

Budowanie aplikacji biznesowych przy użyciu Windows® Presentation Foundation i wzorca MVVM

Raffaele Garofalo

Przekład:
Jakub Niedźwiedź

APN Promise
Warszawa 2011

Budowanie aplikacji biznesowych przy użyciu Windows® Presentation Foundation i wzorca MVVM

© 2011 APN PROMISE SA

Authorized Polish translation of English edition of
Building Enterprise Applications with Windows® Presentation Foundation and the
Model View ViewModel Pattern

ISBN: 978-0-735-65092-3

Copyright © 2011 Raffaele Garofalo. All rights reserved

This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

APN PROMISE SA, ul. Kryniczna 2, 03-934 Warszawa

tel. +48 22 35 51 600, fax +48 22 35 51 699

e-mail: mspress@promise.pl

Microsoft, Windows, Windows Server, Windows Vista, Visual C#, SQL Server, Active Directory, IntelliSense, Silverlight, MSDN, Internet Explorer i Visual Studio to znaki towarowe grupy Microsoft. Wszystkie inne znaki towarowe są własnością ich odnośnych właścicieli.

Książka ta przedstawia poglądy i opinie autora. Informacje i widoki przedstawione w tym dokumencie, w tym adresy URL i inne odniesienia do witryn internetowych, mogą być zmieniane bez powiadomienia. Państwo sami ponoszą ryzyko związane z ich wykorzystywaniem.

Niektóre przykłady zamieszczone w książce są fikcyjne i są jedynie ilustracją omawianej tematyki. Żadne podobieństwa czy powiązania nie były zamierzone i nie należy też wyciągać wniosków o istnieniu takich związków.

Niniejszy dokument nie zapewnia żadnych praw do żadnej własności intelektualnej w żadnym produkcie firmy Microsoft. Mogą Państwo kopiować i wykorzystywać ten dokument jedynie dla własnych potrzeb.

APN PROMISE SA dołożyła wszelkich starań, aby zapewnić najwyższą jakość tej publikacji. Jednakże nikomu nie udziela się rękojmi ani gwarancji.

APN PROMISE SA nie jest w żadnym wypadku odpowiedzialna za jakiegokolwiek szkody będące następstwem korzystania z informacji zawartych w niniejszej publikacji, nawet jeśli APN PROMISE została powiadomiona o możliwości wystąpienia szkód.

ISBN: 978-83-7541-078-5

Przekład: Jakub Niedźwiedź

Redakcja: Marek Włodarz

Korekta: Ewa Swędrowska

Skład i łamanie: MAWart Marek Włodarz

Spis treści

Wstęp	vii
Podziękowania	xv
Errata i wsparcie dla tej książki	xvi

Rozdział 1

Wprowadzenie do aplikacji biznesowych

i wzorca Model View ViewModel	1
Wzorzec Model View ViewModel	1
Aplikacje biznesowe	3
Wybór odpowiedniej technologii	4
Silverlight czy WPF?	4
Narzędzia firmy Microsoft do budowania interfejsu użytkownika	6
Budowanie interfejsu użytkownika	10
Pasek menu	12
Pasek narzędzi	13
Etykiety narzędzi (i ich nadużywanie)	14
Powiadomienia i alarmy	15
Pasek wstążki	17
Ogólne rozważania dotyczące stylów kontroltek	18
Podział interesów	19
Warstwy, poziomy i usługi	21
Podsumowanie	25

Rozdział 2

Wzorce projektowe

Przegląd wzorców projektowych	27
Klasyfikowanie wzorców projektowych	29
Wzorce projektowe interfejsu użytkownika	32
Wzorzec MVC	34
Wzorzec MVP	38
Wzorzec PM i MVVM	42

Zaawansowane wzorce i techniki projektowe.....	46
Wzorzec odwrócenia sterowania	47
Języki DSL: pisanie płynnego kodu	56
Wprowadzenie do TDD	61
Podsumowanie	63

Rozdział 3

Model domenowy65

Wprowadzenie do projektowania sterowanego domeną.....	65
Terminologia DDD	66
Analizowanie domeny aplikacji CRM	67
Jednostka domenowa i obiekt transferu danych	69
Obiekt POCO i O/RM	70
Podejścia do projektowania domeny	72
Skrypt transakcyjny.....	72
Podejście sterowane bazą danych	73
Podejście sterowane domeną	74
Jak utworzyć obiekt w DDD.....	76
Wzorce fabryki	77
Sprawdzanie poprawności jednostek domenowych	79
Klasyczne sprawdzanie poprawności.....	80
Sprawdzanie poprawności z wykorzystaniem atrybutów i adnotacji danych	82
Dostępne platformy sprawdzania poprawności danych	84
Testy jednostkowe modelu domenowego.....	85
Kod przykładowy: model domenowy CRM.....	86
Kontekst osoby	86
Domena Order	92
Podsumowanie	94

Rozdział 4

Warstwa dostępu do danych95

Wprowadzenie	95
Baza danych i procedury składowane	96
Wybór systemu O/RM.....	98
Microsoft Entity Framework	99
NHibernate	103
Inne narzędzia O/RM dla .NET.....	105
Jednostka pracy	106
Cykl życia jednostki pracy	107

Identyfikowanie transakcji biznesowej	108
Wzorzec repozytorium	109
Programowanie sterowane testami: warstwa danych	111
Budowanie rozproszonej warstwy danych przy pomocy RIA i WCF	114
Kod przykładowy: warstwa dostępu do danych aplikacji CRM	117
Elastyczny interfejs <i>IUnitOfWork</i>	117
Mapowanie modelu domenowego przy użyciu Entity Framework	119
Mapowanie domeny przy użyciu NHibernate	123
Zebranie narzędzi	123
<i>UnitOfWork</i> i <i>ISession</i>	124
Podsumowanie	127

Rozdział 5

Warstwa biznesowa

Wprowadzenie	129
Reguła biznesowa nie jest regułą sprawdzania poprawności.	130
Reguły biznesowe w usłudze	133
Wzorzec fasady	134
Reguły biznesowe w przepływie zadań przy użyciu WF 4.0.	135
Różne sposoby wykonywania przepływu zadań	137
Zestawy narzędziowe firm trzecich	140
Technologie służące do sprawdzania poprawności danych	140
Silnik reguł i silnik reguł biznesowych	142
Względy związane z warstwą biznesową	143
Kiedy musimy stworzyć warstwę biznesową?	143
Złe nawyki związane z warstwą BLL	144
Kod przykładowy: Warstwa usługi biznesowej	145
Sprawdzanie poprawności danych przy pomocy Enterprise Library 5.0. ...	145
Ogólny silnik przepływów zadań	148
Usługa dla transakcji biznesowych	149
Podsumowanie	153

Rozdział 6

Warstwa interfejsu użytkownika w MVVM

Wprowadzenie do wzorca MVVM	156
Widok	157
Blend: atropa modelu widoku	158
Model	161
Polecenie w WPF i Silverlight	163

Obejście problemu: MVVM Command	164
Ponowne ocenianie możliwości wykonywania <i>ICommand</i>	167
Model widoku.	168
Interfejs <i>INotifyPropertyChanged</i>	168
Interfejs <i>IDataErrorInfo</i>	170
Szablon <i>DataTemplate</i> w WPF i Silverlight	173
<i>DataTemplate</i> a MVVM	174
Zdarzenia <i>WeakEvent</i> i komunikaty	175
Wzorzec <i>WeakEvent</i>	175
Wzorzec <i>EventAggregator</i>	176
Okna dialogowe i modalne okna wyskakujące	178
Modalny widok w MVVM	178
Odwrócenie sterowania w MVVM	181
Kod przykładowy	182
Microsoft Office Ribbon a MVVM	182
Podsumowanie	184

Rozdział 7

Platformy i zestawy narzędzi MVVM185

Zestawy narzędzi dla MVVM	185
Zestaw narzędzi MVVM Light Toolkit autorstwa Laurenta Bugniona	186
MEFedMVVM	187
Cinch autorstwa Sachy Barbera	189
Inne narzędzia dla MVVM i XAML	189
Narzędzia Karla Shiffletta	190
Radical autorstwa Maura Servientiego	191
Platformy dla złożonych interfejsów użytkownika	192
Microsoft Prism	193
Caliburn	196
O autorze	197

Wstęp

Windows Presentation Framework (WPF), Silverlight i Windows Phone 7 są najnowszymi technologiami budowania elastycznych interfejsów użytkownika (UI) dla aplikacji tworzonych przy użyciu narzędzi firmy Microsoft. Wszystkie te trzy technologie opierają się na języku znaczników XAML przy opisywaniu elementów i układu interfejsu użytkownika, a aplikacje dla wszystkich tych trzech platform można programować przy użyciu najczęściej stosowanych języków Microsoft .NET Framework: Visual C# lub Visual Basic .NET. Będąc programistą .NET planującym stworzenie nowej aplikacji biznesowej przy użyciu .NET Framework warto rozważyć zastosowanie którejś z tych technologii do utworzenia interfejsu użytkownika. Planując zbudowanie aplikacji w oparciu o jedną z tych technologii warto też poważnie rozważyć nauczanie się i zastosowanie wzorca prezentacyjnego Model View ViewModel (MVVM), który jest wzorcem projektowym stworzonym specjalnie dla tych technologii.

Tego właśnie dotyczy niniejsza książka. Można by się zastanawiać, po co kolejna książka dotycząca WPF? Po zapoznaniu się ze spisem treści można by z kolei pomyśleć, po co kolejna książka na temat wzorców projektowych?

Aby odpowiedzieć na te pytania, zacznę od stwierdzenia, że od dawna zauważyłem, iż programiści nie szukają „Biblii wzorców” ani „Biblii opisującej tworzenie układu aplikacji”, chcieliby natomiast mieć dostęp do prostej książki prowadzącej ich przez etapy opracowywania rzeczywistej, ale prostej aplikacji, która *stosuje i wyjaśnia* wzorce – a jednocześnie może być ponownie wykorzystana w przyszłych projektach jako „szablon” dla kolejnych aplikacji.

WPF i Silverlight są młodymi technologiami i procentowy udział programistów przechodzących na te nowe sposoby projektowania interfejsu użytkownika jest nadal niewielki. Jest kilka powodów takiego stanu rzeczy. Po pierwsze, krzywa uczenia się jest dosyć stroma. Jeśli ktoś przyzwyczał się do technologii Windows Forms, Java Swing lub Delphi, to sposób projektowania i organizowania struktury aplikacji przy użyciu XAML i WPF jest znacząco różny – w istocie nazwałbym go „rewolucyjnym”.

W przeszłości korzystałem z dobrze znanych wzorców do budowania aplikacji, w tym z wzorca Model View Presenter w przypadku aplikacji Windows Forms oraz wzorca Model View Controller w przypadku aplikacji ASP.NET. Jednak w przypadku WPF te dwa podejścia są przestarzałe, ponieważ nie mogą korzystać z wydajnych mechanizmów zapewnianych przez XAML. Oczywiście nadal można korzystać z mechanizmów wiązania w WPF stosując wzorec Model View Presenter, ale kosztem zbyt dużego wysiłku. Na szczęście wzorec MVVM zapewnia dobrą alternatywę.

Firma Microsoft we współpracy z kilkoma architektami przetworzyła pierwotny model prezentacyjny, który wiele lat temu zaproponował Martin Fowler. Ta wersja (nazwana wzorcem Model View ViewModel) jest idealnym podejściem w przypadku WPF i Silverlight, ponieważ została zaprojektowana specjalnie dla tych technologii! Niestety podobnie jak XAML, MVVM jest dosyć nową technologią, więc obecnie nie ma zbyt wielu informacji na temat jej implementowania. Jest kilku blogerów stosujących podejście MVVM i piszących na ten temat; są też osoby związane z budowaniem narzędzi przeznaczonych dla MVVM. Niemal wszystko jest nadal w fazie eksperymentów i istnieje niewiele naprawdę konkretnych przykładów.

Stąd wziął się pomysł na książkę dotyczącą budowania aplikacji biznesowej przy użyciu MVVM. W miarę czytania tej książki zobaczymy przykłady, które pokażą, jak zbudować prosty program do obsługi relacji z klientami (CRM) przy pomocy WPF 4, Silverlight 4 i wzorca MVVM. Ta książka będzie przewodnikiem przez cały proces architektoniczny ilustrując odpowiednie podejście do zastosowania MVVM. Skorzystamy też z kilku innych nowych technologii dostarczanych przez platformę Microsoft .NET 4, takich jak Managed Extensions, Windows Workflow Foundation 4 i oczywiście Entity Framework.

Najpierw pokażemy narzędzia. Następnie przejdziemy do budowania aplikacji CRM zaczynając od modelu domenowego, stosując prostą technikę zachowywania danych w relacyjnej bazie danych przy pomocy dwóch najpopularniejszych systemów Object-Relational Mapper (O/RM) dostępnych dla .NET: Entity Framework i NHibernate. Następnie zobaczymy, jak dodać większą elastyczność przy pomocy platformy MEF.

Wreszcie nauczymy się stosować w tym modelu logikę biznesową i sprawdzanie poprawności danych w sposób spełniający wymagania wzorca MVVM. W tej fazie przyjrzymy się też Windows Workflow Foundation (WF) 4.0, stworzonemu przez Microsoft wydajnemu, nowemu silnikowi przepływów zadań oraz zbadamy kroki wymagane do zbudowania prostego silnika przepływów zadań.

Pozostałe rozdziały skupiają się na MVVM. Istnieją cztery główne pojęcia, które trzeba poznać, aby prawidłowo korzystać z MVVM: *polecenia*, *szablon*, *silnik wiązań*, oraz *orkiestracja* wszystkiego razem. Podczas tego procesu odwiedzimy wszystkie warstwy wymagane do ukończenia klasycznej aplikacji biznesowej, a co ważniejsze, będziemy w stanie ponownie wykorzystać opisane tu części jako szablony do budowania przyszłych aplikacji. Istnieją oczywiście pewne różnice pomiędzy WPF a Silverlight, więc niniejsza książka spróbuje omówić je w miarę możliwości.

Na koniec krótko przyjrzymy się dostępnym zestawom narzędzi MVVM, takim jak PRISM, który jest złożoną platformą aplikacyjną dla WPF i Silverlight. Pomoże nam to w określeniu, kiedy i jak powinniśmy korzystać z każdego z nich podczas procesu budowania niewielkiej i elastycznej platformy MVVM.

Kluczowym celem tej książki jest zapewnienie pełnego przewodnika opisującego krok po kroku korzystanie z WPF/Silverlight w połączeniu z MVVM przy tworzeniu ogólnego kodu, który będzie można ponownie wykorzystać w przyszłości.

Budowanie aplikacji biznesowych przy pomocy Windows Presentation Foundation i wzorca Model View ViewModel zapewnia nie tylko solidną analizę sposobu działania wzorca MVVM i jego zastosowania w połączeniu z WPF i Silverlight, ale także oferuje wyczerpujący przewodnik po budowaniu warstwowych aplikacji przy użyciu najczęściej stosowanych technik. Książka ta celowo nie pokazuje *całego* kodu związanego z danym projektem; skupia się natomiast bardziej na zasadach i wzorcach, które programiści powinni stosować przy tworzeniu dobrze zorganizowanych i łatwych w utrzymaniu aplikacji biznesowych.

Ta książka analizuje każdą warstwę składającą się na aplikację biznesową począwszy od modelu domenowego (znanego też jako warstwa biznesowa), poprzez warstwę danych (omawiając przy tym Entity Framework i NHibernate), a kończąc rozdziałem poświęconym regułom biznesowym i Windows Workflow Foundation. Oczywiście pojawi się też rozdział poświęcony wzorcowi MVVM.

Oprócz wzorców i praktyk wyjaśnionych w tej książce, rozdział 7 zawiera przydatną listę platform i wytyczek z otwartym kodem źródłowym stosowanych przez innych członków społeczności .NET do budowania aplikacji implementujących wzorec MVVM w technologiach WPF lub Silverlight.

Kto powinien przeczytać tę książkę

Ta książka jest przeznaczona dla każdego programisty lub architekta oprogramowania .NET, który chce nauczyć się, jak budować aplikacje biznesowe korzystając z dobrze znanych biznesowych wzorców architektonicznych, w tym wzorca MVVM. Czytelnicy powinni mieć już solidnie opanowane podstawowe zagadnienia związane z programowaniem, znać ogólny cel i zastosowanie wzorców, a także oczywiście znać technologie WPF, Silverlight lub Windows Phone 7. Choć książka ta porusza wszystkie te zagadnienia, to nie próbuje nauczyć podstaw programowania lub zasad stosowania wzorców. Jest natomiast przeznaczona dla programistów i architektów, którzy budowali już aplikacje .NET i chcą zająć się projektowaniem i budowaniem biznesowych aplikacji przy pomocy .NET.

W szczególności książka ta przeznaczona jest dla programistów WPF lub Silverlight, którzy mają już doświadczenie w jednej lub obu tych technologiach, ale którzy nie wiedzą jeszcze, jak implementować wzorec MVVM – albo dla programistów, którzy w jakimś stopniu zaznajomili się już z MVVM i chcą opanować techniki efektywnego zastosowania wzorca MVVM. W tym celu trzeba mieć pewną podstawową wiedzę na temat WPF i Silverlight; jeśli jej komuś brakuje, to przed przeczytaniem tej książki zalecam zapoznanie się z tematami poleceń wytyczanych, wiązania danych i stylów.

Założenia

Koncentrując się na wzorcach projektowych, architekturach oprogramowania oraz zwinnych technikach i metodologiach programowania (agile) książka ta zakłada u czytelnika podstawowe umiejętności tworzenia aplikacji WPF lub Silverlight przy pomocy .NET Framework i Visual Studio. Ponadto zakłada, że czytelnik tworzył już aplikacje łączące się z bazą danych i posiadające interaktywny interfejs użytkownika.

Cały kod przykładowy zapewniony w tej książce został utworzony przy pomocy języka Visual C# dostępnego w .NET Framework 4. Trzeba dobrze rozumieć język C#, aby śledzić i wykorzystywać ten kod. Książka szeroko wykorzystuje zarówno WPF, jak i Silverlight, więc należy mieć co najmniej podstawową wiedzę na temat tych dwóch technologii (a także solidne podstawy języka znaczników XAML – książka ta wykorzystuje w przykładach kod XAML).

Organizacja tej książki

Ta książka została opracowana w taki sposób, że każdy rozdział skupia się na określonym zagadnieniu. Pierwszy rozdział „Wprowadzenie do aplikacji biznesowych i wzorca Model View ViewModel” jest ogólnym wprowadzeniem do aplikacji biznesowych, ich składników i ich struktury. Rozdział 2 „Wzorce projektowe” pokazuje pełny przegląd wszystkich dobrze znanych wzorców projektowych i wzorców architektonicznych używanych do programowania aplikacji biznesowych, a zwłaszcza do projektowania luźno powiązanych składników. Rozdział 3 „Model domenowy” jest wprowadzeniem do modelu domenowego i projektowania sterowanego domeną (DDD – Domain-Driven Design). Ilustruje, jak osiągać cele projektowe DDD i jak unikać powszechnych błędów, które zwykle występują przy budowaniu aplikacji DDD. Rozdział 4 „Warstwa dostępu do danych” koncentruje się na warstwie dostępu do danych (DAL – Data Access Layer) i sposobach jej budowania przy użyciu systemu O/RM, takiego jak Entity Framework i/lub NHibernate. Rozdział 5 „Warstwa biznesowa” skupia swoją uwagę na projektowaniu i budowie warstwy logiki biznesowej (BLL – Business Logic Layer) w tym dogłębnie omawia reguły biznesowe, silniki reguł biznesowych i projektowanie architektury zorientowanej na usługi (SOA – Service-Oriented Architecture). Wreszcie rozdział 6, „Warstwa interfejsu użytkownika w MVVM” omawia dogłębnie MVVM, natomiast rozdział 7 „Platformy i zestawy narzędzi MVVM” wymienia dostępne platformy programowe i narzędzia, które mogą być przydatne podczas budowania aplikacji biznesowych przy pomocy MVVM.

Od czego najlepiej zacząć czytanie tej książki

Rozdziały tej książki omawiają różne aspekty budowania aplikacji biznesowej. Poza pierwszymi dwoma rozdziałami, które stanowią ogólny przegląd technik używanych w tej książce, każdy rozdział skupia się na określonej warstwie aplikacji biznesowej.

Poniższa tabela może pomóc w ustaleniu, gdzie najlepiej sięgnąć, jeśli ktoś planuje skupić się tylko na określonej warstwie.

Jeśli ktoś	To powinien:
Jest nowicjuszem w programowaniu aplikacji biznesowych i aplikacji wielowarstwowych	Przeczytać całą książkę i poeksperymentować z rozwiązaniami użytymi jako przykłady w każdym rozdziale.
Jest zaznajomiony z wzorcami projektowymi i architekturami oprogramowania, ale ich jeszcze dobrze nie opanował	Przejrzeć rozdziały 1 i 2, aby przypomnieć sobie podstawowe pojęcia. Następnie uważnie przeczytać pozostałe rozdziały stosując zasady napotkane w każdym rozdziale w swoich codziennych zadaniach.
Jest zainteresowany <i>tylko</i> określoną warstwą, taką jak DAL lub BLL	Uważnie przeczytać określony rozdział, który omawia daną warstwę będącą przedmiotem zainteresowania. Aby jednak ustalić kontekst, należy też przejrzeć pozostałe rozdziały.
Jest zainteresowany <i>tylko</i> wzorcem MVVM	Przeczytać rozdziały 1 i 2, aby utrwalić wiedzę na temat wzorców projektowych i wzorców prezentacyjnych, a następnie uważnie przeczytać rozdziały 6 i 7.

Konwencje i zasady w tej książce

Ta książka przedstawia informacje korzystając z konwencji zaprojektowanych tak, aby informacje te były czytelne i łatwe w odbiorze.

W większości przypadków książka ta zawiera osobne ćwiczenia dla programistów Visual Basic i programistów Visual C#. Można pominąć ćwiczenia, które nie dotyczą wybranego języka.

- Elementy w ramach z etykietami, takimi jak „Uwaga”, zapewniają dodatkowe informacje lub alternatywne metody wykonania danego kroku.
- Tekst, który należy wpisać (oprócz bloków kodu), wyróżniony jest pogrubieniem.
- Znak plus (+) pomiędzy nazwami dwóch klawiszy oznacza, że trzeba nacisnąć te klawisze jednocześnie. Na przykład „Alt+Tab” oznacza, że trzeba przytrzymać wciśnięty klawisz Alt podczas naciskania klawisza Tab.
- Pionowa kreska pomiędzy dwoma lub więcej elementami menu (na przykład File | Close) oznacza, że należy wybrać pierwsze menu lub element menu, potem następne, i tak dalej.

Wymagania systemowe

Do pracy z kodem i przykładami z tej książki potrzebny będzie następujący sprzęt i oprogramowanie:

- Dowolny z następujących systemów operacyjnych: Windows XP z Service Pack 3 (oprócz Starter Edition), Windows Vista z Service Pack 2 (oprócz Starter Edition), Windows 7, Windows Server 2003 z Service Pack 2, Windows Server 2003 R2, Windows Server 2008 z Service Pack 2 lub Windows Server 2008 R2.
- Visual Studio 2010, dowolne wydanie (kilka dodatkowych elementów do pobrania osobno może być wymaganych w przypadku korzystania z produktów Express Edition).
- SQL Server 2008 Express Edition lub wyższa wersja (wydanie 2008 lub R2) z narzędziem SQL Server Management Studio 2008 Express lub wyższą wersją (zawarte w Visual Studio, wydania Express Edition wymagają oddzielnego pobrania).
- Procesor o prędkości 1,6 GHz lub większej (zalecane 2 GHz).
- 1 GB (system 32-bitowy) lub 2 GB (system 64-bitowy) pamięci RAM (należy dodać 512 MB w przypadku uruchamiania w maszynie wirtualnej lub w przypadku wydań SQL Server Express Edition; więcej w przypadku bardziej zaawansowanych wydań SQL Server).
- 3,5 GB dostępnego miejsca na dysku twardym.
- Napęd dysku twardego o prędkości 5400 RPM.
- Karta grafiki zgodna z DirectX 9 działająca z rozdzielczością ekranu 1024 × 768 lub wyższą.
- Napęd DVD-ROM (w przypadku instalowania Visual Studio z płyty DVD).
- Połączenie internetowe do pobierania oprogramowania lub przykładów kodu.

W zależności od konfiguracji Windows, konieczne mogą być uprawnienia administratora lokalnego do zainstalowania lub skonfigurowania produktów Visual Studio 2010 i SQL Server 2008.

Przykłady kodu

Większość rozdziałów w tej książce zawiera ćwiczenia, które pozwalają w interaktywny sposób wypróbować nowy materiał poznany w głównym tekście. Wszystkie projekty przykładowe w postaci przed wykonaniem i po wykonaniu ćwiczenia są dostępne do pobrania ze strony tej książki na witrynie wydawnictwa O'Reilly Media:

<http://oreilly.com/catalog/9780735650923/>

Wystarczy kliknąć łącze Examples na tej stronie. Gdy pojawi się lista plików, należy zlokalizować i pobrać plik MvvmCrm.zip.

UWAGA Aby skorzystać z przykładów kodu, w swoim systemie trzeba mieć zainstalowane programy Visual Studio 2010 i SQL Server 2008. Poniższe instrukcje wykorzystują SQL Server Management Studio 2008 do skonfigurowania przykładowej bazy danych używanej w przykładowych ćwiczeniach. Należy zainstalować najnowsze pakiety serwisowe dla każdego produktu, jeśli są dostępne.



Instalowanie przykładów kodu

Aby zainstalować przykłady kodu na swoim komputerze, należy:

1. Rozpakować plik `MvvmCrm.zip` pobrany ze strony <http://oreilly.com/catalog/9780735650923/>.
2. Przejrzeć wyświetlaną umowę licencyjną dla użytkownika końcowego. Zaakceptować warunki umowy, a następnie kliknąć Next.

UWAGA Jeśli umowa licencyjna się nie pojawi, to można uzyskać do niej dostęp z poziomu tej samej strony WWW, z której pobrano plik `MvvmCrm.zip`.



Korzystanie z przykładów kodu

Struktura rozwiązania Visual Studio dostarczanego z tą książką jest podzielona na sześć różnych projektów, w których każdy projekt składa się z pełnego kodu źródłowego dla związanego z nim rozdziału w książce. Cała aplikacja stanowi program CRM opracowany w WPF.

Podziękowania

Gdy ktoś jest jedynym autorem książki, to jest trwale związany z tym, co daje ona innym; w istocie jest to jeden z powodów, dla których wiele osób chce napisać książkę. Ale nawet wyłączny autor nie jest jedyną osobą odpowiedzialną za powstanie książki. Chciałbym podziękować wszystkim osobom, które pomogły mi w napisaniu i wydaniu tej książki, ponieważ bez nich pozostałaby ona jedynie pomysłem.

To moja pierwsza książka. Pisanie jej było dla mnie wspaniałą przygodą i mam nadzieję, że jest to początek czegoś nowego, do czego czuję się naturalnie predysponowany. Nie byłbym w stanie napisać tej książki bez ogromnej pomocy mojej wspaniałej żony Deborah. Pisanie książki wymaga czasu, a pracuję na pełny etat w firmie ubezpieczeniowej, więc kilka wolnych godzin spośród dni spędzonych na pisaniu książki i wyszukiwaniu dokumentacji (co zajęło pełne sześć miesięcy) zostało zabranych z naszego wspólnego czasu. Bez tak wspaniałej i wyrozumiałej żony prawdopodobnie nie byłbym w stanie poświęcić tego czasu. Wiele razy, gdy byłem bliski zrezygnowania z ukończenia tej książki – ze względu na skomplikowanie i ogrom informacji – stanowczo nakłaniała mnie do ukończenia tej pracy, jak doskonały menedżer projektu! Dziękuję, Debbie!

Chciałbym też podziękować Russellowi Jones'owi, mojemu redaktorowi i głównej osobie z wydawnictwa kontaktującej się ze mną w sprawie tej książki. Jest on jedyną osobą, która wierzyła we mnie od początku i zaangażowała się w przekonanie Microsoft Press do tego projektu. Zawsze będę mu za to wdzięczny. Pomógł mi też w ukończeniu tej pracy na czas i organizował cały projekt.

Na koniec chcę podziękować Davidowi Hillowi, który jest recenzentem technicznym tej książki i moim mentorem. David jest pracownikiem w zespole patterns & practices w firmie Microsoft. Jego nieocenione uwagi podczas pisania tej książki pomogły mi znacznie poprawić moje ogólne pojęcie na temat wzorców prezentacyjnych oraz poprawnie skonstruować architekturę aplikacji biznesowej. David jest elastyczny i skromny. Mam niezwykle szczęście, że miałem okazję z nim pracować i mam nadzieję na kolejną współpracę w przyszłości.

Dziękuję Wam wszystkim!

Errata i wsparcie dla tej książki

Podjęliśmy wszelkie starania w celu zapewnienia poprawności tej książki i dołączonej do niej zawartości. Jeśli ktoś znajdzie błąd, prosimy o zgłoszenie go na naszej witrynie Microsoft Press w serwisie oreilly.com:

1. Przejdź do <http://microsoftpress.oreilly.com>.
2. W polu Search wpisz numer ISBN lub tytuł książki.
3. Wybierz książkę z wyników wyszukiwania.
4. Na stronie katalogowej książki, pod rysunkiem okładki pojawi się lista łączy.
5. Kliknij View/Submit Errata.

Dodatkowe informacje i usługi związane z książką można znaleźć na jej stronie katalogowej. Jeśli potrzebne jest dodatkowe wsparcie, wystarczy wysłać e-maila do Microsoft Press Book Support pod adres msspinput@microsoft.com.

Należy zwrócić uwagę, że wsparcie techniczne dla produktów nie jest oferowane pod powyższymi adresami.

Chcemy poznać opinie Czytelników

W wydawnictwie Microsoft Press zadowolenie Czytelnika jest naszym głównym priorytetem, a informacja zwrotna jest cennym zasobem. Swoje opinie na temat tej książki można zostawiać pod adresem:

<http://www.microsoft.com/learning/booksurvey>

Ankieta jest krótka i przeczytamy każdy zgłoszony komentarz i pomysł. Z góry dziękujemy za wszelkie uwagi!

Kontakt

Możemy pozostać w kontakcie! Jesteśmy dostępni w serwisie Twitter pod adresem <http://twitter.com/MicrosoftPress>.

ROZDZIAŁ 1

Wprowadzenie do aplikacji biznesowych i wzorca Model View ViewModel

Po zakończeniu tego rozdziału będziemy w stanie:

- Zidentyfikować aplikację biznesową.
- Wybrać odpowiednią technologię do utworzenia aplikacji biznesowej.

Wzorzec Model View ViewModel

Wzorzec Model View ViewModel (MVVM) został przedstawiony przez Johna Gossmana (architekta oprogramowania w firmie Microsoft w dziedzinie technologii Windows Presentation Foundation i Silverlight) na jego blogu w 2005 roku. MVVM jest wyspecjalizowaną odmianą wzorca Presentation Model (PM), który został przedstawiony w roku 2004 przez Martina Fowlera.

Jednym z głównych celów wzorca PM jest oddzielenie abstrakcyjnego widoku (View) – widocznego interfejsu użytkownika – od logiki prezentacyjnej, aby łatwiej było testować interfejs użytkownika. Dodatkowymi celami może być umożliwienie ponownego wykorzystywania logiki prezentacyjnej w różnych interfejsach użytkownika i różnych technologiach interfejsów użytkownika, co jest ograniczane przez powiązania pomiędzy interfejsem użytkownika a innym kodem oraz pozwala projektantom interfejsów użytkownika na pracę w bardziej niezależny sposób. MVVM jest wyspecjalizowaną interpretacją wzorca PM zaprojektowaną pod kątem wymagań Windows Presentation Foundation (WPF) i Silverlight.

Strukturalnie aplikacja MVVM składa się przede wszystkim z trzech głównych składników: modelu (*Model*), widoku (*View*) i modelu widoku (*ViewModel*).

- Model jest elementem, który reprezentuje jakieś pojęcie biznesowe; może to być cokolwiek od prostego przedstawienia klienta do skomplikowanego opisu handlu na giełdzie.
- Widok jest graficzną kontrolką lub zbiorem kontrolki odpowiedzialnych za przedstawienie danych modelu na ekranie. Widok może być oknem WPF, stroną Silverlight lub po prostu kontrolką szablonu danych w XAML.

- Model widoku odpowiada za „magię” działającą w tle. Model widoku zawiera logikę interfejsu użytkownika, polecenia, zdarzenia i odwołania do modelu. W MVVM model widoku nie odpowiada za aktualizowanie danych wyświetlanych w interfejsie użytkownika – dzięki świetnemu silnikowi wiązania danych zapewnianemu przez WPF i Silverlight model widoku nie musi tego robić. Dzieje się tak dlatego, że widok obserwuje model widoku, więc jak tylko model widoku ulega zmianie, to interfejs użytkownika się aktualizuje. Aby to było możliwe, model widoku musi implementować interfejs *INotifyPropertyChanged* i wyzwać zdarzenie *PropertyChanged*.

Pierwotnie tylko technologia WPF była wystarczająco zaawansowana, żeby spełniać wymagania wzorca MVVM. W wersji Silverlight 2 mieliśmy opcję implementowania MVVM, ale było to trudniejsze niż implementowanie MVVM w WPF. Obecnie w wersji Silverlight 4 możemy stosować MVVM zarówno w WPF, jak i Silverlight w taki sam sposób korzystając z możliwości wiązania danych, poleceń, zachowań i szablonów danych.

Gdy stosujemy wzorzec MVVM, musimy specjalnie zadbać o model widoku. Ponieważ ma on tak dużo obowiązków, to łatwo jest tworzyć nieuporządkowane rozwiązania, w których będziemy ponownie pisać ten sam kod. Jednak przy zastosowaniu odpowiedniego podejścia wzorzec MVVM może nam zaoszczędzić czas i pomóc w utworzeniu interfejsu użytkownika, który będzie łatwy do testowania i utrzymania. Oczywiście w celu prawidłowego wykorzystania MVVM trzeba koniecznie opanować język XAML i jego strukturę przy budowaniu interfejsu użytkownika. Musimy też wiedzieć, jak działa silnik wiązania XAML, a także jaką strukturę mają obiekty i zachowania poleceń (*ICommand*) oraz szablony danych. Do skutecznego wykorzystania MVVM zarówno w WPF, jak i Silverlight, trzeba też znać różnice pomiędzy WPF a Silverlight.

Ta książka dogłębnie analizuje każdy składnik wzorca MVVM. Na końcu utworzymy prostą aplikację biznesową stosującą MVVM, która będzie mogła być wykorzystywana jako szablon dla dowolnych przyszłych aplikacji MVVM. Jednocześnie zbudujemy niewielką platformę narzędziową MVVM, która będzie działać jako automatyczny składnik do wykorzystania w aplikacjach WPF lub Silverlight upraszczający pisanie aplikacji MVVM. Platforma ta zapewni na przykład podstawową klasę *View-Model*, przykładowego brokera komunikatów i inne funkcje wymagane w typowej aplikacji MVVM.

Aplikacje biznesowe

Z mojego doświadczenia najlepszym sposobem nauczania się nowej technologii jest zbudowanie aplikacji krok po kroku. Aplikacja biznesowa stanowi najlepszy przykład z kilku powodów: jest odpowiednia dla elastycznej technologii interfejsu użytkownika występującej zarówno w WPF, jak i Silverlight; jest otwarta na zastosowanie wzorca MVVM; i jest to typowy rodzaj aplikacji, więc możemy ponownie wykorzystać te przykłady później dla prawdziwych celów biznesowych.

UWAGA Aplikacje biznesowe obsługują działania istotne dla przedsiębiorstwa takie, jak księgowość, zarządzanie łańcuchem dostaw lub planowanie zasobów. Aplikacje biznesowe są zwykle dużymi programami, które zawierają wiele zintegrowanych funkcjonalności i wiążą się z innymi aplikacjami oraz systemami zarządzania baz danych. Są też często nazywane aplikacjami dla przedsiębiorstw.



Aplikacja biznesowa może być dowolną aplikacją istotną dla prowadzenia biznesu: systemem zarządzania relacjami z klientami używanym w biurze, oprogramowaniem księgowym używanym przez departament finansowy do przygotowywania listy płac lub dowolnym innym typem aplikacji biznesowej, która spełnia określone wytyczne i ma określony styl interfejsu użytkownika. Jeśli by się nad tym zastanowić, to takie aplikacje doskonale pasują do pojęcia „szablonu”.

Aplikacje biznesowe są najczęściej zamawianymi przez klientów, a przy tym najłatwiejszymi do zaprogramowania. Jednocześnie bywają najtrudniejszymi do zaprojektowania. Wynika to z tego, że chociaż ich *struktura* jest zwykle dość prosta i powtarzalna, to ich *wymagania* często zmieniają się podczas procesu tworzenia oraz w okresie ich użytkowania.

Coraz częściej aplikacje biznesowe wykorzystują interfejsy WWW, co sprawia, że stają się łatwiej dostępne poprzez przeglądarki, łatwiejsze we wdrażaniu i aktualizowaniu oraz pozwalają na realizację scenariuszy biznesowych, które wymagają dostępu do tych samych funkcji przez partnerów biznesowych i klientów. Wykorzystują też osobiste funkcje aplikacyjne, takie jak poczta elektroniczna i książki adresowe.

Aplikacja biznesowa często zmienia się w sposób przyrostowy podczas projektowania. Książka na temat zarządzania projektami, którą jakiś czas temu przeczytałem (dzięki swojemu szefowi), wspominała, że największe wydatki departamentów IT i producentów oprogramowania wiążą się z *utrzymywaniem* istniejącego oprogramowania. Zwykle osoby zaangażowane w projekt programistyczny dowolnego rodzaju uważają, że najdroższą częścią jest faza *programowania* prowadząca do pierwszej edycji programu, ale tak naprawdę dopiero po wydaniu pierwszej wersji pojawiają się prawdziwe kłopoty. Na przykład założmy, że tworzymy i sprzedajemy aplikację księgową, w której pierwotnie nie zaprojektowano obsługi wypłat dla pracowników. Po jakimś czasie klienci poprosili o tę nową „funkcję”. Jeśli projekt nie jest wystarczająco

elastyczny na przyjmowanie nowych elementów i zmian, to prawdopodobnie stracimy klienta i aplikacja okaże się porażką.

Aplikacja biznesowa świetnie pasuje do technologii WPF/Silverlight i wzorca MVVM, ponieważ skupia się na wszystkich typowych problemach, które mały, średni lub duży zespół napotka podczas różnych faz procesu tworzenia programu, a które można rozwiązać korzystając z tych elastycznych technologii. Niestety książka nie nauczy nas wszystkiego, więc w tej książce nie dowiemy się, jak budować przemysłowe aplikacje CRM albo jak stosować metodologię Scrum w swoim zespole – ale dowiemy się, jak zbudować aplikację biznesową, która implementuje niewielki system do zarządzania klientami korzystając z najnowszych technologii Microsoft.

Wybór odpowiedniej technologii

Ponieważ możemy zbudować aplikację biznesową albo przy pomocy WPF, albo Silverlight, to musimy przeanalizować wymagania projektu, aby ustalić, która technologia jest najbardziej odpowiednia dla tej określonej aplikacji i z których narzędzi chcemy skorzystać do jej zbudowania. Aby odpowiedzieć na te pytania, najpierw zbadamy, jak wybrać technologię pomiędzy Silverlight a WPF, a następnie zbadamy narzędzia, które firma Microsoft obecnie oferuje do projektowania interfejsu użytkownika. Na koniec przejdziemy do analizy typowego układu graficznego aplikacji biznesowej i oczekiwań użytkowników wobec niego.

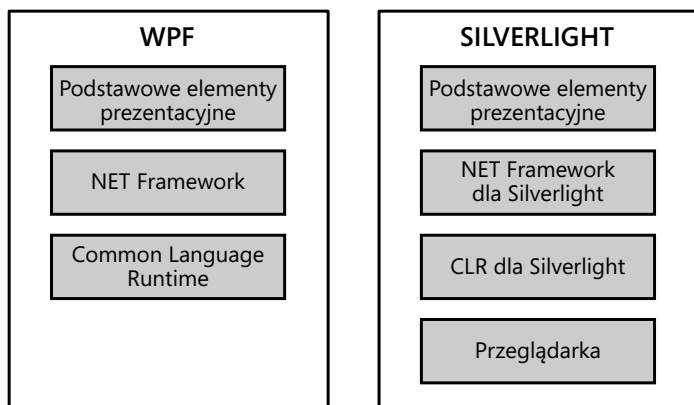
Silverlight czy WPF?

Silverlight i WPF opierają się na tej samej podstawowej technologii: Microsoft .NET Framework. W obu przypadkach budujemy interfejsy użytkownika przy pomocy języka XAML.

Technologia WPF jest następcą Windows Forms, więc została zaprojektowana tak, aby obejmowała pełen zestaw kontrolek interfejsu użytkownika i elementów multimedialnych, przy pomocy których możemy tworzyć bogate i interaktywne aplikacje klienckie dla systemu Windows. Silverlight jest międzyprzeglądarkową, międzyplatformową technologią, która obsługuje bogate aplikacje internetowe. Zasadniczo WPF służy do budowania aplikacji klienckich dla systemu Windows, a Silverlight służy do tworzenia bogatych aplikacji WWW, ale również przeglądarki mogą obsługiwać aplikacje WPF, a aplikacje Silverlight mogą być uruchamiane poza przeglądarką na pulpitach.

Istnieje kompatybilność pomiędzy Silverlight a WPF, ponieważ obie te technologie wykorzystują ten sam język opisu interfejsu użytkownika (XAML), ten sam zestaw składników interfejsu (choć Silverlight wykorzystuje tylko podzbiór tego zestawu), tę samą bazową bibliotekę klas .NET oraz środowisko CLR. Jedyną znaczącą różnicą jest tutaj to, że Silverlight obecnie wykorzystuje inną implementację .NET CLR.

Rysunek 1-1 wyświetla główne różnice pomiędzy tymi dwiema technologiami.



RYSUNEK 1-1 Przegląd architektury WPF i Silverlight

Ponieważ Silverlight skupia się na odbiorcach korzystających z wielu platform i przeglądarek, to firma Microsoft zmuszona była do zmniejszenia i ograniczenia środowiska uruchomieniowego dla tej technologii. Wniosek z tego jest taki, że najlepiej od początku planować aplikację pod kątem ostatecznej grupy docelowej dla naszej aplikacji biznesowej, ponieważ nie wszystkie funkcje WPF będą dostępne w Silverlight i znacznie trudniej jest przejść później z jednej technologii na drugą.

Oczywiście technologie WPF i Silverlight stają się coraz lepsze z każdym wydaniem, więc istnieje nadzieja, że w przyszłości otrzymamy zunifikowaną platformę, ale na razie ważne jest, aby pamiętać, że grupy docelowe dla tych dwóch technologii są nieco inne.

UWAGA Firma Wintellect we współpracy z firmą Microsoft wydała dokument dostępny pod adresem <http://wpfslguidance.codeplex.com>, który w pełni wyjaśnia różnice pomiędzy tymi dwiema technologiami. Dokument ten ma około 69 stron. Jak można się spodziewać, ta książka nie może obejmować wszystkich tych różnic; dlatego jedynie podkreśla najważniejsze z nich.



Pierwsza różnica dotyczy technologii istotnych dla implementacji MVVM. Silverlight nie implementuje wytyczanych poleceń, wyzwalaczy albo szablonów danych w taki sam sposób jak WPF. Dlatego, aby uzyskać takie samo (lub podobne) zachowanie, musimy implementować niestandardową funkcjonalność w Silverlight. Najpierw jednak słowo ostrzeżenia dotyczące wykorzystania wyzwalaczy w WPF i Silverlight przy implementowaniu wzorca MVVM: nie powinny być intensywnie używane, ponieważ łatwo może się zdarzyć, że będą zawierać logikę prezentacyjną, której nie będzie się dało testować. Logika ta nie jest dostępna w modelu widoku, ale jest udostępniana w widoku przez wyzwalacz.

Silverlight 4 zawiera bogaty zestaw kontrolki, stylów i szablonów, z których jednym jest ciekawy szablon aplikacji biznesowej. Z kolei WPF zawiera mniejszy zestaw narzędzi z kontrolkami.

Z której technologii powinniśmy więc skorzystać – Silverlight, czy WPF? Odpowiedź brzmi: należy dokonać wyboru w oparciu o typ budowanej aplikacji i najbardziej typową grupę docelową dla tej aplikacji. Jeśli na przykład zamierzamy stworzyć aplikację biznesową dla departamentu finansowego, która nie będzie używana poza firmą klienta, to WPF jest odpowiednią technologią. Z drugiej strony, jeśli mamy opracować aplikację CRM, która będzie używana przez klientów i menedżerów korzystających z różnych urządzeń, to lepiej umieścić aplikację w przeglądarce, a więc Silverlight byłby odpowiednią technologią.

Możemy łatwo zbudować dwie warstwy interfejsu użytkownika, jeśli prawidłowo korzystamy z wzorca MVVM: jedną warstwę dla WPF i jedną warstwę dla Silverlight. Obecnie wielu programistów stosuje to podejście z dwoma warstwami interfejsu użytkownika.

Ostateczna grupa docelowa i zadania naszej aplikacji stanowią klucze, które powinny wpływać na wybór zastosowanej technologii. Nie musimy się martwić w tym momencie różnicami w zestawie kontrolki lub interfejsie użytkownika; firma Microsoft wydała zestaw narzędzi projektowych (Microsoft Expression Studio), które mogą obsługiwać cały proces projektowania zarówno na potrzeby WPF, jak i Silverlight.

Narzędzia firmy Microsoft do budowania interfejsu użytkownika

Największym problemem dla programistów, którzy chcą przejść na WPF lub Silverlight jest krzywa uczenia się. Obie te technologie wykorzystują nową specyfikację języka opisu interfejsu użytkownika o nazwie XAML, który jest po prostu deklaratywnym językiem znaczników, podobnie jak HTML lub XML. Oczywiście nie jest łatwo korzystać z tego języka do budowy układów graficznych, gdy nie wiemy, jak działa silnik przetwarzający XAML. Podobnie nie jest łatwo zaimplementować pełne wsparcie narzędzi projektowych dla podejścia WYSIWYG. XAML jest bardzo elastycznym językiem znaczników z kilkoma ograniczeniami. Na przykład możemy umieścić kontrolkę *DataGrid* w przycisku typu *Button* – nawet jeśli nie miałoby to sensu z punktu widzenia użyteczności. Taka elastyczność może doprowadzać silniki graficzne do szału.

Żeby pomóc w rozwiązywaniu takich problemów, firma Microsoft wydała pakiet narzędzi graficznych o nazwie Expression Studio. Najnowsza wersja to Expression Studio 4, którą trzeba kupić oddzielnie (można również kupować pojedynczo każde z narzędzi dostępnych w pakiecie Expression). Ten pełen zestaw aplikacji dla projektantów WPF/Silverlight obsługuje cały proces projektowania aplikacji XAML, od początkowego symulowania interfejsu użytkownika po wszystkie elementy projektowe zawarte w finalnym produkcie. Niektóre z narzędzi w pakiecie Expression Studio, takie jak Expression Web, są przeznaczone specjalnie dla projektantów WWW. Program Expression Blend przeznaczony dla projektantów interfejsu użytkownika oddziela nie tylko kod proceduralny od znaczników, ale też oddziela zadania projektowe od zadań

programistycznych pozwalając programistom skupić się na pisaniu kodu biznesowego, a projektantom projektować funkcjonalny interfejs użytkownika bez konieczności znajomości C#, Visual Basic lub dowolnego innego języka .NET. MVVM jest kluczem do tego procesu współpracy pomiędzy projektantem a programistą. W istocie program Expression Blend jest dostarczany z określoną przestrzenią nazw, którą programiści mogą wykorzystać do utworzenia atrapy modelu widoku na potrzeby projektantów. Projektanci następnie mogą wiązać widok z tym odbiciem ostatecznego modelu widoku i projektować warstwę interfejsu użytkownika.

Pakiet Expression Studio w 60-dniowej wersji próbnej można pobrać pod adresem <http://www.microsoft.com/expression/>, kupić go przez sieć lub uzyskać poprzez subskrypcję MSDN.

Expression Blend

Expression Blend jest głównym produktem pakietu Expression Suite dla projektanta WPF/Silverlight. Pliki projektowe tego programu są w pełni kompatybilne z Microsoft Visual Studio. Możemy pracować nad projektem w programie Expression Blend, a następnie otwierać ten projekt w Visual Studio i vice versa. Ta dwukierunkowa kompatybilność ułatwia wykorzystanie programu Expression Blend do zaprojektowania szablonu i kontrolek naszej aplikacji biznesowej, a następnie przejście do Visual Studio w celu napisania kodu .NET. Mimo tej wygody, przechodzenie tam i z powrotem pomiędzy Expression Blend a Visual Studio nie jest obowiązkowe, ponieważ program Expression Blend może przetwarzać XAML oraz budować rozwiązania w językach C# i Visual Basic tak samo jak Visual Studio.

Korzystając z Expression Blend możemy projektować interfejs użytkownika w XAML, tworzyć bibliotekę kontrolek dla Silverlight lub WPF albo po prostu projektować i stosować niestandardowe style w naszej aplikacji XAML. Jedną z naprawdę mocnych funkcji Expression Blend jest możliwość tworzenia szablonów danych w czasie projektowania. Ta możliwość oznacza, że projektant graficzny nie potrzebuje „prawdziwej” bazy danych lub plików z danymi, żeby przedstawiać realistyczne wyniki w programie projektowym; Expression Blend pozwala nam łatwo skonfigurować szablon danych lub może wygenerować taki szablon. Końcowy wynik pojawia się w zintegrowanym środowisku projektowym i wygląda tak, jak wyniki, które moglibyśmy uzyskać przy użyciu danych rzeczywistych.

Najnowsza wersja Expression Blend 4 ma pełną obsługę WPF i Silverlight w czasie projektowania i znacznie ułatwia pracę projektanta. Ponadto Expression Blend udostępnia pakiet Behaviors SDK, który dodaje obsługę wzorca MVVM w czasie projektowania. Funkcja ta sprawia, że Expression Blend jest podstawowym narzędziem projektanta interfejsów użytkownika dla aplikacji wykorzystujących MVVM.

Na koniec, aby wspomnieć kilka nowych funkcji w najnowszej wersji Expression Blend, możemy łatwo budować i emulować aplikacje dla nowej platformy mobilnej Windows Phone 7; tworzyć wspaniałe przejścia i animacje dla swoich aplikacji

Silverlight lub WPF; albo tworzyć, animować i wdrażać dynamiczne symulacje interfejsów użytkownika.

Rysunek 1-2 pokazuje wypełniony szablon danych w programie Expression Blend.

The screenshot shows a window titled "Add Contact" with a close button (X) in the top right corner. The window contains two columns of input fields. The left column includes fields for Name, Email, Phone, Fax, Cell, Address, State (a dropdown menu), and ZIP. The right column includes fields for ID, Twitter, Website, and Blog. Each field is populated with example data. At the bottom right of the window are "OK" and "Cancel" buttons.

Field	Value
Name	Aaberg, Jesper
ID	Auctor
Email	someone@example.com
Twitter	http://www.adventure-works.com/
Phone	(333) 555-0102
Website	http://www.adventure-works.com/
Fax	(222) 555-0101
Blog	http://www.adatum.com/
Cell	(111) 555-0100
Address	4567 Main St., Buffalo, NY 98052
State	AL
ZIP	4567 Main St., Buffalo, NY 98052

RYСУNEK 1-2 Szablon danych w Microsoft Expression Blend

Rysunek 1-2 wykorzystuje prosty szablon danych do wyświetlania stanu modelu widoku i jego danych w czasie projektowania. Kod służący do tego celu jest dość prosty. Wykorzystuje wiązanie danych w celu mapowania wartości pomiędzy interfejsem użytkownika a modelem widoku, jak pokazano w następującym przykładzie:

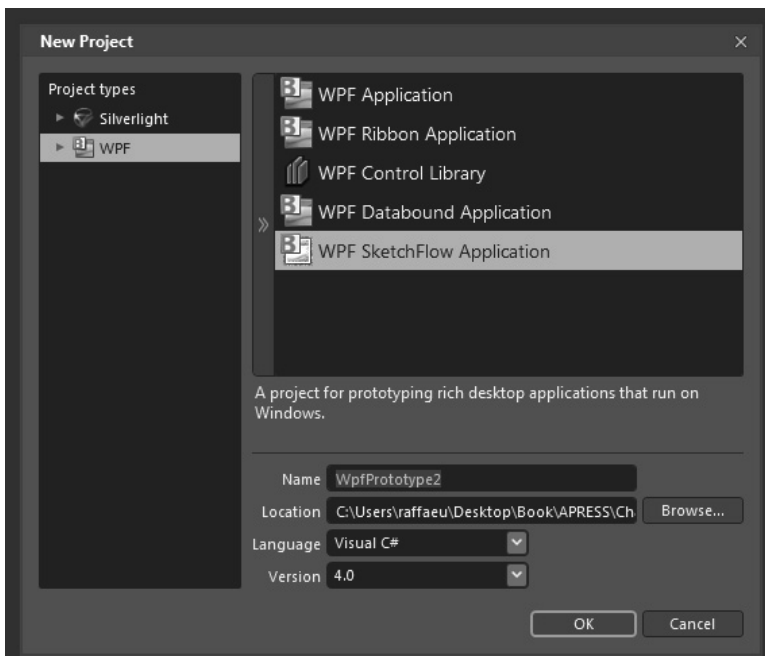
```
<Grid x:Name="LayoutRoot" d:DataContext="{d:DesignData/SampleData/
ContactsViewModelSampleData.xaml}">
  <!--
    pominięty fragment kodu
  -->
  <data:DataGridTextColumn Binding="{Binding Name}" Header="NAME" Width="0.25*" />
  <data:DataGridTextColumn Binding="{Binding Email}" Header="EMAIL" Width="0.35*" />
  <!--
    pominięty fragment kodu
  -->
  <i:InvokeCommandAction Command="{Binding AddContactCommand}" />
</Grid>
```

Microsoft SketchFlow

Microsoft SketchFlow jest funkcją szkicowania interfejsu użytkownika, która dostępna jest w programie Expression Blend. SketchFlow pozwala nam szybko zaprojektować szkic interfejsu użytkownika i dodać pewne minimalne interakcje pomiędzy jego elementami.

Jednym z krytycznych kroków w dostarczaniu nowej aplikacji jest uzyskanie informacji zwrotnych od klienta możliwie jak najwcześniej – nawet zanim rozpocznie się proces programowania interfejsu użytkownika. Korzystając z narzędzia SketchFlow możemy zapewnić szybki szkic naszej aplikacji, a następnie przekazać go użytkownikom końcowym, żeby mogli go przejrzeć i zgłosić uwagi dotyczące ewentualnych modyfikacji. Szkice utworzone przez SketchFlow można publikować w module odtwarzającym dla Silverlight lub WPF, gdzie użytkownicy mogą z nich interaktywnie korzystać, dodawać uwagi i rysunki zapewniające informacje zwrotne. Korzystając z narzędzia SketchFlow do obsługi wczesnych testów interfejsów użytkownika nie musimy projektować pełnego interfejsu przed zebraniem informacji zwrotnych od klientów.

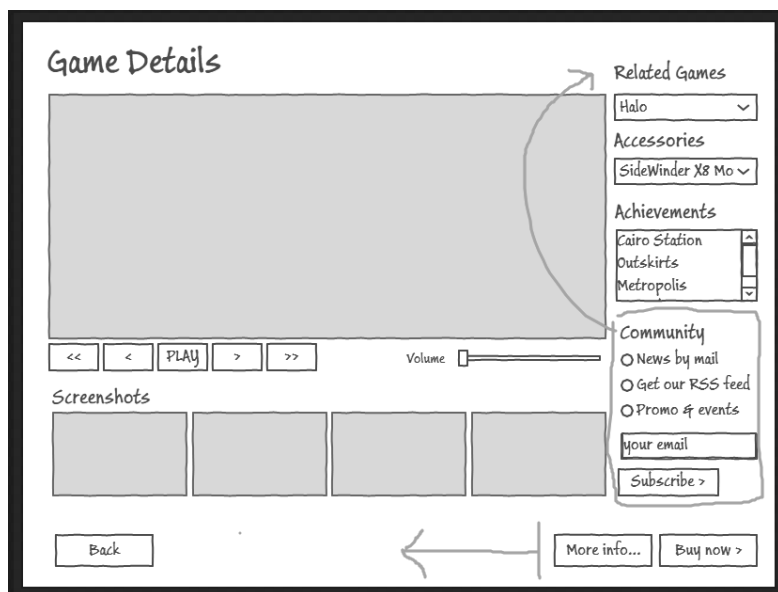
Narzędzie SketchFlow jest w pełni zintegrowane z programem Expression Blend. Jest dostarczane z niestandardowym zestawem kontrolki, które są po prostu klasycznymi kontrolkami XAML opatrzonymi niestandardowym tematem graficznym. Rysunek 1-3 pokazuje główne okno Microsoft SketchFlow.



RYSUNEK 1-3 Strona startowa Microsoft SketchFlow

Korzystając z narzędzia SketchFlow w powiązaniu z Expression Blend możemy wiązać przykładowe dane z symulowanym modelem widoku tak, aby przy stosowaniu wzorca MVVM w tym procesie projekt interfejsu użytkownika mógł być rozwijany niezależnie od logiki biznesowej (na przykład moglibyśmy korzystać z narzędzia SketchFlow w celu dostrajania projektu interfejsu użytkownika). Później możemy odłączyć symulowany model widoku.

Rysunek 1-4 wyświetla końcowy przykład szkicu zbudowanego przy użyciu SketchFlow.



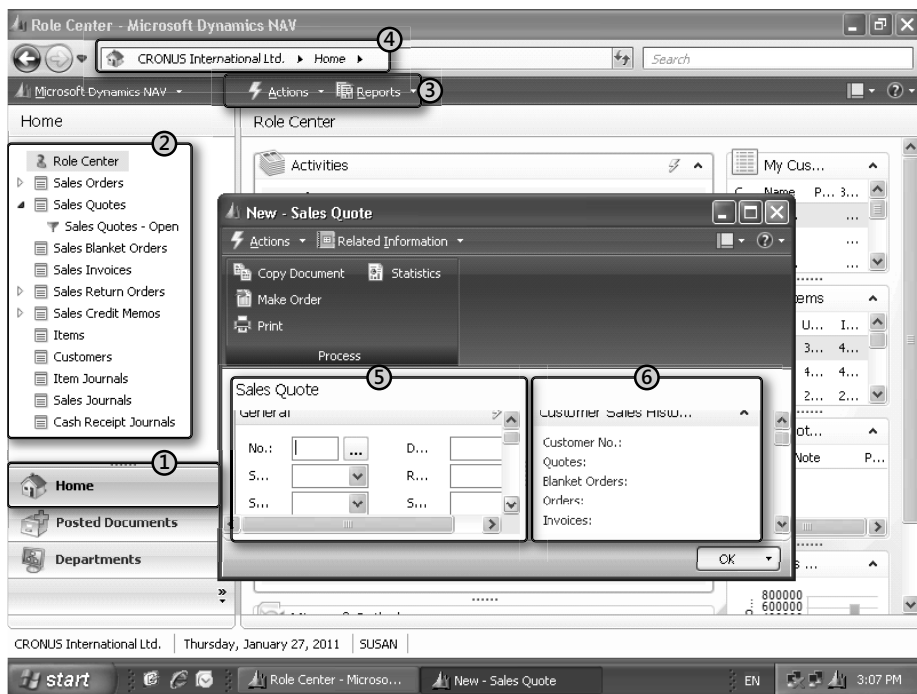
RYSUNEK 1-4 Ostateczny szkic utworzony przy użyciu SketchFlow.

Budowanie interfejsu użytkownika

W dużych firmach programistycznych dzięki korzystaniu z produktów Expression Suite i Visual Studio możemy łatwo podzielić naszą aplikację pomiędzy dwa zespoły bez wpływu na efektywność; jeden zespół będzie korzystał z narzędzi Expression do szkicowania i opracowywania interfejsu użytkownika, natomiast drugi zespół skupi się na implementowaniu podstawowych funkcji i aktywowaniu interfejsu użytkownika. MVVM umożliwia taki czysty podział, ponieważ daje nam możliwość luźnego połączenia interfejsu użytkownika z logiką biznesową zawartą w modelu widoku. Oczywiście to podejście nie oznacza, że musimy dzielić swój zespół na Projektantów i Programistów, jeśli planujemy zastosować wzorzec MVVM.

Jedną z fundamentalnych zasad, które musimy zrozumieć, aby budować udane aplikacje biznesowe lub dowolne inne aplikacje wykorzystujące interfejsy użytkownika, jest to, że *użytkownicy widzą tylko interfejs użytkownika*. Użytkownicy końcowi nie wiedzą (i nie obchodzi ich to), że aplikacja wykorzystuje najnowszą wersję SQL Server albo że przesłanie zamówienia wymaga interakcji z usługą WWW agencji NASA. Użytkownik komunikuje się tylko z interfejsem użytkownika. Dlatego ważne jest, aby skupić się na kilku elementach, które będą generować interfejs użytkownika i nie pozwolą użytkownikowi zgubić się w naszej aplikacji. Zanim zaczniemy definiować

każdą z części interfejsu użytkownika aplikacji biznesowej, warto przyjrzeć się bardzo udanemu przykładowi autorstwa firmy Microsoft, który reprezentuje klasyczną aplikację biznesową: aplikacji Microsoft Dynamics pokazanej na rysunku 1-5.



RYСУNEK 1-5 Dobrze znana aplikacja biznesowa – Microsoft Dynamics

Rysunek 1-5 zawiera kilka wyróżnionych i ponumerowanych elementów, które są omówione w kolejnych akapitach.

Bardzo typowe dla aplikacji biznesowej jest zastosowanie *panelu nawigacyjnego* (obszar 1 na rysunku 1-5). Panel nawigacyjny możemy łatwo zbudować przy pomocy kontrolki widoku z zakładkami dostępnej w XAML oraz zestawu stylów. Można rozpoznać ten panel nawigacyjny jako klasyczny układ nawigacji używany w programach, takich jak Microsoft Outlook. Celem panelu nawigacyjnego jest grupowanie lub zbieranie określonych funkcjonalności w jednym miejscu. Jak widać z rysunku 1-5, program Dynamics grupuje różne ważne części aplikacji pod wysoko-poziomowymi nagłówkami, takimi jak Finance, Inventory, itd. W ten sposób użytkownicy mogą łatwo uzyskać dostęp do dowolnej części aplikacji biznesowej. Gdy użytkownik kliknie jeden z wysoko-poziomowych nagłówków, to sekcja dla tego nagłówka jest ładowana do szczegółowej kontrolki widoku u góry panelu nawigacyjnego (obszar 2).

U góry aplikacji można zobaczyć *pasek narzędzi* (obszar 3) i *pasek menu* (obszar 4). Pasek menu znajduje się u góry aplikacji i powinien zapewniać dostęp do wszystkich istniejących poleceń. Pasek narzędzi zawiera graficzne skróty do najczęściej używanych

poleceń. Pasek narzędzi powinien reagować na zmiany kontekstu; na przykład powinien włączać przycisk Zapisz tylko wtedy, gdy aktualny widok uległ zmianie i wymaga zapisania.



UWAGA Elementy menu również powinny być kontekstowe, ale ponieważ pod elementy menu nie są widoczne podczas pracy użytkownika z interfejsem, to reakcje menu na zmiany kontekstu są mniej oczywiste, niż w przypadku pasków narzędzi.



UWAGA W wielu nowoczesnych aplikacjach, takich jak Office 2010, pasek narzędzi i pasek menu są zastępowane połączeniem obu tych elementów zwanym wstążką (ribbon). Więcej na temat wstążki dowiemy się w dalszej części tego rozdziału.

Widok bieżący (obszary 5 i 6) wyświetla dane, nad którymi użytkownik właśnie pracuje w aplikacji. W tym przypadku program Dynamics wykorzystuje interfejs z wieloma dokumentami (MDI), gdzie każdy otwarty widok ma osobne okno. Innym możliwym podejściem jest wykorzystanie kart dla każdego widoku; jest to na przykład domyślny styl stosowany przez Visual Studio.

Kolejne podrozdziały zapewniają dokładniejsze objaśnienia każdego z głównych obszarów i omawiają najlepsze praktyki związane z budowaniem przydatnych interfejsów użytkownika dla aplikacji biznesowych.

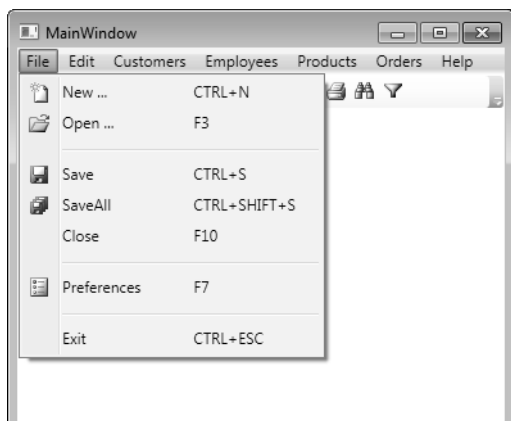
Pasek menu

Pasek menu jest wyświetlany u góry aplikacji. Jest to pojemnik, który powinien zawierać wszystkie dostępne w aplikacji polecenia podzielone na grupy. Pasek menu powinien zawierać również sekcje typowe dla wielu aplikacji, takie jak Plik (File), Edycja (Edit) i Pomoc (Help).

Pasek menu jest istotnym składnikiem aplikacji biznesowej, więc paski menu powinny stosować się do pewnych określonych zasad projektowych; inaczej stracą swój potencjał.

Niektóre typowe reguły dotyczące paska menu są następujące:

- Gdy to tylko możliwe, należy zastosować znak podkreślenia (_) w tekście każdego elementu. W środowisku .NET to podkreślenie definiuje klawisz dostępu do danego elementu. Dodanie podkreśleń pozwala użytkownikom przeglądać menu przy użyciu klawisza Alt w połączeniu z klawiszem dostępu.
- Należy respektować powszechnie stosowane jako skróty kombinacje klawiszy. Na przykład polecenie Save (Zapisz) w systemie Windows jest zwykle reprezentowane przez skrót klawiaturowy Ctrl+S.
- Należy dodawać ikony odpowiadające kontekstowi każdego polecenia. Obrazek powinien być wyraźny i zrozumiały, a jego rozmiar powinien wynosić 16 × 16 pikseli. Rysunek 1-6 pokazuje przykład menu z ikonami.



RYСУNEK 1-6 Przykładowy pasek menu odpowiadający standardowym zaleceniom.

Pasek narzędzi

Pasek narzędzi jest kontrolką graficzną, która zwykle jest umieszczana bezpośrednio pod paskiem menu. Typową cechą paska narzędzi (w przeciwieństwie do paska menu) jest to, że zapewnia użytkownikom wizualne kontrolki zamiast prostych etykiet dla poleceń. Zwykle pasek narzędzi zawiera zestaw przycisków (a czasami też innych kontrolkek), z których każdy ma wyraźny i rozpoznawalny obraz informujący o funkcji danego przycisku. Normalnym rozmiarem dla ikony paska narzędzi jest 22×22 pikseli lub co najwyżej 24×24 pikseli. Można też rozważyć użycie klasycznych obrazów 16-pikselowych, ale w mojej opinii są one zbyt małe dla normalnego paska narzędzi.

Można dołączać etykiety tekstowe do przycisków na pasku narzędzi, ale należy się starać tego unikać, o ile to możliwe. Zwykle użytkownicy widzą tylko obraz wskazujący funkcję przycisku paska narzędzi; im wyraźniejsze i bardziej zrozumiałe będą te obrazy, tym przydatniejszy będzie pasek narzędzi.

Ponieważ pasek narzędzi obsługuje te same polecenia, które są już dostępne w pasku menu, to przy stosowaniu podejścia MVVM możemy korzystać z tego samego modelu widoku dla obu kontrolkek albo zapewniać kolekcję poleceń dla każdego modelu widoku i budować szablony *DataTemplate* do ich przedstawiania na ekranie. W następnych rozdziałach zobaczymy, jak to robić. Moglibyśmy to teoretycznie osiągnąć przy pomocy kontrolki Ribbon dostępnej w WPF i Silverlight, ale złożoność interfejsu użytkownika i powiązanej z nim logiki będzie wymagać specyficznego modelu prezentacyjnego dedykowanego dla kontrolki Ribbon.

Etykiетки narzędzi (i ich nadużywanie)

Pomimo najlepszych intencji projektowych czasami okoliczności zmuszają nas do umieszczania zbyt wielu kontrolkek w jednym obszarze lub umieszczania obrazów i kontrolkek w taki sposób, że nawet najbardziej doświadczony użytkownik może się

łatwo pogubić. Gdy interfejs użytkownika nie jest całkiem jasny, powinniśmy zapewniać użytkownikom dynamiczną informację zwrotną z interfejsu użytkownika, żeby pomóc im w podjęciu jak najlepszych decyzji.

Jednym ze sposobów dla zapewnienia takiej informacji zwrotnej jest kontrolka Tooltip. Jest to niewielkie okienko, które pojawia się, gdy użytkownik zatrzymuje wskaźnik myszy nad określoną kontrolką. Celem kontrolki Tooltip jest zapewnienie natychmiastowego opisu wybranej kontrolki. Tylko tyle, ni mniej, ni więcej. Jeśli na przykład użytkownik umieści kursor myszy nad przyciskiem paska narzędzi z rysunkiem dyskiety, to zdrowy rozsądek podpowiada, że tekst wskazówki (kontrolki Tooltip) dla tego rysunku powinien brzmieć podobnie do: „Zapisz. Zapisz bieżący rekord”. Jest to jasny i zwięzły opis, ale nie jest zbyt skomplikowany.

Podczas gdy kontrolki Tooltip mogą być pomocne, to jest pewien problem, który często obserwuję przy pracy z programistami WPF/Silverlight; ponieważ XAML ma niewiele ograniczeń, łatwo jest dodać kontrolkę Tooltip do kontrolki Datagrid lub dowolnej innej kontrolki. Jednak nadużywanie kontrolek Tooltip mija się z celem. Kluczem jest pamiętać, że celem kontrolki Tooltip jest po prostu zapewnienie tymczasowej i chwilowej pomocy poprzez krótki opis bieżącej kontrolki. Jeśli trzeba rozszerzyć zachowanie tej kontrolki, należy rozważyć zastosowanie innego rozwiązania. Na przykład w WPF występuje ciekawy znacznik o nazwie *Popup*, który może zawierać dowolną kontrolkę i wyświetlać w niej dodatkowe informacje przy użyciu wyskakującego okienka.

