IDŹ DO

PRZYKŁADOWY ROZDZIAŁ

SPIS TREŚCI

KATALOG KSIĄŻEK

KATALOG ONLINE

ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK

DODAJ DO KOSZYKA

CENNIK I INFORMACJE

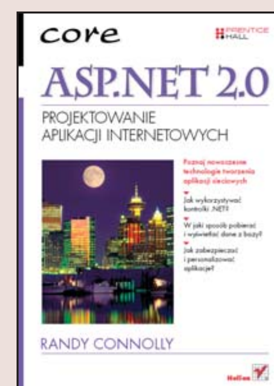
ZAMÓW INFORMACJE O NOWOŚCIACH

ZAMÓW CENNIK

CZYTELNIA

FRAGMENTY KSIĄŻEK ONLINE

ASP.NET 2.0. Projektowanie aplikacji internetowych



Autor: Randy Connolly
Tłumaczenie: Paweł Gonera, Ireneusz Jakóbik
ISBN: 978-83-246-1128-7
Tytuł oryginału: Core Internet Application Development with ASP.NET 2.0
Format: B5, stron: 928
oprawa twarda

Poznaj nowoczesne technologie tworzenia aplikacji sieciowych

- Jak wykorzystywać kontrolki .NET?
- W jaki sposób pobierać i wyświetlać dane z bazy?
- Jak zabezpieczać i personalizować aplikacje?

Wprowadzenie na rynek technologii .NET 2.0 zrewolucjonizowało sposób projektowania i tworzenia aplikacji internetowych. Arsenał programistów wzbogacił się o narzędzia cechujące się niespotykaną dotychczas wydajnością i elastycznością. Potężna biblioteka gotowych komponentów, nowe mechanizmy dostępu do danych, udoskonalone techniki zabezpieczania i personalizowania aplikacji oraz ich uruchamiania w środowisku produkcyjnym – wszystko to sprawiło, że budowanie nowoczesnych systemów działających w sieci stało się łatwe i szybkie. Jednak wraz ze wzrostem możliwości wzrosła także ilość wiedzy, którą musi przyswoić sobie programista zamierzający korzystać z technologii .NET 2.0.

Książka „ASP.NET 2.0. Projektowanie aplikacji internetowych" to doskonały podręcznik, za którego pomocą opanujesz niesamowite możliwości platformy .NET 2.0 w zakresie budowania systemów sieciowych. Dowiesz się, jak działają aplikacje ASP.NET, jak tworzyć formularze na stronach WWW i korzystać ze standardowych kontrolek udostępnianych przez platformę .NET 2.0. Nauczysz się łączyć aplikacje z bazami danych za pomocą mechanizmu ADO.NET oraz wyświetlać dane na stronach z wykorzystaniem kontrolek. Przeczytasz także o technikach projektowania złożonych aplikacji, o zarządzaniu sesjami, zabezpieczaniu aplikacji, stosowaniu mechanizmów personalizacji oraz wdrażaniu aplikacji w środowisku produkcyjnym.

- Tworzenie formularzy
- Model zdarzeń ASP.NET
- Stosowanie kontrolek serwera .NET
- Obsługa wyjątków
- Zarządzanie wyglądem witryny WWW
- Operacje na danych z wykorzystaniem ADO.NET
- Projektowanie aplikacji .NET
- Uwierzytelnianie użytkowników
- Tworzenie usług sieciowych
- Technologia ASP.NET AJAX

Opanuj do perfekcji zasady wykorzystywania technologii .NET 2.0 w programowaniu aplikacji sieciowych!

Wydawnictwo Helion
ul. Kościuszki 1c
44-100 Gliwice
tel. 032 230 98 63
e-mail: helion@helion.pl

Spis treści

O autorze 15

Wstęp 17

Część I ASP.NET 23

Rozdział 1. Wstęp do ASP.NET 25

Dlaczego ASP.NET? 25
Statyczna a dynamiczna zawartość stron WWW 26
Konkurencyjne dynamiczne technologie serwera 26
Zalety ASP.NET 29
.NET Framework 30
Składniki .NET Framework 31
Wykonywanie .NET 35
Formularze WWW ASP.NET 36
Język C# 40
Struktura aplikacji internetowej 41
Visual Studio 2005 43
Projekty WWW Visual Studio 45
Opcje serwera WWW 45
Samouczek: tworzenie formularzy WWW ASP.NET 48
Tworzenie witryny WWW w Visual Studio 48
Dodawanie nowego formularza WWW 50
Dodawanie kodu HTML do formularza WWW 50
Dodawanie logiki programowej 57
Obsługa błędów 60
Użycie debugera Visual Studio 63
Podsumowanie 66
Ćwiczenia 66
Najważniejsze zagadnienia 66
Odnośniki 67

Rozdział 2. Sposób działania ASP.NET 69
Model zdarzeń ASP.NET 69
Przesłanie danych 71
Stan widoku oraz stan kontrolki 73
Cykl życia strony 74
Kompilacja kodu ASP.NET 85
Kolejność kompilacji 88
Klasa Page 90
Request 90
Response 91
Server 92
Cykl życia aplikacji ASP.NET 93
Użytkownik żąda przesłania zasobu ASP.NET z serwera 93
Tworzenie podstawowych obiektów ASP.NET dla żądania 98
Przypisywanie obiektu HttpApplication do żądania 99
Przetwarzanie żądania z użyciem potoku HttpApplication 101
Podsumowanie 104
Ćwiczenia 104
Najważniejsze zagadnienia 104
Odnośniki 105

Rozdział 3. Wykorzystanie standardowych kontrolek serwera WWW 107
Wprowadzenie do kontrolek serwera 107
Kontrolki HTML serwera 108
Kontrolki serwera WWW 108
Kontrolki sprawdzające poprawność 108
Kontrolki użytkownika 109
Własne kontrolki serwera 109
Przegląd kontrolek serwera WWW 109
Wspólne składniki 109
Programowe manipulowanie właściwościami 113
Najważniejsze standardowe kontrolki serwera WWW 115
Kontrolka Label 115
Kontrolka Literal 117
Kontrolka TextBox 119
Kontrolki przycisków 122
Kontrolka CheckBox 128
Kontrolka RadioButton 131
Kontrolki list 132
Kontrolka Image 141
Kontrolka ImageMap 143
Kontrolka HyperLink 146
Kontrolka HiddenField 147
Kontrolka Table 149
Kontrolka Calendar 156
Podsumowanie 171
Ćwiczenia 171
Najważniejsze zagadnienia 172
Odnośniki 172

Rozdział 4. Dodatkowe standardowe kontrolki serwera WWW 173
Przegląd dodatkowych standardowych kontrolek serwera WWW 173
Kontrolka Panel 175
Kontrolki MultiView oraz View 182
Nawigacja pomiędzy widokami 185
Tworzenie paneli z zakładkami przy użyciu MultiView 185
Kontrolka Wizard 190
Zastosowanie kontrolki Wizard 194
Opis układu kreatora Wizard 196
Dostosowywanie kreatora 198
Obsługa zdarzeń kreatora 206
Kontrolka FileUpload 208
Przetwarzanie przesłanego pliku 211
Ograniczanie wielkości przesyłanego pliku 212
Kontrolka PlaceHolder 212
Tworzenie przeglądarki plików 213
Kontrolka AdRotator 220
Plik XML z danymi reklam 221
Wyświetlanie reklam z bazy danych 222
Programowanie kontrolki AdRotator 223
Kontrolka XML 224
Tworzenie pliku XSLT 226
Programowanie kontrolki XML 228
Podsumowanie 233
Ćwiczenia 235
Najważniejsze zagadnienia 236
Odnośniki 236

