'

```
BadScript.ps1:49 SubOne($num) | DivideNum($num)
```

```
[DBG]: PS C:\>>> s
```

```
BadScript.ps1:43 12/$num
```

```
[DBG]: PS C:\>>> $num
```

0

```
[DBG]: PS C:\>>> $num =2
```

```
[DBG]: PS C:\>>> s
```

6

```
BadScript.ps1:50 AddOne($num) | AddTwo($num)
```

```
[DBG]: PS C:\>>> s
```

```
BadScript.ps1:18 $num+1
```

```
[DBG]: PS C:\>>> $num
0
[DBG]: PS C:\>>> s
BadScript.ps1:23 $num+2
[DBG]: PS C:\>>> $num
0
[DBG]: PS C:\>>> s
2
PS C:\>
```

Wiedza tajemna

Najlepsze debugowanie

Juan Carlos Ruiz Lopez, Senior Premier Field Engineer
Microsoft Corporation, Hiszpania

Na szczęście (albo nieszczęście, zależy jak na to patrzeć) mam niewielkie doświadczenie w pracy z debuggerem Windows PowerShell. Jako że język skryptowy Windows PowerShell ma naprawdę duże możliwości, nie trzeba używać wielu pętli potrzebnych w innych językach programowania. Wiele poleceń zakulisowo wykonuje operacje pętlowe. Dzięki temu skrypty są mniej skomplikowane i wymagają mniej debugowania.

W innych językach skryptowych w pętlach często występują błędy zakresu, które bardzo trudno wykryć. W Windows PowerShell problemy tego rodzaju zdarzają się rzadko. Najlepsza diagnostyka to brak diagnostyki. Oczywiście nie zawsze da się uniknąć debugowania, np. gdy wywoła się jakąś funkcję w nieprawidłowy sposób. W większości tych sytuacji wystarczy dodać instrukcję `Write-Debug`, aby wyświetlić otrzymywane zmienne lub parametry. Można nawet utworzyć własną funkcję `MyDebug` ustawiającą kolory i formatowanie drukowanych informacji.

Mimo to zalecam nauczanie się obsługi punktów wstrzymania, ponieważ jest to łatwe, a narzędzie to bywa naprawdę bardzo przydatne. Może punkty wstrzymania nie są potrzebne bardzo często, ale biorąc pod uwagę, jak szybko można nauczyć się ich obsługi, warto to zrobić. Nawet jeśli nigdy nie używasz debugera Windows PowerShell, to przynajmniej pogłębisz swoją wiedzę na temat sposobu działania konsoli.

Moim ulubionym poleceniem diagnostycznym w Windows PowerShell jest `k`, które wywołuje polecenie `Get-PSCallStack`. Lubię je, ponieważ lubię wiedzieć, kto wywołuje poszczególne polecenia, co jest szczególnie ważne, gdy ma się zmienną bibliotekę skryptów, w której funkcje są przemieszczane między modułami.

Reagowanie na punkty wstrzymania

Gdy skrypt dociera do punktu wstrzymania, konsola przekazuje nam kontrolę nad wierszem poleceń. W debugerze można wpisać dowolne polecenie Windows PowerShell, a nawet wykonywać polecenia cmdlet, takie jak `Get-Process` czy `Get-Service`. Ponadto na punkcie wstrzymania można też używać kilku nowych poleceń diagnostycznych Windows PowerShell. Ich lista znajduje się w tabeli 19.4.

TABELA 19.4. Polecenia diagnostyczne Windows PowerShell

| Skrót klawiszowy | Polecenie | Opis |
|------------------|-----------------|--|
| s | Step-into | Wykonuje następne polecenie i wstrzymuje działanie |
| v | Step-over | Wykonuje następne polecenie, ale pomija funkcje i wywołania. Pomińnięte elementy są wykonywane, ale nie w debugerze |
| o | Step-out | Wychodzi z bieżącej funkcji i w przypadku zagnieżdżenia przechodzi do góry o jeden poziom. Jeżeli polecenie znajduje się w bloku głównym, wykonywanie jest kontynuowane do końca skryptu lub następnego punktu wstrzymania. Pomińnięte elementy są wykonywane, ale nie w debugerze |
| c | Continue | Kontynuuje wykonywanie skryptu do końca lub następnego punktu wstrzymania. Pomińnięte elementy są wykonywane, ale nie w debugerze |
| l | List | Wyświetla wykonywaną część skryptu. Standardowo wyświetla bieżącą linię kodu, pięć poprzednich linii oraz 10 następnych. Aby kontynuować operację, należy nacisnąć klawisz <i>Enter</i> |
| l <m> | List | Wyświetla 16 linii skryptu, zaczynając od linii o numerze <m> |
| l <m> <n> | List | Wyświetla <n> linii skryptu, zaczynając od linii o numerze <m> |
| q | Stop | Zatrzymuje wykonywanie skryptu i zamyka debuger |
| k | Get-PSCallStack | Wyświetla bieżący stos wywołań |
| <Enter> | Repeat | Powtarza ostatnie polecenie, jeśli było nim Step-into (s), Step-over (v) lub List (l). W przeciwnym przypadku oznacza zatwierdzenie |
| h lub ? | Help | Wyświetla pomoc |

Dzięki ustawieniu punktu wstrzymania na funkcji `DivideNum` po uruchomieniu skryptu *BadScript.ps1* następuje przerwanie jego działania na linii 49., czyli tej, która zawiera wywołanie funkcji `DivideNum`. Za pomocą polecenia s można przejść do następnej instrukcji i zatrzymać

działanie bezpośrednio przed jej wykonaniem. Polecenie / wyświetla listę pięciu poprzednich wierszy kodu ze skryptu *BadScript.ps1* i dziesięciu następnych.

```
PS C:\> Y:\BadScript.ps1
Hit Command breakpoint on 'Y:\BadScript.ps1:dividenum'

BadScript.ps1:49 SubOne($num) | DivideNum($num)
[DBG]: PS C:\>>> s
BadScript.ps1:43 12/$num
[DBG]: PS C:\>>> l
38: $num*2
39: } #end funkcja TimesTwo
40:
41: Function DivideNum([int]$num)
42: {
43: * 12/$num
44: } #end funkcja DivideNum
45:
46: # *** punkt początkowy skryptu ***
47:
48: $num = 0
49: SubOne($num) | DivideNum($num)
50: AddOne($num) | AddTwo($num)
51:
```

Po przejrzeniu kodu można wykonać polecenie o w celu wyjścia z funkcji *DivideNum*. Pozostały kod funkcji i tak zostanie wykonany, więc pojawi się informacja o błędzie dzielenia przez zero. Do następnego wiersza kodu wykonywalnego nie pojawią się już żadne monity. Instrukcja *v* pozwala przeskoczyć pozostałe funkcje skryptu, które mimo to będą wykonane. Wyniki zostaną wyświetlone w konsoli Windows PowerShell.

```
[DBG]: PS C:\>>> o
Attempted to divide by zero.
At Y:\BadScript.ps1:43 char:5
+ 12/ <<<< $num
    + CategoryInfo          : NotSpecified: (:) [], RuntimeException
    + FullyQualifiedErrorId : RuntimeException
BadScript.ps1:50 AddOne($num) | AddTwo($num)
[DBG]: PS C:\>>> v
2
PS C:\>
```

Wyświetlanie listy punktów wstrzymania

Po ustawieniu kilku punktów wstrzymania dobrze jest wiedzieć, gdzie są one tworzone. Przede wszystkim należy pamiętać, że punkty wstrzymania są przechowywane w środowisku Windows PowerShell, a nie w skryptach. Funkcje diagnostyczne nie powodują zmian w kodzie źródłowym, więc można ich używać bez obaw, że się uszkodzi skrypt. Ponieważ jednak w typowej sesji diagnostycznej w środowisku można ustawić wiele punktów wstrzymania, dobrze jest mieć możliwość wyświetlenia listy wszystkich posiadanych punktów. Służy do tego polecenie *Get-PSBreakpoint*.

```
PS C:\> Get-PSBreakpoint
ID Script          Line Command      Variable      Action
--
11 BadScript.ps1    11 dividenum
13 BadScript.ps1    13 if
3 BadScript.ps1     3 num
5 BadScript.ps1     5 num
6 BadScript.ps1     6 num
7 DivideNum
8 foreach
9 gps
10 foreach
PS C:\>
```

Aby dowiedzieć się, które punkty wstrzymania są aktualnie włączone, można wykonać polecenie `Where-Object` i przekazać wynik do polecenia `Get-PSBreakpoint`.

```
PS C:\> Get-PSBreakpoint | where { $_.enabled }

ID Script          Line Command      Variable      Action
--
11 BadScript.ps1    11 dividenum
PS C:\>
```

Natomiast wynik polecenia `Get-PSBreakpoint` można przekazać do polecenia `Format-Table`.

```
PS C:\> Get-PSBreakpoint |
Format-Table -Property id, script, command, variable, enabled -AutoSize
```

| Id | Script | Command | variable | Enabled |
|----|------------------|-----------|----------|---------|
| 11 | Y:\BadScript.ps1 | dividenum | | True |
| 13 | Y:\BadScript.ps1 | if | | False |
| 3 | Y:\BadScript.ps1 | | num | False |
| 5 | Y:\BadScript.ps1 | | num | False |
| 6 | Y:\BadScript.ps1 | | num | False |
| 7 | | DivideNum | | False |
| 8 | | foreach | | False |
| 9 | | gps | | False |
| 10 | | foreach | | False |

Jako że utworzenie sformatowanej tabeli punktów wstrzymania wymaga trochę pisania, a jest to bardzo przydatna rzecz, można rozważyć napisanie funkcji i dodanie jej do profilu lub własnego modułu diagnostycznego. Funkcja ta znajduje się w skrypcie `Get-EnabledBreakpointsFunction.ps1` i jest pokazana poniżej.

Get-EnabledBreakpointsFunction.ps1

```
Function Get-EnabledBreakpoints
{
    Get-PSBreakpoint |
    Format-Table -Property id, script, command, variable, enabled -AutoSize
}
```

*# *** punkt początkowy skryptu ****

```
Get-EnabledBreakpoints
```

Włączanie i wyłączanie punktów wstrzymania

Czasami podczas debugowania skryptu trzeba wyłączyć jakiś punkt wstrzymania, aby sprawdzić działanie skryptu. Służy do tego polecenie `Disable-PSBreakpoint`.

```
Disable-PSBreakpoint -id 0
```

Jeśli natomiast trzeba włączyć wyłączony punkt wstrzymania, można użyć polecenia `Enable-PSBreakpoint`.

```
Enable-PSBreakpoint -id 1
```

Dobrym zwyczajem jest wyłączanie i włączanie pojedynczych skryptów, aby sprawdzić działanie skryptu i ewentualnie wykryć przyczynę usterek. Do sprawdzania statusu poszczególnych punktów wstrzymania można używać opisanego w poprzednim podrozdziale polecenia `Get-PSBreakpoint`.

Zapiski praktyka

Debugowanie skryptów

Vasily Gusev, administrator systemów, MCSE: Security/Messaging, MCItP: Enterprise/Server administrator, Microsoft MVP: Windows PowerShell

Microsoft Corporation

W konsoli Windows PowerShell 4.0 dostępne są naprawdę bardzo przydatne narzędzia diagnostyczne. Najpierw przyjrzymy się poleceniu `Set-PSBreakpoint`, za pomocą którego można ustawiać punkty wstrzymania na wybranych wierszach skryptów, poleceniach, funkcjach i zmiennych.

Gdy na przykład prowadzę diagnostykę w Windows PowerShell, to zamiast dodawać polecenia `Write-Debug` do skryptów w celu wyświetlenia wartości zmiennych, ustawiam punkty wstrzymania na operacjach przypisywania im wartości. Jest to o wiele prostsze i nie wymaga oczyszczania kodu skryptu po zakończeniu pracy.

```
Set-PSBreakpoint -Variable var -Mode write
```

Gdy ustawię powyższy punkt wstrzymania w konsoli Windows PowerShell, to przed każdą zmianą wartości zmiennej `$var` konsola wstrzymuje wykonywanie poleceń i włącza tryb diagnostyczny. O tym, że konsola działa w trybie diagnostycznym, świadczy napis `[DBG]` na początku każdej liniiki w wierszu poleceń. W trybie tym można wykonywać wszystkie polecenia Windows PowerShell oraz przeglądać i zmieniać wartości zmiennych.

Ale największą zaletą trybu diagnostycznego jest dostępność w nim specjalnych poleceń diagnostycznych, takich jak `Step-into`, `Step-over` i `Step-out`, za pomocą których można przechodzić przez wykonywany kod bez wychodzenia z trybu diagnostycznego. Polecenie `Continue` powoduje wyjście z trybu diagnostycznego i wykonuje cały pozostały kod do końca. Natomiast polecenie `Quit` wyłącza debugger i zatrzymuje dalsze wykonywanie skryptu.

Ponadto w zestawie narzędzi diagnostycznych Windows PowerShell znajduje się jedno bardzo przydatne przy debugowaniu w wierszu poleceń — polecenie `List`. Standardowo wyświetla ono bieżącą pozycję debugera oraz pięć poprzednich i dziesięć następnych wierszy kodu skryptu.

Podczas pracy z debugerem w wierszu poleceń bardzo przydatny jest klawisz `Enter`, którego naciśnięcie powoduje ponowienie ostatniego wpisanego polecenia. Wystarczy więc raz wpisać np. polecenie `Step-into`, a potem naciskać tylko klawisz `Enter`, aby wykonywać je wielokrotnie.

Za pomocą poleceń `h` i `?` można wyświetlić listę wszystkich poleceń debugera wraz z opisami. Jest to przydatne, gdy trzeba sobie odświeżyć pamięć na temat dostępnych narzędzi.

Przy użyciu przełącznika `-command` polecenia `Set-PSBreakpoint` można ustawiać te same punkty wstrzymania w odniesieniu zarówno do zdarzeń wywołania poleceń, jak i funkcji. Nietrudno się domyślić, że parametry `line` i `column` służą do ustawiania punktów wstrzymania w treści skryptu. Oczywiście dodatkowo w parametrze `-script` należy określić, którego skryptu dotyczy.

Zamiast wstrzymywać wykonywanie skryptu i włączać debuger z każdym punktem wstrzymania, mogę powiązać automatyczne wykonanie prawie dowolnej czynności. Jeśli na przykład mam zdiagnozować jakiś długo działający skrypt, a nie mogę cały czas siedzieć przed komputerem, to mogę nakazać debugerowi wysłanie wszystkich zmiennych do pliku XML bez wstrzymywania działania skryptu.

```
Set-PSBreakpoint -Variable var -Mode write -Action {Get-Variable |
Export-Clixml C:\dump.clixml}
```

Potem mogę odczytać zawartość otrzymanego pliku XML za pomocą poniższego polecenia:

```
$Variables = Import-Clixml c:\dump.clixml
```

Ponadto przy tworzeniu punktu wstrzymania mogę określić wyrażenie warunkowe, które spowoduje wykonanie pewnych czynności, jeśli zostanie spełniony określony warunek. Na przykład poniższe polecenie ustawia punkt wstrzymania włączany tylko wtedy, gdy wartość zmiennej `$DebugIsOn` jest ustawiona na `$true`:

```
Set-PSBreakpoint -Variable var -Mode write -Action '
{if ($DebugIsOn){break}}
```

Utworzonymi punktami wstrzymania można zarządzać przy użyciu różnych poleceń z członem `PSBreakpoint` w nazwie. Ich nazw łatwo się domyślić. Na przykład poniższe polecenie usuwa wszystkie punkty wstrzymania z bieżącej sesji:

```
Get-PSBreakpoint | Remove-PSBreakpoint
```

Ponadto można wyłączać i włączać punkty wstrzymania, nie usuwając ich. Służą do tego odpowiednio polecenia `Disable-PSBreakpoint` i `Enable-PSBreakpoint`.

Usuwanie punktów wstrzymania

Po zakończeniu diagnozowania skryptu należy usunąć wszystkie utworzone punkty wstrzymania. Można to zrobić na dwa sposoby. Pierwszy polega po prostu na zamknięciu okna konsoli Windows PowerShell. Podczas gdy sposobem tym można łatwo wyczyścić środowisko,

nie zawsze jest to dobre rozwiązanie, ponieważ czasami w sesji zdefiniowane są zdalne sesje Windows PowerShell lub zmienne zawierające wyniki pewnych zapytań. W razie potrzeby punkty wstrzymania można też usunąć za pomocą polecenia `Remove-PSBreakpoint`. Ale niestety nie ma ono przełącznika `all` do usuwania wszystkich punktów wstrzymania naraz. Należy natomiast przekazać identyfikator usuwanego punktu wstrzymania w parametrze `-id`.

```
Remove-PSBreakpoint -id 3
```

Aby usunąć wszystkie punkty wstrzymania, można przekazać wynik polecenia `Get-PSBreakpoint` do polecenia `Remove-PSBreakpoint`.

```
Get-PSBreakpoint | Remove-PSBreakpoint
```

Aby usunąć punkty wstrzymania dotyczące tylko wybranego skryptu, można przepuścić wyniki przez polecenie `Where-Object`.

```
(Get-PSBreakpoint | Where-Object Script -eq "C:\Scripts\Test.ps1") |  
Remove-PSBreakpoint
```

Wiedza tajemna

Debugowanie skryptów przy użyciu środowiska Windows PowerShell ISE

**Osama Sajid, menedżer ds. programów: łatwość zarządzania systemem Windows
Microsoft Corporation**

Środowisko Windows PowerShell ISE zawiera zarówno edytor skryptów, jak i debugger. Jeśli wczyta się do niego plik skryptu (*.ps1*), to punkty wstrzymania w nim można ustawiać za pomocą klawisza *F9*. Gdy uruchomi się skrypt, jego wykonywanie zostanie wstrzymane w miejscu punktu wstrzymania i będzie można wykonać jedną z następujących typowych czynności diagnostycznych:

- Wykonanie wiersza kodu — opcja *Step Over* (*F10*).
- Wejście do funkcji — opcja *Step Into* (*F11*).
- Wykonanie reszty funkcji i wyjście z niej (*Shift+F11*).

Naciśnięcie klawisza *F5* powoduje wykonywanie skryptu do następnego punktu wstrzymania lub do końca. Wszystkie wymienione polecenia diagnostyczne są dostępne w menu *Debug*.

Gdy działanie debugera jest wstrzymane na punkcie wstrzymania, to można podejrzeć wartość wybranej zmiennej, umieszczając nad nią kursor. Dodatkowo w okienku poleceń pojawiają się znaki *>>>* oznaczające, że aktywny jest tryb diagnostyczny i można wykonywać polecenia, np. ustawiające lub pobierające wartości zmiennych.

Ustawienie punktu wstrzymania na wybranym wierszu kodu jest najprostszą i najczęściej stosowaną metodą diagnostyczną. Ale czasami trzeba wstrzymać wykonywanie skryptu w momencie zmiany wartości określonej zmiennej albo wywołania określonego polecenia.

Wprawdzie w interfejsie Windows PowerShell ISE nie ma specjalnych opcji do tworzenia takich punktów wstrzymania, ale można sobie poradzić, stosując poniższe polecenie:

```
Set-PSBreakpoint -variable val -Mode ReadWrite
```

Polecenie to ustawia punkt wstrzymania na zmiennej o nazwie `val` oraz przy użyciu parametru `mode` o wartości `ReadWrite` zatrzymuje wykonywanie w momencie odczytu wartości tej zmiennej i przed jej zmianą.

Poniższe polecenie ustawia punkt wstrzymania na wykonaniu polecenia `Get-Process`. Gdy konsola dojdzie do niego, przekaże sterowanie do debugera.

```
Set-PSBreakpoint -command Get-Process
```

Inną ciekawą funkcją debugera Windows PowerShell jest możliwość wykonywania skryptów w punktach wstrzymania.

```
Set-PSBreakpoint -Variable val -Mode Read -Action '{Write-Host "Uwaga: wartość X = $x"; if($val-eq 5){break}}'
```

Debugowanie w środowisku Windows PowerShell ISE jest bardzo łatwe, chociaż równie dobrze można je prowadzić tylko w wierszu poleceń, bez użycia menu i skrótów klawiszowych. Na przykład listę punktów wstrzymania można wyświetlić za pomocą polecenia `Get-PSBreakpoint`, a wyłączyć punkt wstrzymania można za pomocą polecenia `Disable-PSBreakpoint`. Gdy wykonywanie skryptu zatrzymuje się na punkcie wstrzymania, w wierszu poleceń debugera dostępne stają się następujące polecenia:

- `s` — *Step-Into*
- `v` — *Step-over*
- `o` — *Step-out*
- `c` — *Continue*
- `q` — *Stop*

Więcej informacji na temat poleceń diagnostycznych można znaleźć w temacie *about_debuggers* w pomocy do Windows PowerShell.

Dodatkowe źródła informacji

- Centrum skryptów portalu TechNet pod adresem <http://www.microsoft.com/technet/scriptcenter> zawiera wiele przykładów skryptów diagnostycznych.
- Wszystkie skrypty opisane w tym rozdziale są dostępne do pobrania w repozytorium centrum skryptów portalu TechNet pod adresem <http://gallery.technet.microsoft.com/scriptcenter/PowerShell-40-Best-d9e16039>.

Rozdział 20

Praca w środowisku Windows PowerShell ISE

- Uruchamianie środowiska Windows PowerShell ISE
- Sposób użycia fragmentów kodu w środowisku Windows PowerShell ISE
- Dodatkowe źródła informacji

Windows PowerShell ISE to zintegrowane środowisko skryptowe wyposażone w funkcje kończenia nazw za pomocą klawisza *Tab*, automatycznego rozwijania składowych oraz dodawania do skryptów gotowych fragmentów kodu źródłowego. W rozdziale tym znajduje się opis najlepszych sposobów wykorzystania tych funkcji.

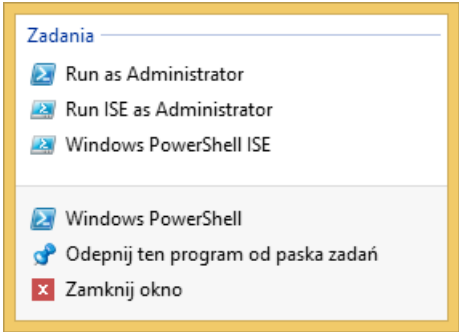
Uruchamianie środowiska Windows PowerShell ISE

W systemie Windows 8.1 środowisko Windows PowerShell ISE jest trochę ukryte. Zresztą podobnie jest w Windows Server 2012 R2, chociaż tam automatycznie do paska zadań na pulpicie dodawany jest skrót do Windows PowerShell ISE. Przypięcie Windows PowerShell do paska zadań w systemie Windows 8.1 to także bardzo dobry zwyczaj.

Jest kilka sposobów na uruchomienie środowiska Windows PowerShell ISE. W systemie Windows Server 2012 R2 na stronie startowej można wpisać **PowerShell**, aby wyszukać zarówno Windows PowerShell, jak i Windows PowerShell ISE. Jednak w systemie Windows 8 tak się nie da. Aby znaleźć Windows PowerShell ISE, należy wpisać Windows **PowerShell_ISE**. Innym sposobem jest kliknięcie prawym przyciskiem myszy ikony Windows PowerShell i wybranie z menu podręcznego opcji *Windows PowerShell ISE* lub *Run ISE as Administrator* (uruchom ISE jako administrator), które widać na rysunku 20.1.

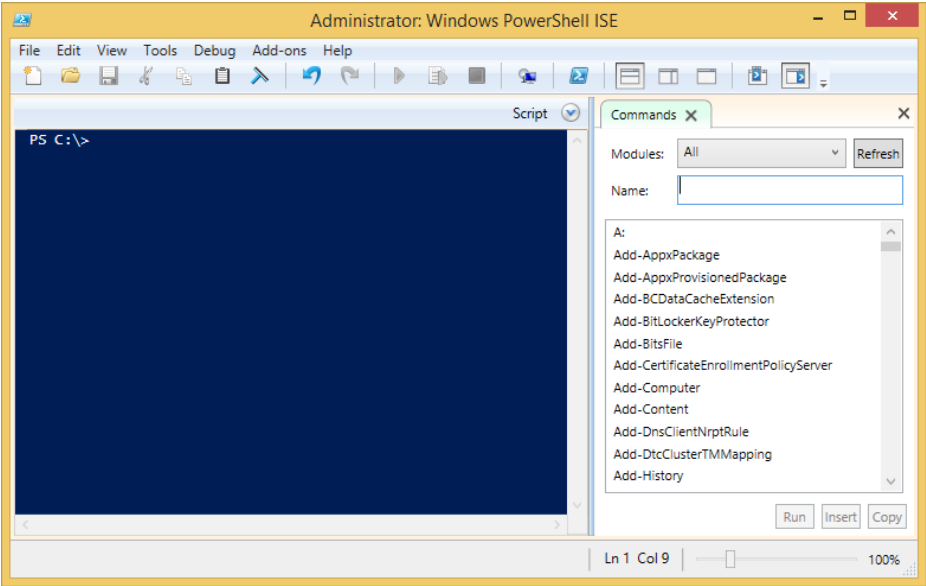
Zawartość okna Windows PowerShell ISE

Okno Windows PowerShell ISE składa się z dwóch części. Po lewej stronie znajduje się interaktywna konsola Windows PowerShell, a po prawej dodatkowe okienko z listą poleceń Windows PowerShell. Jeśli ktoś posługuje się konsolą Windows PowerShell ISE w sposób interaktywny, to przy użyciu



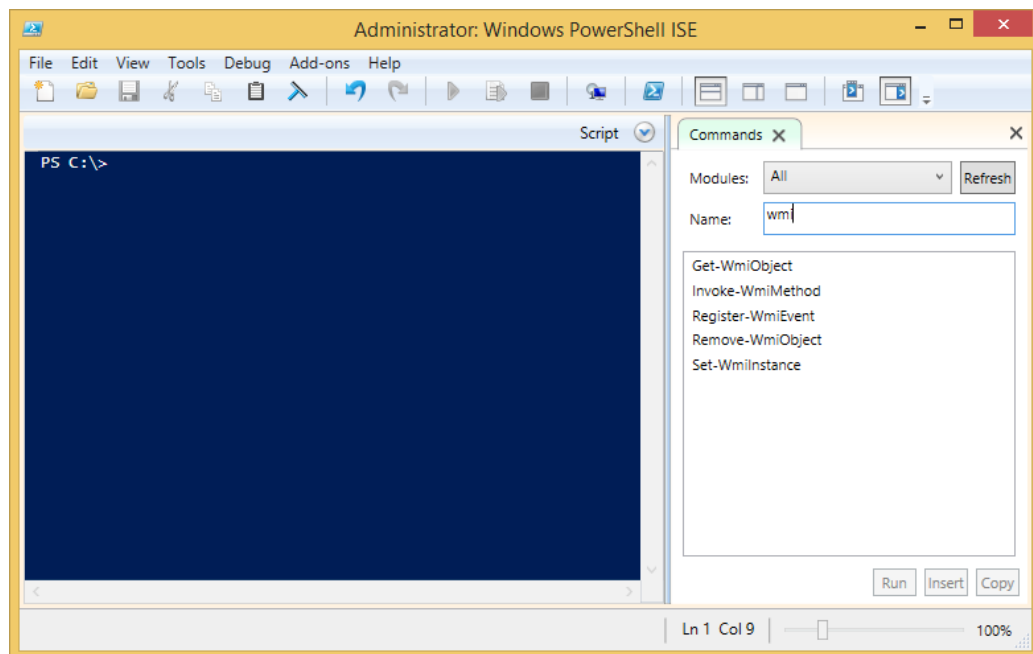
RYSUNEK 20.1. Klikając prawym przyciskiem myszy ikonę Windows PowerShell na pasku zadań systemu Windows, można wyświetlić menu zadań zawierające opcję uruchomienia środowiska Windows PowerShell ISE

tego dodatkowego okienka może budować polecenia za pomocą myszy. Po utworzeniu polecenia można kliknąć przycisk *Run* (wykonaj), aby je skopiować do okna konsoli i wykonać. Na rysunku 20.2 widać podstawowe okno środowiska Windows PowerShell ISE.



RYSUNEK 20.2. Okno środowiska Windows PowerShell ISE składa się z konsoli po lewej i okienka z listą poleceń po prawej stronie

Jeśli zaczniesz wpisywać nazwę polecenia w polu *Name* (nazwa), poniżej wyświetli się lista pasujących poleceń z wszystkich dostępnych modułów Windows PowerShell. Jest to doskonały sposób na znajdowanie poleceń i sprawdzanie ich lokalizacji. Standardowy sposób wyszukiwania polega na użyciu wzorca z symbolami wieloznacznymi, dzięki czemu dla ciągu **wmi** zostanie znalezionych pięć poleceń, jak widać na rysunku 20.3.



RYСУNEK 20.3. Dodatek poleceń wyszukuje polecenia przy użyciu wzorca zawierającego symbole wieloznaczne

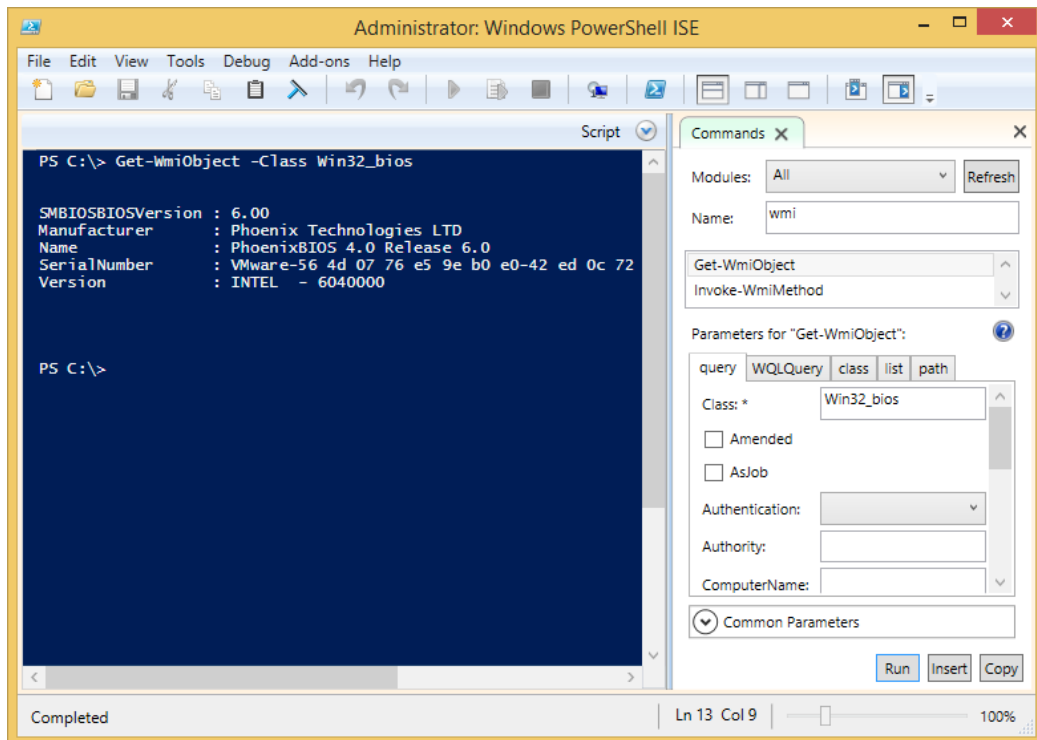
Jeśli na liście znalezionych poleceń znajduje się to, którego szukasz, kliknij je myszą. Spowoduje to wyświetlenie wykazu jego parametrów z podziałem na zestawy rozmieszczone na osobnych kartach. Duży wpływ na wygodę używania tej funkcji ma rozdzielczość ekranu. Im jest większa, tym lepiej. Przy niskiej rozdzielczości ekranu trzeba korzystać z funkcji przewijania, aby zobaczyć wszystkie karty i wszystkie znajdujące się na nich parametry. Łatwo coś przeoczyć. Na rysunku 20.4 widać efekt wysłania za pomocą polecenia `Get-WmiObject` zapytania do klasy WMI `Win32_Bios`. Wpisano nazwę klasy w polu *class* i naciśnięto przycisk *Run*. W wierszu poleceń najpierw zostało wyświetlone polecenie, a następnie wynik jego wykonania.