Ze względu na swój szablon prezentacyjny kontrolki Tooltip mogą być w łatwy sposób stosowane nieprawidłowo. Planując napisanie wskazówki dla określonej kontrolki (pola tekstowego, przycisku, etykiety, itd.) trzeba pamiętać, że kontrolka Tooltip będzie wyświetlana na ekranie jedynie przez kilka sekund. Dlatego powinna zawierać tylko kilka słów opisujących związaną z nią kontrolkę. Jeśli uznamy, że Tooltip wymaga więcej niż kilku słów do opisania kontrolki, to znaczy, że mamy dwa problemy:

- Kontrolka, która wymaga wskazówki, jest umieszczona w złym widoku albo może być całkiem niejasna; warto rozważyć użycie etykiety opisującej tę kontrolkę, aby uprościć (lub wyeliminować) kontrolkę Tooltip.
- Działanie wykonywane przez tę kontrolkę jest zbyt skomplikowane, aby je łatwo opisać. W takich przypadkach konieczne może być rozwiązanie przedstawiające bardziej szczegółową pomoc albo można rozważyć podział operacji na kilka kontrolek z zastosowaniem kreatora lub ikony symbolizującej stan poprawności danych, żeby identyfikować potencjalne błędy powodowane przez tę operację.
- Tak jak w przypadku paska narzędzi i paska menu dobrym podejściem do tworzenia jednorodnych kontrolek Tooltip jest utworzenie określonego szablonu *DataTemplate* przechowywanego we wspólnym słowniku. Przy wykorzystaniu tego podejścia każda kontrolka w naszej aplikacji, która wykorzystuje kontrolkę Tooltip, może korzystać z tego samego stylu. Chcę tutaj podkreślić, że kontrolka Tooltip jest tylko obiektem interfejsu użytkownika, który może zostać wybrany

przez projektanta lub nie; nie definiuje żadnej logiki interfejsu użytkownika, więc nie musi być reprezentowana w modelu widoku.

Powiadomienia i alarmy

Komunikacja z użytkownikami jest niezwykle istotna w budowaniu interfejsu użytkownika. Jednym z często wymaganych zadań komunikacyjnych jest, gdy chcemy powiadomić użytkowników, że wprowadzone lub wybrane dane są nieprawidłowe albo że próbują przeprowadzić nieprawidłową operację. Jednocześnie chcemy, aby nasz interfejs użytkownika był jak najmniej inwazyjny i ograniczać zdenerwowanie użytkowników, o ile to możliwe. Jeśli na przykład za każdym razem, gdy użytkownik klika przycisk Zapisz, interfejs użytkownika będzie pytał o potwierdzenie – na przykład poprzez komunikat „Czy na pewno chcesz zapisać dane?” – to po kilku godzinach korzystania z aplikacji użytkownik będzie naprawdę poirytowany.

Musimy mieć też pewność, że użytkownicy nie będą wykonywać nieprawidłowych lub nieodpowiednich poleceń ani nie będą wprowadzać nieprawidłowych danych. Zwykle osiągamy to poprzez dodanie do interfejsu użytkownika jakiejś logiki sprawdzającej poprawność.

Trzeba jednak pamiętać, że niemal w każdym przypadku znacznie bardziej użyteczne jest wyłączenie zawczasu możliwości przeprowadzania nieprawidłowych działań. Jako przykład, gdy korzystamy z podejścia MVVM, lepiej jest ustawić kontekst *CanExecute* przycisku na wartość *false*, zamiast pozwalać użytkownikom go klikać a następnie wyświetlać im irytujące powiadomienie w okienku *MessageBox*.

WPF i Silverlight oferują różne sposoby komunikacji z użytkownikiem. W przeszłości programiści korzystali z okien dialogowych, które wyskakiwały i przejmowały całą komunikację z aplikacją, dopóki nie zostały zamknięte. Nazwa „okno dialogowe” wzięła się z tego, że okno takie jest formą dialogu pomiędzy interfejsem użytkownika a użytkownikiem. Dialog może być jednokierunkowy (na przykład powiadamianie użytkowników o błędzie) lub dwukierunkowy (użytkownik jest proszony o potwierdzenie lub anulowanie operacji).

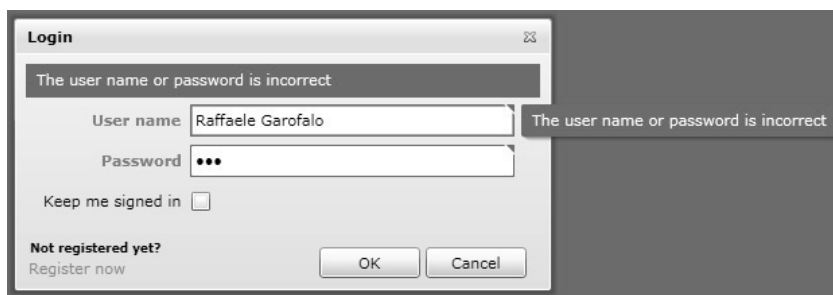
Podejście dialogowe jest łatwe do zaimplementowania, ale bardzo inwazyjne wobec użytkowników. Wyświetlenie okna dialogowego wymusza na użytkownikach kliknięcie przycisku – takiego jak „Tak” lub „Nie”. Ponieważ okna dialogowe przechwytywały całą komunikację z aplikacją, dopóki użytkownik nie zareaguje na nie, to cały pozostały interfejs użytkownika jest po prostu zamrożony. Okna dialogowe są odpowiednie, gdy musimy uzyskać potwierdzenie, na przykład gdy użytkownik próbuje usunąć rekord danych, ale w większości innych przypadków powinniśmy skorzystać z innego podejścia.

Wzorzec MVVM utrudnia implementowanie podejścia dialogowego, ponieważ model widoku nie wie nic na temat widoku, więc nie wie, jak wchodzić z nim w interakcję. Łatwym rozwiązaniem tego problemu jest wzorzec *mediatora*. Wzorzec ten przeanalizujemy dokładniej w rozdziale 6 „Warstwa interfejsu użytkownika w MVVM”.



UWAGA Mediator jest wzorcem projektowym, który zapewnia centralny węzeł kierujący interakcjami pomiędzy wieloma obiektami. Podczas analizy w rozdziale 7 „Platformy i zestawy narzędzi MVVM” złożonej platformy interfejsu użytkownika zbudowanej przez zespół patterns & practices w firmie Microsoft zobaczymy, że istnieją też lepsze sposoby wykonania tego zadania.

Innym interesującym podejściem, które dobrze pasuje do wzorca MVVM, jest zastosowanie obiektów sprawdzających poprawność danych oraz interfejsu *IDataErrorInfo* w modelu widoku. Na przykład, gdy oczekujemy określonego formatu danych w jakiejś kontrolce, możemy skorzystać z powiadamiania o poprawności danych, co zarówno w WPF, jak i w Silverlight jest łatwe do uzyskania. Jeśli użytkownik wprowadzi nieprawidłowe dane w kontrolce, to możemy łatwo wyróżnić tę kontrolkę i wyświetlić komunikat o błędzie. W przeciwieństwie do okien dialogowych takie komunikaty nie wymagają potwierdzenia i nie zamrażają interfejsu użytkownika. Gdy nastąpi błąd sprawdzania poprawności danych, użytkownicy mogą nadal pracować z interfejsem użytkownika, ale nie będą mogli przesłać danych, dopóki nie zostaną one poprawione. Rysunek 1-7 pokazuje przykład aplikacji biznesowej w Silverlight, która sprawdza poprawność danych przed przesłaniem ich na serwer.



RYСУNEK 1-7 Niestandardowe sprawdzanie poprawności danych przy pomocy Silverlight 4.

Kluczowym punktem tutaj jest to, że musimy wybrać odpowiednie metody do obsługi błędów i powiadomień, na które chcemy uzyskać reakcję użytkownika, w przeciwieństwie do tych, które zapewniają wskazówki lub inne aplikacje i na które użytkownik nie musi reagować. W tym drugim przypadku możemy zastosować mniej inwazyjne podejście.

Pasek wstążki

Od czasu wprowadzenia produktów Microsoft Office 2007 kilka lat temu użytkownicy przyzwyczajali się do nowej kontrolki zwanej „wstążką”. Wstążka jest zestawem narzędzi, które łączy w sobie funkcjonalność zarówno paska menu, jak i paska narzędzi połączone razem w bardziej nowoczesnym podejściu. Nazwa wstążka (ang. *ribbon*) powstała na spotkaniu zespołu opracowującego program Outlook 2003, na którym

postanowiono zaimplementować tę nową kontrolkę po raz pierwszy. Pomysł ten został zaimplementowany na podobieństwo średniowiecznego zwoju, gdzie długa wstęga papieru może być przewijana przy użyciu jednego z dwóch wałków. Korzystając z tego pomysłu zespół wprowadził pojęcie zakładki i grup w kontrolce wstążki.

Celem wstążki jest ograniczenie liczby menu obecnych w aplikacji. Gdy planujemy wprowadzenie paska wstążki w naszej aplikacji, to nie będziemy już potrzebować paska narzędzi albo paska menu, ponieważ wstążka zastępuje te funkcje. Wstążki mają pewne bardzo restrykcyjne ograniczenia projektowe, jeśli więc planujemy użycie wstążki, to powinniśmy stosować się do zasad projektowych zdefiniowanych przez firmę Microsoft, które są dostępne pod adresem: <http://msdn.microsoft.com/en-us/library/cc872782.aspx>.

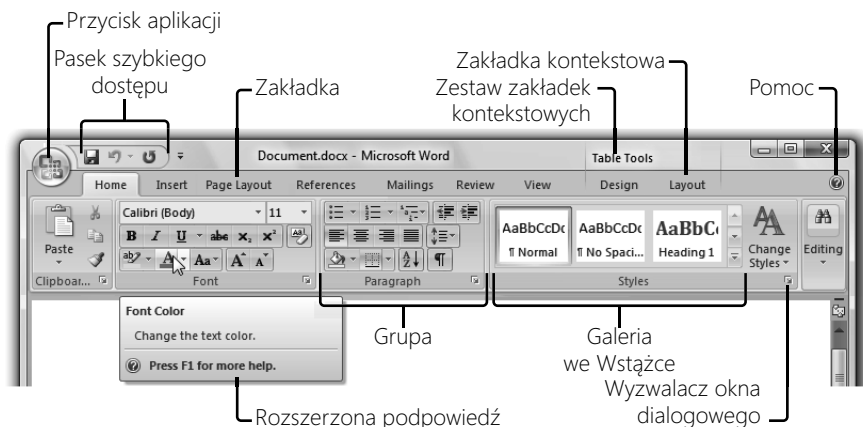
UWAGA Podczas pisania tej książki firma Microsoft wypuściła ostateczną wersję kontrolki Ribbon dla WPF 4 skompilowaną w środowisku .NET 3.5 SP1 z pełną obsługą wzorca MVVM.



Kontrolka Ribbon została wydana w wersji beta dla WPF 3.5 (i działa w WPF 4), ale nie jest dostępna w przypadku Silverlight. Możemy też rozważyć jakieś rozwiązanie autorstwa firmy zewnętrznej, jeśli planujemy wykorzystanie wstążki w aplikacji Silverlight.

Pasek wstążki (kontrolka Ribbon) w WPF został zaprojektowany tak, aby współdziałał z wzorcem MVVM; obsługuje polecenia wytyczone i wiązanie danych, łatwo też go dostosowywać. Struktura kontrolki Ribbon, która wykorzystuje regiony i grupy, chyba najlepiej odpowiada bogatej kontrolce menu na potrzeby MVVM.

Rysunek 1-8 pokazuje główną strukturę paska wstążki wykorzystującej styl wzorowany na Office 2007.



RYСУNEK 1-8 Pasek wstążki w WPF (z witryny MSDN)

Zanim podejmiemy decyzję, czy wstążka dobrze pasuje do naszej aplikacji, musimy mieć świadomość, że skonstruowanie użytecznej wstążki wymaga znacznie więcej wysiłku, niż utworzenie prostego paska menu lub paska narzędzi. Dlatego poniżej przedstawiono kilka elementów, które należy rozważyć:

- **Czy aplikacja jest skomplikowana?** Jeśli budujemy prosty interfejs użytkownika, rozważmy wykorzystanie prostego systemu menu zamiast paska wstążki.
- **Czy użytkownicy mają problemy z odnajdywaniem i wykonywaniem właściwych poleceń?** Dobrze zaprojektowana wstążka może pomóc w złagodzeniu takich problemów z użytecznością.
- **Czy mamy ograniczenia dotyczące miejsca?** Wyświetlanie wstążki wymaga znacznej ilości miejsca na ekranie, jeśli więc planujemy opracowanie aplikacji z niewielkimi oknami, która nie wykorzystuje kart lub wielu okien z dokumentami, to wstążka prawdopodobnie nie będzie odpowiednim rozwiązaniem.

Ogólne rozważania dotyczące stylów kontroltek

W tym rozdziale zobaczyliśmy, że interfejsy użytkownika aplikacji biznesowych mają pewne wspólne cechy wynikające z określonych kontroltek interfejsu i określonych technik prawidłowego wyświetlania kontroltek i widoków.

Stosowanie wspólnych stylów w całej aplikacji jest istotne w aplikacjach biznesowych – zwłaszcza gdy pracujemy z tak elastyczną technologią, jak WPF/Silverlight.

Rozważmy pojęcie czcionki, które obejmuje rodzinę, krój, rozmiar, kolor, itd. W większości aplikacji biznesowych powinniśmy się starać zminimalizować zróżnicowanie czcionek – w istocie najlepiej korzystać tylko z jednego kroju czcionki. W ramach tego pojedynczego kroju czcionki możemy dostosowywać grubość, rozmiar i elementy dekoracyjne, więc jedna czcionka jest wystarczająco elastyczna i może spełniać wiele zadań. Należy unikać mieszania wielu różnych czcionek w tej samej aplikacji.

Po drugie, bardzo ważny dla wielu klientów jest układ kolorów. W tym aspekcie aplikacji musimy rozważyć dwie rzeczy. Czy tworzymy aplikację dla określonego klienta? Jeśli tak, to powinniśmy się starać stosować kolory firmowe w naszej aplikacji – albo przynajmniej powinniśmy się starać zachować te same kolory i style, które występują w innych aplikacjach. Taka ciągłość pomaga użytkownikom odnaleźć się w nowym interfejsie. Po drugie, czy tworzymy aplikację jako część pakietu programów? Emulowanie istniejących pakietów aplikacji może być dobrym pomysłem. Na przykład pakiet Office wykorzystuje udane połączenie kolorów. Emulowanie stylów Office w XAML jest możliwe i sprawia, że aplikacja wygląda profesjonalnie, a przy tym ma znajomy wygląd i łatwy do zrozumienia interfejs, ponieważ większość klientów już pracowała z pakietem Office. Te podobieństwa można przenieść do nowego interfejsu użytkownika.

Wreszcie należy pamiętać, że efektywność XAML opiera się między innymi na szablonach *DataTemplate* i stylach. Nie należy tracić czasu na tworzenie słowników i stylów

dla każdego widoku poprzez zastosowanie operacji kopiowania i wklejania. Trzeba pamiętać, że możemy wykorzystać słownik z innego podzespołu i ładować style na bieżąco. Pracując w ten sposób możemy łatwo zbudować aplikację z niestandardowymi tematami i wdrożyć określony temat dla określonego klienta bez potrzeby zmiany kodu całej aplikacji. Trzeba też pamiętać o tym, żeby nie powielać stylów i układów w wielu miejscach; jeśli scentralizujemy te aspekty przy użyciu szablonów danych i stylów, to możemy zagwarantować większą spójność w interfejsie użytkownika swojej aplikacji.

UWAGA Firma Microsoft wydała zestaw interesujących szablonów, które dostosowują typowe kontrolki WPF. Zespół Silverlight również udostępnił zestaw szablonów dla aplikacji biznesowych w Silverlight. Tematy kontrolki dla WPF i Silverlight można uzyskać pod adresem <http://wpfthemes.codeplex.com>. Jeśli planujemy skorzystać z szablonu aplikacji biznesowej w Silverlight 4, to możemy uzyskać dodatkowe tematy pod adresem <http://www.microsoft.com/downloads/en/details.aspx?FamilyID=e9da0eb8-f31b-4490-85b8-92c2f807df9e&displaylang=en>.



Podział interesów

W informatyce termin „podział interesów” odnosi się do procesu oddzielania fragmentów kodu tak, aby ich funkcjonalności jak najmniej nakładały się na siebie. Główną zasadą tutaj jest to, że chcemy, aby aplikacja była złożona z warstw.

Podział interesów jest kluczową zasadą inżynierii oprogramowania, która stwierdza, że dany problem obejmuje kilka różnych dziedzin, które należy zidentyfikować i rozdzielić, żeby ograniczyć złożoność i osiągnąć wymagane czynniki jakościowe, takie jak solidność, przystosowalność, łatwość utrzymania i możliwość ponownego wykorzystania.

Możemy stosować tę zasadę na różne sposoby. Najczęstszym sposobem rozdzielania interesów jest podział na warstwy według funkcjonalności. Zwykle aplikacja biznesowa będzie się składać z *warstwy interfejsu użytkownika*, która tworzy interfejs graficzny aplikacji; *warstwy domenowej*, która reprezentuje elementy biznesowe (takie, jak klient, zamówienie, itd.), *warstwy biznesowej*, która odpowiada za logikę biznesową aplikacji; oraz *warstwy dostępu do danych*, która odpowiada za zachowywanie i pobieranie danych.

Wszystkie paradygmaty programistyczne pomagają programistom w procesie dążenia do podziału interesów. Na przykład języki programowania zorientowane obiektowo, takie jak Delphi, C++, Java i C#, dokonują podziału na obiekty; wzorzec projektowy, taki jak MVC może oddzielać zawartość od prezentacji i przetwarzanie danych (model) od zawartości. Projektowanie zorientowane na usługi może rozdzielać elementy aplikacji na usługi. Proceduralne języki programowania, takie jak C i Pascal, mogą rozdzielać elementy aplikacji na procedury. Języki programowania zorientowane aspektowo mogą rozdzielać elementy aplikacji na aspekty i obiekty.

Podział interesów jest ważną zasadą projektową także w wielu innych dziedzinach, takich jak planowanie urbanistyczne, architektura i projektowanie przepływu informacji. Celem jest projektowanie systemów w taki sposób, żeby poszczególne funkcje można było optymalizować niezależnie od innych funkcji tak, aby awaria jednej funkcji nie powodowała awarii innych funkcji, a także aby łatwiej było rozumieć, projektować i zarządzać złożonymi, współzależnymi systemami. Do typowych przykładów należą korytarze łączące pokoje, zamiast bezpośrednich przejść pomiędzy pokojami oraz umieszczenie piekarnika w jednym obwodzie, a oświetlenia w innym.

Nieco historii

W 1974 roku Edsger W. Dijkstra, holenderski informatyk, napisał pracę zatytułowaną „O roli myśli naukowej”, która była pierwszą pracą omawiającą pojęcie podziału interesów. W swojej pracy E. Dijkstra wspominał, że:

...podział interesów [jest] jedyną znaną mi dostępną techniką skutecznego porządkowania myśli.

W 1989 roku Chris Reade napisał książkę „Podstawy programowania funkcjonalnego”, w której również wspomina o podziale interesów:

Programista musi zrobić kilka rzeczy na raz, mianowicie, 1. Opisać, co ma zostać obliczone; 2. Zorganizować sekwencję obliczeń w niewielkich krokach; 3. Zorganizować zarządzanie pamięcią podczas obliczeń.

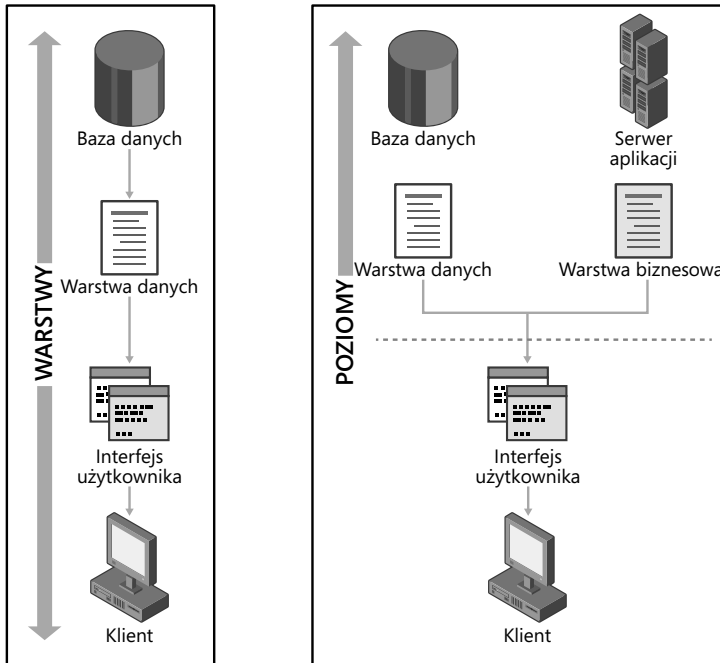
W latach 1990–2000 (nie będziemy zaprzątać sobie głowy dokładniejszymi datami...) Martin Fowler i Eric Evans zaczęli mówić o wzorcach projektowych związanych z projektowaniem kontekstowym, co prowadziło bezpośrednio do nowoczesnego pojęcia podziału interesów.

Obaj autorzy, a w szczególności Martin Fowler, zaczęli nazywać wzorce, omawiać pojęcia i wzorce, takie jak projektowanie oparte na domenach, odwrócenie sterowania, jednostka pracy oraz wiele innych sposobów zwinnego programowania związanych z tworzeniem aplikacji warstwowych, podatnych na testy i łatwych w utrzymaniu. Niedługo potem wielkie firmy i środowiska programistyczne zaczęły implementować środowiska lub bloki aplikacyjne, które umożliwiały podział interesów w tworzonych aplikacjach i działały z typowymi językami programowania, takimi jak .NET, Java i C.

Wszystkie te wzorce i techniki zobaczymy w tej książce i zastosujemy je wobec rzeczywistego problemu.

Warstwy, poziomy i usługi

Wykorzystanie wzorców takich, jak MVVM wymusza na nas podział kodu aplikacji na różne bloki. To dobrze, ponieważ pomaga to tworzyć łatwe do testowania i elastyczne aplikacje. Zwykle takie bloki kodu są zwane *warstwami*. Czasami zestaw warstw ma określoną interakcję (na przykład wzorec MVVM) i tworzą one wspólny *poziom*, na przykład poziom kliencki w aplikacji MVVM. Rysunek 1-9 pokazuje szkic koncepcyjny różnicy pomiędzy warstwami a poziomami.



RYSUNEK 1-9 Różnica pomiędzy aplikacją warstwową a aplikacją wielopoziomową

W typowej aplikacji wielopoziomowej takiej, jak aplikacja MVVM (interfejs użytkownika, logika biznesowa i baza danych) zwykle mamy dwie/trzy warstwy na poziomie i dwa poziomy. Poziomymi tymi są aplikacja kliencka i zdalna baza danych (fizycznie oddzielona), warstwami dla poziomu klienckiego mogą być warstwa interfejsu użytkownika i warstwa prezentacyjna, natomiast dla poziomu bazodanowego mogą to być warstwa biznesowa i warstwa dostępu do danych.

Gdy aplikacja staje się bardziej skomplikowana lub zaczyna być szerzej dystrybuowana, to powinniśmy rozważyć wykorzystanie aplikacji zorientowanej na usługi (SOA). Na przykład w Silverlight nie możemy ponownie wykorzystać plików binarnych dla warstwy domenowej lub warstwy danych, ponieważ nie są one zwykle skompilowane dla środowiska Silverlight CLR. Rozwiązaniem w tym przypadku jest użycie architektury SOA poprzez usługi WCF RIA Services. Te usługi są pośrednikami

wbudowanymi w WCF, dzięki którym możemy udostępniać kod z biblioteki klas, takiej jak warstwa domenowa, skompilowany dla normalnego środowiska .NET CLR w środowisku Silverlight CLR. Oczywiście, jeśli planujemy przejście na architekturę SOA, to powinniśmy mieć na uwadze inne problemy, które możemy napotkać, takie jak jednoczesne wykonywanie kodu, transakcje, dostępność usługi, itd.

Historia warstwy usługowej

Termin „warstwa usługowa” został ukuty przez Martina Fowlera, jednego z najsłynniejszych architektów oprogramowania, który powiedział:

Warstwa usługowa definiuje granicę aplikacji [wzorzec granicy aplikacji Alistaira Cockburna] i jej zestaw dostępnych operacji z perspektywy komunikacji z warstwami klienckimi. Obejmuje logikę biznesową aplikacji sterując transakcjami i koordynując odpowiedzi w implementacji tych operacji.

Jeśli planujemy przestawienie się na warstwę usługową i architekturę SOA, to musimy mieć na uwadze wiele innych okoliczności architektonicznych. Na przykład przy korzystaniu z platformy Entity Framework i usług RIA Services wysiłek konieczny do przejścia na rozwiązanie SOA nie jest duży, ale architektura SOA wymusza na nas przemyślenie pewnych dodatkowych okoliczności.

Architektura SOA jest skomplikowana. Technologie Microsoft, takie jak WCF i usługi RIA dla Silverlight, mogą być pomocne, ale niestety, kiedy faktycznie zaczniemy korzystać z SOA, będziemy musieli rozważyć dodatkowe możliwe problemy, takie jak jednoczesne wykonywanie kodu, transakcje i dostępność. Każdym z nich będziemy musieli się dokładnie zająć.



UWAGA Data Transfer Object będzie omawiany w rozdziale 3 „Model domenowy”.

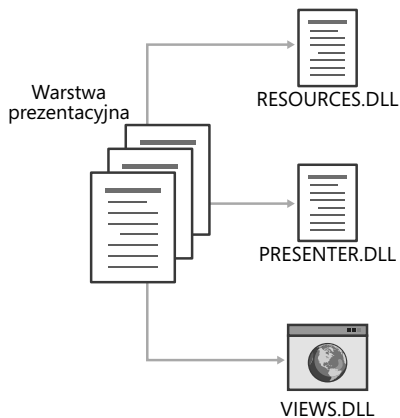
Warstwa biznesowa zawiera „logikę biznesową” naszej aplikacji. Na razie wystarczy wiedzieć, że logika biznesowa musi się w niej znajdować; później zobaczymy, jak skomplikowana może się stać ta warstwa. W istocie składa się zwykle z dwóch warstw: jednej po stronie klienta i jednej po stronie serwera. W przykładowej aplikacji biznesowej dla tej książki warstwa biznesowa będzie zawierać klasy elementów, takie jak *Customers* i *Orders*, oraz logikę biznesową, taką jak przepływy zadań i sprawdzanie poprawności danych.

Wreszcie warstwa dostępu do danych odpowiada za komunikację z repozytorium – magazynem danych, a nie wzorcem repozytorium, o którym przeczytamy więcej w rozdziale 2 „Wzorce projektowe”. Oczywiście magazyn danych może być zwykłą bazą danych, usługą, prostym plikiem tekstowym lub czymkolwiek innym, co może przechowywać dane. Zwykle nie mamy tej warstwy, jeśli korzystamy z relacyjnego mapera obiektów (O/RM), ponieważ stanie się on naszą warstwą danych. Oczywiście

konieczne może być rozszerzenie O/RM; jeśli tak, to takie rozszerzenia będziemy umieszczać w warstwie danych. Gdyby wszystko było tak porządnie zaprojektowane, to bycie architektem oprogramowania nie byłoby takie trudne.

Zacniemy od przeanalizowania każdej warstwy dokładniej, żeby zobaczyć złożoność „dobrze opracowanej” aplikacji warstwowej.

Najpierw mamy warstwę interfejsu użytkownika, która może implementować jeden z następujących wzorców interfejsu użytkownika: MVC, MVP (Model View Presenter) lub MVVM. Celowo pominąłem mniej znane wzorce; wystarczy się skupić jedynie na tych trzech. Ponieważ warstwa prezentacyjna będzie zawierać widoki (okna, gadżety, itd.), orkiestrację (prezenter, kontroler lub widok modelu) oraz zasoby, to prawdopodobnie otrzymamy trzy podwarstwy podobne do struktury na rysunku 1-10.



RYСУNEK 1-10 Jeden z możliwych podziałów warstwy prezentacyjnej

Możemy sobie wyobrazić te trzy warstwy w Visual Studio jako trzy różne projekty bibliotek klas. Korzystając z tego podejścia możemy przypisać każdą warstwę innemu programiście, ale to zależy od rozmiaru projektu i rozmiaru tych warstw. Jeśli będziemy musieli zaktualizować tylko jedną z nich, to ponieważ są luźno powiązane, nie będzie trzeba aktualizować całej warstwy interfejsu użytkownika – a jedynie tę część, której te zmiany dotyczą. Jak widzieliśmy w podrozdziale dotyczącym Expression Blend, możemy teraz dostarczać projektantom atrapę modelu widoku w XAML pozwalając im budować interfejs użytkownika, podczas gdy programiści będą pracować nad podzespołem modelu widoku. Gdyby natomiast widoki i modele widoków były przechowywane w tym samym podzespole (tym samym projekcie Visual Studio), to realizacja tego zadania byłaby trudna.

Warstwa biznesowa jest bardziej skomplikowana. Jeśli stosujemy podejście projektowania sterowanego domeną wykorzystywane w tej książce, to powinniśmy przedstawić „biznes” w formie klas. Na przykład mielibyśmy klasę *Customer*, która zawiera kolekcję klas, takich jak *Address*, itd. Końcowym wynikiem jest struktura klas zwana *Grafem* lub *Modelem*.

W tym momencie można by się zastanawiać „Gdzie napisać kod C#, który pozwoli nam dodać zamówienie dla określonego klienta?”. Jest to powszechna i dość prosta „reguła biznesowa” typowa dla warstwy biznesowej. Zwykle umieszczamy wszystkie takie reguły w wydzielonej warstwie, którą będziemy nazywać „warstwą usługową” (albo jeszcze lepiej „warstwą usług biznesowych”), co nie powinno być mylone z podejściem SOA.

Zastosowanie tej techniki pozwala nam skorzystać z zewnętrznej platformy, takiej jak Windows Workflow, do stosowania reguł biznesowych wobec domeny. Oczywiście możemy rozsyłać takie reguły korzystając z podejścia SOA, ale wtedy uwarunkowania architektoniczne staną się bardziej skomplikowane.

Wreszcie warstwa danych, która zwykle nie jest samym repozytorium, ale składnikiem odpowiedzialnym za wymianę danych pomiędzy domeną a repozytorium. Na przykład, jeśli chcemy utworzyć kolekcję klas *Customer* wykorzystując tabelę *Customer* w bazie danych jako źródło, to użylibyśmy do tego warstwy danych.

Oczywiście to zadanie może być najbardziej kosztowne, jeśli chodzi o zasoby, ponieważ musimy mapować każdy element na odpowiadające mu dane w bazie danych. Będziemy też musieli zapewnić mapowania dla poleceń, takich jak Zapisz, Aktualizuj, itd. Będziemy też musieli utrzymywać schemat procesu mapowania; w przeciwnym razie, gdy zmiany wystąpią w bazie danych, to nasza warstwa danych nie odzwierciedli tych zmian.

Wszystko to jest znacznie łatwiejsze, jeśli korzysta się z narzędzia O/RM. W tej książce zobaczymy, czym jest system O/RM i jak możemy z niego korzystać do mapowania jednostek domenowych na bazę danych.

Podsumowanie

Wzorzec MVVM został wprowadzony przez firmę Microsoft kilka lat temu, żeby zaspokoić zapotrzebowanie na wzorzec prezentacyjny specjalnie przeznaczony na użytek WPF i Silverlight. Wzorzec MVVM jest najlepszym wzorcem prezentacyjnym dla technologii WPF i Silverlight, ponieważ jest w stanie wykorzystać specyficzne wbudowane funkcje Silverlight i WPF, takie jak wiązanie danych, polecenia, zachowania, itd.

Aplikacje biznesowe to takie, które zostały uznane za krytyczne dla prowadzenia biznesu. Jeśli planujemy napisać solidną i łatwą w utrzymaniu aplikację biznesową przy pomocy WPF lub Silverlight, konieczne jest zaimplementowanie wzorca MVVM oraz postępowanie zgodnie z wytycznymi dla interfejsów użytkownika specyficznymi dla aplikacji biznesowych.

Termin „podział interesów” odnosi się do procesu rozdzielania kodu tak, aby możliwie jak najmniej powielał funkcjonalność innego kodu. Główną zasadą tutaj jest to, że chcemy, aby aplikacja składała się z *modułów* zwanych również *warstwami*. Korzystając z tego podejścia możemy tworzyć łatwe do testowania i elastyczne aplikacje, które mogą być opracowywane równolegle przez kilka zespołów.

Wzorce projektowe

Po zakończeniu tego rozdziału będziemy w stanie:

- Zastosować odpowiedni wzorec projektowy dla określonego problemu.
- Rozróżniać trzy główne wzorce prezentacyjne.
- Stosować odwrócenie sterowania i DSL.

Przegląd wzorców projektowych

Napisanie aplikacji komputerowej jest złożonym zadaniem – napisanie takiej, która będzie elastyczna i może być efektywnie rozwijana, jest jeszcze trudniejsze. Jeśli ktoś jest starszym programistą lub architektem oprogramowania, to może już wiedzieć, że chyba najtrudniejszym zadaniem jest wymyślenie, jak pisać kod tylko raz, ponownie wykorzystując, ile się da, aby zaoszczędzić czas i uczynić aplikację łatwiejszą w utrzymaniu.

Zgodnie z hasłem firmy Pirelli Company dotyczącym jej opon: „Siła jest niczym bez odpowiedniej kontroli”, a w tym przypadku kontrola jest bardzo ważna. Przy pisaniu kodu musimy najpierw rozważyć prawdopodobieństwo, że nie będziemy sami pracować nad danym projektem lub aplikacją. Po drugie aplikacja może wymagać utrzymywania i modyfikacji w przyszłości. Na koniec wreszcie – lepiej jest pisać dany kod tylko raz.

Zaawansowani programiści lub architekci oprogramowania prawdopodobnie doświadczili już wielu problemów podczas swojej kariery zawodowej. Być może ktoś ma stałe rozwiązanie dla typowego problemu, które powieli w każdej aplikacji, gdy napotyka ten określony problem. Ten typ rozwiązania – wykorzystanie podobnego kodu do rozwiązywania podobnych problemów – jest zwany *wzorcem* albo częściej *wzorcem projektowym*. Wzorec projektowy jest typowym rozwiązaniem dla częstego problemu, które zostało już zdefiniowane i przetestowane. Od razu powiem, że ta definicja nie oznacza, że każdy wzorec projektowy jest taki sam; wzorec jest raczej przetestowanym podejściem do rozwiązania typowego problemu. Innymi słowy jest to przepis, który musi być dostosowywany w zależności od kontekstu, a nie stosowany w postaci ścisłej, niezmienniej składni.

Ta książka wykorzystuje kilka wzorców projektowych, z których niektóre mogą być znane, a inne całkiem nowe. Na przykład Model View ViewModel (MVVM) jest wzorcem projektowym dla interfejsu użytkownika. Oczywiście wzorce projektowe są wykorzystywane nie tylko przy budowaniu interfejsów użytkownika; istnieją wzorce projektowe dla domeny i innych typowych problemów. Nie będziemy dokładnie badać każdego wzorca projektowego w tej książce, ponieważ nie to jest jej głównym celem; jednak warto przyrzeć się niektórym z powszechnie stosowanych wzorców projektowych, aby zobaczyć, jak można je zastosować w przykładowej aplikacji biznesowej.

Nieco historii

Wzorce projektowe zostały pierwotnie wprowadzone przez Christophera Alexandera w roku 1977 jako typowe rozwiązania powszechnych problemów w dziedzinie architektury budowlanej. Później pod koniec lat osiemdziesiątych Kent Beck i Ward Cunningham zaczęli stosować wzorce w dziedzinie informatyki.

Pierwszą ważną książką na temat wzorców projektowych dla programistów była *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison-Wesley, 1995; ISBN: 0-20163-361-2). Ta dobrze znana praca została napisana przez czterech architektów oprogramowania – tak zwany „Gang of Four” (GOF), w którego skład wchodził Erich Gamma, Richard Helm, Ralph Johnson i John Vlissides. Bardzo polecam ją jako jedną z podstawowych książek dla każdego programisty.

W 2002 roku Martin Fowler napisał bardziej zaawansowaną wersję tej książki: *Patterns of Enterprise Application Architecture* (Addison-Wesley Professional, 2002; ISBN: 0-32112-742-0). Znana potocznie jako „PoEAA” książka ta omawiała problemy architektoniczne, które nie były pierwotnie omówione przez GOF. W mojej opinii jest to kolejny kamień milowy w literaturze informatycznej i również szczerze ją polecam.



UWAGA Żeby dokładnie zgłębić temat wzorców projektowych, warto zajrzeć pod adres <http://msdn.microsoft.com/en-us/practices/default.aspx>. Strona ta została utworzona przez zespół patterns & practices w firmie Microsoft i analizuje architektoniczne aspekty technologii Microsoft.

Klasyfikowanie wzorców projektowych

Podstawowe wzorce projektowe wprowadzone w 1977 roku dzielą się na trzy główne kategorie: *kreacyjne*, *strukturalne* i *behawioralne*, z których każda ma określoną rolę. Ponadto możemy się spotkać z wzorcami specyficznymi dla interfejsów użytkownika i zaawansowanymi wzorcami dla problemów architektonicznych. Te wzorce są członkami nowej klasyfikacji znanej też jako *klasyfikacja architektonicznych wzorców projektowych*.

Wzorce kreacyjne dotyczą rozwiązywania problemów związanych z tworzeniem obiektów. Wzorce strukturalne zajmują się strukturą klas lub obiektów, a wzorce behawioralne zajmują się projektowaniem nowych sposobów komunikacji lub interakcji między obiektami. Znane są one łącznie jako wzorce Gang of Four, ponieważ zostały wprowadzone we wspomnianej wcześniej książce autorstwa członków GOF.

Wzorce kreacyjne

Poniższa tabela przedstawia wzorce, które zostały zaprojektowane do rozwiązywania problemów związanych z procesem tworzenia obiektów i klas.

Nazwa	Opis	Przykład
Abstract Factory (fabryka abstrakcyjna)	Zapewnia metodę do tworzenia obiektów lub klas, które są powiązane lub zależne od siebie bez określania konkretnej klasy.	<i>Factory.CreateProductA();</i> <i>Factory.CreateProductB();</i>
Factory Method (metoda wytwórcza)	Definiuje interfejs do tworzenia obiektu, ale pozwala podklasom decydować, której klasy obiekt ma utworzyć.	<i>FactoryA.Create();</i> <i>FactoryB.Create();</i>
Builder (budowniczy)	Oddziela tworzenie skomplikowanego obiektu od jego reprezentacji tak, żeby ten sam proces budowania mógł tworzyć różne reprezentacje.	<i>Builder.BuildPartA();</i> <i>Builder.BuildPartB();</i> <i>Build.GetFinalProduct();</i>
Prototype (prototyp)	Określa rodzaj obiektów do tworzenia przy użyciu prototypowego wystąpienia i tworzy nowe obiekty poprzez kopiowanie tego prototypu.	<i>Product = Prototype.Clone();</i>
Singleton	Zapewnia, że klasa ma tylko jedno wystąpienie i zapewnia globalny punkt dostępu do tego wystąpienia.	<i>Singleton.DoSomething();</i>

Wzorce strukturalne

Następna tabela wprowadza wzorce, które zostały zaprojektowane do rozwiązywania problemów związanych ze strukturą obiektu lub klasy.

Nazwa	Opis	Przykład
Adapter	Konwertuje interfejs klasy na inny interfejs oczekiwany przez klientów.	<i>Target obj = new Adapter();</i> <i>obj.DoSomething();</i>
Bridge (most)	Oddziela abstrakcję od jej implementacji tak, aby obie mogły się zmieniać niezależnie.	<i>Var obj = new ConcreteA();</i> <i>obj.DoSomething();</i> <i>obj = new ConcreteB();</i> <i>obj.DoSomething();</i>

Nazwa	Opis	Przykład
Composite (kompozyt)	Układa obiekty w struktury drzewiaste reprezentujące całościowe hierarchie.	<i>Composite.Add(objA);</i> <i>Composite.Add(objB);</i>
Decorator (dekorator)	Dynamicznie dołącza dodatkowe zadania dla obiektu.	<i>obj.SetDecorator(decA);</i> <i>obj.DoDecoration();</i> <i>obj.SetDecorator(decB);</i> <i>obj.DoDecoration();</i>
Facade (fasada)	Zapewnia zunifikowany interfejs do zbioru interfejsów w podsystemie.	<i>Facade.MethodFromObjA();</i> <i>Facade.MethodFromObjB();</i>
Flyweight (waga musza)	Wykorzystuje wspólną pamięć do wydajnej obsługi dużej liczby drobnych obiektów.	<i>A = FWFactory.GetFW("A");</i> <i>B = FWFactory.GetFW("B");</i>
Proxy (pełnomocnik)	Zapewnia substytut lub namiastkę dla innego obiektu pozwalające na dostęp do niego.	<i>var proxy = new Proxy();</i> <i>proxy.RequestChannel();</i>

Wzorce behawioralne

Poniższa tabela opisuje wzorce, które zostały zaprojektowane do rozwiązywania problemów związanych z zachowaniem i interakcjami pomiędzy obiektami lub klasami.

Nazwa	Opis	Przykład
Chain of Response (łańcuch zobowiązań)	Unika łączenia nadawcy żądania z jego odbiorcą pozwalając więcej niż jednemu obiektowi obsłużyć dane żądanie.	<i>Employee.SetSupervisor(Manager);</i> <i>Manager.SetSupervisor(Director);</i> <i>Employee.Execute();</i>
Command (polecenie)	Definiuje żądanie jako obiekt i obsługuje operacje, które można cofnąć.	<i>Command.DoSomething();</i> <i>Command.Redo();</i> <i>Command.Undo();</i>
Interpreter (tłumacz)	Mając dany język definiuje reprezentację dla jego gramatyki.	<i>Vocabulary.Add(expressionA);</i> <i>Vocabulary.Add(expressionB);</i> <i>Vocabulary.Translate();</i>
Mediator	Definiuje obiekt, który opisuje interakcje pomiędzy zbiorem obiektów.	<i>Mediator.Add(ObjA);</i> <i>Mediator.Add(ObjB);</i> <i>ObjA.Send("ObjB", Message);</i>
Memento	Bez naruszania niezależności obiektów przechwytuje i udostępnia na zewnątrz wewnętrzny stan obiektu tak, aby można było później przywrócić ten sam stan obiektu.	<i>ObjA.Name = "ObjA";</i> <i>Memento.Save(ObjA);</i> <i>Memento.Restore(ObjA);</i>

Nazwa	Opis	Przykład
Observer (obserwator)	Definiuje zależność jeden-do-wielu pomiędzy obiektami tak, że kiedy stan jednego z obiektów się zmieni, to wszystkie obiekty zależne zostaną powiadomione i automatycznie zaktualizowane.	<i>Observer.Attach(ObjA);</i> <i>Observer.Attach(ObjB);</i> <i>Observer.ChangeSomething();</i> <i>Observer.Notify();</i>
State (stan)	Pozwala obiektowi zmieniać swoje zachowanie, gdy zmienia się jego stan wewnętrzny. Będzie się wydawać, jakby obiekt zmienił swoją klasę.	<i>Context.Add(ObjA);</i> <i>ObjA.ChangeState("A");</i> <i>ObjA.ChangeState("B");</i>
Strategy (strategia)	Definiuje rodzinę algorytmów, obudowuje każdy z nich i pozwala na zamienne ich stosowanie. Przy korzystaniu ze strategii algorytm może się różnić niezależnie od wykorzystujących go klientów.	<i>List.Add(ObjA);</i> <i>List.Add(ObjB);</i> <i>List.SortStrategy(Ascending);</i> <i>List.SortStrategy(Descending);</i>
Template Method (metoda szablonowa)	Definiuje szkielet algorytmu w postaci operacji, przenosząc pewne jego kroki na podklasy.	<i>Template A = new Student();</i> <i>Template B = new Teacher();</i> <i>A.Write(); B.Write();</i>
Visitor (odwiedzający)	Reprezentuje operację, która ma być wykonywana na elementach struktury obiektów.	<i>List.Add(Student("A");</i> <i>List.Add(Student("B");</i> <i>List.Visit(new VoteVisitor());</i>

Nie trzeba teraz zapamiętywać ani w pełni rozumieć wszystkich tych wzorców; wystarczy zapamiętać istnienie tej listy, aby móc do niej sięgnąć w przyszłości. Przykłady niektórych z tych wzorców i sposoby ich prawidłowej implementacji zobaczymy w przykładowej aplikacji biznesowej (systemie CRM), którą utworzymy podczas pracy z tą książką. Zobaczymy też, dlaczego należy wybierać określony wzorec dla określonego problemu. Oczywiście nie będziemy tutaj korzystać z nich wszystkich, ponieważ niektóre stanowią rozwiązania problemów, których nie spotkamy w normalnej aplikacji MVVM. Mimo to warto mieć pełną ich listę i zalecam przejrzanie i eksperymentowanie z tymi wzorcami, ponieważ jedynym sposobem opanowania ich wszystkich jest zrozumienie ich w kontekście konkretnych zastosowań.

Gdy programiści zaczynają poznawać wzorce GOF, to zwykle starają się stosować ten sam wzorec do każdego rozwiązania – ale takie podejście nie jest poprawne. Na przykład, gdybyśmy budowali usługę systemu Windows, to prawdopodobnie przesadą byłoby zastosowanie wobec niej wzorca MVVM.

Wzorce projektowe interfejsu użytkownika

Pełna gałąź wzorców projektowych jest poświęcona budowaniu interfejsów użytkownika. Najlepiej znanymi wzorcami projektowymi interfejsu użytkownika są wzorce Model View Controller (MVC), Model View Presenter (MVP) i Presentation Model (PM), które spotkalismy we wstępie do tej książki jako poprzedników wzorca MVVM. Istnieją też inne wzorce interfejsów użytkownika. Są one podwzorcami wzorców MVC i MVP, ale są rzadko implementowane w .NET Framework.

Interfejs użytkownika jest chyba najbardziej ulotną częścią aplikacji, ponieważ jest podatny na częste zmiany w czasie. Mniej doświadczeni programiści zwykle wiążą interfejs użytkownika i model ze sobą umieszczając logikę biznesową związaną z modelem w kodzie interfejsu użytkownika, co prowadzi do aplikacji trudnych w utrzymaniu. Innym powszechnym problemem jest to, że mniej doświadczeni programiści często mieszają część logiki prezentacyjnej związaną z interfejsem użytkownika z logiką biznesową. To powoduje, że obszar aplikacji możliwy do testowania się zmniejsza i trudniejsze staje się testowanie i utrzymanie podziału interesów w aplikacji.

Zanim zaczniemy mówić o dostępnych wzorcach projektowych i do jakich typów technologii najlepiej pasują, pozwolę sobie wyjaśnić, kiedy powinniśmy używać wzorca projektowego dla interfejsu użytkownika, a kiedy nie. Głównym celem tych wzorców jest oddzielenie logiki biznesowej od interfejsu użytkownika, aby interfejs użytkownika był bardziej podatny na testowanie i łatwiejszy w utrzymaniu oraz aby wyeliminować konieczność pisania logiki biznesowej w interfejsie użytkownika, czego zawsze należy unikać.

Wielu programistów nieprawidłowo rozumie to fundamentalne pojęcie i próbuje uczynić interfejs użytkownika całkowicie niezależnym – próbują oni zupełnie oddzielić model od interfejsu użytkownika. Jednak takie podejście jest niedobre, ponieważ interfejs użytkownika, zwykle definiowany jako widok, ma jakieś zależności od modelu; widok jest zaprojektowany do wyświetlania informacji zapewnianych przez określony model lub zestaw modeli. Te informacje są następnie przetwarzane przez obiekt pośredni, którym we wzorcu MVVM jest model widoku.

Utworzenie niezależnego widoku jest całkiem niezłym osiągnięciem, ale nie jest celem wzorca projektowego dla interfejsu użytkownika. Jednakże przeciwnie założenie też nie jest prawdziwe: ważne jest, aby model był obojętny i nieświadomy względem widoku, ponieważ być może będziemy chcieli ponownie wykorzystać ten model stosując dodatkowe widoki lub w innych aplikacjach. Na razie wystarczy zapamiętać, że zbudowanie niezależnego widoku *nie jest wymaganiem* wzorca interfejsu użytkownika. Wspomnę też tutaj, że model będzie zawierać logikę biznesową powiązaną z operacją biznesową, którą może wykonywać. W każdym razie ta logika biznesowa nie powinna nigdy zawierać żadnej logiki prezentacyjnej lub związanej z interfejsem użytkownika, ponieważ, jak wspomniano wcześniej, model jest niezależny od widoku.

Powinniśmy też mieć na uwadze, że wzorce interfejsów użytkownika, które przeanalizujemy w dalszych częściach tego rozdziału, są propozycjami i wskazówkami,

jak uczynić interfejs użytkownika możliwym do testowania i łatwym w utrzymaniu – ale nie są ograniczeniami. Zwłaszcza w przypadku elastycznego wzorca, takiego jak MVVM, konieczne może być zaprojektowanie jakichś rozwiązań hybrydowych dla rozwiązania określonych problemów, które nie pasują do podstawowej struktury wzorca.

Nieco historii

Ojcem wszystkich wzorców projektowych dla interfejsów użytkownika jest wzorzec MVC, który po raz pierwszy opisał w roku 1979 Trygve Reenskaug, programista pracujący w języku Smalltalk w firmie Xerox. W pierwotnym wzorcu MVC widok odpowiadał za zarządzanie kontrolkami graficznymi wyświetlanymi na ekranie, kontroler odpowiadał za interpretowanie danych wprowadzanych z klawiatury i myszy, a model był obiektem odpowiedzialnym za zarządzanie danymi i zachowaniami domeny aplikacji.

Na przestrzeni lat wzorzec MVC podzielił się na dwa odgałęzienia: bierny wzorzec MVC, gdzie kontroler odpowiada za sterowanie widokiem i modelem oraz czynny wzorzec MVC, gdzie widok aktywnie współpracuje z modelem i nasłuchuje jego zmian.

We wczesnych latach dziewięćdziesiątych XX wieku programiści w firmie Taligent Corporation zaczęli stosować alternatywną interpretację wzorca MVC – wzorzec MVP, który usunął kontrolera i wprowadził prezentera. To podejście znacząco się różni, ponieważ prezenter jest świadom widoków, a każdy widok zna swojego prezentera. Ten wzorzec został szeroko przyjęty zarówno w aplikacjach klienckich, jak i dla sieci WWW.

W roku 2004 Martin Fowler wprowadził swój zestaw korporacyjnych wzorców projektowych, który zawierał przodka wzorca MVVM, wzorzec PM. Niestety ze względu na swoje ścisłe wymagania wzorzec PM nigdy nie odniósł takiego sukcesu jak wzorce MVC lub MVP, ponieważ był zaprojektowany dla technologii podobnej do Windows Presentation Foundation (WPF), która jeszcze się nie ukazała.

W roku 2005 firma Microsoft zastosowała wzorzec PM wobec WPF wprowadzając wzorzec wywodzący się z PM, ale specjalnie dostosowany do WPF – wzorzec MVVM, który w pełni wykorzystuje możliwości silnika wiązania w WPF i Silverlight.

Wzorzec MVC

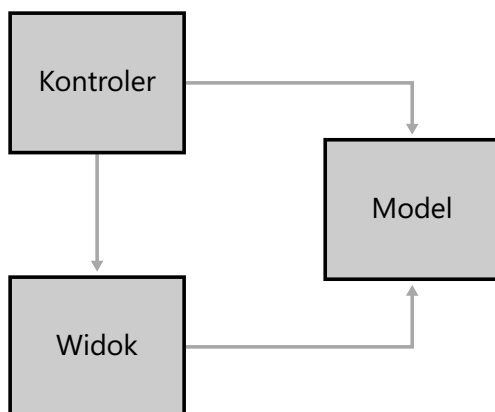
Wzorzec MVC składa się z trzech obiektów, z których każdy odpowiada za określoną funkcję w kontekście interfejsu użytkownika. Możemy stosować wzorzec MVC w aplikacji WWW (która jest z założenia aplikacją bezstanową), gdzie kontroler odpowiada za przetwarzanie danych wprowadzanych przez użytkownika i koordynowanie

wywołań po stronie serwera aż do wyświetlenia widoku (jak w przypadku ASP.NET MVC); jednakże można też stosować ten wzorec w stanowej technologii klienckiego interfejsu użytkownika, takiej jak Windows Forms lub WPF.

W MVC:

- Model (Model) reprezentuje dane w aplikacji w logiczny sposób; odpowiada za przechowywanie danych i uświadamianie innym obiektom zmian danych.
- Widok (View) jest graficznym przedstawieniem modelu; jest odpowiedzialny za wyświetlanie danych z modelu w odpowiedniej formie.
- Kontroler (Controller) jest obiektem sterującym w tym wzorcu; odpowiada za przechwytywanie danych wprowadzanych przez użytkownika (przy pomocy myszy i klawiatury) oraz interakcję z modelem i/lub widokiem.

Rysunek 2-1 pokazuje strukturę podstawowego projektu MVC. Ten projekt jest też zwany biernym wzorcem MVC i jest domyślną implementacją.



RYSUNEK 2-1 Bierny wzorec MVC

Najważniejszym punktem tej implementacji jest to, że model nie jest świadomy ani widoku, ani kontrolera. Model pozostaje niezależny, więc możemy go programować i testować w osobnym kontekście. Jednakże zarówno widok, jak i kontroler są świadome modelu: widok, ponieważ odpowiada za wyświetlanie danych z modelu, a kontroler, ponieważ jest pomostem pomiędzy wejściem od użytkownika a zmianami w modelu. Widok i kontroler są też świadome siebie nawzajem; w domyślnej biernej implementacji kontroler zna swoje widoki, ale widoki nie są świadome swojego kontrolera.

Wydruk 2-1 pokazuje teoretyczną implementację wzorca MVC w języku C#. Ten przykład wykorzystuje platformę ASP.NET MVC.

WYDRUK 2-1 Wzorzec MVC wykorzystujący ASP.NET MVC V2 i C#

```
/// <summary>
/// Prosty model reprezentujący element Employee (pracownik)
/// </summary>
public class Employee
{
    /// <summary>
    /// Imię
    /// </summary>
    public string FirstName { get; set; }

    /// <summary>
    /// Nazwisko
    /// </summary>
    public string LastName { get; set; }

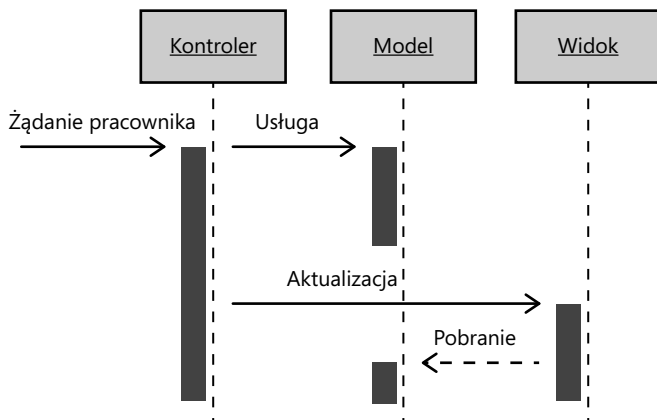
    /// <summary>
    /// Nazwa firmy
    /// </summary>
    public string Company { get; set; }
}

///<summary>
/// Kontroler odpowiadający za wyświetlanie widoków
///</summary>
public class HomeController : Controller
{
    /// <summary>
    /// Działanie, które wyświetla widok Index
    /// </summary>
    public ActionResult DisplayEmployee()
    {
        var model = new Employee
        {
            FirstName = "John",
            LastName = "Smith",
            Company = "Microsoft"
        };
        return View(model);
    }
}

<h2>DisplayEmployee</h2>
<fieldset>
    <legend>Fields</legend>
    <div class="display-label">FirstName</div>
    <div class="display-field"><%= Model.FirstName %></div>
    <div class="display-label">LastName</div>
    <div class="display-field"><%= Model.LastName %></div>
    <div class="display-label">Company</div>
    <div class="display-field"><%= Model.Company %></div>
</fieldset>
```

Implementacja wzorca MVC na wydruku 2-1 wykorzystuje prosty model: pracownika będącego obiektem biznesowym. Kontroler odpowiada za utworzenie modelu w oparciu o żądanie „nowego pracownika” dokonane przez użytkownika. Gdy użytkownik dokonuje żądania nowego pracownika, kontroler tworzy nowe wystąpienie określonego widoku i wstrzykuje model do tego widoku. Na koniec widok przedstawia model korzystając ze znaczników HTML i składni wiązania MVC Framework.

Oczywiście w rzeczywistej aplikacji mielibyśmy usługę lub warstwę danych pobierającą model z bazy danych i przesyłającą go do kontrolera. Rysunek 2-2 pokazuje przebieg procesu dla tej implementacji.



RYСУNEK 2-2 Przebieg żądania w MVC

Rysunek 2-3 pokazuje końcowy wynik w przeglądarce.

DisplayEmployee

Fields

FirstName

John

LastName

Smith

Company

Microsoft

[Edit](#) | [Back to List](#)

RYСУNEK 2-3 Końcowy wynik aplikacji MVC wykorzystującej ASPNET MVC

Zalety i wady wzorca MVC

Wzorzec MVC najlepiej pasuje do aplikacji WWW. Jego siłą jest możliwość wyświetlania tego samego modelu w różnych widokach i możliwość zmieniania sposobu przedstawiania widoku bez wpływania na model (który nie jest świadomy widoków). Inną siłą jest łatwość testowania. Ponieważ widok jest też nieświadomy modelu, to kontroler może po prostu wykorzystywać symulowany model do celów testowych. To sprawia, że MVC dobrze pasuje do podejścia programowania sterowanego testami (TDD).

To powiedziawszy wzorzec MVC może być też używany w aplikacjach klienckich, które nie są bezstanowe (jak aplikacje WWW). W istocie istnieją popularne platformy MVC specjalnie przeznaczone dla technologii klienckich takich, jak Windows Forms lub Java.

Z drugiej strony MVC jest skomplikowanym wzorcem i jest sterowany zdarzeniami; kontroler reaguje na zmiany dokonywane przez użytkowników, o których informuje model i widok. Ponadto aktualizacja danych w MVC może pochłaniać znaczną ilość zasobów, ponieważ w przypadku każdej zmiany widok musi być poinformowany i zaktualizowany przez kontroler. Niektóre nowoczesne platformy, takie jak ASP.NET MVC nie stosują wzorca MVC w jego pierwotnej formie.

Ponadto *pierwotny* wzorzec MVC w oryginalnej postaci nie pasowałby dobrze do nowych technologii interfejsów użytkownika, takich jak WPF i Silverlight. Chciałbym przy tym zauważyć, że istnieje obecnie wiele nowoczesnych wzorców projektowych interfejsów użytkownika, które są nieprawidłowo identyfikowane z nazwą „wzorzec MVC”, ale nie stanowią pierwotnego wzorca MVC; są następcami pierwotnego wzorca.

Wzorzec MVP

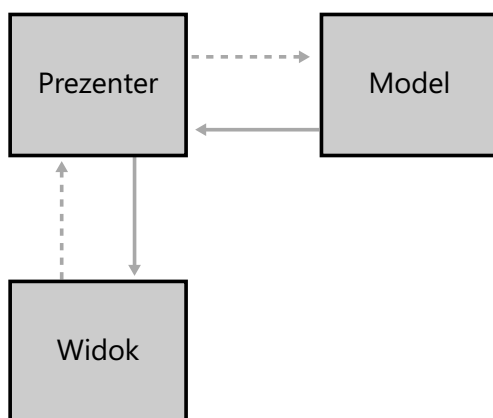
Wzorzec MVP jest określany albo jako ewolucja wzorca MVC, albo jako jego inna interpretacja. Główną różnicą jest to, że we wzorcu MVP widok i prezenter są połączone przy użyciu innego podejścia. W MVC widok jest całkowicie niezależny; w MVP widok jest bierny i deleguje wszelkie działania odpowiadającemu mu prezenterowi. Inną ważną różnicą jest to, że w MVP prezenter komunikuje się z widokiem przy pomocy silnika wiązania lub niestandardowej jego implementacji, jeśli dana technologia interfejsu użytkownika go nie zapewnia. Widok i model w MVP nie są połączone, natomiast w MVC widok jest całkowicie lub częściowo świadomy odpowiadającego mu modelu.

Podobnie do wzorca MVC, wzorzec MVP ma trzy składniki, ale z pewnymi różnicami:

- Model jest taki sam, jak w MVC. Reprezentuje dowolną jednostkę biznesową ze skojarzonymi danymi i logiką biznesową.
- Widok (View) jest interfejsem graficznym odpowiadającym za przedstawianie danych. Bezpośrednio odwołuje się do prezentera, więc może delegować mu interpretację wszelkich działań użytkownika.

- Prezenter (Presenter) steruje logiką interfejsu użytkownika; zna zarówno widok (poprzez interfejs), jak i model. Aktualizuje widok w oparciu o powiadomienia o zmianach pochodzące z modelu i aktualizuje model w oparciu o powiadomienia o zmianach pochodzące z widoku. Jest to obiekt, który obejmuje logikę prezentacyjną i zwykle ustawia wartości właściwości oraz wywołuje metody widoku zamiast korzystać z silnika wiązania.

Wzorzec MVP jest implementowany zarówno dla aplikacji klienckich, jak i WWW. Można przeczytać, że wzorzec MVP pasuje najlepiej do aplikacji WWW, ale z mojego osobistego doświadczenia, ponieważ jego koncepcja jest tak bardzo zależna od prezentera, to lepiej pasuje do aplikacji klienckich, choć jest wystarczająco elastyczny, aby go używać dla aplikacji WWW. Rysunek 2-4 pokazuje pasywną implementację wzorca MVP.



RYSUNEK 2-4 Pasywna implementacja MVP

Wydruk 2-2 pokazuje prosty przykład w Windows Forms, który ilustruje, jak implementować pasywny wzorzec MVP w prostej aplikacji klienckiej.

WYDRUK 2-2 Implementacja MVP wykorzystująca Windows Forms i C#

```
/// <summary>
/// Kontrakt widoku pracownika (Employee)
/// </summary>
public interface IEmployeeView
{
    /// <summary>
    /// Imię
    /// </summary>
    string FirstName { get; set; }

    /// <summary>
    /// Nazwisko
    /// </summary>
```

```

        string LastName { get; set; }

        /// <summary>
        /// Nazwa firmy
        /// </summary>
        string Company { get; set; }
    }

    /// <summary>
    /// Prezentator pracownika odpowiedzialny
    /// za sterowanie logiką interfejsu użytkownika
    /// </summary>
    public sealed class EmployeePresenter
    {
        /// <summary>
        /// Bieżący widok
        /// </summary>
        private IEmployeeView view;

        /// <summary>
        /// Inicjuje nowe wystąpienie klasy <see cref="EmployeePresenter"/>.
        /// </summary>
        /// <param name="view">Widok.</param>
        public EmployeePresenter(IEmployeeView view)
        {
            this.view = view;
        }

        /// <summary>
        /// Inicjuje to wystąpienie.
        /// </summary>
        public void Initialize()
        {
            var model = new Employee
            {
                FirstName = "John",
                LastName = "Smith",
                Company = "Microsoft"
            };

            //Powiązanie modelu z widokiem
            UpdateViewFromModel(model);
        }

        /// <summary>
        /// Aktualizuje widok na podstawie modelu.
        /// </summary>
        /// <param name="model">Model.</param>
        private void UpdateViewFromModel(Employee model)
        {
            this.view.FirstName = model.FirstName;
            this.view.LastName = model.LastName;
            this.view.Company = model.Company;
        }
    }

```

```

}

/// <summary>
/// Konkretny widok.
/// </summary>
public partial class EmployeeView : Form, IEmployeeView
{
    /// <summary>
    /// Odpowiadający prezenter
    /// </summary>
    private EmployeePresenter presenter;

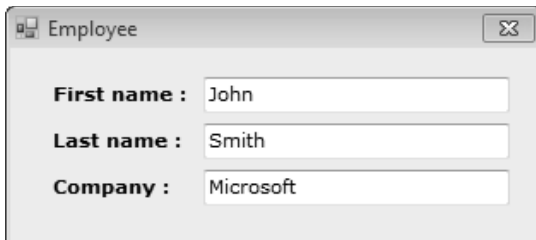
    /// <summary>
    /// Inicjuje nowe wystąpienie klasy <see cref="EmployeeView"/> .
    /// </summary>
    public EmployeeView()
    {
        InitializeComponent();
        this.presenter = new EmployeePresenter(this);
        this.presenter.Initialize();
    }

    /// <summary>
    /// Imię
    /// </summary>
    /// <value></value>
    public string FirstName
    {
        get { return txtFirstname.Text; }
        set { txtFirstname.Text = value; }
    }

    /// pominęto
}

```

Ten przykład znowu wykorzystuje model *Employee*, który widzieliśmy na wydruku 2-1. Jak można zauważyć, najbardziej istotną różnicą w stosunku do wzorca MVC jest to, że widok w MVP jest całkowicie nieświadomy modelu, który przedstawia, ponieważ dane są powiązane z kontrolkami widoku poprzez prezentera. Widok jest całkowicie zależny od prezentera, musi mieć do niego odwołanie, ponieważ nie wie, jak reagować na wejście od użytkownika. Rysunek 2-5 pokazuje efekt końcowy.



RYСУNEK 2-5 Efekt końcowy pasywnego widoku MVP

Zalety i wady wzorca MVP

Tym, czym wzorec MVP odróżnia się od innych wzorców interfejsu użytkownika, są jego role i obowiązki. We wzorcu MVP prezenster steruje całą logiką; widok może jedynie przekazywać powiadomienia dotyczące interakcji z użytkownikiem do prezentera, który może następnie wywoływać metody i zmieniać dane w widoku i/lub w modelu.

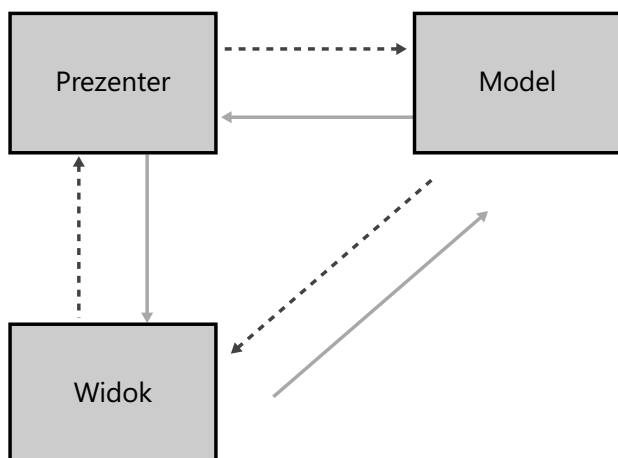
Inny problem tkwi w wywołaniu zwrotnym, które występuje za każdym razem, gdy następuje interakcja użytkownika z widokiem, widok musi wywołać metodę prezentera, a następnie prezenster musi zaktualizować widok.

Wzorec MVP nie jest odpowiedni dla WPF lub Silverlight, ponieważ jego bierna implementacja nie wykorzystuje możliwości silnika wiązania XAML i nie jest w stanie dobrze oddzielić kodu XAML tworzącego interfejs użytkownika od proceduralnego kodu C# koniecznego w widoku, żeby znał on odpowiadającego mu prezentera. Odradzam stosowanie wzorca MVP w aplikacjach WPF i Silverlight. Jeśli ktoś planuje z niego skorzystać, to może okazać się, że nie trzeba żadnej z tych technologii, a lepiej byłoby skorzystać z klasycznej technologii Windows Forms.

Dużą wadą MVP jest to, że cała logika prezentacyjna i każdy proces wiązania musi przechodzić przez prezentera, jeśli więc planujemy w WPF lub Silverlight zastosować wzorec prezentera nadzorującego (Supervising Presenter), który zostanie dokładniej opisany poniżej, to otrzymamy widok, który ma model jako swój kontekst danych oraz oddzielne odwołanie do prezentera.

Alternatywne podejścia do MVP

Innym podejściem jest *prezenter nadzorujący* w MVP. W tym wariacie widok nie jest bierny; zna model, który przedstawia, i wymaga silnika wiązania danych, żeby reagować na zmiany w modelu. Rola prezentera się zmniejsza tak, że odpowiada on tylko za przechwytywanie wejścia użytkownika. Można by sądzić, że to podejście byłoby interesujące, gdyby zostało zastosowane w WPF lub Silverlight – i chyba tak by było, gdybyśmy musieli pracować z kombinacją widok/model, w której interakcja między nimi byłaby bardzo skomplikowana. Z drugiej strony widok ma do utrzymania wiele odwołań, co jest ciężko testować i wymaga więcej interfejsów do zachowania luźnego wiązania. Wreszcie byłoby to skomplikowane, ponieważ musielibyśmy pisać w interfejsie użytkownika kod do zarządzania interakcją z prezensterem. Rysunek 2-6 pokazuje strukturę prezentera nadzorującego w MVP.



RYSUNEK 2-6 Prezenter nadzorujący w MVP

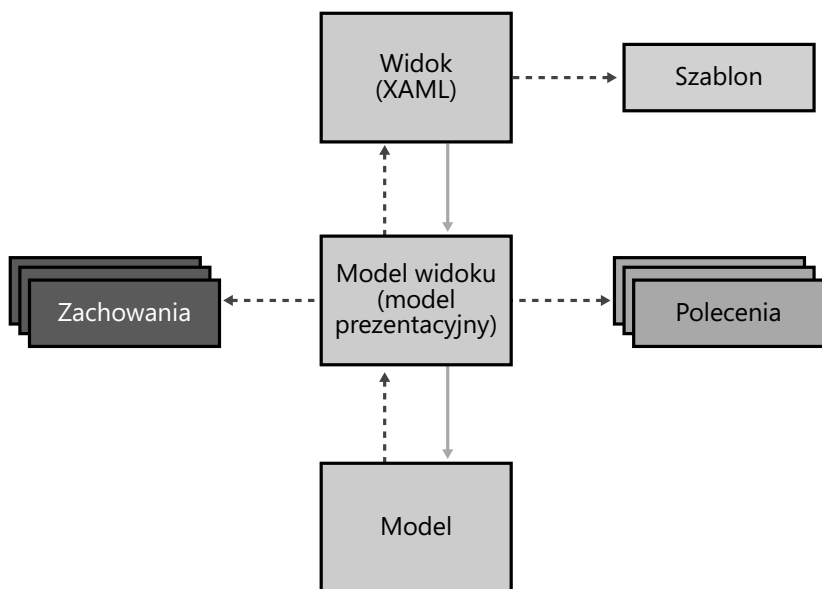
Wzorzec PM i MVVM

Ten podrozdział omawia zarówno wzorzec PM, jak i MVVM, ponieważ są one ze sobą blisko powiązane. Wzorzec PM pojawił się, gdy technologie, takie jak WPF i Silverlight, nie były jeszcze dostępne. Gdy się pojawiły, firma Microsoft zastosowała wzorzec PM wobec WPF i Silverlight używając wzorca MVVM.