Rozdział 5. Obsługa wyjątków i kontrolki sprawdzania poprawności 237
Obsługa błędów 237
Obsługa wyjątków .NET 238
Obsługa wyjątków na poziomie klasy przy wykorzystaniu bloku try…catch 239
Obsługa wyjątków na poziomie strony 242
Obsługa wyjątków na poziomie aplikacji 243
Użycie kontrolek serwera sprawdzających poprawność 249
Proces kontroli poprawności formularza w ASP.NET 251
Kontrolka RequiredFieldValidator 257
Kontrolka ValidationSummary 258
Kontrolka CompareValidator 260
Kontrolka RangeValidator 262
Kontrolka RegularExpressionValidator 263
Kontrolka CustomValidator 270
Grupy kontroli poprawności 275
Podsumowanie 280
Ćwiczenia 280
Najważniejsze zagadnienia 282
Odnośniki 282

Rozdział 6. Dostosowywanie wyglądu witryny i zarządzanie nim ... 283
Zmiana wyglądu kontrolek serwera ... 283
Zastosowanie wspólnych właściwości formatujących ... 283
Użycie CSS dla kontrolek ... 285
Właściwości wyglądu, CSS i ASP.NET ... 290
Użycie tematów i motywów ... 290
Definiowanie motywów ... 292
Tworzenie tematów w Visual Studio ... 292
Stosowanie tematu ... 294
Jak działają tematy ... 294
Przesłanianie tematów ... 295
Motywy nazwane ... 296
Tematy i obrazy ... 297
Tematy i CSS ... 297
Dynamiczne ustawianie tematu ... 300
Tworzenie przykładowej strony z dwoma tematami ... 302
Strony wzorcowe ... 310
Definiowanie strony wzorcowej ... 313
Zagnieżdżone strony wzorcowe ... 316
Jak działają strony wzorcowe? ... 318
Programowanie strony wzorcowej ... 320
Strony wzorcowe i tematy ... 322
Kontrolki użytkownika ... 328
Tworzenie i modyfikacja kontrolek użytkownika ... 328
Dodawanie danych i funkcji do kontrolki użytkownika ... 330
Podsumowanie ... 331
Ćwiczenia ... 332
Najważniejsze zagadnienia ... 333
Odnośniki ... 333

Rozdział 7. Nawigacja w witrynie ASP.NET ... 335
Przedstawienie nawigacji w witrynie ASP.NET ... 335
Model dostawcy ... 337
Mapa witryny XML ... 337
Korzystanie z mapy witryny XML ... 340
Programowanie mapy witryny ... 343
Korzystanie z różnych map witryny ... 345
Inne funkcje mapy witryny ... 348
Kontrolka SiteMapPath ... 348
Nadawanie stylu kontrolce SiteMapPath ... 349
Integracja ciągów zapytania z SiteMapPath ... 353
Kontrolka Menu ... 356
Zastosowanie kontrolki Menu ... 358
Zmiana wyglądu kontrolki Menu ... 361
Obsługa zdarzeń menu ... 369
Kontrolka TreeView ... 375
Opis kontrolki TreeView ... 376
Zastosowanie kontrolki TreeView ... 378
Zmiana wyglądu kontrolki TreeView ... 380
Zastosowanie innych danych w kontrolce TreeView ... 382
Odpowiadanie na zdarzenia kontrolki TreeView ... 386

Podsumowanie ... 392
Ćwiczenia ... 392
Najważniejsze zagadnienia ... 392
Odnośniki ... 394

Część II Operacje na danych 395

Rozdział 8. Łączenie i reprezentacja danych ... 397

Podstawy dołączania danych ... 397
Jak korzystać z dołączania danych? ... 398
Co może być źródłem danych? ... 398
Wykorzystanie kolekcji ... 400
Interfejsy kolekcji ... 400
Użycie standardowych kolekcji ... 401
ArrayList ... 405
Typy ogólne ... 408
Kolekcje słownikowe ... 412
Tworzenie własnych kolekcji ogólnych ... 414
DataSet ... 418
Użycie obiektu DataTable ... 419
Zastosowanie obiektu DataSet ... 425
Relacje między obiektami DataTable ... 429
Integracja XML z DataSet ... 431
Wybór kontenera danych ... 438
Kolekcje .NET jako kontenery danych ... 439
Własne kolekcje jako kontenery danych ... 439
Obiekty DataSet jako kontenery danych ... 440
Typowane obiekty DataSet jako kontenery danych ... 441
Podsumowanie ... 442
Ćwiczenia ... 442
Najważniejsze zagadnienia ... 443
Odnośniki ... 443

Rozdział 9. Zastosowanie ADO.NET ... 445

Wprowadzenie do ADO.NET ... 445
Wybór dostawcy danych ... 449
Klasy dostawcy danych ... 449
Klasy DbConnection ... 450
Ciągi połączenia ... 450
Programowanie DbConnection ... 451
Przechowywanie ciągów połączenia ... 454
Pule połączeń ... 455
Klasy DbCommand ... 455
Tworzenie obiektu DbCommand ... 456
Polecenia SQL do pobierania, dodawania, modyfikowania i usuwania danych ... 457
Procedury składowane ... 458
Wykonywanie połączeń za pomocą obiektu DbCommand ... 459
Wykorzystanie klasy DbParameter ... 460
Użycie transakcji ... 463

Klasy DbDataReader ... 467
Programowanie DbDataReader ... 468
Niejawne zamykanie połączenia ... 470
Samouczek: odczytywanie i modyfikowanie danych ... 471
Klasy DbDataAdapter ... 479
Wypełnianie obiektu DataSet ... 480
Modyfikowanie danych ... 483
Tworzenie kodu ADO.NET niezależnego od dostawcy danych ... 485
Kontrolki źródła danych ... 488
Użycie parametrów ... 491
Modyfikowanie danych ... 494
Jak działają kontrolki źródeł danych? ... 494
Użycie ObjectDataSource ... 495
Podsumowanie ... 505
Ćwiczenia ... 505
Najważniejsze zagadnienia ... 506
Odnośniki ... 507

Rozdział 10. Kontrolki danych ... 509

Wprowadzenie do wielowartościowych kontrolek danych ... 509
Zrozumieć szablony ... 513
Wyrażenia wiązania danych ... 513
Kontrolka DataList ... 517
Używanie szablonów ogólnych ... 520
Łączenie stron za pomocą kontrolki DataList ... 523
Kontrolka Repeater ... 524
Kontrolka FormView ... 526
Przechodzenie między rekordami ... 529
Modyfikowanie danych ... 532
Kontrolka DetailsView ... 540
Dostosowywanie pól kontrolki DetailsView ... 541
Modyfikowanie danych kontrolki DetailsView ... 546
Kontrolka GridView ... 550
Dostosowywanie kolumn kontrolki GridView ... 550
Wybieranie wierszy ... 560
Stronicowanie w kontrolce GridView ... 565
Sortowanie w kontrolce GridView ... 568
Edycja danych w kontrolce GridView ... 570
Pozostałe funkcje kontrolki GridView ... 574
Podsumowanie ... 581
Ćwiczenia ... 581
Najważniejsze zagadnienia ... 582
Odnośniki ... 583