Praca w okienku skryptu

Kliknięcie strzałki w dół znajdującej się obok napisu *Script* w prawym górnym rogu okna wiersza poleceń powoduje otwarcie pustego okienka skryptu. Ponadto takie nowe okienko można otworzyć kliknięciem opcji *New* (nowy) w menu *File* (plik) lub białej kartki po lewej stronie paska narzędzi albo naciskając kombinację klawiszy *Ctrl+N*.

Fakt, że jest to okienko skryptowe, nie oznacza wcale, że aby go używać, trzeba włączyć obsługę skryptów. Dopóki nie zapiszesz pliku, możesz wykonywać dowolnie złożone polecenia bez włączania obsługi skryptów. Jednak po zapisaniu pliku polecenia te stają się skryptem i aby ich używać, trzeba zmienić zasadę wykonywania skryptów.

W połączeniu z okienkiem skryptu można używać okienka *Command*, ale należy wykonać dodatkową czynność. Najpierw utwórz polecenie zgodnie z opisem w poprzednim podrozdziale, a następnie zamiast przycisku *Run* kliknij przycisk *Copy* (kopiuj). Przejdź w wybrane miejsce



RYSUNEK 20.4. Aby wykonać polecenia Windows PowerShell w środowisku Windows PowerShell ISE, wybierz polecenie do wykonania z okienka Command, wprowadź wartości parametrów i kliknij przycisk Run

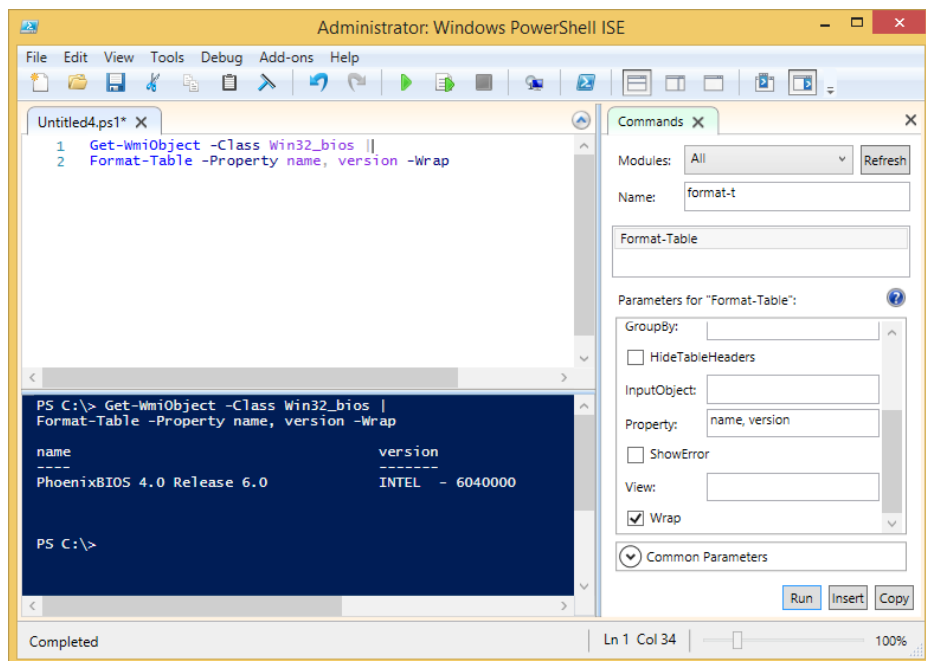
w okienku skryptu i wklej w nim skopiowane polecenie za pomocą opcji *Paste* (wklej) z menu wyświetlanego po naciśnięciu prawego przycisku myszy lub z menu *Edit* albo naciskając kombinację klawiszy *Ctrl+V*.

UWAGA

Jeśli klikniesz przycisk *Insert*, gdy okienko skryptu jest zmaksymalizowane, polecenie zostanie wklejone do ukrytego wiersza poleceń. Powtórne kliknięcie tego przycisku spowoduje wklejenie tego samego polecenia drugi raz w konsoli. Program nie wyświetli żadnego ostrzeżenia, że tak się stało.

Aby wykonać polecenia znajdujące się w okienku skryptu, kliknij znajdujący się pośrodku paska narzędzi zielony trójkąt, naciśnij klawisz *F5* lub kliknij opcję *Run* w menu *File*. Nastąpi przesłanie poleceń z okienka skryptu do okna konsoli i wykonanie ich. Wyniki zostaną wyświetlone pod spodem. Jeżeli polecenia zostaną zapisane w pliku, to przed wykonaniem nie będą już przenoszone do konsoli. Zamiast tego w konsoli będzie widać tylko ścieżkę do pliku i wynik.

Za pomocą okienka *Command* można tworzyć złożone polecenia przekazujące wyniki potokowe z jednego polecenia do drugiego. Na rysunku 20.5 widać efekt przekazania wyniku polecenia *Get-WmiObject* do polecenia *Format-Table*. Własności w poleceniu *Format-Table* i przełącznik *wrap* zostały wybrane w okienku *Command*.



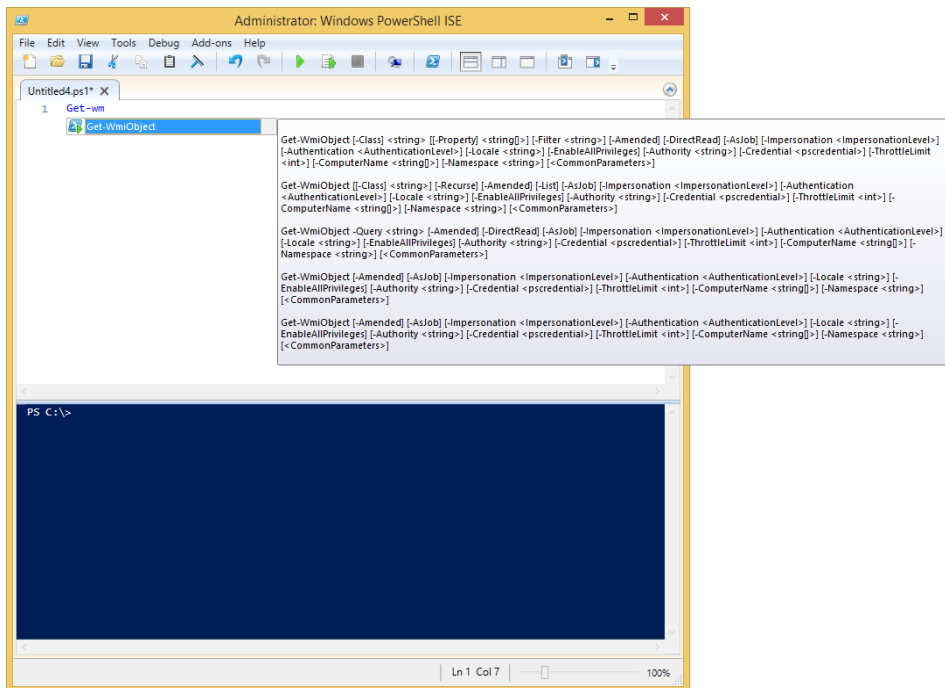
RYСУNEK 20.5. Za pomocą okienka Command można łatwo tworzyć złożone polecenia

Rozwijanie nazw za pomocą klawisza Tab i funkcja IntelliSense

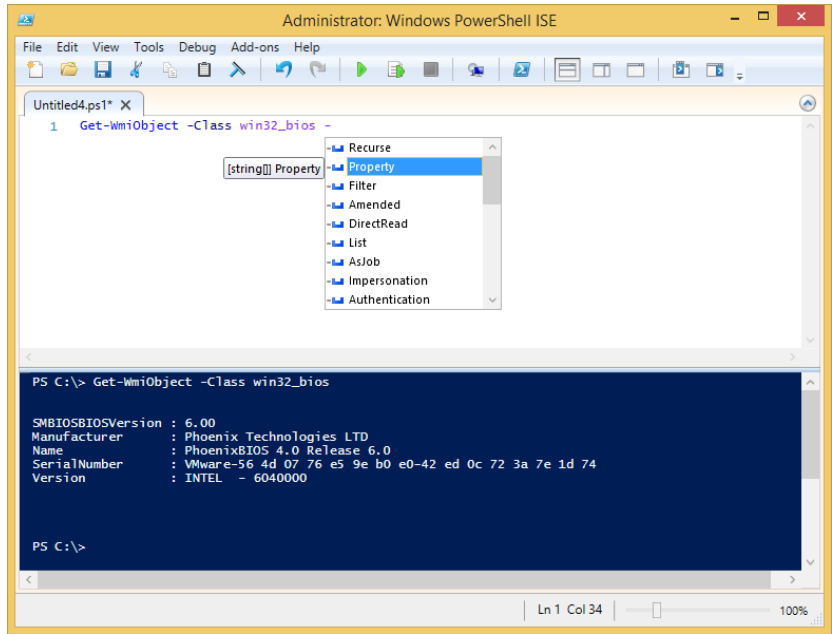
Wielu zaawansowanych skrypciarzy nie używa okienka *Command*, ponieważ zajmuje cenne miejsce na ekranie i zmusza do używania myszy. Dla nich kluczowe znaczenie mają funkcje rozwijania nazw za pomocą klawisza *Tab* i IntelliSense, które umożliwiają szybsze pisanie skryptów. Aby wyłączyć okienko *Command*, kliknij ikonę *x* w jego prawym górnym rogu albo usuń zaznaczenie opcji *Show Command Add-on* (wyświetl okienko *Command*) w menu *View* (widok). Program zapamięta to i nie włączy więcej okienka *Command*, dopóki użytkownik sam tego nie zrobi.

Funkcja IntelliSense dostarcza informacji pomocniczych w wyskakującym okienku i ułatwia wpisywanie poleceń bez dokładnej znajomości ich składni. Gdy zaczniesz wpisywać nazwę polecenia cmdlet, funkcja ta wyświetli wszystkie pasujące do niej możliwości, a kiedy wybierzesz jedno polecenie, zostaną wyświetlone szczegółowe informacje o jego składni.

Gdy po wybraniu konkretnego polecenia przejdziesz do definiowania parametrów, funkcja IntelliSense wyświetli listę dostępnych parametrów, po której można poruszać się za pomocą klawiszy strzałek w górę i w dół. Aby wstawić wybraną opcję, należy nacisnąć klawisz *Enter*. Potem należy wpisać wartość parametru i można przejść do następnego, który również będzie można wybrać z listy. Cały proces powtarza się tyle razy, ile trzeba, aby wstawić kompletne polecenie. Na rysunku 20.7 widać, jak użytkownik wybiera parametr *property* z listy parametrów opcjonalnych IntelliSense.



RYSUNEK 20.6. Po wybraniu polecenia z listy następuje wyświetlenie szczegółów jego składni przez funkcję IntelliSense



RYSUNEK 20.7. Funkcja IntelliSense wyświetla listę parametrów. Gdy wybierze się jeden z nich, następuje wyświetlenie jego typu danych

Sposób użycia fragmentów kodu w środowisku Windows PowerShell ISE

Nawet doświadczeni skrypciarze lubią używać gotowych fragmentów kodu w Windows PowerShell ISE, ponieważ dzięki nim oszczędzają mnóstwo czasu. Aby móc z nich korzystać, trzeba poznać zasady ich używania oraz znać trochę składnię Windows PowerShell. Jeśli spełniasz te warunki, to możesz używać gotowych fragmentów kodu w Windows PowerShell ISE i tworzyć skrypty szybciej, niż myślałeś, że to możliwe.

Tworzenie skryptów przy użyciu gotowych fragmentów kodu

Aby wyświetlić listę gotowych fragmentów kodu Windows PowerShell ISE, należy nacisnąć kombinację klawiszy *Ctrl+J* albo kliknąć opcję *Start Snippets* (włącz gotowe fragmenty) w menu *Edit* (edycja). Po pojawieniu się listy dostępnych fragmentów kodu wpisz pierwszą literę interesującego Cię fragmentu, aby szybko przejść do zawierającej go sekcji (po liście można poruszać się także za pomocą myszy). Kiedy znajdziesz interesujący Cię fragment, naciśnij klawisz *Enter*, aby wstawić go do okienka skryptu Windows PowerShell.

Tworzenie nowych gotowych fragmentów kodu w Windows PowerShell ISE

Każdy, kto przez jakiś czas poużywa gotowych fragmentów kodu w środowisku Windows PowerShell ISE, zastanawia się, jak kiedyś mógł w ogóle bez nich żyć. Jednocześnie nasuwa się pytanie, czy można tworzyć własne gotowe fragmenty kodu. Tak się składa, że jest to bardzo łatwe i istnieje nawet służące do tego specjalne polecenie cmdlet: *New-IseSnippet*.

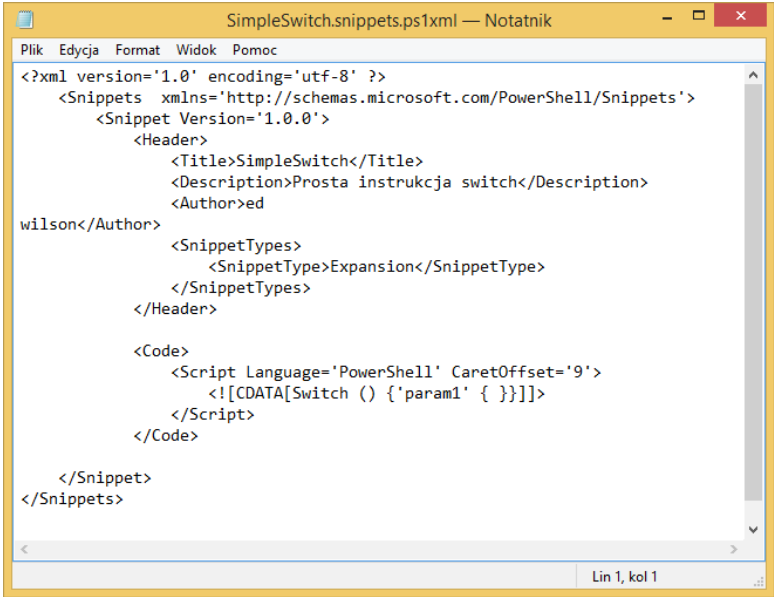
UWAGA

Aby utworzyć gotowy fragment kodu lub użyć gotowego fragmentu kodu zdefiniowanego przez użytkownika, należy zmienić zasadę wykonywania skryptów na taką, która pozwala na ich wykonywanie. Jest to spowodowane tym, że fragmenty kodu są wczytywane z plików XML, a do wczytywania plików potrzebne są uprawnienia umożliwiające wykonywanie skryptów. Zasadę wykonywania skryptów w swoim środowisku można sprawdzić za pomocą polecenia *Get-ExecutionPolicy*. Natomiast do jej ustawiania służy polecenie *Set-ExecutionPolicy*.

Aby utworzyć nowy gotowy fragment kodu, należy użyć polecenia *New-IseSnippet*. Utworzony fragment staje się od razu dostępny do użytku. Składnia tego polecenia jest prosta, ale zajmuje dużo miejsca. Obowiązkowe są tylko trzy parametry: *Description*, *Text* i *Title*. W parametrze *Title* określa się nazwę fragmentu. Podstawowy kod źródłowy definiuje się w parametrze *Text*. Jeśli trzeba podzielić kod na kilka wierszy, należy używać znaku specjalnego *'r'*. Oczywiście w takim przypadku tekst fragmentu musi być ujęty w cudzysłów podwójny, a nie pojedynczy. Poniżej znajduje się przykład utworzenia fragmentu będącego uproszczoną wersją składni instrukcji *switch*. Jest to jeden logiczny wiersz kodu.

```
New-IseSnippet -Title SimpleSwitch -Description "Prosta instrukcja switch" -Author "ed wilson" -Text "Switch () 'r{'param1' { }}r'" -CaretOffset 9
```

Polecenie `New-IseSnippet` tworzy w katalogu `Dokumenty/WindowsPowerShell/Snippets` nowy plik XML o takiej nazwie, jaką podano w parametrze `Title`. Na rysunku 20.8 widać zawartość pliku `SimpleSwitch.snippets.ps1xml` powstałego w wyniku wykonania polecenia przedstawionego powyżej.



```
<?xml version='1.0' encoding='utf-8' ?>
<Snippets xmlns='http://schemas.microsoft.com/PowerShell/Snippets'>
  <Snippet Version='1.0.0'>
    <Header>
      <Title>SimpleSwitch</Title>
      <Description>Prosta instrukcja switch</Description>
      <Author>ed
wilson</Author>
      <SnippetTypes>
        <SnippetType>Expansion</SnippetType>
      </SnippetTypes>
    </Header>
    <Code>
      <Script Language='PowerShell' CaretOffset='9'>
        <![CDATA[Switch () {'param1' { }}]]>
      </Script>
    </Code>
  </Snippet>
</Snippets>
```

RYСУNEK 20.8. Fragmenty kodu Windows PowerShell są zapisywane w pliku `snippets.xml` w folderze konsoli Windows PowerShell

Usuwanie fragmentów kodu zdefiniowanych przez użytkownika

Istnieją polecenia `New-IseSnippet` i `Get-IseSnippet`, a nie istnieje polecenie `Remove-IseSnippet`. Ale to nie przeszkadza, bo jest przecież polecenie `Remove-Item`. Aby usunąć wszystkie własne fragmenty kodu, należy pobrać ich listę za pomocą polecenia `Get-IseSnippet` i usunąć je za pomocą polecenia `Remove-Item`, jak pokazano poniżej:

`Get-IseSnippet` | **Remove-Item**

Jeśli nie chcesz usuwać wszystkich własnych fragmentów kodu, możesz wybrać tylko niektóre z nich za pomocą polecenia `Where-Object`. Poniżej znajduje się przykładowe polecenie wyświetlające wszystkie zdefiniowane w systemie własne fragmenty kodu użytkownika:

PS C:\Windows\system32> Get-IseSnippet

Directory: C:\Users\administrator.IAMMRED\Documents\WindowsPowerShell\Snippets

| Mode | LastWriteTime | | Length | Name |
|-------|---------------|---------|--------|------------------------------|
| ---- | ----- | ----- | ----- | ---- |
| -a--- | 7/1/2012 | 1:03 AM | 653 | bogus.snippets.ps1xml |
| -a--- | 7/1/2012 | 1:02 AM | 653 | mysnip.snippets.ps1xml |
| -a--- | 7/1/2012 | 1:02 AM | 671 | simpleswitch.snippets.ps1xml |

Teraz za pomocą polecenia `Where-Object` (którego aliasem jest `?`) można pobrać wszystkie te zdefiniowane przez użytkownika fragmenty kodu, które w nazwie nie zawierają słowa `switch`. Wszystkie polecenia, które przejdą przez filtr, zostają przekazane do polecenia `Remove-Item`. W poniższym przykładzie użyto przełącznika `what if`, aby zobaczyć, które fragmenty kodu zostałyby usunięte.

```
PS C:\Windows\system32> Get-IsSnippet | ? name -NotMatch 'switch' | Remove-Item -WhatIf
What if: Performing operation "Remove file" on Target
"C:\Users\administrator.IAMMRED\Documents\WindowsPowerShell\Snippets\bogus.snippets.ps1xml".
What if: Performing operation "Remove file" on Target
"C:\Users\administrator.IAMMRED\Documents\WindowsPowerShell\Snippets\mysnip.snippets.ps1xml".
```

Po znalezieniu potwierdzenia, że zostaną usunięte tylko te fragmenty kodu, które trzeba, można skasować przełącznik `what if` z polecenia `Remove-Item` i jeszcze raz wykonać polecenie. Aby sprawdzić, które fragmenty pozostały, można posłużyć się poleceniem `Get-IsSnippet`.

```
PS C:\Windows\system32> Get-IsSnippet | ? name -NotMatch 'switch' | Remove-Item
```

```
PS C:\Windows\system32> Get-IsSnippet
```

```
Directory: C:\Users\administrator.IAMMRED\Documents\WindowsPowerShell\Snippets
```

| Mode | LastWriteTime | Length | Name |
|-------|------------------|--------|------------------------------|
| ---- | ----- | ----- | ---- |
| -a--- | 7/1/2012 1:02 AM | 671 | simpleswitch.snippets.ps1xml |

Dodatkowe źródła informacji

- Centrum skryptów portalu TechNet pod adresem <http://www.microsoft.com/technet/scriptcenter> zawiera wiele przykładów skryptów.
- Na stronie <http://msdn.microsoft.com/en-us/library/bb822049.aspx> znajduje się opis historii wersji platformy .NET.
- Na stronie <http://msdn.microsoft.com/en-us/library/lh925568.aspx> można znaleźć informacje na temat sprawdzania numeru zainstalowanej wersji platformy .NET.
- Wszystkie skrypty opisane w tym rozdziale są dostępne do pobrania w repozytorium centrum skryptów portalu TechNet pod adresem <http://gallery.technet.microsoft.com/scriptcenter/PowerShell-40-Best-d9e16039>.

Rozdział 21

Narzędzia do pracy zdalnej i zadania Windows PowerShell

- Narzędzia do pracy zdalnej Windows PowerShell
- Zadania Windows PowerShell
- Dodatkowe źródła informacji

Narzędzia do pracy zdalnej są tym składnikiem, który sprawia, że konsola Windows PowerShell jest nie tylko ciekawym przyrządem do wykonywania paru poleceń, ale rozbudowanym środowiskiem do zarządzania w ekosystemie przedsiębiorstwa. W tym rozdziale dowiesz się, jak działają narzędzia do pracy zdalnej Windows PowerShell oraz czym różnią się od klasycznych narzędzi tego typu. Ponadto przyjrzyj się paru konkretnym przykładom ich użycia i dowiesz się, czym są zadania w Windows PowerShell.

Narzędzia do pracy zdalnej Windows PowerShell

Do najważniejszych udoskonaleń wprowadzonych w Windows PowerShell 4.0 zaliczają się poprawki dotyczące narzędzi do pracy zdalnej. Ich konfiguracja jest znacznie łatwiejsza niż w Windows PowerShell 2.0 i w wielu przypadkach można w końcu powiedzieć, że po prostu działają. W kwestii narzędzi do pracy zdalnej Windows PowerShell mogą wynikać pewne nieporozumienia, ponieważ jest wiele sposobów na wykonywanie poleceń na zdalnym serwerze. Wybór konkretnego z nich zależy przede wszystkim od konfiguracji sieci i zabezpieczeń.

Klasyczne sposoby pracy zdalnej

Klasyczne techniki pracy zdalnej polegają na nawiązywaniu połączeń ze zdalnymi komputerami za pomocą takich protokołów jak DCOM czy RPC. Wymaga to otwarcia wielu portów w zaporze sieciowej oraz uruchomienia rozmaitych procesów wykorzystywanych przez różne polecenia cmdlet. Wszystkie polecenia cmdlet do pracy zdalnej można znaleźć za pomocą polecenia `Get-Help` z parametrem `computersname`. Poniżej znajduje się przykład użycia tego polecenia i jego wynik.

```

PS C:\> get-help * -Parameter computername | sort name | ft name, synopsis -auto -wrap
Name                               Synopsis
----
Add-Computer                       Add the local computer to a domain or workgroup.
Add-Printer                        Adds a printer to the specified computer.
Add-PrinterDriver                  Installs a printer driver on the specified computer.
Add-PrinterPort                   Installs a printer port on the specified computer.
Clear-EventLog                     Deletes all entries from specified event logs on the local or remote
computers.
Connect-PSSession                 Reconnects to disconnected sessions.
Connect-WSMan                     Connects to the WinRM service on a remote computer.
Disconnect-PSSession              Disconnects from a session.
Disconnect-WSMan                  Disconnects the client from the WinRM service on a remote computer.
Enter-PSSession                   Starts an interactive session with a remote computer.
Get-CimAssociatedInstance

Get-CimAssociatedInstance [-InputObject]
<ciminstance> [[-Association] <string>]
[-ResultClassName <string>] [-Namespace <string>]
[-OperationTimeoutSec <uint32>] [-ResourceUri
<uri>] [-ComputerName <string[]>] [-KeyOnly]
[<CommonParameters>]

Get-CimAssociatedInstance [-InputObject]
<ciminstance> [[-Association] <string>]
-CimSession <CimSession[]> [-ResultClassName
<string>] [-Namespace <string>]
[-OperationTimeoutSec <uint32>] [-ResourceUri
<uri>] [-KeyOnly] [CommonParameters]

Get-CimClass

Get-CimClass [[-ClassName] <string>]
[-Namespace] <string> [-OperationTimeoutSec
<uint32>] [-ComputerName <string[]>] [-MethodName
<string>] [-PropertyName <string>]
[-QualifierName <string>] [CommonParameters]

Get-CimClass [[-ClassName] <string>]
[-Namespace] <string> -CimSession
<CimSession[]> [-OperationTimeoutSec <uint32>]
[-MethodName <string>] [-PropertyName <string>]
[-QualifierName <string>] [CommonParameters]

Get-CimInstance

Get-CimInstance [-ClassName] <string>
[-ComputerName <string[]>] [-KeyOnly] [-Namespace
<string>] [-OperationTimeoutSec <uint32>]
[-QueryDialect <string>] [-Shallow] [-Filter
<string>] [-Property <string[]>]
[CommonParameters]

Get-CimInstance [-InputObject] <ciminstance>
-CimSession <CimSession[]> [-ResourceUri <uri>]
[-OperationTimeoutSec <uint32>]
[CommonParameters]

Get-CimInstance -CimSession <CimSession[]>
-ResourceUri <uri> [-KeyOnly] [-Namespace
<string>] [-OperationTimeoutSec <uint32>]
[-Shallow] [-Filter <string>] [-Property
<string[]>] [CommonParameters]

```

```
Get-CimInstance -CimSession <CimSession[]> -Query
<string> [-ResourceUri <uri>] [-Namespace
<string>] [-OperationTimeoutSec <uint32>]
[-QueryDialect <string>] [-Shallow]
[<CommonParameters>]
```

```
Get-CimInstance [-ClassName] <string> -CimSession
<CimSession[]> [-KeyOnly] [-Namespace <string>]
[-OperationTimeoutSec <uint32>] [-QueryDialect
<string>] [-Shallow] [-Filter <string>]
[-Property <string[]>] [<CommonParameters>]
```

```
Get-CimInstance -ResourceUri <uri> [-ComputerName
<string[]>] [-KeyOnly] [-Namespace <string>]
[-OperationTimeoutSec <uint32>] [-Shallow]
[-Filter <string>] [-Property <string[]>]
[<CommonParameters>]
```

```
Get-CimInstance [-InputObject] <ciminstance>
[-ResourceUri <uri>] [-ComputerName <string[]>]
[-OperationTimeoutSec <uint32>]
[<CommonParameters>]
```

```
Get-CimInstance -Query <string> [-ResourceUri
<uri>] [-ComputerName <string[]>] [-Namespace
<string>] [-OperationTimeoutSec <uint32>]
[-QueryDialect <string>] [-Shallow]
[<CommonParameters>]
```

Get-CimSession

```
Get-CimSession [[-ComputerName] <string[]>]
[<CommonParameters>]
```

```
Get-CimSession [-Id] <uint32[]>
[<CommonParameters>]
```

```
Get-CimSession -InstanceId <guid[]>
[<CommonParameters>]
```

```
Get-CimSession -Name <string[]>
[<CommonParameters>]
```

| | |
|------------------------|---|
| Get-Counter | Gets performance counter data from local and remote computers. |
| Get-EventLog | Gets the events in an event log, or a list of the event logs, on the local or remote computers. |
| Get-HotFix | Gets the hotfixes that have been applied to the local and remote computers. |
| Get-PrintConfiguration | Gets the configuration information of a printer. |
| Get-Printer | Retrieves a list of printers installed on a computer. |
| Get-PrinterDriver | Retrieves the list of printer drivers installed on the specified computer. |
| Get-PrinterPort | Retrieves a list of printer ports installed on the specified computer. |
| Get-PrinterProperty | Retrieves printer properties for the specified printer. |
| Get-PrintJob | Retrieves a list of print jobs in the specified printer. |
| Get-Process | Gets the processes that are running on the local computer or a remote computer. |
| Get-PSSession | Gets the Windows PowerShell sessions on local and remote computers. |
| Get-Service | Gets the services on a local or remote computer. |

| | |
|-------------------|--|
| Get-WinEvent | Gets events from event logs and event tracing log files on local and remote computers. |
| Get-WmiObject | Gets instances of Windows Management Instrumentation (WMI) classes or information about the available classes. |
| Get-WSManInstance | Displays management information for a resource instance specified by a Resource URI. |

Invoke-CimMethod

```
Invoke-CimMethod [-ClassName] <string>
[[-Arguments] <IDictionary>] [-MethodName]
<string> [-ComputerName <string[]>] [-Namespace
<string>] [-OperationTimeoutSec <uint32>]
[-WhatIf] [-Confirm] [<CommonParameters>]
```

```
Invoke-CimMethod [-ClassName] <string>
[[-Arguments] <IDictionary>] [-MethodName]
<string> -CimSession <CimSession[]> [-Namespace
<string>] [-OperationTimeoutSec <uint32>]
[-WhatIf] [-Confirm] [<CommonParameters>]
```

```
Invoke-CimMethod [[-Arguments] <IDictionary>]
[-MethodName] <string> -ResourceUri <uri>
-CimSession <CimSession[]> [-Namespace <string>]
[-OperationTimeoutSec <uint32>] [-WhatIf]
[-Confirm] [<CommonParameters>]
```

```
Invoke-CimMethod [[-Arguments] <IDictionary>]
[-MethodName] <string> -ResourceUri <uri>
[-ComputerName <string[]>] [-Namespace <string>]
[-OperationTimeoutSec <uint32>] [-WhatIf]
[-Confirm] [<CommonParameters>]
```

```
Invoke-CimMethod [-InputObject] <ciminstance>
[[-Arguments] <IDictionary>] [-MethodName]
<string> [-ResourceUri <uri>] [-ComputerName
<string[]>] [-OperationTimeoutSec <uint32>]
[-WhatIf] [-Confirm] [<CommonParameters>]
```

```
Invoke-CimMethod [-InputObject] <ciminstance>
[[-Arguments] <IDictionary>] [-MethodName]
<string> -CimSession <CimSession[]> [-ResourceUri
<uri>] [-OperationTimeoutSec <uint32>] [-WhatIf]
[-Confirm] [<CommonParameters>]
```

```
Invoke-CimMethod [-CimClass] <CimClass>
[[-Arguments] <IDictionary>] [-MethodName]
<string> -CimSession <CimSession[]>
[-OperationTimeoutSec <uint32>] [-WhatIf]
[-Confirm] [<CommonParameters>]
```

```
Invoke-CimMethod [-CimClass] <CimClass>
[[-Arguments] <IDictionary>] [-MethodName]
<string> [-ComputerName <string[]>]
[-OperationTimeoutSec <uint32>] [-WhatIf]
[-Confirm] [<CommonParameters>]
```

```
Invoke-CimMethod [[-Arguments] <IDictionary>]
[-MethodName] <string> -Query <string>
-CimSession <CimSession[]> [-QueryDialect
<string>] [-Namespace <string>]
```



```
[-OperationTimeoutSec <uint32>] [-WhatIf]
[-Confirm] [<CommonParameters>]
```

```
Invoke-CimMethod [[-Arguments] <IDictionary>]
[-MethodName] <string> -Query <string>
[-QueryDialect <string>] [-ComputerName
<string[]>] [-Namespace <string>]
[-OperationTimeoutSec <uint32>] [-WhatIf]
[-Confirm] [<CommonParameters>]
```

Invoke-Command
Invoke-WmiMethod

Runs commands on local and remote computers.
Calls Windows Management Instrumentation (WMI) methods.