Podstawowe zasady PM to utrzymywanie luźno powiązanych relacji pomiędzy modelem prezentacyjnym a widokiem poprzez uczynienie widoku obserwatorem modelu prezentacyjnego i osiągnięcie tego dzięki skorzystaniu z wiązania danych. Model prezentacyjny zna model, ale niekoniecznie musi znać odpowiadający widok. Widok zna tylko swój model prezentacyjny i to wyłącznie poprzez silnik wiązania. Siła i elastyczność wiązania danych w WPF/Silverlight sprawiają, że jest to odpowiedni wzorzec do użycia w aplikacjach WPF/Silverlight.

Wzorzec MVVM jest ewolucją wzorca PM, który ma trzy podstawowe składniki: model, który reprezentuje jednostkę biznesową (jak klasa przykładowa *Employee*), widok (View), który jest interfejsem użytkownika w XAML, i model prezentacyjny (PM) lub model widoku (View Model), który zawiera całą logikę interfejsu użytkownika i odwołanie do modelu, więc działa jako model dla widoku.

Rysunek 2-7 zawiera diagram, który pokazuje, jak zaimplementować wzorzec MVVM. Oczywiście jest to podstawowa implementacja. W tej książce zobaczymy, że musimy implementować wzorzec MVVM na różne sposoby w zależności od typu używanego widoku, co może zwiększyć złożoność w porównaniu do tego, co pokazano na rysunku 2-7.



RYSUNEK 2-7 Podstawowa struktura aplikacji MVVM

Jeśli planujemy pracę w WPF lub Silverlight, to musimy wykorzystać silnik wiązania zapewniany przez te technologie. Dzięki temu nasz model widoku powinien implementować pewne określone interfejsy wymagane przez silnik wiązania WPF i Silverlight.

Jednym z nich jest interfejs *INotifyPropertyChanged* wprowadzony w wersji 2.0 platformy .NET Framework. Ten interfejs implementuje system powiadamiania, który aktywuje się, gdy zmienia się wartość jakiejś właściwości. Jest on wymagany w modelu widoku, aby silnik wiązania w XAML działał poprawnie.

Innym dostosowaniem wzorca PM jest polecenie udostępniane przez interfejs *ICommand*, który jest obecny w WPF i Silverlight. To polecenie może być wiązane z dowolną kontrolką XAML i określa, czy kontrolka może wykonywać określone działanie, czy nie. W WPF polecenie to ma bardziej rozbudowaną implementację w postaci polecenia skierowanego, które jest poleceniem kierowanym poprzez drzewo wizualne interfejsu użytkownika WPF.

Trzecim niestandardowym składnikiem jest szablon danych – *DataTemplate*, struktura XAML, która definiuje, jak przedstawiać określony model widoku lub określony stan modelu widoku. Składniki *DataTemplate* są w istocie widokami, które są przedstawiane w czasie wykonywania aplikacji przez silnik WPF/Silverlight. Są one szczególnie typem widoków, które nie mogą zawierać żadnego kodu, ponieważ są tworzone dynamicznie. Logicznie wyświetlamy model widoku lub model bezpośrednio w interfejsie użytkownika, ale widok tworzony jest w trakcie wykonywania aplikacji i dołączany do modelu widoku lub modelu (poprzez kontekst danych).

Wydruk 2-3 pokazuje uproszczony przykład implementacji wzorca MVVM w WPF. W następnych rozdziałach zobaczymy, jak można dostosowywać każdy składnik modelu widoku.

WYDRUK 2-3 Implementacja MVVM przy użyciu WPF 4

```
/// <summary>
/// model widoku dla widoku Employee
/// </summary>
public sealed class EmployeeViewModel : INotifyPropertyChanged
{
    public EmployeeViewModel()
    {
        var employee = new Employee
        {
            FirstName = "John",
            LastName = "Smith",
            Company = "Microsoft"
        };

        //Wiązanie modelu z modelem widoku
        this.Firstname = employee.FirstName;
        this.Lastname = employee.LastName;
        this.Company = employee.Company;
    }

    #region INotifyPropertyChanged
    /// <summary>
    /// Występuje, gdy wartość właściwości ulega zmianie.
    /// </summary>
    public event PropertyChangedEventHandler PropertyChanged;

    /// <summary>
    /// Wywoływane, gdy właściwość ulegnie zmianie.
    /// </summary>
    /// <param name="name">Nazwa.</param>
    public void OnPropertyChanged(string name)
    {
        var handler = PropertyChanged;
        if (handler != null)
        {
            PropertyChanged(this, new PropertyChangedEventArgs(name));
        }
    }
    #endregion

    /// <summary>
    /// Prywatne pole dla imienia
    /// </summary>
    private string firstname;

    /// <summary>
    /// Pobiera lub ustawia imię.
```

```

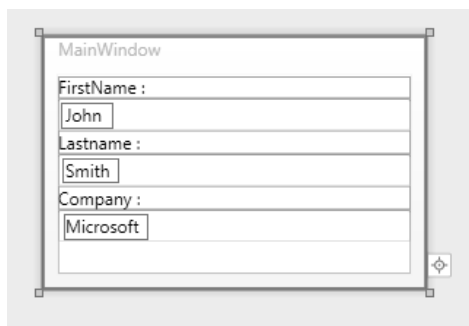
    /// </summary>
    /// <value>Imię.</value>
    public string Firstname {
        get
        {
            return firstname;
        }
        set
        {
            if (firstname != value)
            {
                firstname = value;
                OnPropertyChanged("Firstname");
            }
        }
    }
}
// pominięte
}

<Window.DataContext>
    <vm:EmployeeViewModel />
</Window.DataContext>
<StackPanel Orientation="Vertical">
    <TextBlock>FirstName :</TextBlock>
    <TextBox
        Text="{Binding Path=Firstname,
            Mode=TwoWay,
            UpdateSourceTrigger=PropertyChanged}" />
    <TextBlock>Lastname :</TextBlock>
    <TextBox
        Text="{Binding Path=Lastname,
            Mode=TwoWay,
            UpdateSourceTrigger=PropertyChanged}" />
    <TextBlock>Company :</TextBlock>
    <TextBox
        Text="{Binding Path=Company,
            Mode=TwoWay,
            UpdateSourceTrigger=PropertyChanged}" />
</StackPanel>

```

Ten przykład znowu wykorzystuje jednostkę *Employee* z wydruku 2-1. Możemy tu zobaczyć prosty model widoku, który nie jest niczym innym, jak klasą implementującą interfejs *INotifyPropertyChanged* i udostępniającą właściwości modelu, które chcemy przedstawiać w interfejsie użytkownika. Widok jest oknem XAML, które wykorzystuje wystąpienie modelu widoku jako źródło danych i które wiąże każdą właściwość modelu widoku z określoną kontrolką.

Ponieważ Microsoft Visual Studio 2010 w pełni obsługuje silnik wiązania WPF i Silverlight, i ponieważ wiążemy jednostkę *Employee* w konstruktorze modelu widoku, to nie musimy nawet uruchamiać aplikacji, aby zobaczyć końcowy wynik – widok jest funkcjonalny nawet w trybie projektowania w środowisku programistycznym Visual Studio, jak pokazano na rysunku 2-8.



RYSUNEK 2-8 Funkcjonalna aplikacja MVVM w module projektowym Visual Studio 2010

Zalety i wady MVVM

Po pierwsze, po stronie zalet wzorzec MVVM jest zaprojektowany na użytek WPF lub Silverlight, ale nie jest ograniczony wyłącznie do tych technologii; możemy implementować MVVM również w Windows Forms lub innej technologii interfejsu użytkownika. Jednakże siła i elastyczność WPF albo Silverlight (w tym funkcje, takie jak wiązanie danych, XAML, szablony danych, zachowania, itd.) sprawiają, że MVVM znacznie łatwiej zaimplementować w WPF/Silverlight.

Model widoku jest sednem aplikacji MVVM, więc trzeba rozważyć wszystkie środki zaradcze, żeby nie otrzymać niestabilnej i chaotycznej aplikacji. Należy dokładnie trzymać się wskazówek i eksperymentować z różnymi rozwiązaniami.

Nie należy starać się dopasować swojego modelu widoku do dziwnej architektury tylko dlatego, że nie wiemy, jak napisać konkretne zachowanie lub szablon *Data-Template* w WPF lub Silverlight. Wzorzec ten jest zaprojektowany dla tych technologii, więc trzeba je dobrze opanować, zanim opanuje się sam wzorzec MVVM.

Na koniec jedną z kluczowych zalet przyjęcia wzorca MVVM jest to, że widok jest obserwatorem modelu widoku, co ułatwia oddzielne budowanie interfejsu użytkownika i pozwala nam zastąpić widok później lub nawet w trakcie działania programu bez konieczności zmieniania logiki prezentacyjnej.

Zaawansowane wzorce i techniki projektowe

Wzorce projektowe dostępne wśród wzorców GOF zostały utworzone i przyjęte do rozwiązywania pewnych typowych problemów związanych z programowaniem zorientowanym obiektowo, takich jak tworzenie obiektu albo otwieranie komunikacji pomiędzy dwoma różnymi obiektami. Wzorce interfejsów użytkownika zostały zaprojektowane i przyjęte do oddzielania logiki biznesowej od interfejsu użytkownika oraz w celu ułatwienia testowania interfejsu użytkownika i zwiększenia jego elastyczności. W miarę wzrostu zaawansowania architektur oprogramowania te podstawowe techniki nie były w stanie sprostać zwiększonym wymaganiom architektonicznym.

Dało to początek nowym rozwiązaniom zwanym obecnie *wzorcami korporacyjnymi* albo *wzorcami dla aplikacji korporacyjnych*.

Ta książka nie omawia wszystkich dostępnych wzorców i technik korporacyjnych, ale zapewni przegląd typowych wzorców korporacyjnych i zaprezentuje szczegółowo te, które są fundamentalne dla przyjęcia MVVM.

WIĘCEJ INFORMACJI Warto przeczytać książki *Pattern of Enterprise Application Architecture* – autor: Martin Fowler i *Domain Driven Design* – autor: Eric Evans. Obaj są twórcami korporacyjnych wzorców architektonicznych. Te dwie książki są w mojej opinii lekturą obowiązkową dla każdego starszego programisty i architekta oprogramowania, zwłaszcza jeśli planują budowanie złożonych aplikacji biznesowych wykorzystujących MVVM.

Martin Fowler wyczerpująco omówił wszystkie te wzorce i podzielił je na 10 różnych kategorii, każdą specyficzną dla określonego kontekstu. Pierwszą kategorię stanowi wzorec *logiki domenowej*, który przeanalizujemy w następnym rozdziale. Istnieją trzy kategorie związane z warstwą danych, którą omówimy dokładnie w rozdziale 4 „Warstwa dostępu do danych”. Istnieją też dwa wzorce dla jednoczesnego wykonywania kodu i stanu sesji, które nie są używane w tej książce. W następnym podrozdziale omówimy dwa szczególne wzorce/podejścia, które są obowiązkowe przy stosowaniu wzorca MVVM albo ogólnie przy każdej aplikacji biznesowej: wzorec wstrzykiwania zależności zwany także odwróceniem sterowania i wzorec języka specyficznego dla domeny. Przedstawię też wprowadzenie do podejścia TDD – techniki programowania zwinnego do testowania aplikacji podczas fazy jej rozwoju.

Wzorec odwrócenia sterowania

Termin odwrócenie sterowania (IoC – Inversion of Control) jest techniką programowania komputerów, gdzie przebieg sterowania aplikacją jest odwrócony. Zamiast tego, żeby funkcja wywołująca decydowała, jak skorzystać z obiektu, w tej technice to obiekt wywoływany decyduje, kiedy i jak odpowiedzieć funkcji wywołującej, więc funkcja wywołująca nie odpowiada za sterowanie głównym przebiegiem aplikacji.

To podejście sprawia, że kod jest na tyle elastyczny, iż poszczególne jego części mogą być odłączone od siebie. Kod może nie być świadomy tego, co się dzieje na stosie wywołań, ponieważ obiekt wywoływany nie musi czynić żadnych założeń odnośnie tego, co robi funkcja wywołująca.

Wzorec wstrzykiwania zależności (Dependency Injection) jest po prostu konkretną implementacją odwrócenia sterowania. Niestety, jak stwierdza Martin Fowler w swojej książce, jest wiele zamieszania wokół tych terminów, ponieważ powszechne pojemniki odwrócenia sterowania dostępne dla języków takich, jak Java lub .NET są zwykle *identyfikowane* po prostu jako pojemniki IoC, ale techniki *implementowane* w kodzie przy korzystaniu z tych platform są zgodne z wzorcem wstrzykiwania zależności, który

jest tylko jedną z dostępnych, konkretnych implementacji odwrócenia sterowania. Na przykład, jeśli planujemy pracę z podzieloną na moduły aplikacją WPF/Silverlight przy użyciu dobrze znanej platformy, takiej jak Prism, to możemy zaimplementować odwrócenie sterowania przy użyciu wzorca lokalizatora usług (Service Locator), a nie wstrzykiwania zależności, ponieważ potrzebujemy globalnego pojemnika IoC dostępnego dla wszystkich modułów.

Wyobraźmy sobie, że mamy prosty element *LogWriter*, który jest używany do zapisywania komunikatów albo w określonej bazie danych, albo w podanym pliku. Możemy przedstawić to tak, jak pokazano na rysunku 2-9.



RYСУNEK 2-9 Podstawowa struktura systemu rejestrującego komunikaty w wielu miejscach

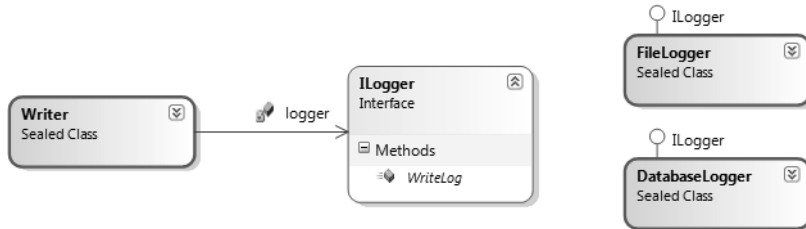
Diagram UML na rysunku 2-9 jest dość jasny; istnieje abstrakcyjna klasa *BaseLogger*, która udostępnia komunikat *WriteLog* oraz dwie konkretne klasy, które dziedziczą po *BaseLogger*. Udostępniają one tę metodę na dwa sposoby: jeden zapisuje komunikat w bazie danych, a drugi w systemie plików. Poniższy kod pokazuje *nieprawidłowy* sposób użycia jednego z tych konkretnych rejestratorów – bez zastosowania implementacji IoC:

```

static void Main(string[] args)
{
    /* Nieprawidłowy sposób
     *
     * */
    var firstLogger = new FileLogger();
    firstLogger.WriteLog("Some Text.");
    var secondLogger = new DatabaseLogger();
    secondLogger.WriteLog("Some other Text.");
    Console.ReadKey();
}
  
```

Największym problemem przy tym podejściu – braku zastosowania implementacji IoC – jest to, że jeśli chcemy określić inny sposób rejestrowania komunikatów podczas działania programu, to trzeba będzie przepisać nieco kodu. To ogromne ograniczenie architektoniczne. Załóżmy na przykład, że chcemy pozbyć się obiektu *FileLogger*. Nie jest to łatwe. Nie możemy go po prostu wyeliminować, ponieważ aplikacja przestałaby działać albo co najmniej trzeba by zmodyfikować ją i przekompiłować, aby działała nadal.

Żeby rozwiązać ten problem, pierwszym krokiem jest rozłączenie istniejącej hierarchii poprzez użycie interfejsu zamiast abstrakcyjnej klasy bazowej, jak zilustrowano na rysunku 2-10. W ten sposób po prostu definiujemy kontrakt pomiędzy konkretnym rejestratorem a jego interfejsem. Aby później zapisać komunikat w innym miejscu, musimy po prostu przedstawić interfejs w określony sposób.



RYСУNEK 2-10 Refaktoring klasy *LogWriter* przy użyciu wspólnego interfejsu

Poniższy kod jest zmienioną wersją wykorzystującą podejście IoC do zadeklarowania typu rejestratora, który ma być użyty podczas działania aplikacji. To podejście nadal jest proceduralne, ponieważ decyduje, którego rejestratora użyć, ale przynajmniej rozłącza kod, więc jest nieco bardziej elastyczną wersją niestandardowego rejestratora.

```

/// <summary>
/// Niestandardowy rejestrator, który może wykorzystywać dowolne miejsce docelowe.
/// </summary>
public sealed class Writer
{
    /// <summary>
    /// Dostęp do wstrzykniętego rejestratora
    /// </summary>
    private ILogger logger;

    /// <summary>
    /// Inicjuje nowe wystąpienie klasy <see cref="Writer"/>.
    /// </summary>
    /// <param name="logger">Rejestrator.</param>
    public Writer(ILogger logger)
    {
        this.logger = logger;
    }

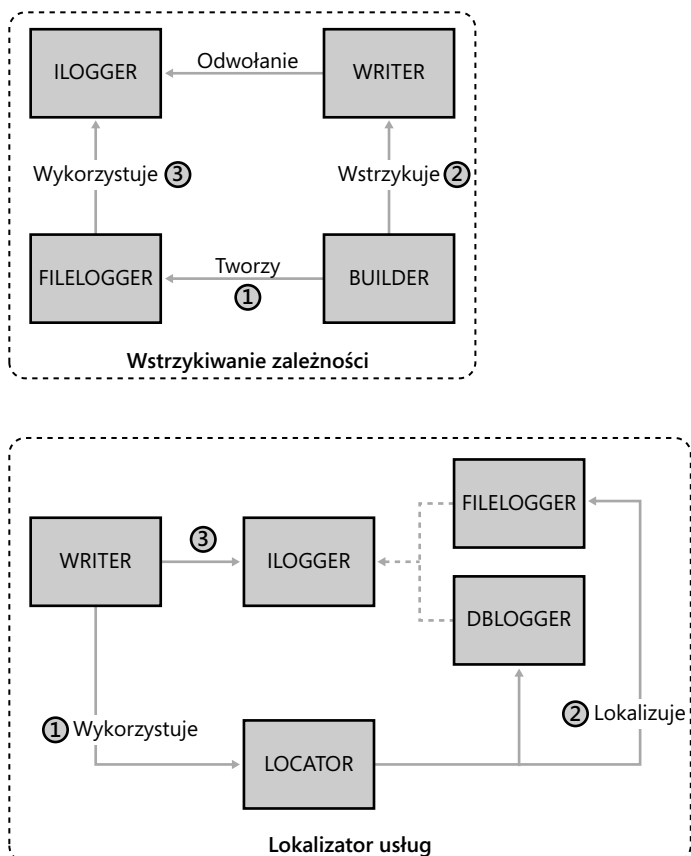
    /// <summary>
    /// Zapisuje podany komunikat.
    /// </summary>
    /// <param name="message">Komunikat.</param>
    public void Write(string message)
    {
        this.logger.WriteLog(message);
    }
}

```

W tym momencie potrzebne jest nam coś pośredniczącego pomiędzy aplikacją a rejestratorem, co będzie mogło ustalać, którego rejestratora użyć w trakcie działania programu. Poniższy przykład wykorzystuje w tym celu kod proceduralny bez korzystania z wzorców wstrzykiwania zależności lub lokalizatora usług:

```
static void Main(string[] args)
{
    // IoC bez pojemnika IoC
    var firstLogger = new FileLogger();
    // Wstrzykiwanie określonego rejestratora
    var writer = new Writer(firstLogger);
    writer.Write("Log for the File system.");
    Console.ReadKey();
}
```

Alternatywnym rozwiązaniem byłoby zaimplementowanie wstrzykiwania zależności lub lokalizatora usług. Rysunek 2-11 pokazuje główną różnicę pomiędzy tymi dwoma podejściami.



RYSUNEK 2-11 Różnice pomiędzy wzorcami wstrzykiwania zależności i lokalizatora usług

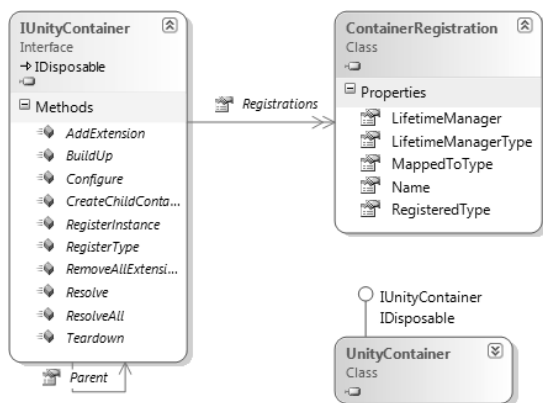
W następnym podrozdziale zobaczymy, jak implementować te dwie techniki korzystając z Microsoft Unity lub Microsoft Managed Extensibility Framework.

Microsoft Unity

Microsoft Unity jest platformą aplikacyjną, która jest dostarczana jako część biblioteki Microsoft Enterprise, ale można też ją pobrać jako samodzielny składnik z witryny CodePlex pod adresem <http://unity.codeplex.com>. W momencie pisania tej książki najnowszą wersją była wersja 2.0 dostępna dla WPF i Silverlight. Można też korzystać z Microsoft Unity w dowolnym innym rodzaju aplikacji .NET.

Unity jest rozszerzalnym pojemnikiem wstrzykiwania zależności, poprzez którego możemy stosować w swoim kodzie wstrzykiwanie zależności korzystając albo z podejścia deklaratywnego (XML), albo z podejścia proceduralnego (C# lub Visual Basic .NET). Dzięki Unity możemy wstrzykiwać kod do konstruktorów, właściwości i metod.

Unity posiada rozszerzalny silnik podstawowy zwany „pojemnikiem” (container), który implementuje interfejs *IUnityContainer* (możemy po nim dziedziczyć, jeśli musimy rozszerzyć istniejącą implementację). Zawiera on trzy główne metody, które wykorzystuje do rejestrowania wystąpienia, pobierania określonego wystąpienia i definiowania czasu życia obiektu. Rysunek 2-12 pokazuje podstawową strukturę Unity.



RYSUNEK 2-12 Podstawowa struktura bloku aplikacyjnego Unity

Wstrzykiwanie zależności przy pomocy Unity

W tym pierwszym przykładzie zobaczymy, jak definiować zasady dla określonego rejestratora i wykorzystywać je w swoich aplikacjach. W tym przypadku rejestrator jest deklarowany w pojemniku, a następnie przykład ten przypisuje platformie Unity odpowiedzialność za utworzenie rejestratora.

Aby to zaimplementować, trzeba zmienić kod w konstruktorze klasy Writer w celu określenia przy pomocy atrybutu *[InjectionConstructor]*, że Unity będzie odpowiadać za tworzenie tego obiektu w czasie wykonywania aplikacji.

```

/// <summary>
/// Inicjuje nowe wystąpienie klasy <see cref="Writer"/>.
/// </summary>
/// <param name="logger">Rejestrator.</param>
[InjectionConstructor]
public Writer(ILogger logger)
{
    this.logger = logger;
}

```

Teraz możemy zmienić kod, aby określić typ rejestratora, z którego chcemy skorzystać razem z typem zapisu i pozostawić obowiązek utworzenia tych obiektów platformie Unity.

```

//Przygotowanie pojemnika
var container = new UnityContainer();
//Określamy, że używanym rejestratorem ma być FileLogger
container.RegisterType<ILogger, FileLogger>();
//i jak zainicjować nowy obiekt Writer
container.RegisterType<Writer>();
//Tutaj Unity wie, jak utworzyć nowego konstruktora
var writer = container.Resolve<Writer>();
writer.Write("Some Text.");

```

Poza tym możemy korzystać z Unity do implementowania wszystkich aspektów wzorca wstrzykiwania zależności. Na przykład możemy napisać pewne zasady, które definiują, jak długo powinno istnieć jakieś wystąpienie określonego obiektu albo możemy przechwytywać tworzenie obiektów i zmieniać kod wstrzykiwany w trakcie działania programu przy użyciu określonych zachowań.

Lokalizator usług przy pomocy Unity

Inną możliwą implementacją wzorca IoC jest użycie lokalizatora usług.



UWAGA Można przeczytać w Internecie, że lokalizator usług jest antywzorcem, ponieważ rozłączanie kodu jest zbyt duże i że nie należy z niego korzystać, ponieważ może utrudniać stwierdzenie, czy nasz kod wykonuje się prawidłowo poza kontekstem środowiska uruchomieniowego, co sprawi, że kod będzie mniej podatny na testowanie. Można też przeczytać, że całkowicie tracimy kontrolę nad wstrzykiwaniem, ponieważ wynikowy kod jest bardziej rozłączony, niż przy korzystaniu z wstrzykiwania zależności. Nie zgadzam się z tym, co pokaże następny przykład.

Żeby zobaczyć, jak możemy napisać lokalizatora usług korzystając z Unity, najlepiej posłużyć się kodem przykładowym. Do wykorzystania lokalizatora usług w Unity potrzebny nam będzie adapter, który znajdziemy na witrynie CodePlex pod adresem <http://commonservicelocator.codeplex.com>. Ten adapter został zbudowany do stosowania wzorca lokalizatora usług w dowolnym spośród dostępnych pojemników IoC dla .NET, w tym Unity, Castle, Spring, StructureMap, itd.

Najpierw tworzymy prosty adapter, żebyśmy mogli skorzystać z lokalizatora usług Microsoft w połączeniu z Unity, jak pokazano w następującym fragmencie kodu:

```
/// <summary>
/// Narzędzie do konfigurowania pojemnika
/// </summary>
public sealed class UnityContainerConfigurator
{
    /// <summary>
    /// Konfiguruje wystąpienie.
    /// </summary>
    /// <returns></returns>
    public static IUnityContainer Configure()
    {
        var container = new UnityContainer()
            .RegisterType<ILogger, FileLogger>()
            .RegisterType<Writer>();
        return container;
    }
}
```

Następnie implementujemy obiekt zapisujący korzystając z lokalizatora usług zamiast z Unity (oczywiście wiadomo, że nie jest to do końca prawdą, ponieważ korzystamy z pojemnika Unity za kulisami); programista zobaczy tutaj implementację lokalizatora usług:

```
// tworzenie nowego wystąpienia pojemnika Microsoft Unity
var provider = new UnityServiceLocator(UnityContainerConfigurator.Configure());
// przypisanie pojemnika do dostawcy lokalizatora usług
ServiceLocator.SetLocatorProvider(() => provider);
// ustalenie obiektów przy pomocy lokalizatora usług
var writer = ServiceLocator.Current.GetInstance<Writer>();
writer.Write("Some Text.");
```

W tym przypadku utworzony dostawca tworzy wystąpienie i rejestruje nowy pojemnik Unity. Następnie przypisuje dostawcę do wystąpienia *ServiceLocator*, a następnie uzyskuje obiekty przy użyciu lokalizatora usług.

Jak można zauważyć, główna różnica polega na tym, że we wzorcu wstrzykiwania zależności sterujemy tworzeniem i przebiegiem kodu. Z kolei podczas korzystania z lokalizatora usług nie mamy już kontroli nad tym, jak i co jest tworzone, a po prostu wywołujemy wspólnego lokalizatora usług i uzyskujemy wystąpienie dostępnego składnika.

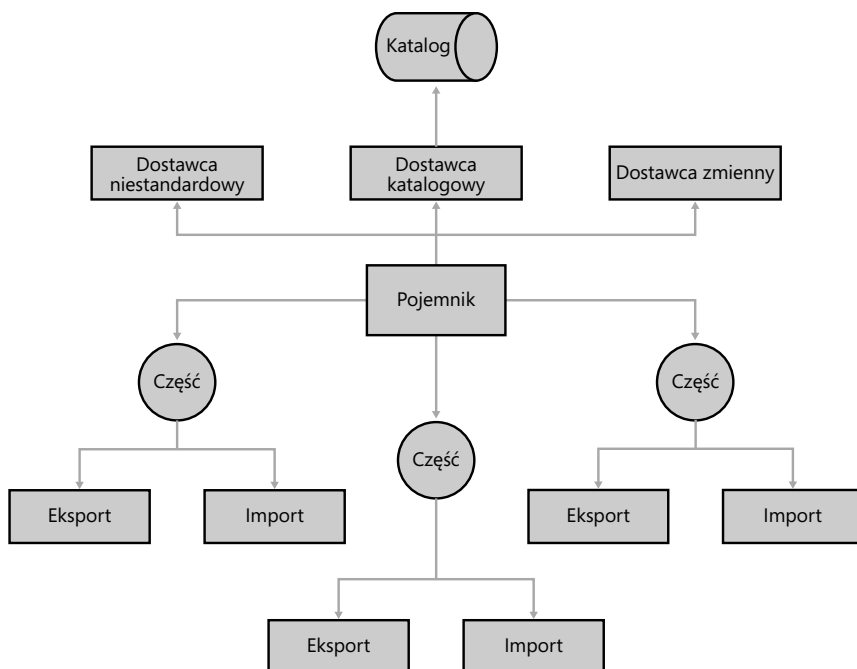
Zwykle korzystamy z lokalizatora usług w rozłączonych aplikacjach, w których programista nie ma dostępu do wspólnych składników, do których nie ma odwołań w bieżącym podzespole. Lokalizator usług pozwala pokonać tę przeszkodę.

Managed Extensibility Framework

Managed Extensibility Framework (MEF) jest blokiem aplikacyjnym stworzonym przez Microsoft Labs i wprowadzonym w .NET 3.5 jako eksperyment w fazie beta, a następnie zaimplementowanym jako część platformy .NET 4. Jest dostępny zarówno dla WPF, jak i dla Silverlight.

MEF nie jest tak naprawdę pojemnikiem IoC albo dokładniej nie jest *tylko* pojemnikiem IoC, jest platformą do zarządzania rozszerzeniami i wtyczkami. Wybór MEF nie oznacza, że nie możemy skorzystać z Unity do podstawowych operacji IoC; są to równoległe i różne bloki aplikacyjne. Ponadto Unity jest bardziej zaawansowaną platformą do wstrzykiwania zależności, ponieważ jest to jej główne zadanie, natomiast głównym celem MEF jest składanie aplikacji z komponentów.

Podobnie do Unity, MEF posiada katalog i pojemnik, które zarządzają wykrywaniem i prezentowaniem wystąpień zarejestrowanych obiektów w trakcie działania programu. Co więcej, przy pomocy MEF możemy deklarować pewne określone składniki w osobnym podzespole i rejestrować je w katalogu MEF. Możemy następnie mieć dostęp do tego katalogu z aplikacji klienckiej w trakcie działania programu i korzystać z tych składników. Główna różnica polega na tym, że MEF nie jest pojemnikiem wstrzykiwania zależności; jest rozszerzalną platformą do uzyskiwania dodatkowych składników podczas działania aplikacji. Jej głównym priorytetem jest umożliwienie rozszerzania dużych aplikacji. Rysunek 2-13 pokazuje podstawową strukturę MEF.



RYSUNEK 2-13 Struktura platformy MEF (źródło: biblioteka MSDN)

Aby skorzystać z MEF, musimy napisać niestandardowego rejestratora, który spełnia wymagania projektowe MEF. W tym przypadku będziemy chcieli określić, jak MEF ma eksportować i wykorzystywać tego rejestratora.

```
/// <summary>
/// Rejestrator dostosowany do MEF
/// </summary>
[Export(typeof(ILogger))]
public class MefLogger : ILogger
{
    /// <summary>
    /// Zapisuje komunikat.
    /// </summary>
    /// <param name="message">Komunikat.</param>
    public void WriteLog(string message)
    {
        Console.WriteLine("String built from MEF: {0}.", message);
    }
}
```

W tym momencie możemy skorzystać z programu i zadeklarować właściwość MEF. Następnie musimy ustawić wystąpienie katalogu MEF; tutaj kod deklaruje, że katalogiem jest wykonujący się podzespół. Następnie możemy łatwo korzystać ze składników.

```
/// <summary>
/// Pobiera lub ustawia obiekt zapisujący.
/// </summary>
/// <value>Obiekt zapisujący.</value>
[Import]
public ILogger Writer { get; set; }

public void Run()
{
    // najpierw budujemy katalog
    var catalog = new AssemblyCatalog(Assembly.GetExecutingAssembly)
    //tworzenie pojemnika wykorzystującego katalog
    var container = new CompositionContainer(catalog);
    container.ComposeParts(this);
    //użycie uzyskanej właściwości
    Writer.WriteLog("Mef message");
}
```

Podejście to jest tutaj znacznie różniące się od korzystania z pojemnika IoC. Korzystając z MEF przygotowujemy katalog składników, a następnie sięgamy do nich bezpośrednio. Warto zauważyć, że ten przykład w żaden sposób nie kontroluje, w jaki sposób utworzyć nowe wystąpienie rejestratora, po prostu uruchamia silnik MEF.

Różnice pomiędzy MEF a Unity

Aby podsumować tę dyskusję, chciałbym skupić naszą uwagę na tym, czym jest pojemnik IoC, czym jest platforma rozszerzająca, taka jak MEF i dlaczego powinniśmy korzystać z jednego lub drugiego rozwiązania.

Główne powody do skorzystania z Unity (lub dowolnego innego pojemnika IoC) to:

- Mamy zależności pomiędzy obiektami.
- Musimy zarządzać czasem życia obiektu.
- Chcemy zarządzać w trakcie działania programu elementami zależnymi, takimi jak pamięć podręczna, konstruktory i właściwości.
- Musimy przechwytywać tworzenie obiektu.

Główne powody do skorzystania z MEF to:

- Musimy implementować zewnętrzne rozszerzenia w swojej aplikacji klienckiej, ale możemy mieć różne implementacje w różnych środowiskach.
- Musimy automatycznie wykrywać dostępne rozszerzenia w trakcie działania programu.
- Potrzebujemy bardziej rozbudowanej i podatnej na rozszerzanie platformy niż normalna platforma wstrzykiwania zależności i chcemy pozbyć się różnych obiektów inicjujących i startowych.
- Musimy implementować rozszerzalność i/lub modułowość w swoich składnikach.

Jeśli nasza aplikacja nie wymaga żadnego z elementów wymienionych na tych listach, to prawdopodobnie nie powinniśmy implementować wzorca IoC i nie potrzebujemy korzystać z Unity i MEF.

Języki DSL: pisanie płynnego kodu

Podejście płynnego interfejsu, któremu się teraz przyjrzymy, nie jest konkretnie związane z wzorcem MVVM i nie musi być implementowane w celu uzyskania dobrych rezultatów przy pracy z wzorcem MVVM.

Z drugiej strony, ponieważ podejście to jest stosowane w tej książce, gdy mowa o MVVM i sposobie pisania niestandardowych fabryk używanych do budowania modeli widoków, to uważam, że warto poświęcić nieco czasu, aby się mu przyjrzeć, choćby po to, żeby zobaczyć, czym jest i jak działa.

Podejście języka specyficznego dla domeny (DSL) skłania do innej interesującej dyskusji na temat wzorców korporacyjnych. DSL jest techniką pozwalającą uczynić kod bardziej płynnym i czytelnym w określonym kontekście. Na przykład, kiedy piszemy zapytanie dla Microsoft SQL Server, to pracujemy z językiem specyficznym dla domeny znanym jako T-SQL; jest to język specyficzny dla domeny, ponieważ nie działa poza specyficznym kontekstem pisania zapytań dla systemu SQL Server. Głównym

celem tej techniki jest sprawienie, aby kod był bardziej czytelny wewnątrz kontekstu, gdzie ma być używany, co pomaga ograniczyć błędy i nieporozumienia.

Konieczne może być zaimplementowanie niestandardowego języka DSL w swojej aplikacji w celu uniknięcia błędów lub nieprawidłowych implementacji dokonywanych przez kolegów. Na przykład możemy mieć niewielką platformę MVVM, którą trzeba implementować w określonej kolejności i chcielibyśmy uniknąć zmian w kolejności elementów na stosie wywołań. Gdy język DSL jest budowany na użytek wewnętrzny, to nosi miano *płynnego interfejsu* – termin ten został użyty po raz pierwszy przez Martina Fowlera i Erica Evansa, gdy pisali o wzorcach korporacyjnych.

Poniższy kod pokazuje, jak można napisać płynny interfejs korzystając z C#:

```
Var mvvmView = FluentEngine
    .BuildCommands()
    .BuildData()
    .InitView()
    .Create();
```

Korzystając z normalnego podejścia można by napisać to samo w następujący sposób:

```
Var mvvmView = new MvvmView();
mvvmView.BuildCommands();
mvvmView.BuildData();
mvvmView.InitView();
```

Można się zgodzić ze mną, że pierwsza implementacja jest łatwiejsza w czytaniu. Drugą ważną kwestią jest jednak to, że pierwsza implementacja zapewnia ograniczenie. Korzystając z podejścia płynnego programista nie może zainicjować widoku przed wywołaniem inicjacji poleceń (BuildCommands) i danych (BuildData).

To podejście może być bardzo łatwe do zaimplementowania, ale musi być projektowane ostrożnie; w przeciwnym razie może się okazać, że język DSL wykorzystuje niestandardowy słownik, który nie jest zawsze czytelny i zrozumiały dla innych programistów, jak w poniższym kodzie:

```
Var mvvmView = FluentEngine
    .BuildPart01()
    .BuildPart02()
    .DoThis()
    .DoThat();
```

Oczywiście jest to ekstremalne spojrzenie na to, jak powinna wyglądać implementacja języka DSL, ale lepiej jest uzgodnić terminologię z członkami zespołu, żeby nie skończyło się na napisaniu języka DSL zrozumiałego tylko dla jego twórcy.

Pisanie płynnego interfejsu w C#

Składnia LINQ firmy Microsoft jest dobrym przykładem płynnego interfejsu. Istotą LINQ jest kolekcja *IQueryable*, której metody zawsze zwracają inną kolekcję *IQueryable*. Ułatwia to tworzenie „łańcuchów” metod, co umożliwia pisanie kodu podobnego do pokazanego poniżej:

```
Var employees = employees
    .Where(x => x.FirstName == "John")
    .Where(x => x.Age > 35)
    .OrderBy(x => x.LastName)
    .First();
```

Ten płynny kod będzie przetłumaczony przez LINQ na coś podobnego jak poniżej w języku SQL:

```
SELECT TOP 1 FROM EMPLOYEE
WHERE FIRSTNAME = 'JOHN'
AND AGE > 35
ORDER BY LASTNAME
```

Rozważmy przez chwilę, jak utworzyć obiekt. Zwykle wywołujemy konstruktora obiektu, często konstruktora bez parametrów, a następnie przypisujemy wartość do każdej właściwości, jak poniżej:

```
Var employee = new Employee();
employee.Firstname = "John";
employee.Lastname = "Smith";
employee.Age = 35;
```

To dość proste, ale co, jeśli chcemy wiedzieć przed utworzeniem obiektu, że ten obiekt będzie prawidłowy albo zapewnić, że właściwość *Employee.Age* nigdy nie będzie niższa od 30? Niestety nie możemy tego zrobić; zamiast tego musimy pamiętać o sprawdzeniu tych warunków przed użyciem utworzonego obiektu.

Gdybyśmy jednak przekształcili ten kod korzystając z wzorca fabryki (Factory) i płynnego interfejsu, to moglibyśmy zapewnić takie ograniczenia. W tym celu najpierw musimy utworzyć interfejs do definiowania kontraktów dostępnych w obiekcie DSL, na przykład:

```
public interface IFluentEmployee
{
    /// <summary>
    /// Imię
    /// </summary>
    /// <param name="firstName">Imię.</param>
    IFluentEmployee FirstName(string firstName);

    /// <summary>
    /// Nazwisko
    /// </summary>
    /// <param name="lastName">Nazwisko.</param>
```



```

    IFluentEmployee LastName(string lastName);

    /// <summary>
    /// Nazwa firmy
    /// </summary>
    /// <param name="company">Firma.</param>
    IFluentEmployee Company(string company);

    /// <summary>
    /// Tworzy to wystąpienie.
    /// </summary>
    /// <returns></returns>
    Employee Create();
}

```

Teraz możemy zaimplementować ten interfejs w niestandardowej klasie i dorobić metody statyczne, aby kod był bardziej płynny:

```

/// <summary>
/// Płynny generator obiektów
/// </summary>
public class FluentEmployee : IFluentEmployee
{
    private static Employee employee;

    private static IFluentEmployee fluent;

    /// <summary>
    /// Inicjuje nowe wystąpienie klasy <see cref="FluentEmployee"/>.
    /// </summary>
    public FluentEmployee()
    {
        fluent = new FluentEmployee();
    }

    /// <summary>
    /// Inicjuje to wystąpienie.
    /// </summary>
    public static IFluentEmployee Init()
    {
        employee = new Employee();
        return fluent;
    }

    /// <summary>
    /// Imię
    /// </summary>
    /// <param name="firstName">Imię.</param>
    public IFluentEmployee FirstName(string firstName)
    {
        employee.FirstName = firstName;
        return fluent;
    }
}

```

```

    /// <summary>
    /// Nazwisko
    /// </summary>
    /// <param name="lastName">Nazwisko.</param>
    public IFluentEmployee LastName(string lastName)
    {
        employee.LastName = lastName;
        return fluent;
    }

    /// <summary>
    /// Nazwa firmy
    /// </summary>
    /// <param name="company">Firma.</param>
    public IFluentEmployee Company(string company)
    {
        employee.Company = company;
        return fluent;
    }

    /// <summary>
    /// Tworzy to wystąpienie.
    /// </summary>
    /// <returns></returns>
    public Employee Create()
    {
        return employee;
    }
}

```

Teraz możemy napisać płynną interpretację konstruktora *Employee* w następujący sposób:

```

var employee = FluentEmployee
    .Init()
    .FirstName("John")
    .LastName("Smith")
    .Company("Microsoft")
    .Create();

```

Jest to proste zadanie z daleko idącymi konsekwencjami; pomaga zapewnić, że każdy programista, który skorzysta z tego kodu, nie pomyli tych metod. Na przykład jest dość jasne, że metoda *FirstName()* zmieni wartość właściwości *FirstName* wystąpienia klasy *Employee*.

Możemy teraz pójść dalej i przetworzyć ten kod ponownie, aby zdefiniować określoną kolejność dla tych metod lub zaimplementować wyrażenia lambda, aby sprawić, że język DSL stanie się całkowicie dynamiczny.

Niestandardową implementację składni DSL zobaczymy w rozdziale 4 i kolejnych rozdziałach. Na przykład w rozdziale 3 „Model domenowy” zobaczymy, jak zbudować niestandardową fabrykę (Factory) i niestandardowe sprawdzanie poprawności danych (Validator) korzystając z techniki DSL i wyrażen lambda.

Wprowadzenie do TDD

Programowanie sterowane testami (TDD) jest techniką tworzenia oprogramowania polegającą na napisaniu najpierw testów – nawet przed napisaniem właściwego kodu. Jest to temat, który będzie powracał w kolejnych rozdziałach tej książki; w tym podrzdziale przedstawię krótki przegląd techniki TDD.

TDD współpracuje bardzo dobrze z wzorcem MVVM ze względu na rozłączoną naturę wzorca MVVM; technika TDD nie jest obowiązkowa przy stosowaniu wzorca MVVM, ale jest bardzo zalecanym elementem.

Współgra to dobrze z główną ideą TDD, którą jest pisanie testów dla kodu przed napisaniem samego kodu. Wszelkie dane wejściowe zapewniane w tym momencie spowodowałyby niepowodzenie testu. Dopiero wtedy implementujemy kod, który przejdzie test. Na koniec przetwarzamy kod i uruchamiamy testy, aby mieć pewność, że proces refaktoringu kodu został poprawnie zaimplementowany.

Początkowo pomysł pisania testów przed napisaniem wymaganego kodu mógłby wydawać się dziwny, ale okaże się, że przy stosowaniu przy tym techniki programowania w parach uzyskujemy lepszy kod. Dla kontrastu, gdybyśmy opracowali kod najpierw, a później próbowali go testować, to znacznie trudniej jest zagwarantować, że kod jest zaimplementowany poprawnie.

Mottem TDD jest „czerwone, zielone, refaktoring”, co oznacza pisanie specyfikacji, sprawdzanie poprawności kodu względem tych specyfikacji, a następnie refaktoring kodu.

Przykład TDD

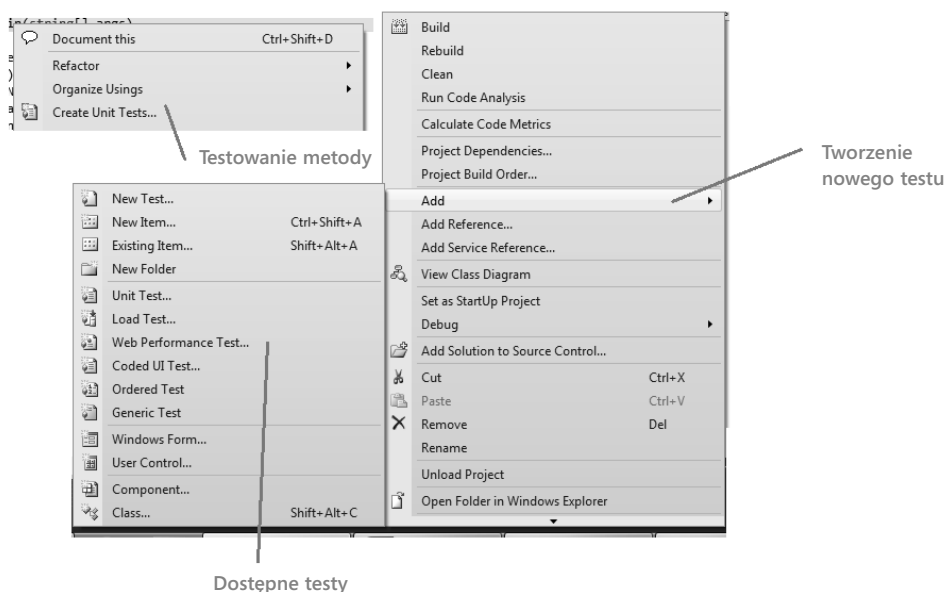
Oto przykład, który ilustruje, jak możemy pisać kod korzystając z TDD. Dla zachowania spójności ten przykład pozostaje przy dobrze nam już znanym elemencie *Employee*. Wystarczy kliknąć prawym przyciskiem myszy tekst `[TestMethod()]`, a następnie wybrać Add New Unit Test; końcowy rezultat powinien wyglądać, jak w następującym przykładzie:

```
/// <summary>
///Test dla nazwy firmy
///</summary>
[TestMethod()]
public void CompanyTest()
{
    Employee target = new Employee();
    string expected = "Microsoft";
    string actual;
    target.Company = expected;
    actual = target.Company;
    Assert.AreEqual(expected, actual);
}
```

Pokazany kod sprawdza, czy właściwość *Company* w klasie *Employee* jest poprawnie wypełniana. Oczywiście jest to prosty (i prawdopodobnie bezproduktywny) test, ale powinien dać ogólny obraz, jak działa TDD.

Narzędzia do testów jednostkowych

Istnieje kilka dobrych narzędzi do budowania testów jednostkowych. Ze względu na oszczędność miejsca ograniczę się do omówienia jedynie dwóch popularnych narzędzi. Ta książka wykorzystuje MSTest, który jest narzędziem do testów jednostkowych, które jest dostarczane z pakietem Microsoft Visual Studio 2010 i Team Foundation Server (TFS). Jest dostępne poprzez zintegrowany interfejs programistyczny Visual Studio. Rysunek 2-14 pokazuje integrację pomiędzy Visual Studio a MSTest.



RYSUNEK 2-14 Opcje dostępne dla Visual Studio i MSTest

Jeśli planujemy wykorzystanie narzędzia MSTest, to docenimy jego pełną integrację z Visual Studio i TFS, co ułatwia tworzenie kodu ze zintegrowanymi testami w procesie budowania. Inna technika programowania zwinnego (agile), zwana integracją ciągłą (CI – Continuous Integration), wymaga codziennego budowania kodu aplikacji; jedynym możliwym, bezpiecznym podejściem, aby to osiągnąć, jest zaimplementowanie metody TDD w całej aplikacji i zintegrowanie jej z procesem budowania kodu. Jeśli korzystamy z Visual Studio 2010 z TFS, to wszystkie te funkcje są dostępne w jednym środowisku.

Inną słynną platformą testową dla .NET jest NUnit, wersja przeniesiona z platformy JUnit używanej w języku Java, a wywodzącej się od narzędzia xUnit. Narzędzie NUnit jest napisane w C# i jest w pełni zgodne z .NET.

Narzędzie NUnit jest bardziej elastyczne niż MSTest i jest dostarczane zarówno w formie wiersza poleceń, jak i środowiska zintegrowanego z Visual Studio. Jeśli planujemy pracę z użyciem podejścia TDD, to warto je wypróbować. Niestety składnia pomiędzy różnymi platformami testowymi znacznie się różni, więc najlepiej wypróbować wcześniej wszystkie dostępne platformy, a następnie wybrać tylko jedną.

Dostępne zasoby dla TDD

TDD jest skomplikowaną techniką, której nie da się nauczyć i zaimplementować w kilka dni. Aby zadziałała, trzeba zaimplementować ją poprawnie. Co ważniejsze TDD wymaga dyscypliny i konsekwencji.

Aby uzyskać więcej informacji na temat TDD, polecam przeczytanie następujących książek:

- *Test-Driven Development in Microsoft .NET*, James W. Newkirk i Alexei A. Vorontsov (Microsoft Press, 2009; ISBN: 978-0-7356-1948-7)
- *The Art of Unit Testing*, Roy Osherove (Manning, 2009; ISBN: 978-1-933988-27-6)

Przydatną listę zasobów dotyczących TDD można też znaleźć na witrynie MSDN pod adresem [http://msdn.microsoft.com/en-us/library/aa730844\(VS.80\).aspx](http://msdn.microsoft.com/en-us/library/aa730844(VS.80).aspx) lub na następującej witrynie społeczności programistów TDD: <http://www.testdriven.com>.

W tej książce często znajdziemy odniesienia do technik TDD razem z wyjaśnieniami, jak poprawnie testować omawiane tutaj implementacje wzorca MVVM.

Podsumowanie

Wzorzec projektowy jest zestawem wskazówek opisujących typowe rozwiązanie typowego problemu, który może być zaadaptowany do określonego przypadku. Typowe wzorce projektowe są znane jako wzorce projektowe Gang of Four i dzielą się na trzy kategorie: kreacyjne, strukturalne i behawioralne. Ta klasyfikacja jest oparta na typie problemu, który dany wzorzec projektowy stara się rozwiązać.

Dodatkowa kategoria wzorców projektowych obejmuje te, które są używane dla interfejsów użytkownika; w tej kategorii istnieją cztery główne wzorce: MVC, MVP, PM i MVVM. Podczas gdy wzorce MVC i MVP są bardziej ogólne i elastyczne, to MVVM jest szczególnie zaprojektowany dla technologii WPF i Silverlight.

Jeszcze jedna kategoria wzorców projektowych jest znana jako wzorce projektowe dla przedsiębiorstwa albo architektoniczne wzorce projektowe. Martin Fowler i Eric Evans stworzyli klasyfikację tych wzorców i wyjaśnili je w książce *Patterns of Enterprise Application Architecture*. Wzorzec IoC jest tylko jednym z tych wzorców

korporacyjnych. Jest przydatny do przenoszenia zależności do obiektu wywoływanego spoza obiektu wywołującego.

Testowanie aplikacji jest istotne dla uniknięcia błędów. Jeśli od początku zaimplementujemy techniki TDD, to możemy zagwarantować, że aplikacja będzie spełniać wymagania projektowe i że kod zostanie przetestowany przed przeniesieniem aplikacji do środowiska produkcyjnego.

Model domenowy

Po zakończeniu tego rozdziału będziemy w stanie:

- Zrozumieć techniki projektowania sterowanego domeną.
- Utworzyć mechanizm sprawdzania poprawności dla modelu domenowego.
- Utworzyć przykładowy model domenowy.

Wprowadzenie do projektowania sterowanego domeną

Kluczową rolą oprogramowania jest rozwiązywanie problemów i spełnianie wymagań. Oczywiście można to osiągać na różne sposoby. Jednym ze sposobów jest wykorzystanie projektowania sterowanego domeną (DDD – Domain-Driven Design). Przy pomocy DDD staramy się rozwiązać problem biznesowy, który charakteryzuje model domenowy poprzez utworzenie zestawu jednostek domenowych reprezentujących różne biznesowe części aplikacji.

Korzystając z techniki DDD piszemy aplikację, która ma solidne podstawy oparte na podejściu zorientowanym obiektowo. Budujemy kod wokół jednostek biznesowych, które składają się na domenę biznesową, a następnie adaptujemy go, żeby spełniał relacje biznesowe pomiędzy jednostkami i ich zachowaniami.

DDD jest zbiorem metodologii i technik stosowanych w określonym kontekście, więc implementacja może być bardzo różna w różnych aplikacjach. Głównym celem aplikacji DDD jest skupienie się na rozumieniu i modelowaniu domeny (wymagań biznesowych), co jest możliwe tylko wtedy, gdy zespół projektowy ma już dogłębną wiedzę na temat wymagań biznesowych. Z tego i innych powodów DDD można zwykle osiągnąć tylko wtedy, gdy zespół pracuje równolegle z grupą analityków, którzy znają już wymagania biznesowe aplikacji. Jeśli zespół zdecyduje się na zastosowanie techniki DDD, to w zasadzie godzimy się na zdefiniowanie wspólnego języka skupiającego się na modelu domenowym zaprojektowanym dla aplikacji, który ograniczy przepaść językową pomiędzy analitykami, architektami i programistami. W istocie, gdy zaczniemy tworzyć aplikację, w którą zaangażowani są ludzie z różnym doświadczeniem (jak

wspomniani wcześniej analitycy, architekci lub programiści), to odkryjemy, że każdy zwykle będzie definiował tę samą rzecz używając innej terminologii. Technika DDD powinna pomóc w znalezieniu wspólnego języka.

Ponieważ podejście DDD działa jako kanał komunikacyjny pomiędzy członkami projektu, istotne jest, aby język zdefiniowany dla modelu domenowego był jasny i jednoznaczny. Jeśli model domenowy jest dobrze zdefiniowany, a język domenowy jest jasny i odzwierciedla zachowania i relacje w domenie, to również logika biznesowa całej domeny będzie jasna i zrozumiała – dla wszystkich członków projektu niezależnie od tego, czy są analitykami, architektami, czy programistami. Język, który tworzymy korzystając z modelu domenowego, powinien nam pozwolić w znalezieniu luk i błędów w modelu, ponieważ jest on jedynym pomostem pomiędzy nami a modelem.

Jednym z podstawowych wymagań DDD jest odizolowanie domeny od reszty aplikacji; musimy utrzymywać model domenowy jako czystą konstrukcję językową dla swojego problemu domenowego. Jasnym jest więc, że podejście DDD wymaga dużo dodatkowego wysiłku i powinniśmy je brać pod uwagę tylko wtedy, gdy problemy domenowe są odpowiednio skomplikowane, a aplikacja jest odpowiednio duża. Chodzi mi o to, że prawdopodobnie nie powinniśmy brać pod uwagę podejścia DDD w przypadku bardzo małych aplikacji ze względu na jego wysoki koszt. Mimo to osobiście zawsze korzystam z podejścia DDD nawet w przypadku bardzo małych domen składających się tylko z dwóch lub trzech jednostek domenowych. Sądzę, że podejście DDD daje moim aplikacjom duży poziom elastyczności w przypadku przyszłej rozbudowy.

Jako podsumowanie tego krótkiego wprowadzenia do podejścia DDD przedstawiam tutaj główne zalety stosowania techniki DDD w swojej aplikacji:

- **Wspólny język** Jeśli zdefiniujemy wspólną domenę dla swojej aplikacji, to utworzymy wspólny język używany w ten sam sposób i w tych samych znaczeniach przez wszystkich członków zespołu.
- **Rozszerzalność** Podejście DDD pozwala nam utworzyć rozszerzalną aplikację, ponieważ domena jest jądrem aplikacji – a z definicji domena jest rozszerzalna i luźno powiązana z resztą kodu, więc rozszerzanie jej i implementowanie nowych funkcji w istniejącym modelu domenowym powinno być dość łatwe.
- **Łatwość testowania** Aplikacja DDD jest z definicji łatwa do testowania.

Terminologia DDD

Do zrozumienia domeny i stworzenia jej modelu potrzebne nam wprowadzenie do typowej terminologii używanej w tym zestawie technik tak, abyśmy mogli zapoznać się ze strukturą DDD.

- *Domena* jest zestawem działań, wiedzy i kontekstów, w których rozwijana jest aplikacja; jest specyficzna dla kontekstu biznesowego aplikacji.

- *Model* jest częścią domeny. Zwykle reprezentuje określony zestaw aspektów związanych z domeną i składa się ze zbioru jednostek oraz obiektów wartości.
- *Jednostka* jest unikalnym obiektem reprezentowanym w domenie przez jednostkę domenową. Jednostki domenowe są unikalne i nie zmieniają się, gdy zmienia się stan aplikacji. Jednostka obejmuje właściwości, zachowania i stany. Obiekt *Customer* w przykładowej aplikacji CRM jest jednostką domenową.
- *Obiekt wartości* jest obiektem używanym do opisanego jakiegoś aspektu domeny, który jest niezmienny i nie ma unikalnej tożsamości w domenie. Na przykład jednostka *Customer* może mieć listę elementów typu *Address*; z których jeden jest obiektem wartości, ponieważ jest używany do opisanego adresu jednostki domenowej typu *Customer*.
- *Zbiórce jednostki nadrzędne* są jednostkami nadrzędnymi używanymi do sterowania relacjami pomiędzy jednostkami podrzędnymi lub podrzędnymi obiektami wartości. Zwykle sterują dostępem do tych obiektów podrzędnych i/lub interakcjami pomiędzy nimi.
- *Wszechobecny język* jest językiem skonstruowanym wokół domeny, którego programiści i analitycy używają do określania poszczególnych aspektów domeny.
- *Kontekst* oznacza świat, w którym dany model może istnieć.

Analizowanie domeny aplikacji CRM

Gdy mamy już definicje za sobą, możemy zacząć od „historii użytkowników”, które reprezentują przykładową aplikację CRM.

UWAGA Historia użytkownika opisuje wymaganie, zadanie lub część procesu biznesowego, które będą przeprowadzane przez użytkownika podczas korzystania z aplikacji. Opisuje proces biznesowy w sposób zrozumiały zarówno dla użytkowników, jak i programistów.



Projekt tej aplikacji będzie sterowany domeną (co będziemy dogłębnie analizować w tym rozdziale), więc domena jest pierwszym składnikiem, który musimy zaprojektować. Projektowanie domeny na początku jest typowym podejściem, które będziemy stosować przy tworzeniu aplikacji MVVM korzystając z techniki DDD do budowy warstwy domenowej.

Historia użytkownika jest szkicem obszaru biznesowego; opisuje, jak różne elementy domeny współpracują ze sobą i jak będą przeprowadzane określone zadania lub procesy biznesowe. Zwykle w aplikacji występuje kilka historii użytkowników.

Przykładowa aplikacja CRM, którą będziemy budować w tej książce, składa się z kilku historii użytkowników, które zostały zebrane poniżej:

„Jako pracownik chcę być w stanie dodawać i zarządzać klientami”.

„Jako pracownik chcę też być w stanie zarządzać zamówieniami złożonymi przez klienta”.

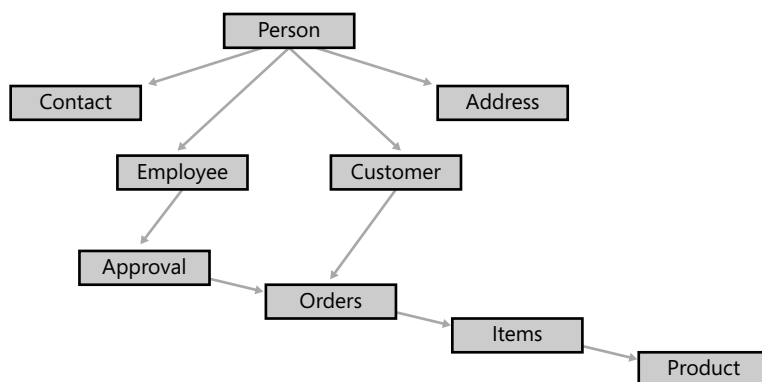
„Jako pracownik chcę też sprawdzać, czy określony produkt zamówiony przez klienta jest dostępny w magazynie”.

„Jako klient chcę móc zamówić dowolny dostępny produkt”.

To dość proste. Szkoda, że te jasne i pełne stwierdzenia nie dają wielu wskazówek programistom. Najpierw więc musimy wyciągnąć kluczowe pojęcia z tej historii użytkownika.

Stwierdzenia te wskazują na jedną domenę, którą nazwiemy CRM.Domain i cztery główne jednostki: *Employee* (pracownik), *Customer* (klient), *Order* (zamówienie) i *Product* (produkt). Będziemy musieli dodać dalsze składniki do tych głównych elementów modelu, takie jak *Address* (adres), *Contact* (kontakt), *OrderLine* (wiersz zamówienia), itd.

Korzystając z kartki papieru i ołówka do utworzenia zgrubnego modelu uzyskamy coś podobnego, jak na rysunku 3-1. Jest to pierwszy szkic modelu domenowego. W dalszej części tego rozdziału zobaczymy, jak zbudować i przetworzyć każdą część tej domeny i wykorzystać ją w aplikacji MVVM.



RYSUNEK 3-1 Szkic modelu domenowego aplikacji CRM

Możemy porównać ten diagram z zestawem prostych obiektów zamodelowanych w języku C# lub Visual Basic .NET, gdzie każdy kształt odpowiada jednej lub kilku klasom z relacjami, które mogą być właściwościami, obiektami złożonymi, kolekcjami obiektów podrzędnych lub odwołaniami do obiektów nadrzędnych.

Jednostka domenowa i obiekt transferu danych

Zacznijmy od definicji jednostki domenowej i obiektu transferu danych (DTO – Data Transfer Object), żeby zapamiętać znaczenie tych pojęć. Można znaleźć inne definicje niż zastosowane przeze mnie, ale pojęcia, które tutaj przedstawiam, są podstawowymi pojęciami DDD, a nie jedynie osobistą interpretacją.

Jednostka domenowa, znana również jako obiekt domenowy, jest pojęciem modelu domenowego, które reprezentuje unikalność elementu w domenie biznesowej i utrzymuje stan tego elementu. Na przykład w modelu domenowym aplikacji CRM jednostka *Employee* jest jednostką domenową.

DTO jest prostym obiektem – może być serializowany i używany do transferu danych pomiędzy warstwami, obiektami i/lub poziomami. Nie ma żadnej logiki biznesowej i zwykle nie ma żadnych cyklicznych odwołań do obiektów nadrzędnych lub podrzędnych. Architekci oprogramowania, tacy jak Martin Fowler stosują termin *obiektu wartości* do zdefiniowania prostego obiektu w modelu domenowym, który nie ma określonej tożsamości.

Pojęcie obiektu DTO jest obowiązkowe w DDD z następującego powodu. Wyobraźmy sobie, że aplikacja MVVM ma proste pole ComboBox w XAML, które chcemy wypełnić wszystkimi dostępnymi pracownikami w bazie danych. Moglibyśmy łatwo uzyskać coś takiego:

```
// Kod C# pobierający dane
Var employees = dataLayer.GetAllEmployees(); // to zwraca typ IList<Employee>
// Pseudo XAML
<combobox
    ItemSource="{Binding Path=employees}"
    DisplayMemberPath="FirstName"
    SelectedValuePath="Id" />
```

W powyższym kodzie dużym problemem jest to, że naprawdę wiążemy całą jednostkę pracownika. Choć ten kod wykorzystuje tylko imię i identyfikator, to w rzeczywistości przechowujemy w pamięci cały obiekt. Na przykład, gdyby jednostka *Employee* miała listę jednostek *Address* zamapowaną jako właściwość, to kod przechowywałby również tę listę w pamięci.

Rozwiązaniem jest spłaszczenie tego obiektu przy użyciu obiektu DTO, który będzie reprezentował tylko dane potrzebne w danym momencie. Jednostka może mieć jeden lub wiele obiektów DTO w zależności od kontekstu. Poniższy przykład wykorzystuje rozszerzenie LINQ do utworzenia nowej listy obiektów DTO zaczynając od listy jednostek.

```
Public class EmployeeDto
{
    string FirstName { get; set; }
    Guid Id { get; set; }
}
```

```
// Kod C# pobierający dane
Var employees = dataLayer
    .GetAllEmployees()
    .Aggregate(new List<EmployeeDto>() => (list, obj)
    {
        var dto = new EmployeeDto { FirstName = obj.FirstName, Id = obj.Id };
        list.Add(dto);
        return list;
    });

// Pseudo XAML
<combobox
    ItemSource="{Binding Path=employees}"
    DisplayMemberPath="FirstName"
    SelectedValuePath="Id" />
```

Trzeba rozważyć dwie dalsze rzeczy związane z jednostkami i obiektami DTO. Po pierwsze, trzeba zauważyć, że chociaż logika biznesowa nie była omawiana, to powinna być zawarta w jednostkach domenowych i usługach domenowych. W przyszłych rozdziałach zobaczymy dlaczego. Druga kwestia dotyczy sposobu, w jaki możemy zamapować jednostkę na obiekt DTO i vice versa. Zobaczymy, jak skorzystać z refleksji lub emisji do utworzenia prostego automatycznego składnika mapującego dla swoich aplikacji i jak korzystać z istniejących narzędzi takich, jak Auto-Mapper (<http://automapper.codeplex.com>) albo Emit Mapper (<http://emitmapper.codeplex.com>).

Oto krótkie podsumowanie tego, czym jest jednostka domenowa razem z jej charakterystykami i ograniczeniami:

- Jednostka domenowa powinna być implementowana bez świadomości, jak będzie zachowywana w bazie danych albo kiedy powinna być zachowywana. Chcemy mieć możliwość wykorzystania domeny w wielu aplikacjach i wielu typach magazynów danych.
- Jednostka domenowa reprezentuje określony problem w domenie, ale nie jest obiektem biznesowym; obejmuje jedynie wymaganą logikę biznesową, nic więcej. Jeśli chcemy dodać logikę biznesową do jednostki domenowej, powinniśmy rozważyć zbudowanie konkretnego obiektu biznesowego (jest to omówione w przyszłych rozdziałach).
- Jednostka domenowa powinna być świadoma swoich ograniczeń w odniesieniu do innych jednostek domenowych dostępnych w tej samej domenie. Powinna wykorzystywać jasną konwencję nazewnictwa i powinna odzwierciedlać wszechobecny język wykorzystując jedynie wbudowane właściwości i metody jednostek.

Obiekt POCO i O/RM

W poprzednim podrozdziale zobaczyliśmy, że unikalną rolą jednostki domenowej jest zajęcie się określonym obszarem lub aspektem w domenie. To pojęcie jest jasne, ale daleko odbiega od rzeczywistości prawdziwej aplikacji. Na podstawie definicji

i paradygmatów DDD jednostka domenowa powinna być zwykłym obiektem (POCO – Plain Old CLR Object) albo raczej obiektem, który nie wie nic o swoim miejscu przechowywania i nie dziedziczy po określonych klasach.

Wracając do poprzedniego przykładu można było zauważyć, że dodałem właściwość *ID* do jednostki *Employee*, żeby zaznaczyć jej unikalność w kolekcji takiej, jak tabela bazy danych. Bez identyfikatora nie bylibyśmy w stanie zidentyfikować zaznaczonej jednostki *Employee* w kontrolce *ComboBox*. W projektowaniu DDD jednostka domenowa powinna być zwykłym obiektem POCO (.NET) lub POJO (Java). W mojej opinii ta zasada działa dobrze tylko na poziomie abstrakcji; nie jest do zrealizowania w praktyce.

Chcę też podkreślić znaczenie tożsamości dla jednostki domenowej. Załóżmy, że mamy dwie jednostki *Employee* zdefiniowane następująco:

```
Var employee = new Employee { FirstName = "John", LastName = "Smith", Age = 54 }  
Var employee = new Employee { FirstName = "John", LastName = "Smith", Age = 23 }
```

Bez dołączenia jakiegoś rodzaju unikalnego identyfikatora (*UniqueId*) do każdej z nich nie bylibyśmy w stanie ich rozróżniać. Nie jest to nierealny przykład; jest wielce prawdopodobne, że organizacja może mieć dwóch różnych pracowników o tym samym imieniu (*FirstName*) i nazwisku (*LastName*), ale w różnym wieku i zajmujących różne stanowiska.

Żeby pracować z systemem O/RM, takim jak Entity Framework lub NHibernate (jak w tej książce), z założenia konieczne będzie dodanie ograniczeń do takich jednostek, które uczynią je unikalnymi w modelu. Dlatego musimy nadać im tożsamość – tak jak robilibyśmy to w przypadku wierszy tabeli korzystając z klucza głównego. To wymaga nie oznacza, że jednostki nie są obiektami POCO, ale narusza nieco prosty projekt obiektu POCO.

Inna kwestia dotyczy braku świadomości odnośnie przechowywania, co ma miejsce, gdy klasy i otaczające je warstwy aplikacji nie znają lub nie interesują się sposobem przechowywania swoich danych. Na przykład w wersji Entity Framework dla .NET 3.5, jeśli chcieliśmy użyć wcześniej istniejących klas, to musieliśmy je modyfikować tak, aby wywodziły się z klasy *EntityObject*. W wersji .NET 4 nie jest to już konieczne. Nie musimy modyfikować swoich jednostek, aby mogły one w pełni uczestniczyć w operacjach Entity Framework. Pozwala nam to budować aplikacje, które stosują luźne powiązania i podział interesów. Dzięki tym wzorcom kodowania nasze klasy zajmują się tylko swoimi własnymi zadaniami. Wiele warstw aplikacji, w tym interfejs użytkownika, nie zależy od logiki zewnętrznej, takiej jak interfejsy Entity Framework API, jednak te zewnętrzne interfejsy API nadal są w stanie współdziałać z naszymi jednostkami.