Rozdział 11. Projektowanie oraz implementacja aplikacji sieciowych ... 585

Projektowanie aplikacji ... 586
Korzystanie z warstw ... 587
Skutki podziału na warstwy ... 589
Model dwuwarstwowy ... 590

Model trójwarstwowy ... 592
Projektowanie oraz implementacja obiektu biznesowego ... 594
Programowe używanie obiektu biznesowego ... 602
Używanie obiektów biznesowych z kontrolką ObjectDataSource ... 603
Model czterowarstwowy ... 606
Projektowanie architektury czterowarstwowej ... 606
Modyfikowanie warstwy dostępu do danych ... 609
Tworzenie złożonej encji domeny ... 613
Tworzenie warstwy logiki aplikacji ... 615
Użycie architektury w warstwie prezentacji ... 618
Podsumowanie ... 627
Ćwiczenia ... 628
Najważniejsze zagadnienia ... 628
Odnośniki ... 629

Rozdział 12. Zarządzanie stanem w środowisku ASP.NET ... 631

Stan przechowywany po stronie klienta ... 632
Stan widoku ... 632
Stan kontrolek ... 636
Pola ukryte ... 636
Łańcuchy zapytań ... 636
Cookies ... 637
Stan aplikacji ... 638
Plik Global.asax ... 639
Stan sesji ... 641
Jak działa stan sesji? ... 642
Dostawcy stanu sesji ... 644
Właściwości profilu ... 650
Pamięć podręczna środowiska ASP.NET ... 650
Buforowanie danych aplikacji ... 651
Zależności pamięci podręcznej ... 655
Buforowanie zwracanych stron ... 657
Podsumowanie ... 660
Ćwiczenia ... 660
Najważniejsze zagadnienia ... 661
Odnośniki ... 661

Część III Implementacja aplikacji sieciowych 663

Rozdział 13. Bezpieczeństwo, członkostwo i zarządzanie rolami ... 665

Wprowadzenie do zagadnień bezpieczeństwa w środowisku ASP.NET ... 666
Omówienie bezpieczeństwa w serwerze IIS ... 667
Proces bezpieczeństwa w środowisku ASP.NET ... 670
Bezpieczeństwo bazujące na uprawnieniach kodu oraz poziomy zaufania środowiska ASP.NET ... 672
Uwierzytelnianie w środowisku ASP.NET ... 675
Uwierzytelnianie formularzy ... 676
Korzystanie z uwierzytelniania formularzy ... 676
Tworzenie formularza logowania ... 679
Jak działa uwierzytelnianie formularzy? ... 684
Używanie biletów uwierzytelniania bez cookies ... 687

Model dostawcy ... 689
Architektura modelu dostawcy ... 690
Tworzenie dostawców niestandardowych ... 692
Członkostwo ... 699
Omówienie systemu członkowskiego ... 699
Konfiguracja dostawcy SqlMembershipProvider ... 700
Używanie interfejsu API członkostwa ... 702
Zarządzanie rolami ... 708
Dostawca ról ... 709
Zarządzanie rolami ... 709
Używanie interfejsu API zarządzania rolami ... 712
Kontrolki logowania ... 717
Kontrolka Login ... 718
Kontrolka LoginName ... 723
Kontrolka LoginStatus ... 723
Kontrolka LoginView ... 724
Kontrolka ChangePassword ... 726
Kontrolka PasswordRecovery ... 727
Kontrolka CreateUserWizard ... 729
Podsumowanie ... 729
Ćwiczenia ... 730
Najważniejsze zagadnienia ... 731
Odnośniki ... 732

Rozdział 14. Personalizacja za pomocą profili oraz składników Web Part ... 733

Profile środowiska ASP.NET ... 733
Definiowanie profili ... 734
Użycie danych profilu ... 735
Jak działają profile? ... 739
Zapisywanie i odczytywanie danych profilu ... 741
Używanie typów niestandardowych ... 742
Praca z użytkownikami anonimowymi ... 745
Kiedy używać profili? ... 749
Składniki Web Part ... 751
Składniki Web Part, strefy Web Part oraz menedżer składników Web Part ... 753
Tworzenie składników Web Part i korzystanie z nich ... 756
Konstruowanie składników Web Part z kontrolek użytkownika ... 762
Tworzenie składników Web Part na podstawie kontrolek niestandardowych ... 766
Zmiana trybów wyświetlania ... 769
Tryb projektowania ... 771
Tryb katalogu ... 773
Tryb edycji ... 774
Połączenia składników Web Part ... 780
Podsumowanie ... 791
Ćwiczenia ... 792
Najważniejsze zagadnienia ... 793
Odnośniki ... 794

Rozdział 15. Usługi WWW ... 795
Wprowadzenie do usług WWW ... 795
Zalety usług WWW ... 797
Konsumowanie usług WWW ... 798
Jak konsumować usługę WWW za pomocą programu Visual Studio? ... 799
Konsumowanie usług WWW w kontrolce użytkownika ... 803
Konsumowanie usługi WWW Amazon ... 806
Konsumowanie usług WWW a wydajność ... 814
Asynchroniczne usługi WWW ... 817
Tworzenie usług WWW ... 822
Tworzenie prostej usługi z cytatami ... 824
Testowanie usługi z cytatami ... 828
Tworzenie fasady usługi WWW dla klasy biznesowej lub klasy logiki aplikacji ... 830
Wskazówki dotyczące tworzenia usług WWW ... 831
Podsumowanie ... 833
Ćwiczenia ... 833
Najważniejsze zagadnienia ... 834
Odnośniki ... 834

Rozdział 16. Internacjonalizacja i wdrażanie ... 837
Internacjonalizacja aplikacji WWW ... 837
Wprowadzenie do plików z zasobami ... 839
Generowanie plików z zasobami ... 839
Lokalizacja plików z zasobami ... 843
Zasoby globalne ... 848
Ustawienia kultury na poziomie strony ... 849
Wdrażanie ... 854
Ręczne kopiowanie plików z komputera programisty na komputer docelowy ... 855
Prekompilacja witryn WWW ... 857
Tworzenie programu instalacyjnego za pomocą narzędzia Web Setup Project ... 861
Podsumowanie ... 867
Ćwiczenia ... 867
Najważniejsze zagadnienia ... 867
Odnośniki ... 868

Dodatki ... 869

Dodatek A Rzut oka na technologię ASP.NET AJAX ... 871

Skorowidz ... 893

11

Projektowanie oraz implementacja aplikacji sieciowych

„Projekt… jest poznaniem związków zachodzących między różnymi rzeczami… Nie możesz wynaleźć projektu. Poznajesz go — w czwartym wymiarze. To znaczy swoją krwią i kośćmi, ale też swoimi oczami”.

— D.H. Lawrence, *Art & Morality*

Środowisko ASP.NET jest na tyle dobrze wyposażoną technologią projektowania, że może być kuszące skupienie się wyłącznie na poszczególnych funkcjach, które wchodzą w jej skład. W niniejszym rozdziale środek ciężkości zostanie przesunięty z konkretnych kontrolek i klas na niektóre zagadnienia związane z tworzeniem w środowisku ASP.NET nieco bardziej złożonych aplikacji sieciowych. Rozdział rozpocznie się omówieniem modelu aplikacji sieciowych oraz niektórych często spotykanych modeli warstwowych tego typu aplikacji, po czym zostanie opisana implementacja przykładowej architektury warstwowej.