Invoke-WSManAction

Invokes an action on the object that is specified by the Resource URI and by the selectors.

Join-DtcDiagnosticResourceManager

```
Join-DtcDiagnosticResourceManager [-Transaction]
<DtcDiagnosticTransaction> [[-ComputerName]
<string>] [[-Port] <int>] [-Volatile]
[<CommonParameters>]
```

Limit-EventLog

Sets the event log properties that limit the size of the event log and the age of its entries.

New-CimInstance

```
New-CimInstance [-ClassName] <string>
[[[-Property] <IDictionary>] [-Key <string[]>]
[-Namespace <string>] [-OperationTimeoutSec
<uint32>] [-ComputerName <string[]>]
[-ClientOnly] [-WhatIf] [-Confirm]
[<CommonParameters>]
```

```
New-CimInstance [-ClassName] <string>
[[[-Property] <IDictionary>] -CimSession
<CimSession[]> [-Key <string[]>] [-Namespace
<string>] [-OperationTimeoutSec <uint32>]
[-ClientOnly] [-WhatIf] [-Confirm]
[<CommonParameters>]
```

```
New-CimInstance [[[-Property] <IDictionary>]
-ResourceUri <uri> -CimSession <CimSession[]>
[-Key <string[]>] [-Namespace <string>]
[-OperationTimeoutSec <uint32>] [-WhatIf]
[-Confirm] [<CommonParameters>]
```

```
New-CimInstance [[[-Property] <IDictionary>]
-ResourceUri <uri> [-Key <string[]>] [-Namespace
<string>] [-OperationTimeoutSec <uint32>]
[-ComputerName <string[]>] [-WhatIf] [-Confirm]
[<CommonParameters>]
```

```
New-CimInstance [-CimClass] <CimClass>
[[[-Property] <IDictionary>] [-OperationTimeoutSec
<uint32>] [-ComputerName <string[]>]
[-ClientOnly] [-WhatIf] [-Confirm]
[<CommonParameters>]
```

```
New-CimInstance [-CimClass] <CimClass>
[[[-Property] <IDictionary>] -CimSession
<CimSession[]> [-OperationTimeoutSec <uint32>]
```

```
[-ClientOnly] [-WhatIf] [-Confirm]
[<CommonParameters>]
```

New-CimSession

```
New-CimSession [[-ComputerName] <string[]>]
[[-Credential] <pscredential>] [-Authentication
<PasswordAuthenticationMechanism>] [-Name
<string>] [-OperationTimeoutSec <uint32>]
[-SkipTestConnection] [-Port <uint32>]
[-SessionOption <CimSessionOptions>]
[<CommonParameters>]
```

```
New-CimSession [[-ComputerName] <string[]>]
[-CertificateThumbprint <string>] [-Name
<string>] [-OperationTimeoutSec <uint32>]
[-SkipTestConnection] [-Port <uint32>]
[-SessionOption <CimSessionOptions>]
[<CommonParameters>]
```

New-EventLog Creates a new event log and a new event source on a local or remote computer.

New-PSSession Creates a persistent connection to a local or remote computer.

New-PSWorkflowSession Creates a workflow session.

New-WSManInstance Creates a new instance of a management resource.

Receive-DtcDiagnosticTransaction

```
Receive-DtcDiagnosticTransaction [[-ComputerName]
<string>] [[-Port] <int>] [[-PropagationMethod]
<DtcTransactionPropagation>] [<CommonParameters>]
```

Receive-Job Gets the results of the Windows PowerShell background jobs in the current session.

Receive-PSSession Gets results of commands in disconnected sessions.

Register-CimIndicationEvent

```
Register-CimIndicationEvent [-ClassName] <string>
[[-SourceIdentifier] <string>] [[-Action]
<scriptblock>] [-Namespace <string>]
[-OperationTimeoutSec <uint32>] [-ComputerName
<string>] [-MessageData <psoobject>]
[-SupportEvent] [-Forward] [-MaxTriggerCount
<int>] [<CommonParameters>]
```

```
Register-CimIndicationEvent [-ClassName] <string>
[[-SourceIdentifier] <string>] [[-Action]
<scriptblock>] -CimSession <CimSession>
[-Namespace <string>] [-OperationTimeoutSec
<uint32>] [-MessageData <psoobject>]
[-SupportEvent] [-Forward] [-MaxTriggerCount
<int>] [<CommonParameters>]
```

```
Register-CimIndicationEvent [-Query] <string>
[[-SourceIdentifier] <string>] [[-Action]
<scriptblock>] [-Namespace <string>]
[-QueryDialect <string>] [-OperationTimeoutSec
<uint32>] [-ComputerName <string>] [-MessageData
<psoobject>] [-SupportEvent] [-Forward]
[-MaxTriggerCount <int>] [<CommonParameters>]
```

```
Register-CimIndicationEvent [-Query] <string>
[[-SourceIdentifier] <string>] [[-Action]
<scriptblock>] -CimSession <CimSession>
```

| | |
|----------------------|---|
| | <pre>[-Namespace <string>] [-QueryDialect <string>] [-OperationTimeoutSec <uint32>] [-MessageData <psobject>] [-SupportEvent] [-Forward] [-MaxTriggerCount <int>] [<CommonParameters>]</pre> |
| Register-WmiEvent | Subscribes to a Windows Management Instrumentation (WMI) event. |
| Remove-CimInstance | <pre>Remove-CimInstance [-InputObject] <ciminstance> [-ResourceUri <uri>] [-ComputerName <string[]>] [-OperationTimeoutSec <uint32>] [-WhatIf] [-Confirm] [<CommonParameters>]</pre> <pre>Remove-CimInstance [-InputObject] <ciminstance> -CimSession <CimSession[]> [-ResourceUri <uri>] [-OperationTimeoutSec <uint32>] [-WhatIf] [-Confirm] [<CommonParameters>]</pre> <pre>Remove-CimInstance [-Query] <string> [[-Namespace] <string>] -CimSession <CimSession[]> [-OperationTimeoutSec <uint32>] [-QueryDialect <string>] [-WhatIf] [-Confirm] [<CommonParameters>]</pre> <pre>Remove-CimInstance [-Query] <string> [[-Namespace] <string>] [-ComputerName <string[]>] [-OperationTimeoutSec <uint32>] [-QueryDialect <string>] [-WhatIf] [-Confirm] [<CommonParameters>]</pre> |
| Remove-CimSession | <pre>Remove-CimSession [-CimSession] <CimSession[]> [-WhatIf] [-Confirm] [<CommonParameters>]</pre> <pre>Remove-CimSession [-ComputerName] <string[]> [-WhatIf] [-Confirm] [<CommonParameters>]</pre> <pre>Remove-CimSession [-Id] <uint32[]> [-WhatIf] [-Confirm] [<CommonParameters>]</pre> <pre>Remove-CimSession -InstanceId <guid[]> [-WhatIf] [-Confirm] [<CommonParameters>]</pre> <pre>Remove-CimSession -Name <string[]> [-WhatIf] [-Confirm] [<CommonParameters>]</pre> |
| Remove-Computer | Removes the local computer from its domain. |
| Remove-EventLog | Deletes an event log or unregisters an event source. |
| Remove-Printer | Removes a printer from the specified computer. |
| Remove-PrinterDriver | Deletes printer driver from the specified computer. |
| Remove-PrinterPort | Removes the specified printer port from the specified computer. |
| Remove-PrintJob | Removes a print job on the specified printer. |
| Remove-PSSession | Closes one or more Windows PowerShell sessions (PSSessions). |
| Remove-WmiObject | Deletes an instance of an existing Windows Management Instrumentation (WMI) class. |
| Remove-WSManInstance | Deletes a management resource instance. |
| Rename-Computer | Renames a computer. |
| Restart-Computer | Restarts ("reboots") the operating system on local and remote computers. |
| Restart-PrintJob | Restarts a print job on the specified printer. |
| Resume-PrintJob | Resumes a suspended print job. |

| | |
|-------------------------------|---|
| Send-DtcDiagnosticTransaction | <pre>Send-DtcDiagnosticTransaction [-Transaction] <DtcDiagnosticTransaction> [[-ComputerName] <string>] [[-Port] <int>] [[-PropagationMethod] <DtcTransactionPropagation>] [<CommonParameters>]</pre> |
| Set-CimInstance | <pre>Set-CimInstance [-InputObject] <ciminstance> [-ComputerName <string[]>] [-ResourceUri <uri>] [-OperationTimeoutSec <uint32>] [-Property <IDictionary>] [-PassThru] [-WhatIf] [-Confirm] <CommonParameters> Set-CimInstance [-InputObject] <ciminstance> -CimSession <CimSession[]> [-ResourceUri <uri>] [-OperationTimeoutSec <uint32>] [-Property <IDictionary>] [-PassThru] [-WhatIf] [-Confirm] <CommonParameters> Set-CimInstance [-Query] <string> -CimSession <CimSession[]> -Property <IDictionary> [-Namespace <string>] [-OperationTimeoutSec <uint32>] [-QueryDialect <string>] [-PassThru] [-WhatIf] [-Confirm] [<CommonParameters>] Set-CimInstance [-Query] <string> -Property <IDictionary> [-ComputerName <string[]>] [-Namespace <string>] [-OperationTimeoutSec <uint32>] [-QueryDialect <string>] [-PassThru] [-WhatIf] [-Confirm] [<CommonParameters>]</pre> |
| Set-PrintConfiguration | Sets the configuration information for the specified printer. |
| Set-Printer | Updates the configuration of an existing printer. |
| Set-PrinterProperty | Modifies the printer properties for the specified printer. |
| Set-Service | Starts, stops, and suspends a service, and changes its properties. |
| Set-WmiInstance | Creates or updates an instance of an existing Windows Management Instrumentation (WMI) class. |
| Set-WSManInstance | Modifies the management information that is related to a resource. |
| Show-EventLog | Displays the event logs of the local or a remote computer in Event Viewer. |
| Stop-Computer | Stops (shuts down) local and remote computers. |
| Suspend-PrintJob | Suspends a print job on the specified printer. |
| Test-Connection | Sends ICMP echo request packets ("pings") to one or more computers. |
| Test-WSMan | Tests whether the WinRM service is running on a local or remote computer. |
| Write-EventLog | Writes an event to an event log. |

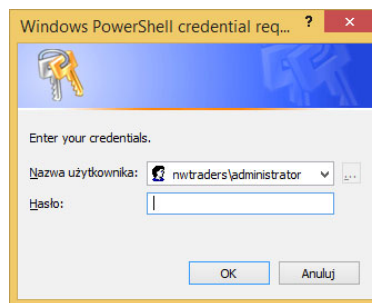
Jak widać, wiele poleceń Windows PowerShell zawierających parametr `computername` dotyczy usługi WSMAN, modelu CIM lub sesji. Aby pozbyć się ich z listy, należy dodać polecenie `Where-Object` (którego aliasem jest `?`). Poniżej znajduje się zmodyfikowana wersja poprzedniego polecenia wraz z wynikiem:

```
PS C:\> Get-Help * -Parameter computername -Category cmdlet | ? modulename -match
'PowerShell.Management' | sort name | ft name, synopsis -AutoSize -Wrap
```

| Name | Synopsis |
|--------------|--|
| ---- | ----- |
| Add-Computer | Add the local computer to a domain or workgroup. |

| | |
|-------------------|--|
| Clear-EventLog | Deletes all entries from specified event logs on the local or remote computers. |
| Get-EventLog | Gets the events in an event log, or a list of the event logs, on the local or remote computers. |
| Get-HotFix | Gets the hotfixes that have been applied to the local and remote computers. |
| Get-Process | Gets the processes that are running on the local computer or a remote computer. |
| Get-Service | Gets the services on a local or remote computer. |
| Get-WmiObject | Gets instances of Windows Management Instrumentation (WMI) classes or information about the available classes. |
| Invoke-WmiMethod | Calls Windows Management Instrumentation (WMI) methods. |
| Limit-EventLog | Sets the event log properties that limit the size of the event log and the age of its entries. |
| New-EventLog | Creates a new event log and a new event source on a local or remote computer. |
| Register-WmiEvent | Subscribes to a Windows Management Instrumentation (WMI) event. |
| Remove-Computer | Removes the local computer from its domain. |
| Remove-EventLog | Deletes an event log or unregisters an event source. |
| Remove-WmiObject | Deletes an instance of an existing Windows Management Instrumentation (WMI) class. |
| Rename-Computer | Renames a computer. |
| Restart-Computer | Restarts ("reboots") the operating system on local and remote computers. |
| Set-Service | Starts, stops, and suspends a service, and changes its properties. |
| Set-WmiInstance | Creates or updates an instance of an existing Windows Management Instrumentation (WMI) class. |
| Show-EventLog | Displays the event logs of the local or a remote computer in Event Viewer. |
| Stop-Computer | Stops (shuts down) local and remote computers. |
| Test-Connection | Sends ICMP echo request packets ("pings") to one or more computers. |
| Write-EventLog | Writes an event to an event log. |

Niektóre z tych poleceń umożliwiają podanie danych poświadczających tożsamość, dzięki czemu do nawiązywania i pobierania danych można używać dowolnego konta. Na rysunku 21.1 widać okno dialogowe do wpisania nazwy użytkownika i hasła, które jest wyświetlane przez jedno z takich poleceń.



RYSUNEK 21.1. Polecenia mające parametr credential wyświetlają okno dialogowe, w którym należy wpisać nazwę użytkownika i hasło

Poniżej znajduje się przykład użycia parametrów computername i credential:

```
PS C:\> Get-WinEvent -LogName application -MaxEvents 1 -ComputerName ex1 -Credential
nwtraders\administrator
```

| TimeCreated | ProviderName | Id Message |
|---------------------|---------------------|----------------------------|
| ----- | ----- | ----- |
| 01-07-2012 11:54:14 | MSExchange ADAccess | 2080 Process MAD.EXE (...) |

Ale jak napisałem wcześniej, polecenia te często wymagają otwarcia luk w zaporze sieciowej lub uruchomienia pewnych usług. Przez to na komputerach o zaostrzonych regułach dostępu ich wykonanie często kończy się niepowodzeniem. Poniżej znajduje się przykład takiego błędu:

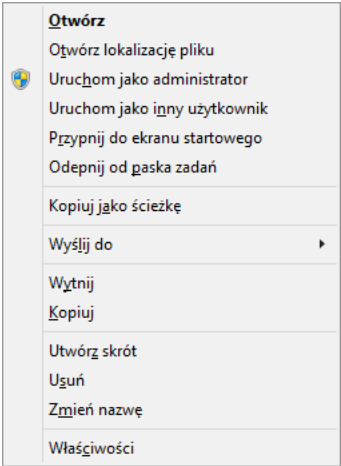
```
PS C:\> Get-WinEvent -LogName application -MaxEvents 1 -ComputerName dc1 -Credential
nwtraders\administrator
Get-WinEvent : The RPC server is unavailable
At line:1 char:1
+ Get-WinEvent -LogName application -MaxEvents 1 -ComputerName dc1 -Credential iam...
+ ~~~~~
    + CategoryInfo          : NotSpecified: (:) [Get-WinEvent], EventLogException
    + FullyQualifiedErrorId : System.Diagnostics.Eventing.Reader.EventLogException,
Microsoft.PowerShell.Commands.GetWinEventCommand
```

Inne polecenia, takie jak np. `Get-Service` czy `Get-Process`, nie mają parametru `credential` i przyjmują rolę aktualnie zalogowanego użytkownika. Oto przykład użycia jednego z tych poleceń:

```
PS C:\> Get-Service -ComputerName hyperv -Name bits
Status Name                               DisplayName
-----
Running bits                               Background Intelligent Transfer Ser...
```

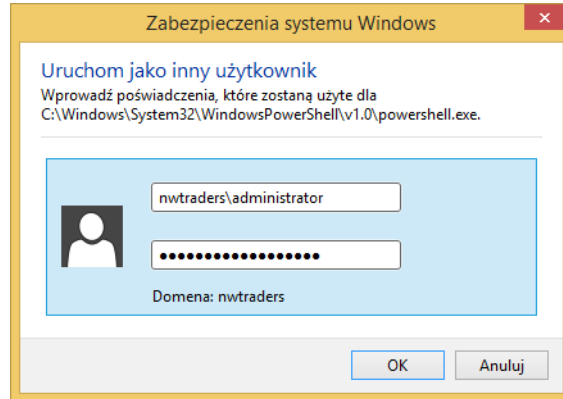
PS C:\>

To, że polecenie nie ma parametru pozwalającego wpisać dane poświadczające tożsamość, nie znaczy, że musi przyjmować rolę aktualnie zalogowanego użytkownika. Jeśli naciśniesz klawisz *Shift* i klikniesz prawym przyciskiem myszy ikonę Windows PowerShell, zostanie wyświetlone menu (pokazane na rysunku 21.2), za pomocą którego można uruchomić program jako inny użytkownik.



RYSUNEK 21.2. Za pomocą menu opcji konsoli Windows PowerShell można uruchomić konsolę jako inny użytkownik

Na rysunku 21.3 pokazano okno dialogowe, w którym należy wpisać dane poświadczające tożsamość innego użytkownika.



RYSUNEK 21.3. Okno dialogowe poświadczeń

Przy użyciu tego okna można wykonać jako dowolny użytkownik także te polecenia cmdlet, które nie mają parametru `credential`.

Zdalne zarządzanie systemem Windows

System Windows Server 2012 R2 standardowo instaluje się z usługą WinRM (ang. *Windows Remote Management* — zdalne zarządzanie systemem Windows) i obsługuje polecenia Windows PowerShell do pracy zdalnej. WinRM to opracowana przez firmę Microsoft implementacja standardu WS-Management Protocol. Umożliwia ona interaktywne łączenie się ze zdalnymi systemami bez tworzenia luk w zaporze sieciowej. Mechanizm ten jest wykorzystywany np. przez polecenia cmdlet CIM. Po zainstalowaniu i uruchomieniu systemu Windows Server 2012 R2 od razu można nawiązać zdalne połączenie i wykonywać polecenia albo można uruchomić interaktywną konsolę Windows PowerShell. Natomiast w systemie Windows 8.1 usługa WinRM jest zablokowana, więc najpierw trzeba ją skonfigurować za pomocą funkcji `Enable-PSRemoting`. Funkcja ta wykonuje następujące czynności:

1. Uruchomienie lub ponowne uruchomienie usługi WinRM.
2. Ustawienie automatycznego uruchamiania usługi WinRM.
3. Utworzenie odbiornika żądań z adresów wszystkich protokołów internetowych.
4. Włączenie wyjątków w zaporze sieciowej dopuszczających ruch WS-Man do wewnątrz.
5. Ustawienie docelowego odbiornika o nazwie `Microsoft.powershell`.
6. Ustawienie docelowego odbiornika o nazwie `Microsoft.powershell.workflow`.
7. Ustawienie docelowego odbiornika o nazwie `Microsoft.powershell32`.

W każdym kroku tego procesu funkcja wyświetla monit o potwierdzenie chęci wykonania danej czynności. Jeśli wiesz, jak wygląda przebieg działania funkcji, i nie planujesz zmieniać żadnych domyślnych ustawień, to możesz dodać parametr `force`, aby polecenie nie wyświetlało monitów. Oto przykład wywołania polecenia `Enable-PSRemoting` z parametrem `-Force`:

```
Enable-PSRemoting -Force
```

Poniżej natomiast znajduje się wynik wykonania funkcji Enable-PSRemoting w trybie interaktywnym.

```
PS C:\> Enable-PSRemoting
WinRM Quick Configuration
Running command "Set-WSManQuickConfig" to enable remote management of this computer by using the
Windows Remote Management (WinRM) service.
This includes:
    1. Starting or restarting (if already started) the WinRM service
    2. Setting the WinRM service startup type to Automatic
    3. Creating a listener to accept requests on any IP address
    4. Enabling Windows Firewall inbound rule exceptions for WS-Management traffic (for http only).

Do you want to continue?
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help (default is "Y"): y
Usługa WinRM została zaktualizowana na potrzeby zdalnego zarządzania.
Aby odbierać zapytania do usługi WS-Man kierowane na dowolny adres IP na tym komputerze, utworzono
odbiorcę usługi WinRM na HTTP://*.
Wyjątek zapory dla usługi WinRM został włączony.
Zasada LocalAccountTokenFilterPolicy została skonfigurowana w celu zdalnego udzielania uprawnień
administracyjnych użytkownikom lokalnym.
```

```
Confirm
Are you sure you want to perform this action?
Performing the operation "Set-PSSessionConfiguration" on target "Name: microsoft.powershell SDDL:
O:NSG:BAD:P(A;;GA;;;BA)(A;;GA;;;RM)S:P(AU;FA;GA;;;WD)(AU;SA;GXGW;;;WD)". This lets selected users
remotely run Windows
PowerShell commands on this computer."
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help (default is "Y"): y
```

```
Confirm
Are you sure you want to perform this action?
Performing the operation "Set-PSSessionConfiguration" on target "Name:
microsoft.powershell.workflow SDDL:
O:NSG:BAD:P(A;;GA;;;BA)(A;;GA;;;RM)S:P(AU;FA;GA;;;WD)(AU;SA;GXGW;;;WD)". This lets selected users
remotely run Windows
PowerShell commands on this computer."
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help (default is "Y"): y
```

```
Confirm
Are you sure you want to perform this action?
Performing the operation "Set-PSSessionConfiguration" on target "Name: microsoft.powershell132
SDDL:
O:NSG:BAD:P(A;;GA;;;BA)(A;;GA;;;RM)S:P(AU;FA;GA;;;WD)(AU;SA;GXGW;;;WD)". This lets selected users
remotely run Windows
PowerShell commands on this computer."
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help (default is "Y"): y
PS C:\>
```

Po zakończeniu konfiguracji wykonaj polecenie Test-WSMan, aby sprawdzić, czy usługa WinRM jest poprawnie skonfigurowana i czy przyjmuje zapytania. W poprawnie skonfigurowanym systemie powinny zostać wyświetlone następujące informacje:

```
PS C:\> Test-WSMan -ComputerName w8c504

wsmid           : http://schemas.dmtf.org/wbem/wsman/identity/1/wsmanidentity.xsd
ProtocolVersion : http://schemas.dmtf.org/wbem/wsman/1/wsman.xsd
```



```
ProductVendor : Microsoft Corporation
ProductVersion : OS: 0.0.0 SP: 0.0 Stack: 3.0
```

Polecenie to działa także w konsoli Windows PowerShell 2.0. Poniższy wynik pochodzi z kontrolera domeny działającego w systemie Windows 2008 z konsolą Windows PowerShell 2.0 i skonfigurowaną usługą WinRM:

```
PS C:\> Test-WSMan -ComputerName dc1
wsmid : http://schemas.dmtf.org/wbem/wsman/identity/1/wsmanidentity.xsd
ProtocolVersion : http://schemas.dmtf.org/wbem/wsman/1/wsman.xsd
ProductVendor : Microsoft Corporation
ProductVersion : OS: 0.0.0 SP: 0.0 Stack: 2.0
```

Jeżeli usługa WinRM nie jest skonfigurowana, system zwraca błąd. Poniżej znajduje się taki błąd z systemu klienta Windows 8.1:

```
PS C:\> Test-WSMan -ComputerName w8c10
Test-WSMan : <f:WSManFault xmlns:f="http://schemas.microsoft.com/wbem/wsman/1/wsmanfault"
Code="2150858770" Machine="w8c10.iammred.net"><f:Message>Klient nie może połączyć się z miejscem
docelowym określonym w żądaniu. Sprawdź, czy usługa w miejscu docelowym jest uruchomiona i
akceptuje żądania. Sprawdź dzienniki i zapoznaj się z dokumentacją usługi WS-Management
uruchomionej w miejscu docelowym, najczęściej jest to usługa IIS lub WinRM. Jeśli usługa docelowa
to WinRM, w miejscu docelowym uruchom następujące polecenie w celu przeanalizowania i
skonfigurowania usługi WinRM: "winrm quickconfig". </f:Message></f:WSManFault>
At line:1 char:1
+ Test-WSMan -ComputerName w8c10
+ ~~~~~
+ CategoryInfo          : InvalidOperation: (w8c10:String) [Test-WSMan], InvalidOperationException
+ FullyQualifiedErrorId : WsManError,Microsoft.WSMan.Management.TestWSManCommand
```

Pamiętaj, że konfiguracja usługi WinRM za pomocą funkcji Enable-PSRemoting nie powoduje utworzenia wyjątku zdalnego zarządzania w zaporze sieciowej, przez co domyślnie polecenia PING wysyłane do klienta Windows 8 nie będą działać.

```
PS C:\> ping w8c504
```

```
Pinging w8c504.iammred.net [192.168.0.56] with 32 bytes of data:
Request timed out.
Request timed out.
Request timed out.
Request timed out.
```

```
Ping statistics for 192.168.0.56:
    Packets: Sent = 4, Received = 0, Lost = 4 (100% loss).
```

Ale polecenia ping wysyłane do serwera Windows 2012 R2 Server działają:

```
PS C:\> ping Server1
```

```
Pinging Server1.iammred.net [192.168.0.57] with 32 bytes of data:
Reply from 192.168.0.57: bytes=32 time<1ms TTL=128
Reply from 192.168.0.57: bytes=32 time<1ms TTL=128
Reply from 192.168.0.57: bytes=32 time<1ms TTL=128
Reply from 192.168.0.57: bytes=32 time<1ms TTL=128
```

```
Ping statistics for 192.168.0.57:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
    Approximate round trip times in milli-seconds:
        Minimum = 0ms, Maximum = 0ms, Average = 0ms
```

Tworzenie zdalnej sesji Windows PowerShell

Jeśli trzeba utworzyć prostą konfigurację na jednym zdalnym komputerze, to najlepszym rozwiązaniem jest zdalna sesja Windows PowerShell. Aby otworzyć zdalną interaktywną sesję Windows PowerShell na komputerze docelowym, należy wykonać polecenie `Enter-PSSession`. Jeśli nie podasz danych poświadczających, zdalna sesja stanie się bieżącym użytkownikiem. Poniżej znajduje się wynik połączenia ze zdalnym komputerem o nazwie `dc1`. Po ustanowieniu połączenia w wierszu poleceń zostaje dodana nazwa zdalnego systemu. Polecenie `Set-Location (sl to jego alias)` zmienia katalog roboczy w zdalnym systemie na `c:\`. Następnie polecenie `Get-WmiObject` pobiera informacje o BIOS-ie zdalnego systemu. Polecenie `exit` zamyka zdalną sesję i wiersz poleceń wraca do pierwotnego stanu.

```
PS C:\> Enter-PSSession -ComputerName dc1
[dc1]: PS C:\Users\Administrator\Documents> sl c:\
[dc1]: PS C:\> gwmi win32_bios
```

```
SMBIOSBIOSVersion : A01
Manufacturer      : Dell Computer Corporation
Name              : Default System BIOS
SerialNumber      : 9HQ1S21
Version           : DELL - 6
```

```
[dc1]: PS C:\> exit
PS C:\>
```

Najlepsze jest to, że polecenie Windows PowerShell `Start-Transcript` przechwytywa wyniki zarówno zdalnej, jak i lokalnej sesji. W utworzonym przez nie wykazie ujęte są wszystkie wpisane polecenia. Poniżej znajduje się przykład ilustrujący rozpoczęcie rejestrowania wykazu, włączenie zdalnej sesji Windows PowerShell, wpisanie polecenia, zamknięcie sesji oraz zatrzymanie rejestrowania wykazu:

```
PS C:\> Start-Transcript
Transcript started, output file is
C:\Users\administrator.IAMMRED\Documents\PowerShell_transcript.20120701124414.txt
PS C:\> Enter-PSSession -ComputerName dc1
[dc1]: PS C:\Users\Administrator\Documents> gwmi win32_bios
```

```
SMBIOSBIOSVersion : A01
Manufacturer      : Dell Computer Corporation
Name              : Default System BIOS
SerialNumber      : 9HQ1S21
Version           : DELL - 6
```

```
[dc1]: PS C:\Users\Administrator\Documents> exit
PS C:\> Stop-Transcript
Transcript stopped, output file is
C:\Users\administrator.IAMMRED\Documents\PowerShell_transcript.20120701124414.txt
PS C:\>
```

Na rysunku 21.4 widać kopię wykazu z poprzedniej sesji.