Podsumowując pojęcie obiektów POCO (POJO) jest zgrabne i jasne w projektowaniu DDD, ale nierealistyczne w rzeczywistych aplikacjach. Jak wspominałem wcześniej, trzeba pamiętać, że są to wskazówki, a nie zasady, więc należy stosować je, gdy to możliwe, a następnie adaptować kod tak, żeby spełniał nasze konkretne potrzeby.

Podejścia do projektowania domeny

Książka Martina Fowlera *Patterns of Enterprise Architecture Application* (PoEAA) wymienia trzy różne podejścia do projektowania domeny.

Biorąc pojęcie domeny jako ogólną definicję Fowler twierdzi, że możemy tworzyć aplikację korzystając z jednego z trzech dostępnych wzorców dla domeny: skryptu transakcyjnego, rekordu aktywnego i modelu domenowego.

Podejście DDD opisane w tej książce wykorzystuje podejście modelu domenowego, ale w przypadku prostszych aplikacji możemy rozważyć użycie podejścia rekordu aktywnego, a jeśli jedynie musimy napisać sekwencyjny zbiór poleceń, to prawdopodobnie będziemy chcieli użyć podejścia skryptu transakcyjnego. Warto zbadać, dlaczego i kiedy powinniśmy skorzystać z każdego z tych wzorców.

Skrypt transakcyjny

Podejście skryptu transakcyjnego jest często używane przez mniej zaawansowanych programistów w sytuacjach takich, jak pierwszy projekt młodego programisty albo prosty skrypt narzędziowy. Głównym zadaniem skryptu transakcyjnego jest zorganizowanie całej logiki w postaci pojedynczej procedury sięgającej do bazy danych bezpośrednio lub poprzez cienki obiekt dostępu do bazy danych. Każda transakcja będzie miała swój własny skrypt transakcyjny, choć wspólne podzadania mogą być wydzielone jako podprocedury.

Na przykład konieczne może być napisanie funkcji, która będzie wypisywać listę dostępnych pracowników. Aby to zrobić przy użyciu podejścia skryptu transakcyjnego, napisalibyśmy kod podobny do pokazanego poniżej, gdzie połączenie, instrukcje SQL i kod C# są wymieszane razem w pojedynczym kroku.

```
Var connection = new SqlConnection();
var command = new SqlCommand(connection, "SELECT * FROM EMPLOYEE");
var reader = command.ExecuteReader();
Connection.Open();
while(reader.Read())
{
    Console.WriteLine("Employee: {0} - {1}" reader["FirstName"], reader["LastName"]);
}
// koniec pseudokodu...
```

Można przyznać, że ten fragment kodu da się szybko napisać oraz łatwiej go czytać i zmieniać, niż w przypadku warstwowego podejścia do aplikacji, ale jest powtarzalny i bardzo trudny do utrzymania. Jest tak dlatego, ponieważ jeśli później zdecydujemy się, że musimy wykonać to samo zapytanie w innej funkcji, to będziemy musieli skopiować (lub przepisać) ten sam kod – i za każdym razem, gdy trzeba będzie coś zmienić, to będziemy musieli dokonać tej zmiany w każdym fragmencie kodu, który wykorzystuje tę instrukcję SQL. Ponadto kod ten jest trudny do testowania, ponieważ logika bazodanowa i kod są całkowicie powiązane ze sobą bez żadnej logiki architektonicznej.

Zastosowanie podejścia skryptu transakcyjnego nie daje się obronić, gdy kod pojedynczej transakcji (procedura) staje się zbyt skomplikowany. W końcu trzeba będzie rozbić kod na mniejsze zestawy transakcji, które będą wywoływane sekwencyjnie przez główne zadanie. W tym momencie zaczniemy całkiem tracić kontrolę nad swoją aplikacją. Za każdym razem, gdy wykryjemy błąd, jedynym rozwiązaniem będzie testowanie całości.

Po prostu nie polecam tej techniki w żadnej sytuacji – za wyjątkiem takiej, gdy mamy sekwencyjne kroki, które trzeba wykonać i przebieg tych sekwencyjnych kroków nie będzie się nigdy zmieniał podczas ewolucji oprogramowania, jak w następującym pseudokodzie:

```
Public void ApproveOrder()
{
    VerifyOrder(order);
    VerifyCustomer(customer);
    AssignOrder(employee, order);
    ApproveOrder(order);
}
```

W tym przypadku podana sekwencja zadań jest wymogiem zaakceptowania jednostki domenowej Order (zamówienie). Najprawdopodobniej ten przepływ zadań nie będzie się nigdy zmieniał, więc możemy z powodzeniem zastosować podejście skryptu transakcyjnego w tej sytuacji. Warto przy tym zauważyć, że ten przykład jest jedynie sekwencją metod; nie osadza wywołań SQL albo wywołań zmieniających interfejs użytkownika w pojedynczej metodzie.

Podjęcie sterowane bazą danych

Programiści często z wielu powodów nie mają możliwości zaprojektowania aplikacji od początku. Na przykład możemy mieć powierzone zadanie przepisania istniejącego systemu, w którym nie można zmienić bazy danych i trudno jest zaprojektować domenę pasującą do istniejącej bazy danych. Możemy też mieć ograniczone zasoby, a czas życia aplikacji będzie krótki, jak w przypadku narzędzia, które będzie działało przez kilka miesięcy. Te przypadki nie sprzyjają pisaniu skomplikowanych wielopoziomowych systemów, których opracowanie prawdopodobnie zajmie więcej czasu niż oczekiwany czas życia aplikacji.

Podjęcie sterowane bazą danych wymusza na nas przyjęcie wzorca rekordu aktywnego. W tym wzorcu głównym graczem w aplikacji jest baza danych i projektujemy domenę tak, aby odzwierciedlała strukturę tej bazy danych. Dlatego będziemy mieli jednostkę domenową dla każdej tabeli w bazie danych, a potrzeby bazy danych będą sterować przebiegiem aplikacji.

To podejście nie jest błędne – zwłaszcza, jeśli dopiero zaczęliśmy korzystać z podejścia DDD, ale jeszcze w pełni go nie opanowaliśmy. Wiele systemów O/RM, takich jak Entity Framework lub NHibernate, oferuje możliwość tworzenia domeny rekordu aktywnego bez utraty możliwości modelu relacyjnego oraz warstwy danych; niestety

to podejście jest wciąż bardzo odległe od bardziej solidnego i skomplikowanego podejścia DDD.

W tym przypadku, ponieważ obiekt (jednostka) jest lustrzanym odbiciem tabeli w relacyjnej bazie danych, to sam obiekt musi odpowiadać za aktualizowanie swojego statusu w bazie danych i musie też mieć świadomość, jak jest zapisywany i pobierany z tej bazy danych, więc zwykle zobaczymy kod podobny do następującego:

```
Var employee = New Employee().Get(1);  
employee.FirstName = "John";  
employee.Save();
```

Powiedziałbym, że jeśli musimy napisać prostą aplikację do wprowadzania danych, która nie zawiera żadnej skomplikowanej logiki biznesowej ani transakcji danych, to wzorzec rekordu aktywnego jest prawdopodobnie dobrym punktem wyjścia. Pamiętajmy jednak, że będzie to zarówno punkt wyjścia, jak i punkt docelowy; ten wzorzec nie może być rozszerzany lub zmieniany jak DDD. Innym problemem z tym wzorcem jest to, że ponieważ jednostki mają pełną kontrolę nad procesem zachowywania danych, to programista może łatwo napisać kod, który nieprawidłowo usunie lub zmieni istniejący rekord.

W mojej opinii powinniśmy unikać tego wzorca przy projektowaniu skomplikowanych i rozszerzalnych aplikacji biznesowych, ponieważ:

- Obsługa różnych wersji rekordu w przypadku wzorca rekordu aktywnego jest nonsensownym podejściem. „Aktywną” jednostką jest wiersz w bazie danych, więc jeśli chcemy utrzymywać historię zmian danych, to musimy klonować rekord aktywny zawsze, gdy ulega zmianie, zachowując starszą wersję i zastępując ją nową wersją.
- Nie możemy oddzielić stanu i zachowania, ponieważ jednostka odpowiada za zachowywanie swojego stanu, a także utrzymuje swoją strukturę danych. Innymi słowy, w przypadku rekordu aktywnego stan jednostki i zachowanie są pomieszczone ze sobą, jak w przykładzie polecenia *Save()* albo właściwości *IsNew*.
- Brakuje podziału interesów i trudno jest testować kod. Z jednej strony jednostka działa tylko, jeśli dostępna jest baza danych, więc nie możemy testować samej jednostki bez testowania również bazy danych – a to łamie jedną z zasad programowania sterowanego testami (TDD), ponieważ testy nie mogą być odpowiednio niezależne.

Podejście sterowane domeną

Podejście sterowane domeną jest konkretną implementacją techniki DDD. W tym przypadku mamy domenę, która steruje całą aplikacją. Domena jest całkowicie nieświadoma bazy danych. Aby wszystko uprościć, możemy skorzystać z systemu O/RM, takiego jak Entity Framework lub NHibernate, żeby pomóc w utworzeniu kodu SQL zachowującego jednostki w bazie danych.

DDD jest najbardziej skomplikowanym podejściem, ponieważ wymaga więcej czasu, więcej testów i więcej elastyczności, a także ponieważ wymaga dogłębnej znajomości procesów biznesowych. Koniec końców jest też najbardziej elastycznym i łatwym w utrzymaniu podejściem, ponieważ domena i baza danych nie są blisko powiązane. Załóżmy, że za miesiąc nasza firma postanowi przejść z bazy SQL Server na MySQL. Dzięki korzystaniu z podejścia DDD będziemy musieli zmienić jedynie łańcuch połączenia w swoim systemie O/RM. Albo załóżmy, że zmieni się kolejność działań w procesie zatwierdzania zamówienia z powodu jakichś restrykcyjnych zasad wprowadzonych w departamencie księgowości. Dzięki DDD będziemy jedynie musieli zidentyfikować ten proces w modelu i go zaktualizować.

W przypadku podejścia sterowanego domeną zwykle pracujemy z systemem O/RM i mamy do czynienia z „jednostką pracy” (), która jest składnikiem utrzymującym sesję bazodanową przy życiu tak, że możemy wykonywać na niej operacje tworzenia, odczytu, aktualizowania i usuwania (CRUD – Create, Read, Update & Delete). Przeanalizujemy ten wzorzec szczegółowo w następnym rozdziale, gdy będziemy budować warstwę danych dla przykładowej aplikacji CRM. Wykorzystując obiekt *UnitOfWork* i podejście DDD wykonalibyśmy kod widziany wcześniej w podrozdziale dotyczącym skryptu transakcyjnego w następujący sposób:

```
Var employees = unitOfWork.Get<Employee>();
foreach (var employee in employees)
{
    Console.WriteLine("Employee: {0} {1}", employee.FirstName, employee.LastName);
}
```

We wprowadzeniu do rozdziału zobaczyliśmy, dlaczego powinniśmy korzystać z podejścia sterowanego domeną i dlaczego jest to bardziej elastyczne niż inne podejścia. Na przyszłość należy pamiętać o następujących stwierdzeniach:

- Technika DDD jest zorientowana na biznes, więc nie musimy znać wszystkich dostępnych tabel w bazie danych, aby zatwierdzać transakcję biznesową w kodzie; domena „mówi za siebie” i jest dobrze zrozumiała zapewniając ogólny język.
- Domena jest rozszerzalna i może być wielokrotnie wykorzystywana, ponieważ jedynym ograniczeniem jest biznes z nią związany i nic innego (takiego, jak baza danych lub platforma programistyczna).
- Domena (sedno biznesu) nie jest ograniczona do określonej technologii. Jeśli będzie nam potrzebna w innej aplikacji, możemy po prostu dodać ją jako składnik i z niej korzystać.
- Jest łatwa w testowaniu; możemy testować domenę, zanim będziemy mieli gotową do testowania jakąkolwiek bazę danych albo interfejs użytkownika. Możemy ją też cały czas rozwijać lub zmieniać i ponownie testować podczas ewolucji naszej aplikacji.

Jak utworzyć obiekt w DDD

W składni programowania zorientowanego obiektowo (OOP) zwykle tworzymy nie-statyczne obiekty korzystając ze słowa kluczowego *new*. Ten proces jest podobny, ale nieco się różni w językach C#, Visual Basic .NET lub Java. Gdy musimy utworzyć nowe wystąpienie obiektu, to po prostu tworzymy go przy pomocy słowa kluczowego *new*, a następnie zaczynamy z niego korzystać zmieniając na przykład wartości jakichś właściwości lub wywołując określoną metodę.

Korzystając z metody konstruktora mamy dwie opcje do wyboru; możemy utworzyć konstruktora bez parametrów albo konstruktora z ograniczeniami, który wymusi na programistach podawanie określonych wartości podczas tworzenia obiektu.

```
//bez parametrów
var employee = new Employee();
employee.FirstName = "John";
employee.LastName = "Smith";
employee.Age = 54;

//konstruktor z parametrami
var employee = new Employee("John", "Smith", 54);
```

Obie metody działają dobrze. Pierwsza jest prawdopodobnie bardziej „otwarta”, ponieważ nie wymusza na programistach określania czegokolwiek – mogą oni po prostu utworzyć nowy, pusty obiekt *Employee*. Druga metoda jest bardziej sterowana danymi, ponieważ wymaga określonych parametrów *FirstName*, *LastName* i *Age* do działania (możemy też skorzystać z parametrów opcjonalnych w C# 4, żeby pomijać podawanie tego ostatniego, ale w tym przypadku mogą zostać ustawione wartości domyślne, które mogą być nieprawidłowe dla danego typu jednostki).

Gdy zaczynamy pracę z MVVM, to pierwszym problemem z architektonicznego punktu widzenia, który napotkamy, jest to, że aplikacja jest warstwowa, więc programista, który napisał model domenowy, może nie być tym samym, który będzie z niego korzystał w modelu widoku albo warstwie biznesowej. Z tych powodów musimy zastosować jakieś ograniczenia określające, jak tworzyć nową jednostkę domenową, aby zapewnić, że ta jednostka będzie prawidłowa i poprawnie tworzona.

Wzorzec fabryki omówiony w rozdziale 2 „Wzorce projektowe” jest wzorcem kreacyjnym zaprojektowanym w celu sterowania tworzeniem określonego obiektu. Wzorzec fabryki jest obowiązkowy w aplikacji DDD, ponieważ zmusza programistów do tworzenia jednostek przy użyciu określonych zasad i obsługuje TDD. Poniższy przykład pokazuje, dlaczego:

```
//fabryka dla obiektu Employee
var employee = Factory.CreateEmployee("John", "Smith", 54);

// wyrzucenie wyjątku, ponieważ wiek nie może być mniejszy niż 1
var employee = Factory.CreateEmployee("John", "Smith", 0);

//tutaj mamy pewną logikę biznesową, której nie musimy znać
var employee = Factory.CloneEmployee(anotherEmployee);
```

Powyższy kod wymusza na programistach tworzenie nowego obiektu *Employee* poprzez dostępne metody fabryki; jeśli programista wprowadzi na przykład nieprawidłowy wiek, to fabryka po prostu wyrzuci wyjątek. To samo odnosi się do procesu klonowania; użycie metody fabryki zmusza programistę do klonowania pracownika w określony sposób sterowany logiką domenową.

Wzorce fabryki

Jeśli planujemy użycie metody fabryki w swojej domenie, to zagwarantujemy sobie ograniczenia w tworzeniu nowej jednostki domenowej w całym projekcie. Wzorce fabryki są podzielone na dwa podwzorce: fabryki abstrakcyjnej i metody wytwórczej. Główna różnica pomiędzy tymi dwoma wzorcami kreatywnymi polega na tym, że pierwszy definiuje ogólną fabrykę, która odpowiada za tworzenie dowolnego obiektu, jak pokazano w poniższym kodzie:

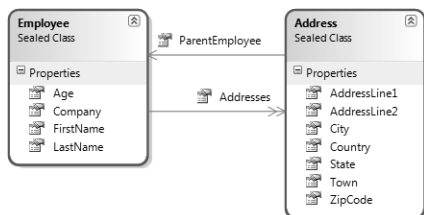
```
var employee = AbstractFactory.CreateEmployee();  
var order = AbstractFactory.CreateOrder();
```

Z kolei wzorec metody wytwórczej jest zorientowany bardziej na jednostkę domenową, więc będziemy mieli klasę fabryki dla każdej dostępnej jednostki w domenie:

```
var employee = EmployeeFactory.Create();  
var order = OrderFactory.Create();
```

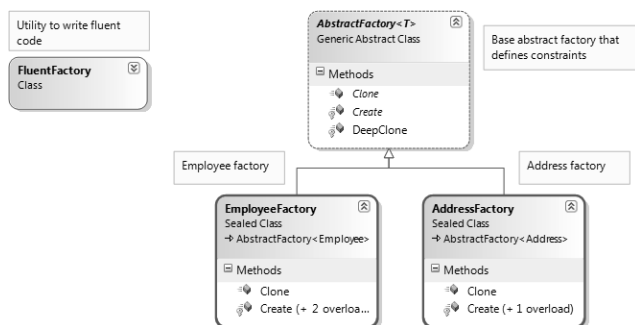
Od nas zależy, który sposób wykorzystamy, ale powinniśmy wziąć pod uwagę proces utrzymywania kodu podczas podejmowania tej decyzji. Jeśli na przykład wybierze-
my zastosowanie fabryki abstrakcyjnej, to za każdym razem, gdy będziemy musieli zmienić proces tworzenia dla określonego typu, możemy naruszyć kod tworzący jakiś inny typ. Ze względu na prostotę korzystam z niestandardowej implementacji fabryki abstrakcyjnej/metody wytwórczej.

Niewielka domena przedstawiona poniżej (rysunek 3-2) reprezentuje jednostkę *Employee* zawierającą zbiór adresów. Pierwszą regułą fabryki jest to, że nie możemy tworzyć obiektu *Address* bez przyłączenia go do nadrzędnego obiektu *Employee*, ponieważ jednostka *Address* bez jednostki *Employee* nie ma żadnego logicznego znaczenia biznesowego w tej domenie.



RYSUNEK 3-2 Przykładowy model domenowy dla wzorca fabryki

Implementacja metody wytwórczej/fabryki abstrakcyjnej, z której korzystam, jest przedstawiona na poniższym diagramie klas (rysunek 3-3), który zawiera fabrykę abstrakcyjną definiującą ograniczenia w moim kodzie, ale ma konkretną implementację fabryki dla każdej jednostki domenowej, żeby zapewnić większą kontrolę i oddzielenie od wzorca fabryki abstrakcyjnej:



RYСУNEK 3-3 Implementacja wzorca fabryki abstrakcyjnej w połączeniu z metodą wytwórczą.

Ta architektura zapewnia jasny sposób tworzenia nowego obiektu do wykorzystania w aplikacji. Na przykład, żeby utworzyć nowy obiekt *Address*, musimy podać nadrzędną jednostkę *Employee*, a żeby utworzyć nowy obiekt *Employee*, musimy podać podstawowe informacje wymagane dla jednostki *Employee*:

```

// Tworzenie nowego obiektu Employee
var employee = FluentFactory
    .Employee()
    .Create("John", "Smith", "Microsoft", 54);

// Tworzenie nowego obiektu Address
var address = FluentFactory
    .Address()
    .Create(employee, "Main Street 14", country: "USA");
employee.Addresses.Add(address);

// Sprawdzanie utworzonego obiektu
Console.WriteLine("Employee: {0} {1} has {2} addresses.",
    employee.FirstName,
    employee.LastName,
    employee.Addresses.Count);
Console.ReadLine();

// klonowanie pracownika i zmiana imienia
var cloned = FluentFactory.Employee().Clone(employee);
cloned.FirstName = "Sarah";

// sprawdzanie sklonowanego obiektu
Console.WriteLine("Employee cloned: {0} {1} has {2} addresses.",
    cloned.FirstName,

```

```

        cloned.LastName,
        cloned.Addresses.Count);
Console.ReadLine();

```

Możemy wciąż umieszczać ograniczenia w swojej fabryce, na przykład wyrzucać wyjątek, jeśli wartość podana jako parametr nie jest prawidłowa (na przykład *Age* < 21) albo jeśli wartość nie należy do określonego zakresu; w tym momencie jednak możemy uświadomić programistom potencjalne wyjątki korzystając z opisów kodu w Visual Studio. Przyjrzyjmy się następującemu kodowi używanemu do tworzenia nowego obiektu *Employee*:

```

/// <summary>
/// Tworzy określony obiekt Employee.
/// </summary>
/// <param name="firstName">Imię.</param>
/// <param name="lastName">Nazwisko.</param>
/// <param name="company">Firma.</param>
/// <param name="age">Wiek.
/// </param>
/// <returns></returns>
/// <exception cref="System.ArgumentNullException">Wyrzucany, gdy
/// wartość Age jest mniejsza od 21</exception>
public Employee Create(
    string firstName,
    string lastName,
    string company,
    int age)
{
    if (age < 21)
    {
        /// <exception cref="System.ArgumentNullException">Wyrzucany, gdy
        /// wartość Age jest mniejsza niż 21</exception>
        throw new ArgumentNullException("The Age should be greater than 21.");
    }
}

```

Możemy też rozważyć pozostawienie otwartej fabryki, a następnie zaimplementowanie wzorca sprawdzania poprawności danych w jednostce domenowej. Więcej na temat wzorca sprawdzania poprawności dowiemy się w kolejnym podrozdziale.

Sprawdzanie poprawności jednostek domenowych

Wcześniej przy omawianiu modelu domenowego napisałem, że jednostka domenowa powinna być niezależna od platformy albo warstwy danych: powinna być zwykłym obiektem POCO. Z tego samego powodu jednostka domenowa nie powinna być związana ze sprawdzaniem poprawności swoich danych, ponieważ różne warunki mogą być wymagane w dwóch różnych aplikacjach korzystających z tej samej domeny. Zwracam uwagę, że to tylko mój punkt widzenia; w środowisku architektów

oprogramowania trwają debaty na temat tego, gdzie umieszczać logikę sprawdzania poprawności danych. Niektórzy uważają, że powinna być ona umieszczana wewnątrz jednostki domenowej, a inni sądzą, że na zewnątrz.

Sprawdzanie poprawności jest procesem, w którym potwierdzamy, że dane w określonym obiekcie lub klasie są prawidłowe. Aby podjąć taką decyzję, potrzebny nam jest dla każdej właściwości zbiór *reguł sprawdzania poprawności* opisujących, kiedy i dlaczego dane będą albo nie będą poprawne.

Jednym z możliwych sposobów zapewnienia obsługi sprawdzania poprawności w modelu domenowym jest zapewnienie *typu nadrzędnego warstwy* (wspólnej klasy lub składnika używanego do zgromadzenia wspólnych zachowań lub właściwości wykorzystywanych przez wszystkie klasy lub obiekty w danej warstwie poprzez dziedziczenie po tym typie nadrzędnym), który może przekazywać sprawdzanie poprawności określonej jednostki do osobnej usługi sprawdzania poprawności. To właśnie dzieje się w przypadku modelu automatycznie generowanego przez Entity Framework, w którym każda klasa dziedziczy po wspólnej klasie bazowej zapewniającej też obsługę sprawdzania poprawności. Alternatywną metodą jest zapewnienie równoległego interfejsu do sprawdzania poprawności określonej jednostki domenowej i korzystanie z tego interfejsu do oddzielenia samej jednostki od procesu sprawdzania poprawności.

Klasyczne sprawdzanie poprawności

Poniższy przykład pokazuje klasyczny sposób sprawdzania poprawności obiektu przy pomocy prostej usługi sprawdzania poprawności z osadzonymi regułami sprawdzania poprawności. Najpierw przedstawię klasę bazową, która definiuje kontrakt dla obiektu sprawdzającego poprawność:

```
public abstract class BaseValidator<T>
{
    /// <summary>
    /// Ustala, czy określona jednostka jest prawidłowa.
    /// </summary>
    /// <param name="entity">Jednostka.</param>
    /// <returns>
    /// <c>true</c> jeśli określona jednostka jest prawidłowa; w przeciwnym razie
    <c>false</c>.
    /// </returns>
    public abstract bool IsValid(T entity);

    /// <summary>
    /// Pobiera lub ustawia błędy.
    /// </summary>
    /// <value>Błędy.</value>
    protected IList<ValidationResult> Errors { get; set; }
}
```

Następnie musimy utworzyć klasę sprawdzającą poprawność dla każdej jednostki domenowej, której poprawność chcemy sprawdzać. Oczywiście korzystając z tego

podejścia możemy łatwo rozszerzać klasę sprawdzając poprawność na inne obiekty, takie jak model widoku w aplikacji WPF/Silverlight lub obiekt DTO w usłudze RIA Service.

```
public sealed class EmployeeValidator : BaseValidator<Employee>
{
    /// <summary>
    /// Ustala, czy określona jednostka jest prawidłowa.
    /// </summary>
    /// <param name="entity">Jednostka.</param>
    /// <returns>
    ///     <c>true</c> jeśli określona jednostka jest poprawna; w przeciwnym razie
    <c>false</c>.
    /// </returns>
    public override bool IsValid(Employee entity)
    {
        var result = true;
        this.Errors = new List<ValidationResult>();
        if (entity.FirstName.Length < 10)
        {
            this.Errors.Add(new ValidationResult(
                "The Firstname should be greater than 10."));
            result = false;
        }
        if (entity.LastName.Length < 10)
        {
            this.Errors.Add(new ValidationResult(
                "The Lastname should be greater than 10."));
            result = false;
        }
        return result;
    }
}
```

Możemy łatwo przetestować ten kod sprawdzając, czy jednostka domenowa nie będzie prawidłowa, jeśli wstawimy puste imię (*FirstName*), jak w poniższym kodzie:

```
/// <summary>
///Test dla IsValid
///</summary>
[TestMethod()]
public void IsValidTest()
{
    EmployeeValidator target = new EmployeeValidator();
    Employee entity = new Employee { FirstName = "", LastName = ""};
    bool expected = false;
    bool actual;
    actual = target.IsValid(entity);
    Assert.AreEqual(expected, actual,
        "The Entity should not be valid at this point.");
}
```

Główne wady tego podejścia to:

- Reguły sprawdzania poprawności są osadzone w niestandardowej klasie, którą trudno jest udokumentować.
- Reguły sprawdzania poprawności składają się z zestawu instrukcji *if* w proceduralnym kodzie C#; gdy zestaw reguł stanie się bardziej skomplikowany, to trudniej będzie testować jego prawidłowość.

Zaletą korzystania z tego podejścia jest to, że model domeny jest całkowicie nieświadomy dostępnych reguł sprawdzania poprawności. Możemy korzystać z tej jednostki domenowej z lub bez obsługi sprawdzania poprawności i możemy zmieniać warunki sprawdzania poprawności w zależności od kontekstu.

Sprawdzanie poprawności z wykorzystaniem atrybutów i adnotacji danych

W wersji .NET Framework 4 dostępna jest przestrzeń nazw o nazwie *System.ComponentModel.DataAnnotations* zarówno dla wspólnego środowiska CLR (WPF), jak i dla uproszczonego środowiska Silverlight CLR. Z przestrzeni nazw *DataAnnotations* można korzystać do różnych celów. Jednym z nich jest sprawdzanie poprawności danych przy użyciu atrybutów, a innym jest wizualne opisywanie pól, właściwości i metod albo dostosowywanie typu danych określonej właściwości. Atrybuty te są podzielone na trzy kategorie: *atrybuty sprawdzania poprawności*, *atrybuty wyświetlania* i *atrybuty modelowania danych*. Ten podrozdział wykorzystuje atrybuty sprawdzania poprawności do definiowania reguł sprawdzania poprawności dla obiektów. Z atrybutów wyświetlania będziemy korzystać w rozdziale 6 „Warstwa interfejsu użytkownika w MVVM”, który jest poświęcony zestawowi narzędzi MVVM, a z atrybutów modelowania danych w rozdziale 4 „Warstwa dostępu do danych”.

Aby skorzystać z przestrzeni nazw *DataAnnotations*, musimy dodać odwołanie do jej podzespołu – odwołanie to nie jest domyślnie dołączane w żadnym projekcie szablonu Visual Studio. Następnie musimy opatrzyć swoje obiekty odpowiednimi atrybutami.

Jako przykład poniższy kod wykorzystuje *nieprawidłowe* podejście polegające na bezpośrednim opatrzeniu jednostki domenowej tymi atrybutami. Następnie przetworzę ten kod, aby jednostka nie była świadoma sprawdzania poprawności swoich danych.

```
public sealed class Customer
{
    /// <summary>
    /// Pobiera lub ustawia imię.
    /// </summary>
    /// <value>Imię.</value>
    [Required(ErrorMessage = "The FirstName is a mandatory Field")]
    [StringLength(10, ErrorMessage =
```



```

        "The FirstName should be greater than 10 characters.")]
public string FirstName { get; set; }

/// <summary>
/// Pobiera lub ustawia nazwisko.
/// </summary>
/// <value>Nazwisko.</value>
[Required(ErrorMessage = "The LastName is a mandatory Field")]
[StringLength(10, ErrorMessage =
    "The LastName should be greater than 10 characters.")]
public string LastName { get; set; }

/// <summary>
/// Pobiera lub ustawia tytuł.
/// </summary>
/// <value>Tytuł.</value>
[Required(ErrorMessage = "The Title is a mandatory Field")]
public string Title { get; set; }
}

```

Poprawność jednostki *Customer* można łatwo sprawdzić korzystając z ogólnego obiektu sprawdzającego poprawność, ponieważ wiemy, że chcemy sprawdzić poprawność tylko tych właściwości, które mają przypisane atrybuty *DataAnnotations*.

```

public sealed class GenericValidator<T>
{
    /// <summary>
    /// Sprawdza poprawność określonej jednostki.
    /// </summary>
    /// <param name="entity">Jednostka.</param>
    /// <returns></returns>
    public IList<ValidationResult> Validate(T entity)
    {
        var results = new List<ValidationResult>();
        var context = new ValidationContext(entity, null, null);
        Validator.TryValidateObject(entity, context, results);
        return results;
    }
}

```

W tym momencie możemy łatwo przetestować działanie obiektu sprawdzającego poprawność dla jednostki *Customer*:

```

/// <summary>
/// Ustala, czy to wystąpienie może sprawdzać poprawność danych klienta.
/// </summary>
[TestMethod]
public void CanValidateCustomer()
{
    Customer entity = new Customer { FirstName = "", LastName = "" };
    GenericValidator<Customer> target = new GenericValidator<Customer>();
    bool expected = false;
    bool actual;
    actual = target.Validate(entity).Count == 0;
}

```

```

        Assert.AreEqual(expected, actual,
            "The Entity should not be valid at this point.");
    }
}

```

Teraz, aby usunąć sprawdzanie poprawności z jednostki domenowej, musimy utworzyć interfejs, który reprezentuje jednostkę domenową i zawiera reguły sprawdzania poprawności, a następnie odziedziczyć jednostkę domenową po tym interfejsie. Na końcu tego procesu powinniśmy być w stanie napisać kod podobny do poniższego:

```

/// <summary>
/// Ustala, czy to wystąpienie może sprawdzać poprawność danych klienta.
/// </summary>
[TestMethod]
public void CanValidateCustomer()
{
    Customer entity = new Customer { FirstName = "", LastName = "" };
    GenericValidator<ICustomer> target = new GenericValidator<ICustomer>();
    bool expected = false;
    bool actual;
    actual = target.Validate(entity).Count == 0;
    Assert.AreEqual(expected, actual,
        "The Entity should not be valid at this point.");
}
}

```

Dostępne platformy sprawdzania poprawności danych

Technika sprawdzania poprawności, która została właśnie przedstawiona, jest tylko jedną z technik dostępnych dla .NET. Zaletą korzystania z *DataAnnotations* jest doskonała współpraca z WPF i Silverlight oraz działanie we wszystkich warstwach aplikacji MVVM. W części dotyczącej modelu widoku zobaczymy, dlaczego podejście wykorzystujące *DataAnnotations* idealnie pasuje do WPF lub Silverlight.

Inną ciekawą platformą utworzoną przez Microsoft jest moduł *Validation Application Block*, który jest dostępny wraz z Microsoft Enterprise Library 5.0 (<http://entlib.codeplex.com/>). *Validation Application Block* wykorzystuje to samo ogólne podejście – sprawdzanie poprawności obiektu względem zestawu reguł zdefiniowanych przy pomocy atrybutów (adnotacji danych) albo zewnętrznego pliku XML. Główną różnicą w stosunku do *DataAnnotations* jest proces używany do sprawdzania poprawności obiektu, ale w obu przypadkach powinniśmy uzyskać ten sam wynik końcowy.

Inną platformą, częścią otwartego projektu NHibernate, jest *NHibernate Validation Framework*. Jest ona dostępna pod adresem <http://sourceforge.net/projects/nhcontrib/> jako część projektu NHibernate Contrib. Główną wadą korzystania z tej platformy jest to, że jeśli nie zamierzamy korzystać z NHibernate jako swojego systemu O/RM, to wprowadzimy dodatkowe zależności w swoich warstwach, które mogą nie być potrzebne. Platforma ta wymaga również opatrzenia jednostek regułami sprawdzania poprawności związanymi z określonym systemem O/RM.

Podsumowując ważne jest, aby domena była czysta i nieświadoma reguł sprawdzania poprawności albo stosowanych metod, trzeba też zdecydować się na wykorzystanie odpowiedniej platformy dla typu pisanej aplikacji. W tej książce będziemy przede wszystkim korzystać z funkcji adnotacji danych zapewnianej w środowisku .NET Framework.

Testy jednostkowe modelu domenowego

Należy utworzyć proces dla testowania modelu domenowego przed rozpoczęciem pisania kodu dla samego modelu; to zagwarantuje nam, że będziemy testować zgodność kodu z oczekiwanymi wynikami, a nie odwrotnie.

Gdy piszemy model domenowy, to zwykle dołączamy pewne niewielkie reguły biznesowe do swojego kodu, które powinny być sprawdzane tak, abyśmy mieli pewność, że model działa prawidłowo. Na przykład model domenowy dla aplikacji CRM będzie zawierał jednostkę *Person*, która będzie zawierać zbiór jednostek *Address* w postaci kolekcji *IList<T>*. Chcemy zagwarantować, że w kontekście całego modelu jednostka *Person* może mieć ustawioną jedną i tylko jedną jednostkę *Address* jako adres domyślny. Inna reguła mówi, że o ile nie zostanie to sprecyzowane inaczej, pierwszy adres dodany do listy adresów jednostki *Person* będzie adresem domyślnym.

Na potrzeby tego przykładu powinniśmy być w stanie napisać pierwszy test podobny do następującego:

```
var person = PersonFactory.Create();
var address01 = AddressFactory.Create();
var address02 = AddressFactory.Create();
person.AddAddress(address01);
person.AddAddress(address02);
Assert.IsNotNull(person.DefaultAddress);
Assert.IsTrue(person.DefaultAddress == address01);
```

Innym testem – nudnym, ale przydatnym – jest sprawdzanie wartości każdej właściwości jednostki przed rozpoczęciem sprawdzania poprawności samej jednostki. Na przykład możemy chcieć się upewnić, że gdy wywołamy właściwość tylko do odczytu *FullName* jednostki *Person*, to wynik będzie się składał z właściwości *FirstName*, spacji i właściwości *LastName*.

```
var person = PersonFactory.Create("John", "Smith");
Assert.AreEqual(person.FullName, "John Smith");
```

Stałe testowanie definicji naszego modelu domenowego na zgodność z regułami naszego modelu gwarantuje nam, że wszelkie zmiany modelu nie będą niekorzystnie wpływać na istniejącą strukturę danych i istniejący przebieg zadań w modelu.

Sprawdzanie poprawności jest też kolejną ciekawą częścią, którą trzeba przetestować, aby upewnić się, że to podejście działa zgodnie z oczekiwaniami. Jedyne problemy, jakie możemy napotkać przy testowaniu sprawdzania poprawności jest związany z tym,

że powinniśmy trwale kodować reguły sprawdzania poprawności w swoich testach, aby mieć pewność, że testujemy prawidłowość zestawu reguł sprawdzania poprawności; z drugiej strony przydaje się to do śledzenia, co się zmieniło w samym zestawie reguł sprawdzania poprawności.

```
var person = PersonFactory.Create("John", "Smith");  
// Sprawdzanie poprawności zwracania wartości Boolean  
Assert.IsTrue(Validator.Validate(person));  
var invalidPerson = PersonFactory.Create();  
Assert.IsFalse(Validator.Validate(invalidPerson));
```

Kod przykładowy: model domenowy CRM

Począwszy od tego rozdziału koniec każdego rozdziału zawiera podrozdział o nazwie „Kod przykładowy”, w którym będziemy budować aplikację CRM wykorzystując wiedzę nabytą we wcześniejszych częściach danego rozdziału. W tym rozdziale zobaczyliśmy, czym jest model domenowy, jak powinien być implementowany i testowany, a także przyjrzelśmy się implementacji fabryki oraz procesowi sprawdzania poprawności.

Najpierw wróćmy więc do historii użytkownika przedstawionej przez klienta podczas zamawiania nowej aplikacji CRM.

„Jako firma sprzedająca produkty chcę być w stanie zarządzać swoimi zamówieniami (Order); potrzebuję systemu, który monitoruje dostępność produktów (Product), rejestru do administrowania klientami (Customer) i procesu zatwierdzania zamówień przez jednego z pracowników (Employee) zarejestrowanych w systemie.”

Zdefiniowałem tę historię użytkownika przy pomocy pojedynczej domeny składającej się z jednostek odpowiadających za administrowanie pracownikami, klientami i informacjami o nich oraz jednostki odpowiadającej za składanie i zatwierdzanie zamówienia w oparciu o listę przesłanych i dostępnych produktów.

Kontekst osoby

Jednostki *Employee* i *Customer* mogą być pogrupowane według pewnych wspólnych informacji, takich jak imię (*FirstName*), nazwisko (*LastName*), itd. Ale istnieją też właściwości, które są bardziej związane z jednostką *Customer* niż *Employee*. Na przykład może nam nie zależeć na wyświetlaniu adresu pracownika, ale konieczna może być informacja, jak się z nim skontaktować; z drugiej strony użytkownik prawdopodobnie musi wiedzieć wszystko na temat klienta, który złożył zamówienie, ponieważ musi wiedzieć, gdzie przesłać to zamówienie i jak skontaktować się z klientem, jeśli wystąpi problem ze złożonym zamówieniem.

Nasza domena będzie miała na razie bardo prosty nadrzędny typ warstwy, którym będzie klasa *DomainObject*. Ta klasa ma tylko jedną właściwość *PrimaryKey* typu

GUID, która pomoże nam rozróżniać poszczególne jednostki dostępne w kontekście domeny. W następnym rozdziale zobaczymy, dlaczego ważne jest, aby zdefiniować typ klucza głównego jednostki przed podjęciem decyzji dotyczącej docelowego magazynu danych.

UWAGA Podjąłem decyzję projektową, że będę dekorował jednostki domenowe regułami sprawdzania poprawności. Robię to jedynie dlatego, że chcę pokazać, jak korzystać z atrybutów sprawdzania poprawności – a jednocześnie będziemy mogli zbadać strukturę jednostek domenowych. W przypadku rzeczywistych aplikacji zalecam osadzanie atrybutów sprawdzania poprawności w zewnętrznym interfejsie, który zostanie zaimplementowany przez jednostkę domenową.



Poniższy kod pokazuje bazowy typ nadrzędny warstwy *DomainObject*:

```
/// <summary>
/// Podstawowy obiekt domenowy
/// </summary>
public abstract class DomainObject
{
    /// <summary>
    /// Pobiera lub ustawia klucz główny.
    /// </summary>
    /// <value>Klucz główny.</value>
    [Required(ErrorMessage = "The Primary Key can't be null or empty.")]
    public Guid PrimaryKey { get; set; }
}
```

Model będzie zawierał jeszcze jedną klasę abstrakcyjną o nazwie *Person*, która definiuje pewne wspólne właściwości i metody dostępne dla jednostek *Employee* i *Customer*. Ta klasa musi być abstrakcyjna, ponieważ nie chcemy, aby została przypadkiem użyta bezpośrednio przez jakiegoś programistę w kodzie, ale jednocześnie nie chcemy pisać tego samego kodu dwa razy.

```
public interface class Person : DomainObject
{
    /// <summary>
    /// Pobiera lub ustawia imię.
    /// </summary>
    /// <value>Imię.</value>
    [Required(ErrorMessage = "The FirstName can't be null or empty.")]
    public string FirstName { get; set; }

    /// <summary>
    /// Pobiera lub ustawia nazwisko.
    /// </summary>
    /// <value>Nazwisko.</value>
    [Required(ErrorMessage = "The LastName can't be null or empty.")]
    public string LastName { get; set; }
}
```

```

/// <summary>
/// Pobiera lub ustawia pełne imię i nazwisko.
/// </summary>
/// <value>Pełne imię i nazwisko.</value>
public string FullName
{
    get { return String.Format("{0} {1}", FirstName, LastName); }
}

/// <summary>
/// Pobiera lub ustawia tytuł.
/// </summary>
/// <value>Tytuł.</value>
[Required(ErrorMessage = "The Title can't be null or empty.")]
public string Title { get; set; }

/// <summary>
/// Pobiera lub ustawia datę urodzenia.
/// </summary>
/// <value>Data urodzenia.</value>
[Required(ErrorMessage = "The Birth Date can't be null or empty.")]
public DateTime BirthDate { get; set; }

/// <summary>
/// Pobiera lub ustawia wartość wskazującą, czy to wystąpienie jest aktywne.
/// </summary>
/// <value><c>true</c>, jeśli to wystąpienie jest aktywne;
/// w przeciwnym razie, <c>false</c>.</value>
public bool IsActive { get; set; }

/// <summary>
/// Pobiera lub ustawia kontakty.
/// </summary>
/// <value>Kontakty.</value>
public IList<Contact> Contacts { get; set; }

/// <summary>
/// Pobiera domyślny kontakt.
/// </summary>
/// <value>Domyślny kontakt.</value>
public Contact DefaultContact
{
    get
    {
        if (Contacts == null)
        {
            return null;
        }
        return Contacts.Where(x => x.IsDefault).FirstOrDefault();
    }
}

/// <summary>
/// Dodaje kontakt.
/// </summary>

```

```

/// <param name="contact">Kontakt.</param>
public void AddContact(Contact contact)
{
    if (Contacts == null)
    {
        Contacts = new List<Contact>();
    }

    // Jeśli nie ma domyślnego adresu, ustaw ten jako domyślny
    if (Contacts.Where(x => x.IsDefault).Count() < 1)
    {
        contact.IsDefault = true;
    }

    //Jeśli to jest nowy adres domyślny
    if (contact.IsDefault)
    {
        foreach (Contact cont in Contacts)
        {
            cont.IsDefault = false;
        }
    }

    // Jeśli adresu nie ma jeszcze na liście
    if (!Contacts.Any(x => x.PrimaryKey == contact.PrimaryKey))
    {
        Contacts.Add(contact);
    }
}

```

Można już zauważyć, że ta jednostka wprowadza dwie nowe jednostki: *Contact* i *Address*. Jednostka *Contact* będzie udostępniana zarówno przez jednostkę *Employee*, jak i *Customer*, ponieważ dla każdej z nich potrzebny może być kontakt domyślny. Jednostka *Address* jest udostępniana jedynie przez jednostkę *Customer*, ponieważ znajomość określonego adresu pracownika (*Employee*) nie jest wymagana przez budowaną przez nas aplikację biznesową.

```

public sealed class Contact : DomainObject
{
    /// <summary>
    /// Pobiera lub ustawia typ kontaktu.
    /// </summary>
    /// <value>Typ kontaktu.</value>
    public ContactType ContactType { get; set; }

    /// <summary>
    /// Pobiera lub ustawia nazwę.
    /// </summary>
    /// <value>Nazwa.</value>
    [Required(ErrorMessage = "The Name is a mandatory field")]
    public string Name { get; set; }
}

```

```

    /// <summary>
    /// Pobiera lub ustawia opis.
    /// </summary>
    /// <value>Opis.</value>
    [Required(ErrorMessage = "The Description is a mandatory field")]
    public string Description { get; set; }

    /// <summary>
    /// Pobiera lub ustawia numer.
    /// </summary>
    /// <value>Numer.</value>
    [Required(ErrorMessage = "The Number is a mandatory field")]
    public string Number { get; set; }

    /// <summary>
    /// Pobiera lub ustawia wartość wskazującą, czy to wystąpienie jest domyślne.
    /// </summary>
    /// <value>
    ///     <c>true</c>, jeśli to wystąpienie jest domyślne;
    ///     w przeciwnym razie <c>false</c>.
    /// </value>
    public bool IsDefault { get; set; }
}

```

Warto zwrócić uwagę, że jednostka *Contact* wprowadza pierwszy typ wyliczeniowy (enum) w naszej aplikacji. Korzystanie z typów wyliczeniowych w domenie jest ważne, ponieważ wprowadzają one zrozumiałe nazwy i eliminują błędy, które mogą wystąpić przy używaniu łańcuchów tekstowych lub wartości liczbowych.

Poniższa jednostka *Address* jest podobna do jednostki *Contact* i różni się jedynie udostępnianymi właściwościami.



UWAGA Jeśli stosujemy podejście Martina Fowlera, to powinniśmy klasyfikować jednostki *Contact* i *Address* raczej jako obiekty wartości, a nie jednostki domenowe, ponieważ reprezentują one niewielki składnik większej jednostki domenowej, a same nie są prawdziwymi jednostkami domenowymi – nie są reprezentowane samodzielnie w domenie.

```

public sealed class Address : DomainObject
{
    /// <summary>
    /// Pobiera lub ustawia pierwszy wiersz adresu.
    /// </summary>
    /// <value>Pierwszy wiersz adresu.</value>
    [Required(ErrorMessage = "The AddressLine1 is a mandatory field")]
    public string AddressLine1 { get; set; }

    /// <summary>
    /// Pobiera lub ustawia drugi wiersz adresu.
    /// </summary>
    /// <value>Drugi wiersz adresu.</value>
    public string AddressLine2 { get; set; }
}

```



```

    /// <summary>
    /// Pobiera lub ustawia miejscowość.
    /// </summary>
    /// <value>Miejscowość.</value>
    public string Town { get; set; }

    /// <summary>
    /// Pobiera lub ustawia miasto.
    /// </summary>
    /// <value>Miasto.</value>
    public string City { get; set; }

    /// <summary>
    /// Pobiera lub ustawia stan.
    /// </summary>
    /// <value>Stan.</value>
    [Required(ErrorMessage = "The AddressLine1 is a mandatory field")]
    public string State { get; set; }

    /// <summary>
    /// Pobiera lub ustawia kraj.
    /// </summary>
    /// <value>Kraj.</value>
    [Required(ErrorMessage = "The AddressLine1 is a mandatory field")]
    public string Country { get; set; }

    /// <summary>
    /// Pobiera lub ustawia kod pocztowy.
    /// </summary>
    /// <value>Kod pocztowy.</value>
    [Required(ErrorMessage = "The AddressLine1 is a mandatory field")]
    public string ZipCode { get; set; }

    /// <summary>
    /// Pobiera lub ustawia wartość wskazującą, czy to wystąpienie jest domyślne.
    /// </summary>
    /// <value>
    ///     <c>true</c>, jeśli to wystąpienie jest domyślne;
    ///     w przeciwnym razie <c>false</c>.
    /// </value>
    public bool IsDefault { get; set; }
}

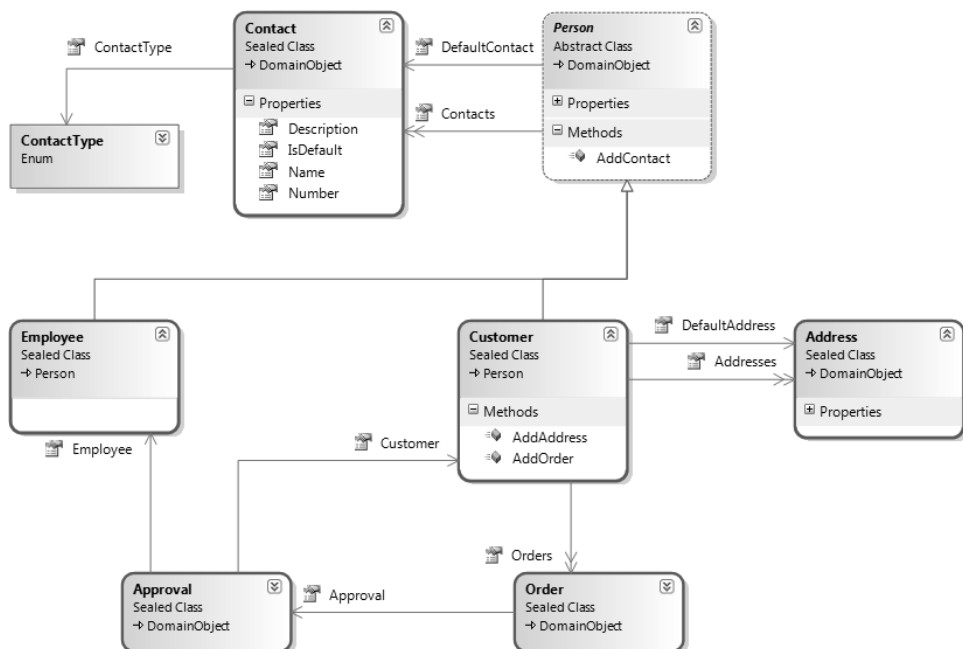
```

Jednostka domenowa *Employee* została tu pominięta, ponieważ, jak widać z rysunku 3-4, jej implementacja jest bardzo zrozumiała; dziedziczy po klasie *Person* i nie ma żadnych dodatkowych właściwości.

W efekcie mamy teraz dwie główne jednostki w tym modelu: *Employee* i *Person*. Różnicę pomiędzy nimi stanowi, jak zauważyłem wcześniej, lista *Addresses*, ale poza tym jednostka *Customer* zawiera listę zamówień, a jednostka *Employee* zawiera listę zatwierdzeń.

Rysunek 3-4 pokazuje ostateczny diagram domeny *Person*. Dla zaoszczędzenia miejsca nie zawarłem tutaj pozostałego kodu, ale pełen kod źródłowy jest dostępny do

pobrania jako zawartość towarzysząca tej książce (zgodnie z instrukcjami pobierania we wstępie do tej książki).



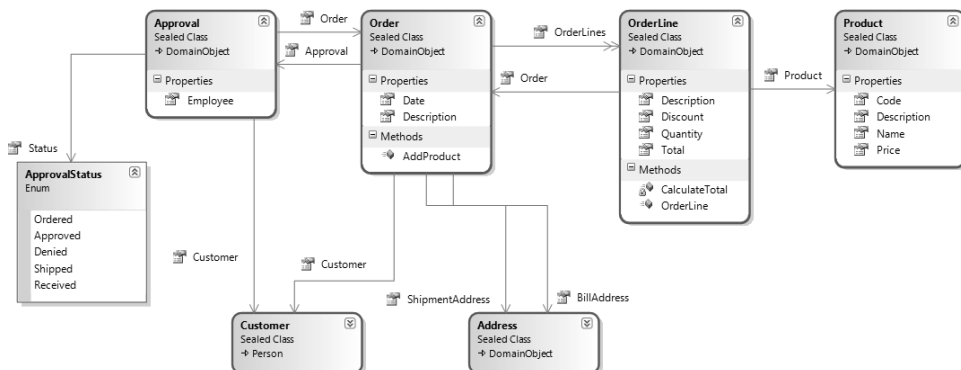
RYSUNEK 3-4 Część pełnego modelu domenowego, domena *Person*.

Domena Order

Przykładowa aplikacja CRM odpowiada za monitorowanie procesu zamówienia dla określonej kombinacji klienta/produktów. Proces zamawiania składa się z trzech głównych jednostek: samego zamówienia, listy elementów na zamówieniu i każdego elementu, który składa się z produktu, ilości, ceny jednostkowej i sumy całkowitej opartej na zastosowanym rabacie.

W odwrotnej kolejności pierwszą jednostką, którą napotkamy, jest jednostka *Product*, która reprezentuje unikalny produkt w magazynie firmy. *Product* jest jednostką domenową odpowiadającą za reprezentowanie produktu i jego właściwości w całej domenie. Rysunek 3-5 reprezentuje pełną domenę dla procesu zamówienia.

Na tym diagramie widać jednostkę *Order* występującą w jednostce *OrderLine*, która definiuje ilość i cenę całkowitą dla danego zamówienia. Jednostka *Order* ma odwołania do dwóch jednostek *Address*: jedno dla adresu na fakturze (*BillingAddress*) i jedno dla adresu dostawy (*ShipmentAddress*). Każda jednostka *Order* podlega procesowi zatwierdzania, w którym uczestniczy sama jednostka *Order*, jednostka *Employee* i jednostka *Customer*.



RYSUNEK 3-5 Model domenowy zamówienia.

Kilka ciekawych wierszy kodu zajmuje się obliczaniem sumy całkowitej elementu *OrderLine*, jak pokazano w poniższym kodzie:

```

/// <summary>
/// Oblicza sumę.
/// </summary>
private void CalculateTotal()
{
    if (Discount > 0)
    {
        Total = Product.Price * Quantity * Discount;
    }
    else
    {
        Total = Product.Price * Quantity;
    }
}

```

Jako ograniczenie w konstruktorze *OrderLine* musimy już wiedzieć, jak zbudować element *OrderLine* przed jego utworzeniem.

```

/// <summary>
/// Inicjuje nowe wystąpienie klasy <see cref="OrderLine"/>.
/// </summary>
/// <param name="order">Zamówienie.</param>
/// <param name="product">Produkt.</param>
/// <param name="quantity">Ilość.</param>
/// <param name="discount">Rabat.</param>
public OrderLine(Order order, Product product, int quantity, decimal discount)
{
    this.Product = product;
    this.Quantity = quantity;
    this.Discount = discount;
    this.Order = order;
    CalculateTotal();
}

```

Cała transakcja jest zawarta w metodzie *AddProduct* jednostki *Order*.

```
/// <summary>
/// Dodaje produkt.
/// </summary>
/// <param name="product">Produkt.</param>
/// <param name="quantity">Ilość.</param>
/// <param name="discount">Rabat.</param>
public void AddProduct(Product product, int quantity, decimal discount = 0)
{
    if (OrderLines == null)
    {
        OrderLines = new List<OrderLine>();
    }
    OrderLines.Add(new OrderLine(this, product, quantity, discount));
}
```

Cały kod projektu dostępny z tą książką zawiera wszystkie testy jednostkowe dla domeny, fabryki dla każdej jednostki domenowej i wszystkie reguły sprawdzania poprawności. Aby uniknąć drukowania wielu stron kodu C#, ta książka zawiera tylko wybrane kroki z procesu tworzenia modelu domenowego.

Podsumowanie

W tym rozdziale utworzyliśmy podstawowy model domenowy dla aplikacji CRM. Jest to kontekst biznesowy (nie logika biznesowa) dla aplikacji, którą wykonamy w kolejnych rozdziałach. Możemy łatwo rozszerzać tę domenę korzystając z innego podejścia albo dodając nowe jednostki, takie jak te zaproponowane poniżej:

- *Product* powinien odwoływać się do jednostki *Magazine* lub *Stock*, która będzie przechowywała zapasy w magazynie i aktualizowała dostępność produktów.
- Proces zatwierdzania (*Approval*) używany w procesie zamawiania (*Order*) powinien kontaktować się z odpowiednim klientem (*Customer*) i pracownikiem (*Employee*) korzystając z ich adresów domyślnych za każdym razem, gdy zmienia się status zatwierdzenia.

Jako dalsze ulepszenia wykorzystamy konkretną warstwę biznesową w rozdziale 5 „Warstwa biznesowa”, gdzie zobaczymy, jak implementować niestandardowe reguły i przepływ zadań do zatwierdzania, odrzucania i kończenia zamówienia.

Warstwa dostępu do danych

Po zakończeniu tego rozdziału będziemy w stanie:

- Rozpoznać i wybrać odpowiedni system Object/Relational Mapper.
- Stworzyć elastyczną warstwę dostępu do danych.
- Utworzyć mapowanie przy pomocy Entity Framework i NHibernate.

Wprowadzenie

Warstwa dostępu do danych (DAL – Data Access Layer) oddziela dostęp do danych i przechowywanie danych od reszty aplikacji zapewniając podział interesów pozwalający odizolować mechanikę przechowywania i pobierania danych od wykorzystania tych danych w aplikacji. To oznacza, że aplikacja i magazyn danych mogą rozwijać się łatwiej albo nawet zostać wymienione na inne elementy. Jednakże obie te części nie są całkowicie rozłączone; istnieje „kontrakt” pomiędzy nimi, który jest tak zaprojektowany, aby warstwa DAL zapewniała dostęp do określonych danych (jednostek), których potrzebuje aplikacja niezależnie od tego, jak te dane są faktycznie przechowywane. Warstwa DAL pozwala nam pisać aplikację w kontekście jednostek, które możemy odczytywać i aktualizować poprzez DAL. Warstwa DAL wymusza również wszelkie reguły biznesowe lub logikę biznesową, żeby zapewniać spójność danych.

Czy planujemy zastosowanie biblioteki firmy trzeciej, otwartej platformy, bardziej skomplikowanego systemu O/RM – Object/Relational Mapper, czy też własnej warstwy DAL, to powinniśmy zawsze starać się utrzymać ich użycie wewnątrz samego składnika DAL. Oznacza to, że warstwa DAL powinna udostępniać atomowe metody, które są w stanie wykonywać zapytania i komunikować się z obiektami wewnętrznymi składającymi się na DAL bez ujawniania tych obiektów w innych warstwach. Innym problemem, który możemy napotkać przy budowaniu niestandardowej warstwy DAL, jest proces mapowania. Warstwa DAL musi zapewniać mechanizm tłumaczący pomiędzy modelem danych (jednostkami domenowymi) wykorzystywanym przez aplikację a wewnętrznym magazynem i schematem danych. Możemy to robić ręcznie i jednocześnie uzyskać pewne korzyści, jeśli chodzi o wydajność i możliwość dostosowywania

albo możemy wykorzystać platformę O/RM, taką jak Microsoft Entity Framework lub NHibernate (żeby wymienić tylko dwie), aby to tłumaczenie uczynić bardziej elastycznym i zwiększyć poziom standardowości.

Użycie warstwy DAL w aplikacji MVVM nie jest obowiązkowym wymaganiem, ponieważ te dwa wzorce architektoniczne nie są powiązane ze sobą; warstwa DAL opisuje sposób wydzielania składnika dostępu do danych, natomiast MVVM opisuje prezentacyjny wzorzec projektowy. Zwykle dodajemy warstwę DAL do aplikacji MVVM, aby utrzymać jasny podział interesów i zwiększyć elastyczność aplikacji. W sytuacji idealnej jednostki domenowe powinny obsługiwać interfejsy i funkcje czyniące z nich idealne klasy modelu na użytek aplikacji MVVM (takie jak *INotifyPropertyChanged*, *IErrorInfo*, itd.). W ten sam sposób dobrze zaprojektowana warstwa DAL pozwala aplikacji pobierać model, który jest jej potrzebny dla określonego ekranu i obsługuje jego aktualizowanie.

Użycie warstwy DAL w aplikacji izoluje interfejs użytkownika i domenę od bazy danych. Przykładowa aplikacja MVVM będzie miała strukturę składającą się z domeny (rozdział 3 „Model domenowy”), warstwy DAL (ten rozdział), warstwy biznesowej (rozdział 5 „Warstwa biznesowa”) i warstwy interfejsu użytkownika (rozdział 6 „Warstwa interfejsu użytkownika w MVVM”).

Baza danych i procedury składowane

System zarządzania bazami danych (DBMS) jest zbiorem programów komputerowych sterujących tworzeniem, utrzymywaniem i wykorzystywaniem bazy danych.

W mojej opinii najgroźniejszą rzeczą, którą możemy zrobić w bazie danych, jest utworzenie procedur składowanych. Możliwe, że wielu czytelników nie zgodzi się ze mną w tej kwestii. Oto, dlaczego z perspektywy projektowania sterowanego domeną (DDD) procedury składowane są niedobre – nie tylko w aplikacji MVVM wykorzystującej warstwę danych, ale także w przypadku bardziej ogólnych aplikacji biznesowych, które zostały podzielone na warstwy w celu zwiększenia możliwości testowania i zmniejszenia wysiłku związanego z ich utrzymaniem.

Głównym celem użycia warstwy danych jest wyodrębnienie przechowywania danych poza aplikację tak, że inne warstwy nie są świadome mechanizmów przechowywania danych używanych przez warstwę DAL. Jeśli magazyn danych wykorzystuje procedury składowane, to ten cel można osiągnąć – w istocie kilka implementacji systemów O/RM zapewnia pełną obsługę procedur składowanych. Z drugiej strony głównym celem istnienia warstwy DAL jest unikanie ścisłego powiązania pomiędzy kodem C# a bazą danych, poza kontraktem, który definiujemy jako część warstwy DAL. Tak długo, jak baza danych spełnia ten kontrakt, nie musimy przejmować się w ogóle tym, jaka jest wewnętrzna implementacja. Tworzymy atrapę warstwy DAL tak, abyśmy mogli wszystko testować po stronie aplikacji. Podobnie możemy przeprowadzać testy po stronie warstwy DAL, aby zapewnić, że baza danych spełnia kontrakt.

Idealnie nie przechowywalibyśmy logiki biznesowej aplikacji w procedurze składowanej, ponieważ rola procedury składowanej powinna się skupiać bardziej na utrzymywaniu spójności danych w bazie danych. Jednakże w praktyce tworzenie solidnej logiki biznesowej i spójności danych może być traktowane jako dwie strony medalu. Testowanie logiki biznesowej przechowywanej wewnątrz procedury składowanej jest skomplikowanym zadaniem, to samo dotyczy utrzymywania i zabezpieczania bazy danych. Te sprawy są dobrze rozumiane przez administratorów baz danych i wiele organizacji może nie tolerować kosztów budowania i utrzymywania bazy danych, gdzie wszystkie aplikacje muszą trzymać się zasad, żeby zapewnić spójność i bezpieczeństwo danych, ponieważ nie byłoby to praktyczne.

Aby pomóc w wyjaśnieniu, dlaczego powinniśmy unikać używania procedur składowanych i wprowadzać elastyczną kombinację DAL-O/RM w swoich aplikacjach, przygotowałem kilka typowych argumentów, których używam w dyskusjach z administratorami baz danych:

- **Utrzymywanie** Procedury składowane nie są łatwe w utrzymaniu. Gdy zmieniamy procedurę składowaną, to często musimy zmienić jej sygnaturę tak, aby zawierała nowy parametr. W wyniku tej zmiany każdy fragment kodu wykorzystujący tę procedurę składowaną staje się nieprawidłowy – ale system zarządzania bazą danych nie oferuje sposobu na odnajdywanie zależności pomiędzy procedurą składowaną a kodem C#, który ją wykorzystuje.
- **Bezpieczeństwo** W Microsoft SQL Server możemy definiować i nadawać szczegółowy dostęp do pojedynczego pola pojedynczego wiersza w tabeli, co daje niezłe zabezpieczenie, ale w przypadku aplikacji DAL możemy po prostu nadawać zabezpieczenia poprzez warstwę bezpieczeństwa bez martwienia się uwierzytelnianiem i autoryzacją po stronie bazy danych.
- **Wydajność** Większość nowoczesnych systemów O/RM może generować plany wykonywania i optymalizować kod Dynamic SQL tworzony przez warstwę DAL, co skutecznie przeczy fantastycznym historiom, jakoby procedury składowane były bardziej wydajne niż dynamiczne instrukcje SQL.

Jeśli ktoś jest nadal zmotywowany do korzystania z procedur składowanych w swoich aplikacjach, to oczywiście może to robić. Powinniśmy też wziąć pod uwagę, że systemy O/RM, takie jak NHibernate i Entity Framework, są w stanie zastępować swój automatycznie generowany kod T-SQL wykorzystując kilka wstępnie zaprojektowanych procedur składowanych, z których możemy korzystać.

Wybór systemu O/RM

System O/RM (Object-Relational Mapper) jest platformą odpowiadającą za konwersję danych pomiędzy dwoma rozłącznymi systemami. Jednym z tych systemów jest zwykle model obiektowy, a drugim jest obiekt bazodanowy, taki jak tabela lub widok.

Jak widzieliśmy w rozdziale 3, istnieją różne sposoby tworzenia domeny aplikacji. W zależności od tego, czy używamy wzorca rekordu aktywnego, czy DDD, nasze narzędzie O/RM będzie skonfigurowane inaczej.

Pojęcie O/RM stanowi mechanizm zachowywania danych, który jest łatwy do zrozumienia, ale trudny do osiągnięcia. Przechowuje instrukcje mapowania obiektu domenowego na jeden lub kilka obiektów bazodanowych w *słowniku mapowania*. Korzystając z tego słownika narzędzie O/RM na bieżąco generuje kod konieczny do pobierania i zachowywania danych z bazy danych w modelu domenowym i vice versa. Zwykle generuje również klasę odpowiadającą modelowi domenowemu zwaną jako *pośrednik*, która wywodzi się z klasy modelu domenowego dodając aspekt utrwalania danych.

Co więcej, wiele narzędzi O/RM oferuje możliwość przechowywania danych w pamięci podręcznej, pisania transakcji dla magazynu danych, a nawet może zapewniać obiektowo zorientowany język programowania zapytań, który jest w pełni zintegrowany z wszechobecnym językiem domenowym tłumaczonym na żądanie na język specyficzny dla domeny (DSL – Domain-Specific Language) zrozumiały przez magazyn danych. Niektóre narzędzia O/RM oferują też możliwość przełączania się pomiędzy różnymi bazami danych, na przykład z SQL Server do Oracle albo z MySQL do IBM DB2.

Poniższa lista pokazuje, dlaczego ważne jest wykorzystanie narzędzia O/RM w aplikacji MVVM:

- **Izolacja** Możemy całkowicie odizolować domenę od magazynu danych. Jest to podstawowa reguła aplikacji DDD i oczywiście dowolnej biznesowej aplikacji MVVM.
- **Upraszczenie** Narzędzia O/RM eliminują potrzebę pisania kodu do tworzenia, modyfikowania i odpytywania obiektów bazodanowych.
- **Poprawiona łatwość utrzymania** Dzięki narzędziu O/RM musimy zmieniać tylko model domenowy; magazyn danych będzie się dostosowywał automatycznie, jeśli planujemy wykorzystanie O/RM do sterowania utrzymywaniem schematu naszej bazy danych.
- **Nawigacja w domenie** Nie jest łatwo zrozumieć, jak działa przepływ w bazie danych czytając jedynie dostępny schemat tabel, ale zwykle dość łatwo jest odczytać diagram UML modelu domenowego, aby zrozumieć, jak została zaprojektowana aplikacja. Jeśli korzystamy z narzędzia O/RM, to domena będzie naszą jedyną dokumentacją. Oczywiście w tym przypadku możemy osiągnąć ten sam cel korzystając z niestandardowej warstwy DAL; ważne jest, że poza tym mamy też model domenowy.
- **Funkcje** Moim motto jest, że nie ma co odkrywać Ameryki. Narzędzia O/RM oferują funkcje, takie jak pamięć podręczna, możliwości transakcyjne, sprawdzanie zbieżności, itd.

To prawdopodobnie wystarczy. Narzędzia O/RM są fajne i błyszczące – ale nie wszystko złoto, co się świeci. Na przykład narzędzie O/RM generuje dynamiczny kod SQL

na żądanie, więc prawdopodobnie nie będzie dobrze pasować do niektórych specyficznych projektów architektonicznych. Oto dwa inne względy warte rozważenia:

- **Krzywa uczenia się** Z mojego doświadczenia wynika, że istnieje problem z krzywą uczenia się zwłaszcza w przypadku narzędzi O/RM lub technologii, takiej jak XAML. Jeśli ktoś nie wie, jak korzystać z nich poprawnie, to może łatwo używać ich w zły sposób, co przełoży się na gorsze wyniki.
- **Operacje na dużych ilościach danych** Choć większość nowoczesnych narzędzi O/RM jest w stanie przeprowadzać operacje masowe, to nie zawsze wybierają one w tym celu najlepszą dostępną metodę. Na przykład obcięcie tabeli jest znacznie efektywniejsze niż wyczyszczenie zawartości *IList<Employee>*, które podczas wykonywania przez narzędzie O/RM jest często tłumaczone na zbiór poleceń *DELETE*.

Istnieje kilka typów narzędzi O/RM na rynku, od bardzo zaawansowanych po proste narzędzia dla nowicjuszy. W kolejnych podrozdziałach pokażę najbardziej godne zaufania narzędzia O/RM i w oparciu o swoje własne doświadczenie przedstawię zalecenia, kiedy korzystać z poszczególnych narzędzi.

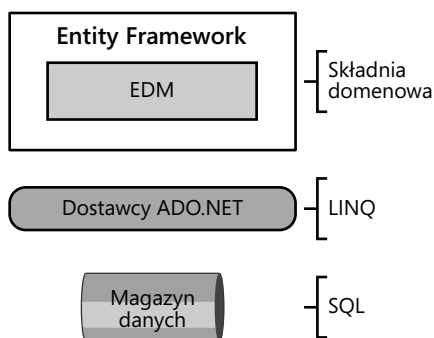
Microsoft Entity Framework

Firma Microsoft wprowadziła system Entity Framework jakiś czas temu. W momencie pisania tej książki platforma Entity Framework jest dostępna w wersji 4 i jest dostarczana z Microsoft .NET Framework 4.0.

Najnowsza wersja tego narzędzia O/RM spełnia wszystkie wymagania standardowego systemu O/RM, w tym późne ładowanie, niezależność od mechanizmów przechowywania danych, dostępność projektanta interfejsu użytkownika oraz tłumacza zapytań, który pozwala programistom korzystać z dobrze znanej składni LINQ.

Możemy podłączać Entity Framework do różnych baz danych, w tym SQL Server, Oracle, DB2, itd. Ta możliwość oraz dostępność narzędzia graficznego do projektowania mapowania sprawia, że Entity Framework jest narzędziem O/RM łatwym do nauczenia się i zastosowania. Jednakże ma chyba mniej możliwości niż inne systemy O/RM dostępne na rynku.

Entity Framework składa się z warstwy, która wykorzystuje istniejącą technologię Microsoft ADO.NET, co sprawia, że nauczenie się składni i zastosowanie Entity Framework jest łatwe dla programistów Microsoft, którzy już znają ADO.NET. Rysunek 4-1 wyświetla podstawową strukturę Entity Framework 4.