Chociaż dobry projekt rzeczywiście nie musi być, jak to twierdzi Lawrence, wyłącznie wynikiem racjonalnego rozumowania, istnieje całkiem pokaźna literatura na temat projektowania oprogramowania, która dostarcza wielu wskazówek i nauk, które mogą być przydatne podczas projektowania aplikacji sieciowych. Szczegółowe omówienie zagadnienia współczesnego projektowania oprogramowania wykracza poza zakres tej książki. W materiałach dodatkowych, wymienionych pod koniec niniejszego rozdziału, wskazano kilka szczególnie przydatnych prac poświęconych temu bardzo obszernemu zagadnieniu. Główną natomiast rolą tego rozdziału jest omówienie różnych, najlepszych podejść do projektowania aplikacji sieciowych, a następnie zilustrowanie ich kilkoma przykładowymi implementacjami tych projektów.

Projektowanie aplikacji

Środowisko ASP.NET dysponuje wieloma funkcjami, które wspomagają szybkie projektowanie aplikacji sieciowych. Wydajne deklaracyjne kontrolki umożliwiają programistom tworzenie wyszukanych interfejsów służących do wyświetlania i edycji danych przy małym nakładzie programowania lub w ogóle bez konieczności pisania kodu. Wygoda korzystania z programu Visual Studio Designer i jego moc także zachęcają programistów do szybkiej implementacji stron WWW. Środowisko ASP.NET i Visual Studio zdają się mówić: „Nie marnuj czasu na projektowanie i planowanie; bierz się do przeciągania i upuszczania i zakończ projekt!”. Taka możliwość szybkiego opracowania strony jest zachęcająca zwłaszcza podczas tworzenia witryny WWW z ograniczoną liczbą ***przypadków użycia***, czyli niewielkim zakresem wymogów funkcjonalnych.

Funkcjonalności wielu „prawdziwych” aplikacji sieciowych nie da się jednak opisać za pomocą zaledwie kilku przypadków użycia. W rzeczywistości mogą być dziesiątki, a nawet setki opisanych przypadków, które wymagają wysiłku wielu programistów poświęcających sporo czasu, aby je wszystkie zaimplementować. To właśnie podczas pracy nad takim rodzajem aplikacji sieciowych szybki styl programowania może w rzeczywistości spowolnić ogół procesów tworzenia aplikacji.

Prawdziwe projekty programistyczne są wyjątkowo wrażliwe na zmiany wymagań; projekty sieciowe są chyba jeszcze bardziej wrażliwe. Oznacza to, że funkcje aplikacji sieciowej są rzadko w pełni określone przed rozpoczęciem projektowania. Dodawane są nowe funkcje, a z innych się rezygnuje. Zmienia się model danych oraz wymogi co do ich przechowywania. W miarę przemieszczania się projektu przez poszczególne etapy cyklu projektowania środowisko uruchomieniowe zmienia się począwszy od laptopa programisty, przez serwer testowy, serwer docelowy, a skończywszy być może na farmie serwerów sieciowych. Programista może początkowo testować program z bazą danych programu Access, po czym przenieść dane do serwera SQL, a następnie, po fuzji firm, powrócić do bazy danych Oracle. Na kilka tygodni przed testowaniem alfa klient może wprowadzić zmiany, które w przypadku pewnych informacji spowodują konieczność pracy z zewnętrzną usługą zamiast z lokalną bazą danych. Analitycy użyteczności mogą mocno skrytykować strony witryny, co może spowodować konieczność wprowadzenia zmian w interfejsie użytkownika.

To właśnie w takim środowisku programowania sieciowego zwyczaje szybkiego opracowywania projektów mogą przynieść więcej szkód niż korzyści. Przestrzeganie jednak zasad prawidłowego projektowania oprogramowania może w tym samym środowisku zwrócić się w postaci całkiem wymiernych zysków. Poświęcenie czasu na utworzenie dobrze zaprojektowanej infrastruktury aplikacji może sprawić, że aplikacje sieciowe będą łatwiejsze do modyfikowania i w pielęgnacji, prostsze w rozbudowie i poszerzaniu ich funkcjonalności, mniej podatne na błędy, a także łatwiejsze w tworzeniu. W pierwszym podrozdziale niniejszego rozdziału podjęto próbę rzucenia nieco światła na proces projektowania tego typu infrastruktury aplikacji.

Korzystanie z warstw

Jedną z najważniejszych zalet środowiska ASP.NET w porównaniu z czysto skryptowymi technologiami, takimi jak ASP albo PHP, jest możliwość tworzenia bardziej elastycznych aplikacji za pomocą aktualnie obowiązujących dobrych praktyk projektowania obiektowo zorientowanego oprogramowania. Prawdopodobnie najważniejsza z tych praktyk polega na podzieleniu projektu aplikacji na dyskretne warstwy logiczne.

Czym jest warstwa? ***Warstwa*** to po prostu grupa klas, które są spokrewnione funkcjonalnie lub logicznie. Używanie warstw jest sposobem na organizowanie projektu oprogramowania w grupy klas, które służą wspólnemu celowi. Warstwa zatem nie jest rzeczą lecz zasadą organizacji. Oznacza to, że warstwy są sposobem na projektowanie aplikacji.

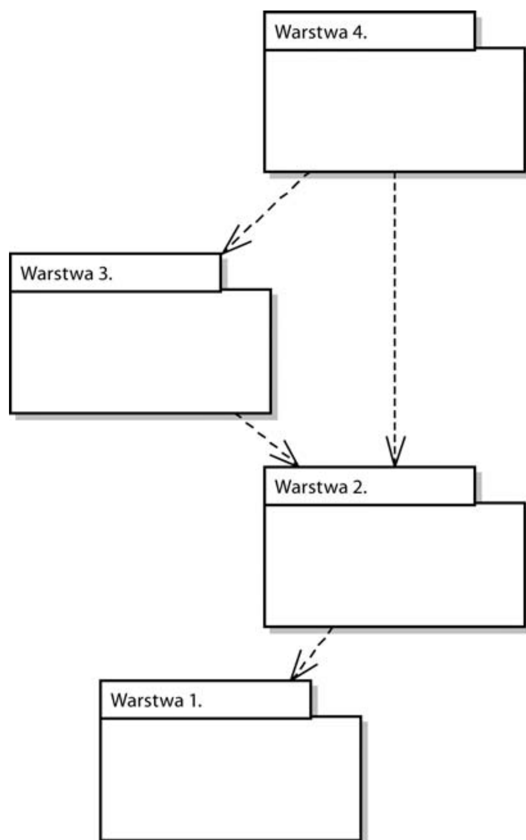
Powodem, dla którego tak wielu programistów aplikacji przyjęło warstwy jako zasadę organizacji swoich aplikacji, jest fakt, że warstwa nie stanowi dowolnej grupy klas. Każda warstwa aplikacji powinna być ***spójna*** (czyli klasy powinny dotyczyć mniej więcej tego samego i mieć zbliżony poziom abstrakcji). Spójne warstwy i klasy są zazwyczaj łatwiejsze do zrozumienia, wielokrotnego używania i pielęgnowania.

Celem dzielenia na warstwy jest rozdzielenie funkcji oprogramowania między klasy w taki sposób, aby zminimalizować ***skojarzenia*** danej klasy z innymi klasami. Skojarzenie odnosi się do liczby klas, z której korzysta określona klasa. Kiedy pewna klasa korzysta z innej klasy, jest od niej zależna. Wszelkie zmiany wprowadzone w interfejsie używanej klasy mogą mieć wpływ na klasę od niej zależną. Kiedy klasy aplikacji są silnie skojarzone, zmiana w jednej klasie może mieć wpływ na wiele innych klas. Zredukowanie skojarzeń sprawia, że projekt staje się elastyczniejszy i łatwiej rozszerzalny.