Jeśli przewidujesz, że będzie trzeba wykonać wiele połączeń ze zdalnym systemem, to do utworzenia zdalnej sesji użyj polecenia `New-PSSession`. Polecenie to pozwala na zapisanie sesji w zmiennej oraz umożliwia wchodzenie do sesji i wychodzenie z niej dowolną liczbę razy, bez konieczności usuwania i tworzenia kolejnych sesji. W poniższym przykładzie zdalna sesja

```

*****
Windows PowerShell transcript start
Start time: 20120701124414
Username : IAMMRED\administrator
Machine : W8S504 (Microsoft Windows NT 6.28504.0)
*****

Transcript started, output file is C:\Users\administrator.IAMMRED\Documents
\PowerShell_transcript.20120701124414.txt
PS C:\> Enter-PSSession -ComputerName dc1
[dc1]: PS C:\Users\Administrator\Documents> gwmi win32_bios

SMBIOSBIOSVersion : A01
Manufacturer      : Dell Computer Corporation
Name              : Default System BIOS
SerialNumber      : 9HQ1S21
Version           : DELL - 6

[dc1]: PS C:\Users\Administrator\Documents> exit
PS C:\> Stop-Transcript
*****
Windows PowerShell transcript end
End time: 20120701124437
*****

```

RYSUNEK 21.4. Polecenie Start-Transcript rejestruje polecenia i wyniki otrzymywane ze zdalnej sesji Windows PowerShell

utworzona za pomocą polecenia `New-PSSession` zostaje zapisana w zmiennej `$dc1`. Następnie wchodzimy do niej za pomocą polecenia `Enter-PSSession`, pobieramy nazwę zdalnego hosta i wychodzimy za pomocą polecenia `exit`. Potem znowu wchodzimy do sesji, pobieramy ostatni proces i po raz kolejny wychodzimy. Na koniec za pomocą polecenia `Get-PSSession` pobieramy listę sesji Windows PowerShell i usuwamy je wszystkie za pomocą polecenia `Remove-PSSession`.

```

PS C:\> $dc1 = New-PSSession -ComputerName dc1 -Credential iammred\administrator
PS C:\> Enter-PSSession $dc1
[dc1]: PS C:\Users\Administrator\Documents> hostname
dc1
[dc1]: PS C:\Users\Administrator\Documents> exit
PS C:\> Enter-PSSession $dc1
[dc1]: PS C:\Users\Administrator\Documents> gps | select -Last 1

```

| Handles | NPM(K) | PM(K) | WS(K) | VM(M) | CPU(s) | Id | ProcessName |
|---------|--------|-------|-------|-------|--------|------|-------------|
| 292 | 9 | 39536 | 50412 | 158 | 1.97 | 2332 | wsmprovhost |

```

[dc1]: PS C:\Users\Administrator\Documents> exit
PS C:\> Get-PSSession

```

| Id | Name | ComputerName | State | ConfigurationName | Availability |
|----|----------|--------------|--------|----------------------|--------------|
| 8 | Session8 | dc1 | Opened | Microsoft.PowerShell | Available |

```

PS C:\> Get-PSSession | Remove-PSSession
PS C:\>

```

Wykonywanie pojedynczego polecenia Windows PowerShell

Jeśli trzeba wykonać tylko jedno polecenie Windows PowerShell, to nie ma sensu tworzyć całej interaktywnej sesji zdalnej. Zamiast tego lepiej użyć polecenia `Invoke-Command`. Jeżeli chcesz wykonać tylko jedno polecenie, to użyj polecenia `Invoke-Command`, podając nazwę komputera i dane poświadczające potrzebne do nawiązania połączenia. Poniżej przedstawiono przykład zastosowania tej techniki w celu sprawdzenia ostatniego procesu działającego na zdalnym serwerze `Ex1`:

```
PS C:\> Invoke-Command -ComputerName ex1 -ScriptBlock {gps | select -Last 1}
Handles NPM(K)    PM(K)          WS(K) VM(M)    CPU(s)      Id ProcessName      PSComputerName
-----
224      34    47164         51080   532      0.58    10164 wsmprovhost      ex1
```

Jeśli trzeba wykonać kilka poleceń albo przewidywanych jest więcej połączeń, to można użyć parametru sesji polecenia `Invoke-Command` w taki sam sposób jak w poleceniu `Enter-PSSession`. W poniższym przykładzie został utworzony nowy obiekt `PSSession` dla zdalnego komputera o nazwie `dc1`. Za pomocą sesji zdalnej pobrano dwie informacje. Po zakończeniu pracy sesja zapisana w zmiennej `$dc1` zostaje usunięta.

```
PS C:\> $dc1 = New-PSSession -ComputerName dc1 -Credential iammred\administrator
PS C:\> Invoke-Command -Session $dc1 -ScriptBlock {hostname}
dc1
PS C:\> Invoke-Command -Session $dc1 -ScriptBlock {Get-EventLog application -Newest 1}

Index Time                EntryType Source                                InstanceID Message PSCompu
-----
17702 Jul 01 12:59      Information ESENT                                701 DFSR... dc1
```

```
PS C:\> Remove-PSSession $dc1
```

Użycie w tym przykładzie polecenia `Invoke-Command` ilustruje jedną z największych zalet narzędzi do pracy zdalnej Windows PowerShell: możliwość wykonania jednego polecenia na dużej liczbie zdalnych systemów. Kluczem jest tutaj to, że parametr `computername` polecenia `Invoke-Command` przyjmuje tablicę nazw komputerów. W poniższym przykładzie zapisano tablicę nazw komputerów w zmiennej `$cn`. Następnie w zmiennej `$cred` zapisano obiekt poświadczeń dla zdalnych połączeń. Potem za pomocą polecenia `Invoke-Command` nawiązano połączenia z wszystkimi zdalnymi komputerami, aby pobrać informacje o ich BIOS-ie. Ciekawą cechą tej techniki jest dodatek parametru `PSComputerName` do zwracanego obiektu, co ułatwia powiązanie danych z komputerami. Poniżej znajdują się wszystkie opisane polecenia i ich wyniki:

```
PS C:\> $cn = "dc1","dc3","ex1","sql1","wsus1","wds1","hyperv1","hyperv2","hyperv3"
PS C:\> $cred = get-credential iammred\administrator
PS C:\> Invoke-Command -cn $cn -cred $cred -ScriptBlock {gwmi win32_bios}

SMBIOSBIOSVersion : BAP6710H.86A.0072.2011.0927.1425
Manufacturer       : Intel Corp.
Name                : BIOS Date: 09/27/11 14:25:42 Ver: 04.06.04
SerialNumber       :
Version            : INTEL - 1072009
PSComputerName     : hyperv3

SMBIOSBIOSVersion : A11
```

```

Manufacturer      : Dell Inc.
Name              : Phoenix ROM BIOS PLUS Version 1.10 A11
SerialNumber      : BDY91L1
Version           : DELL - 15
PSComputerName    : hyperv2

SMBIOSBIOSVersion : A01
Manufacturer      : Dell Computer Corporation
Name              : Default System BIOS
SerialNumber      : 9HQ1S21
Version           : DELL - 6
PSComputerName    : dc1

SMBIOSBIOSVersion : 090004
Manufacturer      : American Megatrends Inc.
Name              : BIOS Date: 03/19/09 22:51:32 Ver: 09.00.04
SerialNumber      : 3692-0963-10421-7503-9631-2546-83
Version           : VRTUAL - 3000919
PSComputerName    : wsus1

SMBIOSBIOSVersion : V1.6
Manufacturer      : American Megatrends Inc.
Name              : Default System BIOS
SerialNumber      : To Be Filled By O.E.M.
Version           : 7583MS - 20091228
PSComputerName    : hyperv1

SMBIOSBIOSVersion : 080015
Manufacturer      : American Megatrends Inc.
Name              : Default System BIOS
SerialNumber      : None
Version           : 091709 - 20090917
PSComputerName    : sql1

SMBIOSBIOSVersion : 080015
Manufacturer      : American Megatrends Inc.
Name              : Default System BIOS
SerialNumber      : None
Version           : 091709 - 20090917
PSComputerName    : wds1

SMBIOSBIOSVersion : 090004
Manufacturer      : American Megatrends Inc.
Name              : BIOS Date: 03/19/09 22:51:32 Ver: 09.00.04
SerialNumber      : 89921-9999-0865-2542-2186-80421-69
Version           : VRTUAL - 3000919
PSComputerName    : dc3

SMBIOSBIOSVersion : 090004
Manufacturer      : American Megatrends Inc.
Name              : BIOS Date: 03/19/09 22:51:32 Ver: 09.00.04
SerialNumber      : 2301-9053-4386-9162-8072-56621-16
Version           : VRTUAL - 3000919
PSComputerName    : ex1

PS C:\>

```

Zadania Windows PowerShell

Do tworzenia zadań Windows PowerShell służy polecenie `Start-Job`. Polecenie, które ma zostać wykonane w ramach zadania, umieszcza się w bloku skryptu, a poszczególnym zadaniom przypisywane są kolejno nazwy według wzoru `Job1`, `Job2` itd. W poniższym przykładzie widać zadanie `Job10`:

```
PS C:\> Start-Job -ScriptBlock {get-process}
Id      Name      PSJobTypeName State      HasMoreData Location
--      -
10      Job10     BackgroundJob Running     True       localhost

PS C:\>
```

Zadaniom przypisywane są też identyfikatory będące kolejnymi liczbami. Pierwsze zadanie utworzone w konsoli ma zawsze identyfikator 1. Do pobierania informacji o zadaniu można używać zarówno identyfikatora, jak i nazwy, jak pokazano poniżej:

```
PS C:\> Get-Job -Name job10
Id      Name      PSJobTypeName State      HasMoreData Location
--      -
10      Job10     BackgroundJob Completed   True       localhost

PS C:\> Get-Job -Id 10
Id      Name      PSJobTypeName State      HasMoreData Location
--      -
10      Job10     BackgroundJob Completed   True       localhost

PS C:\>
```

Po zakończeniu pracy przez zadanie można je odebrać. Służy do tego polecenie `Receive-Job`, które zwraca takie same informacje, jakie są zwracane wtedy, gdy zadanie jest nieużywane. Poniżej pokazano wynik zadania `Job1` (skrótowy dla zaoszczędzenia miejsca):

```
PS C:\> Receive-Job -Name job10
Handles  NPM(K)    PM(K)      WS(K)  VM(M)  CPU(s)      Id ProcessName
-----
62       9         1672       6032   80     0.00       1408 apdproxy
132      9         2316       5632   62     1364       atieclxx
122      7         1716       4232   32     948        atiesrxx
114      9         14664      15372  48     1492       audiodg
556     62        53928      5368   616    3.17       3408 CCC
58       8         2960       7068   70     0.19       928 conhost
32       5         1468       3468   52     0.00       5068 conhost
784     14        3284       5092   56     416        csrss
529     27        2928      17260  145    496        csrss
182     13        8184      11152  96     0.50       2956 DCPSysMgr
135     11        2880       7552   56     2056       DCPSysMgrSvc
... (dalsze wyniki zostały pominięte)
```

Po odebraniu zadania sprawa jest zakończona, tzn. dane znikają, chyba że zapisze się je w zmiennej. Ilustruje to poniższe polecenie:

```
PS C:\> Receive-Job -Name job10
PS C:\>
```

Co ciekawe, zadanie nadal istnieje i polecenie `Get-Job` nadal zwraca informacje o nim, jak widać poniżej:

```
PS C:\> Get-Job -Id 10
```

| Id | Name | PSJobTypeName | State | HasMoreData | Location |
|----|-------|---------------|-----------|-------------|-----------|
| 10 | Job10 | BackgroundJob | Completed | False | localhost |

Dlatego dobrym zwyczajem jest usuwanie pozostałości zadań za pomocą polecenia `Remove-Job`. Dzięki temu uniknie się zamętu z aktywnymi, ukończonymi i oczekującymi na wykonanie zadaniami. Po usunięciu zadania polecenie `Get-Job` zwraca błąd przy próbie pobrania informacji o nim, jak pokazano poniżej:

```
PS C:\> Remove-Job -Name job10
PS C:\> Get-Job -Id 10
Get-Job : The command cannot find a job with the job ID 10. Verify the value of the Id parameter
and then try the command again.
At line:1 char:1
+ Get-Job -Id 10
+ ~~~~~
+ CategoryInfo          : ObjectNotFound: (10:Int32) [Get-Job], PSArgumentException
+ FullyQualifiedErrorId : JobWithSpecifiedSessionNotFound,
    Microsoft.PowerShell.Commands.GetJobCommand
```

Kiedy używam poleceń do pracy z zadaniami, to zazwyczaj nadaję zadaniom własne nazwy. Na przykład zadanie zwracające obiekty procesów za pomocą polecenia `Get-Process` może mieć nazwę `getProc`. O wiele łatwiej zapamiętuje się takie kontekstowe nazwy niż rodzajowe oznaczenia `Job1`, `Job2` itd. Nie przejmuj się też długością swoich nazw zadań, ponieważ możesz używać symboli wieloznacznych. Przy odbieraniu zadania nie zapomnij zapisać otrzymanego obiektu w zmiennej, jak pokazano poniżej:

```
PS C:\> Start-Job -Name getProc -ScriptBlock {get-process}
```

| Id | Name | PSJobTypeName | State | HasMoreData | Location |
|----|---------|---------------|---------|-------------|-----------|
| 12 | getProc | BackgroundJob | Running | True | localhost |

```
PS C:\> Get-Job -Name get*
```

| Id | Name | PSJobTypeName | State | HasMoreData | Location |
|----|---------|---------------|-----------|-------------|-----------|
| 12 | getProc | BackgroundJob | Completed | True | localhost |

```
PS C:\> $procObj = Receive-Job -Name get*
PS C:\>
```

Obiektu zwróconego w zmiennej można użyć w innych poleceniach Windows PowerShell. Należy tylko pamiętać, że obiekt ten jest poddany deserializacji. Widać to w poniższym przykładzie, w którym użyłem aliasu `gm` polecenia `Get-Member`:

```
PS C:\> $procObj | gm
```

```
TypeName: Deserialized.System.Diagnostics.Process
```

Oznacza to, że nie wszystkie normalne składowe obiektu `.NET System.Diagnostics.Process` są dostępne. Poniżej widać normalne metody (`gps` jest aliasem polecenia `Get-Process`, `gm` jest aliasem polecenia `Get-Member`, a `-m` to wystarczająco długi fragment nazwy parametru `-membertype`, aby nie pomylić go z żadnym innym).

```
PS C:\> gps | gm -m method
```

```
    TypeName: System.Diagnostics.Process
```

| Name | MemberType | Definition |
|---------------------------|------------|--|
| ----- | ----- | ----- |
| BeginErrorReadLine | Method | System.Void BeginErrorReadLine() |
| BeginOutputReadLine | Method | System.Void BeginOutputReadLine() |
| CancelErrorRead | Method | System.Void CancelErrorRead() |
| CancelOutputRead | Method | System.Void CancelOutputRead() |
| Close | Method | System.Void Close() |
| CloseMainWindow | Method | bool CloseMainWindow() |
| CreateObjRef | Method | System.Runtime.Remoting.ObjRef CreateObjRef(type requestedType) |
| Dispose | Method | System.Void Dispose() |
| Equals | Method | bool Equals(System.Object obj) |
| GetHashCode | Method | int GetHashCode() |
| GetLifetimeService | Method | System.Object GetLifetimeService() |
| GetType | Method | type GetType() |
| InitializeLifetimeService | Method | System.Object InitializeLifetimeService() |
| Kill | Method | System.Void Kill() |
| Refresh | Method | System.Void Refresh() |
| Start | Method | bool Start() |
| ToString | Method | string ToString() |
| WaitForExit | Method | bool WaitForExit(int milliseconds), System.Void WaitForExit() |
| WaitForInputIdle | Method | bool WaitForInputIdle(int milliseconds), bool WaitForInputIdle() |

Poniżej znajdują się metody z obiektu poddanego deserializacji, zwrócone przez takie samo polecenie co wcześniej:

```
PS C:\> $procObj | gm -m method
```

```
    TypeName: Deserialized.System.Diagnostics.Process
```

| Name | MemberType | Definition |
|----------|------------|--|
| ----- | ----- | ----- |
| ToString | Method | string ToString(), string ToString(string format, System.IFormatProvider formatProvider) |

```
PS C:\>
```

Poniżej znajduje się lista wszystkich poleceń cmdlet zawierających w nazwie rzeczownik job:

```
PS C:\> Get-Command -Noun job | select name
```

| Name |
|-------------|
| ----- |
| Get-Job |
| Receive-Job |
| Remove-Job |
| Resume-Job |
| Start-Job |
| Stop-Job |
| Suspend-Job |
| Wait-Job |

Przy uruchamianiu zadania Windows PowerShell za pomocą polecenia `Start-Job` można zdefiniować nazwę dla zwróconego obiektu zadania. Można też zwrócony obiekt przypisać do zmiennej za pomocą zwykłego przypisania. Jeśli wykonasz obie te czynności, otrzymasz dwie kopie tego samego obiektu, jak pokazano poniżej:

```
PS C:\> $rtn = Start-Job -Name net -ScriptBlock {Get-Net6to4Configuration}
PS C:\> Get-Job -Name net
```

| Id | Name | PSJobTypeName | State | HasMoreData | Location |
|----|------|---------------|-----------|-------------|-----------|
| 18 | net | BackgroundJob | Completed | True | localhost |

```
PS C:\> $rtn
```

| Id | Name | PSJobTypeName | State | HasMoreData | Location |
|----|------|---------------|-----------|-------------|-----------|
| 18 | net | BackgroundJob | Completed | True | localhost |

Odebranie zadania za pomocą polecenia `Receive-Job` powoduje zużycie danych, tzn. nie da się za jakiś czas pobrać ich po raz drugi. Ilustruje to poniższy przykład:

```
PS C:\> Receive-Job $rtn
```

```
Description      : 6to4 Configuration
State             : Default
AutoSharing       : Default
RelayName         : 6to4.ipv6.microsoft.com.
RelayState        : Default
ResolutionIntervalSeconds : 1440
```

```
PS C:\> Receive-Job $rtn
```

Przykład ten pokazuje też, co się dzieje, gdy blok skryptu zwróci błąd. Jeśli użyte jest polecenie `Receive-Job`, następuje wyświetlenie informacji o błędzie. Aby dowiedzieć się, co dokładnie spowodowało błąd, należy użyć obiektu `Job` zapisanego w zmiennej `$rtn` lub pod nazwą `net`. Poniżej pokazano przykład użycia obiektu ze zmiennej `$rtn`:

```
PS C:\> $rtn.Command
Get-Net6to4Configuration
```

Aby zrobić porządek, należy usunąć pozostałości obiektów zadań poprzez pobranie ich i przekazanie do polecenia `Remove-Job`:

```
PS C:\> Get-Job | Remove-Job
PS C:\> Get-Job
PS C:\>
```

Nowo utworzone zadanie Windows PowerShell zostaje uruchomione w tle. Nie jest wyświetlana jakakolwiek informacja o tym, czy zadanie zostało wykonane poprawnie. Nie wiadomo nawet, czy w ogóle zostało wykonane. Jedynym sposobem na dowiedzenie się tego jest kilkakrotnie wykonanie polecenia `Get-Job` w celu sprawdzenia, kiedy zmieni się status. W wielu przypadkach nie jest to żadnym problemem, a nawet może być pożądane, jeśli chcemy odzyskać kontrolę nad konsolą od razu po rozpoczęciu wykonywania zadania. Jednak czasami dobrze by było otrzymać powiadomienie o wykonaniu zadania. W takim przypadku można użyć polecenia `Wait-Job`,

któremu należy przekazać identyfikator lub nazwę zadania. Jeśli to zrobisz, konsola Windows PowerShell pozostanie wstrzymana, dopóki zadanie się nie skończy. Gdy to nastąpi, w konsoli zostanie wyświetlona informacja o statusie. Wówczas za pomocą polecenia Receive-Job można odebrać poddane deserializacji obiekty i zapisać je w zmiennej:

```
PS C:\> $rtn = Start-Job -ScriptBlock {gwmi win32_product -cn hyperv1}
PS C:\> $rtn
```

| Id | Name | PSJobTypeName | State | HasMoreData | Location |
|----|-------|---------------|---------|-------------|-----------|
| -- | ---- | ----- | ----- | ----- | ----- |
| 22 | Job22 | BackgroundJob | Running | True | localhost |

```
PS C:\> Wait-Job -id 22
```

| Id | Name | PSJobTypeName | State | HasMoreData | Location |
|----|-------|---------------|-----------|-------------|-----------|
| -- | ---- | ----- | ----- | ----- | ----- |
| 22 | Job22 | BackgroundJob | Completed | True | localhost |

```
PS C:\> $prod = Receive-Job -id 22
PS C:\> $prod.Count
2
```

W nowo otwartej konsoli Windows PowerShell za pomocą polecenia Start-Job uruchomiono nowe zadanie. Zwrócony obiekt zapisano w zmiennej \$rtn. Można go przekazać do polecenia Stop-Job, aby zatrzymać wykonywanie zadania. Jeśli spróbujesz pobrać informacje o zadaniu bezpośrednio z obiektu zapisanego w zmiennej \$rtn, zostanie wygenerowany błąd, jak pokazano poniżej:

```
PS C:\> $rtn = Start-Job -ScriptBlock {gwmi win32_product -cn hyperv1}
PS C:\> $rtn | Stop-Job
PS C:\> Get-Job $rtn
Get-Job : The command cannot find the job because the job name
System.Management.Automation.PSRemotingJob was not found. Verify the value of the Name parameter,
and then try the command again.
At line:1 char:1
+ Get-Job $rtn
+ ~~~~~
+ CategoryInfo          : ObjectNotFound: (System.Manageme...n.PSRemotingJob: String) [Get-Job],
  PSArgumentException
+ FullyQualifiedErrorId : JobWithSpecifiedNameNotFound,Microsoft.
PowerShell.Commands.GetJobCommand
```

Obiekt zadania można przekazać do polecenia Get-Job, aby przekonać się, że zadanie jest zatrzymane. Za pomocą polecenia Receive-Job można odebrać informacje o zadaniu, a przy użyciu własności count można sprawdzić, ile produktów znajduje się w zmiennej, jak pokazano poniżej:

```
PS C:\> $rtn | Get-Job
Id      Name      PSJobTypeName  State      HasMoreData  Location
--      -
2       Job2      BackgroundJob  Stopped    False         localhost

PS C:\> $products = Receive-Job -Id 2
PS C:\> $products.count
0
```

Na liście tej brak jakichkolwiek pakietów oprogramowania. Jest to spowodowane tym, że polecenie `Get-WmiObject` nie zdążyło pobrać informacji z klasy `Win32_Product`.

Jeśli chcesz zachować dane zwrócone przez zadanie na później, ale nie chcesz do tego celu tworzyć tymczasowej zmiennej, użyj parametru `keep`. W poniższym przykładzie użyto polecenia `Get-NetAdapter` w celu pobrania informacji o karcie sieciowej:

```
PS C:\> Start-Job -ScriptBlock {Get-NetAdapter}
Id      Name      PSJobTypeName State      HasMoreData Location
--      -
4       Job4      BackgroundJob Running    True      localhost
```

Jeśli trzeba sprawdzić status przed chwilą utworzonego zadania, można użyć parametru `newest` zamiast numeru zadania, który trudniej zapamiętać. Poniżej pokazano przykład zastosowania tej techniki:

```
PS C:\> Get-Job -Newest 1
Id      Name      PSJobTypeName State      HasMoreData Location
--      -
4       Job4      BackgroundJob Completed  True      localhost
```

Aby odebrać informacje z tego zadania i zachować je do późniejszego użytku, należy dodać parametr `keep`, jak pokazano poniżej:

```
PS C:\> Receive-Job -Id 4 -Keep

ifAlias           : Ethernet
InterfaceAlias    : Ethernet
ifIndex           : 12
ifDesc            : Microsoft Hyper-V Network Adapter
ifName            : Ethernet_7
DriverVersion      : 6.2.8504.0
LinkLayerAddress   : 00-15-5D-00-2D-07
MacAddress         : 00-15-5D-00-2D-07
LinkSpeed          : 10 Gbps
MediaType          : 802.3
PhysicalMediaType  : Unspecified
AdminStatus        : Up
MediaConnectionState : Connected
DriverInformation  : Driver Date 2006-06-21 Version
                   : 6.2.8504.0 NDIS 6.30
DriverFileName     : netvsc63.sys
NdisVersion        : 6.30
ifOperStatus       : Up
RunspaceId         : 9ce8f8e6-1a09-4103-a508-c60398527
<reszta informacji opuszczona>
```

Pracę ze zwróconymi danymi można normalnie kontynuować, jak pokazano poniżej:

```
PS C:\> Receive-Job -Id 4 -Keep | select name

name
----
Ethernet

PS C:\> Receive-Job -Id 4 -Keep | select transmitlinksp*

TransmitLinkSpeed
-----
10000000000
```

Dodatkowe źródła informacji

- Centrum skryptów portalu TechNet pod adresem <http://www.microsoft.com/technet/scriptcenter> zawiera wiele przykładów skryptów.
- Na stronie <http://msdn.microsoft.com/en-us/library/bb822049.aspx> znajduje się opis historii wersji platformy .NET.
- Na stronie <http://msdn.microsoft.com/en-us/library/1h925568.aspx> można znaleźć informacje na temat sprawdzania numeru zainstalowanej wersji platformy .NET.
- Strona główna portalu MSDN znajduje się pod adresem <http://msdn.microsoft.com>.

Rozdział 22

Przebiegi pracy w Windows PowerShell

- Do czego służą przebiegi pracy w Windows PowerShell
- Równoległe wykonywanie poleceń
- Aktywności przebiegów pracy
- Ustalanie punktów kontrolnych dla przebiegów pracy w Windows PowerShell
- Dodawanie aktywności sekwencyjnej do przebiegu pracy
- Dodatkowe źródła informacji

Przy użyciu przebiegów pracy w Windows PowerShell specjaliści IT mogą rozwiązywać wiele typowych problemów dotyczących skryptów. Na przykład system musi zostać ponownie uruchomiony, aby dokończyć konfigurację przed uruchomieniem drugiego skryptu. Innym typowym przypadkiem jest wykonywanie zestawu poleceń w sposób równoległy lub szeregowy. Na początku tego rozdziału znajduje się opis sytuacji, w których uzasadnione jest użycie przebiegów pracy, a w dalszej części opisane są punkty kontrolne i sekwencjonowanie przebiegów pracy.

Do czego służą przebiegi pracy w Windows PowerShell

Zaletą przebiegów pracy w Windows PowerShell jest to, że polecenia składają się z sekwencji powiązanych ze sobą czynności. Przy ich użyciu można na przykład wykonywać długotrwałe polecenia. Przebieg pracy może przetrwać ponowne uruchomienie komputera, odłączenie sesji, a nawet może zostać zawieszony i ponownie uruchomiony bez utraty jakichkolwiek danych. Jest to możliwe dzięki temu, że przebiegi pracy automatycznie zapisują stan i dane na początku i na końcu działania. Ponadto można używać specjalnych samodzielnie zdefiniowanych punktów kontrolnych do zapisywania informacji. Jeśli wystąpi niemożliwa do naprawienia usterka, to można wznowić wykonywanie od ostatniego zapisanego punktu kontrolnego i nie rozpoczynać całego procesu od nowa.

UWAGA

System Windows PowerShell Workflow bazuje na Windows Workflow Foundation. Zamiast w języku XAML przepływy pracy można więc pisać przy użyciu składni Windows PowerShell. W razie potrzeby można też spakować przepływ pracy w moduł Windows PowerShell.

Dwa główne powody do użycia Windows PowerShell Workflow to niezawodność i wydajność przy wykonywaniu skomplikowanych lub długotrwałych poleceń. Powody te można rozbić na następujące kluczowe elementy:

1. Równoległe wykonywanie zadań.
2. Ograniczanie przepływu pracy.
3. Ograniczanie połączeń.
4. Buforowanie połączeń.
5. Integracja z sesjami rozłączania.

Wymagania przepływów pracy

Przepływ pracy wykorzystujący polecenia Windows PowerShell można wykonać, jeśli cel (zarządzany węzeł) ma zainstalowaną przynajmniej konsolę Windows PowerShell 2.0. Konsola ta jest niepotrzebna, jeżeli przepływ pracy nie zawiera żadnych poleceń Windows PowerShell. Na komputerach niezawierających tej konsoli można używać poleceń WMI i CIM, co oznacza, że przepływy pracy w Windows PowerShell można używać w zróżnicowanych środowiskach.

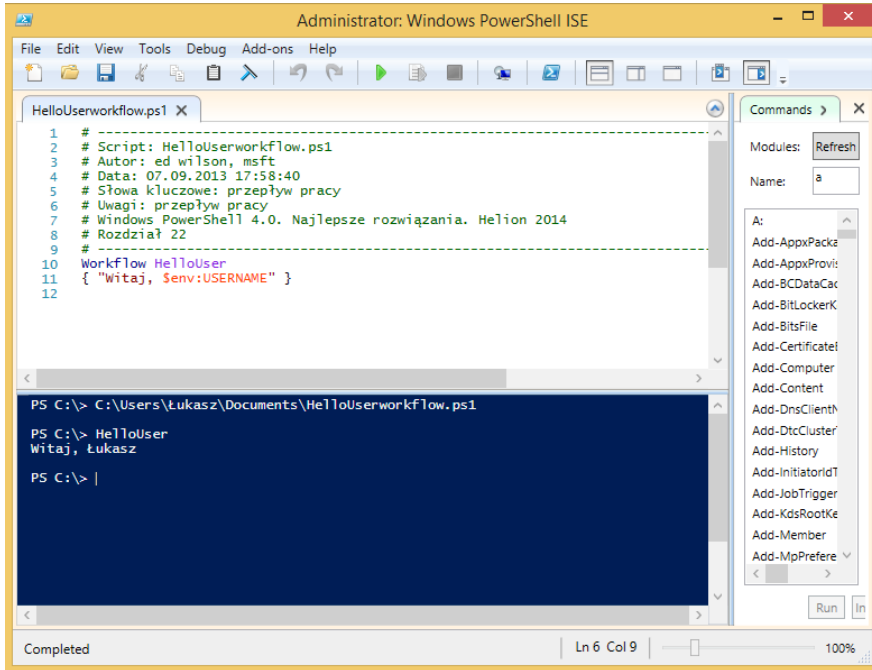
Komputer wykonujący przepływ pracy nazywa się gospodarzem (klientem, hostem). Musi mieć zainstalowaną przynajmniej konsolę Windows PowerShell 3.0 i włączone narzędzia do pracy zdalnej. Ponadto komputer docelowy (węzeł zarządzany) musi mieć przynajmniej konsolę Windows PowerShell 2.0 i włączone narzędzia do pracy zdalnej, jeśli w przepływie pracy używane są polecenia Windows PowerShell.