RYСУNEK 4-1 Warstwowa struktura Microsoft Entity Framework

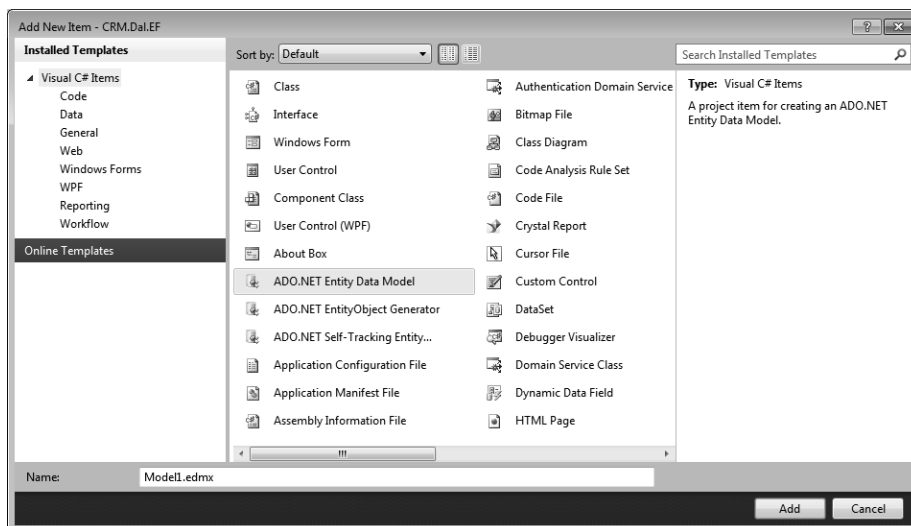
Aby utworzyć nowy model danych wykorzystując Entity Framework 4, musimy zrozumieć, jakie są dostępne modele i dłaczego powinniśmy utworzyć dany typ. Opcje Entity Framework, pokazane na rysunku 4-2, obecnie pozwalają nam tworzyć trzy różne modele:

- **ADO.NET Entity Data Model (model danych jednostki ADO.NET)** Korzystając z tego modelu Entity Framework utworzy nowy, pusty model domenowy. Mamy opcję utworzenia początkowego modelu na podstawie istniejącej bazy danych (podejście „baza danych najpierw”) albo utworzenia pustego modelu domenowego, który później zamapujemy do bazy danych.
- **ADO.NET Entity Object Generator (generator obiektów jednostek ADO.NET)** Korzystając z tej opcji możemy utworzyć silnie typowany kontekst obiektowy i klasy świadome mechanizmów przechowywania danych zaczynając od istniejącej domeny Entity Framework korzystając z funkcji szablonów tekstowych (TT – Text Templating) w Microsoft Visual Studio 2010.



UWAGA W Visual Studio szablon tekstowy jest mieszaniną bloków tekstu i logiki sterującej, które mogą wygenerować plik tekstowy. Logika sterująca jest napisana w postaci fragmentów kodu programu w Microsoft Visual C# lub Microsoft Visual Basic. Wygenerowany plik może zawierać tekst dowolnego rodzaju, taki jak stronę WWW lub plik zasobów, albo kod źródłowy programu w dowolnym języku. Szablony tekstowe mogą być używane do tworzenia części wynikowej aplikacji. Mogą być też używane do generowania kodu, przez co szablony pomagają budować część kodu źródłowego aplikacji.

- **ADO.NET Self-Tracking Entity (śledząca się jednostka ADO.NET)** Korzystamy z tej opcji do generowania typów jednostek, które mają możliwość rejestrowania zmian wartości różnych właściwości niezależnie od Entity Framework.



RYSUNEK 4-2 Dostępne opcje dla Entity Framework wewnątrz Microsoft Visual Studio 2010.

Obecnie w Entity Framework wersja 4 nadal brakuje implementacji pojęcia obiektów POCO. W istocie, jeśli mamy istniejący model domenowy, taki jak ten utworzony w rozdziale 3, to jedynym rozwiązaniem jest utworzenie ręcznego mapowania dla każdej jednostki z pominięciem możliwości zintegrowanego projektanta Entity Framework. Alternatywnie trzeba byłoby użyć projektanta Entity Framework do utworzenia jednostki pośredniczącej dla każdej jednostki domenowej.

Oto zalety i wady zastosowania Entity Framework do mapowania danych:

Zalety

- Entity Framework jest narzędziem łatwym do nauczenia się, łatwym do wykorzystania i ma rozbudowanego kreatora i projektantów interfejsu użytkownika pomagających nam w utworzeniu modelu na podstawie istniejącej bazy danych przy pomocy jedynie kilku kliknięć.
- Jest to narzędzie w pełni zintegrowane z językiem zapytań LINQ w środowisku .NET.
- Jest to produkt firmy Microsoft, który będzie korzystał z ciągłych uaktualnień i rozszerzeń, ma też chyba najbardziej zorganizowany cykl kolejnych wersji.
- Entity Framework jest idealnym narzędziem dla podejścia „baza danych najpierw”, ponieważ wykorzystuje generator kodu T4, który może wygenerować pełen model domenowy zaczynając od istniejącej bazy danych.

Wady

- Entity Framework wymusza na nas użycie LINQ to Entities, rozszerzenia LINQ zaprojektowanego dla Entity Framework i mającego nieco inną składnię niż oryginalny dostawca LINQ.
- Brakuje mu elastycznej obsługi późnego ładowania i obiektów POCO, które są obecne w innych narzędziach O/RM.
- Generowany przez niego kod SQL nie jest idealny, a w niektórych sytuacjach wydajność jest nie do zaakceptowania.

Korzystając z Entity Framework możemy pisać pliki mapowań na dwa sposoby. Jednym ze sposobów jest dekorowanie obiektów POCO przy pomocy atrybutów Entity Framework określających typ danych SQL, nazwę pola w magazynie danych i reguły sprawdzania poprawności dla każdej właściwości. Oto przykład tego podejścia:

```
public class Book
{
    [Key]
    public string ISBN { get; set; }

    [StringLength(256)]
    public string Title { get; set; }

    public string AuthorSSN { get; set; }

    [RelatedTo(RelatedProperty="Books", Key="AuthorSSN", RelatedKey="SSN")]
    public Person Author { get; set; }
}
```

Drugim sposobem jest wykorzystanie trzech dostępnych plików XML do określenia mapowań bazodanowych, mapowań jednostek i mapowań relacji, jak zademonstrowano tutaj:

```
<Association Name="CustomerOrders">
  <End Type="ExampleModel.Customer" Role="Customer" Multiplicity="1" />
  <End Type="ExampleModel.Order" Role="Order" Multiplicity="*">
    <OnDelete Action="Cascade" />
  </End>
  <ReferentialConstraint>
    <Principal Role="Customer">
      <PropertyRef Name="Id" />
    </Principal>
    <Dependent Role="Order">
      <PropertyRef Name="CustomerId" />
    </Dependent>
  </ReferentialConstraint>
</Association>
```

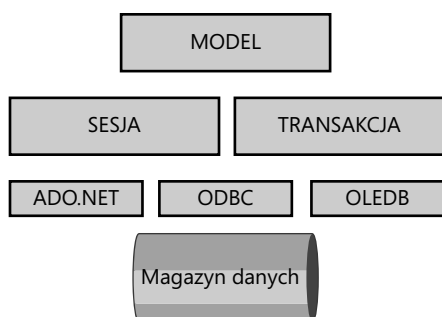
NHibernate

Zapoczątkowane w roku 2005 narzędzie NHibernate jest otwartym systemem O/RM dla .NET, który wywodzi się z udanego otwartego systemu O/RM dla języka Java o nazwie Hibernate. Obecnie jest to jedno z najbardziej popularnych i najczęściej używanych narzędzi O/RM w środowisku otwartego oprogramowania; ma dobre wsparcie ze strony społeczności oraz wiele wtyczek i generatorów kodu.

Zasada działania NHibernate jest dosyć prosta: tworzymy swój model domenowy zgodny z POCO, a następnie decydujemy, jak mapować go na bazę danych. Możemy wykorzystać istniejącą bazę danych (podejście „baza danych najpierw”) albo utworzyć nową (podejście „domena najpierw”).

NHibernate ma wszystkie cechy profesjonalnego narzędzia O/RM, takie jak niewrażliwość na mechanizmy zachowywania danych, strategię opóźnionego ładowania, złożony język zapytań, pełną obsługę LINQ i wiele innych funkcji. Przegląd tego produktu możemy zobaczyć pod adresem <http://nhforge.com>, gdzie znajduje się oficjalna witryna społeczności NHibernate.

Rysunek 4-3 pokazuje strukturę NHibernate wersja 3 (najnowsza wersja w momencie pisania tej książki). U góry diagramu znajduje się model domenowy, który jest całkowicie nieświadomy stosowanego narzędzia O/RM; następnie mapowanie, które można osiągnąć korzystając z osobnego podzespołu składającego się z plików XML lub klas C#; a na koniec jądro silnika NHibernate, które tłumaczy pliki mapowania na obiekty pośredniczące kontaktujące się z docelowym magazynem danych. NHibernate obsługuje tę architekturę korzystając z *obiekту sesji*, który jest kontekstowym obiektem przechowującym stan obiektów modelu domenowego i śledzącym ich zmiany tak, że programista może ustalić, czy dana jednostka jest nowa, czy już istnieje oraz czy była modyfikowana. Wszystkie te kroki są jasne i mogą być wykonywane przy użyciu wzorca transakcji bazodanowej.



RYSUNEK 4-3 Architektura NHibernate

Zalety

- NHibernate jest niezwykle potężnym i elastycznym narzędziem. Możemy uruchamiać wywołania SQL bezpośrednio z sesji O/RM, uzyskiwać dostęp do obiektu połączenia OLEDB/ODBC, zarządzać transakcjami oraz konfigurować procedury składowane i widoki.
- NHibernate jest produktem z otwartym kodem i bardzo dużą społecznością. Dzięki temu możemy korzystać w Internecie z rozbudowanej dokumentacji, takiej jak łatwych do znalezienia książek w formie elektronicznej, blogów, przykładów kodu i przykładowych projektów.
- Możemy znaleźć wiele darmowych wtyczek dla NHibernate, które wspomagają takie zadania, jak konfigurowanie mapowania domeny, pisanie zapytań LINQ, czy korzystanie z projektanta graficznego interfejsu użytkownika wewnątrz Visual Studio.
- NHibernate jest projektem z otwartym kodem źródłowym, co sprawia, że można go łatwo dostosowywać i konfigurować. Możemy uzyskać najnowszą wersję i ją skonfigurować, a nawet zmienić kod podstawowy, jeśli chcemy lub musimy. Poza tym NHibernate ma bardzo rozbudowany silnik śledzący; możemy monitorować tworzone przez niego mapowania i dynamiczny kod SQL.

Wady

- Jest to produkt z kodem otwartym – tak, tak – zdaję sobie sprawę, że dopiero co przedstawiłem zalety projektu otwartego. Ale drugą stroną medalu w przypadku aplikacji z kodem otwartym jest to, że mogą stanowić duże ryzyko, ponieważ nie ma gwarantowanego wsparcia ani żadnej pewności, że produkt będzie dostępny tak długo, jak będzie nam potrzebny.
- Ma bardzo stromą krzywą uczenia się. Na początku NHibernate nie jest łatwym rozwiązaniem. Konfiguracja jest niezwykle skomplikowana (zwłaszcza jeśli wybierzemy metodę XML), a podstawowy silnik ma tysiące opcji i metod, które mogą dać nam słabą wydajność, jeśli będą użyte nieprawidłowo.
- Działa najlepiej razem z niestandardowym kodem SQL wygenerowanym przez ten silnik. Nie powinien być używany ze zmodyfikowanymi instrukcjami SQL albo w operacjach masowych.

Poniższy przykład kodu pokazuje klasyczne mapowanie używane przez NHibernate i jego przodka dla języka Java – Hibernate. W szczególności ten przykład pokazuje plik XML z rozszerzeniem `.hbm.xml`. Wszystko jest zdefiniowane przy użyciu określonego formatu XML, opartego na trzech głównych plikach `.xsd`, które możemy łatwo zaimportować do Visual Studio – po czym Visual Studio zapewni nam mechanizm IntelliSense.

```

<?xml version="1.0" encoding="utf-8" ?>
<hibernate-mapping xmlns="urn:hibernate-mapping-2.2"
  namespace="QuickStart" assembly="QuickStart">

  <class name="Cat" table="Cat">
    <id name="Id">
      <generator class="identity" />
    </id>

    <property name="Name">
      <column name="Name" length="16" not-null="true" />
    </property>
    <property name="Sex" />
    <many-to-one name="Mate" />
    <bag name="Kittens">
      <key column="mother_id" />
      <one-to-many class="Cat" />
    </bag>
  </class>
</hibernate-mapping>

```

FluentNHibernate jest jeszcze jednym narzędziem, które jest warte rozważenia i pozwala nam pisać pliki mapowań korzystając z C#. FluentNHibernate jest otwartą wtyczką dostępną pod adresem <http://fluentnhibernate.org>. W następnym podrozdziale zobaczymy kilka przykładów demonstrujących, jak działa.

Zespół NHibernate niedawno wypuścił nową wersję NHibernate, 3.0, która będzie dostępna w formie RTM od początku roku 2011. W tej wersji zespół dodał pełną obsługę składni LINQ oraz wewnętrzne, płynne mapowanie do kodu C# o nazwie confORM. Wstępna wersja tej wtyczki nie jest jeszcze dostępna, ale kod powinien wyglądać podobnie, jak na poniższym przykładzie:

```

var orm = new ObjectRelationalMapper();
orm.TablePerClass<Animal>();
orm.TablePerClass<User>();
orm.TablePerClass<StateProvince>();
orm.TablePerClassHierarchy<Zoo>();
orm.ManyToMany<Human, Human>();
orm.OneToOne<User, Human>();
orm.PoidStrategies.Add(new NativePoidPattern());

```

Inne narzędzia O/RM dla .NET

Entity Framework i NHibernate są chyba najczęściej używanymi narzędziami O/RM w świecie .NET – być może dlatego, że oba są darmowe. Narzędzie Entity Framework jest dostarczane z platformą .NET Framework, a NHibernate jest narzędziem z otwartym kodem źródłowym.

Jeśli ktoś szuka narzędzia do generowania kodu, które działa również jako O/RM, to można rozważyć narzędzie O/RM firmy trzeciej. Większość z nich nie jest darmowa. Poniższa tabela próbuje zestawiać zalety i wady dwóch takich produktów:

Nazwa	URL	Zalety	Wady
Subsonic	http://subsonicproject.com	Rekord aktywny Automatyczne generowanie kodu	Otwarte źródło, Brak elastyczności
Genome	http://www.genom-e.com	Elastyczne Automatyczne generowanie kodu	Drogie, Uboga dokumentacja

Jednostka pracy

Jeśli planujemy napisanie swojej warstwy DAL przy pomocy narzędzia O/RM, to nie będziemy musieli włożyć wiele wysiłku w pisanie niestandardowego kodu SQL i niestandardowego kodu transakcyjnego do zapisywania lub pobierania jednostek. Z drugiej strony potrzebny nam porządny „orkiestrator”, który będzie odpowiadał za stan jednostki. Na przykład musimy wiedzieć, czy jednostka jest nowa, czy też żądana jednostka jest już w pamięci albo musi zostać pobrana z magazynu danych.

Martin Fowler wprowadził pojęcie jednostki pracy (UoW – Unit of Work) [patrz rozdział 2 „Wzorce projektowe”]. Jednostka pracy odpowiada za utrzymywanie listy obiektów, na które wpływa transakcja biznesowa oraz koordynowanie zapisywania zmian i rozwiązywania problemów ze współbieżnością operacji.

Każde z narzędzi O/RM omówionych w poprzednim podrozdziale wykorzystuje u swoich podstaw pojęcie jednostki pracy. NHibernate wykorzystuje obiekty *Session* oraz *ITransaction*, Entity Framework wykorzystuje klasę *ObjectContext*, a podstawowy LinQToSQL wykorzystuje *DataContext*. Są one implementowane inaczej, ale główne cele są takie same. Poniższy kod pokazuje podstawowe pojęcie jednostki pracy:

```
Public IUnitOfWork<T>
{
    void MarkDirty<T> (T entity);
    void MarkNew<T> (T entity);
    void MarkDeleted<T> (T entity);
    void Commit();
    void Rollback();
}
```

Chodzi tu przede wszystkim o to, że jednostka pracy zajmuje się stanem jednostki danych. Jedyną rzeczą, o której musimy pamiętać, jest oznaczenie jednostki specjalnym statusem, a następnie zatwierdzenie (*Commit*) lub wycofanie (*Rollback*) transakcji biznesowej. Następny przykład pseudokodu pokazuje, jak to działa dla kompletnej transakcji wykorzystującej wzorec UoW:

```
var unitOfWork = Container.Resolve<IUnitOfWork>(); //odwrócenie sterowania dla pobrania
IUnitOfWork
var customer = Factory.CreateCustomer();
```



```

var order = Factory.CreateOrder();
customer.Orders.Add(order);
try {
    unitOfWork.MarkNew(customer); //zaznaczenie nowej jednostki
    unitOfWork.Commit(); //zatwierdzenie zmian w bazie danych
} catch(Exception ex) {
    unitOfWork.Rollback(); //wycofanie, jeśli wystąpi błąd
}

```

Jeśli umieścimy ten wzorec w warstwie DAL, to nie będziemy musieli implementować niestandardowej warstwy DAL dla każdego narzędzia O/RM, z którego będziemy zamierzali skorzystać. Możemy mieć na przykład aplikację złożoną z dwóch warstw DAL: jedną jest Entity Framework, a drugą jest bardziej skomplikowana warstwa DAL wykorzystująca NHibernate. W tym przypadku konieczne może być jedynie zaimplementowanie interfejsu *IUnitOfWork* na dwa różne sposoby, wykorzystanie bardziej odpowiedniego dla każdej warstwy DAL bez konieczności zmieniania kodu samego elementu *UnitOfWork*.

W wielu projektach widziałem zbytnio rozbudowany element *UnitOfWork*. Na przykład widziałem implementację *UnitOfWork*, która była w stanie pobierać dane, wykonywać instrukcje SQL i wiele więcej. Oczywiście w tym momencie nie jest to już tak naprawdę wzorec *UnitOfWork*; jest to wzorec repozytorium (zajmiemy się nim dokładniej w następnych podrozdziałach). Trzeba pamiętać, że jedynym celem *UnitOfWork* jest utrzymywanie stanu zestawu obiektów biznesowych dla powiązanej transakcji biznesowej.

Cykl życia jednostki pracy

Inną ważną kwestią jest cykl życia jednostki pracy. Cykl życia zależy od typu pisanej aplikacji. Na przykład transakcja biznesowa, która występuje w aplikacji klienckiej WPF, może się bardzo różnić od transakcji biznesowej, która ma miejsce w bardziej zorientowanej na WWW aplikacji Silverlight.

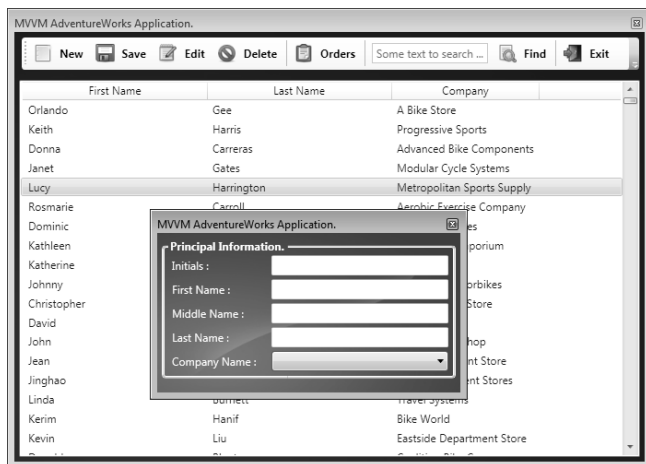
W klasycznym wzorcu MVVM zwykle lubię kojarzyć cykl życia elementu *UnitOfWork* z odpowiadającym mu powiązaniem widok/model widoku. W ten sposób wszystko, co dzieje się w modelu widoku, jest obejmowane przez transakcję UoW. Jeśli muszę pracować nad aplikacją WWW, to staram się uzyskać pomoc z obiektu *HttpContext* i przechowywać w nim element UoW. Trzeba jednak pamiętać, że wszystko zależy od typu pisanej aplikacji; cykl życia elementu UoW jest zależny od kontekstu.

Przewodnik firmy Oracle po transakcjach bazodanowych (http://download.oracle.com/docs/cd/B14099_19/web.1012/b15901/xactions002.htm#i1132715) definiuje jednostkę pracy jako kontekst biznesowy, który otwiera transakcję, jak tylko działanie biznesowe się rozpoczyna i steruje tym działaniem poprzez zatwierdzanie lub wycofywanie tej transakcji. Do nas zależy ustalenie, kiedy działanie biznesowe jest gotowe i kiedy trzeba zatwierdzić jakieś zmiany.

Identyfikowanie transakcji biznesowej

W aplikacji MVVM transakcja biznesowa jest zwykle ściśle związana z połączeniem widok/model widoku, ale w niektórych przypadkach to wymaganie może nie być spełnione. Na przykład możemy mieć główny widok, który wyświetla wszystkich dostępnych klientów. Jednocześnie możemy mieć polecenie, które pozwala użytkownikom dodawać, modyfikować lub usuwać klienta z tej listy.

Ponieważ wiele widoków zawiera podwidoki, to zwykle istnieje też hierarchia modeli widoku; związek z transakcją jest zwykle na poziomie ekranu i może się składać z więcej niż jednego powiązania widok/model widoku. Rysunek 4-4 pokazuje praktyczny przykład w WPF.



RYSUNEK 4-4 Przykładowa aplikacja WPF

Można zauważyć, że występują tutaj dwa typy transakcji biznesowych. Pierwszy ma miejsce od razu po załadowaniu głównego widoku. W tym momencie wywołujemy element UoW i pobieramy wszystkie dostępne dane z magazynu danych. Następnie wyświetlamy te dane w widoku wykorzystując swój model widoku. Ta transakcja kończy się, jak tylko dane zostają wyświetlone w widoku.

Druga transakcja może mieć miejsce za każdym razem, gdy dodajemy, modyfikujemy lub usuwamy klienta. Ta transakcja jest dość prosta, a przy tym wymagająca. Na przykład kliknięcie polecenia Add Customer (Dodaj klienta) tworzy nowy widok/model widoku z powiązaniem elementem UoW, który może zostać zatwierdzony lub nie w zależności od określonej logiki biznesowej. W następującym przykładzie dane są zatwierdzane w wywołaniu polecenia. Kliknięcie Save (Zapisz) dodaje nowego klienta do bieżącej sesji elementu UoW. W tym momencie powinniśmy zatwierdzić transakcję i odświeżyć główny model widoku. Czy to jest unikalna transakcja, czy zestaw transakcji? No cóż, w sensie transakcji biznesowej jest to tylko jedna transakcja, która zawiera następujące kroki:

```

Using (var uow = Container.Resolve<IUnitOfWork>())
{
    try{
        uow.BeginTransaction();
        var customer = CustomerViewModel.GetModel();
        uow.SaveOrUpdate(customer);
        MainViewModel.Refresh();
        uow.Commit();
    }catch(Exception ex){
        uow.Rollback();
        MainViewModel.Refresh();
    }
}

```

Powyższy pseudokod zaczyna się od pozyskania jednostki *Customer* z modelu widoku *CustomerViewModel*, a następnie próbuje dodać lub zaktualizować tę jednostkę. Następnie odświeża model widoku *MainViewModel*. Jeśli coś się nie powiedzie, kod wycofuje całą transakcję i odświeża model widoku *MainViewModel*, który nie powinien wyświetlić nowego klienta, ponieważ transakcja się nie udała.

Podsumowując, transakcja biznesowa obsługiwana w elemencie UoW nie powinna być traktowana tak samo, jak prosta transakcja bazodanowa; zamiast tego należy ją traktować jako zestaw operacji, które wykonają jeden lub kilka „kroków” biznesowych. Obiekt *ITransaction* w NHibernate na przykład może obsługiwać wiele wywołań SQL wykonując je tylko wtedy, gdy wywołamy polecenie *Commit*. Jednocześnie w oparciu o naszą konfigurację zaktualizuje bieżącą sesję i oczywiście listę dostępnych klientów, aby odpowiadała zmianom dokonany przez interfejs użytkownika.

Wzorzec repozytorium

W poprzednim podrozdziale zobaczyliśmy samo sedno warstwy DAL, element UoW. Niestety przy pomocy podstawowego wzorca UoW możemy jedynie dodawać, usuwać lub aktualizować jednostki i wykonywać transakcje biznesowe, ale do pełnej warstwy DAL czegoś nam jeszcze brakuje. Musimy być w stanie pobierać zbiór jednostek, ładować z opóźnieniem powiązane jednostki podrzędne i nadrzędne oraz tworzyć zapytania i dzielić wyniki na strony bez wpływu na wydajność bazy danych. Ze względu na te i inne wymagania biznesowe warstwy DAL musimy skorzystać z wzorca repozytorium.

Martin Fowler po raz pierwszy wprowadził pojęcie wzorca repozytorium (Repository) w swojej książce *Patterns of Enterprise Application Architecture* (patrz rozdział 2). Repozytorium powinno działać jako przechowywany w pamięci zbiór jednostek i powinno umożliwiać kwerendy oraz procesy pobierania danych dla tych jednostek bez wpływu na wydajność bazy danych.

Podstawowe repozytorium powinno obsługiwać dodawanie i usuwanie jednostek oraz aktualizowanie istniejących jednostek. Powinno też zapewniać łatwy sposób na ich pobieranie i odpytywanie. Poniższy pseudokod wyświetla typowy wzorzec repozytorium:

```

interface ICustomerRepository
{
    void AddCustomer(Customer customer);
    void RemoveCustomer(Customer customer);
    void UpdateCustomer(Customer customer);

    IQueryable<Customer> GetCustomers();
    Customer GetCustomer(object primaryKey);
}

```

Korzystając z tego „kontraktu” możemy zaimplementować konkretne repozytorium *CustomerRepository*, które będzie w stanie wykonywać wszelkiego rodzaju operacje CRUD – tworzenie (Create), odczytywanie (Read), aktualizowanie (Update) lub usuwanie (Delete) w magazynie danych korzystając z elementu UoW. Oczywiście każda z tych operacji może być zawarta w jednej lub wielu transakcjach biznesowych. W szczególności warto zwrócić uwagę na metodę *GetCustomers*, która, jak można zauważyć, nie zwraca wcale „prawdziwej” kolekcji; zwraca kolekcję *IQueryable<T>*. Dzieje się tak dlatego, że *IQueryable* jest jedynym typem kolekcji, który jest w stanie wykonywać wywołania do bazy danych, gdy mamy dostęp tylko do jednego z elementów w kolekcji. To oznacza, że możemy stosować dodatkowe filtry LINQ wobec tej kolekcji, zanim wykonane zostanie ostateczne wywołanie do bazy danych. Na przykład, jeśli napiszemy następujący kod, a narzędziem O/RM będzie Entity Framework lub NHibernate:

```

var repository = Container.Resolve<ICustomerRepository>();
var customers = repository.GetCustomers()
    .Where(x => x.Company == "Microsoft")
    .OrderBy(x => x.FirstName);
// wywołanie do bazy danych
customers.ToList();

```

to ostateczny kod T-SQL wygenerowany przez nasze narzędzie O/RM, o ile będzie ono dobrze skonfigurowane, będzie wyglądać podobnie, jak poniżej:

```

SELECT * FROM
TBL_CUSTOMER C WHERE C.COMPANY = 'Microsoft'
ORDER BY C.FIRSTNAME

```

Bardziej szczegółową analizę typu kolekcji *IQueryable<T>* można znaleźć pod adresem <http://msdn.microsoft.com/en-us/library/system.linq.iqueryable.aspx>. *IQueryable<T>* jest częścią przestrzeni nazw *System.Linq*, a nie częścią przestrzeni nazw *System.Collections*, jak inne kolekcje dostępne w .NET. Oczywiście, aby korzystać z niej właściwie, musimy mieć pewność, że nasz system O/RM w pełni obsługuje LINQ lub jednego z dostawców LINQ.

Aby uzyskać większy stopień ogólności, możemy rozważyć zastosowanie typów ogólnych .NET i utworzyć bardziej uniwersalny kontrakt wzorca repozytorium podobny do poniższego:

```

Interface IRepository<T>
{

```

```

void Add(T entity);
void Remove(T entity);
void Update(T entity);

T Get(object primaryKey);
IQueryable<T> GetEntities();
}

```

Ten kod tworzy bardziej ogólny kontrakt, który możemy wielokrotnie wykorzystywać w swoich aplikacjach i który w połączeniu z ogólnym elementem UoW stanowi bardzo elastyczną warstwę DAL wielokrotnego użytku. Później w tym rozdziale zbudujemy proste klasy *UoW* oraz *IRepository* korzystając z typów ogólnych i zobaczymy, jak łatwo jest podłączać te dwa wzorce do tego samego projektu MVVM korzystając z dwóch różnych narzędzi O/RM.

Przykładowa implementacja ogólnego repozytorium powinna wyglądać następująco:

```

public GenericRepository<T> : IRepository<T>
{
    private IUnitOfWork uow;

    public GenericRepository()
    {
        this.uow = Container.Resolve<uow>();
    }

    public void Add<T> (T entity){
        try
        {
            uow.BeginTransaction();
            uow.Add(entity);
            uow.Commit();
        }catch (Exception ex){
            uow.Rollback();
        }
    }
}

```

Programowanie sterowane testami: warstwa danych

W rozdziale 2 podkreśliłem, jak ważne jest programowanie sterowane testami (TDD) i dlaczego trzeba je wprowadzić na początku procesu tworzenia oprogramowania. Oczywiście warstwę danych również trzeba testować.

Przy testowaniu warstwy danych powinniśmy iść w dwóch różnych kierunkach w zależności od układu warstwy DAL. Pierwszym krokiem jest testowanie mapowania w O/RM; musimy mieć pewność, że jednostka jest zamapowana poprawnie w narzędziu O/RM i że każde pole jest zamapowane na odpowiednie pole w bazie danych. Na przykład nie chcemy, aby właściwość *FirstName* jednostki *Customer* była mapowana na pole *LastName* w tabeli *Customer*.

Aby przetestować mapowanie, powinniśmy wykonać kilka prostych kroków zależnych od używanego systemu O/RM. Po pierwsze, chcemy mieć pewność, że podczas tworzenia i zapisywania nowej jednostki, na przykład *Customer*, jej wartości są poprawnie zachowywane w bazie danych. Aby przeprowadzić ten test, po prostu musimy utworzyć nową jednostkę, zapisać ją, pobrać, a następnie porównać znowu z wartościami statycznymi.

```
public void CanSaveFirstName()
{
    var firstName = "John";
    var customer = Factory.Create<Customer>();
    customer.FirstName = firstName;
    GenericRepository<Customer>.Add(customer);
    var expected = GenericRepository<Customer>.Get(customer.PrimaryKey);
    Assert.AreEqual(expected.FirstName, firstName);
}
```

Takie operacje mogą zabierać nieco czasu, ale są konieczne, gdy musimy mieć stuprocentową pewność, że mapowanie zostało poprawnie wykonane i nie brakuje żadnej właściwości. Jeśli korzystamy z Entity Framework, to być może nie musimy wykonywać tego zestawu kroków, ponieważ projektant interfejsu użytkownika ułatwia zobaczenie, czy mapowanie zostało przeprowadzone poprawnie, zwłaszcza gdy generujemy swoją domenę zaczynając od istniejącej bazy danych.

Jeśli planujemy budowę aplikacji zaczynając od domeny i ręcznie piszemy mapowania dla swoich obiektów POCO, to musimy zagwarantować poprawność mapowania korzystając z podejścia TDD, co może być bardzo kosztowne, jeśli chodzi o czas i zasoby.

Powinniśmy też pamiętać, że magazyn danych używany na potrzeby TDD nie powinien być tym samym, którego używamy w końcowej aplikacji, w przeciwnym razie otrzymamy zestaw „fałszywych” rekordów w produkcyjnym magazynie danych.

Drugi typ testów powinien być wykonywany tylko wtedy, gdy nie mapujemy swojego modelu automatycznie i jeśli określamy inne nazwy dla pól w magazynie danych niż nazwy używane dla pól jednostki. W mojej firmie na przykład pracujemy z bardzo prywatnymi informacjami, więc nie możemy korzystać z faktycznych nazw dla pól bazodanowych, ponieważ stanowiłoby to potencjalne naruszenie zasad bezpieczeństwa. Aby to osiągnąć, zapisujemy tabele i pola używając liczbowych mapowań, jak pokazano w poniższej tabeli:

Tabela	Pole	Opis
Tbl801	801_1	Id
Tbl801	801_2	FirstName

Ten typ nazewnictwa może być trudny do odszyfrowania i jest też podatny na błędy. Z tego powodu, gdy tworzę klasę mapującą lub XML dla określonej jednostki, muszę mieć pewność, że mapowanie jest wykonane poprawnie – a jedynym sposobem, żeby

to zrobić, jest wykonanie instrukcji *SELECT* na magazynie danych, a następnie sprawdzenie, czy mapowane pole w mojej jednostce jest takie samo jak w magazynie danych.

Jeśli planujemy pracę z NHibernate, to możemy rozważyć użycie bardzo wygodnego narzędzia do zapisywania mapowań w C# o nazwie FluentNHibernate (dostępne pod adresem <http://fluentnhibernate.org>). To narzędzie szeroko wykorzystuje wyrażenia lambda do tworzenia plików mapowań. Jest też bardzo przydatnym składnikiem TDD, którego możemy używać do sprawdzania poprawności mapowań i ich funkcjonowania w magazynie danych.

Poniższy przykład pokazuje, jak FluentNHibernate pozwala nam pisać „płynne” pliki mapowań:

```
public class CatMap : ClassMap<Cat>
{
    public CatMap()
    {
        Id(x => x.Id);
        Map(x => x.Name)
            .Length(16)
            .Not.Nullable();
        Map(x => x.Sex);
        References(x => x.Mate);
        HasMany(x => x.Kittens);
    }
}
```

Następny przykład kodu pokazuje, jak możemy testować swoje mapowanie do magazynu danych wykorzystując składnik FluentNHibernate. Kod ten tworzy nową jednostkę *Employee*, zachowuje ją w bazie danych, a następnie sprawdza, że jednostka została poprawnie zachowana.

```
[Test]
public void CanCorrectlyMapEmployee()
{
    new PersistenceSpecification<Employee>(session)
        .CheckProperty(c => c.Id, 1)
        .CheckProperty(c => c.FirstName, "John")
        .CheckProperty(c => c.LastName, "Doe")
        .VerifyTheMappings();
}
```

Jeszcze jedna kwestia, którą trzeba wziąć pod uwagę: pamiętajmy, że mapowanie domeny w normalnej aplikacji MVVM ma miejsce tylko raz podczas całego cyklu życia aplikacji. Oczywiście będziemy musieli zmieniać i modyfikować domenę oraz magazyn danych podczas życia aplikacji, ale główny schemat nie powinien się zmieniać. Dobra warstwa TDD dla warstwy DAL zagwarantuje, że każda zmiana, której dokonamy w domenie, zostanie prawidłowo odzwierciedlona w odpowiadającym jej magazynie danych.

Budowanie rozproszonej warstwy danych przy pomocy RIA i WCF

To, co zobaczyliśmy do tej pory, wystarcza, jeśli nasza warstwa DAL jest napisana w C# lub Visual Basic .NET i jeśli jest używana przez „normalną” aplikację .NET wykorzystującą wspólne środowisko CLR, na przykład przez aplikację kliencką WPF. To oznacza, że możemy napisać swoją niestandardową i ogólną warstwę DAL tylko raz i wielokrotnie wykorzystywać ją w swojej aplikacji WPF, aplikacji ASP.NET lub w prostej aplikacji Windows Service. Jeśli wykorzystujemy podejście trójpoziomowe, to sprawy nieco się komplikują. Dzieje się tak dlatego, ponieważ będziemy musieli dołączyć dodatkową abstrakcję używając Windows Communication Foundation (WCF) lub jakiejś technologii usług WWW, która zapewni atomowe metody do wywoływania warstwy DAL przechowywanej na poziomie serwera aplikacyjnego.

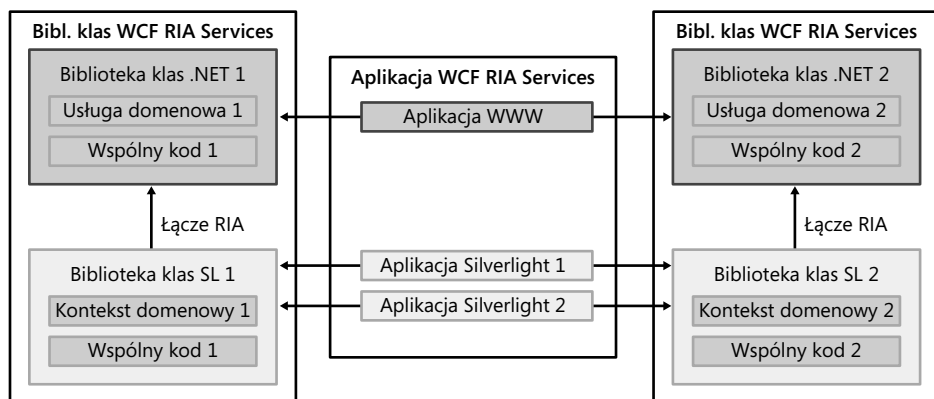
Niestety po przejściu na aplikację Silverlight ten proces nie zda egzaminu, ponieważ Silverlight działa w ramach mniejszej i prostszej wersji środowiska CLR zaprojektowanej wyłącznie dla Silverlight, która nie jest (obecnie) w stanie uruchamiać bibliotek DLL skompilowanych dla środowiska Common CLR. Na szczęście istnieje rozwiązanie nie wymagające ponownego kompilowania wszystkich warstw w celu spełnienia wymagań Silverlight; inaczej za każdym razem, gdy zmienimy coś w domenie lub w warstwie DAL, to będziemy musieli również przekompilować wszystko tylko na potrzeby aplikacji Silverlight.

Firma Microsoft wprowadziła usługę WCF RIA Service for Silverlight. Pojęcie tej usługi jest bardzo proste, ale ma duży potencjał. WCF jest rozbudowaną technologią, która pozwala nam komunikować się pomiędzy warstwami przy użyciu protokołu SOAP – Simple Object Access Protocol (jak usługa WWW) i udostępniać dane oraz struktury danych pomiędzy domeną a innymi warstwami przy użyciu XML. Możemy wykorzystać usługi WCF RIA Services do udostępniania swojej warstwy DAL i swojej domeny skompilowanych dla środowiska Common CLR aplikacjom działającym w Silverlight (patrz rysunek 4-5) bez konieczności ponownego kompilowania wszystkiego po każdej zmianie.

Na rysunku 4-5 widać, że istnieją dwie wyraźne granice, z których jedną jest sama aplikacja Silverlight wraz ze „sklonowaną” logiką aplikacji, wykorzystywaną też przez usługę WWW i druga granica złożona z warstwy danych i bazy danych też udostępniana przez usługę WCF Service. W tym przypadku istnieją co najmniej dwa różne poziomy: jednym jest aplikacja MVVM w technologii Silverlight, a drugim jest poziom danych złożony z magazynu danych i warstwy danych.

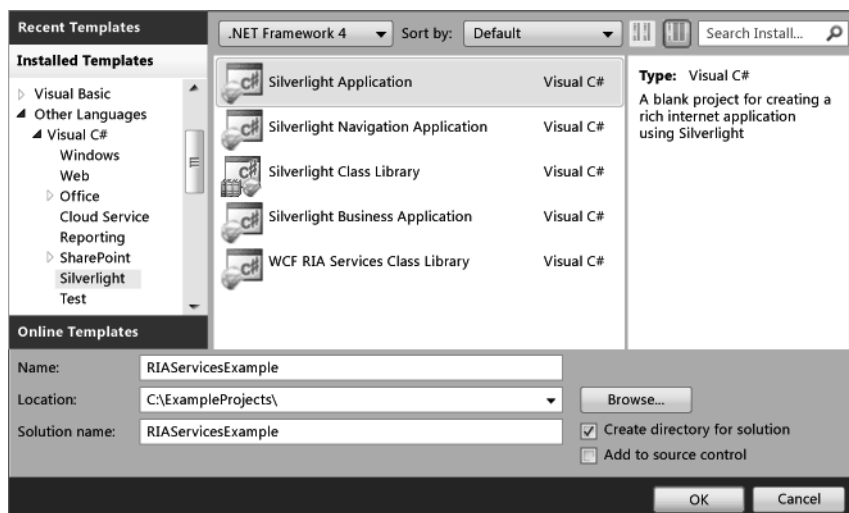
Możemy wykorzystywać usługi WCF RIA Services w swoich aplikacjach WPF, jeśli chcemy, ale pierwotnie zostały one zaprojektowane w celu rozwiązania problemu polegającego na tym, że Silverlight generuje kod wyłącznie dla środowiska Silverlight CLR.

Aby utworzyć aplikację WCF RIA, musimy mieć podstawową aplikację kliencką Silverlight. Ta aplikacja będzie zawierać projekt Silverlight i witrynę WWW w technologii ASP.NET lub ASP.NET MVC utrzymującą skompilowaną aplikację Silverlight.

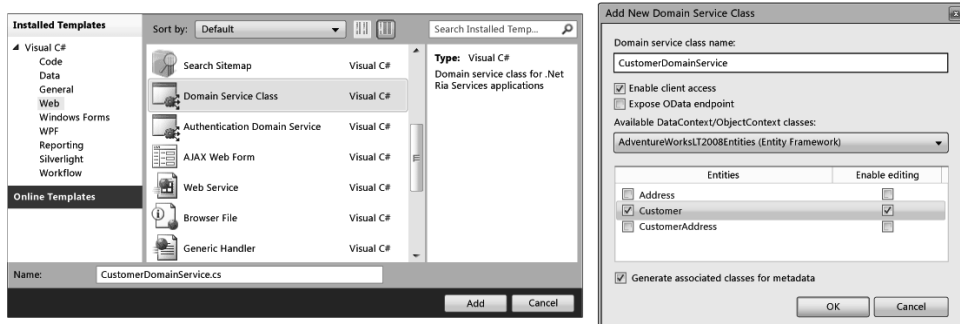


RYСУNEK 4-5 Podstawowa struktura usług WCF RIA Services

Po utworzeniu bazowego klienta Silverlight musimy dodać nową usługę WCF RIA Service do aplikacji WWW, jak pokazano na rysunku 4-6. W tym momencie jesteśmy w stanie udostępnić w aplikacji Silverlight warstwę DAL, domenę i co tylko potrzebujemy z poziomu Common CLR. Rysunek 4-7 podsumowuje te kroki i pokazuje, jak proste jest wykorzystanie i udostępnianie domeny w aplikacji Silverlight.



RYСУNEK 4-6 Jak utworzyć aplikację Silverlight, która zawiera usługi WCF RIA Services.



RYSUNEK 4-7 Dwa proste kroki dodające istniejącą domenę do usługi WCF RIA Service.



UWAGA Ściśle mówiąc zwykle będziemy korzystać z tego kreatora przy tworzeniu modelu domenowego opartego dokładnie na schemacie bazodanowym wybranej tabeli, co nie jest typowe dla prawdziwej aplikacji biznesowej.

Po dodaniu odwołań do aplikacji Silverlight jesteśmy gotowi na odpytywanie swojej domeny przy wykorzystaniu istniejącej warstwy DAL z poziomu kodu Silverlight. Ponieważ usługi RIA Services mają pełną obsługę kodu XAML, jest to też bardzo dobra kombinacja dla wzorca MVVM.

Poniższy kod ładuje dane do modelu widoku (*ViewModel*) i wiąże je z siatką w widoku Silverlight:

```
public class GridViewModel : ViewModel
{
    private CustomerDomainContext customerContext = new CustomerDomainContext();
    public LoadOperation<Customer> VMDataSource;
    public GridViewModel()
    {
        InitializeView();
        VMDataSource = this.customerContext.Load(
            this.customerContext.GetCustomersByLastNameLetterQuery(LetterValue.Text),
            CustomerLoadedCallback, null);
    }
}
//XAML

<Page.Resources>
    <vm:GridViewModel />
</Page.Resources>
<myGrid:DataGrid DataSource="{Binding Path=VMDataSource}" ...
```



UWAGA Usługi WCF RIA Services dobrze współpracują z O/RM, Entity Framework i NHibernate.

Ten przykład wprowadził *CustomerDomainContext* – niestandardową klasę, która dziedziczy po klasie *DomainContext* i jest tworzona przez kreatora WCF RIA Services Wizard. Klasa kontekstu domenowego jest generowana w projekcie klienckim dla każdej usługi domenowej w projekcie serwerowym. Wywołujemy metody w klasie kontekstu domenowego odpowiadające metodom usługi domenowej, z których chcemy skorzystać.

Rozwój WCF RIA Services można śledzić na bieżąco na oficjalnej witrynie WWW firmy Microsoft dla tej technologii (<http://www.silverlight.net/getstarted/riaservices>), gdzie można znaleźć dużo przydatnego kodu, przykładów i samouczków. Jeśli mamy istniejącą warstwę DAL i planujemy zbudowanie swojej następnej aplikacji biznesowej przy użyciu Silverlight, to powinniśmy rozważyć wykorzystanie usług WCF RIA Services w celu zaoszczędzenia czasu i pracy.

Oczywiście, jeśli planujemy skorzystanie z tego podejścia, to będziemy musieli wziąć pod uwagę dodatkowe kwestie związane z podejściem SOA, takie jak współbieżność rekordów, opóźnienia danych i wąskie gardła w sieci.

Kod przykładowy: warstwa dostępu do danych aplikacji CRM

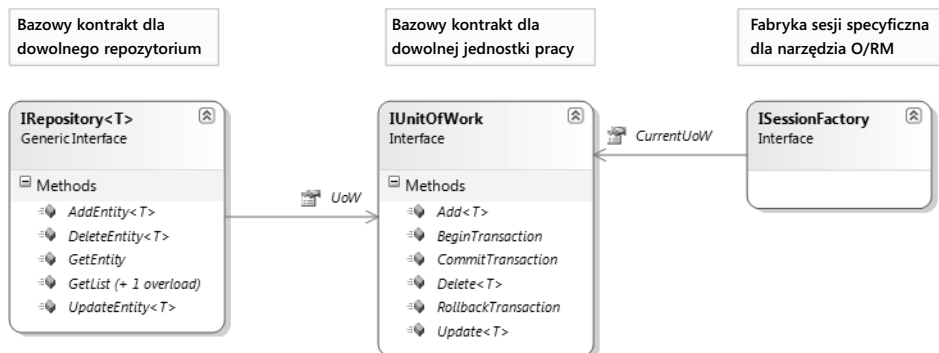
W podrozdziale dotyczącym kodu przykładowego w rozdziale 3 utworzyliśmy model domenowy dla przykładowej aplikacji CRM składający się z dwóch głównych grup jednostek: jednostki *Person* i jednostki *Order*. W tym rozdziale zamapujemy tę domenę do bazy danych wykorzystując dwa różne narzędzia O/RM: Entity Framework i NHibernate.

Elastyczny interfejs *IUnitOfWork*

Zanim utworzymy mapowanie domenowe, musimy utworzyć podstawowy interfejs *IUnitOfWork*, który będzie używany w poszczególnych warstwach aplikacji MVVM. Diagram klas na rysunku 4-8 przedstawia interfejs *IUnitOfWork*, który jest bardzo zbliżony do elementu UoW objaśnionego wcześniej w tym rozdziale. Element UoW udostępnia trzy główne polecenia do zmieniania stanu jednostki (*Create*, *Update*, *Delete*), a także funkcje do wykonywania tych poleceń w kontekście transakcyjnym.

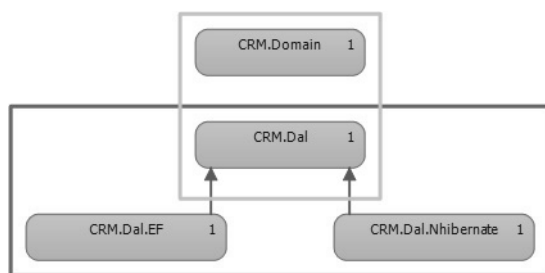
Element UoW będzie specyficzny dla każdego narzędzia O/RM, ponieważ Entity Framework wykorzystuje *ObjectContext*, natomiast NHibernate wykorzystuje interfejs *ISession*. Z tego powodu interfejs *IUnitOfWork* nie będzie udostępniał prawdziwego obiektu *DataContext*, a jedynie podstawowe polecenia potrzebne do wykonywania instrukcji SQL. Aplikacja wykorzystuje interfejs *ISessionFactory* z tego samego powodu; w zależności od używanego narzędzia O/RM będziemy tworzyć specyficzny, konkretny obiekt *UnitOfWork*, który będzie miał zależny od O/RM element *DataContext*.

Na koniec potrzebny nam będzie ogólny element *Repository*, który będzie przeprowadzał operacje CRUD w kontekście transakcyjnym. Obiekt *Repository* będzie udostępniać aktualny interfejs *IUnitOfWork* tak, że będziemy w stanie wykonywać konkretne polecenia, jak również predefiniowane operacje CRUD dostępne poprzez *Repository*.



RYСУNEK 4-8 Abstrakcyjna warstwa CRM.Dal: podstawowa warstwa dla każdej warstwy DAL.

Musimy też utworzyć dwie dodatkowe warstwy: jedną dla Entity Framework i jedną dla NHibernate. Jak wspominałem wcześniej, ten krok jest wymagany, ponieważ możemy wybrać, z którego narzędzia O/RM chcemy korzystać, ale nie możemy stosować tego samego konkretnego elementu *UnitOfWork* wobec dwóch różnych narzędzi O/RM, ponieważ zarządzają one przechowywaniem jednostek i kontekstem danych w różny sposób. Ostateczny wynik powinien wyglądać podobnie jak diagram na rysunku 4-9. Trzeba pamiętać, że cały kod przywoływany w tej książce jest dostępny w przykładowej aplikacji CRM zawartej w dostępnym do pobrania pakiecie dodatkowym dla tej książki.



RYСУNEK 4-9 Domena CRM z gotową warstwą danych

Rysunek 4-9 pokazuje, że domena jest całkowicie nieświadoma wykorzystywanej techniki zachowywania danych. Jednocześnie abstrakcyjna warstwa danych, którą będziemy wykorzystywać w swojej aplikacji MVVM, nie zna konkretnej implementacji.

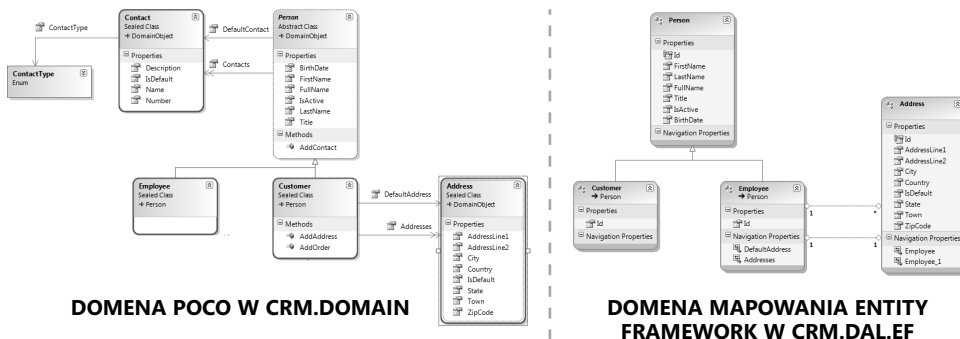
Mapowanie modelu domenowego przy użyciu Entity Framework

Korzystając z Entity Framework możemy budować domenę korzystając z dwóch różnych podejść DDD: tworząc najpierw domenę lub najpierw bazę danych. Jak to zostało omówione w rozdziale 3, powinniśmy unikać podejścia budującego najpierw bazę danych, o ile nie wynika to z konkretnej bazy danych odziedziczonej po starszym projekcie. Ponieważ aplikacja CRM jest nowa, nie będziemy związani z konkretnym schematem bazodanowym, więc zastosujemy tutaj podejście budujące najpierw domenę.

Przed wersją .NET Framework 4.0 nie byliśmy w stanie stworzyć kompletnej domeny opartej o obiekty POCO i podłączyć jej do nowego modelu Entity Framework, jednak w wersji 4 możemy teraz zrealizować to zadanie bez konieczności przepisywania czegośkolwiek. Niestety Entity Framework wymaga zestawu klas pośredniczących, które działają jako obiekty mapujące dla jednostek domenowych POCO. Dlatego musimy w projektancie Entity Framework zaimplementować ten sam model, który mamy w domenie. To obowiązkowe wymaganie powoduje istotny problem; Entity Framework musi widzieć, jak ładować z opóźnieniem i jak zachowywać jednostkę POCO. Ponieważ jednostka jest obiektem POCO, to jedynym sposobem, aby tego dokonać, jest utworzenie obiektu pośredniczącego, który zastępuje w jednostce POCO specyficzną konfigurację dla bazy danych.

Na początek utworzymy nowy projekt i nazwijmy go **CRM.DAL.EF**. Będzie to konkretna implementacja warstwy danych dla Entity Framework. Następnie dodajmy te trzy odwołania: *CRM.DAL* (nasza bazowa warstwa danych), *System.Data.Entity* (Entity Framework) i *System.ComponentModel.Composition* (MEF do projektowania wtyczek).

Następnie dodajmy nowy pusty plik .edmx (ADO.NET Entity Model). Musimy utworzyć te same jednostki, które istnieją w naszej podstawowej domenie. Ten krok jest dość frustrujący i nazbyt czasochłonny, ale jest to jedyne aktualnie dostępne rozwiązanie, żeby nasze domenowe obiekty POCO były nieświadome Entity Framework. Końcowy wynik powinien być podobny do rysunku 4-10.



RYСУNEK 4-10 Porównanie diagramów klas: domena z obiektami POCO a domena Entity Framework.

W serwisie www.codeplex.com znalazłem kilka niestandardowych narzędzi, które mogą nam pomóc w wygenerowaniu niestandardowego diagramu mapowania Entity Framework z istniejącej domeny POCO. Oczywiście do tego typu podejścia najlepiej pasuje generowanie modelu Entity Framework z istniejącej bazy danych, a następnie budowanie obiektów pośredniczących dla modelu domenowego. W tym rozdziale widzieliśmy wady i zalety tego narzędzia O/RM. Nie powinniśmy korzystać z Entity Framework, jeśli planujemy zachowanie modelu domenowego całkowicie w postaci obiektów POCO, ponieważ w aktualnej wersji (CTP 4) nadal brakuje procesu mapowania pomiędzy obiektami pośredniczącymi Entity Framework a jednostkami domenowymi POCO.

Tworzenie konkretnej warstwy DAL dla Entity Framework

Następnym krokiem jest utworzenie obiektu pośredniczącego *ObjectContext*, który będzie mógł zachowywać jednostki POCO. Poniższy kod mapuje każdą jednostkę pośredniczącą Entity Framework na odpowiadającą jej jednostkę w *CRM.Domain*. W tym celu musimy dodać odwołanie do warstwy *CRM.DAL.EF* wskazujące na *CRM.Domain*. Nie będziemy ponownie wykorzystywać tej części warstwy DAL, ponieważ jest ona specyficzna dla przykładowej aplikacji MVVM dla tej specyficznej domeny.

```
namespace CRM.DAL.EF
{
    public class CRMObjectContext : ObjectContext
    {
        public CRMObjectContext(string connectionString) : base(connectionString)
        {
            /// <summary>
            /// Pobiera lub ustawia pracowników.
            /// </summary>
            /// <value>Pracownicy.</value>
            public ObjectSet<Employee> Employees { get; set; }

            /// <summary>
            /// Pobiera lub ustawia klientów.
            /// </summary>
            /// <value>Klienci.</value>
            public ObjectSet<Customer> Customers { get; set; }
        }
    }
}
```

Teraz gdy mamy *ObjectContext*, możemy zacząć implementować konkretną warstwę DAL po stronie Entity Framework. Pierwszą klasą, którą musimy zaimplementować, jest oczywiście klasa *UnitOfWork*, która będzie dziedziczyć po *CRM.DAL.IUnitOfWork*. Musimy poinformować MEF, że wykorzystujemy wtyczkę *IUnitOfWork* dodając atrybut *[Export]* do konkretnej implementacji *UnitOfWork*, jak pokazano w następującym kodzie:

```
[Export(typeof(IUnitOfWork))]
public class UnitOfWork : IUnitOfWork
```

```

{
    private ObjectContext orm;

    /// <summary>
    /// Inicjuje nowe wystąpienie klasy <see cref="UnitOfWork"/>.
    /// </summary>
    /// <param name="orm">ORM.</param>
    public UnitOfWork(ObjectContext orm)
    {
        this.orm = orm;
    }
}

```

W konstruktorze elementu UoW wstrzykniemy aktualny *ObjectContext*, który jest pośrednikiem używanym w naszym modelu domenowym.

Następny krótki przykład kodu pokazuje prostą metodę *Add*. Po prostu musimy dodać jednostkę do *ObjectContext*. Następnie metoda *CommitTransaction* spróbuje zaktualizować sesję Entity Framework w kontekście transakcyjnym. Jeśli operacja *Add* się nie uda, to Entity Framework nie potrzebuje metody wycofującej, ponieważ automatycznie wycofa stan jednostek:

```

/// <summary>
/// Dodaje określoną jednostkę.
/// </summary>
/// <typeparam name="T"></typeparam>
/// <param name="entity">Jednostka.</param>
public void Add<T>(T entity)
{
    try
    {
        this.orm.AddObject(EntitySetName, entity);
    }
    catch (Exception ex)
    {
        throw new Exception(string.Format(
            "An error occurred during the Add Entity.\r\n{0}", ex.Message));
    }
}

/// <summary>
/// Zatwierdza transakcję.
/// </summary>
public void CommitTransaction()
{
    try
    {
        if (tx == null)
        {
            throw new TransactionException(
                "The current transaction is not started!");
        }
        orm.SaveChanges(false);
        tx.Complete();
        orm.AcceptAllChanges();
    }
}

```

```

        catch (Exception ex)
        {
            throw new Exception(string.Format(
                "An error occurred during the Commit transaction.\r\n{0}", ex.Message));
        }
        finally
        {
            tx.Dispose();
        }
    }
}

```

Drugą klasą, którą zaimplementujemy, jest ogólne repozytorium *Repository*, które dziedziczy po *CRM.DAL.IRepository* i udostępnia wszystkie dostępne metody z interfejsu *IRepository*. *Repository* będzie też mieć atrybut *Export* tak, abyśmy mogli skorzystać z MEF w warstwie MVVM do ładowania wybranego systemu O/RM na żądanie.

```

[Export(typeof(IRepository))]
public class Repository : IRepository
{
    [Import]
    private IUnitOfWork uow;

    /// <summary>
    /// Dodaje jednostkę.
    /// </summary>
    /// <typeparam name="T"></typeparam>
    /// <param name="entity">Jednostka.</param>
    /// <returns></returns>
    public T AddEntity<T>(T entity)
    {
        try
        {
            uow.BeginTransaction();
            uow.Add(entity);
            uow.CommitTransaction();
            return entity;
        }
        catch (Exception ex)
        {
            uow.RollbackTransaction();
            throw new Exception(string.Format(
                "An error occurred during the Add Entity method.", ex));
        }
    }
}

```

Warto zwrócić uwagę, że kod oznacza prywatne pole *IUnitOfWork* atrybutem *[Import]*. MEF skorzysta z tego atrybutu do realizacji odpowiadającego konkretnego elementu *UnitOfWork* w trakcie działania programu, gdy będziemy tworzyć nowe wystąpienie klasy *Repository*. Wszystkie pozostałe metody repozytorium powinny być implementowane podobnie: otwieramy transakcję, wywołujemy odpowiednią metodę *UnitOfWork*, a następnie wywołujemy na koniec *CommitTransaction* lub *RollbackTransaction*.

To kończy implementację *UnitOfWork* dla Entity Framework. Mamy w pełni elastyczną warstwę DAL dla modelu domenowego, która nie jest świadoma modelu mapowania zaprojektowanego przy użyciu projektanta .edmx w Entity Framework.

Mapowanie domeny przy użyciu NHibernate

Proces mapowania domeny przy użyciu NHibernate jest prostszy i powinien zająć mniej czasu – ale trzeba pamiętać, że wszystko zależy od stosowanego podejścia DDD. W tym przypadku, jeśli korzystamy z podejścia tworzącego najpierw domenę, NHibernate pozwala nam wygenerować wszystko automatycznie od mapowania po ostateczny schemat bazy danych. Jeśli korzystamy z podejścia tworzącego najpierw bazę danych, to NHibernate będzie wymagać więcej wysiłku niż Entity Framework, żeby wygenerować pliki mapowań.

Zebranie narzędzi

Najpierw przejdźmy do www.nhforge.com i pobierzmy najnowszą dostępną wersję narzędzia NHibernate. W momencie pisania tej książki dostępne było dość stabilne wydanie beta 2 wersji 3. Po pobraniu wersji, z której chcemy skorzystać, przejdźmy do www.fluentNHibernate.com i pobierzmy najnowszą wersję FluentNHibernate, abyśmy mogli tworzyć mapowania korzystając z kilku wierszy kodu zamiast ręcznie pisać podatne na błędy pliki XML.

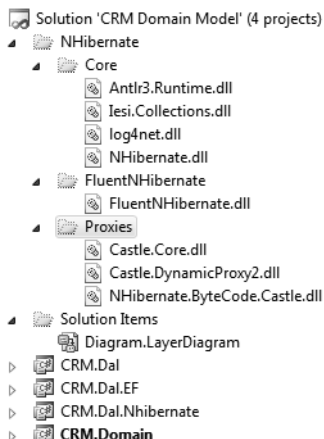
Utwórzmy nowe rozwiązanie w Visual Studio i nazwijmy je **CRM.DAL.NHibernate**. Dodajmy następujące trzy odwołania: *CRM.DAL* (abstrakcyjna warstwa DAL), *CRM.Domain* (model domenowy) i *System.ComponentModel.Composition* (do obsługi MEF).

Znajdźmy folder, gdzie zainstalowaliśmy NHibernate; powinniśmy zobaczyć zestaw podzespołów, które są *obowiązkowe* do uruchamiania tego narzędzia O/RM. Aby w pełni zainstalować NHibernate, powinniśmy mieć pobrane trzy różne pakiety: *NHibernate[wersja].GA*, główny silnik: *LINQtoNHibernate[wersja].GA*, dostarczający obsługę LINQ; oraz *FluentNHibernate*, jak to omówiliśmy na początku tego podrozdziału.

Powinniśmy dodać odwołania do wszystkich trzech pakietów w swojej konkretnej warstwie DAL dla NHibernate. Pokazane to zostało na rysunku 4-11.

Zanim zaczniemy, krótko objaśnię tę listę podzespołów. Folder o nazwie *Core* zawiera silnik NHibernate; *Log4Net.dll*, który jest otwartym dziennikiem przyłączonym do tego narzędzia O/RM; *Iesi.Collection.dll* jest kolekcją pośredników używaną przez narzędzie O/RM; a *Antlr3.Runtime.dll* jest narzędziem profilującym dla tego systemu O/RM. W przypadku *FluentNHibernate.dll* nazwa mówi sama za siebie. Folder *Proxies* jest jednym z dostępnych silników pośredniczących do tworzenia klas pośredniczących na żądanie. Możemy wybierać spośród narzędzi *Castle*, *LinFu* lub *Spring*; niestety narzędzie *Unity* nie jest dostępne dla tego systemu O/RM. Należy

wybrać żądany silnik pośredniczący. Dla celów demonstracyjnych będę korzystał tutaj z silnika Castle, gdyż jest najłatwiejszy do skonfigurowania.



RYSUNEK 4-11 Podzespoły wymagane przez NHibernate

Po dodaniu odwołań do wszystkich podzespołów możemy zacząć budować mapowanie NHibernate i fabrykę sesji. Zaczniemy od *UnitOfWork*. Chcemy zrobić to samo, co zrobiliśmy w przypadku warstwy DAL dla Entity Framework: utworzyć nową klasę *UnitOfWork*, która dziedziczy po *CRM.DAL.IUnitOfWork* i zaimplementować każdą z metod.

UnitOfWork i *ISession*

Klasa *UnitOfWork* dla NHibernate jest nieco inna, ponieważ nie ma obiektu *ObjectContext*. Zamiast tego mamy obiekt *ISession* generowany przez fabrykę sesji. Oto, jak to działa:

```
[Export(typeof(IUnitOfWork))]  
public class UnitOfWork : IUnitOfWork  
{  
    private ITransaction tx;  
  
    private ISession orm;  
  
    public UnitOfWork(ISession orm)  
    {  
        this.orm = orm;  
    }  
}
```

Interfejs *ISession* ma więcej możliwości niż obiekt *ObjectContext* zapewniany przez Entity Framework i jest też bardziej elastyczny. Niestety ze względu na jego charakterystykę musimy zapewnić cały potrzebny kod do wykonywania poprawnej transakcji

biznesowej. Sam interfejs *ISession* będzie utrzymywał transakcję przy życiu tak długo, jak chcemy.

```
public void BeginTransaction()
{
    if (tx != null)
    {
        tx = orm.BeginTransaction();
    }
}

public void CommitTransaction()
{
    if (tx == null)
    {
        throw new Exception("The current transaction has not been initialized.");
    }
    tx.Commit();
}

public void RollbackTransaction()
{
    if (tx == null)
    {
        throw new Exception("The current transaction has not been initialized.");
    }
    tx.Rollback();
}
```

Pozostałe metody są dość zbliżone do tego, co już widzieliśmy w przypadku Entity Framework. Główną różnicą jest to, że NHibernate nie musi przyłączać się do istniejącej jednostki, jeśli pozbedziemy się *ISession*. Mechanizm ignorowania sposobu zachowywania danych w NHibernate jest w stanie zrozumieć, czy jednostka jest nową jednostką, czy istniejącą jednostką.

NHibernate ma bardzo specyficzny sposób zarządzania sesją domenową reprezentowaną przez fabrykę sesji, statyczną klasę, która jest w stanie za jednym zamachem wygenerować wszystkie wymagane obiekty pośredniczące i połączenia z systemu O/RM. Nasza warstwa DAL będzie pobierać sesję z fabryki sesji, która zwróci nowy element UoW. Kod przykładowy wykorzystuje wtyczkę FluentNHibernate, a krótki fragment poniżej pokazuje, jak tworzyć automatyczne mapowanie (*pola domenowe = pola bazy danych*) tylko jednym wywołaniem:

```
[Export(typeof(ISessionFactory))]
public class SessionFactory : ISessionFactory
{
    private IUnitOfWork uow;

    public IUnitOfWork CurrentUoW {
        get
        {
            if (uow == null)
```

```

        {
            uow = GetUnitOfWork();
        }

        return uow;
    }
}

public SessionFactory()
{
}

/// <summary>
/// Pobiera jednostkę pracy.
/// </summary>
/// <returns></returns>
private IUnitOfWork GetUnitOfWork()
{
    var session = Fluently.Configure()
        .Database(
            MsSqlConfiguration.MsSql2008
            .ConnectionString(x => x.FromAppSetting("DatabaseConnection")))
        .Mappings(m => m.AutoMappings
            .Add(AutoMap.AssemblyOf<Person>)
            .Add(AutoMap.AssemblyOf<Customer>)
            .Add(AutoMap.AssemblyOf<Employee>)
            .BuildConfiguration());
    var uow = new UnitOfWork(session);
    return uow;
}

```

Ten przykład tworzy nową sesję SQL 2008, która podobnie do sesji Entity Framework pobierze łańcuch połączenia z pliku App.Config/Web.Config i utworzy automatyczne mapowanie dla każdej jednostki dodanej do konfiguracji. To tyle. Korzystając z podejścia budującego najpierw domenę nie musimy robić nic innego – interfejs *ISession* jest gotowy do użycia.

Repozytorium

Implementację repozytorium pozostawiłem na koniec, ponieważ nie powinna się ona różnić dla obu narzędzi O/RM. Jedyną różnicą jest to, że w Entity Framework będziemy odpytywali obiekt *ObjectContext*, natomiast w NHibernate będziemy odpytywali obiekt *ISession*.

Jedyną ważną sprawą, żebyśmy mogli korzystać z opóźnionego ładowania i tworzenia dynamicznego kodu SQL, jest zwracanie zawsze kolekcji *IQueryable<T>*.

Poniższy kod wykorzystuje element *UnitOfWork* w Entity Framework w celu pobrania listy klientów:

```

public IQueryable<T> GetList<T>() where T : class
{

```

```

        return ((ObjectContext)this.UoW.orm).CreateObjectSet<T>();
    }

```

Oto ten sam kod wykorzystujący repozytorium w NHibernate:

```

public IQueryable<T> GetList<T>() where T : class
{
    return ((ISession)this.UoW.orm).Linq<T>();
}

```

Oba te repozytoria pozwalają nam napisać kod podobny do następującego:

```

var customers = GetList<Customer>()
    .Where(c => c.FirstName == "John")
    .OrderBy(c => c.FirstName)

```

Implementacja ta będzie taka sama dla wszystkich implementacji repozytorium. W ten sposób będziemy korzystać z obiektów *IQueryable<T>* bez konieczności znajomości, z jakiej warstwy DAL faktycznie korzystamy w MVVM.

Podsumowanie

W tym rozdziale zobaczyliśmy wiele nowych elementów i pojęć, między innymi systemy O/RM. Jeśli ktoś jeszcze nie pracował z narzędziem O/RM, to powinien przejrzeć kod przykładowy z tej książki i poświęcić nieco czasu na zbadanie dokumentacji i samouczków opartych na wybranym systemie O/RM. Celem ćwiczenia w tym rozdziale jest pokazanie, jak pisać dynamiczny kod, który można wykorzystać wielokrotnie. Oczywiście w rzeczywistej aplikacji biznesowej MVVM prawdopodobnie nigdy nie będziemy musieli mapować swojego modelu domenowego do dwóch różnych systemów O/RM – ale chodzi o to, że jeśli będziemy stosować te techniki, to będziemy w stanie ponownie wykorzystywać fragmenty swojego kodu w kolejnych aplikacjach MVVM.

Zobaczyliśmy, że możemy nadal wykorzystywać „klasyczny styl” pisania niestandardowego kodu T-SQL – albo możemy poświęcać mniej czasu i skupiać się bardziej na logice biznesowej aplikacji delegując najtrudniejsze zadania do systemu O/RM. Narzędzie O/RM jest po prostu platformą aplikacyjną używaną jako pomoc w tłumaczeniu modelu domenowego na coś, co jest zrozumiałe dla bazy danych, a wszystko to bez utraty zasad POCO dbających o to, aby domena nie była świadoma mechanizmów zachowywania danych.