Pewien stopień skojarzeń jest oczywiście niezbędny w każdej aplikacji — w przeciwnym razie klasy nie wchodziłyby we wzajemne interakcje. Organizując klasy aplikacji w warstwy, można mieć nadzieję na uzyskanie mniejszego stopnia skojarzeń niż w przypadku, gdyby podział na warstwy nie był przyjętą zasadą organizacji. Warstwa może zależeć od *interfejsu* innej warstwy, jednak powinna być niezależna od jej *implementacji*.

Przy takim podejściu każda warstwa cechuje się ściśle ograniczoną liczbą zależności. ***Zależność*** (znana także jako relacja korzystania) to taki związek między dwoma elementami, w którym zmiana w jednym elemencie ma wpływ na drugi element. Na rysunku 11.1 poszczególne warstwy są zależne tylko od warstw, które znajdują się „poniżej”, to znaczy od warstw, które są bardziej „niskopoziomowe” czy też bardziej zależne od elementów zewnętrznych, takich jak bazy danych albo usługi sieciowe.

Proszę zwrócić uwagę, co oznacza zależność w odniesieniu do warstw. Oznacza mianowicie, że klasy w warstwie „powyżej” korzystają z klas i metod warstwy (lub warstw) „poniżej”, ale już nie vice versa. Rzeczywiście, jeśli zależności zachodzą między wszystkimi warstwami w obie strony, traci się całkowicie korzyści z podziału na warstwy.

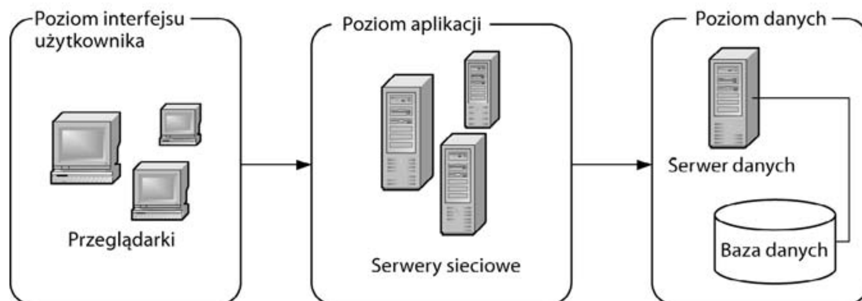


Rysunek 11.1. *Wizualizacja warstw*

Schemat warstw pokazany na rysunku 11.1 jest *otwartym* (luźnym) schematem podziału na warstwy, ponieważ niektóre warstwy zależą od więcej niż jednej innej warstwy. W schemacie *zamkniętym* (nieprzejrzystym) każda warstwa zależy tylko od jednej niższej warstwy. Chociaż schemat zamknięty byłby idealny, to jednak w praktyce jest on czasami trudny do osiągnięcia. Jak zauważył Martin Fowler w swojej książce *Architektura systemów zarządzania przedsiębiorstwem. Wzorce projektowe*, „większość (architektur warstwowych) jest w większości nieprzejrzysta".

Na koniec muszę jeszcze wspomnieć, że niektórzy autorzy używają terminu ***poziom*** w tym samym znaczeniu, w jakim ja używam terminu *warstwa*. Większość jednak współczesnej literatury na temat architektury i projektowania oprogramowania używa terminu *poziom* w zupełnie innym znaczeniu. Znaczenie to odnosi się do granic procesu. Różne poziomy najczęściej oznaczają różne miejsca w sieci. Typowa aplikacja sieciowa może być na przykład rozpatrywana jako architektura trójpoziomowa: stacja robocza użytkownika jest poziomem interfejsu użytkownika, serwer sieciowy jest poziomem aplikacji, a system zarządzania bazą danych działający na odrębnym serwerze danych jest poziomem bazy danych, co pokazano na rysunku 11.2. W dalszej części niniejszej książki termin *poziom* będzie używany w tym drugim znaczeniu, z kolei słowo *warstwa* będzie odnosić się do pojęciowego grupowania klas w obrębie aplikacji.

Rysunek 11.2.
Poziomy



Skutki podziału na warstwy

Istnieje wiele korzyści, które może przynieść projektowanie aplikacji z użyciem warstw. Pierwszą i najważniejszą zaletą korzystania z warstw jest fakt, że wynikowa aplikacja powinna być o wiele bardziej elastyczna i adaptowalna dzięki niższemu ogólnemu poziomowi skojarzeń w aplikacji. Jeśli niski poziom skojarzeń łączy się z wysokim stopniem spójności w obrębie warstwy (łącznie z dobrze określonym interfejsem dostępu do warstwy), programista powinien mieć możliwość modyfikowania, poszerzania lub wzbogacania warstwy bez wywierania nadmiernego wpływu na resztę aplikacji. Jak zauważył Craig Larman w swojej książce *Agile and Interactive Development*, modyfikowanie i pielęgnacja aplikacji zwykle zabiera większość czasu poświęconego na jej opracowanie. W rezultacie zwiększenie możliwości pielęgnowania aplikacji jest istotną zaletą.

Inna korzyść podziału na warstwy polega na możliwości ponownego użycia danej warstwy w innych aplikacjach, zwłaszcza jeśli została ona zaprojektowana z myślą o wielokrotnym użyciu. Autor używał w dziesiątkach aplikacji sieciowych nieco bardziej złożonych wersji warstw, które zostaną zaimplementowane w dalszej części tego rozdziału. Wreszcie kolejną zaletą warstw jest możliwość testowania funkcji aplikacji zawartych w warstwie odrębnie i niezależnie od innych warstw.

Używanie warstw ma jednak też swoje wady. Jedna z nich polega na tym, że aplikacja może stać się nieco trudniejsza w zrozumieniu i opracowaniu dla programistów, którzy znają jedynie model projektowania oparty na języku skryptowym strony. Chociaż zazwyczaj warto ponieść taki koszt, to jednak w przypadku bardzo prostych aplikacji sieciowych, składających się zaledwie z kilku stron, używanie wielu warstw abstrakcji prawdopodobnie nie będzie opłacalne.

Inna wada korzystania z warstw polega na tym, że dodatkowe poziomy abstrakcji mogą spowodować spadek wydajności wykonywania się programu. Biorąc jednak pod uwagę czas poświęcany na komunikowanie się komputerów w sieci, dodatkowy czas przeznaczony na komunikację między obiektami w komputerze jest względnie nieistotny.

Ostatnią wadą podziału na warstwy jest trudność w ustanowieniu prawidłowego schematu podziału, który zostanie zastosowany. Jeśli w aplikacji jest za mało warstw, a każda warstwa ma za dużo obowiązków, wówczas osiągnięcie pełni korzyści z podziału na warstwy może być niemożliwe. Jeśli z kolei w aplikacji jest za wiele warstw, zapewne będzie ona zbyt

skomplikowana, a tym samym mniej elastyczna i trudniejsza do zrozumienia. W następnych kilku podrozdziałach spróbuję odnieść się do tych wad, analizując kilka przykładowych architektur warstwowych.