Prosty przepływ pracy

Choć przepływy pracy w Windows PowerShell to przede wszystkim sieciowe narzędzie do zarządzania używane na dużą skalę, można ich też używać w komputerach lokalnych, np. gdy trzeba wykonać jakieś czasochłonne zadanie. Jeśli więc dopiero poznajesz przepływy pracy, to dobrze jest zacząć naukę od utworzenia prostego przepływu działającego na własnym komputerze. Definicja przepływu pracy zaczyna się od słowa kluczowego `Workflow`, po którym należy wpisać nazwę przepływu pracy. Dalej wpisuje się blok skryptu w klamrze. Składnia ta jest bardzo podobna do definicji funkcji Windows PowerShell. Poniżej znajduje się przykładowa definicja przepływu pracy z pliku *HelloUserworkflow.ps1*.

```
Workflow HelloUser
{ "Witaj, $env:USERNAME" }
```

Podobnie jak w przypadku funkcji Windows PowerShell kod przepływu pracy należy załadować i wykonać, aby móc z niego korzystać. Jeśli skrypt zawierający przepływ pracy wykona się w konsoli Windows PowerShell ISE, to staje się on dostępny do użytku w wierszu poleceń, jak widać na rysunku 22.1.



RYSUNEK 22.1. Uruchomienie przepływu pracy w okienku skryptu, a następnie wykonanie go w wierszu poleceń

W przepływach pracy można używać normalnych poleceń i zwykłej logiki Windows PowerShell. Poniższy przepływ pracy pobiera za pomocą polecenia `Get-Date` godzinę w formacie 24-godzinny. Jeśli jest godzina wcześniejsza niż 12, wyświetla napis „miłego poranka”, między 12 a 18 wyświetla napis „dzień dobry”, a po godzinie 18 pisze „dobry wieczór”. Jest to kod źródłowy z pliku *HelloUserTimeWorkflow.ps1*.

HelloUserTimeWorkflow.ps1

```
Workflow HelloUserTime
{
    $dateHour = Get-date -UFormat '%H'
    if($dateHour -lt 12) {"miłego poranka"}
    ELSEIF ($dateHour -ge 12 -AND $dateHour -le 18) {"dzień dobry"}
    ELSE {"dobry wieczór"}
}
```

Równoległe wykonywanie poleceń

Jednym z powodów, dla których używa się przepływów pracy w Windows PowerShell, jest łatwe wykonywanie poleceń w sposób równoległy, co pozwala zaoszczędzić dużo czasu.

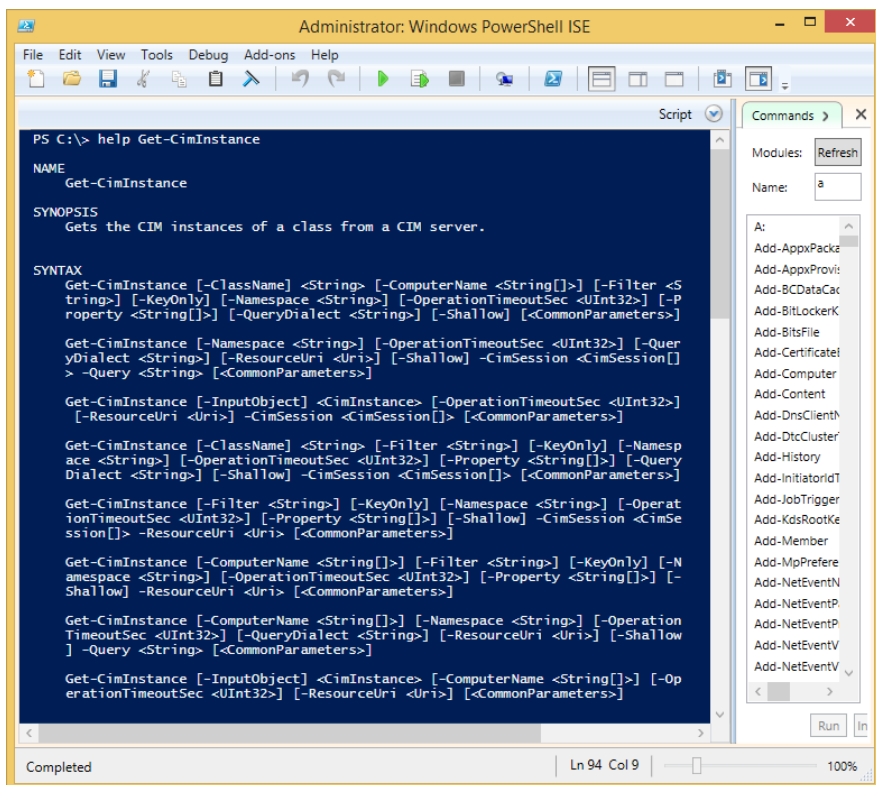
UWAGA

Przykłady ilustrujące oszczędność czasu uzyskiwaną dzięki użyciu przepływów pracy w Windows PowerShell i równoległemu wykonywaniu poleceń można znaleźć w doskonałym artykule „Use PowerShell Workflow to Ping Computers in Parallel”, napisanym przez MVP Windows PowerShell, Niklasa Goude'a (<http://blogs.technet.com/b/heyscriptingguy/archive/2012/11/20/use-powershell-workflow-to-ping-computers-in-parallel.aspx>).

Aby wykonać równoległą czynność przy użyciu przepływu pracy w Windows PowerShell, należy użyć słowa kluczowego `Foreach` z parametrem `-Parallel`. Za nimi wpisuje się operację i powiązany blok skryptu. Ilustruje to poniższy przykład:

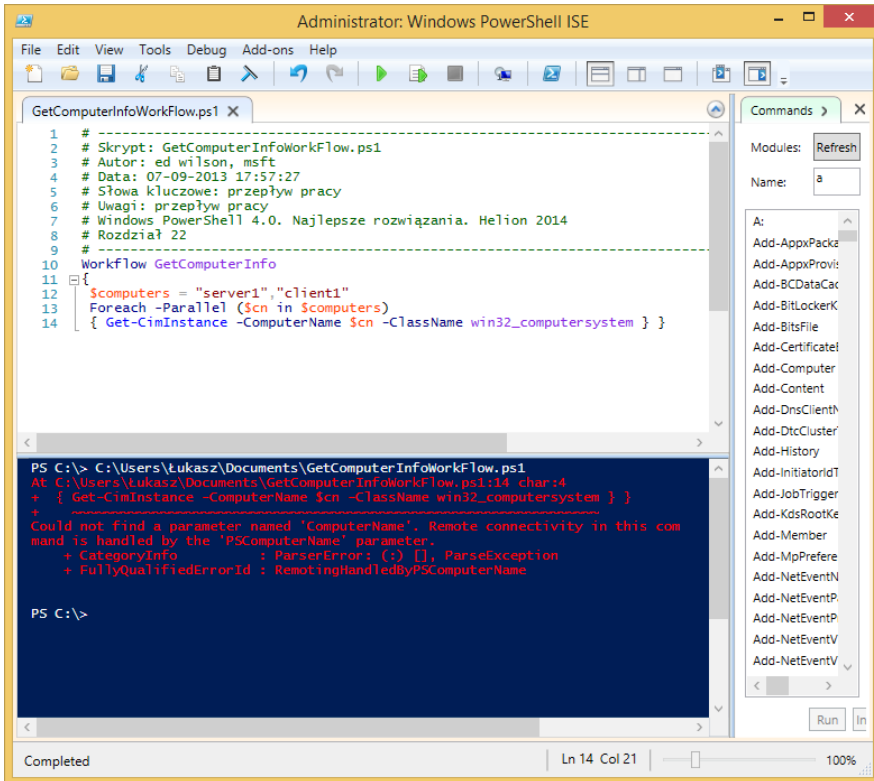
```
Foreach -Parallel ($c in $computers)
{ Get-CimInstance -PSComputerName $c -ClassName win32_computersystem }
```

Należy tylko pamiętać, że przy wywoływaniu polecenia `Get-CimInstance` z bloku skryptu równoległej pętli `Foreach` trzeba użyć automatycznie dodawanego parametru `PSComputerName`, a nie parametru `ComputerName`, którego normalnie używa się z tym poleceniem. W taki właśnie sposób przepływy pracy w Windows PowerShell obsługują nazwy komputerów. W składni polecenia `Get-CimInstance` w wierszu poleceń w ogóle nie znajdziesz parametru `PSComputerName`, jak widać na rysunku 22.2.



RYSUNEK 22.2. Polecenie `Get-CimInstance` nie ma parametru `PSComputerName`

Na szczęście, jeżeli zapomnisz o parametrze `-PSComputerName` i spróbujesz wykonać przepływ pracy, zostanie zgłoszony błąd. Informacje o nim są na tyle dokładne, że można się zorientować, w czym tkwi problem i jak go rozwiązać. Spójrz na rysunek 22.3.



RYСУNEK 22.3. Brak parametru `PSComputerName` powoduje wyświetlenie dokładnej informacji o błędzie

Po zmianie nazwy parametru w poleceniu `Get-CimInstance` przepływ pracy zostanie wykonany bezbłędnie.

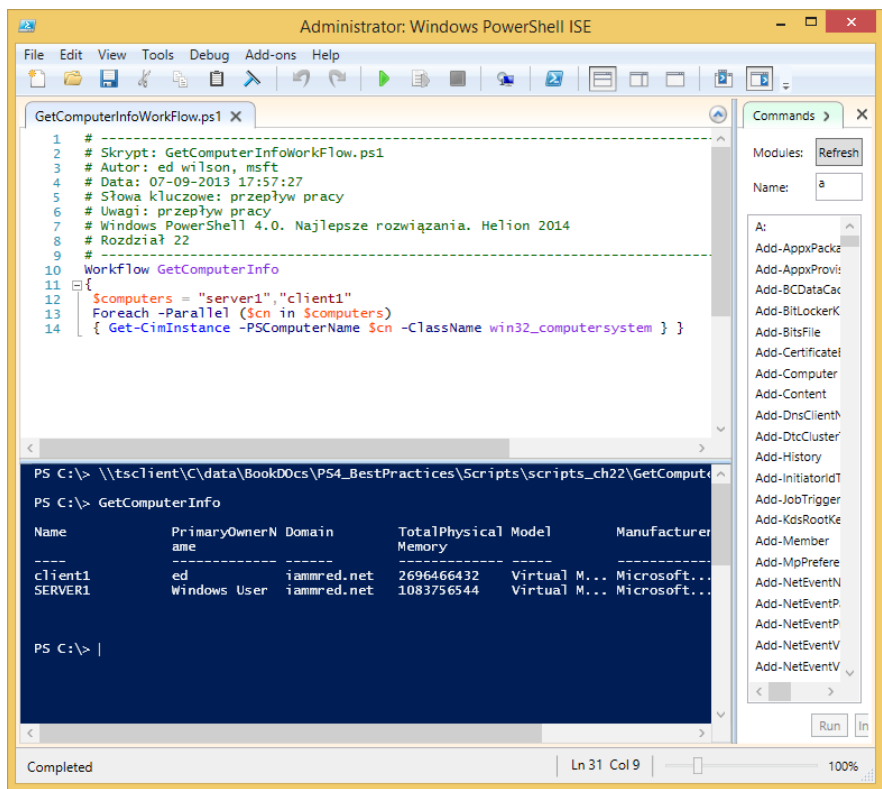
Poniżej znajduje się kompletny kod źródłowy skryptu `GetComputerInfoWorkflow.ps1`:

GetComputerInfoWorkflow.ps1

```

Workflow GetComputerInfo
{
    $computers = "server1","client1"
    Foreach -Parallel ($cn in $computers)
    { Get-CimInstance -PSComputerName $cn -ClassName win32_computersystem } }
  
```

Po wykonaniu tego przepływu pracy otrzymasz listę informacji o wszystkich serwerach, których nazwy są zapisane w zmiennej `$computers`. Cały skrypt i wynik jego działania pokazano na rysunku 22.4.



RYSUNEK 22.4. Przepływ pracy zwraca szczegółowe informacje o komputerach

Aktywności przepływów pracy

Przepływy pracy w Windows PowerShell składa się z kilku aktywności. Aktywności w ogóle są podstawową jednostką pracy w przepływach pracy w Windows PowerShell. Wyróżnia się pięć typów aktywności, a ich opis znajduje się w tabeli 22.1.

Polecenia Windows PowerShell jako aktywności

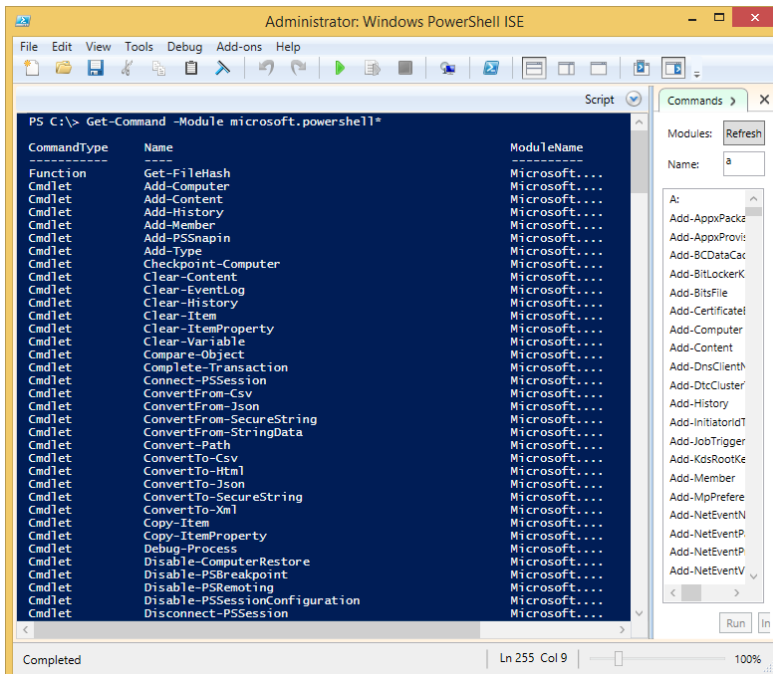
Polecenia Windows PowerShell z rdzennych modułów są automatycznie zaimplementowane jako aktywności, których można używać w przepływach pracy w Windows PowerShell. Wszystkie te moduły mają na początku nazwy przyrostek `Microsoft.PowerShell` i można je znaleźć za pomocą poniższego polecenia `Get-Command`:

```
Get-Command -Module microsoft.powershell*
```

Na rysunku 22.5 widać to polecenie i wynik jego wykonania.

TABELA 22.1. Aktywności przepływów pracy

| Aktywność | Opis |
|--|---|
| CheckPoint-Workflow (alias = PSPersist) | Pobiera punkt kontrolny. Zapisuje stan i dane przepływu pracy będącego w toku. Jeśli wykonywanie przepływu pracy zostanie przerwane lub ponownie uruchomione, to można wznowić działanie od dowolnego punktu kontrolnego. Przy użyciu aktywności Checkpoint-Workflow, parametru PSPersist i zmiennej PSPersistPreference można tworzyć niezawodne przepływy pracy. |
| ForEach -Parallel | Wykonuje instrukcje znajdujące się w bloku skryptu jeden raz dla każdego elementu kolekcji. Elementy są przetwarzane równolegle. Instrukcje bloku skryptu są wykonywane szeregowo. |
| Parallel | Wszystkie instrukcje znajdujące się w bloku skryptu mogą być wykonywane jednocześnie. Porządek wykonywania jest niezdefiniowany. |
| Sequence | Tworzy sekwencyjny blok instrukcji w równoległym skrypcie. Blok ten jest wykonywany równolegle z innymi aktywnościami w równoległym bloku skryptu. Instrukcje znajdujące się w bloku sekwencyjnym są wykonywane szeregowo w kolejności wpisania. Aktywności tej można używać tylko w równoległym bloku skryptu. |
| Suspend-Workflow | Zatrzymuje tymczasowo wykonywanie przepływu pracy. Aby wznowić wykonywanie, należy użyć polecenia Resume-Job. |



RYSUNEK 22.5. Rdzenne polecenia Windows PowerShell

Niedozwolone polecenia rdzenne

Nie wszystkich rdzennych poleceń Windows PowerShell można używać jako automatycznych aktywności przepływów pracy. Powodem jest to, że niektóre z tych poleceń źle współpracują z przepływami pracy. Wystarczy spojrzeć na ich listę, aby dowiedzieć się dlaczego. Lista ta znajduje się w tabeli 22.2.

TABELA 22.2. Niedozwolone rdzenne polecenia Windows PowerShell

| Polecenie | Polecenie |
|----------------------|---------------------|
| Add-History | Invoke-History |
| Add-PSSnapin | New-Alias |
| Clear-History | New-Variable |
| Clear-Variable | Out-GridView |
| Complete-Transaction | Remove-PSBreakpoint |
| Debug-Process | Remove-PSSnapin |
| Disable-PSBreakpoint | Remove-Variable |
| Enable-PSBreakpoint | Set-Alias |
| Enter-PSSession | Set-PSBreakpoint |
| Exit-PSSession | Set-PSDebug |
| Export-Alias | Set-StrictMode |
| Export-Console | Set-TraceMode |
| Get-Alias | Set-Variable |
| Get-History | Start-Transaction |
| Get-PSBreakpoint | Start-Transcript |
| Get-PSCallStack | Stop-Transcript |
| Get-PSSnapin | Trace-Command |
| Get-Transaction | Undo-Transaction |
| Get-Variable | Use-Transaction |
| Import-Alias | Write-Host |

Nieautomatyczne aktywności z poleceń

Jeśli polecenie nie należy do jednego z rdzennych modułów Windows PowerShell, to nie znaczy, że jest całkowicie wykluczone. Istnieje spora szansa, że jest wręcz odwrotnie. Jeśli więc w przepływie pracy w Windows PowerShell używane jest polecenie Windows PowerShell spoza rdzennego zestawu, konsola automatycznie wykona je jako aktywność `InlineScript`. Aktywność `InlineScript` umożliwia wykonywanie poleceń w przepływie pracy w Windows PowerShell i udostępnianie danych, które w innym przypadku mogłyby być niedostępne. W bloku `InlineScript` można wywoływać wszystkie polecenia Windows PowerShell i wyrażenia oraz współużytkować stan i dane w obrębie sesji. Dotyczy to także zaimportowanych modułów i wartości zmiennych. Na przykład polecenia wymienione w tabeli 22.2, których nie można używać w przepływach pracy w Windows PowerShell, mogą być używane w aktywnościach `InlineScript`.

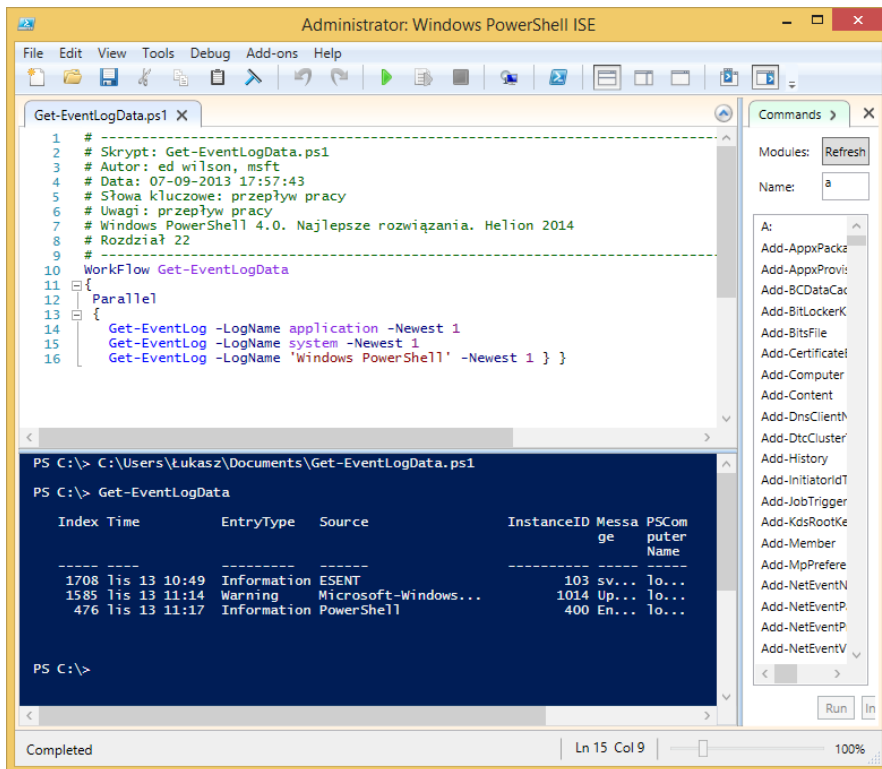
Aktywności równoległe

Aby utworzyć przepływ pracy w Windows PowerShell zawierający aktywność równoległą, należy użyć słowa kluczowego `Parallel`, po którym wpisuje się blok skryptu. Technikę tę zastosowano w przepływie pracy *Get-EventLogData.ps1*, którego kod źródłowy pokazano poniżej.

Get-EventLogData.ps1

```
Workflow Get-EventLogData
{
    Parallel
    {
        Get-EventLog -LogName application -Newest 1
        Get-EventLog -LogName system -Newest 1
        Get-EventLog -LogName 'Windows PowerShell' -Newest 1 } }
```

Po uruchomieniu skryptu zawierającego przepływ pracy *Get-EventLogData* należy przejść do wiersza poleceń konsoli Windows PowerShell ISE i wykonać ten przepływ pracy. Spowoduje to równoległe wykonanie polecenia *Get-EventLog*, którego efektem jest szybkie pobranie danych z dziennika zdarzeń. Jeśli w wywołaniu nie przekażesz żadnego parametru, wykonanie nastąpi na komputerze lokalnym, jak pokazano na rysunku 22.6.



RYСУNEK 22.6. Wykonanie przepływu pracy bez parametrów powoduje zwrócenie informacji o zdarzeniach

Do wielkich zalet przepływów pracy w Windows PowerShell należy dostęp do kilku automatycznych parametrów. Jednym z nich jest parametr `PSComputerName`. Mogę użyć tego parametru do uruchomienia przepływu pracy na dwóch zdalnych serwerach bez wykonywania dodatkowej pracy (ten przepływ pracy nie istnieje na żadnym z tych serwerów, a jedynie na mojej stacji roboczej).

Ustalanie punktów kontrolnych dla przepływów pracy w Windows PowerShell

Jeśli masz przepływ pracy w Windows PowerShell i musisz zapisać jego stan lub dane na dysku podczas działania, to możesz użyć punktu kontrolnego. Dzięki temu nie trzeba będzie zaczynać wszystkiego od nowa, tylko od ostatniego punktu kontrolnego, jeżeli coś przerwie wykonywanie przepływu pracy. Ustalanie punktów kontrolnych (ang. *checkpointing*) czasami opisuje się jako utrwalanie przepływów pracy (ang. *persisting*). Zważywszy na to, że przepływy pracy działają w dużych rozproszonych sieciach i operują czasochłonnymi zadaniami, mechanizmy obsługi przerw w pracy mają bardzo duże znaczenie.

Czym są punkty kontrolne

Punkt kontrolny to obraz bieżącego stanu przepływu pracy. Zawiera on aktualne wartości zmiennych i dotychczas wygenerowane wyniki. Przy tworzeniu punktu kontrolnego wszystkie te informacje zostają zapisane na dysku. W przepływie pracy można skonfigurować kilka punktów kontrolnych i jest kilka metod robienia tego. Niezależnie od zastosowanej techniki konsola Windows PowerShell do odzyskiwania sprawności i wznowiania przepływu pracy zawsze używa ostatniego punktu kontrolnego. Jeżeli przepływ pracy działa jako zadanie (np. poprzez użycie wspólnego parametru przepływów pracy `AsJob`), konsola Windows PowerShell zachowuje punkt kontrolny tego przepływu pracy do momentu usunięcia zadania (np. za pomocą polecenia `Remove-Job`).

Tworzenie punktów kontrolnych

Punkt kontrolny można umieścić w dowolnym miejscu przepływu pracy w Windows PowerShell, np. przed albo za dowolnym poleceniem lub którąkolwiek aktywnością. Ale trzeba mieć świadomość, że każdy punkt kontrolny zużywa zasoby i powoduje przerwę w przetwarzaniu przepływu pracy — czasami jest to nawet widoczne. Ponadto przy każdym wykonaniu przepływu pracy na komputerze docelowym następuje utworzenie punktu kontrolnego.

UWAGA

Gdzie najlepiej by było zdefiniować punkty kontrolne? Osobiście lubię je ustawiać za poważniejszymi fragmentami kodu, np. takimi, których wykonanie jest czasochłonne. Można też je ustawiać za sekcjami zużywającymi dużą ilość zasobów albo używającymi zasobów, które nie zawsze są dostępne.

Dodawanie punktów kontrolnych

Do przepływów pracy w Windows PowerShell można dodawać punkty kontrolne na kilku poziomach, np. na poziomie przepływu pracy albo aktywności. Dodanie punktu kontrolnego na poziomie przepływu pracy powoduje utworzenie punktu kontrolnego na początku i na końcu tego przepływu.

Punkty kontrolne na poziomie przepływu pracy są darmowe

Najłatwiejszym sposobem na dodanie punktu kontrolnego do przepływu pracy w Windows PowerShell jest użycie parametru `-PSPersist` przy wywoływaniu przepływu pracy.

Przepływ pracy zdefiniowany w pliku *Get-CompInfoWorkflowCheckPointWorkflow.ps1* pobiera informacje o karcie sieciowej, dysku i pojemności.

Get-CompInfoWorkflowCheckPointWorkflow.ps1

```
workflow Get-CompInfo
{
    Get-NetAdapter
    Get-Disk
    Get-Volume
}
```

Aby spowodować utworzenie punktu kontrolnego, należy wywołać przepływ pracy z parametrem `-PSPersist` ustawionym na `$true`. Poniżej znajduje się odpowiednie polecenie:

```
Get-CompInfo -PSComputerName server1, server2 -PSPersist $true
```

Po uruchomieniu przepływu pracy następuje pojawienie się paska postępu. Jego wykonanie z powodu punktów kontrolnych zajmuje kilka sekund. Na rysunku 22.7 widać ten pasek postępu.

Po utworzeniu punktów kontrolnych wykonywanie przepływu pracy przebiega bardzo szybko, po czym następuje wyświetlenie zgromadzonych informacji. Na rysunku 22.8 widać wynik oraz polecenia wiersza poleceń użyte do wywołania tego przepływu pracy.

Punkt kontrolny na poziomie aktywności

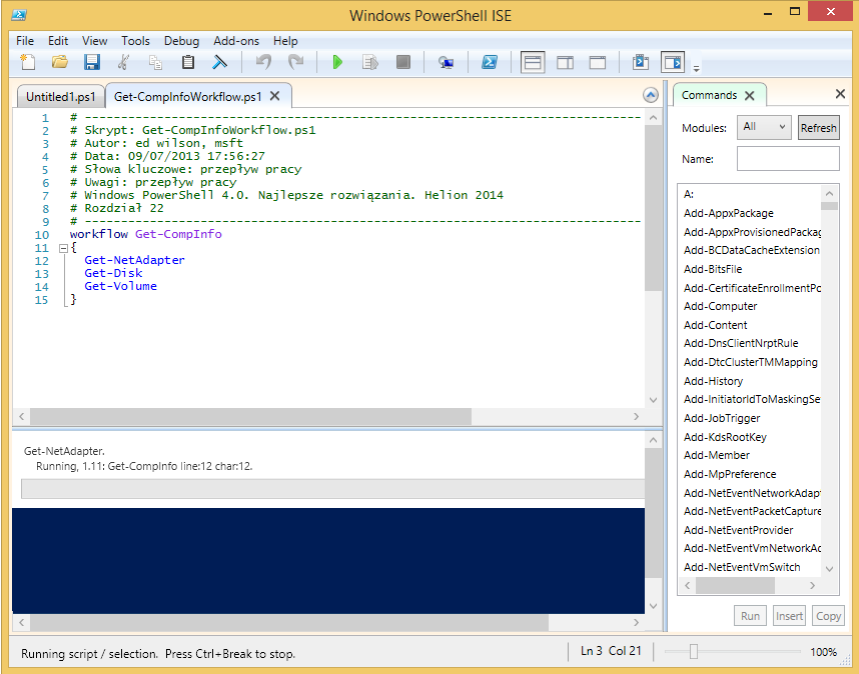
Jeśli używane jest rdzenne polecenie Windows PowerShell, to ma ono automatyczny parametr `PSPersist`. Wówczas można utworzyć punkt kontrolny na poziomie aktywności. Parametru `PSPersist` używa się w taki sam sposób jak na poziomie przepływu pracy. Aby spowodować wykonanie punktu kontrolnego, należy ustawić wartość na `$True`. Aby wyłączyć punkt kontrolny, wartość tę należy zmienić na `$False`.

W przepływie pracy zdefiniowanym w pliku *Get-CompInfoWorkflowPersist.ps1* punkt kontrolny został ustawiony po zakończeniu pierwszej i trzeciej aktywności.