Istnieją inne systemy O/RM dla platformy .NET Framework, ale najpopularniejszymi (i darmowymi) są Entity Framework i NHibernate. Podczas gdy Entity Framework zaprojektowano bardziej dla podejścia budującego najpierw bazę danych, to NHibernate jest bardziej elastyczny w stosowaniu wraz z podejściem budującym najpierw domenę. Widzieliśmy jednak, że każdy system O/RM może być stosowany z każdym z tych dwóch podejść DDD.

Warstwa biznesowa

Po zakończeniu tego rozdziału będziemy w stanie:

- Utworzyć i wykonywać reguły biznesowe.
- Utworzyć prawidłową warstwę logiki biznesowej.
- Zastosować zdobytą wiedzę w aplikacji przykładowej.

Wprowadzenie

Jednym z najbardziej czasochłonnych zadań – i chyba najbardziej kosztownym, jeśli chodzi o utrzymanie – jest warstwa biznesowa naszej aplikacji (przy założeniu oczywiście, że aplikacja, nad którą pracujemy, taką posiada).

Zarówno w starszych, jak i w nowoczesnych aplikacjach, warstwa biznesowa zwykle składa się z zagnieżdżonego zbioru klas. Niektóre aplikacje przechowują logikę biznesową w warstwie domenowej; inne przechowują ją w bazie danych korzystając z procedur składowanych lub widoków. Najgorsze aplikacje przechowują logikę biznesową w chaotyczny sposób rozrzuconą po całym kodzie.

Niewiele jest mniej przyjemnych rzeczy niż próba utrzymywania aplikacji, w której logika biznesowa jest rozrzucona wszędzie. Nie tylko stracimy dużo czasu próbując ustalić, jak działa kod, ale każda dokonana zmiana może spowodować nieoczekiwane zachowania w innych częściach aplikacji.

W czystym, nowoczesnym projekcie warstwa biznesowa powinna być oddzielną warstwą aplikacji. Powinna mieć świadomość warstwy domenowej i prawdopodobnie również warstwy danych. Warstwa biznesowa jest logicznym sercem każdej aplikacji biznesowej i powinna być jedynym miejscem, w którym umieszczana jest logika biznesowa aplikacji. Powinna być też łatwa w utrzymaniu i dobrze udokumentowana.

W aplikacji Model View ViewModel (MVVM) warstwa biznesowa składa się z zestawu usług i reguł biznesowych definiujących procesy biznesowe, które program ma wykonywać. Warstwa ta jest wykorzystywana przez modele widoków, które powinny zawierać tylko logikę prezentacyjną. Ten schemat pozwala nam zachować luźne wiązanie pomiędzy logiką biznesową a logiką prezentacyjną.



UWAGA Podczas gdy termin „reguły biznesowe” jest często używany na oznaczenie dowolnej logiki niezwiązanej z interfejsem użytkownika, to w tej książce termin ten obejmuje tylko logikę, która implementuje procesy biznesowe i zapewnia spójność danych. Ten kod jest niezależny od części klienckiej aplikacji.

Zanim zbadamy, jak osiągnąć cel polegający na utrzymywaniu logiki biznesowej oddzielonej od widoków i jednostek domenowych w przykładowej aplikacji CRM, chcę wyjaśnić różnicę pomiędzy regułami sprawdzania poprawności a regułami biznesowymi.

Reguła biznesowa nie jest regułą sprawdzania poprawności

Reguła sprawdzania poprawności jest dowolnym kryterium, które opisuje, jak sprawdzać poprawność określonej wartości określonego obiektu. Przykłady obejmują ograniczenie długości pola w tabeli bazodanowej albo komunikaty „pole wymagane” wyświetlane w widoku. Reguły sprawdzania poprawności są zwykle stosowane wobec obiektu, którego poprawność trzeba zbadać, *zanim* jego wartość zostanie zapisana w magazynie danych lub *zanim* zostanie przetworzony przez inną transakcję, która wymaga poprawnej wartości.

Natomiast reguła biznesowa jest dowolną regułą, która działa w celu *zmiany wartości* obiektu w oparciu o zestaw reguł lub w oparciu o określone zachowanie. Reguły biznesowe często mają jedynie na celu informowanie użytkowników, czy dane działanie może, czy nie może być wykonane po sprawdzeniu określonych obiektów biorących udział w danej transakcji.



UWAGA Zauważyłem, że wiele osób nie widzi różnicy pomiędzy regułami biznesowymi a regułami sprawdzania poprawności. Ponieważ oba te typy reguł służą temu samemu panu, to możemy zbierać oba te typy reguł razem – powinniśmy być jednak świadomi, że są to dwa różne typy reguł. Pierwszy typ reguł zwanych regułami sprawdzania poprawności zapewnia spójność i poprawność danych, drugi typ reguł zwanych regułami biznesowymi obsługuje procesy i operacje biznesowe.

Zwykle definiujemy reguły sprawdzania poprawności w dwóch oddzielnych warstwach: modelu domenowego i interfejsu użytkownika. W pierwszym przypadku ustawiamy ograniczenia, żeby zapewnić poprawność jednostki domenowej przed zapisaniem jej w magazynie danych lub przetworzeniem jej przez inny składnik. Możemy definiować te reguły ręcznie korzystając z kodu proceduralnego, korzystając z platform firm trzecich lub poprzez proste atrybuty dekoracyjne. Poniższy kod pokazuje, jak możemy sprawdzić poprawność obiektu przy użyciu przestrzeni nazw *System.ComponentModel* dostępnej w Microsoft .NET Framework:

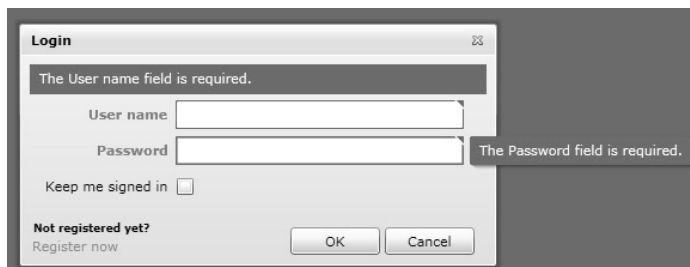

```

/// <summary>
/// Pobiera lub ustawia imię.
/// </summary>
/// <value>Imię.</value>
[Required(ErrorMessage = "The First Name can't be null or empty.")]
[StringLength(50, ErrorMessage = "The First Name can't be greater than 50 characters.")]
public string FirstName { get; set; }

```

Powyższy kod zdefiniował kilka prostych reguł sprawdzania poprawności dla jednostki domenowej *Person*, a w szczególności dla jej właściwości *FirstName*. Chcemy zapewnić, że wartość tej właściwości nie będzie pusta i że nie przekroczy 50 znaków długości. Każdy błąd sprawdzania poprawności jest związany z określonym komunikatem o błędzie. W rozdziale 6 „Warstwa interfejsu użytkownika w MVVM” zobaczymy, jak można łatwo wiązać te właściwości z modelem widoku.

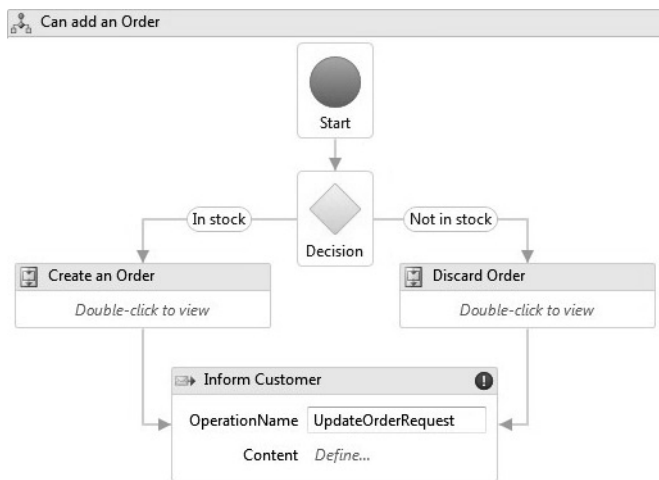
Drugie zastosowanie tej reguły związane jest z warstwą interfejsu użytkownika. Na przykład w prostej biznesowej aplikacji Silverlight możemy napotkać sprawdzanie poprawności podczas próby zalogowania się przy użyciu nieprawidłowych poświadczeń. Rysunek 5-1 pokazuje klasyczną regułę sprawdzania poprawności zastosowaną w widoku MVVM aplikacji Silverlight.



RYСУNEK 5-1 Widok Silverlight z komunikatem sprawdzania poprawności

Reguła biznesowa wykorzystuje inne podejście, ponieważ składa się z zestawu reguł, które nie są łatwo definiowane przy użyciu prostego podejścia wykorzystującego atrybuty. Zwykle musimy pisać instrukcje *if/else* albo *switch*, w zależności od typu zasad, które chcemy zweryfikować. Platforma .NET Framework zawiera darmowe i przydatne narzędzie pomagające w tym zadaniu: Windows Workflow Foundation wersja 4 (WF 4.0). Oczywiście problemem nie jest tutaj jedynie pisanie skomplikowanych instrukcji *if/else* lub *switch*. Problem wiąże się z logiką aplikacyjną, której używamy do zaimplementowania procesu biznesowego albo do zapewnienia, że złożony zestaw reguł zostanie zastosowany, czego nie możemy wyrazić w postaci pojedynczego atrybutu dla wartości.

Na przykład możemy rozważyć przepływ zadań pokazany na rysunku 5-1 jako zestaw reguł.



RYСУNEK 5-2 Niestandardowy przepływ zadań tworzący zamówienie.

Przepływ zadań na rysunku 5-2 wykonuje zestaw reguł biznesowych, aby sprawdzić, że przesłane zamówienie (*Order*) może zostać zrealizowane; na przykład, czy żądany produkt jest w magazynie i dostępny do wysłania. Jeśli sprawdzenie się powiedzie, to zamówienie zostanie zrealizowane i komunikat zostanie wysłany do kolejki. W przeciwnym razie zamówienie zostanie odrzucone, a inny komunikat zostanie wysłany do kolejki, aby poinformować klienta, że produktu nie ma w magazynie.

Możemy też osiągnąć ten proces wykorzystując następujący pseudokod w C#:

```

public class CustomRules
{
    [Import]
    private IRepository repository;
    public void CanAddAnOrder(Order order, Customer customer)
    {
        foreach (var orderLine in order.OrderLines)
        {
            var available =
                repository.GetEntity<Product>(orderLine.Product.PrimaryKey)
                    .AmountInStock;
            if (!available)
            {
                RemoveOrder(order, customer);
                break;
            }
            ConfirmOrder(order, customer);
        }
    }

    private void RemoveOrder(Order order, Customer customer) { }
    private void ConfirmOrder(Order order, Customer customer) { }
}

```

Ten drugi przykład demonstruje, że tłumaczenie zestawu reguł na język programowania jest procesem podatnym na potencjalne błędy. Co więcej ten zestaw reguł jest całkowicie niezrozumiały dla każdego, kto nie jest obeznany w kodzie, na przykład analityk lub menedżer projektu, który musi też wiedzieć, jak działa ten zestaw reguł. Ponadto proces utrzymywania aktualnej dokumentacji dla takich reguł może być czasochłonny.

Z kolei podczas korzystania z narzędzia graficznego takiego, jak WF 4.0 do implementacji zestawu reguł, możemy udostępniać reguły programistom zaangażowanym w proces tworzenia oprogramowania, a także nie-programistom. Innymi słowy reprezentacja wizualna sprawia, że nasz kod reguł staje się bardziej czytelny i łatwiejszy w utrzymaniu przez wszystkich.

W tym rozdziale zobaczymy inne narzędzia firm trzecich dostępne do stosowania tego podejścia do projektowania i dlatego to podejście jest znacznie lepsze niż niestandardowy kod C# rozproszony po warstwach aplikacji biznesowej. Po pierwsze, rozważmy możliwość testowania naszych reguł biznesowych; jeśli są one częścią określonej warstwy/składnika, to możemy testować je łatwo z wykorzystaniem zestawu danych testowych. Po drugie, weźmy pod uwagę dokumentację: wizualny przepływ zadań jest też czytelny i zrozumiały dla osób nietechnicznych, takich jak pracownicy operacyjni albo audytor, który może weryfikować logikę biznesową zastosowaną w aplikacji.

Reguły biznesowe w usłudze

Reguły biznesowe muszą być przechowywane w jakiejś warstwie i chyba najlepszym miejscem na to jest dodatkowy podzespół widoczny dla warstwy biznesowej. Nie jest to dodatkowa warstwa, ale rozszerzenie warstwy logiki biznesowej (BLL), które zawiera jedynie fizyczne przepływy zadań. Nie jest tak bardzo istotne, jakiej technologii użyjemy do utworzenia swoich reguł biznesowych (przepływy zadań, kod proceduralny lub pliki XML), ale ważne jest, żeby te reguły były trzymane oddzielnie od innego kodu w aplikacji biznesowej, żebyśmy mogli łatwo je utrzymywać i testować.

Przydatne jest też wykonywanie tych reguł przy pomocy kodu, który wykorzystuje ten sam format w całej aplikacji biznesowej, żebyśmy mogli łatwo rozpoznawać wywołanie reguły biznesowej i stosować je w spójny sposób.

Projektowanie na podstawie usługi jest wzorcem projektowym wprowadzonym przez Martina Fowlera, w którym sedno naszej transakcji biznesowej odbywa się w usłudze, która wie wszystko o modelu i warstwie danych. Jest to też jedyny obiekt odpowiadający za podejmowanie decyzji biznesowych.

Wykorzystując projektowanie według usługi w warstwie biznesowej delegujemy wykonywanie reguły lub zestawu reguł, sprawdzanie poprawności obiektów i poszczególne transakcje biznesowe do klasy usługi, która nie potrzebuje nic więcej poza obiektami biorącymi udział w danym procesie. Najlepszym sposobem, aby to uzyskać,

jest utworzenie zestawu klas usługowych opartych na transakcjach biznesowych, które będzie przeprowadzać nasza aplikacja. Im bardziej ziarnisty kod w tej części aplikacji, tym łatwiej będzie utrzymywać tę ważną warstwę.

Pseudokod dla takiej klasy usługowej wyglądałby podobnie do poniższego:

```
var svc = Container.Resolve<IService<Customer>>>();  
var order = svc.CreateProcess(ProcessEnum.CreateOrder, myPerson, myOrderLines);  
var result = svc.Verify(RulesEnum.AddOrder, myPerson, order);
```

Powyższy kod jest prostą, ogólną klasą, która może wykonywać przepływy zadań w oparciu o typ operacji, które chcemy przeprowadzić. Klasa ta jest elastyczna, łatwa do analizowania i utrzymywania. Możemy definiować operacje korzystając ze zbioru wartości wyliczeniowych, żeby zwiększyć czytelność kodu, a następnie odwoływać się do konkretnego przepływu zadań poprzez tę samą nazwę.

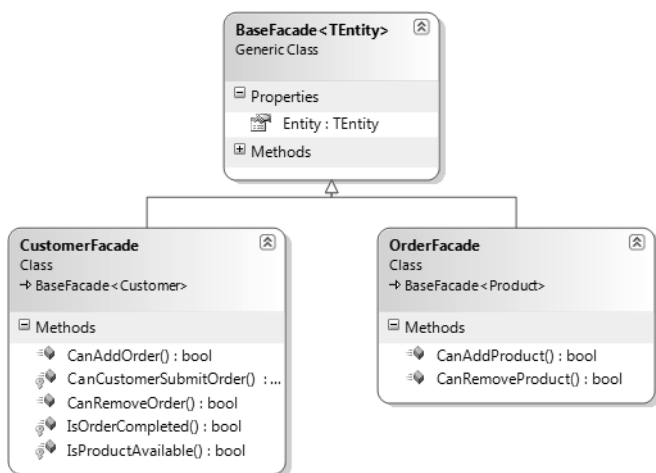
Wzorzec fasady

Innym interesującym sposobem byłoby wykorzystanie jednego lub kilku wzorców projektowych, które widzieliśmy w rozdziale 2 „Wzorce projektowe”, żeby nasza warstwa biznesowa była bardziej elastyczna. Na przykład wzorzec fasady dobrze pasowałby do bazowej usługi, która udostępnia proste metody poprzez wspólny interfejs fasadowy ukrywający prawdziwe interakcje pomiędzy systemami. W następującym przykładzie chcemy udostępnić metodę *AddOrder* w naszej fasadzie, ale nie chcemy wymagać od programistów wiedzy, co się dzieje w tle tego procesu.

Rysunek 5-3 ilustruje klasyczną warstwę usługową w postaci fasady. Każda fasada w tym przypadku wykorzystuje ogólną implementację do zidentyfikowania głównej jednostki uczestniczącej w danym procesie, a każda usługa udostępnia pewne metody biznesowe wykonujące zbiór transakcji, które nie są widoczne poza usługą fasadową. Na przykład metoda *CanAddOrder* (czy można dodać zamówienie) wykonuje wewnętrznie dodatkowe metody takie, jak:

- *IsProductAvailable* (czy produkt jest dostępny)
- *IsOrderCompleted* (czy zamówienie jest gotowe)
- *CanCustomerSubmitOrder* (czy klient może złożyć zamówienie)

Ponieważ te trzy dodatkowe metody są oznaczone jako prywatne, to nie są widoczne poza usługą fasadową. W ten sposób możemy oddzielać kod, ale też możemy sprawić, że korzystanie z warstwy usługowej jest mniej podatne na błędy, ponieważ wymusza na programistach wywoływanie tylko udostępnionych na zewnątrz metod, takich jak *CanAddOrder*, a nie zapewnia bezpośredniego dostępu do szczegółowych metod wykorzystywanych przez fasadę.



RYSUNEK 5-3 Usługa biznesowa wykorzystująca wzorec fasady

Reguły biznesowe w przepływie zadań przy użyciu WF 4.0

Gdy budujemy swoją warstwę biznesową, ważne jest, abyśmy umożliwili osobom biznesowym jej analizowanie i zrozumienie – tak jak w przypadku modelu domenowego. Korzystając z odpowiedniej kombinacji modelu domenowego i warstwy biznesowej osoba nie będąca programistą powinna być w stanie zrozumieć projekt naszej aplikacji i logikę biznesową, która za nią odpowiada.

W poprzednim podrozdziale dowiedzieliśmy się, że trudno jest osadzać niestandardową, ogólną logikę w procedurze C# w taki sposób, żeby była łatwo zrozumiała. Oczywiście możemy pisać zgrabne i czyste interfejsy, ale często to nie wystarcza zwłaszcza dlatego, że warstwa biznesowa często się rozrasta i zmienia w czasie życia aplikacji.

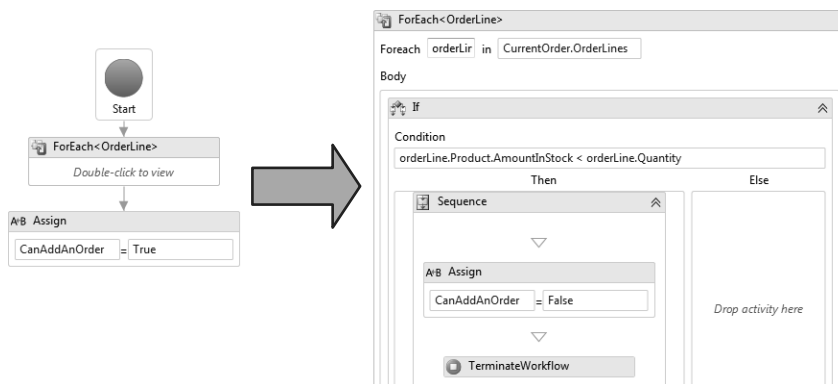
Technologia WF 4.0 będąca częścią .NET Framework 4 ma zaktualizowany silnik przepływów zadań zbudowany wokół kodu XAML – tak jak normalna aplikacja MVVM. Ta wersja całkowicie różni się od wersji poprzedniej; jest zarówno bardziej elastyczna, jak i pozwala nam budować niestandardowe przepływy zadań i zestawy reguł dla naszej warstwy biznesowej.

Podstawowym zadaniem WF 4.0 jest upraszczanie pisania proceduralnego przepływu zadań, który może podejmować decyzje i adaptować się w oparciu o wartości otrzymywane jako parametry wejściowe. Oczywiście te wartości mogą pochodzić albo z jednej, albo z wielu jednostek modelu domenowego lub też być prostymi wartościami skalarnymi.

Następny przykład wykorzystuje nowy typ projektu „Workflow Activity Library” dostępny w .NET Framework 4 i Microsoft Visual Studio 2010. Dodałem dwa

odwołania do tego projektu: jedno do ogólnej warstwy danych, którą utworzyliśmy w rozdziale 4 „Warstwa dostępu do danych” i jedno do modelu domenowego, który utworzyliśmy w rozdziale 3 „Model domenowy”. Ten przepływ zadań ma za zadanie zapewnić, że użytkownik może dodawać i zatwierdzać zamówienie (*Order*). W tym przypadku będziemy mieli dwa parametry: aktualnego klienta (*Customer*) i zamówienie (*Order*) do przetworzenia.

Na początek ten przepływ zadań zawiera jedną instrukcję *if* sprawdzającą, czy produkty występujące w tym zamówieniu (*Order*) są dostępne. Każdy produkt ma właściwość *AmountInStock*, która informuje nas, ile elementów jest dostępnych w magazynie. Każdy wiersz zamówienia (*OrderLine*) zawierający produkt ma właściwość (*Quantity*) i właściwość (*Product*). Reguła opisuje następującą zasadę: *Jeśli żądana ilość jest większa niż dostępna w magazynie, to zamówienie nie może zostać przetworzone*. Rysunek 5-4 dokumentuje ten proces wykorzystując WF 4.0.



RYСУNEK 5-4 Przepływ zadań dla transakcji biznesowej dodawania zamówienia

W tym przykładzie przekazujemy dwa parametry wejściowe do przepływu zadań: *CurrentCustomer* i *CurrentOrder*. Otrzymujemy wyjściowy parametr typu *Boolean* o nazwie *CanAddOrder*. Wywołanie tego z ostatecznego kodu będzie wyglądać podobnie, jak pokazano poniżej:

```
var customer = MVVMView.CurrentCustomer;
var order = MVVMView.CurrentOrder;
var canAddOrder = OrderService.CanAddOrder(customer, order);
if (canAddOrder)
{
    customer.AddOrder(order);
    uow.Save(customer);
}
```

Oczywiście powinniśmy również rozpocząć i zamknąć transakcję – ten kod zakłada, że elementy *Order* i *Customer* zostały sprawdzone pod względem poprawności przed rozpoczęciem tego kodu.

Różne sposoby wykonywania przepływu zadań

WF 4.0 oferuje dwa sposoby uruchamiania przepływu zadań. Sam przepływ zadań jest jedynie plikiem XAML skompilowanym wewnątrz projektu .dll (Activities Library). .NET 4 może odczytywać te specjalne pliki XAML, deserializować je na żądanie, a następnie przekazywać je do silnika wykonawczego WF lub przetwarzać plik XAML osadzony w bibliotece klas korzystając z obiektu Workflow Application.

WorkflowInvoker

Pierwszą (i najprostszą) metodą uruchomienia przepływu zadań (która została odziedziczona po wersji 3 silnika przepływów zadań) jest *WorkflowInvoker*. Ta klasa wymaga, aby przepływ zadań, który chcemy uruchomić, był już dostępny, więc albo musimy znać prawdziwą nazwę pliku z kodem przepływu zadań, albo mieć kod XAML. Działa to w następujący sposób:

```
// przede wszystkim tworzymy przepływ zadań w pamięci
Activity wf;
using (Stream xaml = File.OpenRead("CanAddOrder.xaml"))
{
    wf = ActivityXamlServices.Load(xaml);
}
```

Następnie wywołujemy statyczny silnik przepływów zadań (Workflow Engine) i przekazujemy mu wszelkie wejściowe i wyjściowe parametry w kolekcji *IDictionary<string, object>*, gdzie kluczem kolekcji jest nazwa parametru, a obiektem jest jego bieżąca wartość.

```
var params = new Dictionary<string, object>
{
    { "CurrentCustomer", myCustomer },
    { "CurrentOrder", myOrder },
}
// uruchom przepływ zadań
var output = WorkflowInvoker.Invoke(wf, params);
// dostęp do wyników
Console.WriteLine("Can Execute? {0}", output["CanExecute"]);
```

Klasa *WorkflowInvoker* zwraca inną kolekcję typu *<string, object>*, która zawiera wszystkie dostępne parametry wyjściowe.

Zalety i wady WorkflowInvoker

Klasa *WorkflowInvoker* jest dość nieskomplikowana i łatwa w użyciu. W istocie jest zbyt prosta; nie daje nam dużej kontroli nad przepływem zadań. Na przykład nie możemy śledzić stanu przepływu zadań i nie możemy monitorować jego wykonywania przy użyciu zdarzeń. Dlatego powinniśmy korzystać z klasy *WorkflowInvoker* jedynie w przypadku prostych przepływów zadań takich, jak sprawdzanie warunku

CanExecute w menu kontekstowym lub przycisku polecenia. Możemy też korzystać z niej do testowania swoich przepływów zadań przed przeniesieniem ich do środowiska produkcyjnego, ale trzeba mieć na uwadze, że klasa *WorkflowInvoker* nie jest zaprojektowana dla bardziej skomplikowanego środowiska.

WorkflowApplication i WCF

Jeśli planujemy zbudowanie warstwy BLL swojej aplikacji MVVM przy użyciu WF 4.0, to powinniśmy się skoncentrować na bardziej złożonym silniku obsługującym przepływy zadań o nazwie *WorkflowApplication*. Ten składnik wykorzystuje te same kolekcje parametrów wejściowych i wyjściowych, ale mamy też dostęp do poszczególnych zdarzeń i wywołań asynchronicznych, dzięki którym możemy budować bardziej złożone i skomplikowane silniki reguł.

Aby uruchomić przepływ zadań w ten sposób, najpierw musimy pobrać swój bieżący przepływ zadań. W tym celu nie musimy znać ścieżki do pliku XAML, a jedynie nazwę klasy dostępnej w DLL, jak pokazano na poniższym przykładzie:

```
WorkflowApplication wf = new WorkflowApplication(new CanAddOrder());
// parametry
var params = new Dictionary<string, object>
{
    { "CurrentCustomer", myCustomer },
    { "CurrentOrder", myOrder },
}
```

Następnie tworzymy nowe wystąpienie niestatycznej klasy *WorkflowApplication* i subskrybujemy wszystkie dostępne zdarzenia tak, abyśmy mogli mieć pełną kontrolę nad wykonywaniem zestawu reguł:

```
wf.Completed = delegate(WorkflowApplicationCompletedEventArgs e)
{
    // Obsługa zakończenia wykonywania przepływu zadań
};
wf.Aborted = delegate(WorkflowApplicationAbortedEventArgs e)
{
    // Obsługa przerwania wykonywania przepływu zadań
};
wf.OnUnhandledException =
    delegate(WorkflowApplicationUnhandledExceptionEventArgs e)
{
    // obsługa błędu
    return UnhandledExceptionAction.Terminate;
};
wf.Run();
```

Korzystając z tego podejścia możemy wykonywać zestaw reguł na przykład na poziomie pośrednim, który może być szybszy i sprawniejszy niż komputer kliencki albo możemy po prostu monitorować wykonywanie przepływu zadań i podejmować decyzje biznesowe bez wyrzucania wyjątków w interfejsie użytkownika.

AppFabric i wykonywanie usług

Microsoft AppFabric jest zbiorem zintegrowanych technologii ułatwiających budowanie, skalowanie i zarządzanie aplikacjami WWW, które działają pod kontrolą IIS i Windows Server. Windows Server AppFabric można pobrać pod adresem <http://msdn.microsoft.com/en-us/windowsserver/ee695849> i zainstalować poprzez prosty w użyciu instalator.

Jedną z funkcji AppFabric jest zapewnienie możliwości utrzymywania i wykonywania przepływów zadań Windows Workflow poprzez zestaw usług WCF Services tak, że warstwa BLL naszej aplikacji może być przechowywana na osobnym serwerze aplikacyjnym. Ponieważ technologia AppFabric jest również oparta na platformie .NET Framework 4, to automatycznie zapewnia zestaw funkcjonalności, które mogą być przydatne przy budowaniu serwera aplikacyjnego, który musi utrzymywać jedną lub więcej warstw BLL aplikacji.

AppFabric jest złożonym produktem, który chyba zasługuje na całą książkę, ale nam chodzi tutaj o to, że AppFabric jest właściwą technologią do zbudowania w miarę skomplikowanej warstwy BLL ze skalowalnym i łatwym do utrzymania serwerem aplikacji. Na poparcie tej tezy i dla lepszego zrozumienia tego produktu przedstawiam poniżej listę funkcji, które udostępnia serwer Windows AppFabric:

- Wdrażanie i zarządzanie usługami WCF i WF utrzymywanymi przy użyciu WAS
- Konfigurowanie i zachowywanie przepływów zadań, ich stanów i wyników ich wykonania
- Dedykowany, możliwy do przeszukiwania magazyn do zarządzania
- Pełna integracja z Windows PowerShell
- Możliwość dostosowywania monitorowania utrzymywanych usług

Zalety i wady *WorkflowApplication*

Jeśli planujemy wykorzystanie silnika *WorkflowApplication*, to względy za tym przemawiające są zasadniczo przeciwne do tych za użyciem *WorkflowInvoker*. Klasa *WorkflowApplication* wymaga więcej wysiłku, ale pozwala nam na bardziej stabilne i dające więcej możliwości uruchamianie przepływów zadań. Jednocześnie zapewnia więcej kontroli i opcji, które mogą być nam potrzebne, jeśli planujemy wykorzystanie WF jako silnika zestawów reguł.

WF oferuje też wiele dodatkowych funkcji takich, jak utrzymywanie przepływu zadań bezpośrednio w aplikacji WPF albo zachowywanie stanu przepływu zadań w SQL, co pozwala nam wstrzymywać i wznowiać wykonywanie przepływu zadań zgodnie z wymaganiami.

Opis tych i innych funkcji, jak również inne informacje są dostępne w dokumentacji WF pod adresem <http://msdn.microsoft.com/en-us/netframework/aa663328.aspx>.

Zestawy narzędziowe firm trzecich

W tym przypadku warstwa biznesowa będzie bardzo skomplikowana i prawdopodobnie nie będziemy mieli czasu, aby nauczyć się i opanować skomplikowaną technologię taką, jak WF. Trzeba też mieć na uwadze, że WF jest technologią otwartą, więc żeby dostosować ją do wymagań klienta, będziemy musieli poświęcić jej nieco czasu.

Zauważyłem też, że zwykle w dużym środowisku warstwa biznesowa jest pozostawiana analitykom, którzy wiedzą, jak pisać reguły biznesowe, ale niestety w większości przypadków nie wiedzą, jak tłumaczyć te reguły na coś użytecznego dla aplikacji MVVM. W takich przypadkach powinniśmy brać pod uwagę wykorzystanie narzędzia firmy trzeciej, które złagodzi związany z tym wysiłek; będzie tylko trzeba podłączyć tę technologię do naszej warstwy BLL. Istnieją odpowiednie narzędzia firm trzecich, które mogą zapewnić analitykom łatwego w użyciu projektanta, a jednocześnie dają programistom rozbudowany silnik reguł, który można podłączyć do dowolnej aplikacji biznesowej.

Technologie służące do sprawdzania poprawności danych

Microsoft Enterprise Library obecnie w wersji 5.0 prawdopodobnie spełni wszystkie nasze wymagania związane z dodawaniem sprawdzania poprawności danych do naszych jednostek domenowych. Enterprise Library można uzyskać pod adresem <http://msdn.microsoft.com/en-us/library/fff632023.aspx>. Enterprise Library zawiera Validation Application Block (VAB); przydatną platformę, która zapewnia domyślny zestaw reguł sprawdzania poprawności oraz rozbudowany i elastyczny silnik reguł.



UWAGA Trzeba pamiętać, że sprawdzanie poprawności danych powinno być wymagane nie tylko w modelu domenowym, ale również w modelu widoku interfejsu użytkownika naszej aplikacji MVVM i w dowolnym miejscu, gdzie sprawdzanie poprawności danych jest wymagane przez projekt.

VAB pozwala nam dekorować klasy regułami sprawdzania poprawności zapewnianymi przez Enterprise Library lub zapewnianymi przez nas niestandardowymi regułami. Później możemy sprawdzić poprawność obiektu i zebrać wszelkie błędy sprawdzania poprawności. Poniższy kod ilustruje, jak przeprowadzać podstawowe sprawdzanie poprawności przy użyciu biblioteki VAB:

```
using Microsoft.Practices.EnterpriseLibrary.Validation;
using Microsoft.Practices.EnterpriseLibrary.Validation.Validators;
public class Customer
{
    [StringLengthValidator(0, 20)]
    public string CustomerName;

    public Customer(string customerName)
    {
        this.CustomerName = customerName;
    }
}
```

```

    }
}
public class MyExample
{
    private ValidatorFactory factory;
    public MyExample(ValidatorFactory valFactory)
    {
        factory = valFactory;
    }
    public void MyMethod()
    {
        Customer myCustomer = new Customer("A name that is too long");
        Validator<Customer> customerValidator
            = factory.CreateValidator<Customer>();
        // Sprawdzanie poprawności wystąpienia w celu uzyskania kolekcji błędów sprawdzania
        poprawności.
        ValidationResult r = customerValidator.Validate(myCustomer);
        if (!r.IsValid)
        {
            throw new InvalidOperationException("Validation error found.");
        }
    }
}

```

VAB zawiera domyślny zestaw atrybutów sprawdzania poprawności, do którego należą:

- Sprawdzanie zawierania określonych znaków
- Sprawdzanie zakresu daty/godziny
- Sprawdzanie z domeną
- Sprawdzanie konwersji wartości wyliczeniowych
- Sprawdzanie, czy wartość różna od null
- Sprawdzanie kolekcji obiektów
- Sprawdzanie obiektu
- Sprawdzanie alternatywy kilku sprawdzeń
- Sprawdzanie porównujące właściwość
- Sprawdzanie zakresu
- Sprawdzanie wyrażeń regularnych
- Sprawdzanie względnej daty/godziny
- Sprawdzanie długości łańcucha znaków
- Sprawdzanie konwersji typu
- Sprawdzanie elementu

Mogą nas też zainteresować inne platformy sprawdzania poprawności oferowane przez firmy trzecie; większość z nich jest godna zaufania i ma otwarty kod źródłowy, co oznacza, że nie trzeba kupować licencji, aby wykorzystywać je w swoich aplikacjach

MVVM. Dodatkowymi platformami sprawdzania poprawności przeznaczonymi dla .NET, które moglibyśmy zbadać, są:

- **EVIL** (<http://evil.codeplex.com>) Otwarty projekt, który działa bardzo podobnie do biblioteki VAB wykorzystując dekoracje i zestawy reguł.
- **Active Record** (<http://www.castleproject.org/activerecord/index.html>) Otwarta wtyczka dla NHibernate, która transformuje naszą domenę w domenę Active Record.
- **Conditions** (<http://conditions.codeplex.com>) Inna otwarta platforma, która wykorzystuje płynny interfejs (patrz rozdział 2) zamiast atrybutów.

Silnik reguł i silnik reguł biznesowych

Po przejściu do pojęcia reguł biznesowych dyskusja staje się bardziej skomplikowana. Silnik reguł biznesowych powinien zwykle być w stanie wykonywać niestandardowe reguły, zapewniać płynną składnię, która jest zrozumiała przez użytkowników nietechnicznych, i zapewniać łatwe do czytania i modyfikowania narzędzie autorskie.

Oczywiście, jeśli szukamy wszystkich tych wymagań gotowych do użycia w jednym narzędziu, to prawdopodobnie będziemy musieli sprawdzić i zakupić silnik reguł biznesowych pochodzący od firmy trzeciej – co będzie kosztować.

Dwa narzędzia, które tutaj zobaczymy, są najpopularniejsze w środowisku .NET. To niekoniecznie oznacza, że są one najlepszymi albo najbardziej elastycznymi narzędziami, jakie są dostępne. Ponadto ceny tych narzędzi zaczynają się mniej więcej od 100 000 USD.

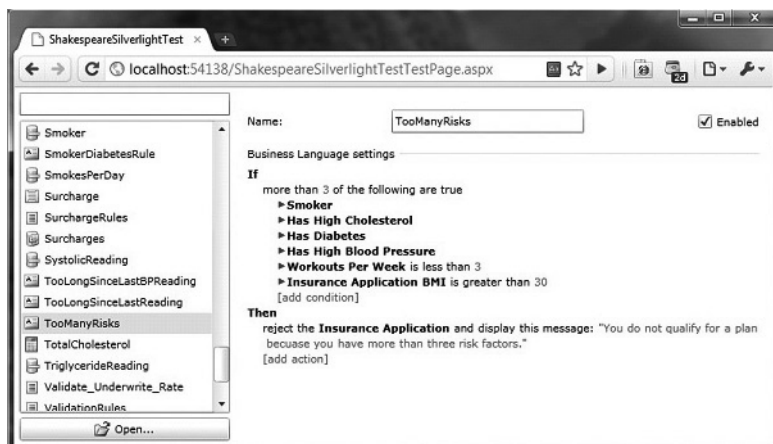
Istnieją różne narzędzia dostępne w sieci; to, które pokażę tutaj i następne, są tylko niektórymi z nich i w żadnym wypadku nie oznacza to, że należy zastosować to właśnie narzędzie jako swój silnik reguł biznesowych.

InRule for .NET

InRule jest bardzo elastycznym i łatwym w użyciu silnikiem reguł biznesowych, który można łatwo przyłączyć do dowolnej aplikacji .NET. Zapewnia kontrolki użytkownika do edycji reguł bezpośrednio w naszej aplikacji i ma jasną, łatwą do zrozumienia infrastrukturę.

InRule nie jest być może najbardziej skalowalnym rozwiązaniem, ale cena nie jest tak wysoka, jak w przypadku wielu innych silników reguł biznesowych. Wersję demonstracyjną InRule można pobrać pod adresem <http://www.inrule.com>, a następnie skorzystać z dostępnych samouczków, aby zobaczyć, jak korzystać z tego narzędzia.

Rysunek 5-5 pokazuje niestandardową aplikację Silverlight wykorzystującą InRule do edycji reguł biznesowych.



RYSUNEK 5-5 Narzędzie autorskie InRule w aplikacji Silverlight

Względy związane z warstwą biznesową

W tym rozdziale widzieliśmy, jak skomplikowana może być warstwa biznesowa i dlaczego jest wymagana w aplikacjach MVVM. Oczywiście obecność warstwy biznesowej w naszej aplikacji może wprowadzać pewne problemy, które trzeba wziąć pod uwagę.

Kiedy musimy tworzyć warstwę biznesową?

Dobłą praktyką jest zawsze próba zrozumienia, czy dana funkcja jest naprawdę potrzebna w naszej aplikacji MVVM tak, żeby można było uniknąć zbytniego skomplikowania aplikacji. Warstwa biznesowa może być mniej lub bardziej skomplikowana i powinniśmy zawsze mieć na uwadze poniższe uwagi, zanim rozpoczniemy tworzenie warstwy BLL w swojej aplikacji:

- Jeśli musimy wykonywać znaczną liczbę operacji i/lub reguł biznesowych w celu zachowania nowych danych w magazynie danych naszej aplikacji, to prawdopodobnie dobrze byłoby utworzyć warstwę BLL w aplikacji. Z drugiej strony, jeśli jedynie musimy zapisywać i pobierać dane z bazy danych oraz wyświetlać je w interfejsie użytkownika, to zaprojektowanie i wykorzystanie warstwy BLL będzie prawdopodobnie niewarte zachodu.
- Jeśli musimy sprawdzać poprawność swoich obiektów przed zapisaniem ich w magazynie danych albo przed przejściem do następnego kroku w przebiegu aplikacji, to warstwa BLL jest dobrym sposobem na odizolowanie procesu sprawdzania poprawności danych od reszty aplikacji. Jeśli jednak pracujemy z prostymi danymi, które nie wymagają wiele sprawdzania poprawności, to nie potrzebujemy warstwy BLL.

- Jeśli logika naszej aplikacji jest dynamiczna, skomplikowana i zmienna (będzie ulegać zmianom podczas procesu rozwijania oprogramowania) oraz musi być udokumentowana do celów kontrolnych, to *musimy* mieć warstwę BLL – i prawdopodobnie trzeba będzie rozważyć zakup silnika reguł biznesowych. Dzięki temu będziemy też w stanie utrzymywać warstwę BLL oddzieloną od innych składników aplikacji i będziemy mogli ją rozwijać niezależnie.

Podsumowując, jeśli nasza aplikacja wymaga znaczącej ilości logiki biznesowej i/lub sprawdzania poprawności danych, to powinniśmy rozważyć utworzenie warstwy BLL, żeby funkcjonalność ta była oddzielona od reszty kodu naszej aplikacji. Uprości to zarówno utrzymywanie aplikacji, jak i pomoże w jej dokumentacji.

Złe nawyki związane z warstwą BLL

Kilka ostatnich słów w związku z warstwą BLL dotyczy złych nawyków, z którymi się spotkałem, a których należy zawsze unikać.

Po pierwsze, warstwa BLL nie jest jednostką pracy (UoW) w naszej aplikacji, nie jest też repozytorium dla naszej warstwy dostępu do danych. Warstwa BLL ma atomowe metody, które mogą wykonywać zestaw transakcji biznesowych, na przykład:

```
var result = BLL.CanAddOrderToCustomer(myCustomer, myOrder);
```

Jest bardzo prawdopodobne, że metoda *CanAddOrderToCustomer* implementuje zestaw operacji, w których biorą udział UoW, repozytorium i model domenowy, jak w następującym przykładzie:

```
public void CanAddOrderToCustomer(myCustomer, myOrder)
{
    UnitOfWork.StartTransaction();
    var available = repository.Get<Order>(myOrder).AmountInStock;
    if (available)
    {
        myCustomer.Orders.Create(myOrder);

        repository.Update<Customer>(myCustomer);
        ...
    }
}
```

To nie oznacza, że warstwa BLL powinna mieć metodę taką, jak *GetCustomer(int id)*, ponieważ ta metoda powinna być udostępniana przez repozytorium, a nie implementowana w warstwie BLL. Bardzo prawdopodobne, że w aplikacji będzie inne miejsce, które nie będzie musiało korzystać z funkcjonalności BLL, ale będzie musiało załadować dane klienta o podanym identyfikatorze.

Po drugie BLL jest warstwą, której używamy do mówienia „językiem biznesowym”, więc powinniśmy zawsze używać jasnych i związanych konwencji nazewniczych. Metoda, która sprawdza, czy zamówienie może zostać dodane, powinna nazywać się na

przykład *CanCustomerCreateOrder* lub *CanItemBePurchased* zamiast krótszej, ale mniej zrozumiałej nazwy *AddOrder* lub *PurchaseItem*. Trzeba pamiętać, że warstwa BLL zawiera logikę biznesową aplikacji, więc kiedyś będziemy musieli ją zmieniać lub aktualizować, ze względu na zmianę logiki biznesowej lub zmianę procesu.

Na koniec przypomnę o konieczności testowania, testowania i jeszcze raz testowania. Musimy testować każdą metodę warstwy BLL na rzeczywistych danych, zwłaszcza jeśli warstwa BLL wykonuje obliczenia i statystyki. Każda metoda musi być zbadana na zbiorze testów, które można uruchomić w przyszłości, gdy będziemy musieli aktualizować logikę biznesową aplikacji.

Kod przykładowy: Warstwa usługi biznesowej

Teraz, gdy już mamy gotowy model domenowy i wiemy, jak zachowywać i pobierać model domenowy z bazy danych, potrzebny jest nam inteligentny sposób na wykonywanie reguł logiki biznesowej w ramach modelu domenowego i sprawdzanie poprawności jednostek domenowych przy użyciu określonego zestawu reguł sprawdzania poprawności.

W procesie sprawdzania poprawności przykładowa aplikacja CRM będzie wykorzystywać Enterprise Library 5.0; w szczególności przykład ten wykorzystuje blok VAB i typy ogólne C# do zbudowania ogólnego mechanizmu sprawdzania poprawności.

Do obsługi reguł biznesowych aplikacja wykorzystuje Windows WF 4.0. Zobaczmy też, jak utworzyć prosty silnik *FluentEngine*, który będzie w stanie uruchamiać dowolny przepływ zadań.

Sprawdzanie poprawności danych przy pomocy Enterprise Library 5.0

Pierwszym krokiem jest pobranie najnowszej wersji biblioteki Enterprise Library, która jest dostępna pod adresem <http://entlib.codeplex.com>. Następnie trzeba uruchomić polecenie *Build* dostępne w programie instalacyjnym Enterprise Library 5.0. W rezultacie powinniśmy otrzymać dwa foldery (w zależności od wybranych opcji instalacyjnych): jeden zawierający kod źródłowy Enterprise Library i jeden ze skompilowaną wersją gotową do wdrożenia. Biblioteka DLL, która nam będzie potrzebna, nosi nazwę *Microsoft.Practices.EnterpriseLibrary.Validation.dll*.

Dodajmy odwołanie do tej biblioteki DLL w warstwie CRM.Domain tak, abyśmy mogli dodawać reguły sprawdzania poprawności danych dla każdej jednostki domenowej. Poniższy przykład kodu pokazuje jednostkę *Person* z zastosowanymi kilkoma podstawowymi regułami sprawdzania poprawności danych:

```
/// <summary>
/// Pobiera lub ustawia imię.
/// </summary>
```

```

    /// <value>Imię.</value>
    [NotNullValidator(ErrorMessage = "The First Name can't be null or empty.")]
    [StringLengthValidator(50, ErrorMessage =
        "The First Name lenght can't be greater than 50 characters.")]
    public string FirstName { get; set; }

    /// <summary>
    /// Pobiera lub ustawia nazwisko.
    /// </summary>
    /// <value>Nazwisko.</value>
    [NotNullValidator(ErrorMessage = "The Last Name can't be null or empty.")]
    [StringLengthValidator(50, ErrorMessage =
        "The Last Name lenght can't be greater than 50 characters.")]
    public string LastName { get; set; }

    /// <summary>
    /// Pobiera lub ustawia datę urodzenia.
    /// </summary>
    /// <value>Data urodzenia.</value>
    [NotNullValidator(ErrorMessage = "The Birth Date can't be null or empty.")]
    [RelativeDateTimeValidator(18, DateTimeUnit.Year, 100, DateTimeUnit.Year,
        ErrorMessage = "The Birth Date can't be lower than 18 years.")]
    public DateTime BirthDate { get; set; }

```

Powyższy kod stara się odzwierciedlać ograniczenia schematu bazy danych w jednostkach domenowych tak, aby aplikacja mogła sprawdzać poprawność każdej jednostki przed zapisaniem lub pobraniem jej z bazy danych. Następnie możemy zastosować ten krok w jednostce UoW tak, żeby każda jednostka przekazywana do niej była automatycznie sprawdzana przed zatwierdzeniem transakcji.

Teraz musimy sprawdzić poprawność tej jednostki, a jeśli nie będzie prawidłowa, to powinniśmy zwrócić wynik typu *Boolean* w połączeniu z kolekcją błędów wygenerowanych przez proces sprawdzania poprawności. Każda jednostka, która dziedziczy po bazowym obiekcie domenowym, może być sprawdzana, więc nie ma lepszego miejsca niż obiekt domenowy na wprowadzenie tego procesu sprawdzania poprawności. Z założenia blok VAB udostępnia kolekcję o nazwie *ValidationResults*, która zawiera wyniki procesu sprawdzania poprawności i właściwość typu *Boolean* o nazwie *IsValid*. Możemy pobrać kolekcję wyników sprawdzania poprawności korzystając z klasy *Validator* udostępnianej przez blok aplikacji.

Najpierw otworzymy klasę obiektu domenowego i dodajmy właściwość tylko do odczytu, która udostępnia wyniki sprawdzania poprawności:

```

    /// <summary>
    /// Pobiera błędy sprawdzania poprawności.
    /// </summary>
    /// <value>Błędy.</value>
    public ValidationResults Errors { get; private set; }

```

Teraz musimy udostępnić właściwość *IsValid*, która uruchomi proces sprawdzania poprawności w tle. Zanim to zrobimy, chcę pokazać, jak można zastosować wzorzec

fasady sprawdzania poprawności w bloku VAB. Gdy chcemy sprawdzić poprawność nowego obiektu, możemy po prostu skorzystać z fasady fabryki sprawdzania poprawności udostępnionej w bibliotece. Niestety składnia tej fasady jest następująca:

```
// opcja wykorzystująca typy ogólne
var validator = ValidationFactory.CreateValidator<T>();
// druga opcja bez typów ogólnych
var validator = ValidationFactory.CreateValidator(Type);
```

To oznacza, że nie możemy udostępnić metody z klasy bazowej bez udostępnienia jej sygnatury ogólnej; w przeciwnym razie, gdy wywołamy tę metodę z odziedziczonej klasy, to podczas sprawdzania poprawności skontrolowane zostaną tylko właściwości klasy bazowej. Sprytnym rozwiązaniem mogłoby być:

```
/// <summary>
/// Pobiera lub ustawia wartość wskazującą, czy to wystąpienie jest prawidłowe.
/// </summary>
/// <value><c>true</c>, jeśli to wystąpienie jest prawidłowe;
/// w przeciwnym razie <c>false</c>.</value>
public virtual bool IsValid { get; private set; }

/// <summary>
/// Sprawdza poprawność tego wystąpienia.
/// </summary>
/// <typeparam name="T"></typeparam>
/// <returns></returns>
protected bool Validate<T>()
{
    Errors = ValidationFactory.CreateValidator<T>().Validate(this);
    return Errors.IsValid;
}
```

Teraz możemy zmienić proces sprawdzania poprawności, jeśli chcemy (należy zwrócić uwagę, że powyższa zmiana oznaczyła właściwość jako *virtual*, a nie *abstract*). Gdy musimy zaimplementować proces sprawdzania poprawności, jak w przypadku klasy *Person*, którą wcześniej udekorowaliśmy atrybutami Enterprise Library, możemy po prostu zmodyfikować właściwość *IsValid* w ten sposób:

```
/// <summary>
/// Pobiera lub ustawia wartość wskazującą, czy to wystąpienie jest prawidłowe.
/// </summary>
/// <value><c>true</c>, jeśli to wystąpienie jest prawidłowe;
/// w przeciwnym razie <c>false</c>.</value>
public override bool IsValid
{
    get
    {
        return base.Validate<Person>();
    }
}
```

W tym momencie mamy prosty mechanizm sprawdzania poprawności, który możemy analogicznie wykorzystywać w całym modelu domenowym. Korzystając z tego samego wzorca fasady możemy sprawdzać w ten sam sposób poprawność modelu widoku.

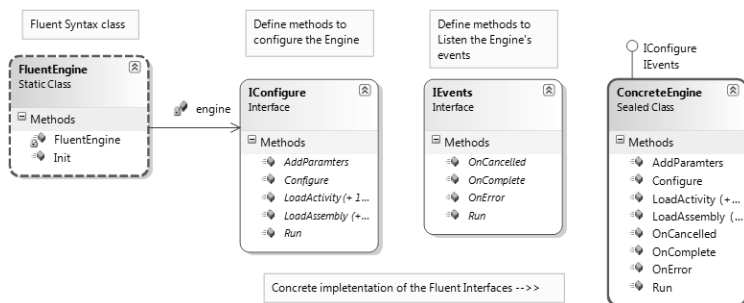
Więcej dogłębnych informacji dotyczących Enterprise Library można znaleźć w kompletnej dokumentacji w formacie PDF, którą można pobrać z witryny CodePlex pod adresem <http://entlib.codeplex.com/releases/view/46741>. Dokumentacja ta zawiera wiele przykładów i samouczków.

Ogólny silnik przepływów zadań

Wcześniej ten rozdział pokazał, jak technologia Workflow Foundation może być idealnym rozwiązaniem do zbudowania wewnętrznego silnika reguł biznesowych. Podczas gdy składnia ładowania i uruchamiania przepływu zadań nie jest idealna, to korzystając z wzorca płynnego języka, który był przedstawiony w rozdziale 2, możemy utworzyć płynny silnik, który zarówno będzie łatwy w użyciu, jak i będzie w stanie przetworzyć dowolny przepływ zadań.

Musimy pamiętać, że mamy dwie opcje uruchamiania przepływu zadań: prosty, statyczny *WorkflowInvoker* lub bardziej skomplikowany *WorkflowApplication*. Korzystając z tej drugiej opcji możemy monitorować stan przepływu zadań i dodawać niestandardowe zachowania (takie jak zapis do określonego dziennika) lub dołączać zdarzenia.

Zacznijmy od dodania infrastruktury do stworzenia płynnej składni. Schemat na rysunku 5-6 pokazuje diagram UML dla płynnego silnika przepływów zadań. Korzystając z tego silnika możemy załadować podzespół, który zawiera zestaw przepływów zadań, załadować określony przepływ zadań, tworzyć obiekty nasłuchujące zdarzeń WF przy pomocy składni lambda i oczywiście uruchamiać przepływ zadań.



RYSUNEK 5-6 Schemat UML dla płynnego silnika



UWAGA Aby przypomnieć sobie, jak budować płynny interfejs, można wrócić do podrozdziału „Języki DSL: pisanie płynnego kodu” w rozdziale 2.

Tutaj utworzyłem dwa interfejsy tak, że płynny interfejs będzie miał dwa główne kroki. Pierwszy krok ładuje i inicjuje przepływ zadań. W drugim kroku konfigurujemy

zdarzenia, których chcemy nasłuchiwać. Oba interfejsy mogą bezpośrednio uruchamiać przepływ zadań – ale, jeśli nie będziemy nasłuchiwać przynajmniej zdarzenia *onComplete*, to nie będziemy wiedzieć, kiedy zakończy się wykonywanie przepływu zadań.

Klasa statyczna jest używana jedynie do tworzenia zgrabniejszej, płynnej składni, żeby uniknąć brzydkiego użycia słowa kluczowego *new*.

Ostateczna składnia używana do uruchomienia przepływu zadań powinna wyglądać podobnie, jak poniżej:

```
//Inicjowanie klasy silnika
FluentEngine.Init()
    //Ładowanie podzespołu i przepływu zadań
    .LoadAssembly("MyWorkflowLibrary.dll")
    .LoadActivity("CanAddAnOrder.xaml")
    //przygotowanie kolekcji parametrów
    // powinna ona zawierać parametry wejściowe/wyjściowe
    .AddParameters(new Dictionary<string, object>
    {
        { "Order", null },
        { "Customer", null }
    })
    .Configure()
    //gdy WF zakończy pracę
    .OnComplete(() => {
        Console.WriteLine("Complete!");
    })
    //gdy WF wywoła błąd
    .OnError((ex) => {
        Console.WriteLine("Error: {0}", ex);
    })
    .Run();
```

Możemy korzystać z tej czytelnej składni w aplikacji MVVM, aby uruchamiać i monitorować przepływ zadań. Bardziej dogłębne informacje na temat silnika przepływów zadań można znaleźć w projekcie CRM.BL.WF, który zawiera implementację silnika przepływów zadań i wszystkie przepływy zadań dla przykładowej aplikacji CRM.

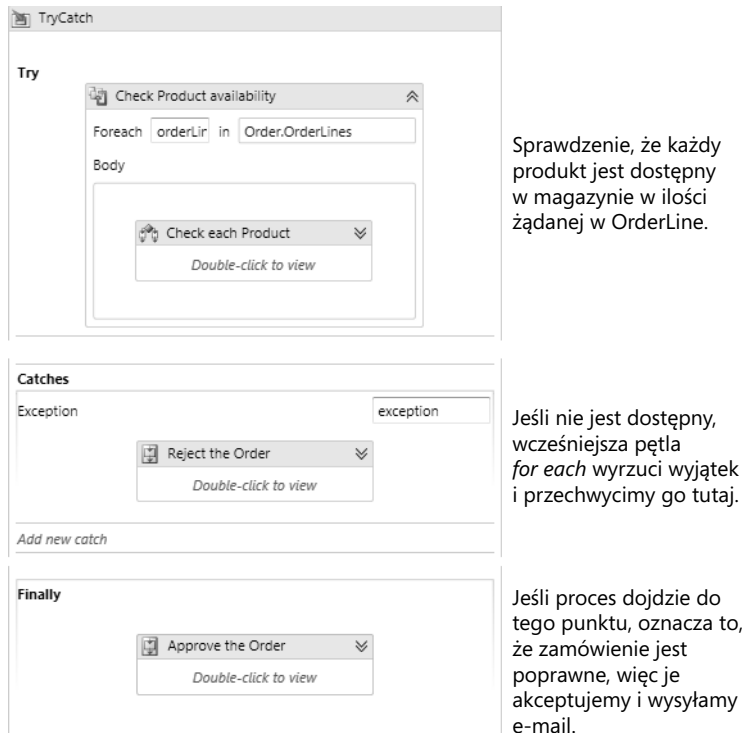
Usługa dla transakcji biznesowych

Mając gotowy podstawowy kod musimy zaimplementować wzorzec fasady dla warstwy BLL, aby przygotować usługi do użytku. W tej części zobaczymy, jak implementować usługę obsługującą proces dodawania nowego zamówienia od początku do końca. Następnie można opcjonalnie utworzyć niestandardowe reguły biznesowe albo można po prostu otworzyć końcowy projekt i zobaczyć, jak zaimplementowałem warstwę BLL.

Pierwszą ważną sprawą jest historia użytkownika, która będzie kierować transakcją biznesową:

Jako użytkownik chcę być w stanie tworzyć zamówienie (Order) i przesyłać zamówienie poprzez dodawanie go do odpowiedniego klienta (Customer), a następnie wysłanie potwierdzenia pocztą elektroniczną. Dla każdego produktu (Product) w zamówieniu muszę sprawdzić, czy dany produkt jest w magazynie.

Rysunek 5-7 pokazuje wynikowy przepływ zadań dla tej historii użytkownika podzielony na trzy części, aby łatwiej go było czytać.



RYSUNEK 5-7 Gotowy przepływ zadań dodający zamówienie

Ten przepływ zadań wymaga dwóch parametrów wejściowych *Customer* i *Order*. Zwraca wynik typu *Boolean*. Kod uruchamiający ten przepływ zadań powinien wyglądać podobnie, jak poniżej:

```
public bool CanAddAnOrder(Customer customer, Order order)
{
    //Inicjowanie klasy silnika
    FluentEngine.Init()
    //Ładowanie podzespołu i przepływu zadań
    .LoadAssembly("CRM.BL.WF.dll")
    .LoadActivity("CanAddAnOrder.xaml")
    //przygotowanie kolekcji parametrów
    // powinna ona zawierać parametry wejściowe/wyjściowe
```

```

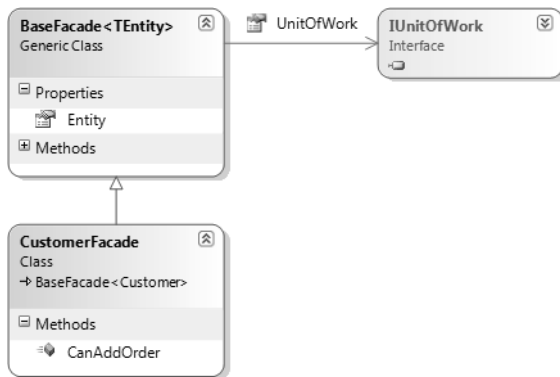
        .AddParameters(new Dictionary<string, object>
        {
            { "Order", order },
            { "Customer", customer },
            { "CanAddOrder", false}
        })
        .Configure()
        //gdy WF zakończy pracę
        .OnComplete((parm) => {
            return (bool)parm["CanAddOrder"];
        })
        //gdy WF wywoła błąd
        .OnError((ex) => {
            return false;
        })
        .Run();
    }
}

```

Ten przepływ zadań jest też zgodny z biurowymi wymaganiami dotyczącymi dokumentowania kodu związanego z transakcją biznesową.

Poprzedni kod powinien być zawarty w usłudze fasadowej. W używanym tutaj mamy interfejs *IUnitOfWork* i główną jednostkę biorącą udział w transakcjach wstrzyknięte w trakcie działania programu.

W warstwie biznesowej utworzyłem podstawową usługę, która ładuje prawidłowy interfejs *IUnitOfWork* wykorzystując platformę MEF – Managed Extensibility Framework (patrz rozdział 4) w trakcie działania aplikacji i odpowiednie jednostki wykorzystujące typy ogólne. Rysunek 5-8 pokazuje podstawową strukturę UML warstwy CRM.BL.



RYСУNEK 5-8 Struktura klasy fasady usługi biznesowej

Kod dla klasy bazowej jest dość oczywisty. Mamy klasę ogólną, która wymaga bieżącej jednostki jako parametru typu *DomainObject* oraz interfejs *IUnitOfWork* wstrzykiwany podczas działania aplikacji przez silnik MEF.

```

public class BaseFacade<TEntity> where TEntity : DomainObject
{
    /// <summary>
    /// Pobiera lub ustawia jednostkę pracy.
    /// </summary>
    /// <value>Jednostka pracy.</value>
    [Import]
    public IUnitOfWork UnitOfWork { get; private set; }

    /// <summary>
    /// Pobiera lub ustawia jednostkę.
    /// </summary>
    /// <value>Jednostka.</value>
    public TEntity Entity { get; private set; }

    /// <summary>
    /// Inicjuje nowe wystąpienie klasy <see cref="BaseFacade<TEntity>"/>.
    /// </summary>
    /// <param name="entity">Jednostka.</param>
    public BaseFacade(TEntity entity)
    {
        this.Entity = entity;
    }
}

```

Na podstawie tej ogólnej, usługowej klasy bazowej możemy tworzyć konkretną klasę usługi dla każdej jednostki i wykorzystywać element UoW lub podłączoną jednostkę bezpośrednio w usłudze. Poniższy kod jest na przykład usługą fasadową dla jednostki *Customer*. Ma metodę o nazwie *CanAddAnOrder*, która wymaga jedynie jednostki *Order*, ponieważ jednostka *Customer* jest wstrzykiwana w konstruktorze. MEF tworzy interfejs *IUnitOfWork*.

Wykorzystując to rozwiązanie możemy łatwo implementować wzorzec transakcji, gdzie dla ciągu kroków biznesowych włączamy wszystko do transakcji biznesowej, implementowanej w tym przypadku przez element UoW i instrukcję *Try/Catch*.

```

public class CustomerFacade : BaseFacade<Customer>
{
    public CustomerFacade(Customer customer) : base(customer)
    {
    }

    public bool CanAddOrder(Order order)
    {
        try
        {
            bool result = false;
            UnitOfWork.BeginTransaction();

            //poprzedni kod wykonujący przepływ zadań
            // result = WYKONANIE PRZEPŁYWU ZADAŃ
        }
    }
}

```

```

        if (result)
        {
            Entity.AddOrder(order);
            UnitOfWork.Update(Entity);
        }
        UnitOfWork.CommitTransaction();
        return result;
    }
    catch (Exception ex)
    {
        UnitOfWork.RollbackTransaction();
        throw new ApplicationException(
            "The CanAddOrder process has thrown an exception.", ex);
    }
}
}

```

Ten przykład stanowi punkt startowy dla dowolnej usługi biznesowej. Stosując to podejście uzyskamy dwie warstwy; jedna (CRM.BL w kodzie przykładowym) będzie warstwą bazową, która zawiera wszystkie klasy usług fasadowych, druga będzie warstwą przepływów zadań (CRM.BL.WF w kodzie przykładowym), która zawiera wszystkie reguły biznesowe (przepływy zadań lub proceduralny kod C#).

Zdaję sobie sprawę, że duży wysiłek jest związany z umieszczaniem tej logiki poza domeną lub warstwą danych, ale zaleta tego podejścia stanie się jasna, jak tylko będziemy musieli coś zmienić w aplikacji.

Trzeba też pamiętać o tym, że jeśli utrzymujemy logikę biznesową poza interfejsem użytkownika i poza domeną, to możemy być w stanie ponownie z niej skorzystać w innych aplikacjach bez konieczności przepisywania kodu.

Podsumowanie

Warstwa biznesowa znana również jako warstwa logiki biznesowej jest chyba najbardziej skomplikowaną i rozbudowaną warstwą aplikacji biznesowej. Warstwa logiki biznesowej jest zwykle podzielona na dwie główne części: reguły sprawdzania poprawności i reguły biznesowe. Jest to często mylone przez programistów, ale te dwie części mają dwa radykalnie różne cele.

Reguły sprawdzania poprawności odpowiadają za sprawdzanie poprawności wartości obiektu zgodnie ze zbiorem podstawowych reguł, takich jak sprawdzanie wyrażen regularnych, braku wartości null, długości łańcucha tekstowego, itd. Reguły biznesowe składają się z zestawów reguł, które definiują, jak dany obiekt powinien się zachowywać w oparciu o zestaw okoliczności lub wartości.

Możemy łatwo ustanawiać reguły sprawdzania poprawności wykorzystując przestrzeń nazw *System.ComponentModel* platformy .NET Framework albo wykorzystując dowolną otwartą bibliotekę sprawdzania poprawności, taką jak VAB z pakietu Enterprise Library 5.0.

Aby zaimplementować reguły biznesowe, możemy skorzystać z jednej z rozbudowanych (ale drogich) platform firm trzecich albo możemy dostosować silnik zestawów reguł zapewniany przez Windows Workflow Foundation 4.0, jak pokazano w tym rozdziale.

Chociaż budowanie elastycznej architektury dla warstwy biznesowej jest czasochłonne, to czas poświęcony na zbudowanie tej warstwy jest czasem, który zaoszczędzimy w przyszłości podczas utrzymywania aplikacji.

Warstwa interfejsu użytkownika w MVVM

Po zakończeniu tego rozdziału będziemy w stanie:

- Identyfikować części składające się na wzorec MVVM.
- Stosować wzorec polecenia i menedżera WeakEvent.
- Zapewniać dodatkowe usługi i elementy dla MVVM.

W tym rozdziale w końcu zagłębimy się we wzorec MVVM (Model View ViewModel) i zobaczymy, jak powinien być on implementowany, aby zachowywać całkowite rozdzielenie pomiędzy deklaratywną składnią interfejsu użytkownika opartą na XAML a kodem logiki prezentacyjnej interfejsu użytkownika.

Jak wspominałem w rozdziale 1 „Wprowadzenie do aplikacji biznesowych i wzorca Model View ViewModel”, firma Microsoft wprowadziła wzorec MVVM kilka lat temu, a nadal jest on gorącym tematem dyskusji w wielu grupach użytkowników i na forach. Ten rozdział zawiera więcej niż tylko mój osobisty punkt widzenia na temat tego, jak powinniśmy implementować MVVM, aby spełniać podstawowe zasady składające się na ten wzorec.

Chcemy implementować wzorec MVVM w dowolnej aplikacji biznesowej budowanej przy użyciu Silverlight lub Windows Presentation Foundation (WPF), ponieważ:

- Cała aplikacja kliencka powinna w pełni nadawać się do testowania, a w tym celu logika prezentacyjna widoku powinna być oddzielona od deklaratywnego kodu XAML, który go tworzy. Użycie wzorca prezentacyjnego, takiego jak MVVM, przenosi więcej zachowań aplikacji do klas będących poza interfejsem użytkownika, które mogą być łatwiej testowane.
- Chcemy ułatwić pracę projektanta interfejsu użytkownika pozostawiając tworzenie logiki prezentacyjnej innemu zespołowi/programiście.
- Oddzielenie logiki interfejsu użytkownika od deklaratywnych znaczników definiujących interfejs użytkownika ułatwia ponowne wykorzystywanie modelu widoku (ViewModel) w różnych widokach.

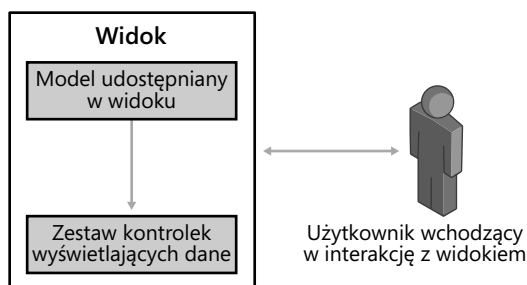
- Możemy łatwiej rozwijać lub zmieniać interfejs użytkownika bez zmieniania związanej z nim logiki prezentacyjnej aplikacji korzystając z możliwości silników *DataTemplate* i *DataBinding* zapewnianych przez znaczniki XAML.

Ze względu na swoją skomplikowaną strukturę prawidłowa implementacja wzorca MVVM wymaga głębokiego zrozumienia działania mechanizmów *DataTemplate*, *DataBinding*, *Styling* i *Dependencies* w WPF i Silverlight. Jednakże celem tej książki jest udzielenie wskazówek dotyczących implementacji aplikacji biznesowej przy użyciu wzorca MVVM – wyjaśnianie, jak te mechanizmy działają w WPF lub Silverlight, wykracza poza zakres tej książki. Zapewnię *ogólny przegląd* zastosowania *DataTemplate* i *Commanding*, ale jeśli ktoś nie zna dogłębnie tych pojęć, a zwłaszcza *DataTemplate* i *DataBinding*, to polecam zakup książki poświęconej danej technologii (WPF lub Silverlight), a następnie zapoznanie się i wypróbowanie tych dwóch złożonych mechanizmów XAML przed rozpoczęciem prób prawidłowego zaimplementowania samego wzorca MVVM.

Wprowadzenie do wzorca MVVM

Przed rozpoczęciem implementowania wzorca MVVM powinniśmy zobaczyć, jak on działa i jakie są jego główne składniki.

Gdy pomyślimy o zwykłej aplikacji warstwowej, to możemy zauważyć, że interfejs użytkownika jest złożony z czterech głównych obiektów składających się na jego cykl życia (patrz rysunek 6-1): widoku, który jest obiektem graficznym; modelu, który wiąże się z widokiem i reprezentuje pojęcia lub jednostki biznesowe; zbioru kontroltek używanych do interakcji pomiędzy widokiem a użytkownikiem oraz wyświetlających dane z modelu przy użyciu mechanizmu wiązania; a także zbioru zdarzeń wzbudzanych albo przez model, albo przez użytkownika poprzez widok.



RYСУNEK 6-1 Ogólna struktura widoku interfejsu użytkownika

Aby przełożyć to pojęcie na MVVM, możemy powiedzieć, że widokiem jest w XAML obiekt *UserControl/Page/Window*, który zawiera zestaw kontroltek i silnik wiązania *DataBinding*. Modelem jest jedna lub więcej jednostek domenowych udostępnianych

przez model widoku (ViewModel). Model widoku obejmuje logikę prezentacyjną w sposób, który nie jest specyficzny dla interfejsu użytkownika, natomiast widok obejmuje sam interfejs użytkownika. Możemy przeprowadzać testy jednostkowe modelu widoku bez konieczności odwoływania się do skomplikowanych testów interfejsu użytkownika, ponieważ model widoku jest „tylko kodem”.