Kiedy należy nadać nazwy modelom podziału na warstwy, okazuje się, że brakuje standardowego nazewnictwa. Analizując literaturę, można odnieść wrażenie, że istnieją dziesiątki różnych schematów dzielenia na warstwy. W tym przypadku będzie jednak inaczej. Jak zauważa Eric Evans w swojej książce *Domain-Driven Design*, przemysł oprogramowania zorientowanego obiektowo dzięki swojemu doświadczeniu i przyjętym konwencjom zbliżył się do architektur warstwowych, łącznie z wypracowaniem zbioru w miarę standardowych warstw, które jednak nie mają standardowych nazw. W następnych trzech podrozdziałach zostaną omówione trzy najczęściej spotykane modele warstw.

Model dwuwarstwowy

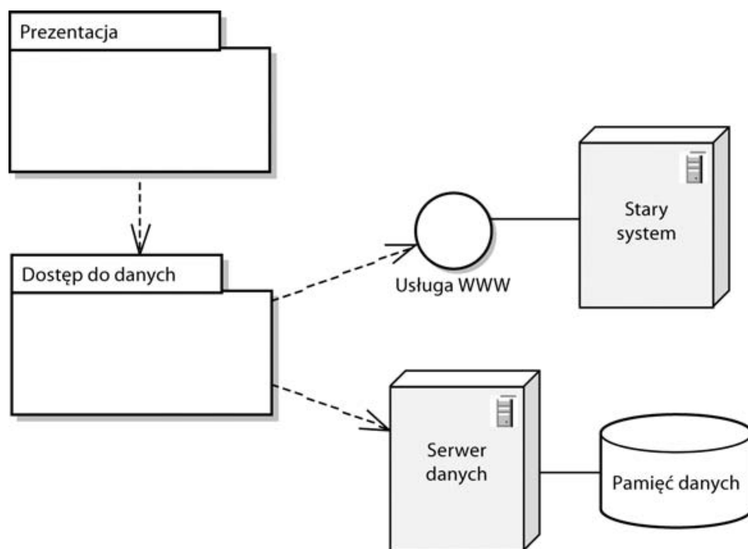
Pokazano już w poprzednim rozdziale, że jest możliwe utworzenie sterowanej danymi aplikacji sieciowej ASP.NET niemal zupełnie bez konieczności programowania dzięki kontrolce `SqlDataSource` oraz innym kontrolkom danych, takim jak `DetailsView` i `GridView`. Niemniej, jak już wspomniano w rozdziale 9., wielu programistów nie lubi umieszczać szczegółów dostępu do bazy danych w formularzach WWW, nawet jeśli dzieje się to w deklaracyjnych formach kontrolki `SqlDataSource`. W rozdziale 9. utworzono serię spokrewnionych klas, które obsługiwały dostęp do bazy danych za pośrednictwem programowania w technologii ADO.NET, a następnie użyto ich w formularzach WWW, korzystając z kontrolki `ObjectDataSource`. Był to przykład architektury dwuwarstwowej.

W modelu dwuwarstwowym szczegóły dostępu do danych są zawarte w odrębnych klasach, które są różne od samych formularzy WWW. Dwie warstwy występujące w takim modelu można nazwać warstwą prezentacji oraz warstwą dostępu do danych.

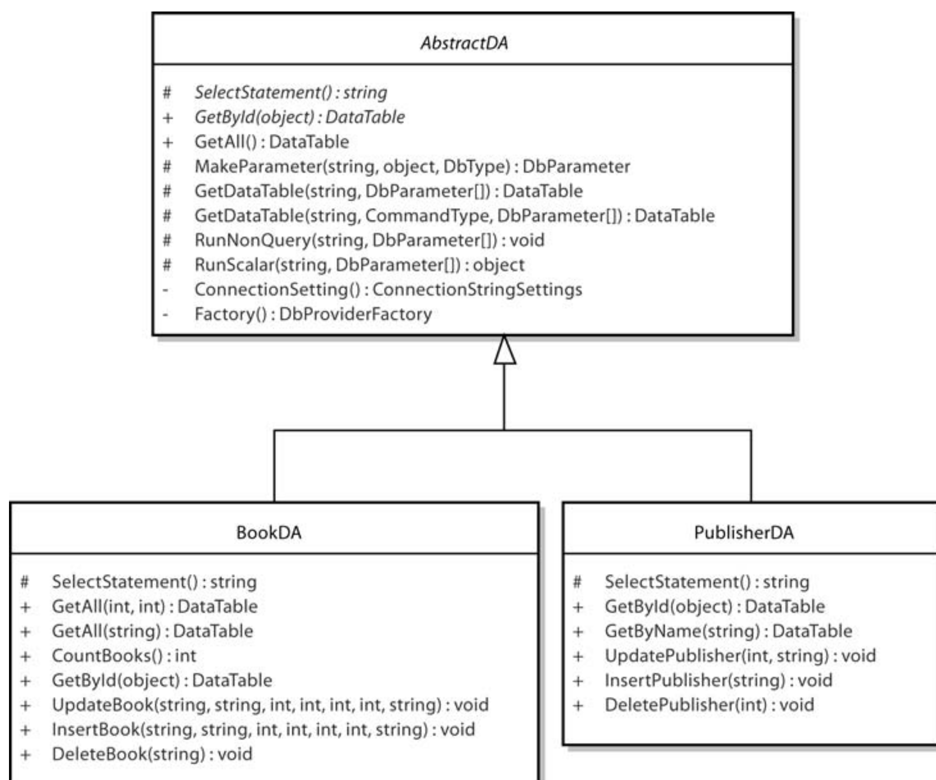
Warstwa prezentacji udostępnia interfejs użytkownika aplikacji oraz obsługuje interakcję między użytkownikiem i aplikacją. W tej książce warstwa prezentacji składa się ze stron ASP.NET wraz z ich klasami z kodem ukrytym oraz dodatkowymi kontrolkami użytkownika lub kontrolkami niestandardowymi. ***Warstwa dostępu do danych*** udostępnia komunikację z systemami zewnętrznymi, takimi jak bazy danych, systemami rozpowszechniania komunikatów lub zewnętrzne usługi WWW, co pokazano na rysunku 11.3.

Warstwa dostępu do danych z rozdziału 9. zawierała cały kod ADO.NET. Każda z klas warstwy była odpowiedzialna za tworzenie, odczytywanie, aktualizację i usuwanie w pojedynczej tabeli bazy danych (***CRUD***, od ang. *Create*, *Retrieve*, *Update*, *Delete*). Na rysunku 11.4 pokazano dwie przykładowe klasy dostępu do danych oraz ich wspólną klasę nadrzędną.

Należy zauważyć, że gdy dane są odczytywane z bazy w warstwie dostępu do danych, są one zawarte w klasie `DataTable` (chociaż można by też użyć klasy `DataSet`), po czym przekazywane do warstwy prezentacji.



Rysunek 11.3. *Model dwuwarstwowy*



Rysunek 11.4. *Warstwa dostępu do danych*

Zaletą modelu dwuwarstwowego jest jego lekkość i względna łatwość implementacji. Model ten może jednak nie być idealnym rozwiązaniem, jeśli logika aplikacji witryny staje się bardziej złożona. Termin ***logika aplikacji*** odnosi się do międzydomenowej logiki albo przepływu procesów w samej aplikacji. Przykładem logiki aplikacji może być przepływ pracy w sklepie internetowym związany z obsługą zamówienia. Po kliknięciu przez użytkownika przycisku służącego do składania zamówień aplikacja sieciowa musi utworzyć zamówienie w bazie danych, skontaktować się z systemem płatności i przedłożyć płatność. Jeśli nastąpi niepowodzenie, zamówienie musi zostać wycofane, a użytkownik powiadomiony. W przeciwnym razie zamówienie musi zostać przekazane, system realizacji zamówienia powiadomiony, a strona z potwierdzeniem zamówienia wyświetlona. Taki przepływ pracy ma wiele etapów z wieloma zestawami logiki warunkowej.