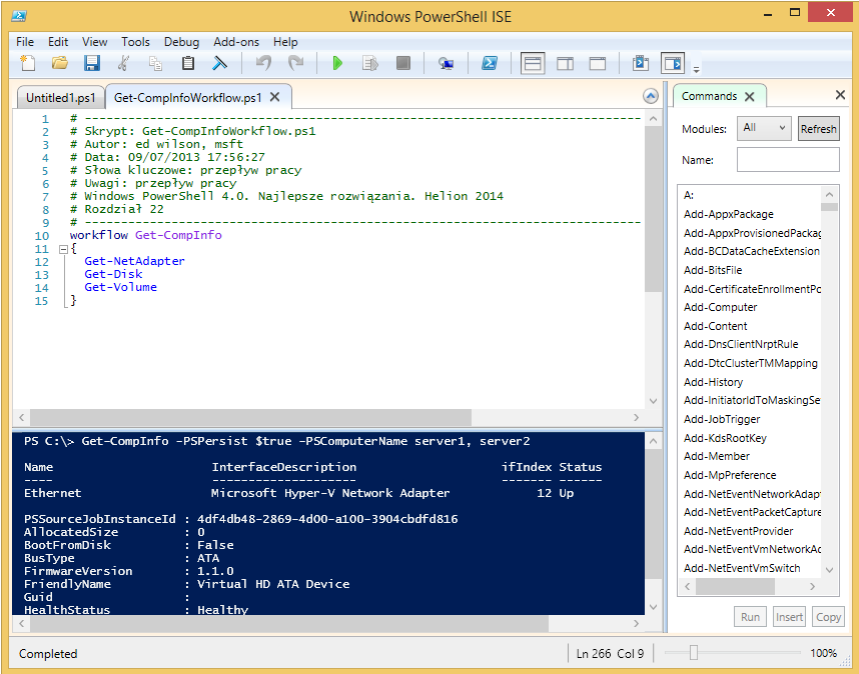
Get-CompInfoWorkflowPersist.ps1

```
workflow Get-CompInfo
{
    Get-process -PSPersist $true
    Get-Disk
    Get-service -PSPersist $true
}
```

W skrypcie tym przepływ pracy pobiera informacje o procesie i tworzy punkt kontrolny. Następnie wyświetla informacje o dysku i ponownie wykonuje punkt kontrolny.



RYSUNEK 22.7. Punkty kontrolne wydłużają czas wykonywania przepływów pracy



RYSUNEK 22.8. Po wykonaniu skryptu w celu wczytania przepływu pracy wywołano przepływ pracy na dwóch komputerach oraz utworzono punkty kontrolne

Sposób użycia aktywności Checkpoint-Workflow

Aktywność CheckPoint-Workflow powoduje natychmiastowe utworzenie punktu kontrolnego przez przepływ pracy. Można ją umieścić w dowolnym miejscu. Jej wielką zaletą jest to, że umożliwia tworzenie punktów kontrolnych w przepływach pracy, w których aktywnościami nie są rdzenne polecenia Windows PowerShell. Oznacza to, że punkty kontrolne można tworzyć np. w przepływach pracy zawierających polecenia Get-NetAdapter, Get-Disk czy Get-Volume. Użycie aktywności Checkpoint-Workflow jest konieczne z tego względu, że do poleceń nienależących do rdzennych modułów Windows PowerShell nie jest dodawany parametr -PSPersist. Poniżej znajduje się kod źródłowy skryptu *Get-CompInfoWorkflowCheckPointWorkflow.ps1*, w którym użyto tej aktywności.

Get-CompInfoWorkflowCheckPointWorkflow.ps1

```
workflow Get-CompInfo
{
    Get-NetAdapter
    Get-Disk
    Get-Volume
    Checkpoint-Workflow
}
```

Dodawanie aktywności sekwencyjnej do przepływu pracy

Aby dodać aktywność sekwencyjną do przepływu pracy w Windows PowerShell, należy użyć słowa kluczowego Sequence i dodać blok skryptu. Polecenia znajdujące się w tym bloku zostaną wykonane szeregowo w określonej kolejności. Kluczowe w tym przypadku jest to, że aktywność Sequence znajduje się w aktywności Parallel. Aktywność Sequence służy do wykonywania poleceń sekwencyjnie w zdefiniowanym porządku. W aktywności Parallel kolejność wykonywania poleceń jest nieuporządkowana. Polecenia znajdujące się w bloku Sequence są wykonywane równolegle z wszystkimi pozostałymi poleceniami znajdującymi się w aktywności Parallel. Ale polecenia z bloku Sequence są wykonywane w takiej kolejności, w jakiej je wpisano. Ilustracja tej techniki znajduje się w skrypcie *Get-WinFeatureServersWorkflow.ps1*, którego zawartość przedstawiono poniżej.

Get-WinFeatureServersWorkflow.ps1

```
workflow get-winfeatures
{
    Parallel {
        InlineScript {Get-WindowsFeature -Name PowerShell*}
        InlineScript {$env:COMPUTERNAME}
        Sequence {
            Get-date
            $PSVersionTable.PSVersion } }
}
```

W powyższym przepływie pracy kolejność wykonania skryptu śródliniowego `Get-WindowsFeature` i aktywności `Sequence` jest nieokreślona. Na pewno wiadomo jednak, że polecenie `Get-Date` zostanie wykonane przed pobraniem wartości własności `PSVersion`, ponieważ taka kolejność jest zdefiniowana w bloku `Sequence`.

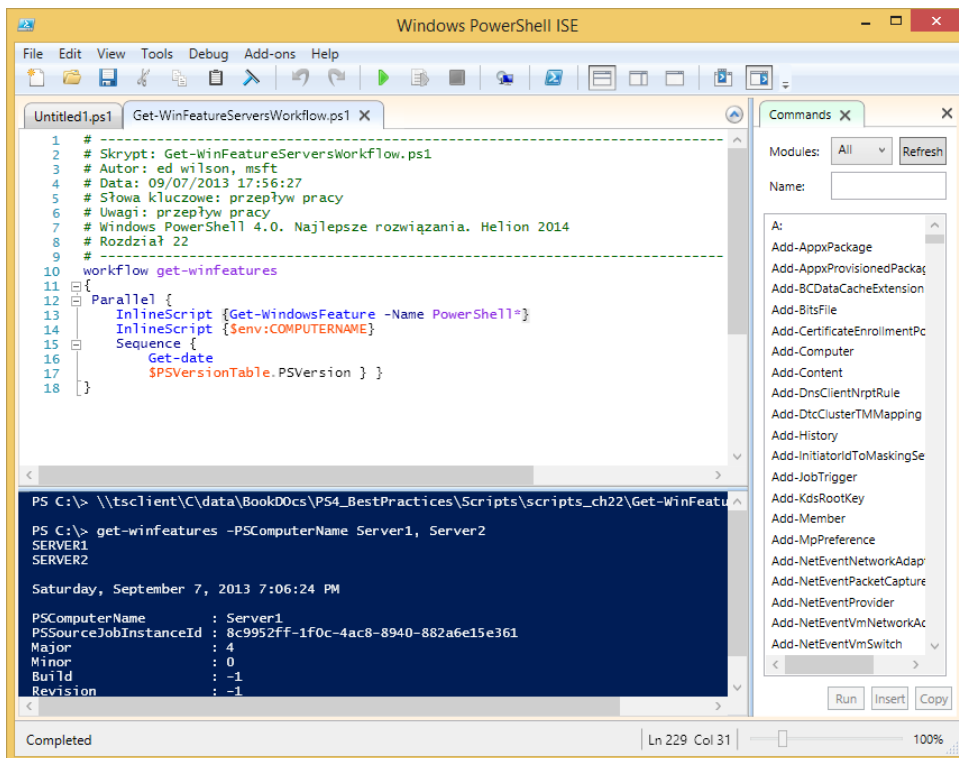
UWAGA

W Windows PowerShell 3.0 można było wywołać polecenie `Windows PowerShell` z systemu, który nie zawierał tego polecenia bezpośrednio w przepływie pracy. W Windows PowerShell 4.0 tego typu aktywności muszą znajdować się w aktywnościach `InlineScript`.

Aby wykonać przepływ pracy, najpierw uruchom zawierający go skrypt. Następnie wywołaj przepływ pracy i w parametrze `PSComputerName` prześlij mu dwie nazwy komputerów. Polecenie powinno wyglądać tak:

```
get-winfeatures -PSComputerName server1, server2
```

Na rysunku 22.9 pokazano okno konsoli Windows PowerShell ISE po wykonaniu tego przepływu pracy. Widać na nim kolejność wykonania poleceń. Zwróć uwagę, że polecenia z bloku `Sequence` zostały wykonane w takiej kolejności, w jakiej je zdefiniowano, tzn. najpierw `Get-Date`, a potem `$PsVersionTable.PSVersion`, mimo że znajdują się w bloku `Parallel`.



RYСУNEK 22.9. Kolejność wykonywania aktywności nie jest gwarantowana poza blokiem `Sequence`

Niektóre zalety przepływów pracy

Jedną z zalet tego przepływu pracy jest to, że wykonałem go na moim laptopie z systemem Windows 8.1. I co w tym takiego niezwykłego? To, że polecenie `Get-WindowsFeature` nie działa w systemach operacyjnych komputerów biurowych. Wykonałem na laptopie polecenie, którego nie ma na tym laptopie, ale jest na serwerach `Server1` i `Server2`. Wystarczyło tylko umieścić polecenie w aktywności `InlineScript`.

Inną zaletą przepływów sterowania jest dostępność aktywności `InlineScript`. Mogę uzyskać dostęp do zmiennej środowiskowej ze zdalnych serwerów. Aktywność `InlineScript` pozwala mi robić takie rzeczy, których w inny sposób nie dałoby się zrobić. Zwiększa elastyczność przepływów pracy.

Dodatkowe źródła informacji

- Centrum skryptów portalu TechNet pod adresem <http://www.microsoft.com/technet/scriptcenter> zawiera wiele przykładów skryptów.
- Na stronie <http://msdn.microsoft.com/en-us/vstudio/jj684582> znajduje się szczegółowa dokumentacja Windows Workflow Foundation.
- Na stronie <http://technet.microsoft.com/library/jj129719.aspx> znajduje się szczegółowy opis automatycznych parametrów przepływów pracy.
- Na stronie <http://technet.microsoft.com/en-us/library/jj574194.aspx> znajduje się szczegółowy opis specyficznych parametrów przepływów pracy.
- Wszystkie skrypty opisane w tym rozdziale są dostępne do pobrania w repozytorium centrum skryptów portalu TechNet pod adresem <http://gallery.technet.microsoft.com/scriptcenter/PowerShell-40-Best-d9e16039>.

Rozdział 23

Usługa konfiguracji żadanego stanu programu PowerShell

- Podstawowe informacje o Usłudze konfiguracji żadanego stanu programu PowerShell
- Kontrolowanie dryfu konfiguracji
- Dodatkowe źródła informacji

Zabójczą cechą konsoli Windows PowerShell 4.0 jest Usługa konfiguracji żadanego stanu programu (ang. *Desired State Configuration* — DSC). Wszystkie prezentacje na temat tej usługi przedstawione na konferencji TechEd 2013 zarówno w Ameryce Północnej, jak i w Europie zostały wysoko ocenione i były szeroko komentowane przez uczestników. Bez wątpienia jest to coś, co wywołało duże zainteresowanie w branży. Czym zatem jest Usługa konfiguracji żadanego stanu programu, jak jej używać, jakie wymagania trzeba spełnić, by móc ją zaimplementować oraz w czym może pomóc administratorowi?

Podstawowe informacje o Usłudze konfiguracji żadanego stanu programu PowerShell

Usługa DSC to zestaw rozszerzeń Windows PowerShell do zarządzania systemami, przeznaczona zarówno dla oprogramowania, jak i środowiska, w którym to oprogramowanie działa. Jako że DSC wchodzi w skład systemu Windows Management Framework 4.0 (który zawiera Windows PowerShell 4.0), usługa ta jest niezależna od systemu operacyjnego i można ją uruchamiać w dowolnym komputerze, w którym działa środowisko Windows PowerShell 4.0. Usługa DSC udostępnia następujących dostawców zasobów:

- Archive
- Environment
- File
- Group
- Log
- Package
- Registry

- Script
- Service
- User
- WindowsFeature
- WindowsProcess

Każdy z tych dostawców ma standardowy zestaw własności konfiguracyjnych. W tabeli 23.1 znajduje się ich szczegółowy wykaz.

TABELA 23.1. Dostawcy zasobów DSC i ich własności

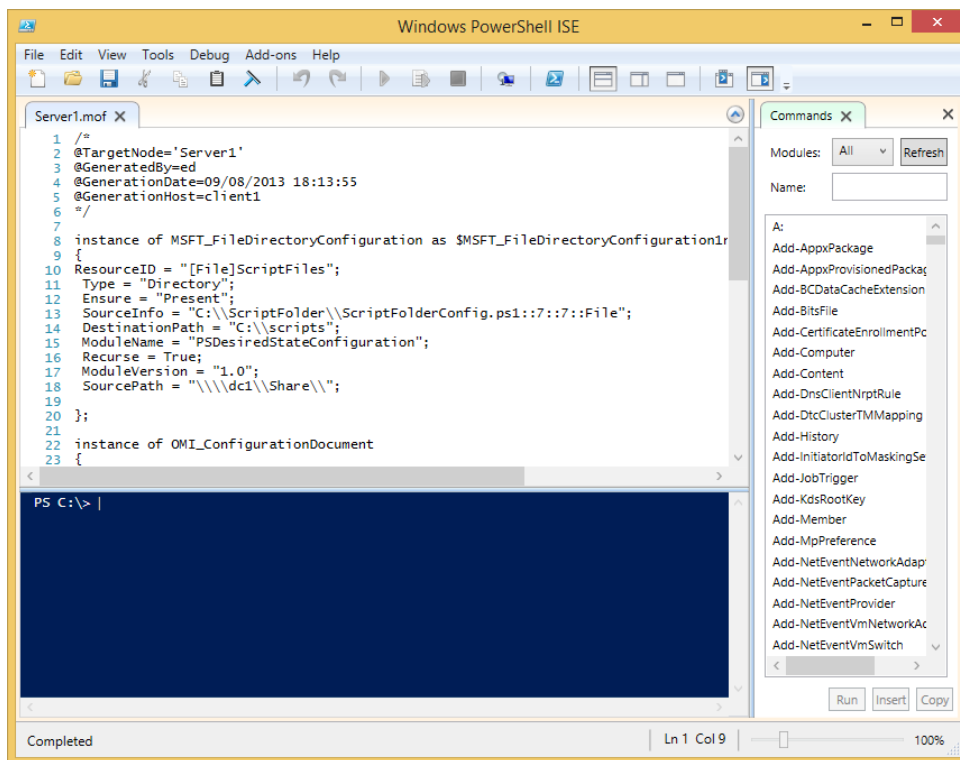
| Dostawca | Własności |
|----------------|--|
| Archive | Destination, Path, Checksum, DependsOn, Ensure, Force, Validate |
| Environment | Name, DependsOn, Ensure, Path, Value |
| File | DestinationPath, Attributes, Checksum, Contents, Credential, DependsOn, Ensure, Force, MatchSource, Recurse, SourcePath, Type |
| Group | GroupName, Credential, DependsOn, Description, Ensure, Members, MembersToExclude, MembersToInclude |
| Log | Message, DependsOn |
| Package | Name, Path, ProductId, Arguments, Credential, DependsOn, Ensure, LogPath, ReturnCode |
| Registry | Key, ValueName, DependsOn, Ensure, Force, Hex, ValueData, ValueType |
| Script | GetScript, SetScript, TestScript, Credential, DependsOn |
| Service | Name, BuiltInAccount, Credential, DependsOn, StartupType, State |
| User | UserName, DependsOn, Description, Disabled, Ensure, FullName, Password, PasswordChangeNotAllowed, PasswordChangeRequired, PasswordNeverExpires |
| WindowsFeature | Name, Credential, DependsOn, Ensure, IncludeAllSubFeature, LogPath, Source |
| WindowsProcess | Arguments, Path, Credential, DependsOn, Ensure, StandardErrorPath, StandardInputPath, StandardOutputPath, WorkingDirectory |

Nie trzeba jednak ograniczać się tylko do tych dwunastu standardowych dostawców zasobów, ponieważ można też tworzyć własnych.

Proces DSC

Do utworzenia konfiguracji przy użyciu DSC potrzebny jest plik MOF (ang. *Managed Object Format*). Składni MOF używa Instrumentacja zarządzania Windows (WMI), więc jest to standardowy format tekstowy. Na rysunku 23.1 widać przykładowy plik MOF dla serwera o nazwie Server1.

Własny plik MOF najłatwiej jest utworzyć przez utworzenie skryptu konfiguracyjnego DSC i wywołanie jednego z 12 standardowych dostawców albo przy użyciu swojego dostawcy. Definicja skryptu konfiguracyjnego zaczyna się od słowa kluczowego `Configuration` i nazwy konfiguracji. Następnie otwiera się blok skryptu, w którym określa się węzeł (node) i dostawcę zasobów. Węzeł to komputer docelowy konfiguracji. W skrypcie `ScriptFolderConfi.ps1` znajduje się konfiguracja



RYSUNEK 23.1. Plik DSC MOF to plik tekstowy w takim samym formacie jak używany w WMI

tworząca katalog na serwerze docelowym o nazwie `Server1`. Wykorzystuje ona dostawcę `File`. Pliki źródłowe są kopiowane z udziału sieciowego. Parametr `DestinationPath` określa folder, jaki ma zostać utworzony na serwerze. Parametr `Type` informuje, że utworzony ma zostać katalog. Parametr `Recurse` w tym przypadku oznacza, że skopiowane mają zostać wszystkie foldery ze ścieżki źródłowej. Poniżej znajduje się kompletny kod źródłowy skryptu `ScriptFolderConf.ps1`:

ScriptFolderConf.ps1

```

Configuration ScriptFolder
{
    node 'Server1'
    {
        File ScriptFiles
        {
            SourcePath = "\\dc1\Share\"
            DestinationPath = "C:\scripts"
            Ensure = "Present"
            Type = "Directory"
            Recurse = $true
        }
    }
}

```

Po wykonaniu skryptu *ScriptFolderConfig.ps1* w środowisku Windows PowerShell ISE następuje załadowanie konfiguracji *ScriptFolder* do pamięci. Następnie konfigurację tę można wywołać w taki sam sposób jak funkcję. Gdy się to zrobi, utworzy ona plik MOF dla każdego zdefiniowanego węzła. Ścieżki do konfiguracji używa się w wywołaniu polecenia *Start-DscConfiguration*. Jak widać, cały proces składa się z następujących trzech etapów:

1. Wykonanie skryptu zawierającego konfigurację w celu wczytania jej do pamięci.
2. Wywołanie konfiguracji i przekazanie parametrów wymaganych do utworzenia pliku MOF dla każdego określonego węzła.
3. Wywołanie polecenia *Start-DscConfiguration* i przekazanie ścieżki do plików MOF utworzonych w punkcie 2.

Proces ten jest widoczny na rysunku 23.2. Konfiguracja znajduje się w górnym okienku, natomiast w wierszu poleceń widać, że najpierw wykonano skrypt, następnie wywołano konfigurację, a następnie rozpoczęto konfigurację za pomocą plików MOF.

The screenshot shows the Windows PowerShell ISE interface. The top pane displays the content of *ScriptFolderConfig.ps1*, which defines a DSC configuration named *ScriptFolder* for a node named *Server1*. The configuration includes a *File* resource named *ScriptFiles* that copies files from *\\dc1\Share* to *C:\scripts*, ensuring they are present as a directory and recursing into subdirectories.

```
1 #Requires -version 4.0
2
3 Configuration ScriptFolder
4 {
5     node 'Server1'
6     {
7         File ScriptFiles
8         {
9             SourcePath = "\\dc1\Share\"
10            DestinationPath = "C:\scripts"
11            Ensure = "Present"
12            Type = "Directory"
13            Recurse = $true
14        }
15    }
16 }
17 }
```

The bottom pane shows the PowerShell prompt where the script is executed, followed by the *ScriptFolder* configuration being loaded. The output shows the directory *C:\ScriptFolder* containing a file named *Server1.mof* with a length of 1336 bytes, last written on 9/8/2013 at 6:24 PM.

```
PS C:\> C:\ScriptFolder\ScriptFolderConfig.ps1
PS C:\> ScriptFolder

Directory: C:\ScriptFolder

Mode                LastWriteTime         Length Name
----                -
-a---             9/8/2013  6:24 PM          1336 Server1.mof

PS C:\> Start-DscConfiguration -Path c:\ScriptFolder
```

The final output shows a table of running DSC jobs:

| Id | Name | PSJobTypeName | State | HasMoreData | Location | Comments |
|----|-------|-----------------|---------|-------------|----------|----------|
| 11 | Job11 | Configuratio... | Running | True | Server1 | Star |

The status bar at the bottom indicates the session is 'Completed' at line 23, column 9, with a zoom level of 100%.

RYСУNEK 23.2. Aby wykonać konfigurację na zdalnym serwerze, należy wykonać polecenie *Start-DscConfiguration* i przekazać mu ścieżkę do folderu zawierającego odpowiednie pliki MOF

Parametry konfiguracji

Do tworzenia parametrów konfiguracji używa się słowa kluczowego `param` w podobny sposób jak w funkcjach. Instrukcję `param` należy wpisać bezpośrednio za znakiem otwierającym blok skryptu konfiguracji. Jak widać w skrypcie *ScriptFolderVersion.ps1*, parametrom można nawet przypisywać wartości domyślne. Podczas tworzenia konfiguracja automatycznie otrzymuje trzy parametry domyślne: `instancename`, `outputpath` oraz `configurationdata`. Wartość `instancename` konfiguracji służy do identyfikowania każdego zasobu określonego w konfiguracji — w typowej sytuacji dobre jest ustawienie domyślne. Parametr `outputpath` przechowuje ścieżkę do zapisania pliku MOF konfiguracji. Za jego pomocą można spowodować zapisanie tego pliku w innym folderze niż folder, w którym znajduje się uruchomiony skrypt. Domyślnie pliki MOF tworzone są w tym samym folderze, w którym znajduje się uruchomiony skrypt tworzący konfigurację. Możliwość zapisania ich w innym miejscu ułatwia pracę z nimi i ich modyfikowanie. Parametr `configurationdata` przyjmuje tablicę skrótów z danymi konfiguracji. Ponadto przy wywoływaniu konfiguracji dostępne są wszystkie parametry zdefiniowane w sekcji `param`.

Proces tworzenia pliku MOF można uprościć, wywołując konfigurację bezpośrednio ze skryptu ją tworzącego. W skrypcie *ScriptFolderVersion.ps1* do konfiguracji dodano drugiego dostawcę zasobów o nazwie `Registry`. Za jego pomocą tworzony jest klucz rejestru `ForScripting` w kluczu `HKLM\Software`. Nazwa wartości rejestru to `ScriptsVersion`, a w danych podano wartość `1.0`. Poniżej znajduje się opisywany fragment kodu:

```
Registry AddScriptVersion
{
    Key = "HKEY_Local_Machine\Software\ForScripting"
    ValueName = "ScriptsVersion"
    ValueData = "1.0"
    Ensure = "Present"
}
```

Wywołanie dodatkowego dostawcy zasobów znajduje się od razu pod klamrą zamykającą blok poprzedniego dostawcy, którym jest `File`.

Poniżej znajduje się kompletny kod źródłowy skryptu *ScriptFolderVersion.ps1*:

ScriptFolderVersion.ps1

```
Configuration ScriptFolderVersion
{
    Param ($server = 'server1')
    node $server
    {
        File ScriptFiles
        {
            SourcePath = "\\dc1\Share\"
            DestinationPath = "C:\scripts"
            Ensure = "present"
            Type = "Directory"
            Recurse = $true
        }
        Registry AddScriptVersion
        {
            Key = "HKEY_Local_Machine\Software\ForScripting"
            ValueName = "ScriptsVersion"
```

```

        ValueData = "1.0"
        Ensure = "Present"
    }

}

ScriptFolderVersion

```

Ustawianie zależności

Przy wywoływaniu konfiguracji DSC nie wszystko dzieje się naraz i dlatego konieczne jest zapewnienie wykonywania działań w odpowiedniej kolejności. Służy do tego słowo kluczowe `DependsOn`. Na przykład w skrypcie *ScriptFolderVersionUnzip.ps1* użyto dostawcy zasobów `Archive` do rozpakowania skompresowanego pliku, który jest kopiowany ze wspólnego folderu. Pliki skryptów są kopiowane z udziału przy użyciu aktywności `ScriptFiles` wspieranej przez dostawcę `File`. Jako że pliki te muszą być pobrane z udziału sieciowego przed rozpakowaniem skompresowanego folderu, użyto słowa kluczowego `DependsOn`. Ponieważ aktywność `ScriptFiles` tworzy strukturę folderów zawierającą ten skompresowany folder, ścieżkę używaną przez dostawcę `Archive` można wpisać na sztywno. Ścieżka ta jest lokalna w odniesieniu do serwera, na którym uruchomiono konfigurację. Poniżej znajduje się kod opisywanej aktywności `Archive`:

```

Archive ZippedModule
{
    DependsOn = "[File]ScriptFiles"
    Path = "C:\scripts\PoshModules\PoshModules.zip"
    Destination = $modulePath
    Ensure = "Present"
}

```

Skrypt *ScriptFolderVersionUnzip.ps1* pobiera ścieżkę do folderu z modułami Windows PowerShell znajdującego się w katalogu *Program Files* ze zmiennej środowiskowej `$env:PSModulePath`. Następnie wywołuje konfigurację i przekierowuje plik MOF do folderu *C:\Server1\Config*. Później wywołuje polecenie `Start-DscConfiguration`, przekazując mu nazwę zadania oraz parametr `-Verbose`, aby zapewnić bardziej szczegółowe informacje o postępie. Poniżej znajduje się kompletny kod źródłowy skryptu *ScriptFolderVersionUnzip.ps1*:

```

ScriptFolderVersionUnzip.ps1

Configuration ScriptFolderVersionUnzip
{
    Param ($modulePath = ($env:PSModulePath -split ';' |
        ? {$_ -match 'Program Files'}),
        $Server = 'Server1')
    node $Server
    {
        File ScriptFiles
        {
            SourcePath = "\\dc1\Share\"
            DestinationPath = "C:\scripts"
            Ensure = "present"
            Type = "Directory"
            Recurse = $true
        }
    }
}

```

```

    }
    Registry AddScriptVersion
    {
        Key = "HKEY_Local_Machine\Software\ForScripting"
        ValueName = "ScriptsVersion"
        ValueData = "1.0"
        Ensure = "Present"
    }
    Archive ZippedModule
    {
        DependsOn = "[File]ScriptFiles"
        Path = "C:\scripts\PoshModules\PoshModules.zip"
        Destination = $modulePath
        Ensure = "Present"
    }
}

ScriptFolderVersionUnZip -output C:\server1Config
Start-DscConfiguration -Path C:\server1Config -JobName Server1Config -Verbose

```

Dane konfiguracji

Aby zmienić sposób działania konfiguracji, należy zdefiniować dane konfiguracji. Można to zrobić w osobnym pliku albo przy użyciu tablicy tablic skrótów. Aby utworzyć lokalnego użytkownika, należy dodać do danych konfiguracji ustawienie `PSDscAllowPlainTextPassword = $true` — jest to konieczne, nawet jeśli nie podaje się bezpośrednio hasła jako zwykłego tekstu. W skrypcie konfiguracyjnym *DemoUserConfig.ps1* dane poświadczające tożsamość użytkownika są dostarczane do konfiguracji przez polecenie `Get-Credential`. Tworzy ono bezpieczny łańcuch. Ale błąd wygenerowany podczas wykonywania konfiguracji informuje, że zaszyfrowane hasło w postaci tekstu można zapisać tylko pod warunkiem, że zostanie wyrażona na to zgoda w konfiguracji. Błąd ten widać na rysunku 23.3.

Poniżej znajduje się kompletny kod źródłowy skryptu *DemoUserConfig.ps1*:

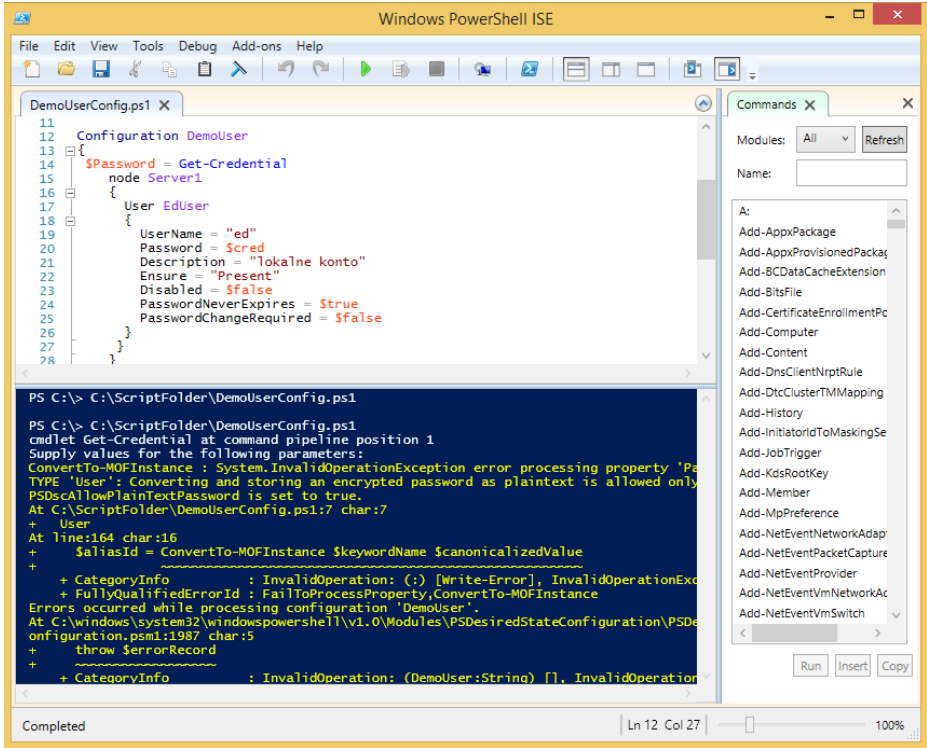
```

DemoUserConfi.ps1

Configuration DemoUser
{
    $Password = Get-Credential
    node Server1
    {
        User EdUser
        {
            UserName = "ed"
            Password = $cred
            Description = "lokalne konto"
            Ensure = "Present"
            Disabled = $false
            PasswordNeverExpires = $true
            PasswordChangeRequired = $false
        }
    }
}

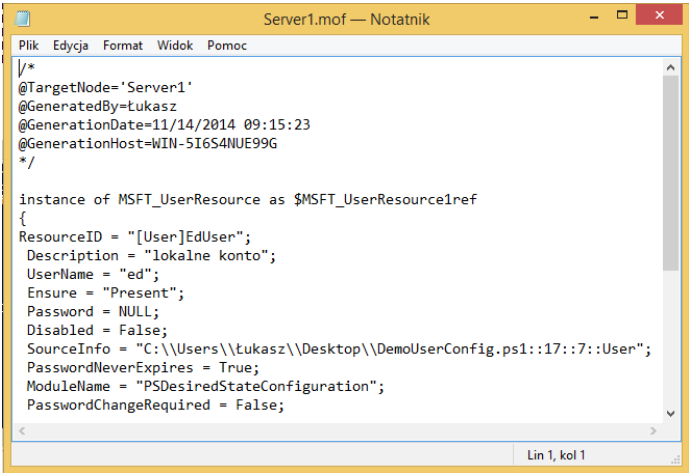
DemoUser

```



RYSUNEK 23.3. Jeśli konfiguracja nie pozwala na zapisywanie haseł w postaci zwykłego tekstu, zostaje wygenerowany błąd

Problemem nie jest sposób przekazywania hasła do konfiguracji, tylko to, co dzieje się po jej uruchomieniu — deszyfracja hasła i zapisanie go w postaci tekstowej w pliku MOF. Spójrz na rysunek 23.4.