Rysunek 6-2 pokazuje graficzne przedstawienie tego pojęcia wykorzystując prosty widok WPF.



RYСУNEK 6-2 Prosty widok w WPF ze związanym z nim modelem widoku

Gdy wiążemy model widoku z widokiem, to udostępniamy właściwości i zdarzenia modułu widoku interfejsowi użytkownika w celu ich przedstawienia użytkownikowi. Interakcje pomiędzy widokiem a modelem widoku następują poprzez wiązania danych, polecenia, itd.

W tym momencie wyzwaniem stanowi umiejętność dostosowania szablonu *DataTemplate* dla widoku w celu prawidłowego powiązania tych właściwości oraz określenie, które właściwości udostępnić poprzez model widoku. Szablony danych są szczególnym sposobem definiowania interfejsu użytkownika bez dodatkowego kodu – zasadniczo są sposobem definiowania widoku tak, aby był automatycznie wiązany z modelem widoku. Szablony danych są sposobem konstruowania interfejsu użytkownika, ale nie stanowią głównego wyzwania.

Widok

We wzorcu MVVM widok (View) jest interfejsem graficznym odpowiadającym za wyświetlanie danych użytkownikom i interakcje z użytkownikami. Jeśli piszemy aplikację WPF, widokiem może być obiekt *UserControl*, *Window* lub *Page*; jednakże w aplikacji Silverlight widokiem będzie kontrolka użytkownika Silverlight, strona Silverlight lub wyskakujące okno Silverlight.

Aby widok był zgodny z wzorcem MVVM, najpierw musimy dodać odwołanie do odpowiadającego mu modelu widoku we właściwości *DataContext* widoku. To pozwala nam rozpocząć wiązanie właściwości i poleceń modelu widoku z odpowiednimi

kontrolkami udostępnianymi w widoku. W tym celu po prostu dodajemy odwołanie do *DataContext* korzystając z podejścia proceduralnego, jak w następującym kodzie:

```
/// <summary>
/// Ustawia źródło danych.
/// </summary>
/// <param name="dataSource">Źródło danych.</param>
public void SetViewModel(PersonModel dataSource)
{
    this.DataContext = dataSource;
}
```

Alternatywnie możemy dodać odwołanie do źródła danych wykorzystując podejście deklaratywne XAML, jak w następującym kodzie XAML:

```
<Window x:Class="MVVM.MainWindow"
        xmlns:vm="clr-namespace:MVVM"
        Title="MainWindow" Height="250" Width="250">
    <Window.DataContext>
        <vm:PersonViewModel />
    </Window.DataContext>
```

Po wykonaniu tego kroku możemy bezpiecznie pozostawić projektantowi interfejsu użytkownika resztę pracy związanej z tworzeniem układu i przypisywaniem kontrolkom widoku powiązań z właściwościami modelu widoku. Niestety ten przykład nie w pełni separuje widok od modelu widoku, a także wprowadza ograniczenie pomiędzy rzeczywistym modelem widoku a modelem widoku wykorzystywanym przez projektanta – ponieważ w tym przykładzie są tym samym obiektem. Lepszym podejściem jest utworzenie fikcyjnego modelu widoku tylko dla projektantów tak, aby mogli nadal dostosowywać interfejs użytkownika, podczas gdy my (lub ktoś inny) możemy nadal pracować nad logiką prezentacyjną dla tego widoku.

Oczywiście właściwości, zdarzenia, polecenia, itd., zapewniane przez model widoku, reprezentują kontrakt dla widoku. Widok i model widoku nie są całkowicie niepowiązane; są luźno powiązane. Jeśli zdefiniujemy ten kontrakt z góry, to możemy utworzyć atrapę modelu widoku tak, aby projektant interfejsu użytkownika mógł skupić się na jego projektowaniu, natomiast programista mógł skupić się na implementowaniu i testowaniu modelu widoku.

Blend: atrapa modelu widoku

Zanim przełożymy odpowiedzialność za tworzenie powiązań projektantom, możemy ułatwić im pracę pozostając w zgodzie z MVVM poprzez wprowadzenie do procesu narzędzi Microsoft Expression Blend i Expression SDK. W ten sposób możemy umożliwić projektantom dokładny podgląd tworzonych widoków w Microsoft Visual Studio i/lub Expression Blend.

Firma Microsoft wprowadziła nową przestrzeń nazw dla projektantów WPF i Silverlight dostępną pod adresem <http://schemas.microsoft.com/expression/blend/2008>.

Expression Blend jest narzędziem do projektowania interfejsu użytkownika. Ma pakiet SDK zapewniający *zachowania*, które można rozszerzać. Zachowania są sposobem na opakowanie interaktywności w łatwe do ponownego wykorzystania składniki, które można przeciągać myszą na projekt interfejsu użytkownika. Expression Blend zapewnia również funkcję danych przykładowych, która umożliwia projektantom projektowanie interfejsu użytkownika z użyciem fikcyjnych danych. W terminologii wzorca MVVM te fikcyjne dane zapewniają atrapę modelu widoku.

Poniższy kod dodaje tę przestrzeń nazw do widoku wykorzystując przedrostek *des*:

```
xmlns:des="http://schemas.microsoft.com/expression/blend/2008"
des:DesignWidth="300" des:DesignHeight="300"
des:DataContext="{Binding SampleViewModel}"
```

Kod *des:DataContext* po prostu pozwala nam określić kontekst danych, który będzie używany w trakcie projektowania; rzeczywisty kontekst danych będzie używany podczas uruchamiania kodu. Dzięki tej technice możemy zastosować atrapę modelu widoku w trakcie projektowania tak, aby projektant mógł tworzyć projekt interfejsu użytkownika w oparciu o fikcyjne dane, które oczywiście zostaną zastąpione prawdziwymi danymi w trakcie działania aplikacji. Powyższy kod wiąże wykorzystywany w czasie projektowania kontekst danych widoku z klasą o nazwie *SampleViewModel*, Expression Blend tworzy obiekt *SampleDataSource* będący właściwie kolekcją właściwości, które możemy definiować, co pozwala projektantowi stworzyć atrapę modelu widoku z takimi samymi właściwościami, jak rzeczywisty model widoku, ale z wykorzystaniem fikcyjnych wartości danych.

W tym celu musimy utworzyć dodatkowy plik XAML, który miałby działać jako atrapa modelu widoku, a następnie wypełnić go fikcyjnymi danymi (które odwzorowują dane z rzeczywistego modelu widoku). Projektanci wykorzystują ten plik fikcyjnych danych podczas procesu projektowania logiki prezentacyjnej. Poniżej przedstawiono przykład *atrasy modelu widoku* w XAML dla widoku *Person*:

```
<vm:Customer
  xmlns:vm="clr-namespace:CRM.Domain.Domain;assembly=CRM.Domain"
  Title="Mr." FirstName="John" LastName="Smith"
  BirthDate="12/31/1970" IsActive="True">
  <vm:Customer.Contacts>
    <vm:Contact
      Name="Home Phone" ContactType="Phone"
      Number="111-11-11" IsDefault="True" />
    <vm:Contact
      Name="Office Phone" ContactType="Phone"
      Number="111-22-22" IsDefault="False" />
    <vm:Contact
      Name="Email" ContactType="Email"
      Number="john.smith@email.com" IsDefault="False" />
  </vm:Customer.Contacts>
  <vm:Customer.Addresses>
    <vm:Address
```

```

        AddressLine1="4 Main Street" City="New York"
        Country="USA" State="NY" ZipCode="11040" />
    <vm:Address
        AddressLine1="54 The Road" City="Seattle"
        Country="USA" State="WA" ZipCode="12000" />
    </vm:Customer.Addresses>
</vm:Customer>

```

Ten przykład tworzy wystąpienie modelu *Customer*, ponieważ model widoku udostępnia jedno wystąpienie tej klasy wraz z powiązaną z nim listą kontaktów i adresów. Pierwszy wiersz dodaje wystąpienie klasy *CRM.Domain.Customer* przy użyciu deklaracji *clr-namespace*, która jest dostępna w XAML; korzystamy z tej samej deklaracji, aby dodać odwołanie w widoku opartym na XAML. Ponieważ model *Customer* ma kilka podrzędnych kolekcji wymaganych dla ostatecznego widoku, są one również podane i wypełnione jakimiś fikcyjnymi danymi. To daje projektantom wszystko, czego potrzebują do utworzenia widoku w Expression Blend.

Gdy już projektanci mają wszystko na miejscu, mogą wiązać atrapę modelu widoku z widokiem i pracować nad procesem wiązania, szablonem *DataTemplate* oraz stylami bez przerywania pracy programisty nad innymi częściami aplikacji. Poniższy kod pokazuje składnię wiązania używaną w widoku XAML w celu skorzystania z wcześniej utworzonych fikcyjnych danych.

```

<UserControl x:Class="CRM.MVVM.WPF.DetailsView.CustomerDetails"
    <!--POMINIĘTE -->
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    mc:Ignorable="d"
    d:DesignHeight="500" d:DesignWidth="500" DataContext="{Binding}"
    d:DataContext="{d:DesignData Source=/DesignData/CustomerSampleData.xml}">
    <UserControl.Resources>

    <!--POMINIĘTE -->

    <StackPanel>
        <TextBlock>First Name :</TextBlock>
        <TextBox Text="{Binding FirstName}"/>
        <TextBlock>Last Name :</TextBlock>
        <TextBox Text="{Binding LastName}"/>
        <TextBlock Text="Data of Birth : " />
        <DatePickerTextBox Text="{Binding Path=BirthDate.Date, StringFormat=\{0:d\}"} />
        <TextBlock>Is Active :</TextBlock>
        <CheckBox IsChecked="{Binding IsActive}"/>
    </StackPanel>

```

Wykorzystując to podejście projektant może pracować bezpośrednio albo w Expression Blend, albo w Visual Studio, żeby testować interfejs użytkownika bez faktycznego uruchamiania aplikacji, ponieważ silnik obrazujący interfejs użytkownika dostępny w Expression Blend i Visual Studio może obrazować dane w czasie projektowania, jak pokazano na rysunku 6-3.

RYСУNEK 6-3 Widok XAML w czasie projektowania wykorzystujący atrapę modelu widoku.

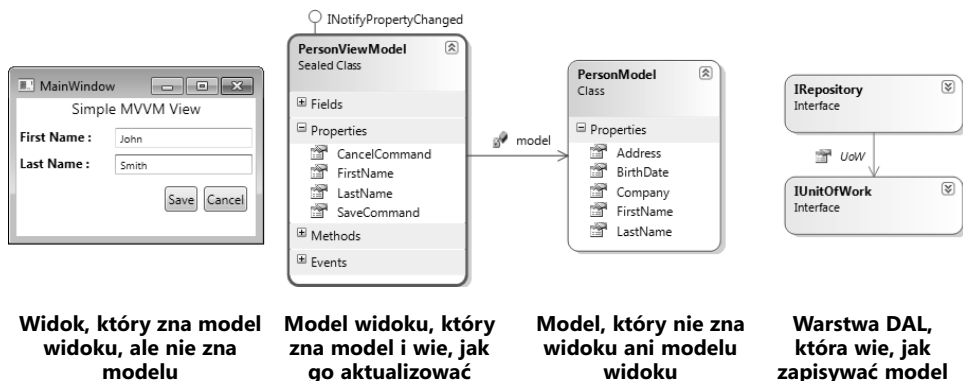
To podejście wygląda na dość wygodne na pierwszy rzut oka, więc warto zwrócić uwagę na kilka możliwych wad jego zastosowania.

Po pierwsze, jeśli zamierzamy dopasowywać interfejs użytkownika wykorzystując strukturę ostatecznego modelu widoku, to musimy utworzyć model widoku na czas projektowania, który odpowiada wszystkim właściwościom, poleceniom i zachowaniom, które planujemy udostępnić w ostatecznym modelu widoku. Po drugie musimy pamiętać, że w przypadku ogromnych modeli widoków – na przykład modelu widoku, który udostępnia 30 właściwości plus polecenia i reguły sprawdzania poprawności danych – utworzenie atrapy modelu widoku zajmie znaczącą ilość czasu. Na koniec trzeba pamiętać, że może nie być możliwe utworzenie fikcyjnej wartości, która będzie odpowiadać każdej właściwości, poleceniu lub zachowaniu, jeśli nasz rzeczywisty model widoku jest skomplikowany, więc konieczne może być pomyślenie o innych rozwiązaniach. Na przykład możemy tworzyć skomplikowaną atrapę modelu widoku złożoną z nadrzędnego modelu widoku i podrzędnych modeli widoków. Taka złożoność może być trudna do przedstawienia przy użyciu znaczników XAML.

Model

Jak można było zauważyć, ten rozdział omawiał model widoku – a nie model omawiany w rozdziale 3. W istocie model jest jednostką domenową deklarowaną oraz udostępnianą w modelu domenowym i nie powinien być mylony z modelem widoku, który jest udostępniany w widoku.

Rozwijając to dalej, model jest jednostką odpowiedzialną za przenoszenie danych do i z magazynu danych; jest to obiekt znany warstwie danych i warstwie biznesowej. Z kolei model widoku jest modelem dla widoku – częścią modelu udostępnianą dla tego określonego widoku i obejmującą sprawdzanie poprawności oraz zachowania potrzebne w tym konkretnym przypadku interfejsu użytkownika, a także całą logikę prezentacyjną. Rysunek 6-4 pokazuje przepływ modelu i dlaczego jest on często tak różny od modelu widoku.



RYSUNEK 6-4 Przepływ używany w aplikacji MVVM

Trzeba wziąć pod uwagę, że *nigdy* nie będziemy implementować interfejsu *ICommand* w modelu, ponieważ model nie jest związany z interfejsem użytkownika, ale będziemy prawdopodobnie udostępniać model bezpośrednio z modelu widoku, aby ułatwić tworzenie niestandardowego szablonu *DataTemplate* dla określonej jednostki domenowej z naszego modelu domenowego. Jest to problem, który teraz przeanalizujemy. Częstym błędem, który widziałem w implementacjach MVVM, jest to, że przekazują one model widokowi z modelu widoku tak, że (na przykład) ścieżka wiązania właściwości *FirstName* w modelu *Person* byłaby udostępniona w widoku w następujący sposób:

```
<TextBox Grid.Column="2" Grid.Row="1" Text="{Binding PersonModel.FirstName}" />
```

Moim zdaniem, zamiast pozwalać widokowi na bezpośrednie wiązanie z właściwością z modelu, model widoku powinien udostępniać swoją własną, oddzielną właściwość o nazwie *FirstName*, która reprezentuje właściwość *FirstName* modelu *Person*:

```
<TextBox Grid.Column="2" Grid.Row="1" Text="{Binding FirstName}" />
```

Przy zastosowaniu tego podejścia model widoku staje się modelem dla widoku maskującym rzeczywisty model. Poprawia to bezpieczeństwo aplikacji, ponieważ nie udostępniamy całego modelu bezpośrednio w widoku. Ponadto pomaga oddzielić widok od modelu, ponieważ widok nie musi już nic bezpośrednio wiedzieć na temat modelu.

Udostępnianie modelu w modelu widoku

W przeszłości prowadziłem dyskusje na ten temat z wieloma osobami. Rozwiązanie udostępniające model bezpośrednio z modelu widoku jako właściwość publiczną, a następnie wiążące go bezpośrednio z widokiem, jest chyba najłatwiejszym i najszybszym rozwiązaniem – ale nie stanowi *czystego* sposobu implementowania wzorca rozdzielonej prezentacji, w którym widok powinien być jedynie luźno powiązany z modelem poprzez konkretny model widoku.

Zamiast tego model widoku powinien deklarować swoje własne właściwości ukrywając szczegóły modelu przed widokiem. Zapewnia to większą elastyczność i pomaga zapobiegać przedostawaniu się problemów związanych z modelami widoków do klas modeli.

Musimy pamiętać, że jeśli planujemy udostępniać właściwości swojego modelu bezpośrednio w widoku, poprzez umieszczanie modelu jako właściwości modelu widoku, to powinniśmy implementować interfejs *INotifyPropertyChanged* także w obiekcie jednostki domenowej, a nie tylko w modelu widoku; w przeciwnym razie, gdy widok zmieni model, to model widoku nie będzie w stanie zobaczyć tej zmiany, ponieważ silnik wiązania WPF lub Silverlight wzbudzi powiadomienie o zmianie.

Z drugiej strony użycie podejścia z przepisywaniem każdej właściwości modelu (włącznie z relacjami pomiędzy obiektami podrzędnymi i nadrzędnymi) wewnątrz odpowiadającego mu modelu widoku jest żmudnym, czasochłonnym i podatnym na błędy zadaniem, co zwiększa ilość pracy potrzebnej na testowanie i utrzymywanie.

Można się zastanawiać, które podejście jest najlepsze. Uczciwie mówiąc nie ma „najlepszego” podejścia; są tylko różne podejścia do tego samego problemu. Jeśli chcemy udostępniać model bezpośrednio w widoku tak, abyśmy mogli łatwo napisać szablon *DataTemplate*, który reprezentuje daną jednostkę domenową, to będziemy musieli zanieczyścić swoje jednostki domenowe interfejsem *INotifyPropertyChanged*. Z drugiej strony, jeśli chcemy być purystami, to będzie wymagało od nas napisania i przetestowania dużo większej ilości kodu. Powiedziałbym, że „najlepsze” podejście zależy od stopnia skomplikowania architektury naszej aplikacji.

Polecenie w WPF i Silverlight

Jedną z najciekawszych funkcji w WPF i Silverlight jest interfejs *ICommand* i sposób jego działania. Interfejs *ICommand* udostępnia metody *Execute* i *CanExecute*, które umożliwiają nam kontrolę nad wykonywaniem polecenia. Wykorzystując silnik wiązania w WPF lub Silverlight oraz implementację *ICommand* jesteśmy w stanie utworzyć model widoku, który udostępnia polecenia *ICommand* w widoku i wiąże kontrolki, takie jak *Button*, *Link*, itd. z tymi poleceniami. Interfejs *ICommand* pozwala nam sterować wykonywaniem polecenia w oparciu o zmiany, które mogą następować w modelu widoku. Na przykład możemy włączyć polecenie *Save* w widoku tylko po tym, gdy model widoku wywołał metodę *OnPropertyChanged()* przynajmniej raz.

Zwykle będziemy musieli udostępniać te polecenia w modelu widoku jako właściwości publiczne, aby prawidłowo tworzyć wiązanie pomiędzy widokiem a modelem widoku. Udostępnianie właściwości *ICommand* z modelu widoku pozwala widokowi

wiązać się z poleceniem przygotowanym przez model widoku. Możemy implementować interfejs *ICommand* na wiele sposobów. Musimy też zaimplementować jakąś logikę prezentacyjną w swoim modelu widoku, aby ustalić, czy polecenie może, czy nie może być wykonywane.

Typowym rozwiązaniem jest po prostu utworzenie publicznej właściwości typu *ICommand* w modelu widoku z prywatną procedurą dostępu, która może sprawdzać jakąś logikę prezentacyjną w samym modelu widoku. Inną możliwą implementacją jest utworzenie niestandardowej klasy dla każdego polecenia, które dziedziczy po interfejsie *ICommand* i udostępnia go w modelu widoku – ale oczywiście wtedy musielibyśmy tworzyć niestandardową klasę polecenia dla każdego polecenia dostępnego w aplikacji. Można by sądzić, że to podejście jest czasochłonne, jednak w przypadku typowych poleceń, takich jak Nowy, Zapisz, Usuń, Cofnij, czy Ponów, będziemy musieli napisać niestandardową implementację tych poleceń tylko raz. W przypadku innych poleceń możemy chcieć skorzystać z podejścia MVVM Command, które zostało wyjaśnione w następnym podrozdziale.

Obejście problemu: MVVM Command

W wersji Silverlight 3 nie było obsługi funkcji poleceń, która jest dostępna w WPF. Niestety również wersja Silverlight 4 nie obsługuje poleceń w taki sam sposób jak WPF. Jednak dzięki interfejsowi *ICommand* , który poznaliśmy wcześniej, możemy łatwo wiązać polecenie z menu w Silverlight i ponownie wykorzystywać to samo polecenie przy wiązaniu z przyciskiem w WPF bez potrzeby przepisywania kodu.

Niektóre narzędzia, takie jak Prism (platforma zespołu Microsoft patterns & practices), MVVM Light Toolkit (platforma Laurenta Bugniona) i Caliburn (projekt CodePlex), mają swoje własne implementacje *ICommand* , które można wykorzystywać zarówno w aplikacjach Silverlight, jak i WPF. Kod, który zobaczymy w tym podrozdziale, robi to samo tworząc element *MVVMCommand* , który możemy udostępniać w modelach widoków, zamiast kodować implementację *ICommand* .

Najpierw tworzymy nowy projekt CRM.MVVM, który będzie platformą narzędziową dla wzorca MVVM. Tutaj projekt przechowuje pewne klasy narzędziowe, do których należą implementacje polecenia MVVM dla WPF i Silverlight.

Pierwszą klasą, którą zbudujemy, jest *MvvmCommand* , która musi implementować interfejs *ICommand* . Definiuje ogólną *Funkcję<T>* dla oceny właściwości *CanExecute* i *Delegata<T>* dla implementacji *Execute* . Te metody są wstrzykiwane do konstruktora polecenia przy pomocy następującego kodu:

```
/// <summary>
/// Niestandardowe polecenie MVVM
/// </summary>
public class MvvmCommand : ICommand
{
    private readonly Func<object, bool> canExecute;
```

```

private readonly Action<object> executeAction;
private bool canExecuteCache;

/// <summary>
/// Inicjuje nowe wystąpienie klasy <see cref="MvvmCommand"/>.
/// </summary>
/// <param name="executeAction">Wykonywane działanie.</param>
/// <param name="canExecute">Można wykonać.</param>
public MvvmCommand(Action<object> executeAction, Func<object, bool> canExecute)
{
    this.executeAction = executeAction;
    this.canExecute = canExecute;
}
}

```

Oczywiście ten typ implementacji wymusza na nas implementację logiki wykonywania metody *Execute* i *CanExecute* poza samym poleceniem – prawdopodobnie bezpośrednio w model widoku, który je udostępnia.

Najpierw implementujemy metodę *CanExecute*, która ocenia, czy polecenie może, czy nie może być wykonywane. To działanie jest obsługiwane zarówno przez silniki WPF, jak i Silverlight, ale w różny sposób. Na przykład WPF ma klasę menedżera poleceń, która ponownie sprawdza interfejs użytkownika (i oczywiście powiązany model widoku) za każdym razem, gdy interfejs użytkownika się zmieni. Zmiana interfejsu użytkownika automatycznie wywołuje ponowną ocenę działania *CanExecute*. Z kolei silnik Silverlight nie ma menedżera poleceń, więc musimy implementować to sprawdzenie sami.

Poniższy kod przedstawia prostą implementację *CanExecute*, która wywołuje zdarzenie za każdym razem, gdy ta wartość polecenia jest sprawdzana:

```

/// Definiuje metodę, która określa, czy polecenie
/// może zostać wykonane w swoim aktualnym stanie.
/// </summary>
/// <param name="parameter">Dane wykorzystywane przez polecenie. Jeśli polecenie
/// nie wymaga przekazania danych, ten obiekt może być ustawiony na null.</param>
/// <returns>
/// true, jeśli to polecenie może być wykonywane; w przeciwnym razie, false.
/// </returns>
public bool CanExecute(object parameter)
{
    if (CanExecuteChanged != null)
    {
        CanExecuteChanged(this, new EventArgs());
    }
    return canExecute(parameter);
}

public event EventHandler CanExecuteChanged;

```

Teraz, gdy możemy oceniać możliwość wykonywania polecenia, możemy po prostu skojarzyć delegata zapewnianego w konstruktorze z tym, który jest wymagany przez interfejs *ICommand* w następujący sposób

```
/// <summary>
/// Definiuje metodę, która ma zostać wywołana, gdy to polecenie jest uruchamiane.
/// </summary>
/// <param name="parameter">Dane wykorzystywane przez polecenie. Jeśli polecenie
/// nie wymaga przekazania danych, ten obiekt może być ustawiony na null.</param>
public void Execute(object parameter)
{
    executeAction(parameter);
}
```

Jak to skonfigurowano tutaj, możemy deklarować polecenie w modelu widoku z prywatnym dostępem i przypisać dwa wyrażenia lambda, żeby uzyskać konkretną implementację polecenia MVVM w następujący sposób:

```
public sealed class PersonViewModel : BaseViewModel<Person>
{
    public ICommand SavePerson { get; private set; }

    /// <summary>
    /// Inicjuje nowe wystąpienie klasy <see cref="PersonViewModel"/> .
    /// </summary>
    /// <param name="model">Model.</param>
    public PersonViewModel(Person model)
        : base(model){}

    /// <summary>
    /// Inicjuje polecenia.
    /// </summary>
    private void InitCommands()
    {
        SavePerson = new MvvmCommand(
            (parm) =>
            {
                // wykonaj
                PersonService.Save(model);
            },
            (parm) =>
            {
                // canExecute, można zapisać, jeśli
                // model jest poprawny ...
                return model.IsValid;
            });
    }
}
```

Teraz za każdym razem, gdy zmieniamy *Person* (w tym konkretnym przypadku), interfejs użytkownika na nowo sprawdzi polecenie *Save*, a jeśli model *Person* nie będzie prawidłowy, to wyłączy przycisk *Save*. Potrzebna nam jeszcze tylko ostatnia zmiana w celu naprawienia problemu z brakiem menedżera połączeń w Silverlight.

Ponowne ocenianie możliwości wykonywania *ICommand*

To, czego dokonaliśmy do tej pory, jest elastyczne i łatwe do testowania; możemy utworzyć ogólne polecenie *MvvmCommand*, udostępnić je jako obiekt interfejsu *ICommand* i zadeklarować kod do wykonywania polecenia i oceny możliwości jego wykonania przy użyciu anonimowego delegata, co również wygląda świetnie i jest dosyć czytelne.

W tym momencie możemy spróbować utworzyć podstawowy obiekt modelu widoku, który udostępnia kilka właściwości oraz właściwość *ICommand*, podobnie jak na wydruku poniżej, oraz powiązać te właściwości z widokiem. Chcemy włączać właściwość *FormatCommand* tylko wtedy, jeśli tekst w kontrolce *TextBox* jest różny od null.

Metoda *CanExecute* obiektu *ICommand* jest wykonywana tylko wtedy, gdy silnik wiązania danych tworzy wiązanie pomiędzy elementem interfejsu użytkownika a poleceniem, a później możliwość wykonywania polecenia jest sprawdzana tylko wtedy, gdy coś się zmieni, a obiekt *CommandManager* nasłuchuje wystąpienia tej zmiany. Na przykład, gdy zmienimy tekst w kontrolce *TextBox*, obiekt *CommandManager* nie jest świadom tej zmiany i nie aktualizuje polecenia, więc przycisk uruchamiający polecenie pozostanie wyłączony.

Gdyby kod nie był w modelu widoku, ale w kodzie stojącym za danym obiektem *Window* lub *UserControl*, to moglibyśmy wywołać statyczną metodę *CommandManager.InvalidateRequerySuggested*, która wzbudza zdarzenie *RequerySuggested*, które z kolei ponownie sprawdza możliwość wykonywania wszystkich poleceń wewnątrz obiektu *CommandManager*. Niestety, jeśli wywołamy tę metodę wewnątrz obiektu modelu widoku, to po prostu nie zadziała, ponieważ nie mamy bezpośredniego dostępu do obiektu *CommandManager* w widoku.

Inną dużą wadą wykorzystania obiektu *CommandManager* jest to, że nie sprawdza on możliwości wykonywania tylko jednego polecenia; ocenia ponownie możliwość wykonania wszystkich poleceń przyłączonych do kolekcji *CommandBinding*.

Ewentualną alternatywą jest ręczne sprawdzanie możliwości wykonania polecenia za każdym razem, gdy zmieni się właściwość *OriginalText*, jak w następującym kodzie:

```
public string OriginalText
{
    get { return originalText; }
    set
    {
        originalText = value;
        OnPropertyChanged(vm => vm.OriginalText);
        (FormatCommand as MvvmCommand).OnCanExecuteChanged();
    }
}
```

Inną ciekawą alternatywą byłoby uświadamianie poleceniu zmian, które mogą występować w modelu widoku i ponowne wykonywanie metody *OnCanExecuteChanged()*, jeśli ta „zmiana” nastąpi w modelu widoku. Innymi słowy oznacza to, że polecenie *ICommand* będzie nasłuchiwać zdarzenia *PropertyChanged* wzbudzanego przez model widoku.

Model widoku

Przypomnijmy sobie, że klasyczna definicja modelu widoku (ViewModel) we wzorcu MVVM brzmi: „model widoku jest modelem zapewnianym dla określonego widoku”, co niekoniecznie wystarcza do opisanego znaczenia tego obiektu podczas procesu tworzenia aplikacji.

Model widoku powinien spełniać cztery główne wymagania:

- Zapewniać dane, które muszą być udostępniane w widoku
- Zapewniać zestaw poleceń dostępnych w widoku
- Implementować interfejs *INotifyPropertyChanged*
- Implementować interfejs *IDataErrorInfo*

Oczywiście nie wszystkie implementacje modelu widoku muszą spełniać wszystkie cztery wymagania. W zależności od sytuacji model widoku niekoniecznie musi udostępniać zestaw poleceń *ICommand* albo implementować interfejs *IDataErrorInfo* (używany do sprawdzania poprawności danych w interfejsie użytkownika) albo interfejs *INotifyPropertyChanged* (który wzbudza powiadomienia o zmianach w interfejsie użytkownika). Implementacja danego modelu widoku zależy od konkretnych przypadków użycia. Jednak wszystkie modele widoków będą udostępniać przynajmniej niektóre wartości.

Poprzedni podrozdział omówił zalety i wady udostępniania modelu z poziomu modelu widoku. Dla jasności udostępnię bezpośrednio w widoku jednostki domenowej aplikacji przykładowej z modelu widoku – ale nie oznacza to, że *musimy* udostępniać dane w swoim modelu widoku przy użyciu tego podejścia.

Następne podrozdziały wyjaśniają, jak powinniśmy implementować te cztery wymagania modelu widoku i jak tworzyć niestandardowe, podstawowe modele widoków, które możemy następnie ponownie wykorzystywać w swoich przyszłych aplikacjach MVVM.

Interfejs *INotifyPropertyChanged*

Interfejs *INotifyPropertyChanged* jest dostępny od wersji 2.0 platformy .NET Framework. Znajduje się w podzespole *System.dll* i jest udostępniany poprzez przestrzeń nazw *System.ComponentModel*. *INotifyPropertyChanged* zapewnia mechanizm powiadamiania klienta lub dowolnego innego odbiorcę, że zmianie uległa wartość jakiejś właściwości (lub całego obiektu). Udostępnia zdarzenie *PropertyChanged*, które wymaga niestandardowej implementacji w dziedziczących klasach. Jeśli powiążemy obiekt, który implementuje ten interfejs, ze źródłem danych XAML, to widok będzie otrzymywał powiadomienie za każdym razem, gdy obiekt ten się zmieni. W ten sam sposób, jeśli powiążemy taki obiekt ze źródłem danych Windows Form, to samo zachowanie wystąpi bez konieczności modyfikowania kodu ze względu na zmianę źródła danych.

W tym momencie możemy traktować dowolny obiekt, który implementuje interfejs *INotifyPropertyChanged*, jako obiekt obserwowalny (*Observable*) będący abstrakcyjnym typem obiektu, który utworzymy tutaj dla swojego pakietu narzędzi MVVM. Jedynym wymaganiem jest, aby obiekt obserwowalny implementował interfejs *INotifyPropertyChanged*, który jest abstrakcyjny, gdyż jest klasą bazową; nie chcemy korzystać z niego bezpośrednio. Na koniec chcemy zdefiniować właściwość, która się zmieniła, korzystając z wyrażeń lambda.

Na początek utworzymy nową klasę w projekcie CRM.MVVM i nazwijmy ją **ObservableObject**, jak pokazano tutaj:

```
public abstract class ObservableObject<T> : INotifyPropertyChanged
```

Każdy obiekt, który implementuje ten interfejs, będzie wykorzystywał samego siebie jako typ *<T>*. W ten sposób możemy teraz użyć sztuczki z wyrażeniem lambda do automatycznego ustalenia nazwy właściwości. Następnym krokiem jest implementacja interfejsu:

```
#region Implementation of INotifyPropertyChanged
```

```
/// <summary>
/// Występuje, gdy zmienia się wartość właściwości.
/// </summary>
public event PropertyChangedEventHandler PropertyChanged;

/// <summary>
/// Wywoływane, gdy [właściwość się zmienia].
/// </summary>
/// <param name="property">Właściwość.</param>
protected virtual void OnPropertyChanged(Expression<Func<T, object>> property)
{
    if (property == null || property.Body == null)
    {
        return;
    }

    var memberExp = property.Body as MemberExpression;
    if (memberExp == null)
    {
        return;
    }

    PropertyChangedEventHandler handler = PropertyChanged;
    if (handler != null)
    {
        handler(this, new PropertyChangedEventArgs(memberExp.Member.Name));
    }
}
```

```
#endregion
```

Powyższy kod deklaruje zdarzenie *PropertyChanged*, które wymaga argumentu *PropertyChangedEventArgs* zawierającego nazwę właściwości, która się zmieniła. Następnie mamy sygnaturę delegata *OnPropertyChanged* będącego metodą, która będzie wywoływana za każdym razem, gdy właściwość się zmieni. Warto zauważyć, że ten kod nie wykorzystuje wyniku skompilowanego wyrażenia lambda (co jest wolniejsze), ponieważ chcemy odczytywać jedynie wartość zawartą w wyrażeniu lambda; ta technika nie wpłynie na wydajność wykonywania kodu. Jeśli są jacyś subskrybenci zdarzenia, wzbudzamy to zdarzenie dołączając nazwę właściwości.

Teraz możemy zaimplementować tę klasę w swoim podstawowym modelu widoku w następujący sposób:

```
public class BaseViewModel<T> : ObservableObject<BaseViewModel<T>> where T : class
{
    public T model;

    /// <summary>
    /// Pobiera lub ustawia model.
    /// </summary>
    /// <value>Model.</value>
    public T Model
    {
        get { return model; }
        set
        {
            if (model == value)
            {
                return;
            }
            model = value;
            OnPropertyChanged(vm => vm.Model);
        }
    }
}
```

Ten przykład pokazuje bazową klasę modelu widoku, która wymaga ogólnego modelu *<T>*. Jest to model, który będziemy udostępniać w widoku wykorzystując silnik *Data-Bind* w XAML. Jeśli model się zmienia, to powiadomi interfejs użytkownika wywołując zdarzenie *OnPropertyChanged*.

Możemy zrobić to samo z prostymi właściwościami, takimi jak łańcuchy tekstowe lub liczby całkowite poprzez model widoku.

Interfejs *IDataErrorInfo*

Interfejs *IDataErrorInfo* również znajduje się w przestrzeni nazw *System.ComponentModel*. Jego zadaniem jest zapewnianie informacji o błędach obiektowi powiązanemu z interfejsem klienta (widokowi). Ten interfejs był dostępny w .NET Framework od wersji 1.0 (choć miał nieco inną strukturę), ale stał się sławny dopiero po pojawieniu się WPF i Silverlight. Jednakże możemy łatwo korzystać z niego w aplikacji klienckiej

Windows lub ASP.NET do implementowania sprawdzania poprawności danych w widoku.

Interfejs ten udostępnia dwie właściwości: *Error* i *Item*. Właściwość *Error* reprezentuje aktualny błąd sprawdzania poprawności. Jest najczęściej implementowany po stronie klienta, więc nie będziemy implementować tej właściwości w przykładowym pakiecie narzędzi, ponieważ będziemy wyświetlać błędy sprawdzania poprawności korzystając z szablonu danych XAML.

Element *Item* jest wywoływany za każdym razem, gdy element w widoku (który ma włączone sprawdzanie poprawności, więc jakakolwiek zmiana wywołuje silnik sprawdzania poprawności) zmienia swoją wartość i/lub wymaga sprawdzenia poprawności danych.

Prosta implementacja tego interfejsu w bazowym modelu widoku powinna wyglądać podobnie, jak poniżej:

```
/// <summary>
/// Pobiera <see cref="System.String"/> z nazwą określonej kolumny.
/// </summary>
/// <value></value>
public virtual string this[string columnName]
{
    get
    {
        var errorMessage = string.Empty;
        switch (columnName)
        {
            case "Model":
                if (this.Model == null)
                {
                    errorMessage = "The View can't be bound to an empty model.";
                }
                break;
        }
        return errorMessage;
    }
}
```

Powinien być opatrzony słowem kluczowym *virtual* tak, abyśmy mogli zmieniać tę implementację w każdym konkretnym modelu widoku. Na przykład *PersonView-Model* mógłby mieć inne reguły sprawdzania poprawności. Ten proces nie jest zbyt efektywny, zajmuje sporo czasu i jest prawdopodobnie redundantny, ponieważ być może zastosowaliśmy już jakieś reguły sprawdzania poprawności w bazowym modelu.

Lepszym sposobem realizacji tego zadania jest wykorzystanie biblioteki Microsoft Enterprise Library, którą poznaliśmy w rozdziale 5 „Warstwa biznesowa”, aby sprawdzać poprawność zarówno w modelu widoku, jak i w bazowym modelu, który ma już reguły sprawdzania poprawności.

W tym celu najpierw potrzebna jest nam klasa *ViewModelValidator*, którą będziemy mogli wywoływać, gdy tylko będziemy musieli sprawdzić poprawność właściwości

modelu widoku lub poprawność całego obiektu. Moduł Validation Application Block (VAB) dostępny w Enterprise Library oferuje zgrabny i łatwy sposób sprawdzania poprawności obiektu, więc proponuję skorzystanie z niego i upewnienie się, że wszelkie występujące błędy sprawdzania poprawności są związane z właściwością, której poprawność staramy się sprawdzić. Oto kod do sprawdzenia poprawności pola:

```
public sealed class ViewModelValidator
{
    /// <summary>
    /// Sprawdza poprawność pola.
    /// </summary>
    /// <param name="entity">Jednostka.</param>
    /// <param name="field">Pole.</param>
    /// <returns></returns>
    public static string ValidateField<T>(T entity, string field)
    {
        var validationResults =
            ValidationFactory.CreateValidator<T>().Validate(entity);
        var errorMessage = new StringBuilder();
        // jeśli jednostka jest poprawna, nie idziemy dalej
        if (validationResults.IsValid)
        {
            return errorMessage.ToString();
        }
        // potwierdź, czy błędy dotyczą tego pola
        var errors = validationResults.Where(x => x.Key == field);
        if (errors.Count() > 0)
        {
            foreach (var validationResult in errors)
            {
                errorMessage.AppendLine(validationResult.Message);
            }
        }
        // zwraca komunikat błędu jako string z \r\n
        return errorMessage.ToString();
    }
}
```

A oto zmiana w bazowym modelu widoku:

```
/// <summary>
/// Pobiera <see cref="System.String"/> z nazwą określonej kolumny.
/// </summary>
/// <value></value>
public virtual string this[string columnName]
{
    get
    {
        return ViewModelValidator.ValidateField(this, columnName);
    }
}
```

Sztuczka polega teraz na powiązaniu modelu widoku z widokiem i utworzeniu określonej metody wizualnej – na przykład kontrolki *TextBox* – do wyświetlania wszelkich błędów generowanych podczas procesu sprawdzania poprawności. Rysunek 6-5 przedstawia funkcjonalny styl sprawdzania poprawności danych, który wykorzystuje kontrolkę *TextBox* w WPF.

RYСУNEK 6-5 Szablon sprawdzania poprawności danych zastosowany wobec kontrolki *TextBox* w WPF.

Aby zobaczyć więcej informacji na temat szablonu sprawdzania poprawności, należy zbadać właściwość dołączaną *Validation.ErrorTemplate*. Ta właściwość pozwala nam zdefiniować określony szablon dla kontrolki wyświetlającej błąd sprawdzania poprawności danych.

Oczywiście w celu wywołania w ogóle sprawdzania poprawności danych musimy zdefiniować wiązanie, jak pokazano w następującym przykładzie kodu, określając atrybuty *ValidatesOnDataErrors* i *ValidateOnExceptions*, żeby silnik wiązania był świadomy dostępności sprawdzania poprawności danych w kontekście *DataContext*:

```
<TextBox Text="{  
    Binding ValidatesOnDataErrors=True,  
    Path=FirstName,  
    ValidatesOnExceptions=True  
}">  
</TextBox>
```

Szablon *DataTemplate* w WPF i Silverlight

Innym ważnym aspektem silnika interfejsu użytkownika WPF/Silverlight jest szablon *DataTemplate*, który opisuje, jak przedstawiać dane, które są powiązane z kontrolką. Technologia Windows Forms nie ma łatwego sposobu na dostosowywanie elementów udostępnianych przez kontrolkę listy, taką jak *Listbox*, więc trzeba było tworzyć w modelu niestandardową właściwość udostępnianą w elemencie *DataSource* kontrolki, aby korzystać z niej przy wyświetlaniu zawartości tej kontrolki.

W przypadku WPF lub Silverlight możemy jednak łatwo powiązać klasę *Person* z elementem *UserControl* i powiązać właściwość tej klasy będącą kolekcją (na przykład listę adresów) z kontrolką *Listbox*. Dostosowując szablon danych możemy sprawić, że interfejs użytkownika kontrolki *Listbox* będzie przypominał bardziej prostą kontrolkę *Grid* i uniknąć przy tym zastosowania bardziej skomplikowanej kontrolki *Grid*. Poniższy kod wykorzystuje szablon *DataTemplate* do wyświetlenia trzech właściwości jednostki *Address* w kontrolce *Listbox*:

```
<ListBox ItemsSource="{Binding Contacts}"
  Grid.Column="5" Grid.ColumnSpan="3"
  Grid.Row="2" Grid.RowSpan="3">
  <ListBox.ItemTemplate>
    <DataTemplate>
      <StackPanel Orientation="Horizontal">
        <TextBlock Text="{Binding Name}" />
        <TextBlock Text=" : " />
        <TextBlock Text="{Binding Number}" />
      </StackPanel>
    </DataTemplate>
  </ListBox.ItemTemplate>
</ListBox>
```

Innym interesującym podejściem byłoby użycie interfejsu *IList<ICommand>* w modelu widoku. Następnie korzystając z szablonu *DataTemplate* moglibyśmy tworzyć dynamiczną listę przycisków lub listę poleceń w pasku poleceń, co dałoby nam niezwykle dynamiczny widok.

DataTemplate a MVVM

Dlaczego szablon *DataTemplate* jest tak ważny dla wzorca MVVM? Głównym celem wzorca MVVM – albo ściślej mówiąc jednym z głównych powodów wykorzystania wzorca MVVM – jest oddzielenie logiki prezentacyjnej od logiki interfejsu użytkownika tak, aby nasze modele widoków były luźno powiązane i łatwe do ponownego wykorzystania. Jeśli korzystamy z szablonu *DataTemplate* do wyświetlania danych udostępnianych przez model widoku w widoku, to uzyskamy więcej elastyczności w przyszłości przy zmienianiu i adaptowaniu tego widoku do nowych wymagań projektowych albo nowej technologii klienckiej, na przykład przy przejściu z WPF na Silverlight.

Ponieważ też szablon *DataTemplate* jest w zasadzie widokiem bez dodatkowego kodu, to możemy wiązać interfejs użytkownika bezpośrednio z modelem widoku i stosować szablon *DataTemplate*, który staje się wtedy widokiem dla tego modelu widoku.

Wróćmy do rysunku 6-3, który reprezentuje widok szczegółów klienta powiązany z modelem widoku *CustomerViewModel*. Ile razy prawdopodobnie skorzystamy z tego widoku? W rzeczywistej aplikacji CRM prawdopodobnie skorzystalibyśmy z tego widoku kilka razy, na przykład do przedstawienia wybranego klienta, do utworzenia nowego klienta, do wyświetlenia szczegółów klienta w widoku zamówienia, itd.

Można zobaczyć, że to podejście może być bardzo czasochłonne; musielibyśmy wiele razy zastosować te same reguły sprawdzania poprawności danych i pisać ten sam widok w XAML. Korzystając z szablonu *DataTemplate* możemy utworzyć prostą kontrolkę *UserControl* zarówno w Silverlight, jak i w WPF, powiązać ją z modelem widoku *CustomerViewModel*, a następnie dodać odwołanie do tego szablonu *DataTemplate* w każdym widoku, który go wymaga, oszczędzając dzięki temu czas i wielokrotnie wykorzystując ten sam kod.

Zdarzenia *WeakEvent* i komunikaty

Jeśli ktoś jest zaznajomiony z programowaniem zdarzeń w .NET, to być może już wie, czym jest zdarzenie i jak bolesne może być tworzenie i niszczenie zdarzeń przylączonych do określonego formularza.

W WPF i .NET Framework 4 firma Microsoft wprowadziła nowy typ zdarzenia o nazwie *WeakEvent*, które implementuje wzorzec projektowy słabego zdarzenia – mechanizm, który jest w stanie samodzielnie zarządzać procesem subskrypcji i anulowania zdarzeń. Te same wyniki możemy uzyskać w Silverlight, jak zobaczymy w dalszej części tego rozdziału.

Wzorzec *WeakEvent*

W staromodnym stylu zarządzania zdarzeniami cykl życia obiektu, który nasłuchuje zdarzenia (*odbiorcy*), może być inny od oczekiwanego, ponieważ jest sterowany cyklem życia obiektu wzbudzającego zdarzenie (*źródła*). W tym przypadku jedynym możliwym rozwiązaniem jest usunięcie odbiorcy ze źródła *deklaratywnie* przez odłączenie procedury obsługi zdarzenia od źródła.

W wersji .NET 4 możemy teraz korzystać z dwóch różnych obiektów do zaimplementowania wzorca *WeakEvent*: *WeakEventManager*, klasy, po której powinniśmy dziedziczyć, aby utworzyć niestandardowego menedżera zdarzeń oraz *IWeakEventListener*, interfejsu, który powinien być implementowany przez każdego odbiorcę słabego zdarzenia.

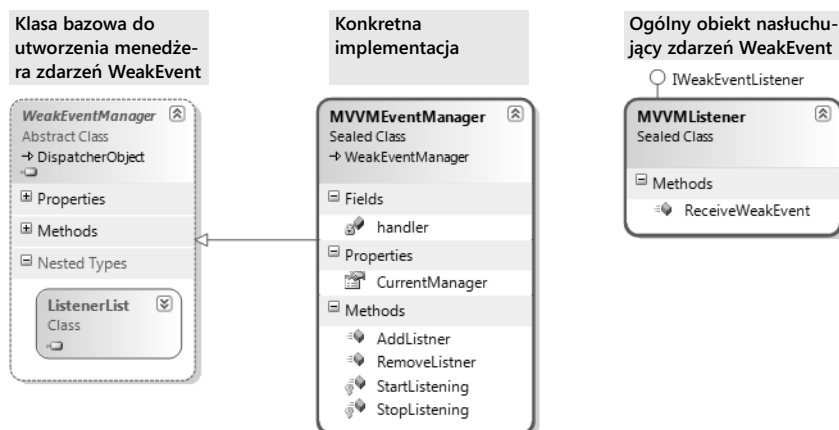
Rysunek 6-6 wyświetla podstawową implementację wzorca *WeakEvent* w .NET. Jak pokazano na rysunku 6-6, ta implementacja jest czasochłonna i wymaga dużo kodu, zwłaszcza gdy zdamy sobie sprawę, że ta implementacja powinna zostać zastosowana do każdego zdarzenia, którego chcemy nasłuchiwać z poziomu interfejsu użytkownika.

Wolelibyśmy raczej przejść na bardziej ogólne rozwiązanie (dostępne w kodzie źródłowym do pobrania dla tej książki) i utworzyć fabrykę, która jest w stanie subskrybować i usuwać odbiorców zdarzenia przy użyciu wyrażeń lambda i delegatów. Końcowy wynik wyglądałby następująco:

```
MyEventFactory.Listen<MyEventHandler>(  
    () =>  
    {
```

```
// tutaj zaimplementuj zdarzenie ...
});
```

Niestety to rozwiązanie napotyka inny problem .NET, ponieważ trudno jest subskrybować i anulować subskrypcję zdarzenia – nawet słabego – korzystając ze składni wyrażen lambda. Można więc w ogóle pominąć to rozwiązanie i rozważyć przyjęcie bardziej solidnego rozwiązania: wzorca komunikatu.



RYSUNEK 6-6 Implementacja wzorca *WeakEvent*

Wzorzec *EventAggregator*

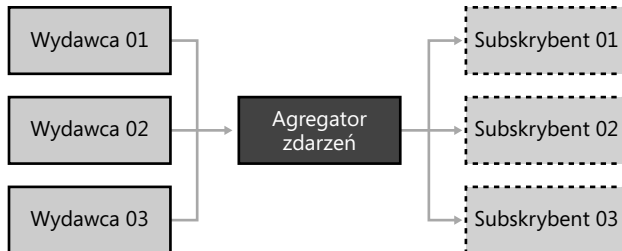
Zawsze z przyjemnością śledziłem wszelkie wskazówki zapewniane przez zespół patterns & practices w firmie Microsoft, zwłaszcza dotyczące wzorca Smart Client Software Factory (SCSF). Wiem, że został zaprojektowany dla Windows Forms, ale był świetny w swoich czasach (około roku 2004/2005), a w istocie wzorzec wydawca/subskrybent jest nadal dostępny i implementowany w Prism – złożonej platformie do tworzenia interfejsu użytkownika aplikacji w WPF i Silverlight.

Najlepszą częścią tego bloku aplikacyjnego był sposób, w jaki zarządzał on zdarzeniami w bezpieczny dla wątków sposób bez wpływania na wydajność lub interfejs użytkownika (trzeba pamiętać, że w Windows Forms, a także w WPF lub Silverlight, nie możemy nasłuchiwać w interfejsie użytkownika zdarzeń wzbudzanych przez inny wątek).

Wzorzec komunikatu (Message) jest zgodny z tą prostą logiką. Mamy *nadawcę* i *subskrybenta*, którzy obaj mogą wysyłać i odbierać komunikaty. Te komunikaty zawierają oczywiście pewne dane i mogą być wysyłane w dowolnym momencie w cyklu życia aplikacji. Jeśli się nad tym zastanowimy, jest to po prostu alternatywne rozwiązanie dla wzorca zdarzenia (Event).

W Prism i SCSF firma Microsoft wprowadziła obiekt *EventAggregator* będący wydajnym menedżerem zdarzeń, który może subskrybować dowolny typ zdarzenia

i wywoływać jakieś działanie za każdym razem, gdy to zdarzenie zostanie wzbudzone. Obiekt ten jest zgodny z wzorcem komunikatu. Jego jedynym wymaganiem jest to, że musimy implementować swoje zdarzenia korzystając z określonej klasy bazowej. Rysunek 6-7 pokazuje, jak działa *EventAggregator*.



RYСУNEK 6-7 Wzorec *EventAggregator*

EventAggregator będzie zawierał kod, który subskrybuje zdarzenie, i kod, który przekazuje zdarzenie subskrybentom. Implementacja powinna wyglądać podobnie, jak poniżej:

```
// wyślij "komunikat"
var event = EventAggregator.Get<MyEvent>();
event.Send("parameters");

// subskrybuj "komunikat"
var event = EventAggregator.Get<MyEvent>();
event.Subscribe(
    () =>
    {
        // zrób coś tutaj
    });
```

To rozwiązuje kilka problemów: nie musimy już martwić się utrzymywaniem zasobów przy życiu przez odbiorców, a także wprowadzamy standardowy wzorec komunikacyjny. Podobnie jak każdy inny wzorec, wzorec komunikatu ma kilka wymagań, które musimy spełnić w swoim projekcie, takich jak:

- Musimy gdzieś zarejestrować komunikat. Zwykle krok ten przeprowadzany jest w metodzie *inicjującej* w naszym interfejsie użytkownika.
- Po zarejestrowaniu komunikatu musimy utrzymywać przy życiu agregatora, który będzie powiadamiał odbiorców tego komunikatu. Jest to idealne wymaganie dla pojemnika odwrócenia sterowania. W tym przypadku zwykle wykorzystuję wzorec lokalizatora usług.
- Tak jak to robi Prism i inne platformy złożonego interfejsu użytkownika aplikacji, powinniśmy dodawać do naszych komunikatów ograniczenia sprawiające, że będą one łatwo wykrywalne i unikalne. Rozpoznanie komunikatu, który przekazuje ogólną wartość tekstową, może być trudne.

Okna dialogowe i modalne okna wyskakujące

Jednym z najbardziej skomplikowanych zadań, które musimy zrealizować podczas korzystania z wzorca MVVM, jest podtrzymywanie *dialogu* pomiędzy użytkownikiem a widokiem poprzez okno dialogowe lub modalne okno wyskakujące. To zadanie jest trudne nie z powodu jakichś trudności w tworzeniu widoku okna dialogowego w WPF lub Silverlight, ale ponieważ luźnie powiązany projekt wyróżniający wzorzec MVVM utrudnia realizację tego zadania utrzymując oddzielenie widoku od logiki prezentacyjnej.

Zanim zajmiemy się podejściami do tego zagadnienia, chcę pokazać różnicę pomiędzy oknem *MessageBox* wyświetlanym w WPF lub Silverlight a modalnym oknem wyskakującym. Zwykle okno dialogowe jest określonym widokiem, który pojawia się przed wszelkimi innymi widokami i uniemożliwia użytkownikom dalsze wykonywanie programu, dopóki modalny widok nie uzyska określonych danych lub potwierdzenia. Ten typ okna, do którego należą zarówno *MessageBox*, jak i *FolderDialogBox*, jest zwany *modalnym oknem dialogowym*. Inne okna, takie jak okno dialogowe *FindAndReplace*, które jest dostępne w większości edytorów, również pojawiają się przed innymi widokami, ale nie są modalne, ponieważ pozwalają nam edytować zawartość widoku znajdującego się pod spodem. Ten typ okna jest zwany *niemodalnym oknem dialogowym*.

WPF oferuje zestaw domyślnych okien dialogowych, które są często używane w aplikacjach. Niektóre wyświetlają użytkownikom komunikat i wymagają odpowiedzi, która jest zwykle wyborem spośród opcji „Tak”, „Nie” i „Anuluj”. Ten typ okna dialogowego jest znany jako *MessageBox*; jest to klasyczne, modalne okno dialogowe. Innymi typami są okna *OpenFileDialog*, *SaveFileDialog*, itd. Te typy okien dialogowych uniemożliwiają użytkownikom dalszą pracę z główną aplikacją, dopóki nie wykonają oni jakiegoś działania wewnątrz widoku modalnego. Po zakończeniu tego działania wykonywanie wraca do pierwotnej aplikacji, która odczytuje wyniki z widoku okna dialogowego.

Możemy też utworzyć niestandardowy widok, który może działać jak okno dialogowe, po prostu uruchamiając widok przy pomocy polecenia *ShowDialog* w języku C# lub Visual Basic .NET.

Modalny widok w MVVM

Przy korzystaniu z wzorca MVVM typowe podejście do wyświetlania okna dialogowego nie zadziała, ponieważ kod okna dialogowego funkcjonuje wewnątrz widoku, a więc znajduje się w logice interfejsu użytkownika. Jednak w MVVM taki kod powinien być wykonywany w logice prezentacyjnej, w kodzie modelu widoku.

Istnieją różne podejścia do uzyskania wyświetlenia okna dialogowego. Badając kod w dostępnych, otwartych pakietach narzędzi dla MVVM, takich jak MVVM Light, Prism i Caliburn, możemy odkryć, że każdy z nich wykorzystuje inne podejście, ale wszystkie one są skuteczne. Od nas samych zależy zrozumienie i wybranie, które będzie najlepsze dla nas i naszych potrzeb.

Usługa modalna

Pierwszy przykład, który tu zobaczymy, jest nazywany „podejściem usługowym”. Korzystając z tej metody tworzymy usługę odpowiedzialną za zarządzanie widokami okien dialogowych. Ogólnym pomysłem jest utworzenie jakiegoś kodu (w tym przypadku usługi), który będzie działał jako usługa *ModalService*, która jest w stanie tworzyć i niszczyć dowolny typ okna dialogowego oraz zwracać wynik okna dialogowego. Oto uproszczony pseudokod dla kontraktu usługi:

```
public interface IDialogService
{
    bool? ShowDialogMessage(string title, string message);
    bool? ShowDialogView(object view, object viewModel);
    void ShowMessage(string title, string message);
    void ShowView(object view, object viewModel);
}
```

Jednym z możliwych sposobów zaimplementowania tej usługi jest wykorzystanie platformy odwrócenia sterowania, która przedstawi usługę bezpośrednio w naszych modelach widoków. W tym momencie możemy utworzyć polecenie, które może wyświetlać okno *MessageBox* i czekać, aż zwrócona zostanie wartość *Boolean* (równa *true*, jeśli użytkownik potwierdzi jakieś działanie, a równa *false* w przypadku anulowania działania oraz równa *null*, jeśli użytkownik zamknie okno *MessageBox* bez odpowiedzi).

```
private IDialogService dialogService;

public ICommand SavePerson { get; private set; }

/// <summary>
/// Inicjuje polecenia.
/// </summary>
private void InitCommands()
{
    ShowMessage = new MVVM.Commanding.MvvmCommand(
        (sender) =>
        {
            var result =
                dialogService.ShowDialogMessage("Confirm?", "This is a confirm message.");
            if (result.HasValue && result.Value)
            {
                // zrób coś
            }
        },
        (sender) =>
        {
            return true;
        });
}
```

Lubię to podejście, ponieważ nie wymaga wiele wysiłku i nie zanieczyszcza widoku albo logiki interfejsu użytkownika kodem wymagającym do zaimplementowania okna

dialogowego. Oczywiście jest to tylko punkt początkowy – jeśli planujemy wykorzystanie tego podejścia, to powinniśmy pamiętać, że:

- Konieczna może być zwracana wartość innego typu.
- Wynik okna dialogowego może wywołać komunikat lub wzbudzić zdarzenie, które zaktualizuje nadrzędny model widoku/widok.
- Okno dialogowe może wymagać odbierania dodatkowych komunikatów i aktualizacji swojego stanu (na przykład okno dialogowe z paskiem postępu, które reaguje na zmiany długoterminowego działania).

Podejście wykorzystujące mediatora

WPF Disciples, grupa składająca się z programistów zainteresowanych technologią WPF i wzorcem MVVM proponuje inne ciekawe podejście.

Podejście wykorzystujące mediatora emuluje wzorec mediatora, który stara się zawrzeć logikę komunikacji pomiędzy dwoma lub więcej obiektami w zewnętrznej klasie o nazwie *Mediator*. Wzorec mediatora obejmuje dwa główne kroki: subskrypcję i powiadamianie. W przypadku subskrypcji dowolny obiekt może zapisać się do mediatora sprawiając, że będzie on świadomy danego subskrybenta. Mediator następnie nasłuchuje powiadomień wywoływanych przez te obiekty i reaguje w oparciu o pewną określoną logikę biznesową.

W MVVM możemy korzystać z elementu *Mediator* do osiągnięcia zadania komunikacyjnego; widoki zapisują się w mediatorze, a mediator następnie nasłuchuje i przekierowuje komunikaty wzbudzane przez zarejestrowane widoki. To podejście jest podobne do podejścia wykorzystującego zdarzenie, gdzie odbiorca rejestruje delegata do zdarzenia i oczekuje na powiadomienie, gdy zdarzenie zostanie wzbudzone. W platformie Prism wydanej przez zespół patterns & practices element *EventAggregator* wykorzystuje wzorec mediatora w połączeniu z elementami *Action<T>* i *WeakEvent* w celu realizacji tego zadania, jak pokazano w następującym kodzie:

```
// Agregator zdarzeń w PRISM
// subskrypcja
eventAggregator.Get<MyEvent>().Subscribe(
    (message) =>
    {
        // jakiś kod
    });
// wysyłanie powiadomienia
eventAggregator.Get<MyEvent>().SendMessage(MyMessage);
```

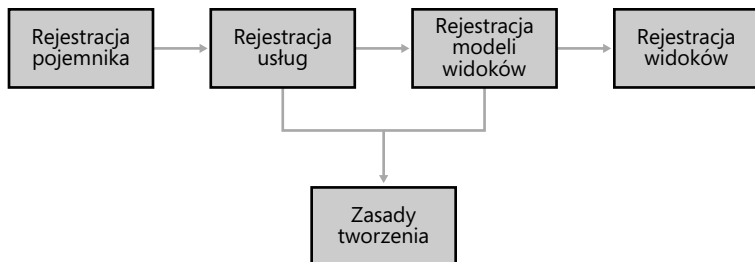
Możemy korzystać z wzorca mediatora w połączeniu z usługą dialogową, ponieważ razem realizują dwa różne zadania. Korzystając z wzorca mediatora nie musimy czekać na wynik okna dialogowego, ponieważ subskrybent zostanie powiadomiony, gdy okno dialogowe będzie gotowe; z kolei zwykła usługa dialogowa po prostu blokuje aplikację w oczekiwaniu na odpowiedź lub dane od użytkownika.

Odwrócenie sterowania w MVVM

Jeśli planujemy zbudowanie swoich narzędzi MVVM, to powinniśmy też zastanowić się, jak zamierzamy uruchamiać aplikację; jak chcemy ładować wspólne usługi i pojemniki; a także jak utrzymywać przy życiu triadę MVVM i inne obiekty związane z naszą aplikacją. Powinniśmy też rozważyć, jak zarządzać luźnym powiązaniem pomiędzy obiektami i ich zależnościami podczas tworzenia.

Na początku tego rozdziału podczas omawiania widoku dowiedzieliśmy się, jak wiązać kontekst danych dla widoku z modelem widoku wykorzystywanym w czasie projektowania w celu zapewnienia odpowiedniej elastyczności, tak aby projektanci mogli kontynuować proces tworzenia interfejsu użytkownika naszej aplikacji MVVM bez konieczności posiadania ukończonego modelu widoku. Jednak to podejście naprawia tylko część problemu, ponieważ nie widzieliśmy jeszcze, jak *wstrzykiwać* model widoku do widoku podczas działania programu. Nie widzieliśmy też, jak prawidłowo inicjować zależności w łańcuchu modelu widoku. Korzystając z odwrócenia sterowania (IoC) będziemy unikali zależności pomiędzy widokiem a modelem widoku, ponieważ będzie ona wstrzykiwana w trakcie działania aplikacji przez samą platformę IoC.

Jeśli korzystamy z pojemnika IoC, takiego jak Microsoft Unity, możemy wydzielić problem inicjowania aplikacji. Informacje na temat odwrócenia sterowania w rozdziale 2 przedstawiły, czym jest kontener IoC i jak powinien być używany. Rysunek 6-8 pokazuje normalny przepływ wykorzystywany przez aplikację MVVM inicjowaną przez kontener IoC.



RYСУNEK 6-8 Sekwencja inicjująca aplikacji MVVM

Normalny przebieg odpowiadałby następującej sekwencji:

- Aktywacja kontenera IoC
- Rejestracja wspólnych usług i narzędzi z wykorzystaniem zasad do definiowania cykli życia
- Rejestracja modeli widoków i względnych zależności w usługach nawigacyjnych i narzędziach
- Rejestracja widoków i ostateczne wstrzyknięcie odpowiadającego modelu widoku

Korzystając z tej sekwencji w pseudokodzie możemy teraz zgrać triadę MVVM w następujący sposób:

```
// pobierz model widoku
var vm = iocContainer.Resolve<IPersonsViewModel>();
vm.InitializeData();

// ustal widok
var view = iocContainer.Resolve<IPersonsView>();
// wiązanie danych
view.DataContext = vm;

// otwórz widok korzystając z usługi
var service = iocContainer.Resolve<INavigator>();
service.ShowView(view);
```

Niektóre z platform MVVM dostępnych już w środowisku .NET wykorzystują ten typ podejścia do ustalenia łańcucha zależności pomiędzy usługami, modelami widoków i widokami po prostu przez ustalenie określonego widoku, który wywołuje ten proces w pojemniku IoC.

Kod przykładowy

W dziale dotyczącym kodu przykładowego w tym rozdziale zobaczymy, jak rozwiązywać niektóre typowe problemy, które możemy napotkać podczas pracy nad aplikacją biznesową w WPF lub Silverlight, która zwykle będzie się składać z określonego zestawu kontrolki i składników interfejsu użytkownika. Na przykład potrzebna nam może być wiedza, jak zbudować kontrolkę wstążki w MVVM albo jak utworzyć podstawowe modele widoków, z których będziemy mogli korzystać w całej aplikacji, itd.

Pierwszym problemem, którym się tutaj zajmiemy, jest podłączenie kontrolki Microsoft Office Ribbon w WPF.

Microsoft Office Ribbon a MVVM

Jeśli planujemy pracę z kontrolką Microsoft Office Ribbon w aplikacji WPF, to pierwszym krokiem jest przejście do projektu CodePlex dla WPF (<http://wpf.codeplex.com>) i znalezienie części dotyczącej Office Ribbon. Będziemy musieli odwiedzić witrynę Office UI License pod podanym adresem URL i zaakceptować umowę licencyjną związaną z wykorzystaniem kontrolki WPF Ribbon w swojej aplikacji; następnie musimy pobrać najnowszą wersję kontrolki Ribbon i odwołać się do biblioteki Ribbon DLL w swojej aplikacji MVVM.



WIĘCEJ INFORMACJI Jeśli ktoś nie zna kontrolki Ribbon, to niezwykle przydatna będzie część witryny MSDN poświęcona wskazówkom i samouczkom na temat korzystania z kontrolki WPF Ribbon. Można ją znaleźć pod adresem <http://msdn.microsoft.com/en-us/library/cc872782.aspx>.

Tutaj skupiamy się na tym, jak można podłączyć kontrolkę Office Ribbon do naszej aplikacji MVVM, a nie na tym, jak budować kontrolkę Ribbon, co wykracza poza zakres tej książki.

UWAGA Z witryny WWW projektu Ribbon możemy też pobrać szablony dla Visual Studio w wersji beta pomagające nam zbudować określony widok lub aplikację WPF, która integruje się z kontrolką Ribbon.



Następujący kod pokazuje, jak możemy utworzyć prostą zakładkę kontrolki Ribbon, która zawiera grupę z pojedynczym przyciskiem.

```
<ribbon:RibbonGroup x:Name="Group1"
    Header="Group1">
    <ribbon:RibbonButton x:Name="Button1"
        LargeImageSource="Images\LargeIcon.png"
        Label="Button1" />
</ribbon:RibbonGroup>
```

Ponieważ kontrolka Ribbon w pełni implementuje wiązanie poleceń, to możemy sterować stanem i wykonywaniem kontrolki Ribbon poprzez dostęp do jej właściwości *Command* tak, jak robilibyśmy to z normalną kontrolką Button.

```
<ribbon:RibbonButton x:Name="Button1" Command="MyMVVMCommand"
```

W tym momencie kontrolka Ribbon będzie się zachowywać jak każda inna kontrolka interfejsu użytkownika powiązana z poleceniem MVVM implementującym interfejs *ICommand*.

Innym zalecanym podejściem jest wiązanie każdego elementu wstążki z określonym elementem *DataContext* i wiązanie każdej właściwości elementu *DataContext* z odpowiadającymi im właściwościami kontrolki Ribbon przy użyciu na przykład stylów WPF. Da nam to większą kontrolę nad zachowaniem wstążki.

```
<ribbon:RibbonButton DataContext="{x:Static data:WordModel.Cut}" />

<!--styl RibbonControl -->
<Style x:Key="RibbonControlStyle">
    <Setter Property="ribbon:RibbonControlService.Label"
        Value="{Binding Label}" />
    <Setter Property="ribbon:RibbonControlService.LargeImageSource"
        Value="{Binding LargeImage}" />
```

Oto zalecany sposób definiowania stylu dla przycisku Ribbon i wiązania odpowiedniego polecenia.

```
<!-- RibbonButton -->
<Style TargetType="{x:Type ribbon:RibbonButton}"
    BasedOn="{StaticResource RibbonControlStyle}">
    <Setter Property="Command" Value="{Binding Command}" />
</Style>
```

Korzystając z takiego kodu możemy zachować interfejs *ICommand* jako wzorzec implementacyjny wstążki i wiązać polecenie z kontrolką Ribbon korzystając z wiązania dwukierunkowego tak, aby stan wstążki był zawsze sprawdzany ze stanem interfejsu użytkownika.



UWAGA Pakiet WPF Ribbon SDK zawiera interesujący przykład, który ilustruje, jak wypełniać kontrolkę Ribbon z zachowaniem ograniczeń MVVM wykorzystując kolekcję poleceń powiązaną z całą kontrolką Ribbon.

Podsumowanie

Wzorzec Model View ViewModel jest oparty na wzorcu modelu prezentacyjnego wprowadzonym przez Martina Fowlera, ale ma określoną implementację związaną z WPF i Silverlight. Wzorzec ten obejmuje trzy składniki, które można określić jako widok, model i model widoku.

Podstawowym silnikiem aplikacji MVVM jest obiekt modelu widoku, który zawiera logikę prezentacyjną aplikacji i działa jako pośrednik wiążący model z widokiem. Standardowy model widoku powinien implementować interfejs *INotifyPropertyChanged*, żeby być kompatybilnym z silnikiem wiązania WPF i Silverlight, a także powinien implementować interfejs *IDataErrorInfo* tak, aby mógł powiadamiać interfejs użytkownika o błędach sprawdzania poprawności. Zwykle model widoku odpowiada za udostępnianie i zarządzanie swoimi poleceniami powiązanymi z widokiem.

Korzystając z silnika WPF możemy zbudować bardzo elastyczną aplikację MVVM – zwłaszcza jeśli skorzystamy z szablonu *DataTemplate* i stylów zapewnianych przez silnik interfejsu użytkownika. Aby powiadamiać i prowadzić interakcję z użytkownikami, możemy korzystać z okien dialogowych, widoków modalnych i okien wyskakujących; każde z nich powinno być używane zgodnie z określonymi wymaganiami projektowymi interfejsu użytkownika.

Ze względu na swój *zależnościowy* projekt wzorzec MVVM działa niezwykle dobrze w połączeniu z kontenerem IoC, takim jak Unity, który pozwala nam tworzyć proste przepływy zadań do zarządzania aplikacją.