Model dwuwarstwowy może okazać się daleki od ideału, jeśli zachodzi potrzeba dodania do aplikacji bardziej złożonych reguł biznesowych. Określenie ***reguły biznesowe*** odnosi się do standardowych typów walidacji danych wejściowych omówionych w rozdziale 5., a także do bardziej złożonych reguł dotyczących danych, często charakterystycznych dla metod, którymi posługuje się organizacja, prowadząc swoją działalność.

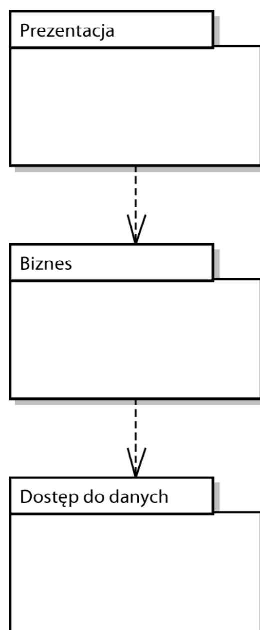
W witrynach, w których logika aplikacji lub reguły biznesowe są złożone, taka dodatkowa złożoność musi być zazwyczaj zaimplementowana w warstwie prezentacji, gdy użyto modelu dwuwarstwowego. W konsekwencji kody ukryte zwykle stają się zbyt skomplikowane i przepełnione powtarzalnym kodem logiki biznesowej i aplikacji. Dokonałem kiedyś dla klienta refaktoryzacji dwuwarstwowej aplikacji sieciowej ASP.NET, w której każda klasa z kodem ukrytym dla wszystkich z ponad trzydziestu formularzy WWW liczyła ponad 4000 wierszy. W rezultacie ich modyfikacja była prawdziwym koszmarem z powodu ich długości i złożoności. Rozwiązaniem dla tego określonego przypadku była zmiana architektury witryny na model trójwarstwowy.

Model trójwarstwowy

W architekturze trójwarstwowej nowa warstwa zostaje dodana do modelu dwuwarstwowego. Ta nowa warstwa jest ogólnie znana pod nazwą ***warstwy biznesowej*** (rysunek 11.5).

Najważniejszą zmianą w porównaniu z modelem dwuwarstwowym jest brak konieczności umieszczania danych z bazy w obiektach klasy `DataTable`. Warto przypomnieć, że w modelu dwuwarstwowym dane są zwracane z przykładowych klas dostępu do danych w obiektach klasy `DataTable` (zamiast których można użyć obiektów klasy `DataSet`). Kiedy obiekt klasy `DataTable` albo `DataSet` zostanie bezpośrednio powiązany z kontrolką, ma to swoje niewątpliwe zalety. Czasami jednak zachodzi potrzeba programowego uzyskania i przetworzenia poszczególnych danych z obiektów. Jak już pokazano w rozdziale 8., zakodowanie takiej funkcji jest nieco skomplikowane. Jeśli należy na przykład pobrać wartość pola `LastName` z pierwszego wiersza `DataRow` pierwszej tabeli `DataTable` w zbiorze `DataSet`, kod będzie wyglądać następująco:

```
txtLastName.Text = myDataSet.Tables[0].Rows[0]["LastName"];
```



Rysunek 11.5. *Architektura trójwarstwowa*

Co gorsza, ponieważ tabela DataTable prawdopodobnie odzwierciedla układ tabeli bazy danych, do warstwy prezentacji został wprowadzony kod zależny od bazy danych. Ponadto jeśli nawet program nie jest pisany pod kątem wypełnionych obiektów klasy DataTable albo DataSet, tego typu podejście dwuwarstwowe nadal gwarantuje, że w formularzach WWW znajdzie się duża ilość kodu specyficznego dla bazy danych. Weźmy na przykład pod uwagę następującą definicję kontrolki GridView:

```
<asp:GridView ID="grdSample" runat="server"
   DataSource="someSource" >

   <Columns>
      <asp:BoundField DataField="Isbn" HeaderText="Isbn" />
      <asp:TemplateField HeaderText="Title">
         <ItemTemplate>
           <%# Eval("Title") %>
         </ItemTemplate>
      </asp:TemplateField>
   </Columns>
</asp:GridView>
```

W tym kodzie znajdują się odniesienia do dwóch szczegółów implementacji bazy danych, a mianowicie do pól Isbn oraz Title. W modelu dwuwarstwowym (lub jeszcze gorzej w jednowarstwowym) takie szczegóły będą porozrzucane we wszystkich formularzach WWW. Ściśle kojarząc formularze WWW z implementacją bazy danych, w znaczącym stopniu ogranicza się ich elastyczność. Oznacza to, że każda zmiana w bazie danych (jak na przykład zmiana nazw pól) prawdopodobnie będzie skutkować koniecznością poświęcenia mnóstwa czasu na wprowadzanie poprawek do formularzy, które korzystają z tej bazy. Chociaż może to być do przyjęcia w przypadku hojnego i wyrozumiałego klienta, który płaci za przepracowane godziny, to jednak w przypadku większości prawdziwych klientów taka sytuacja nie powinna mieć miejsca.

Projektowanie oraz implementacja obiektu biznesowego

Przewaga podejścia trójwarstwowego polega na możliwości lepszego odizolowania warstwy prezentacji od potencjalnych zmian w bazie danych. Formularze WWW nie prowadzą interakcji ze specyficznymi dla bazy danych obiektami klasy `DataTable` albo `DataSet`, lecz z klasami ogólnie nazywanymi ***obiektami biznesowymi***. W takim podejściu każdy rekord jest reprezentowany przez odrębną instancję odpowiedniej klasy biznesowej. Wartości pól pochodzące z rekordów są przechowywane w danych składowych obiektu biznesowego. Wartości tych danych są z kolei udostępniane za pośrednictwem właściwości, jak to pokazano w poniższym przykładzie:

```
public class SampleBusinessObject
{

   // Dane składowe
   private int _id;
   private string _name;
   …

   // Właściwości
   public int Id
   {
      get { return _id; }
      set { _id= value; }
   }
   public string Name
   {
      get { return _name; }
      set { _name = value; }
   }
   …
}
```

Formularz WWW może zatem uzyskać dostęp do danych z rekordu, korzystając z dużo czystszej składni właściwości. Do tych właściwości można uzyskiwać dostęp programowo w pokazany poniżej sposób:

```
txtName.Text = aCustomer.Name
```

Do właściwości można również odnosić się deklaratywnie w wyrażeniach wiązania danych:

```
<asp:GridView ID="grdSample" runat="server"
   DataSource="someSource" >

   <Columns>
      <asp:BoundField DataField="Id" HeaderText="Id" />
      <asp:TemplateField HeaderText="Name">
         <ItemTemplate>
            <%# Eval("Name") %>
         </ItemTemplate>
      </asp:TemplateField>
   </Columns>
</asp:GridView>
```

Główna zaleta takiego podejścia polega na tym, że warstwa prezentacji nie zależy już od szczegółów implementacji bazy danych, ponieważ następuje odwołanie do właściwości obiektu, a nie do nazwy pola w tabeli `DataTable`. Ponadto dzięki wyeliminowaniu obiektu `DataSet` albo `DataTable` kod stał się zdecydowanie czystszy i bardziej czytelny.