RYSUNEK 23.4. Dzięki zgodzie konfiguracji hasło zostaje zapisane w postaci tekstowej w pliku MOF

Jako że hasło jest zapisywane w postaci tekstowej w pliku MOF, programiści Windows PowerShell woleli się upewnić, że użytkownik wie, co robi. (Przy okazji zamiast zapisywać hasło w formie tekstowej, można je zaszyfrować przy użyciu certyfikatu).

Po utworzeniu danych konfiguracji wywołujemy tę konfigurację i podajemy nowo utworzone dane, jak pokazano poniżej:

```
$configData = @{
    AllNodes = @(
        @{
            NodeName = "Server1";
            PSDscAllowPlainTextPassword = $true
        }
    )
}
```

```
ScriptFolder -ConfigurationData $configData
```

Tworzenie użytkowników przy użyciu dostawcy User

Aby utworzyć użytkownika lokalnego, należy wywołać dostawcę User i podać nazwę tego użytkownika. Hasło do własności Password jest przekazywane jako obiekt `PSCredential`. Nie jest to to samo co zwykły obiekt `SecureString`, którego można się było spodziewać w tej sytuacji. Jest tak, ponieważ obiekt `PSCredential` zawiera zarówno nazwę użytkownika, jak i hasło (jako obiekt `SecureString`). Następną jest własność `Description` z opisem oraz własność `Disabled` określająca, czy konto ma być włączone, czy wyłączone. Aby utworzyć wyłączone konto, własności `Disabled` należałoby przypisać wartość `$True`. Dwie ostatnie konfigurowane własności to `Password` ➔ `NeverExpires` i `PasswordChangeRequired`. Poniżej znajduje się fragment kodu, w którym zastosowano opisaną technikę:

```
User EdUser
{
    UserName = "ed"
    Password = $cred
    Description = "lokalne konto"
    Ensure = "Present"
    Disabled = $false
    PasswordNeverExpires = $true
    PasswordChangeRequired = $false
}
```

Tworzenie grup przy użyciu dostawcy Group

Aby utworzyć lokalną grupę przy użyciu dostawcy Group, należy przekazać nazwę tej grupy do własności `GroupName` oraz zdefiniować jej opis. Członków grupy dodaje się za pomocą tablicy nazw użytkowników. Ponieważ dodawani do grupy użytkownicy muszą istnieć, do wyrażenia tej zależności należy użyć własności `DependsOn`. Pokazano to w poniższym przykładzie:

```
Group Scripters
{
    GroupName = "Skrypciarze"
    Credential = $cred
    Description = "Skrypciarze"
    Members = @("ed")
    DependsOn = "[user]Eduser"
}
```

Poniżej znajduje się kompletny kod źródłowy skryptu *ScriptFolderVersionUnzipCreateUsersAndProfile.ps1*:

ScriptFolderVersionUnzipCreateUsersAndProfile.ps1

```
Configuration ScriptFolder
{
    Param ($modulePath = ($env:PSModulePath -split ';' |
        ? {$_ -match 'Program Files'}))
    node Server1
    {
        User EdUser
        {
            UserName = "ed"
            Password = $cred
            Description = "lokalne konto"
            Ensure = "Present"
            Disabled = $false
            PasswordNeverExpires = $true
            PasswordChangeRequired = $false
        }
        Group Scripters
        {
            GroupName = "Scripters"
            Credential = $cred
            Description = "Skrypciarze"
            Members = @("ed")
            DependsOn = "[user]Eduser"
        }
        File ScriptFiles
        {
            SourcePath = "\\dc1\Share\"
            DestinationPath = "C:\scripts"
            Ensure = "present"
            Type = "Directory"
            Recurse = $true
        }
        Registry AddScriptVersion
        {
            Key = "HKEY_Local_Machine\Software\ForScripting"
            ValueName = "ScriptsVersion"
            ValueData = "1.0"
            Ensure = "Present"
        }
        Archive ZippedModule
        {
            DependsOn = "[File]ScriptFiles"
            Path = "C:\scripts\PoshModules\PoshModules.zip"
            Destination = $modulePath
            Ensure = "Present"
        }
        File PoshProfile
        {
            DependsOn = "[File]ScriptFiles"
            SourcePath = "C:\scripts\PoshProfiles\Microsoft.PowerShell_profile.ps1"
            DestinationPath = "$env:USERPROFILE\WindowsPowerShell\
                Microsoft.PowerShell_profile.ps1"
```

```

        Ensure = "Present"
        Type = "File"
        Recurse = $true
    }

}

$cred = get-credential
$configData = @{
    AllNodes = @(
        @{
            NodeName = "Server1";
            PSDscAllowPlainTextPassword = $true
        }
    )
}

ScriptFolder -ConfigurationData $configData
Start-DscConfiguration Scriptfolder

```

Kontrolowanie dryfu konfiguracji

Ponieważ konfiguracje DSC są idempotentne, można je wykonywać wielokrotnie bez obawy, że utworzy się wiele zasobów lub spowoduje błędy. Jeśli więc wywoła się parę razy tę samą konfigurację, a w czasie między wywołaniami nic się nie zmieni, to wywołania te nie wprowadzą żadnych modyfikacji. Jeśli natomiast między jednym a drugim wywołaniem coś się zmieni w konfiguracji serwera, to łatwo będzie można przywrócić żądany stan serwera. Nie trzeba nawet martwić się dryfem konfiguracji — wystarczy ponownie wykonać konfigurację, aby mieć pewność, że serwer zostanie przywrócony do odpowiedniego stanu. W przypadkach gdy stan serwera musi być zgodny ze stanem DSM, można regularnie uruchamiać polecenie `Start-DscConfiguration` za pomocą kreatora harmonogramu zadań.

Innym sposobem na sprawdzanie, czy doszło do dryfu konfiguracji, jest wywołanie funkcji `Test-DscConfiguration`. W tym celu należy utworzyć sesję CIM do zdalnych serwerów, których konfiguracje mają zostać sprawdzone. Najlepiej to zrobić na serwerze, przy użyciu którego utworzono DSC, ponieważ zapewniony będzie dostęp do pliku MOF. Po utworzeniu sesji CIM przekaz ją do funkcji `Test-DscConfiguration`. Technikę tę ilustruje poniższy przykład:

```

PS C:\> $session = New-CimSession -ComputerName server1, server2 -Credential
iammred\administrator

PS C:\> Test-DscConfiguration -CimSession $session
True
True

```

Poniżej znajduje się kod źródłowy skryptu *SetServicesConfig.ps1*, który tworzy dwa pliki konfiguracyjne MOF — po jednym dla każdego serwera wymienionego w tablicy `node`.

SetServicesConfig.ps1

```

Configuration SetServices
{
    node @('Server1', 'Server2')
    {
        Service Bits
        {
            Name = "Bits"
            StartUpType = "Automatic"
            State = "Running"
            BuiltinAccount = 'LocalSystem'
        }
        Service Browser
        {
            Name = "Browser"
            StartUpType = "Disabled"
            State = "Stopped"
            BuiltinAccount = 'LocalSystem'
        }
        Service DHCP
        {
            Name = "DHCP"
            StartUpType = "Automatic"
            State = "Running"
            BuiltinAccount = 'LocalService'
        }
    }
}

SetServices -OutputPath C:\ServerConfig
Start-DscConfiguration -Path C:\ServerConfig

```

Na rysunku 23.5 widać wynik wykonania tej konfiguracji i efekt jej sprawdzenia za pomocą sesji CIM.

The screenshot shows the Windows PowerShell ISE window titled "Administrator: Windows PowerShell ISE". The script editor displays a file named "SetServicesConfig.ps1" with the following content:

```

1 Configuration SetServices
2 {
3     node @('Server1', 'Server2')
4     {
5         ...
6     }
7 }
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30 SetServices -OutputPath C:\ServerConfig
31 Start-DscConfiguration -Path C:\ServerConfig
32

```

The console window shows the execution output:

```

Name           : Job74
ChildJobs       : {Job75, Job76}
PSBeginTime    : 9/8/2013 10:49:24 PM
PSEndTime      :
PSJobTypeName   : ConfigurationJob
Output         : {}
Error          : {}
Progress       : {}
Verbose        : {}
Debug          : {}
Warning        : {}
State          : Running

PS C:\> $session = New-CimSession -ComputerName server1, server2 -Credential iammred\administr
PS C:\> Test-DscConfiguration -CimSession $session
True
True
PS C:\> |

```

The status bar at the bottom indicates "Completed", "Ln 44 Col 9", and "100%".

RYSUNEK 23.5. Użycie CIM do sprawdzenia konfiguracji węzła skonfigurowanego za pomocą DSC

Dodatkowe źródła informacji

- Centrum skryptów portalu TechNet pod adresem <http://www.microsoft.com/technet/scriptcenter> zawiera wiele przykładów skryptów.
- Wszystkie skrypty opisane w tym rozdziale są dostępne do pobrania w repozytorium centrum skryptów portalu TechNet pod adresem <http://gallery.technet.microsoft.com/scriptcenter/PowerShell-40-Best-d9e16039>.

O autorze



ED WILSON jest specjalistą i znanym ekspertem od skryptów w firmie Microsoft. Prowadzi blog pod tytułem *Hey, Scripting Guy!*, na którym codziennie publikuje nowe wpisy. Ponadto wygłasza prelekcje na konferencjach TechEd i wewnętrznych konferencjach Microsoftu TechReady. Jest certyfikowanym trenerem Microsoftu i autorem kursu Windows PowerShell dla najważniejszych klientów firmy Microsoft z całego świata. Napisał 10 książek wydanych przez wydawnictwo Microsoft, z których siedem dotyczy pisania skryptów dla systemu Windows. Ponadto brał udział w procesie wydawniczym

dwunastu innych książek. Jego najnowsze książki wydane przez wydawnictwo Microsoft to *Windows PowerShell 3.0 Step by Step* i *Windows PowerShell 3.0 First Steps*. Wilson jest posiadaczem ponad 20 certyfikatów branżowych, m.in. Microsoft Certified Systems Engineer (MCSE) i Certified Information Systems Security Professional (CISSP). Zanim przeszedł do pracy w firmie Microsoft, był starszym konsultantem w firmie będącej partnerem Microsoft Gold Certified firmy Microsoft i specjalistą od Active Directory oraz Exchange. W wolnym czasie lubi zajmować się stolarką, nurkować oraz fotografować podwodny świat.

Skorowidz

.NET, 92, 111
.NET Framework, 88

A

abstrakcyjne drzewa składni, 451
Active Directory, 444, 445, 481
ADO, ActiveX Data Object, 88, 208
adres IP, 256
ADSI, Active Directory Services Interface, 88
agent aktualizacji, 221
akcelerator
 [ADSISeacher], 209
 [WMICLASS], 275, 403
aktualizowanie
 dokumentacji, 287
 pomocy, 35–37
aktywności
 nieautomatyczne, 604
 przepływów pracy, 602, 603
 równoległe, 605
 sekwencyjne, 609
aktywność
 Checkpoint-Workflow, 603, 609
 ForEach -Parallel, 603
 InlineScript, 604, 611
 Parallel, 603
 Sequence, 603
 Suspend-Workflow, 603
algorytm MD5, 27
alias, 122
 cat, 366
 ft, 51
aliasy
 parametru, 341
 stałe, 125

tylko do odczytu, 124
nazw funkcji, 127
typów danych, 249
AllUsersAllHosts, 148
AllUsersCurrentHost, 148
analiza
 defragmentacji, 219
 składniowa skryptów, 451
 składniowa skryptu, 455
API, pplication programming interface, 88
API C Windows, 433
aplikacja, 117, *Patrz także* narzędzie
aplikacje IIS, 96
argumenty pozycyjne, 236
AST, abstract syntax tree, 451
atrybut
 [ordered], 529
 mandatory, 337
 ValidateRange, 409
atrybuty
 parametrów, 339, 340
 weryfikacyjne, 346, 451
automatyzacja, 88, 90
 testów interfejsów, 434
 zadań, 87
AWDS, Active Directory Web Service, 62

B

baza danych skryptów, 227
Bellée Chris, 226, 454
bezpieczeństwo, 29, 72, 96
bezpieczeństwo danych, 492
blok
 Catch, 392, 407
 Finally, 392
 Try, 392

błąd, 135, 147, 160, 170, 199, 205, 281, 330, 336,
338, 391, 403, 407, 520, 582, 585, 620
błędy
 logiczne, 441, 527
 składniowe, 443, 524
 wykonawcze, 524
 zakresu, 408
brak
 dostawców WMI, 396
 uprawnień, 389
 znaku dolara, 342
Brasser Jaap, 341
Brundage James, 151, 391, 417, 430, 432

C

Canastreiro Luís, 112, 164
Carlos Ruiz Lopez Juan, 234, 251
Carter Marc, 317
Cedeno Enrique, 440, 444
certyfikat, 472
chmura, 144
Christopher Jim, 307
CLR, Common Language Runtime, 112
COM, Component Object Model, 88, 136, 185
Costantini Peter, 299
Craig Burley James, 261
CurrentUserAllHosts, 148
CurrentUserCurrentHost, 148
czyszczenie zawartości pliku, 444

D

dane
 dziennika, 516
 konfiguracji, 619
 wejściowe parametru, 342
 WMI, 247
 XML, 431
DCOM, Distributed Component Object
 Model, 95
debuger aliasu, 552
debugowanie, 483, 523, 554
debugowanie skryptów, 515, 545, 558, 560
definiowanie
 funkcji, 240, 244
 komentarzy, 281
 logiki biznesowej, 254
defragmentacja, 219

Dekens Luc, 529
diagnostyka, 546
diagnostyka szczegółowa, 527
dodawanie
 aktywności sekwencyjnej, 609
 dokumentacji, 271
 komentarza, 275, 288
 punktów kontrolnych, 607
dokumentacja, 222, 271, 287
 Active Directory, 70, 73
 SDK, 185
dokumentowanie
 przebiegu testowania, 419
 skryptów, 447
dołączanie
 dwukropka, 409
 pliku, 154
dopisywanie danych, 504
dostawca
 CIMWin32, 396
 Group, 621
 MSIProv, 398
 User, 621
 WMI, 172, 396
 zasobów DSC, 614
dostęp do
 folderu, 103, 137
 konfiguracji sesji, 481
 obiektu, 338
 obiektu COM, 136
 punktów wstrzymania, 549
dryf konfiguracji, 623
DSC, Desired State Configuration, 17, 613
dysk
 HKCR, 400, 401
 PowerShell, 141, 142
 sieciowy, 513
dziennik, 431, 441, 443, 502, 504
 aplikacji, 519
 śledzenia, 506
 Windowsupdate.log, 516
 zdarzeń, 478, 518, 519

E

edytor skryptów, 490
edytowanie, 484
EFS, Encrypting File System, 349
egzemplarz klasy, 454

eksportowanie
 danych, 445
 historii poleceń, 204
 poświadczeń, 354
 ETS, Extended Type System, 143
 ewaluacja zmiennej, 405

F

Farr Ian, 492, 508
 filtr, 263, 266
 filtr HasMessage, 266
 filtrowanie klas, 48
 filtry wyszukiwania LDAP, 209–213
 Finke Douglas, 240
 folder
 %username%, 135
 Backups, 34
 Dokumenty, 29
 foldery
 skryptów, 482
 tymczasowe, 277
 wyjściowe, 511
 fragmenty kodu, 569, 570
 funkcja, 231, 242, 427
 AddOne, 264, 265
 Add-Registry, 531
 Add-RegistryValue, 533
 add-two, 543
 bletch, 262
 Check-AllowedValue, 408
 Check-Number, 343
 ConvertFrom-Cab, 188
 CreateFilePath, 225
 CreateSelection, 224
 DivideNum, 556
 Enable-PSRemoting, 583–585
 Expand-Cab, 221
 Format-NonIPOutput, 260
 Get-AllowedComputer, 386–388
 Get-Bios, 442
 Get-Change, 427, 428
 Get-Choice, 383
 Get-Comments, 279
 Get-ComputerInfo, 321, 322
 Get-CountryByIP, 510
 Get-Discount, 254
 Get-FileName, 453
 Get-Folder, 511
 Get-ieStartPage, 275, 276

Get-MemberOf, 103
 Get-MoreHelp, 126, 128
 Get-MyModule, 315–317
 Get-OsVersion, 114
 Get-TempFile, 428
 Get-TextStatistics, 235–239
 Get-Type, 152
 Get-ValidWmiClass, 403–405
 Get-Version, 165
 Get-Volume, 163, 164
 Get-WmiClass, 131–133
 Get-WmiClasses, 283, 284
 Get-WmiInformation, 405
 Get-WmiProvider, 398, 402
 IntelliSense, 567
 Kontrola konta użytkownika, 96
 New-Cab, 186, 187
 New-DDF, 192
 New-LocalUser, 435
 New-ModulesDrive, 313
 New-TempFile, 514
 New-TestConnection, 344
 ParseAction, 154
 Remove-OutPutFile, 278
 Set-LocalGroup, 437
 Set-ScreenSaverTimeout, 178
 Start-Transcript, 441
 Test-ComputerPath, 384
 Test-IsAdmin, 108
 Test-IsAdministrator, 518
 Test-ModulePath, 309
 Test-Scripts, 427
 Write-Path, 238

funkcje
 definiowanie logiki biznesowej, 254
 przenośność, 244
 rozwijania nazw, 525
 samodzielność, 244
 spójność wyników, 243
 z innych skryptów, 152
 z wieloma parametrami, 244, 252

G

gałąź
 CLSID, 401
 Current_User, 522
 HKEY_Classes_Root, 400
 Hkey_Current_User, 520

- generowanie
 - dokumentacji, 450
 - pliku konfiguracyjnego, 481
 - raportów, 503
- gip, 24
- gotowe fragmenty kodu, 569
- Goude Niklas, 480
- GPO, 469
- grupa, 103, 104
- grupa zabezpieczeń
 - dodawanie użytkownika, 76
 - nazwa, 75
 - ścieżka, 75
 - usuwanie użytkownika, 76
 - zakres, 75
- Gusev Vasily, 558

H

- hasło, 77, 348, 620, 621
- Helmick Jason, 24
- Hicks Jeffery, 450
- hierarchiczna przestrzeń nazw, 167
- Hill Keith, 304
- historia poleceń, 204
- Hofferle Jason, 87, 248
- Holmes Lee, 354
- Huffman Clint, 173

I

- identyfikacja wersji, 89
- identyfikator
 - GUID, 203
 - klasy dostawcy, 401
 - OID, 351
 - ObjectGUID, 81
 - RID, 71
 - SID, 54, 81, 99, 103, 444
- IIS, Internet Information Services, 96
- importowanie
 - pliku, 445
 - poświadczeń, 354
- indeksowanie zmiennej \$args, 331
- informacje
 - diagnostyczne, 436, 526
 - dotyczące domeny, 71
 - o błędach, 37, 125, 133, 147, 161, 517, 593
 - o czasie logowania, 502
 - o dostawcy klasy, 396
 - o karcie sieciowej, 595
 - o komputerach, 602
 - o odmowie dostępu, 97
 - o postępie testów, 428
 - o typie obiektu, 403
 - o Usłudze konfiguracji, 613
 - o wersji platformy .NET, 112
 - o wykonywaniu funkcji, 438
 - w nagłówku, 290
 - zbędne, 293

- infrastruktura WMI, 167
- instalowanie
 - konsoli, 25
 - modułów, 308
 - modułu Active Directory, 62
- instrukcja
 - #Requires, 97, 314, 393
 - [cmdletbinding()], 434
 - exit, 438
 - Foreach, 174, 187
 - if, 388, 397
 - On Error Resume Next, 406
 - Param, 335, 381, 409, 416
 - Return, 261, 438
 - Switch, 357
 - Throw, 333
 - Trap, 391, 407

- instrumentacja, 515
- instrumentacja zarządzania Windows, 167
- integralność, 492
- interfejs
 - API, 433
 - automatyzacji, 88
 - programistyczny, 88
 - usług Active Directory, 88
- ISE, Integrated Scripting Environment, 317

J

- jednostka organizacyjna, 74, 211
- język VBScript, 40, 91, 393, 406
- Jones Don, 43, 483, 489, 545

K

- katalog LDAP, 167
- Kearney Sean, 40
- klamra, 126

klasa

- __provider, 397, 398
- _Namespace, 168
- _provider, 172
- Enum, 110
- ErrorRecord, 251
- InvocationInfo, 180
- Io.Path, 513
- Microsoft.Win32.Registry, 92
- PromptForChoice, 358
- PSObject, 511
- PSParser, 456
- Security.Principal.WindowsBuiltInRole, 109
- Security.Principal.WindowsIdentity, 518
- SecurityIdentifier, 100
- System.Enum, 109, 179
- System.Environment, 113, 115
- System.IO.FileInfo, 311
- System.IO.Path, 412
- System.Management.Automation.LineBreak, 547
- System.Math, 182
- System.Random, 160
- System.String, 403
- System.TimeSpan, 424, 428
- System.Version, 113
- Win32_Bios, 50, 565
- Win32_ComputerSystem, 412
- Win32_Desktop, 176
- Win32_LogicalDisk, 245
- Win32_OperatorSystem, 165
- Win32_Process, 52
- Win32_Product, 398
- Win32_Service, 51
- Win32_UserAccount, 54
- Win32_Volume, 218
- WindowsIdentity, 98
- WMI Win32_NetworkAdapterConfiguration, 256
- Word.Application, 224

klasy

- abstrakcyjne, 49
- Association, 52
- CIM, 45
- dynamiczne, 173
- pospolite, 173
- rdzenne, 173
- WMI, 45, 47, 49, 173

klauzula WHERE, 118

klawisz Tab, 567

Klindt Todd, 119

klucz \$scriptRoot, 520

klucze rejestru, 222

- ForScripting, 520, 617

kod ADSI, 65

kolejność wykonywania aktywności, 610

kolekcja, 174

komentarze, 275, 452

- efektywne, 286

- jednowierszowe, 271, 280, 294

- na końcu wiersza, 295

- opisywanie struktur, 295

- wielowierszowe, 279

komentarzowy blok nagłówkowy, 449

komunikacja, 299

konfiguracje DSC, 623

konfigurowanie, 616, 619

- konsoli, 33

- profilu, 121

- przycisków, 139

- środowiska skryptowego, 121

- węzła, 625

- żądanego stanu programu, 613

konkatenacja, 278, 405

konsola

- MMC, 207

- PWA, 43

- Windows PowerShell, 21, 23

- Windows PowerShell ISE, 94

konstrukcja

- Begin-Process-End, 338

- Try-Catch, 407

- Try-Catch-Finally, 334, 391

konstruktor, 159, 182

konsumenci WMI, 167

kontener users, 77

konto

- komputera, 74

- użytkownika, 78

kontrola

- dryfu konfiguracji, 623

- konta użytkownika, 96, 389

- wersji, 490, 496

- wersji skryptów, 489

- wykonywania poleceń, 30

konwencje nazewnictwa, 151

konwersja łańcucha, 403, 405

krokowe wykonywanie skryptu, 541

kwalifikator

- abstract, 49
- association, 48
- deprecated, 48
- dynamic, 49
- singleton, 48
- supportsupdate, 48

L

LDAP, 167, 208

liczba

- błędów, 413
- opcji, 382
- poleceń, 24, 158
- własności i egzemplarzy, 50

liczenie testów, 428

lista

- czasowników, 235
- dostawców WMI, 172
- dostępnych modułów, 302
- gotowych fragmentów kodu, 569
- jednostek organizacyjnych, 215
- przestrzeni nazw, 170
- punktów wstrzymania, 556
- skryptów, 491

logika

- biznesowa, 254, 255
- programu, 254

logowanie, 467, 475–480

lokalizacje sieciowe, 513

Lopez Juan Carlos Ruiz, 554

Ł

ładowanie modułów, 305

łańcuch

- miejscowy, 192
- połączenia, 356

łączenie zmiennych, 405

M

magazyn

- certyfikatów użytkownika, 473
- skryptów, 492

Maheu Georges, 166

mapowanie dysków, 201

maszyna wirtualna, 445

Mayer Keith, 90

McGlone Ashley, 61

mechanizm potwierdzania, 76

Mell Bill, 32

Menedżer certyfikatów, 472, 473

menu

- Debug, 546, 560
- opcji, 582

metoda

- AddDays, 422
- ChangeStartMode(), 175, 176
- Connect(), 455
- create, 47
- createcab, 186
- DerfagAnalysis, 218
- DownloadString, 431
- FindAll, 209
- GetCurrent, 97, 518
- GetNames, 109, 179
- GetStringValue, 92
- GetTempFileName, 412, 513
- GetType, 403
- GetValues, 110
- IsInRole, 108
- Namespace, 190
- popup, 138–141
- PromptForChoice, 382, 383
- RegRead, 91
- saveas, 225
- split, 310
- Tokenize, 456
- ToString, 99, 353, 520
- WshShell.popup, 139

metody

- klas WMI, 47
- pisania skryptów, 164
- pobierania danych, 328
- statyczne, 182
- zwracania danych, 358

Minasi Mark, 364

model COM, 88

moduł, 118, 301, 395

- Active Directory, 61
- dokumentacja, 70, 73
- instalowanie, 62
- nazwa lokalizacji, 73
- zastosowanie, 63
- znajdowanie kont użytkowników, 78, 81
- znajdowanie wyłączonych użytkowników, 80
- BasicFunctions.psm1, 319

- CimCmdlets, 46
- DotNet, 152
- PSCX, 144
- moduły
 - instalowanie, 308
 - ładowanie, 305
 - sprawdzanie zależności, 314
 - tworzenie, 319
 - tworzenie folderu, 309
 - z udziałów, 318
- modyfikowanie
 - rejestr, 465, 466
 - skryptów, 256
 - wartości, 178
 - zmiennej path, 286
- monitowanie o informacje, 357
- Moravec David, 27
- możliwości konsoli, 23
- Muscetta Daniele, 470

N

- nadpisywanie dziennika, 500
- nagłówki, 290
 - komentarzowy, 449
 - skryptu, 493
- narzędzia
 - do pracy zdalnej, 94–96, 573
 - RSAT, 63
 - wiersza poleceń, 28
- narzędzie
 - CMD.exe, 142, 376
 - CSVDE, 444
 - DSMode.exe, 217
 - DSQuery.exe, 215, 216
 - fsutil, 24
 - IPConfig.exe, 24
 - Kinect, 390
 - LDIFDE, 444
 - MakeCab.exe, 192
 - NetDom, 88
 - NetSH, 88, 191
 - PoshPAIG, 324
 - SolarWinds Network Configuration Manager, 33
 - Streams.exe, 464
 - System Center Operations Manager, 221
 - VersionRecall, 497
 - VSS, 496

- WbemTest, 396
- WSH, 496
- nawias
 - okrągły, 248
 - trójkątny, 219
- nazwa
 - dostawcy WMI, 397, 399
 - główna użytkownika, 212
 - jednostki organizacyjnej, 211
 - lokalizacji, 73
 - modułu, 395
- niedozwolone polecenia rdzenne, 604
- niepoprawna wersja skryptu, 495
- niepoprawne typy danych, 403
- nieprzechwycone wyniki, 263
- Norman Richard, 375
- notacja
 - funkcyjna, 543
 - metodowa, 543
- numer
 - poprawki, 113
 - wersji, 493, 494
 - głównej, 113
 - kompilacji, 113
 - pomocniczej, 113

O

- obiekt, 454
 - \$wshShell, 137
 - COM, 143
 - COM WshShell, 93
 - DateTime, 413, 422
 - DirectoryEntry, 216
 - GPO, 470, 475
 - makecab.makecab, 188
 - MSGraph.Application, 505
 - poświadczeń, 355
 - PSCredential, 354, 355, 621
 - ScriptInfo, 180, 181
 - selection, 224
 - shell, 188
 - Shell.Application, 188, 190
 - Win32_OperatingSystem, 165
 - WindowsIdentity, 98, 105
 - Word.Document, 225
 - WshShell, 136–141
 - WshSpecialFolders, 137
 - XML, 431

obsługa

- aplikacji zewnętrznych, 191
- błędów, 379, 404
- brakujących parametrów, 380
- COM, 185
- hasel, 445
- IP, 259
- parametrów, 129
- parametrów nazwanych, 131
- platformy .NET, 182
- poleceń, 157
- skryptów, 146
- wejścia i wyjścia, 327
- WMI, 167

odblokowywanie kont użytkowników, 78

odczytywanie

- pliku tekstowego, 198
- rejstru, 91, 92

odmowa dostępu, 97, 525

ograniczanie

- liczby opcji, 382
- możliwości wyboru, 382
- wartości parametru, 409

ograniczenia typów parametrów funkcji, 249, 131

ogranicznik [Object[]], 339

ograniczona zasada wykonywania, 147

okienko

- Command, 565, 567
- skryptu, 566

okno

- poświadczeń, 583
- Testera, 396
- Windows PowerShell ISE, 563, 564

opcja

- Step Into, 560
- Step Over, 560

opcje pomocy, 35

operator

- contains, 101, 385, 387
- like, 101
- match, 101, 130
- [], 337
- przekierowania, 499, 501
- zakresu, 123

operatory filtrów, 211

opisywanie struktur, 295

P

pakiet

- MSI, 466, 482
- VSS, 496
- Windows Management Framework 4.0, 25

parametr, 243

- \$baseLineScript, 427
- action, 551
- append, 413, 515
- autosize, 208
- baseLineScript, 426
- class, 366
- classname, 45, 50
- commandtype, 215
- computer, 335
- computername, 117, 165, 206, 330, 573
- confirm, 76
- credential, 355, 581
- debug, 186, 434, 526
- description, 136, 569
- destination, 191
- discover, 67
- Encoding, 507
- Expression, 427
- filepath, 367
- filter, 50, 55, 68, 178, 213
- folder, 510
- force, 81, 125, 223, 583
- groupScope, 75
- identity, 80
- inputobject, 55
- keep, 595
- LDAPFilter, 213
- log, 427
- members, 76
- membertype, 591
- mode, 561
- modifiedScript, 426
- namespace, 45
- nazwany, 131
- numberOfTests, 427
- parent, 277
- password, 357
- path, 199
- PipelineVariable, 341
- Process, 413
- property, 50
- PSComputerName, 588, 600, 601
- qualifier, 48