Platformy i zestawy narzędzi MVVM

Chociaż wzorzec Model View ViewModel (MVVM) nie jest jeszcze tak sławnym wzorcem prezentacyjnym, jak Model View Presenter (MVP) i Model View Controller (MVC), to stanowi ewolucję tych wzorców i szybko zyskuje sławę wśród społeczności .NET. Choć istnieje od stosunkowo niedawna, to już pojawiło się wiele platform i zestawów narzędzi, które możemy wykorzystać do implementowania MVVM w technologiach Window Presentation Foundation (WPF), Silverlight lub Windows Phone 7.

Ten rozdział omawia niektóre z najczęściej używanych platform MVVM dostępnych w środowisku .NET. Zobaczmy, jak mogą one rozwiązać wiele spośród problemów omawianych w poprzednich rozdziałach, a związanych z logiką prezentacyjną w aplikacji MVVM – z których większość występuje ze względu na to, że WPF i Silverlight są dwiema różnymi technologiami. Ten rozdział zbada też pobieżnie technologię o nazwie „Prism”, na której rozwój zespół patterns & practices w firmie Microsoft poświęcił wiele czasu i wysiłku.

Platformy te nie tylko wyjaśniają, jak implementować wzorzec MVVM w WPF i Silverlight, ale też zapewniają zestaw narzędzi, dzięki którym możemy budować modułarne i bardzo funkcjonalne aplikacje.

Zestawy narzędzi dla MVVM

Kiedy szukam zestawu narzędzi dla MVVM, to zwykle przygotowuję listę wymagań, które dana platforma ma z łatwością obsługiwać, a następnie staram się ustalić, czy sprawdzana platforma spełni moje wymagania. Wzorzec MVVM w podstawowej implementacji ma ścisły zbiór zasadniczych wymagań, takich jak implementacja interfejsu *INotifyPropertyChanged*, sprawdzanie poprawności danych w interfejsie użytkownika i obsługę poleceń. Ponadto istnieje zbiór zaawansowanych funkcjonalności ogólnie związanych z interfejsem użytkownika aplikacji i obejmujących możliwość testowania widoków, komunikaty pomiędzy widokami, obsługę zdarzeń, itd.

Poprzedni rozdział pokazał, że nie jest łatwo zaimplementować polecenie MVVM, z którego moglibyśmy łatwo korzystać w ten sam sposób zarówno w WPF, jak i w Silverlight. Zwykle będzie to wymagać przepisania kodu, ponieważ te dwie technologie

obsługują interfejs *ICommand* w różny sposób. Nie jest też łatwo udostępniać model w modelu widoku bez konieczności pisania redundantnego kodu z powodu braku automatyzacji w klasach bazowych.

Ten rozdział zbada wszystkie te wymagania omawiając zbiór zestawów narzędzi MVVM, z których możemy korzystać do szybkiego rozwiązywania tych problemów.



UWAGA Trzy główne pakiety narzędzi MVVM, które zostaną przedstawione w dalszej części rozdziału, są udostępniane przez społeczność .NET. Można je pobrać i od razu zacząć używać w swoich projektach. Jednakże dostępność takich gotowych zestawów narzędzi nie powinna nas powstrzymywać przed zaimplementowaniem swojego własnego pakietu narzędzi. Warto też zauważyć, że te trzy nie są jedynymi dostępnymi zestawami narzędzi. Istnieją inne, mniej dojrzałe pakiety narzędzi na witrynie CodePlex i innych witrynach społeczności .NET, które można wypróbować i stosować. Te trzy przedstawione tutaj mają bogatą dokumentację i aktywne społeczności, co sprawia, że dobrze nadają się jako punkt startowy do ćwiczenia i opowywania wzorca MVVM.

Zestaw narzędzi MVVM Light Toolkit autorstwa Laurenta Bugniona

Laurent Bugnion jest świetnym programistą i laureatem nagrody Most Valuable Person (MVP) przyznanej przez firmę Microsoft za działania związane z technologią Silverlight. Trzy lub cztery lata temu Laurent napisał pakiet narzędzi MVVM dla WPF, który został rozwinięty również o obsługę Silverlight i Windows Phone 7. Ten pakiet narzędzi nosi obecnie nazwę MVVM Light Toolkit.

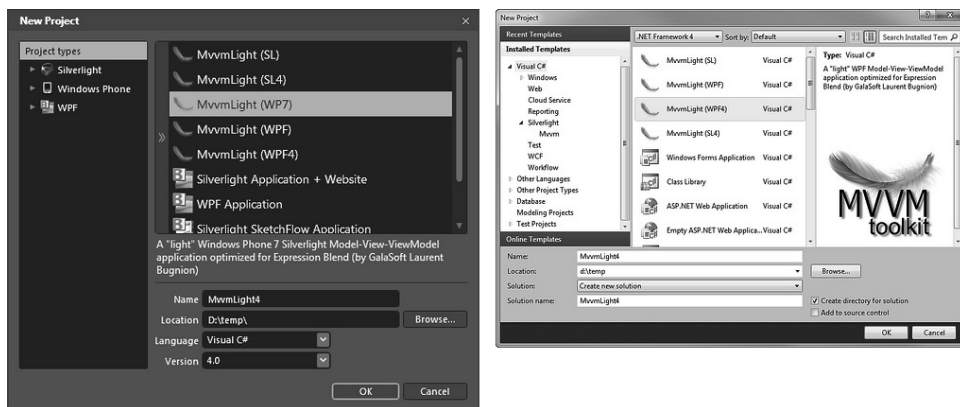
Według stanu na początek roku 2011 ten pakiet narzędzi jest obecnie dostępny w wersji 3. Ma pełną obsługę technologii WPF, Silverlight oraz Windows Phone 7. Można go pobrać pod adresem <http://www.galasoft.ch/mvvm/getstarted/>. MVVM Light Toolkit integruje się dobrze zarówno z Visual Studio 2010, jak i Microsoft Expression Blend.

Po zainstalowaniu tego zestawu narzędzi na swoim komputerze można zobaczyć, że instalacja dodała kilka rozszerzeń do programów Microsoft Visual Studio 2010 i Expression Blend, które zawierają nowe szablony projektów i fragmenty kodu (code-snippets) dla WPF, Silverlight i Windows Phone 7.



UWAGA W Visual Studio 2010 „code-snippet” jest specjalnym plikiem z rozszerzeniem .snippet, który współpracuje z technologią IntelliSense dając nam szybki i łatwy sposób wstawiania gotowych fragmentów kodu w swoich projektach.

Rysunek 7-1 pokazuje tę integrację w Visual Studio 2010 i Expression Blend.



RYSUNEK 7-1 Integracja pakietu MVVM Toolkit z Expression Blend i Visual Studio 2010

Ten zestaw narzędzi zapewnia następujące funkcje związane z pisaniem aplikacji MVVM:

- Nową klasę *ViewModelBase*, którą możemy stosować dla dowolnego obiektu modelu widoku;
- Klasę *Messenger*, która implementuje wzorzec wydawca/subskrybent, a także pełen zbiór składników komunikacyjnych, takich jak *NotificationMessage*, *DialogMessage*, *PropertyChangedMessage*, itd.;
- Elastyczną klasę *RelayCommand* (polecenia MVVM), która działa zarówno w WPF, jak i Silverlight;
- Pomocników i narzędzi dla WPF i Silverlight obsługujących integrację z wielowątkowym interfejsem użytkownika, integrację z Expression Blend, szablony elementów, szablony projektów w Visual Studio i wiele więcej.

MEFedMVVM

MEFedMVVM jest biblioteką służącą do budowania aplikacji wykorzystujących platformę Managed Extensibility Framework (MEF) przy użyciu Silverlight lub WPF; zapewnia zbiór klas bazowych i składników, poprzez które możemy implementować wzorzec MVVM z pomocą MEF.

UWAGA MEF jest biblioteką udostępnianą w podzespole System.ComponentModel.Composition, która zajmuje się problemem projektowania rozszerzalnych i podzielonych na odrębne składniki aplikacji. Sposób działania MEF widzieliśmy już w poprzednich rozdziałach, zwłaszcza podczas omawiania odwrócenia sterowania przy pomocy MEF i Unity.



MEFedMVVM jest projektem z otwartym kodem źródłowym utrzymywany na witrynie <http://www.codeplex.com> i dostępnym do pobrania pod adresem <http://mefedmvvm.codeplex.com/>. Obecnie ma pełne wsparcie zarówno dla WPF, jak i Silverlight 4.

Funkcją, która odróżnia tę platformę od innych platform MVVM, jest wzorzec atrybutu stosowany wobec każdej klasy. Ten wzorzec atrybutu transformuje te klasy w model widoku lub inny składnik dla wzorca MVVM. Na przykład, aby utworzyć klasę modelu widoku w MEFedMVVM, musimy po prostu zaimplementować następujący kod:

```
[ExportViewModel("MyViewModel")]
public class MEFViewModel
{
    // implementacja modelu widoku
}
```

W tym momencie element *ViewModelLocator* przyłączony do widoku odnajdzie ten model widoku. Będziemy w stanie deklarować model widoku w trybie edycji XAML po prostu korzystając z właściwości połączonych zapewnianych przez ten pakiet narzędzi, jak pokazano poniżej:

```
<UserControl
    <-- przestrzeń nazw XAML -->
    MEFed:ViewModelLocator.ViewModel = "MyViewModel">
```

Korzystając z tego podejścia MEFedMVVM zapewnia częściowy mechanizm dla wstrzykiwania zależności tak, że nasze modele widoków mogą deklarować zależne usługi i składniki i nie musimy się przejmować ich tworzeniem. Na przykład poniższy kod wykorzystuje MEF do wstrzykiwania usług do określonej klasy modelu widoku:

```
[ExportViewModel("MyViewModel")]
public class MEFViewModel
{
    [ImportingConstructor]
    public MEFViewModel(IMyService myService){
        // zapisz wystąpienie myService ...
    }
}
```

MEFedMVVM zapewnia zestaw gotowych narzędzi do implementowania wzorca MVVM. Oczywiście jest to lżejsza platforma od MVVM Light Toolkit i stosuje inne podejście, które jest bardziej zorientowane na wzorzec odwrócenia sterowania (IoC). MEFedMVVM zawiera następujące składniki:

- Wzorzec atrybutu do deklarowania klas modeli widoków
- Pełną integrację z MEF
- Wsparcie dla trybu projektowania w Visual Studio

Cinch autorstwa Sachy Barbera

Cinch jest jeszcze jedną platformą MVVM z otwartym kodem źródłowym, której autorem jest Sacha Barber posiadający tytuł MVP dla Visual C#. Píše on wiele artykułów na temat C# i WPF. Pakiet Cinch jest dostępny w witrynie CodePlex pod adresem <http://cinch.codeplex.com/> i jest zgodny zarówno z WPF, jak i Silverlight.

Cinch ma wiele funkcji, w tym:

- Elastyczne tworzenie nadających się do edycji obiektów modeli widoków, które zawierają obsługę sprawdzania poprawności i powiadamiania o błędach interfejsu użytkownika;
- Kompletny zestaw menedżerów zdarzeń *WeakEvent*, a także implementację wzorca mediatora;
- Pomocników dla wielowątkowości, które upraszczają interakcję pomiędzy interfejsem użytkownika a wywołaniami w innych wątkach;
- Obsługę różnych platform IoC oraz platformy MEF.

Platforma Cinch zasługuje na poświęcenie jej nieco czasu i uwagi. Jednym ze sposobów na śledzenie rozwoju tej platformy jest czytanie artykułów Sachy Barbera, które omawiają funkcje i różne wersje tego narzędzia. Niestety Cinch nie jest jeszcze zintegrowany z Visual Studio 2010, a pełne zrozumienie jego mechanizmów i struktury zajmuje sporą ilość czasu.

Pełen zestaw samouczków napisanych przez Sachę można znaleźć pod adresem <http://www.codeproject.com/KB/WPF/Cinch.aspx>.

Inne narzędzia dla MVVM i XAML

Wzorzec MVVM nie jest strasznie skomplikowany; skomplikowana jest wiedza wymagana przez technologię, z której zamierzamy skorzystać. Na przykład, jeśli planujemy utworzenie aplikacji MVVM przy użyciu WPF, to musimy mieć dogłębną wiedzę na temat działania WPF – a to jest czasochłonne. Gdy pracujemy z określoną technologią interfejsu użytkownika, taką jak WPF albo Silverlight, to musimy dokładnie poznać tę technologię, aby móc wykorzystać jej wszystkie możliwości.

Innym problemem, który możemy napotkać, gdy zaczniemy pracować z WPF lub Silverlight, jest proces budowania interfejsu użytkownika – jak on działa i jak możemy go optymalizować? W XAML możemy rozmieszczać w widoku dany zbiór kontrolki interfejsu użytkownika na wiele różnych sposobów. Na przykład możemy rozmieszczać kontrolki korzystając z obiektów *StackPanel*, *GridPanel*, itd.

W społeczności .NET znalazłem kilka bardzo przydatnych narzędzi z otwartym kodem źródłowym, które osobiście wykorzystuję w swojej codziennej pracy i które znacznie ułatwiają mi pracę. Są to narzędzia zapewniające dodatkowe mechanizmy IntelliSense w Visual Studio; narzędzia do przygotowywania zbioru klas bazowych

wymaganych przez wzorzec MVVM oraz narzędzia zapewniające kreatorów i narzędzia wymagane przez naszą aplikację MVVM.

Ten podrozdział nie obejmuje wszystkich dostępnych narzędzi zapewnianych przez społeczność .NET i projekty z otwartym źródłem wspomina jedynie o kilku z nich, które znam i widziałem w działaniu. Zachęcam wszystkich do samodzielnego badania społeczności tworzących projekty z otwartym kodem źródłowym w poszukiwaniu nowych – i być może lepszych – narzędzi niż te, o których tutaj wspominam.

Narzędzia Karla Shiffletta

Poznałem Karla niedawno w firmie Microsoft w Seattle na dorocznym spotkaniu zespołu patterns & practices. Jego dokonania na rzecz społeczności WPF są bezcenne; jest on zatrudniony w firmie Microsoft jako menedżer oprogramowania przy projekcie Prism prowadzonym przez zespół patterns & practices, który dokładniej omówię w dalszej części tego rozdziału. Karl stworzył jak dotąd trzy ważne projekty, które mogą znacznie ułatwić życie programisty WPF/SL. Tymi trzema projektami są:

XAML Power Toys

XAML Power Toys jest zbiorem rozszerzeń dla Microsoft Visual Studio 2008 i Visual Studio 2010, które wzbogacają możliwości projektantów WPF/SL i edytora XAML. Pakiet ten zapewnia:

- Zbiór kreatorów do wizualnego tworzenia układu widoku XAML
- Zbiór klas bazowych i kreatorów do tworzenia modeli widoków i obiektów poleceń
- Podobne funkcje dla Silverlight

XAML Power Toys można pobrać pod adresem <http://karlshifflett.wordpress.com/xaml-power-toys/>.

XAML Editor

XAML Editor jest rozbudowanym i przydatnym dodatkiem dla technologii IntelliSense w XAML, który wzbogaca możliwości IntelliSense wbudowane w Visual Studio dodając filtry i inne funkcje. Gdy piszemy kod XAML korzystając z tego dodatku, kontekstowy mechanizm IntelliSense jest bardziej czytelny, a interfejs użytkownika ma kilka dodatkowych i bardzo przydatnych funkcji.

XAML Editor można pobrać korzystając z menedżera rozszerzeń Visual Studio albo pod adresem <http://visualstudiogallery.msdn.microsoft.com/1a67eee3-fdd1-4745-b290-09d649d07ee0/>.

In the Box Tutorial (MVVM)

„In the Box” jest zbiorem samouczków wbudowanych w interfejs użytkownika Visual Studio zapewniających szkolenie na określone tematy, takie jak MVVM, poprzez czytanie dokumentacji i interakcję z odpowiadającym jej kodem w tym samym wystąpieniu Visual Studio, w którym uruchomiono samouczek. Karl Shifflett niedawno opublikował pierwszy samouczek, który wykorzystuje to podejście. Jest on dostępny pod adresem <http://karlshifflett.wordpress.com/2010/11/07/in-the-box-ndash-mvvm-training/>. Materiały do pobrania zawierają pełny zestaw samouczków i wskazówek w języku C#, które obejmują pisanie aplikacji MVVM w technologii WPF od początku do końca.

Radical autorstwa Maura Servientiego

Mauro Servienti jest włoskim laureatem nagrody Microsoft MVP dla Visual C# i moim przyjacielem. Często występuje na spotkaniach włoskiej społeczności .NET i promuje wzorzec MVVM.

Na jego blogu (<http://www.topics.it/>) można znaleźć mnóstwo przydatnych informacji na temat MVVM. Jeszcze bardziej pasjonujące jest to, jak podchodzi on do pewnych funkcji brakujących w MVVM tworząc przemyślane i przydatne rozwiązania.

W roku 2010 Mauro zdecydował się opublikować „zestaw narzędzi” (choć wydaje mi się, że jest to nawet coś więcej) w serwisie CodePlex. Nazwał ten zestaw narzędzi „Radical”. Można go pobrać pod adresem <http://radical.codeplex.com>.

Można by się zastanawiać, dlaczego zwracam uwagę na konkretny zestaw narzędzi C#, gdy można znaleźć setki podobnych w serwisie CodePlex. Odpowiedzią jest to, że Radical stosuje inne podejście; dlatego właśnie lubię korzystać z tego zestawu podczas swoich codziennych zadań. Radical zapewnia unikalny zestaw narzędzi szczególnie nakierowanych na pomoc w prawidłowej implementacji wzorca MVVM. Na przykład jeden zbiór narzędzi o nazwie Memento pozwala nam tworzyć obiekty z możliwościami wycofywania i ponawiania zmian. Poniższy wydruk pokazuje prostą implementację tej usługi:

```
IChangeTrackingServiceProvider provider = ChangeTrackingServiceProvider.GetCurrent();

provider.CreateTrackingService();
IList<Person> list = new EntityCollection<Person>();
Person p = new Person();
p.FirstName = "Mauro";
p.LastName = "Servienti";
list.Add( p );

IChangeTrackingService svc = provider.GetTrackingService();
if( svc.IsChanged ) {
    svc.RejectChanges();
}
```

Innym ciekawym narzędziem jest interfejs *IMonitor* z poleceniem *DelegateCommand* stanowiące kombinację szczególnej implementacji interfejsu *ICommand* (dla WPF i Silverlight) oraz wzorca obserwatora, który wykorzystuje zdarzenia *WeakEvent* do aktualizowania możliwości wykonywania polecenia.

```
var command = DelegateCommand.Create()  
    .OnCanExecute( o => true )  
    .OnExecute( o => { } )  
    .TriggerUsing( PropertyChangedObserver  
        .Monitor( this )  
        .HandleChangesOf( vm => vm.IsValid ) );
```

Ten zestaw narzędzi jest dobrze zaprojektowany i zapewnia wiele funkcji. Poznanie Radical jest warte zachodu, a Mauro zawsze jest chętny do zapewnienia wsparcia. Oto krótka lista funkcji dostępnych w pakiecie Radical:

- Metody rozszerzające i pomocnicy dla rozmaitych sytuacji, w tym dla LINQ, list, obiektów, i wiele więcej;
- Implementacja silnika brokera komunikatów (wydawca/subskrybent);
- Klasy bazowe dla jednostek domenowych;
- Obserwatorzy, którzy mogą monitorować stany obiektów;
- Menedżer wątków, który wykorzystuje wygodne podejście płynnego interfejsu;
- Obiekty sprawdzające poprawność danych i kontrakty dla kodu, które wykorzystują płynny interfejs;
- Mnóstwo rozszerzeń XAML, pomocników, zachowań, efektów dla interfejsu użytkownika, konwerterów, i wiele więcej.

Platformy dla złożonych interfejsów użytkownika

Jak widzieliśmy w tej książce, gdy budujemy aplikację WPF/Silverlight, implementowanie wzorca MVVM jest tylko jedną częścią tego procesu; musimy też implementować mechanizm zachowywania danych – sposób luźnego wiązania warstwy logiki biznesowej z resztą aplikacji oraz mechanizm orkiestracji interfejsu użytkownika.

Wzorzec MVVM definiuje metodologię oddzielającą interfejs użytkownika od prezentacji w przypadku aplikacji budowanych przy użyciu WPF/Silverlight lub Windows Phone 7, ale nie wyjaśnia, jak zarządzać interfejsem użytkownika, jak go składać, jak otwierać komunikację pomiędzy kilkoma widokami oraz jak realizować inne zadania związane z interfejsem użytkownika.

Termin „platforma dla złożonego interfejsu użytkownika”, wprowadzony i stosowany przez zespół patterns & practices w firmie Microsoft w ramach projektu Composite Application Block (CAB), obejmuje zestaw narzędzi, platform, wzorców i wskazówek,

które pomogą nam budować aplikacje przy użyciu luźno powiązanych składników, które rozwijają się niezależnie od reszty aplikacji. Blok CAB został opracowany i zoptymalizowany dla technologii Windows Form, natomiast Prism zaprojektowano dla WPF i Silverlight.

Projekty Prism stanowią znaczącą inwestycję ze strony firmy Microsoft w rozwój platformy dla złożonego interfejsu użytkownika. Jedną z dobrych alternatyw dla Prism jest Caliburn, prostsza, ale mająca mniejsze możliwości platforma do budowania aplikacji ze złożonym interfejsem użytkownika wykorzystujących WPF lub Silverlight.

Microsoft Prism

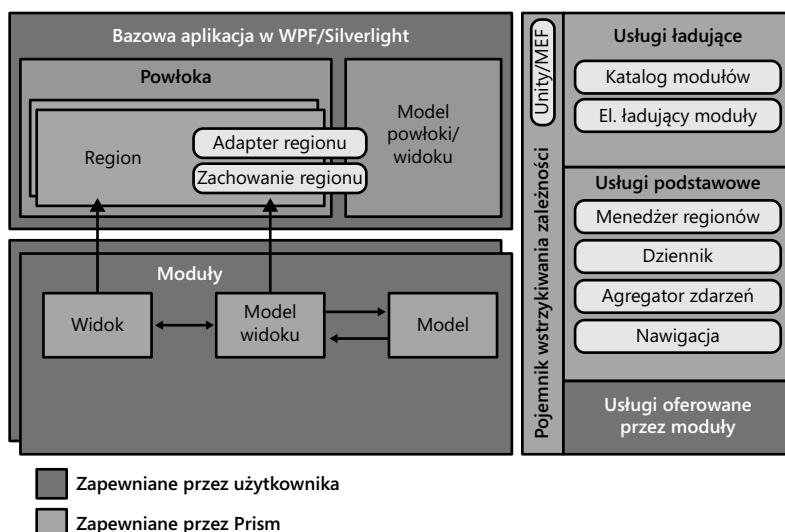
Prism jest rozbudowaną i dobrze zaprojektowaną od strony architektonicznej platformą do budowania złożonych interfejsów użytkownika. Najnowsza wersja (numer 4) działa z technologiami WPF, Silverlight i Windows Phone 7. Zapewnia zbiór wskazówek i kompletną platformę, dzięki której możemy budować modułarne i luźno powiązane aplikacje, oszczędzając wiele czasu i wysiłku. Cała infrastruktura jest dostępna jako dwie biblioteki .dll dostarczane w tym projekcie. Dodatkowy zasób stanowi aktywna społeczność CodePlex, która udostępnia dokumentację, samouczki i zapewnia fora dyskusyjne.

CodePlex utrzymuje też kod źródłowy i pliki binarne dla platformy Prism pod adresem <http://compositewpf.codeplex.com/>. Ten podrozdział zapewnia jedynie krótki przegląd działania tej skomplikowanej platformy, więc warto samemu pobrać platformę PRISM i dogłębnie ją zbadać, aby w pełni zapoznać się z jej możliwościami.

Oto podsumowanie opisu platformy Prism z witryny CodePlex (<http://compositewpf.codeplex.com/>):

Prism zapewnia porady mające pomóc nam w łatwiejszym projektowaniu oraz budowaniu bogatych, elastycznych i łatwych w utrzymaniu aplikacji Windows Presentation Foundation (WPF), aplikacji internetowych Silverlight i aplikacji Windows Phone 7. Wykorzystując wzorce projektowe, które stosują ważne architektoniczne zasady projektowe, takie jak podział interesów i luźne wiązanie, Prism pomaga nam projektować i budować aplikacje wykorzystujące luźno powiązane składniki, które mogą ewoluować niezależnie od siebie, ale które mogą być łatwo i sprawnie integrowane w całą aplikację. Takie aplikacje są często nazywane aplikacjami złożonymi.

Rysunek 7-2 pochodzący z witryny MSDN pokazuje strukturę złożonej aplikacji zbudowanej przy użyciu Prism.



RYSUNEK 7-2 Projekt architektoniczny aplikacji Prism

Rysunek 7-2 pokazuje, że aplikacja Prism składa się zwykle z zestawu składników, które najczęściej charakteryzują strukturę aplikacji złożonej lub modularnej. Składnikami tymi są:

- **Powłoka i regiony** Powłoka jest bazową aplikacją, do której ładowane są moduły, natomiast regiony są logicznymi pojemnikami używanymi do definiowania lokalizacji, do których będą ładowane widoki w interfejsie użytkownika.
- **Moduły i katalog modułów** Moduły są pakietami funkcji, które mogą być rozwijane i testowane niezależnie od siebie. Katalog jest odpowiedzialny za orkiestrację procesu ładowania tych modułów.
- **Agregator zdarzeń** Jest to konkretna implementacja wzorca wydawca/subskrybent, która pozwala na komunikację pomiędzy widokami.
- **IoC i usługi** W platformie Prism usługi są składnikami używanymi do gromadzenia funkcjonalności niezwiązanych z interfejsem użytkownika, a odwrócenie sterowania (IoC) jest używane do wstrzykiwania tych usług do składników aplikacji.

Prism jest nie tylko platformą dla aplikacji ze złożonym interfejsem użytkownika, ale zapewnia też wskazówki i samouczki dotyczące implementowania wzorca MVVM. Wielu programistów próbujących zacząć od platformy Prism zniechęca się ogromną ilością dokumentacji i przykładów zapewnianych w tej platformie aplikacyjnej, co sprawia, że uważają Prism za platformę zbyt skomplikowaną i niełatwą w użyciu. Jednakże nie jest to prawdą. Platforma Prism jest łatwa zarówno do nauczania się, jak i do zaimplementowania. Gdy już zaczniemy z nią pracować, to nie będziemy chcieli wrócić do bardziej tradycyjnych aplikacji wielowidokowych lub wielodokumentowych

(MDI), ponieważ odkryjemy, że brakować nam będzie narzędzi i funkcji zapewnianych przez Prism.

Calcium SDK

Calcium jest zestawem narzędzi z otwartym kodem źródłowym służącym do tworzenia złożonych aplikacji w WPF i Silverlight (wydanie alfa), który wykorzystuje bibliotekę Composite Application Library. Zapewnia wiele elementów potrzebnych do szybkiego budowania skomplikowanych aplikacji modularnych. Calcium składa się z aplikacji klienckiej i serwerowych usług WCF, które obsługują interakcję i komunikację pomiędzy klientami. Calcium zawiera mnóstwo gotowych modułów i usług oraz gotową do użycia infrastrukturę.

Narzędzia Calcium są utrzymywane na dedykowanej witrynie <http://calciumsdk.net/> i można je pobrać za darmo. Witryna Calcium zawiera łącza do przydatnych filmów wideo i samouczków.

Pakiet CalciumSDK zawiera następujące funkcje, które usprawniają platformę Prism:

- Zaawansowane zarządzanie modułami i menedżera modułów do włączania i wyłączania modułów w trakcie działania aplikacji.
- Szablony Visual Studio do szybkiego tworzenia projektów Calcium, w tym aplikacji klienckich, szablonów modułów MVVM i serwerowych projektów WCF (zarówno w C#, jak i VB.NET).
- Obsługę tematów z dołączonymi dwoma atrakcyjnymi tematami.
- Dwukierunkowe usługi komunikacyjne, które wykorzystują ten sam interfejs API do interakcji z użytkownikami z poziomu klienta lub serwera. Na przykład możemy kontaktować się z użytkownikiem z poziomu serwera powodując pojawianie się okien komunikatów na kliencie!
- Zaawansowaną obsługę poleceń z interfejsami do zawartości, które określają włączane polecenia i widoki.
- Adaptery regionów dla pasków narzędzi i menu.
- Logowanie po stronie klienta.
- Wstępnie zbudowane moduły, takie jak przeglądarka WWW, edytor tekstów, okno wynikowe, itd.
- Interfejs z zakładkami ze wskazaniem zmodyfikowanych plików (do wykorzystania w wielu modułach).
- Moduł User Affinity, który wspomaga tworzenie funkcji obsługujących współpracę pomiędzy użytkownikami aplikacji.
- System zarządzania zadaniami: cofnij, ponów, powtórz.

Caliburn

Ostatnią platformą omawianą w tej książce jest Caliburn – projekt z otwartym kodem źródłowym utrzymywany na witrynie CodePlex pod adresem <http://caliburn.codeplex.com/>. Platforma Caliburn została przedstawiona programistom WPF mniej więcej w tym samym czasie, kiedy firma Microsoft wydała pierwszą wersję Prism.

Początkowo, ze względu na brakujące funkcje w Prism, to Caliburn zyskał większą publiczność, ponieważ był pierwszą platformą do budowania aplikacji ze złożonym interfejsem przy pomocy WPF. Ostatnio Caliburn zaczął tracić zwolenników; w mojej opinii to dlatego, że Prism obecnie zapewnia więcej funkcji.

Dla tych, którzy nie potrzebują pełnego zestawu funkcji, Caliburn jest też dostarczany w wersji mikro zwanej Caliburn.Micro, dostępnej pod adresem <http://caliburnmicro.codeplex.com/>.

Caliburn nie wymaga, ani nawet nie zaleca, użycia określonego wzorca prezentacyjnego w WPF lub Silverlight; działa i zapewnia przykłady dla MVC, MVP i MVVM. Głównym zadaniem Caliburn jest uproszczenie tworzenia aplikacji ze złożonym interfejsem użytkownika i ułatwianie testowania warstwy interfejsu użytkownika i warstwy prezentacyjnej.

W Caliburn aplikacja jest sterowana przez prezentera, którym we wzorcu MVVM jest składnik modelu widoku. Model widoku jest skojarzony z odpowiadającym mu widokiem w XAML, a kontener IoC odpowiada za rozpoznawanie i kojarzenie tych obiektów.

Najnowsza wersja Caliburn współpracuje z MEF i wprowadza pojęcie „atrybutów dekoracyjnych” służących do ustalania zależności takich, jak już widzieliśmy w podrozdziale dotyczącym pakietu narzędzi MEFedMVVM.

Podsumowując Caliburn jest platformą dla złożonych interfejsów użytkownika, która jest bardzo zbliżona do Prism, ale jest projektem rozwijającym się nieco wolniej niż Prism.

Indeks

_ (podkreślenie), znak w pasku menu 12

A

Action<T> 179
adapter, wzorzec 27
Add, metoda 120
AddOrder, metoda 132
AddProduct, metoda 92
Address, jednostka 87
adnotacje 80
adnotacje danych
informacje 82
sprawdzanie poprawności jednostek
domenowych 80
ADO.NET 97
ADO.NET Entity Data Model 97
ADO.NET Entity Object Generator 97
ADO.NET Self-Tracking Entity 98
alarmy 15
Alexander, Christopher, wzorce
projektowe 26
aplikacje biznesowe 3
.NET Framework vii
reguły biznesowe 131
warstwa biznesowa 127, 138
warstwa interfejsu użytkownika
w MVVM 153
wzorce projektowe 25
aplikacje biznesowe w Silverlight, reguły
sprawdzania poprawności i warstwa
interfejsu użytkownika 129
aplikacje do wprowadzania danych 71
aplikacje korporacyjne
model domenowy 63
MVVM (Model View ViewModel) 1
platformy i zestawy narzędzi MVVM 183
warstwa biznesowa 127
warstwa DAL 93
warstwa interfejsu użytkownika
a MVVM 153
wzorce projektowe 25

aplikacje MVVM
mapowanie domeny 112
warstwa biznesowa 127, 138, 141, 147
warstwa DAL 93-94
aplikacje MVVM w WPF 188
aplikacje WPF, widok 155
aplikacje WPF/Silverlight, platformy dla
złożonego interfejsu użytkownika 191
AppFabric 137
Approval, proces zatwierdzania 92
ASP.NET MVC, wzorzec MVC 33, 35
atrapa interfejsu użytkownika 9
atrapa modelu widoku 156
atrapy, przykłady przy użyciu Sketchflow 10
atrybuty
atrybuty modelowania danych 80
atrybuty sprawdzania poprawności 80
atrybuty wyświetlania 80
InjectionConstructor, atrybut 49
sprawdzanie poprawności jednostek
domenowych 80
wzorzec atrybutu w klasach MVVM 187
Auto-Mapper 68

B

BaseLogger, klasa 46
bazowy obiekt domenowy 144
bazy danych
domena 71
jednostka domenowa 68, 71
warstwa DAL a procedury składowane 94
Behaviors SDK 7
bezpieczeństwo, warstwa DAL 95
biblioteki
Enterprise Library 138, 143, 146, 169
MEFedMVVM 185
biznes. *Zobacz* aplikacje biznesowe
BLL (Business Logic Layer) 131
AppFabric 137
transakcje biznesowe 147
zastosowanie 141

błędy sprawdzania poprawności 129

błędy, metody obsługi 16

budowniczy, wzorzec 27

Bugnion, Laurent 184

Build, polecenie 143

C

C#

MVC, wzorzec 33

niestandardowa logika ogólna a reguły
biznesowe 133

płynny interfejs 55

CAB (Composite Application Block),
Composite UI Framework 191

Calcium SDK 193-194

Caliburn 162, 176

CanAddOrder, metoda 132

CanExecute

kontekst 16

metoda 161, 165

przepływ zadań 136

Castle 121

Cinch 187

CommandBinding, kolekcja 166

CommandManager 166

Common CLR 113

Composite Application Block. *Zobacz* CAB
confORM 103

Contact, jednostka 87

CRM, aplikacje

aplikacje biznesowe 4

warstwa usługi biznesowej 143

CRM, domena 65, 117

CRM, model domenowy

informacje 83

kod przykładowy 84

CRM, warstwa dostępu do danych 115

IUnitOfWork, interfejs 115

mapowanie modelu domenowego za pomocą
EF 117

CRUD, operacje

podejście sterowane domeną 72

repozytorium 115

Customer, jednostka 85

cykl życia

UoW (jednostka pracy) 106

widoku 154

D

DAL (warstwa dostępu do danych) 93

baza danych a procedury składowane 94
informacje 93

kod przykładowy: warstwa dostępu do
danych CRM 115

mapowanie domeny za pomocą
NHibernate 121

O/RM 95

rozproszone warstwy danych 112

TDD 109

UoW (jednostka pracy) 104

wzorzec repozytorium 107

DataContext

IUnitOfWork, interfejs 115

określanie 157

DataSource, widok i MVVM 156

DataTemplate

dostosowywanie 155

jednostka domenowa 161

kontrolki Tooltip 15

MVVM 173

WPF a Silverlight 171

wzorzec PM 41

XAML 19

DB2, EF 97

DBMS (system zarządzania bazami danych) 94

DDD (projektowanie sterowane domeną) 63

analizowanie domeny CRM 65

DAL 94

mapowanie modelu domenowego 117

terminologia 64

tworzenie obiektów 74

dekoracje Visual Studio 77

dekorator, wzorzec 27

DelegateCommand 189

Design Patterns: Elements of Reusable
Object-Oriented Software 26

Dijkstra, Edsger W. 20

Dispatcher 185

DomainContext, klasa 115

DomainObject, klasa 84

DomainObject, typ nadrzędny warstwy 85
domena 70

definicja 64
 izolowanie od reszty aplikacji 64
 mapowanie przy użyciu NHibernate 121
 podejście sterowane bazą danych 71
 podejście sterowane domeną 72
 skrypt transakcyjny 70
 domena najpierw 110
 domena zamówienia 90
 domyślne okna dialogowe 176
 DSL (język specyficzny dla domeny) 54
 DTO (obiekt transferu danych) 67
 dwukierunkowe usługi komunikacyjne 194

E

Emit Mapper 68
 Employee, jednostka 85
 Enterprise Library 138, 143, 146, 169
 Entity Framework
 informacje 70, 97
 jednostki pośredniczące 119
 mapowanie modelu domenowego 117
 podejście sterowane domeną 72
 warstwa DAL 93
 Error, właściwość 168
 etykiety tekstowe, przyciski paska narzędzi 13
 EventAggregator 179
 EventAggregator, wzorzec 174
 EVIL 140
 Execute, metoda 161
 Expression Blend 7, 156, 158, 184
 Expression SDK 156
 Expression Studio 7
 Expression Suite 10
 Extensible Application Markup Language.
 Zobacz XAML (Extensible Application
 Markup Language)

F

fabryka abstrakcyjna, wzorzec 27
 fabryka abstrakcyjna/metoda wytwórcza 75
 fasada sprawdzania poprawności,
 wzorzec 144
 fasada, klasy usługi 151
 fasada, wzorzec 27, 132
 faza rozwoju, aplikacje biznesowe 4

FindAndReplace, okno dialogowe 176
 Fluent Workflow Engine 147
 FluentNHibernate 103, 110, 121
 wtyczka 123
 FormatCommand, właściwość 165
 Fowler, Martin 21, 31

G

Genome 103
 GOF, wzorce 30, 44

H

Hibernate 101
 historie użytkowników
 domena CRM 65
 kod przykładowy: model domenowy CRM 84
 HttpContext, obiekt 106

I

ICommand, interfejs
 informacje 41
 kontrolka wstążki 180
 metody 161
 model 159
 pakiety narzędzi MVVM 183
 WPF i Silverlight 161
 ICommand, właściwość 161, 165
 ID, właściwość 68
 IDataErrorInfo, interfejs 16, 168
 identyfikowanie transakcji biznesowej 106
 IErrorInfo 94
 Iesi.Collection.dll 121
 if, instrukcje 80
 if/else, instrukcje 129
 ikony 12
 IMonitor 189
 inicjowanie 179
 inicjująca metoda 176
 InjectionConstructor, atrybut 49
 INotifyPropertyChanged, interfejs
 informacje 166
 model widoku 3, 160
 MVVM, wzorzec 183
 obiekty POJO 94
 system powiadamiania 41
 InRule 140

interesy 19

interfejs równoległy do sprawdzania
poprawności określonej jednostki
domenowej 78

interfejs użytkownika aplikacji
biznesowej 10

ogólny styl i rozważania na temat
kontrolerek 18

pasek menu 12

pasek narzędzi 13

pasek wstążki 16

powiadomienia i alarmy 15

Tooltip 13

interfejsy

ICommand, interfejs 41, 159, 161, 180, 183

IDataErrorInfo, interfejs 16, 168

INotifyPropertyChanged, interfejs 3, 41, 160,
166, 183

interfejs równoległy 78

interfejsy WWW: aplikacje biznesowe 4

IRepository, interfejs 120

ISessionFactory, interfejs 115

IUnitOfWork, interfejs 115

IUnityContainer, interfejs 49

IWeakEventListener, interfejs 174

płynny interfejs 54, 140

interfejsy WWW, aplikacje biznesowe 4

IoC, wzorzec 45

Microsoft Unity 49

wstrzykiwanie zależności 49

IQueryable, kolekcja 55

IRepository, interfejs 120

IRepository, klasy 108

ISession

mapowanie domeny za pomocą

NHibernate 122

obiekt 125

ISessionFactory, interfejs 115

IsNew, właściwość 72

IsValid, właściwość 144

Item, właściwość 168

ITransaction, obiekt 107

IUnitOfWork, interfejs 115

IUnityContainer, interfejs 49

IWeakEventListener, interfejs 174

izolacja, O/RM 96

J

jednostka domenowa

DataTemplate 161

informacje 67

INotifyPropertyChanged, interfejs 160

klasy bazowe 191

model 159

jednostka domenowa Order, sekwencja
przepływu zadań 71

jednostki

informacje 64

obiekty DTO 68

repozytoria 107

jednostki domenowe

Enterprise Library 138

model i MVVM 154

obiekty DTO 68

ograniczenia schematu bazy danych 144

warstwa DAL 93

jednostki domenowe POCO 118

jednostki POCO 119

języki programowania zorientowane
aspektowo 20

języki programowania zorientowane
obiektoowo 20

K

kategorie, .NET framework 80

klasy

BaseLogger, klasa 46

DomainContext, klasa 115

DomainObject, klasa 84

IRepository 108

klasy bazowe dla jednostki domenowej 191

Messenger, klasa 185

modelu 93

modelu widoku 187

MVVM 187

pośredniczące 118

Repository, klasa 120

usługi fasadowej 151

Validator, klasa 144

ViewModelBase, klasa 185

ViewModelValidator, klasa 169

warstwa biznesowa 127

WeakEventManager, klasa 174

wzorce behawioralne 28
 wzorce kreatywne 26
 wzorce strukturalne 27
 XAML Power Toys 188
 klasy modelu 93
 klasy pośredniczące 118
 klasyfikacja architektonicznych wzorców projektowych 26
 klasyfikowanie wzorców projektowych 26
 wzorce behawioralne 28
 wzorce kreatywne 26
 wzorce strukturalne 27
 kod przykładowy
 model domenowy CRM 84
 warstwa dostępu do danych CRM 115
 warstwa usługi biznesowej 143
 wstążka Microsoft Office i MVVM 180
 kompozyt, wzorzec 27
 komunikacja, Mediator 177
 komunikaty
 komunikaty o błędach 129
 MVVM i zdarzenia WeakEvent 173
 ograniczenia 176
 komunikaty o błędach, błędy sprawdzania poprawności 129
 kontekst 65
 kontekst biznesowy, model domenowy 92
 kontekst domenowy 84
 kontekst osoby 84
 kontrolki
 Silverlight kontra WPF 6
 Tooltip 15
L
 LINQ
 NHibernate 101
 składnia 55, 97, 103
 LINQtoSQL 56
 Listbox 171
 logika biznesowa
 BLL 142
 jednostka domenowa 68
 testowanie 94
 warstwa biznesowa 127
 logika interfejsu użytkownika, oddzielenie od deklaratywnych znaczników interfejsu 153

logika prezentacyjna, modeli widoku i MVVM 154
 LogWriter, refaktoring 47
 lokalizator usług, wzorzec 48

Ł

łańcuch zobowiązań, wzorzec 28
 łączenie metod 55

M

Managed Extensibility Framework.
 Zobacz MEF
 mapowanie
 mapowanie domen 112
 pliki 121
 słownik 95
 testowanie 109
 mapowanie automatyczne 123
 MDI 12
 mediator, wzorzec 28
 MEF (Managed Extensibility Framework)
 Cinch 187
 informacje 52
 MEFedMVVM, biblioteka 185
 Memento 189
 memento, wzorzec 28
 menedżer wątków 191
 Message, wzorzec komunikatu 175
 MessageBox 176
 Messenger, klasa 185
 metoda inicjująca 176
 metoda konstruktora 74
 metoda szablonowa, wzorzec 28
 metoda wytwórcza, wzorzec 27
 metody rozszerzające 191
 metody skrótowe 132
 metody. Zobacz także polecenia
 Add, metoda 120
 AddOrder, metoda 132
 AddProduct, metoda 92
 CanAddOrder, metoda 132
 CanExecute, metoda 161, 165
 Execute, metoda 161
 inicjująca metoda 176
 jednostki Employee i Customer 85
 łączenie w łańcuchu 55

metody (cd.)

- metoda inicjująca 176
- metoda konstruktora 74
- metody rozszerzające 191
- metody skrótowe 132
- obsługa błędów 16
- OnCanExecuteChanged(), metoda 166
- OnPropertyChanged(), metoda 161
- warstwa BLL 142
- Microsoft Dynamics 11
- Microsoft Enterprise Library 138
- Microsoft Entity Framework 97
- Microsoft Expression Blend 7, 156, 158, 184
- Microsoft Expression SDK 156
- Microsoft Expression Studio 7
- Microsoft Expression Suite 10
- Microsoft Service Locator 51
- Microsoft SketchFlow 9
- Microsoft Unity
 - informacje 49
 - Microsoft Unity 49
 - pojemnik IoC 179
 - porównanie z MEF 53
 - Service Locator 50
- Microsoft.Practices.EnterpriseLibrary.
 - Validation.dll 143
- modalne okna wyskakujące 176
- modalne okno dialogowe 176
- model
 - definicja 1, 64
 - MVC, wzorzec 32
 - MVP, wzorzec 35
 - MVVM 154, 159
 - PM, wzorzec 40
 - wzorce projektowe interfejsu użytkownika 30
- model domenowy 63
 - DDD (projektowanie sterowane domeną) 63
 - jak tworzyć obiekt w DDD 74
 - jednostka domenowa a obiekt DTO 67
 - kod przykładowy: model domenowy CRM 84
 - mapowanie przy użyciu Entity
 - Framework 117
 - NHibernate 101
 - obiekt POCO i O/RM 68
 - podejścia do projektowania domeny 70
 - pośrednik 95

- sprawdzanie poprawności jednostek
 - domenowych 78
- testy jednostkowe 83
- model domenowy zgodny z obiektami
 - POCO 101
- Model View ViewModel. *Zobacz* MVVM
- model widoku 166
 - atrapa modelu widoku 156
 - DataTemplate i wzorzec PM 41
 - definicja 1
 - ICommand, właściwość 161
 - IDataErrorInfo, interfejs 168
 - INotifyPropertyChanged, interfejs 166
 - MVVM 44, 187
 - MVVM a jednostki domenowe 154
 - platforma IoC 176
 - ponowne wykorzystywanie w różnych
 - widokach 153
 - sprawdzanie poprawności jednostki
 - domenowej 79
 - transakcje biznesowe 107
 - udostępnianie modelu 160
 - udostępnianie właściwości 160
 - warstwa biznesowa 127
 - wzorce projektowe interfejsu użytkownika 30
- model widoku w czasie projektowania 159
- moduły. *Zobacz także* warstwy
 - podział interesów 24
- most, wzorzec 27
- możliwość utrzymywania, O/RM 96
- MSTest 60
- Multiple Document Interface (MDI) 12
- MVC (Model View Controller), wzorzec
 - projektowy interfejsu użytkownika 30
- MVC, wzorce 31
- MVP, prezenter nadzorujący 39
- MVP, wzorzec 30, 35
- MVVM (Model View ViewModel) 1, 153, 183
 - aplikacje biznesowe 3
 - DataTemplate w WPF i Silverlight 171
 - informacje vii, 1
 - interfejs ICommand w WPF i Silverlight 161
 - interfejs użytkownika aplikacji
 - biznesowej 10
 - jako wzorzec projektowy 25
 - języki DSL 54

kod przykładowy: wstążka Microsoft Office
 i MVVM 180
 model 159
 model widoku 166
 MVVM, wzorzec 154
 narzędzia do budowania interfejsu
 użytkownika 6
 odwrócenie sterowania 179
 okna dialogowe i modalne okna
 wyskakujące 176
 pakiety narzędzi 183
 platformy dla złożonych interfejsów
 użytkownika 191
 PM, wzorzec 40
 Silverlight kontra WPF (Windows
 Presentation Foundation) 4
 SoC 19
 tworzenie obiektów w DDD 75
 widok 155
 XAML 187
 zdarzenia WeakEvent i komunikaty 173
 MVVM Light 176, 184
 MVVM, wzorzec
 informacje 154
 model widoku 30
 PM, wzorzec 40
 TDD 59
 widok 155
 WPF 31
 MySQL, podejście sterowane domeną 72

N

narzędzia do budowania interfejsu
 użytkownika 6
 Expression Blend 7
 SketchFlow 9
 nawigacja w domenie, O/RM 96
 nawyki związane z warstwą BLL 142
 .NET
 aplikacje biznesowe
 CLR 4
 IDataErrorInfo, interfejs 168
 INotifyPropertyChanged, interfejs 166
 InRule 140
 język zapytań LINQ 98
 kategorie 80

O/RM 103, 126
 programowanie zdarzeń 173
 rozproszone warstwy danych 112
 silnik reguł i silnik reguł biznesowych 140
 typy ogólne 108
 WF 4 129
 NHibernate
 informacje 101
 mapowanie domeny 121
 O/RM 70, 72
 TDD 110
 warstwa DAL 93
 NHibernate Validation Framework 82
 niemodalne okno dialogowe 176
 niestatyczne obiekty 74
 niezależne widoki 30
 niezależność od sposobu zachowywania
 danych 68, 70
 NUnit 61

O

O/RM (Object/Relational Mapper) 95
 .NET 103
 FM 97
 NHibernate 101
 obiekt POCO 68
 obiekt biznesowy, jednostka domenowa 68
 obiekt domenowy. *Zobacz* jednostka
 domenowa
 obiekt sesji 101
 obiekt wartości 65, 68
 obiekty
 cykl życia widoku 154
 DTO (obiekt transferu danych) 67
 HttpContext, obiekt 106
 ISession, obiekt 125
 ITransaction, obiekt 107
 model 159
 niestatyczne obiekty 74
 obiekt jednostki domenowej 160
 obiekt sesji 101
 obiekt wartości 65
 obiekty DSL 56
 obiekty modelu widoku 187
 obiekty POCO 68, 94, 110
 ObjectContext, obiekt 125

obiekty (*cd.*)

- sprawdzanie poprawności 169
- tworzenie nowego obiektu na użytek aplikacji 76
- tworzenie w DDD 74
- Unity 54
- wzorce behawioralne 28
- wzorce kreacyjne 26
- wzorce strukturalne 27
- obiekty POCO
 - budowa aplikacji zaczynając od domeny 110
 - O/RM 68
 - warstwa DAL 94
- obiekty sprawdzające poprawność,
 - Radical 191
- Object Mapper. *Zobacz* O/RM
- ObjectContext 119
- ObjectContext, obiekt 125
- obrazowanie szablonu 15
- obserwator, wzorzec
 - definicja 28
 - IMonitor z DelegateCommand 189
- odbiorca 173
- odwiedzający, wzorzec 28
- odwrócenie sterowania w MVVM 179
- ograniczenia
 - fabryki 77
 - jednostka domenowa 68
 - komunikatów 176
- ograniczenia schematu bazy danych 144
- okienko nawigacyjne 11
- okna dialogowe
 - informacje 16
 - MVVM 176
- OnCanExecuteChanged(), metoda 166
- onComplete, zdarzenie 147
- OnPropertyChanged(), metoda 161
- open source
 - Active Record 140
 - Calcium SDK 193
 - Cinch 187
 - MEFedMVVM, biblioteka 185
 - NHibernate 102
- OpenFileDialog 176
- operacje masowe, O/RM 96
- operacje, kontrolki i wskazówki 15

- opóźnione ładowanie 125

- Oracle, EF 97

- Order, proces 90

- OrderLine 91

- OriginalText, właściwość 166

- osadzone reguły sprawdzania poprawności 78

P

- paradygmaty programowania 20

- pasek menu 12

- pasek narzędzi 13

- pasek wstążki 16

- pasywny MVC, wzorzec 32

- Patterns of Enterprise Application Architecture 26

- pełnomocnik, wzorzec 27

- Person, domena 89

- platforma odwrócenia sterowania Cinch 187

- podjęcie usługowe 176

- platformy dla złożonych interfejsów użytkownika 191

- Calcium SDK 194

- Prism 191

- plug-and-play 74

- płynna składnia 147

- płynny interfejs 54

- sprawdzanie poprawności 191

- warunki 140

- PM (Presentation Model), wzorzec projektowy interfejsu użytkownika 30

- PM, wzorzec 40

- POCO, pojęcia

- Entity Framework 99

- O/RM 126

- podjęcie mediatora: widok modalny w MVVM 177

- podjęcie usługowe 176

- podkreślenie (⏏), znak w pasku menu 12

- podział interesów. *Zobacz* SoC

- pojemnik odwrócenia sterowania 179

- pojemnik wstrzykiwania zależności, Unity 49

- pole tekstowe WPF 170

- polecenia

- Silverlight 162

- wiązanie 180

polecenia skierowane 6
 polecenia. *Zobacz także* metody
 Build, polecenie 143
 pasek wstążki 18
 polecenia paska narzędzi 13
 polecenia skierowane 6
 Save(), polecenie 72
 ShowDialog, polecenie 176
 polecenie, wzorzec 28
 pomocnicy dla wątków 187
 pośrednik, model domenowy 95
 powiadomienia 15
 powiadomienia o zmianach, wzorzec MVP 36
 poziomy 21
 prezydent nadzorujący, wzorzec 39
 prezydent, wzorzec MVP 36
 PrimaryKey 84
 Prism 162, 176, 191
 proceduralne języki programowania 20
 procedury składowane, baza danych i warstwa
 DAL 94
 Product 92
 Product, jednostka 90
 projekt zorientowany na usługi 20
 projektanci WPF/Silverlight 188
 projektowanie kontekstowe 20
 projektowanie przez usługę 131
 projektowanie sterowane domeną. *Zobacz*
 DDD
 PropertyChanged, zdarzenie 3, 166
 prototyp, wzorzec 27
 przeglądarki, wzorzec MVC 34
 przepływ zadań według projektu 131
 przepływy zadań Windows, AppFabric 137
 przestrzenie nazw
 DataAnnotations, przestrzeń nazw 80
 dla WPF i Silverlight 156
 System.ComponentModel, przestrzeń
 nazw 128
 przykładowa aplikacja WPF 107

R

Radical 189
 Reade, Chris 20
 Reenskaug, Trygve 31
 refaktoring, LogWriter 47

reguły biznesowe
 a reguły sprawdzania poprawności 128
 a silnik reguł 140
 poprzez przepływ zadań i WF 4.0 133
 poprzez usługę 131
 reguły sprawdzania poprawności a reguły
 biznesowe 128
 rejestrowanie po stronie klienta i serwera 194
 rekord aktywny 70, 140
 RelayCommand 185
 Repository, klasa 120
 repozytorium
 IUnitOfWork 115
 mapowanie domeny za pomocą
 NHibernate 125
 repozytorium, wzorzec 107
 RIA Service
 DTO 79
 rozproszona warstwa danych 112
 Ribbon, kontrolka 13, 180
 rozproszone warstwy danych, budowanie za
 pomocą RIA i WCF 112
 rozszerzalność 64
 rozszerzenia, MEF 54

S

Save(), polecenie 72
 SaveFileDialog 176
 schemat kolorów 19
 SCSF (Smart Client Software Factory) 174
 sekwencja przepływu zadań, jednostka
 domenowa Order 71
 Service Locator 50-51
 sesja domenowa 123
 Shifflett, Karl 188
 ShowDialog, polecenie 176
 silnik brokera komunikatów 191
 silnik przepływów zadań 146
 silnik reguł i reguły biznesowe 140
 Silverlight
 Calcium SDK 193
 Cinch 187
 DataTemplate 171
 Dispatcher 185
 ICommand, interfejs 161
 informacje vii

Silverlight (*cd.*)

- a WPF (Windows Presentation Foundation) 4
- InRule 141
- Laurent Bugnion 184
- MEF 52
- MessageBox 176
- MVC, wzorzec 35
- MVP, prezenter nadzorujący 39
- MVP, wzorzec 39
- PM, wzorzec 40-41
- polecenia 162
- Prism 191
- przestrzeń nazw 156
- rozproszona warstwa danych 113
- UserControl 173
- wstrzykiwanie zależności, wzorzec 45
- XAML 187
- Silverlight CLR 114
- singleton, wzorzec 27
- SketchFlow 9
- składnia lambda 147, 174
- skrótowe kombinacje klawiszy 12
- skrypt transakcyjny 70
- Smart Client Software Factory (SCSF) 174
- SOA, podejście 115
- SOAP, protokół, rozproszona warstwa danych 113
- SoC (podział interesów) 19
 - warstwa DAL 93
 - wzorce projektowe interfejsu użytkownika 30
- sprawdzanie poprawności
 - jednostka domenowa 68
 - obiektów 169
 - pakiety narzędzi firm trzecich 138
 - sprawdzanie poprawności jednostek domenowych 78
 - testy jednostkowe modelu domenowego 83
 - warstwa usługi biznesowej 143
- sprawdzanie poprawności danych 143
- sprawdzanie poprawności jednostek domenowych 78
 - dostępne platformy sprawdzania poprawności 82
 - klasyczne sprawdzanie poprawności 78
 - korzystanie z atrybutów i adnotacji danych 80

- sprawdzanie wykonywania polecenia 189

SQL

- Entity Framework 99
- NHibernate 101
- WorkflowApplication 138
- SQL Server
 - Entity Framework 97
 - języki DSL 54
- stan i zachowanie 72
- stan, wzorzec 28
- strategia, wzorzec 28
- struktura, aplikacji biznesowych 4
- style
 - Silverlight kontra WPF 6
 - XAML 19
- Subsonic 103
- switch, instrukcje 129
- system zarządzania bazami danych (DBMS) 94
- System.ComponentModel, przestrzeń nazw 128
- szablon danych
 - Expression Blend 8
 - Silverlight 6
- szablon sprawdzania poprawności 170
- szablony
 - obrazowanie szablonu 15
 - Silverlight kontra WPF 6
 - szablon danych w Expression Blend 8
 - szablon sprawdzania poprawności 170
 - szablony Visual Studio 193
- środowisko zintegrowane Visual Studio, wzorzec PM i MVVM 43

T

T-SQL

- informacje 54
- kod 95, 126
- T4, generator kodu 99
- TDD (programowanie sterowane testami) 59
 - MVC, wzorzec 35
 - przykład 60
 - testy jednostkowe 60
 - warstwa DAL 109
 - wzorce fabryki 75
 - zasoby 61

terminologia DDD (projektowanie sterowane domeną) 64

testowanie

- aplikacje DDD 64
- interfejsu użytkownika bez uruchamiania aplikacji 158
- logiki biznesowej 94
- podejście sterowane bazą danych 72
- podejście sterowane domeną 74
- testy jednostkowe modelu domenowego 83
- znaczenie 61

testy jednostkowe

- model domenowy 83
- TDD 60

tłumacz, wzorzec 28

Tooltip 13

transakcje biznesowe

- identyfikowanie 106
- UoW (jednostka pracy) 107

tworzenie dynamicznego kodu SQL 125

typ nadrzędny warstwy 78

U

udostępnianie modelu w modelu widoku 160

Unity

- informacje 49
- lokalizator usług 50
- pojemnik IoC 179
- porównanie z MEF 53
- wstrzykiwanie zależności 49
- zależności 54

UoW (jednostka pracy) 104

- cykl życia 106
- identyfikowanie transakcji biznesowej 106
- mapowanie domeny za pomocą NHibernate 122
- warstwa usługi biznesowej 144

UoW, konstruktor 119

UserControl 173

usługa dla transakcji biznesowych 147

usługa modalna 176

usługa sprawdzania poprawności 78

usługa WCF RIA

- informacje 112
- rozproszona warstwa danych 113

witryna WWW 115

usługi 21

usługi domenowe, obiekty DTO 68

utrzymywanie

- aplikacji biznesowych 4
- procedur składowanych 95

użytkownicy końcowi

- informacje 10
- komunikacja 15

V

VAB (Validation Application Block)

- informacje 138
- sprawdzanie poprawności obiektów 169

Validation Application Block 82

ValidationResults 144

Validator, klasa 144

ViewModelBase, klasa 185

ViewModelLocator 187

ViewModelValidator, klasa 169

Visual Studio

- Calcium 193
- Expression Blend 7
- MSTest 60
- MVVM Light Toolkit 184
- NHibernate 102
- testowanie interfejsu użytkownika 158

W

waga musza, wzorzec 27

warstwa biznesowa

- informacje 127
- kiedy tworzyć 141
- kod przykładowy: warstwa usługi biznesowej 143
- model 159
- navyki związane z warstwą BLL 142
- reguły biznesowe a reguły sprawdzania poprawności 128
- reguły biznesowe poprzez przepływ zadań i WF 4.0 133
- reguły biznesowe poprzez usługę 131
- zestawy narzędzi firm trzecich 138

warstwa danych

- model 159
- warstwa biznesowa 127

warstwa domenowa

informacje 20

warstwa biznesowa 127

warstwa dostępu do danych 20

warstwa interfejsu użytkownika

aplikacje biznesowe 20

reguły sprawdzania poprawności 129

warstwa logiki biznesowej. *Zobacz* BLL

warstwa usługi biznesowej 143

warstwy. *Zobacz także* moduły

informacje 21

podział interesów 24

reguły sprawdzania poprawności 128

SoC 19

warunki 140

WCF (Windows Communication Foundation)

AppFabric i przepływy zadań Windows 137

WorkflowApplication 136

WeakEvent, wzorzec 173

WeakEventManager, klasa 174

wersje 72

weryfikacja możliwości wykonania polecenia 189

WF (Workflow Foundation)

reguła biznesowa 129

reguły biznesowe poprzez przepływ zadań w WF 4.0 133

zdarzenia 147

wiązanie

atrasy modelu widoku z widokiem 158

modelu widoku z widokiem 155

widoku z poleceniem oferowanym przez model widoku 161

właściwości modelu widoku z kontrolkami widoku 156

widok

DataTemplate 173

definicja 1

dodawanie przestrzeni nazw 156

model widoku 166

MVP, wzorzec 36

MVVM 44, 155

obiekt w cyklu życia 154

PM, wzorzec 40

prezenter nadzorujący w MVP 39

transakcje biznesowe 106

wiązanie z poleceniem oferowanym przez model widoku 161

wzorce projektowe interfejsu użytkownika 30

widok interfejsu użytkownika, ogólna struktura 154

widok modalny 176

Windows Forms. *Zobacz także* WPF (Windows Presentation Foundation)

MVP, wzorzec 36

Windows Phone 7, Prism 191

Windows Presentation Foundation. *Zobacz* WPF

Windows Server, AppFabric 137

właściwości

Error, właściwość 168

FormatCommand, właściwość 165

ICommand, właściwość 161, 165

ID, właściwość 68

IsNew, właściwość 72

IsValid, właściwość 144

Item, właściwość 168

jednostek Employee i Customer 85

model widoku 160

OriginalText, właściwość 166

właściwości modelu widoku 156

właściwości modelu widoku 156

Workflow Activity Library, typ projektu 133

WorkflowApplication

kontrola nad wykonywaniem zestawu reguł 137

WCF 136

wykonywanie przepływu zadań 146

wywołania asynchroniczne 136

zalety i wady 138

WorkflowInvoker 136, 146

WPF (Windows Presentation Foundation)

Calcium SDK 193

Cinch 187

DataTemplate 171

Dispatcher 185

ICommand, interfejs 161

informacje vii

kontra Silverlight 4

MEF 52

MessageBox 176

MVC, wzorzec 35
 MVP, wzorzec 39
 PM, wzorzec 40-41
 prezenter nadzorujący w MVP 39
 Prism 191
 przestrzeń nazw 156
 rozproszona warstwa danych 112
 UserControl 173
 wstrzykiwanie zależności, wzorzec 45
 XAML 187
 WriteLog, komunikat 46
 wspólny język 64
 wstrzykiwanie zależności
 lokalizator usług 50
 MEF (Managed Extensibility Framework) 52
 wstrzykiwanie zależności, wzorzec 45, 48
 wszechobecny język 65
 wtyczki, NHibernate 102
 wydajność
 pakiety O/RM 95
 zarządzanie zdarzeniami 175
 wyjątki, wyrzucanie 77
 wymagania, aplikacje biznesowe 4
 wyrzucanie wyjątków 77
 wyzwalacze 6
 wzorce
 mediator, wzorzec 177
 Message, wzorzec komunikatu 175
 MVVM, wzorzec 183
 obserwator, wzorzec 189
 WeakEvent, wzorzec 173
 wzorce behawioralne 28
 wzorce fabryki 75
 wzorce kreacyjne 26
 wzorce projektowe 25
 DSL 54
 informacje 25
 IoC 45
 TDD 59
 wzorce projektowe interfejsu użytkownika 30
 MVC, wzorzec 31
 MVP, wzorzec 35
 wzorzec PM i MVVM 40
 wzorce strukturalne 27
 wzorce. *Zobacz także* wzorce projektowe
 EventAggregator, wzorzec 174

fasada, wzorzec 132
 wzorzec sprawdzania poprawności,
 fabryka 77

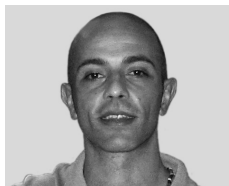
X

XAML (Extensible Application Markup
 Language) 187
 informacje 6
 kod 153
 kontrolki 9
 Power Toys 188
 Radical 189, 191
 samouczek In the Box 188
 silnik DataBind 168
 UserControl/Page/Window 154
 widok 158
 XAML Editor 188
 XAML Power Toys 188
 znaczniki 153, 159

Z

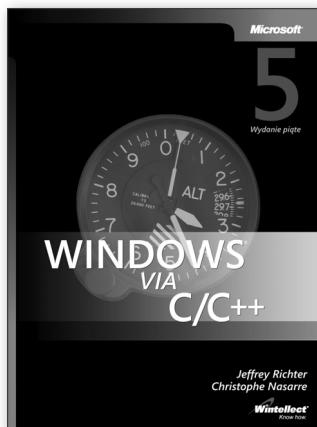
zachowanie i stan 72
 zależności, Unity 54
 zapytania, obiekt ObjectContext i obiekt
 ISession 125
 zbiorcze jednostki nadrzędne 65
 zdarzenia
 onComplete, zdarzenie 147
 programowanie zdarzeń w .NET 173
 PropertyChanged, zdarzenie 3, 166
 WorkflowApplication 136
 zdarzenia WeakEvent 173
 Cinch 187
 EventAggregator, wzorzec 174
 EventAggregator 179
 obserwator, wzorzec 189
 WeakEvent, wzorzec 173
 znaczniki deklaratywne interfejsu
 użytkownika 153

O autorze



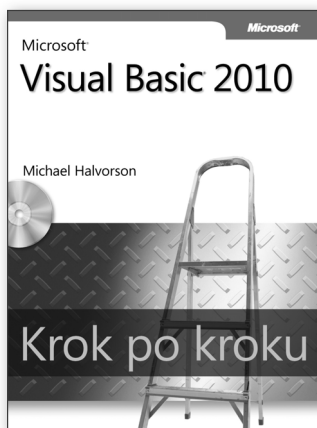
Raffaele Garofolo jest architektem oprogramowania .NET zajmującym się zawodowo budowaniem aplikacji biznesowych. Jest pasjonatem technologii .NET i Windows Presentation Foundation, a swój wolny czas poświęca na pisanie artykułów i wpisów na blogu dotyczących Windows Presentation Foundation oraz MVVM (<http://blog.raffaeu.com>).

Podręczniki i materiały źródłowe dla programistów



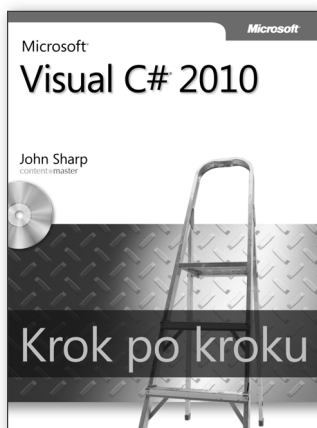
978-83-7541-023-5: **Windows via C/C++ wydanie 5**

Klasyczna książka Jeffreya Richtera, uaktualniona o informacje dotyczące systemów Windows XP, Windows Vista i Windows Server 2008. Zapewnia dogłębne, wyczerpujące wskazówki, zaawansowane techniki i rozbudowane przykłady kodu pomagające programować aplikacje oparte na systemach Windows.



978-83-7541-065-5: **Microsoft Visual Basic® 2010 Krok po kroku**

Praktyczny przewodnik krok po kroku nauki programowania w języku Visual Basic. Naucz się sam tworzyć aplikacje Visual Basic dla systemu Windows i sieci Web, koncentrując się na pojedynczych zagadnieniach. Niezależnie od Twoich umiejętności, w podręczniku znajdziesz praktyczne instrukcje, których wykonywanie prowadzi do opanowania najważniejszych narzędzi i metod – a przy okazji od razu powstają Twoje własne projekty!



978-83-7541-066-2: **Microsoft Visual C#® 2010 Krok po kroku**

Praktyczny przewodnik nauki programowania w języku Visual C#. Naucz się krok po kroku, jak tworzyć własne aplikacje w oparciu o język Visual C# 2010 oraz platformę Microsoft .NET Framework 4.0. Książka jest przeznaczona dla osób posiadających już podstawowe umiejętności w zakresie programowania, pozwalając w serii ćwiczeń praktycznie opanować zasady języka C# i tworzenie aplikacji i składników dla systemu Windows.

Więcej ofert i informacji na stronie www.ksiazki.promise.pl

Plik zabezpieczony watermarkiem jawnym i niejawnym: IFORMAT-EbookPoint/Helion

Microsoft® Office Specialist

Zmieńmy pracowników w superspeców od Office



Learn more >



Certiport to dostawca uznawanych na całym świecie certyfikacji w zakresie IT. Pakiet Microsoft Office stanowi dla organizacji okazję ulepszania umiejętności informatycznych w zakresie komputerów biurowych. Z tym oprogramowaniem jest związana certyfikacja Microsoft Office Specialist (MOS), cenne poświadczenie kwalifikacji, które uznaje umiejętności potrzebne do korzystania ze wszystkich elementów i funkcjonalności aplikacji Office. Poświadczenie kwalifikacji MOS może pomóc pracownikom w osiągnięciu większej skuteczności i wydajności przy jednoczesnym zmniejszeniu zależności od pomocy technicznej.

Dlaczego certyfikacja MS Office jest ważna dla pracowników, kierownictwa wydziałów i całej organizacji?

- Potwierdza programy szkoleniowe i rozwój pracownika
- Weryfikuje umiejętności i zdolności informatyczne użytkowników
- Stanowi czynnik wyróżniający przy ocenie podań o pracę
- Pozwala pracownikom na łatwiejsze i bardziej efektywne wykonywanie zadań i projektów dzięki sprawdzonym umiejętnościom w posługiwaniu się Microsoft Office
- Pozwala użytkownikom na pełne wykorzystanie oprogramowania aplikacyjnego w organizacji



Podstawowy certyfikat Microsoft Office Specialist potwierdza umiejętności dotyczące pakietu Microsoft Office. Dostępne są egzaminy dla następujących produktów:

- | | |
|---------------|---------------|
| • Word | • Access® |
| • Excel® | • Project® |
| • PowerPoint® | • SharePoint® |
| • Outlook® | |

Lista ośrodków szkoleniowych oferujących testy Microsoft Office jest dostępna na stronie www.certiport.promise.pl

www.certiport.com/mos
www.microsoft.com/learning

CERTIPORT®
Certiport Solution Provider

Kontakt

APN Promise SA
ul. Kryniczna 2
03-934 Warszawa

www.certiport.promise.pl
email: certiport@promise.pl
tel. (22) 355 16 42

Promise
• centrum
wiedzy
www.promise.pl