- recurse, 412
- reset, 77
- script, 558
- step, 539
- strict, 542
- Text, 569
- Title, 569
- trace, 534
- TypeName, 454
- ValidatePattern, 344
- variable, 368
- Verbose, 179, 436
- whatif, 434, 437–440
- width, 507
- Wrap, 208
- parametry
 - instrukcji #Requires, 394
 - konfiguracji, 617
 - obowiązkowe, 337, 381
 - polecen, 34
 - standardowe, 434
 - wiersza poleceń, 426
- parser AST, 455
- pasek stanu, 532
- pętla, 128
- pętla foreach, 171, 339
- Pfeiffer Mike, 502
- pisanie
 - funkcji, 242
 - skryptów, 157
- planowanie skryptów, 59
- platforma .NET, 92, 111
- plik
 - \$files, 422
 - Autoexec.bat, 148
 - dotnettypes.format.ps1xml, 362
 - konfiguracji, 481
 - package.xml, 221, 222
 - passwordHash.txt, 354
 - TroubleShoot.bat, 29
 - WindowsUpdate.log, 134
 - wsusscn2.cab, 222
- pliki
 - .bat, 465
 - .cab, 186, 187, 222
 - .ddf, 192
 - .ps1, 143
 - .psm1, 319
 - .xml, 204
 - ADM, 471
 - cabinet, 185
 - CSV, 502
 - MOF, 614, 616
 - pomocy, 36, 37
 - tekstowe, 198, 386, 509, 516
 - wsadowe, 29
- pobieranie
 - danych, 328
 - danych WMI, 247
 - dokumentacji
 - z komentarzy, 452
 - z pomocy, 447
 - hasel, 348
 - informacji, 93, 95
 - łańcuchów połączenia, 356
- podpisywanie kodu, 472
- polecenia, 24
 - Active Directory, 213
 - CIM, 45
 - cmdlet, 27, 30, 31, 573
 - diagnostyczne, 392, 546, 555
 - New-Aduser, 77
 - promieniste, 205
 - rdzenne, 604
- polecenie
 - Add-Content, 509
 - Add-Member, 511
 - cd, 23
 - Clear-Host, 541
 - CLS, 376
 - Complete-Transaction, 179
 - Continue, 558
 - ConvertFrom-SecureString, 355
 - ConvertTo-Html, 370, 502, 509
 - Copy-Item, 200, 205
 - dir, 23
 - Disable-PSBreakpoint, 561
 - Enable-PSRemoting, 583
 - Enter-PSsession, 70, 587
 - Export-Clixml, 359, 509
 - Export-Csv, 370, 502, 509
 - ForEach-Object, 193, 204, 332, 423
 - Format-List, 81
 - Format-Table, 51, 207, 557, 566
 - Get-ADDefaultDomainPasswordPolicy, 72
 - Get-ADDomain, 71
 - Get-ADDomainController, 67, 72
 - Get-ADForest, 71
 - Get-ADOrganizationalUnit, 216
 - Get-ADRootDSE, 73

polecenie

Get-Aduser, 77
 Get-ADUser, 77–83
 Get-Alias, 122, 133
 Get-ChildItem, 187, 193, 342, 412, 422, 456
 Get-CimAssociatedInstance, 52–56
 Get-CimClass, 45–48
 Get-CimInstance, 46–51, 600
 Get-Command, 66, 602
 Get-Content, 198, 200, 219, 366, 525
 Get-Credential, 481
 Get-Date, 267, 422, 442
 Get-EventLog, 520, 605
 Get-ExecutionPolicy, 147, 468, 569
 Get-FileHash, 27
 Get-Help, 38, 125, 286, 323, 484, 573
 Get-History, 204
 Get-IseSnippet, 571
 Get-Item, 374, 407
 Get-Job, 591, 594
 Get-Member, 52, 177, 454, 507, 591
 Get-Module, 63, 65, 321
 Get-NetAdapter, 595
 Get-NetIPConfiguration, 24
 Get-Process, 118, 125, 157, 360, 561, 591
 Get-PSBreakpoint, 556, 560
 Get-PSDrive, 401
 Get-PSSession, 213, 587
 Get-Service, 117, 157, 206, 218
 Get-Variable, 329
 Get-WebServiceProxy, 510
 Get-WindowsFeature, 611
 Get-WmiClass, 130
 Get-WmiObject, 172, 219, 330, 405, 442, 595
 help, 42
 Import-Clixml, 359
 Import-Module, 213, 223
 Import-Module ActiveDirectory, 62
 Import-PSSession, 107
 Invoke-Command, 65, 94, 205, 477, 588
 Invoke-Expression, 198, 413
 Invoke-History, 204
 ipconfig, 28, 365
 Join-Path, 134, 192, 225, 401
 Measure-Command, 424, 426, 430
 Move-Item, 514
 New-ADGroup, 75
 New-ADOrganizationalUnit, 74
 New-Alias, 123
 New-DDF, 194
 New-EventLog, 517
 New-IseSnippet, 570
 New-Item, 123, 134, 150, 533
 New-ItemProperty, 521
 New-Object, 185, 430, 454
 New-PSDrive, 313
 New-PSSession, 586, 587
 New-Variable, 135, 342
 Out-File, 219, 366, 413, 506, 513
 Out-Host, 364
 Out-Null, 401, 521
 Out-String, 520
 ping, 197, 384
 Quit, 558
 Read-Host, 352, 412, 434
 Receive-Job, 593
 Register-PSSessionConfiguration, 481
 Remove-Item, 125, 571
 Remove-Job, 591
 Remove-PSBreakpoint, 560
 Remove-PSDrive, 402
 Remove-PSSession, 213, 587
 Rename-ADObject, 74
 Save-Help, 37
 Search-ADAccount, 78
 Select-Object, 209, 361, 529
 Select-String, 240
 Set-ADAccountPassword, 77
 Set-Alias, 123, 124
 Set-Content, 509
 Set-ExecutionPolicy, 146, 466, 471
 Set-Item, 134
 Set-ItemProperty, 521
 Set-Location, 142, 586
 Set-PSBreakpoint, 547, 558
 Set-PSDebug, 529–534, 539–542, 548
 Set-StrictMode, 543
 Split-Path, 187
 Start-DscConfiguration, 623
 Start-Job, 594
 Start-Transaction, 178
 Start-Transcript, 441, 515, 587
 Step-into, 558
 Stop-Job, 594
 Stop-Process, 157
 Stop-Transcript, 442, 515
 substring, 225
 Tee-Object, 367, 509, 515
 Test-Connection, 526
 Test-Path, 154, 342, 400, 533

- Test-WSMan, 584
- Update-Help, 36
- Wait-Job, 593
- Where-Object, 117, 124, 425, 509, 580
- Write-Debug, 179, 180, 192, 441, 525
- Write-Error, 509
- Write-Host, 434, 551
- Write-Verbose, 178, 400, 438, 545
- Write-Warning, 509
- pomoc, 35, 38, 41, 447, 484
 - do skryptu, 271
 - komentarzowa, 281
 - techniczna, 484
- porównanie szybkości działania, 424
- potok, 423
- potwierdzenie wykonania poleceń, 31
- powtarzalność, 218
- poziom
 - drugi śledzenia, 533
 - pierwszy śledzenia, 532
 - zasady wykonywania, 146
- praca zdalna, 94, 95, 573
- proces DSC, 614
- profil, 144, 148–152
 - AllUsersAllHosts, 150
 - CurrentUserAllHosts, 150
 - CurrentUserCurrentHost, 149, 150
- program, *Patrz* narzędzie
- programowanie, 234
- programy do kontroli wersji, 496
- projektowanie
 - modułów, 301
 - skryptów, 159, 229, 231
- protokół XMPP, 143
- Prox Boe, 324
- przechowywanie
 - dzienników, 513
 - informacji, 522
 - skryptów, 492
 - tekstu, 509
- przechwytywanie błędów, 391
- przedrostek ! CALL, 534
- przeglądanie
 - danych, 421
 - tablicy, 128
- przeglądarka Internet Explorer, 465
- przekazywanie
 - opcji do poleceń, 34
 - wartości, 330
 - wielu parametrów, 129
- przekierowanie, 219
- przełącznik
 - bypass, 462, 468
 - debug, 526, 528
 - forceDiscover, 67
 - UseTransaction, 178
 - whatif, 438
 - wrap, 566
- przepływ pracy, workflow, 94, 483, 597
- przesłanie istniejących poleceń, 126
- przestrzenie nazw WMI, 45, 168
- przestrzeń nazw, 167, 374
- przetwarzanie
 - łańcuchów, 99
 - tokenów, 456
- przyciski metody popup, 139
- przypisywanie
 - wartości domyślnej, 380
 - zmiennych globalnych, 374
- przywracanie danych, 492
- pulpit zdalny, 201
- punkt kontrolny, 606
 - na poziomie aktywności, 607
 - na poziomie przepływu pracy, 607
- punkt wstrzymania, 547–550
- PWA, PowerShell Web Access, 43

R

- Rahim Ibrahim Abdul, 417, 430, 432
- raport, 415, 432, 457
- reagowanie na punkty wstrzymania, 555
- reguła zabezpieczeń, 393
- rejestr, 91, 178, 222, 465, 520
- rejestr błędów, 405
- rejestrowanie
 - danych, 442, 508, 509
 - wyników, 499, 500
 - zdarzeń, 519
- repozytorium, 492
- Riedel Alexander, 496
- Ring Jan Egil, 37
- rodzaje błędów, 391
- rola FSMO, 65, 70
- role użytkownika, 107
- Rottenberg Hal, 142
- rozwiązywanie problemów, 504, 523
- rozwijanie nazw, 567
- równoległe wykonywanie poleceń, 599

RPC, remote procedure call, 95
 RSAT, Remote Server Administration Tools, 65
 rzutowanie wartości parametrów, 451

S

Sajid Osama, 560
 scenariusz używania, 379
 Schneider Andy, 546
 Schwinn Dave, 369
 SDK, Software Development Kit, 185
 sekcja
 Function, 416
 robocza skryptu, 279
 Shell Brandoe, 242
 Siddaway Richard, 468
 Siepser Gary, 338, 390
 składnia podstawowa, 411
 składowe
 klasy SecurityIdentifier, 100
 klasy System.Math, 183–185
 klasy Win32_LogicalDisk, 245–247
 obiektu Shell.Application, 188, 190
 obiektu WshShell, 137
 skompilowane pliki pomocy, 464
 skrót, hash, 353
 skrypt
 AddOne.ps1, 372
 AddTwoError.ps1, 543, 544
 BackUpFiles.ps1, 272
 BadScript.ps1, 535
 CheckForPdfAndCreateMarker.ps1, 288
 CheckNumberRange.ps1, 343
 CheckProviderThenQuery.ps1, 398
 CmdLineArgumentsTime.ps1, 292
 ConversionFunctions.ps1, 153, 241
 ConvertToFahrenheit_Include.ps1, 154, 291
 ConvertUseFunctions.ps1, 154
 Copy-Modules.ps1, 312, 321
 CreateCab.ps1, 187, 188
 CreateCab2.ps1, 194
 CreateFileNameFromDate.ps1, 272
 CreateRegistryKey.ps1, 521, 531, 534
 CreateScriptingRegistryKey.ps1, 223
 DebugRemoteWMI_Session.ps1, 526
 DefragAnalysisReport.ps1, 220
 DemoConsoleBeep.ps1, 293
 DemoTrapSystemException.ps1, 251
 DemoUserConfig.ps1, 619
 DisplayProcessor.ps1, 485
 DotSourceScripts.ps1, 258
 ExpandCab.ps1, 191
 ExportBiosToCsv.ps1, 158
 FilterHasMessage.ps1, 266
 FilterToday.ps1, 267
 FindDisabledUserAccounts.ps1, 287
 FindLargeDocs.ps1, 255
 FunctionGetIPDemo.ps1, 258
 GetAdminFunction.ps1, 110
 Get-AllowedComputer.ps1, 387, 388
 Get-AllowedComputerAndProperty.ps1, 389
 Get-Bios.ps1, 329, 332, 389
 Get-BiosArray1.ps1, 331
 Get-Biosarray2.ps1, 331
 Get-BiosInformation.ps1, 380
 Get-BiosInformationDefaultParam.ps1, 381
 GetBiosMandatoryParameter.ps1, 337
 Get-BiosMandatoryParameterWithAlias.ps1, 341
 Get-BiosParam.ps1, 336
 Get-ChoiceFunction.ps1, 383
 GetCmdletsWithMoreThanTwoAliases.ps1, 122
 GetCommentsFromScript.ps1, 452
 GetComputerInfoWorkFlow.ps1, 601
 Get-ComputerWmiInformation.ps1, 419
 Get-CountryByIP.ps1, 510, 511
 Get-DiskSpace.ps1, 247
 GetDrivesCheckAllowedValue.ps1, 408, 409
 GetDrivesValidRange.ps1, 410
 Get-EnabledBreakpointsFunction.ps1, 557
 Get-EventLogData.ps1, 605
 Get-IPObjectDefaultEnabled.ps1, 259
 Get-MemberOf.ps1, 103
 Get-ModifidFiles.ps1, 421, 422
 Get-ModifidFilesUsePipeline.ps1, 423–425
 Get-MoreHelpWithAlias.ps1, 127
 Get-OSVersion.ps1, 115
 Get-PowerShellRequirements.ps1, 26
 Get-PsVersionNet.ps1, 92
 Get-PsVersionRegistry.ps1, 89
 Get-PSVersionRemoting.ps1, 94
 Get-PsVersionWmi.ps1, 92
 Get-PSVersionWorkflw.ps1, 93
 GetRandomObject.ps1, 159
 GetRunningService.ps1, 117, 118
 Get-ScriptHelp.ps1, 447
 Get-ScriptVersion.ps1, 493
 GetServicesInSvchost.ps1, 294
 GetSetieStartPage.ps1, 276

- Get-ValidWmiClassFunction.ps1, 404, 405
- Get-Version.ps1, 165
- get-VM.ps1, 395
- Get-WindowsEdition.ps1, 495
- Get-WinFeatureServersWorkflow.ps1, 609
- Get-WmiClass.ps1, 130
- Get-WmiClass2.ps1, 132
- Get-WmiClass2WithAlias.ps1, 133
- GetWmiClassesFunction1.ps1, 283
- Get-WmiProviderFunction.ps1, 402
- Get-WmiProviders.ps1, 172
- InLineGetIPDemo.ps1, 257
- InternetScript.ps1, 464
- LogChartProcessWorkingSet.ps1, 505
- LogonScriptWithLogging.ps1, 501
- MandatoryParameter.ps1, 381
- MeasureAddOneFilter.ps1, 264
- MeasureAddOneFunction.ps1, 264
- My-Function.ps1, 528
- New-LocalGroupFunction.ps1, 438
- New-TempFile.ps1, 513, 514
- ParseScriptCommands.ps1, 455–457
- PingComputers.ps1, 344
- PingIpAddress.ps1, 345
- PinToStartAndTaskBar.ps1, 33
- PromptForChoice.ps1, 358
- RecursiveWMINameSpaceListing.ps1, 169
- RemoteWMISSessionNoDebug.ps1, 526
- RemoveUserFromGroup.ps1, 76
- RequireModuleVersion.ps1, 395
- RequiresModule.ps1, 118
- SaveWmiInformationAsDocument.ps1, 225
- ScriptFolderConf.ps1, 616
- ScriptFolderVersion.ps1, 617
- ScriptFolderVersionUnzip.ps1, 618
- ScriptFolderVersionUnzipCreateUsersAnd
 - ↳ Profile.ps1, 622
- SearchAllComputersInDomain.ps1, 295
- SearchForWordImages.ps1, 289
- Set-LocalGroupFunction.ps1, 436
- Set-SaverTimeOut.ps1, 181
- SetScriptExecutionPolicy.vbs, 467
- SetServicesConfig.ps1, 623
- SimpleTypingError.ps1, 542
- SimpleTypingErrorNotReported.ps1, 542
- skrypt StringArgsArray.ps1, 332
- skrypt StringArgsArray2.ps1, 332
- skrypt
 - Switch_DebugRemoteWMISSession.ps1, 526, 527
 - skrypt TestAdminCreateEventLog.ps1, 518
 - skrypt Test-ComputerPath.ps1, 384
 - skrypt Test-IsAdminFunction.ps1, 108, 394
 - Test-IsInRole.ps1, 110
 - Test-Script.ps1, 467
 - Test-ScriptHarness.ps1, 411–415
 - Test-TwoScripts.ps1, 426, 429
 - TranscriptBios.ps1, 441, 442
 - UpdatehelptrackErrors.ps1, 37
 - UseADCmdletsToCreateOuComputerAnd
 - ↳ User.ps1, 75
 - UseGetMemberOf.ps1, 103, 106
 - ValidateRange.ps1, 343
 - WriteBiosInfoToWord.ps1, 290
- skrypty, 157
 - analiza składowa, 455
 - brak obsługi aplikacji, 191
 - brak obsługi COM, 185
 - brak obsługi platformy, 182
 - brak obsługi poleceń, 157
 - brak obsługi WMI, 167
 - braki, 292
 - diagnostyczne, 483
 - dokumentowanie, 447
 - kontrola wersji, 489
 - kończenie pracy, 393
 - korzyści stosowania, 217
 - krok po kroku, 536, 541
 - lista warunków używania, 291
 - logowania, 475–480
 - obsługa błędów, 379
 - opisywanie struktur, 295
 - pomocy technicznej, 484
 - porównanie szybkości działania, 424
 - powody napisania, 293
 - projektowanie, 231
 - projektowanie pomocy, 271
 - raportujące, 483
 - rozwiązywanie problemów, 523
 - samodzielne, 483
 - skomplikowane konstruktory, 159
 - stosowanie, 197
 - testowanie, 411
 - testowanie wydajności, 421
 - uprawnienia, 393
 - uruchamianie, 417, 475
 - wdrażanie, 459
 - zasady wykonywania, 461
 - zdolność do adaptacji, 223

słowo kluczowe

- Configuration, 614
- constant, 125
- DependsOn, 618
- Filter, 255
- Function, 126, 192, 232, 239, 244
- Mandatory, 381
- Parallel, 605
- param, 526
- read-only, 124
- Workflow, 598
- Snover Jeffrey, 95, 107, 116
- spójność wyników, 243
- sprawdzanie
 - konfiguracji węzła, 625
 - poprawności danych, 342
 - wartości granicznych, 408
 - wersji platformy .NET, 112
 - wersji systemu, 165
 - zależności modułów, 314
 - zawartości tablicy, 385
 - sprawdzenie wersji systemu, 165
- Stahler Wes, 479
- standard WS-Management Protocol, 583
- standardowe
 - czasowniki, 232
 - konwencje nazewnicze, 158
- status usługi bits, 157
- Stewart Bill, 406
- stosowanie znaków specjalnych, 212
- Stranger Stefan, 221
- strefa internetowa, 463
- struktura komentarza, 275
- struktury zagnieżdżone, 295
- strumień
 - Debug, 393, 509
 - Error, 509
 - Success, 509
 - Verbose, 509
 - Warning, 509
- strzałka przekierowująca, 366
- sygnatura metody popup, 138
- symbol |, 126
- symbole wieloznaczne, 214, 565
- system
 - zabezpieczeń typów, 451
 - ETS, 143
 - operacyjny, 113, 115
- szacowanie wydajności, 426

sztuczki, 163

szukanie błędów, 415

szyfrowanie plików, 349

Ś

ścieżka, 91, 141

- do folderu, 616

- do funkcji, 127

- do konfiguracji, 616

- lokalna, 205

- UNC, 198, 200, 513

śledzenie

- wykonywania skryptu, 531

- zmian, 491, 495

środowisko

- skryptowe, 121

- Windows PowerShell ISE, 561, 563

T

Tabdilio Mark, 267

tablica, 128

- \$Error, 252

- \$servers, 388

techniki obsługi błędów, 406

technologia

- ADO, 208

- Adobe Flash, 433

- DCOM, 95

- Microsoft Silverlight, 433

- RPC, 95

- WMI, 95, 176

testowanie

- aplikacji graficznych, 432

- funkcji, 435

- interfejsów API, 430

- interfejsów użytkownika, 433

- podstawowej składni, 411

- skryptów, 411, 440

- usług sieciowych, 430

- własności, 387

- wydajności skryptów, 421, 425

- zaawansowane, 443

token, 455

transakcja, 178

Truman Jeff, 33

tryb

- interaktywny, 584

- Set-PSDebug -strict, 542

- ściśły, 542

Tsaltas Deae, 273

tworzenie

aliasów, 122

aliasów nazw funkcji, 127

aliasów stałych, 125

aliasu parametru, 341

biblioteki funkcji, 153

dysków PowerShell, 141

dysku modułu, 313

dziennika zdarzeń, 519

dzienników, 431, 441, 515

egzemplarzy klas, 454

fragmentów kodu, 569

funkcji, 125

grup, 621

grup zabezpieczeń, 75

jednostki organizacyjnej, 74

jednowierszowych komentarzy, 280

kluczy rejestru, 222

komentarzy wielowierszowych, 279

konta komputera, 74

modułu, 319

obowiązkowego parametru, 337

parametrów obowiązkowych, 381

plików cabinet, 185, 192

pliku konfiguracji, 481

pliku MOF, 617

profilu, 148, 150

punktów kontrolnych, 606

punktu wstrzymania, 553

ścieżki, 135

tablicy, 386

tablicy skrótów, 530

użytkownika, 77, 621

zadań, 590

zdalnej sesji, 586

zmiennych, 134

Tyler Jonathan, 111

typy danych, 403

U

UAC, User Account Control, 389

udziały UNC, 465, 525

umysł nadrzędny, 308

UNC, Universal Naming Convention, 198,
465, 513

unikanie wprowadzania błędów, 490

UPN, User Principal Name, 212

upraszczanie kodu, 425

uprawnienia, 389, 481

administracyjne, 109, 394, 466

skryptu, 393

uruchamianie

przepływu pracy, 599

skryptu, 417, 475

zadania, 593

usługa

Active Directory, 73

AeLookupSvc, 206

AWDS, 62

bits, 157

DSC, 613

SkyDrive, 143, 144

Windows Update, 221

WinRM, 583, 585

WMI, 45

WSMAN, 580

usługi

domenowe, 82

sieciowe REST, 431

sieciowe SOAP, 431

ustalanie punktów kontrolnych

dla przepływów pracy, 606

ustawianie

punktu wstrzymania, 550

na poleceniu, 552

na wierszu kodu, 547

na zmiennej, 549

zabezpieczeń, 468

zależności, 618

ustawienia pulpitu, 176

ustawienie

AllSigned, 468

Bypass, 472

RemoteSigned, 469, 471

Restricted, 470

Undefined, 472

Unrestricted, 468, 471

usuwanie

błędów wykonawczych, 527

fragmentów kodu, 570

niepotrzebnych informacji, 51

punktów wstrzymania, 559

usterek, 491

użytkownik, 77

używanie

atrybutów parametrów, 339

CIM, 625

dzienników, 504, 519

używanie

- filtrów, 266
- fragmentów kodu, 569
- parametrów standardowych, 434
- parsera AST, 455
- pomocy komentarzowej, 281
- potoku, 423
- profilu, 151
- przełącznika debug, 528
- wielu argumentów parametrów, 347

V

VSS, Visual SourceSafe, 496

W

Walker Jason, 170

wartości

- domyślne parametru, 380
- graniczne, 408
- logiczne, 388
- metody popup, 140
- parametrów instrukcji #Requires, 394
- parametru, 409
- reprezentujące ikony, 141
- WMI, 91
- wyliczeniowe, 404

wartość

- Continue, 404
- Inquire, 404
- null, 400, 501
- SilentlyContinue, 404
- Stop, 404

WASP, Web Application Services Platform, 434

WbemTest, 396

wczytywanie danych, 328

wdrażanie

- konsoli, 26
- lokalne, 482
- pakietu MSI, 482
- skryptu, 459
- zasady wykonywania, 465, 469

wersja

- modułu, 395
- platformy .NET, 111
- systemu operacyjnego, 115, 165, 235

weryfikowanie danych, 385

węzeł, node, 614

wielokrotne wykorzystanie kodu, 224, 240

wielowierszowe znaczniki komentarzowe, 279

wiersz poleceń, 28, 214, 328

Wilhite Brian, 46, 64, 202

Willett Andrew, 515

Wilson Ed, 627

Windows PowerShell ISE, 563

Windows Server 2012 R2, 583

WinRM, Windows Remote Management, 583

własności

- klasy __provider, 398, 399
- klasy InvocationInfo, 180
- obiektu ScriptInfo, 181
- statyczne, 182
- użytkownika, 83

własność

- \$filepath, 220
- CLSID, 400
- ProductType, 165
- ScreenSaverTimeout, 178
- ScriptName, 223

włączanie

- konsoli, 32
- obsługi skryptów, 146, 565
- punktów wstrzymania, 558
- trybu ścisłego, 542
- debugera, 550

WMI, Windows Management Instrumentation, 45, 88, 167, 235

Wouters Jeff, 84

wprowadzanie danych do zmiennej globalnej, 372

WSH, Windows Script Host, 406, 496

wstrzymywanie działania skryptu, 558

wybór

- profilu, 148
- zasady wykonywania, 461

wydajność skryptów, 421, 426, 432

wygaszacz ekranu, 177, 178

wyjątek ParameterBindingException, 334

wyjątki

- typu CommandNotFoundExceptions, 392
- w zaporze systemu, 202

wykonywanie

- aktywności, 610
- pojedynczego polecenia, 588
- przepływu pracy, 605, 610
- skryptu, 535, 539–541

wykrywanie

- bieżącego użytkownika, 97
- roli użytkownika, 107

wyłączanie punktów wstrzymania, 558

wymagania

dotyczące

aplikacji, 117

bezpieczeństwa, 96

modułów, 118, 395

systemu operacyjnego, 113

uprawnień administratora, 394

wersji .NET, 111

przepływów pracy, 598

strukturalne, 96

systemowe, 18

wynik

audytu, 477

testu wydajności, 425

wysyłanie

danych dziennika, 516

plików tekstowych, 516

wyników

do pliku, 366, 367

na adres e-mail, 371

na ekran, 360, 367

z funkcji, 371

zapytań, 208

wyszukiwanie

danych, 267

egzemplarzy klas, 49

poleceń, 159, 565

wyszukiwarka Bing.com, 433

wyświetlanie

błędów, 500

informacji, 42

monitu, 352

punktów wstrzymania, 556

składni polecenia, 568

własności użytkownika, 83

wyników, 431

wywołanie

funkcji, 383

skryptu logowania, 480

Y

Yoder Jason A., 217

Z

zabezpieczenia przeglądarki, 464

zachowanie zgodności, 491

zadania, 573, 590

zadanie Job10, 590

zakres

CurrentUser, 472

grupy, 75

LocalMachine, 472

Process, 471

zasady wykonywania, 147

zalety przepływów pracy, 611

zapisywanie

błędów, 517

danych, 421

w dzienniku zdarzeń, 517

w pliku, 513

w pliku tekstowym, 499

w rejestrze, 520

hasła, 620

w pliku tekstowym, 350

w rejestrze, 351

w skrypcie, 349

w usługach domenowych, 351

nieprzechwyconych wyników, 262

zapytania, 208

zapytania LDAP, 208

zapytanie Get-EventLog, 477

zarządzanie użytkownikami, 74

zasady

biznesowe, 528

grupy, 469, 476

wykonywania skryptów, 146, 461, 471

zasoby WMI, 167

zastosowania skryptów, 87

zastosowanie

akceleratora [ADSISeacher], 209

komentarzy jednowierszowych, 294

modułu Active Directory, 63

zatrzymywanie usług, 175

zatwierdzanie poleceń, 30

zawieszenie polecenia, 31

zdalna sesja, 586

zdalne

interaktywne sesje, 96

komputery, 94

zarządzanie systemem, 583

zdalny pulpit, 201

zdobywanie informacji, 38

zestaw słów kluczowych, 296

zgodność wersji, 160

zmienianie nazw lokalizacji, 73

zmienna, 134

- \$args, 129, 328–335
- \$baseLine, 428
- \$bios, 368
- \$changedFiles, 425
- \$clsID, 401
- \$computer, 203, 335
- \$credential, 355
- \$debug, 191
- \$DebugPreference, 179, 194, 434, 526
- \$driveData, 247
- \$error, 251, 333, 413, 509
- \$errorActionPreference, 404–407, 412, 504
- \$etime, 414
- \$files, 422
- \$hklm, 91, 92
- \$InputObject, 514
- \$isAdmin, 108
- \$key, 92
- \$LastExitCode, 509
- \$localappdata, 134
- \$makecab, 186
- \$MaximumErrorCount, 509
- \$modulePath, 311
- \$myInvocation.InvocationName, 441
- \$noun, 385
- \$null, 424
- \$number, 372
- \$numberOfTests, 428
- \$outpath, 77
- \$outputPath, 277
- \$path, 91, 239, 412
- \$ping, 369
- \$pinToStart, 32
- \$pinToTaskBar, 32
- \$profile, 150
- \$providerName, 399
- \$proxy, 510
- \$PSVersionTable, 93
- \$report, 412
- \$return, 139
- \$sourceCab, 191
- \$startTime, 413, 504
- \$tmpFile, 514
- \$trace, 505
- \$value, 92
- \$verbosePreference, 400

- \$wshShell, 91

- \$wuLog, 134

- docs, 136

- path, 286

- zmienne globalne, 372

- znacznik

- [ordered], 530

- czasu, 442

- description, 510

- example, 510

- inputs, 510

- komentarzowy, 279

- notes, 510

- outputs, 510

- skryptu, 155

- specjalny pomocy, 283

- synopsis, 510

- Zone.Identifier, 464

- znaczniki

- komentarzy wielowierszowych, 280

- parametrów, 337

- pomocy funkcji, 282, 283

- znajdowanie

- dostępnych komputerów, 384

- kont użytkowników, 78

- metod, 47

- modułów, 302

- nieużywanych kont użytkowników, 81

- wyłączonych użytkowników, 80

- znak

- #, 419

- @, 192

- dolara, 247, 329

- potoku (|), 295

- 't, 259

- znaki

- specjalne, 105

- specjalne filtrów, 211, 212

- zwracanie danych, 358

Ż

- źródła

- błędów wykonawczych, 526

- informacji, 43, 57, 85, 119, 156, 195, 227, 268,

- 300, 325, 377, 410, 445, 457, 474, 487, 497,

- 522, 561, 571, 596, 611, 625

PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION



1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW
w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

<http://program-partnerski.helion.pl>