Jeśli jest potrzebna weryfikacja wyrażeń wiązania danych podczas kompilacji, można zastosować bardziej rozwlekłą alternatywę dla metody `Eval`, którą jest pokazane poniżej polecenie `Container.DataItem`:

```
<ItemTemplate>
   <%# ((Book)Container.DataItem)Book.Name %>
</ItemTemplate>
```

Warstwa biznesowa zatem implementuje i reprezentuje biznes, czyli funkcje, logikę, procesy i dane aplikacji. W zależności od aplikacji w klasach tej warstwy mogą być hermetyzowane:

- Dane biznesowe lub aplikacji różnych klas niestandardowych.
- Reguły logiki lub reguły biznesowe walidacji dla aplikacji.
- Proces lub przepływ pracy biznesu albo aplikacji, taki jak proces zamawiania i potok realizacji albo przepływ pracy wsparcia klienta.

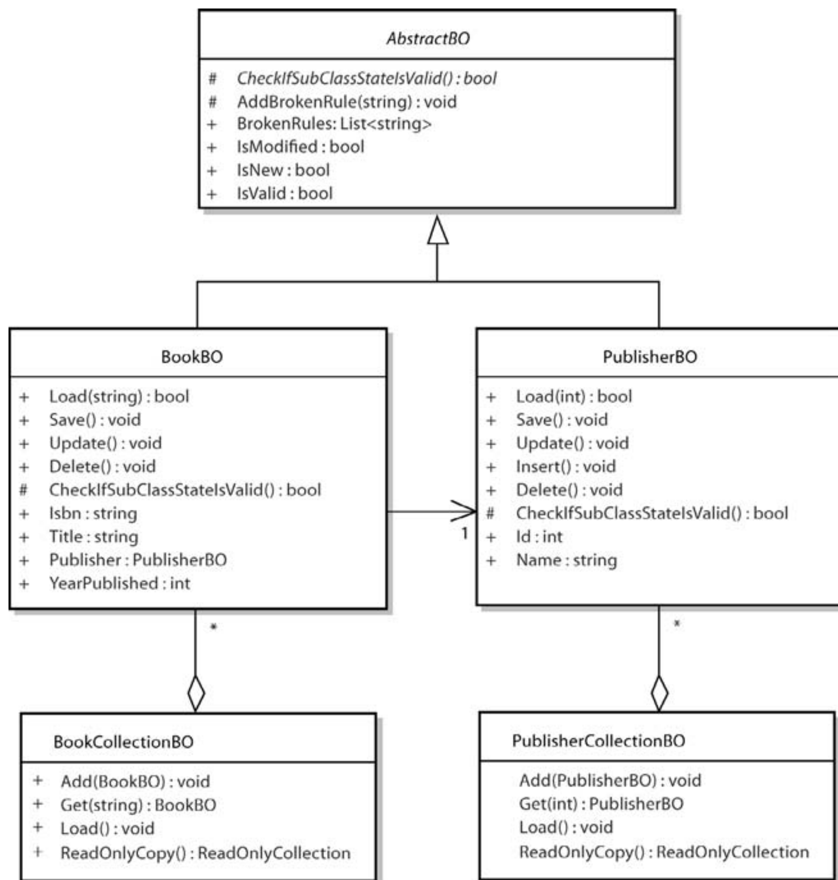
Usunięcie reguł biznesowych oraz logiki aplikacji z warstwy prezentacji sprawia, że formularze WWW stają się o wiele prostsze. Teraz mogą one zawierać wyłącznie znaczniki oraz obsługę zdarzeń interfejsu użytkownika. Wszystko inne jest zawarte w pozostałych klasach.

Na rysunku 11.6 pokazano część klas, których można użyć w obrębie warstwy biznesowej w przykładowej aplikacji Book Catalog.

Należy zauważyć, że obiekty biznesowe nie powinny być wyłącznie pojemnikami dla danych na temat obiektu. Powinny one również zawierać logikę. Mogą na przykład zawierać metody służące do uzyskiwania lub zapisywania swoich danych (za pomocą korzystania z klas z warstwy dostępu do danych), implementacji logiki aplikacji, autoryzacji dostępu oraz walidacji danych względem reguł biznesowych.

Kolejnym elementem (rysunek 11.6) wymagającym uwagi jest potrzeba istnienia dodatkowych obiektów biznesowych, które będą reprezentować zbiory obiektów biznesowych. Te klasy zbiorów biznesowych także nie powinny być jedynie pojemnikami dla danych, ale powinny zawierać również logikę. W przykładzie pokazanym na rysunku 11.6 klasy zbiorów biznesowych zawierają możliwość ładowania siebie samych ze źródła danych (za pomocą klas z warstwy dostępu do danych), a także aktualizacji lub dodawania siebie do źródła danych.

Jest prawdopodobne, że obiekty biznesowe aplikacji będą mieć część wspólnych funkcji. Te funkcje można umieścić we wspólnej klasie nadrzędnej. Na rysunku 11.6 jest nią klasa `AbstractBO`, której kod przedstawiono na listingu 11.1. Ma ona dane składowe, które śledzą, czy obiekt jest nowy, czy był modyfikowany oraz czy dane podklasy są prawidłowe względem reguł biznesowych. Ponieważ klasy, które korzystają z tych obiektów biznesowych mogą „chcieć” mieć możliwość sprawdzenia, które reguły zostały naruszone, klasa `AbstractBO` przechowuje zbiór naruszonych reguł, a także metody służące do dodawania oraz odczytywania zbiorów, które ze względu na prostotę przykładu zostały przedstawione jako lista



Rysunek 11.6. *Przykładowe obiekty biznesowe*

łańcuchów. Ponieważ każdy obiekt biznesowy „wie", jak określić, czy dane są prawidłowe, klasa `AbstractBO` definiuje abstrakcyjny składnik `CheckIfSubClassStateIsValid`, który jest zaimplementowany przez każdą z jej konkretnych podklas.

> W tym przykładzie każdy obiekt biznesowy jest odpowiedzialny za sprawdzanie swoich reguł biznesowych. Inne podejście, bardziej skomplikowane, ale zarazem o większych możliwościach adaptacji, polega na hermetyzowaniu tych reguł w obrębie ich własnej hierarchii obiektów. Do tych reguł będzie mógł stosować się jakiś typ odrębnej klasy zarządzania regułami. Więcej informacji o tym rodzaju podejścia znajduje się w książce Jimmy'ego Nilssona *Applying Domain-Driven Design and Patterns*.

Listing 11.1. *AbstractBO.cs*

```
using System;
using System.Data;
using System.Collections.Generic;

namespace ThreeLayer.Business
{
    /// <podsumowanie>
    /// Reprezentuje klasę nadrzędną dla wszystkich obiektów biznesowych
```

```
/// </podsumowanie>
public abstract class AbstractBO
{
   // Dane składowe
   protected const int DEFAULT_ID = 0;
   // Flagi wskazujące, czy obiekt jest nowy, czy był modyfikowany
   private bool _isNew = true;
   private bool _isModified = false;
   // Zbiór opisów wszystkich naruszonych reguł
   private List<string> _brokenRules = new List<string>();

   /// <podsumowanie>
   /// Każda klasa jest odpowiedzialna za sprawdzanie, czy jej
   /// stan (dane składowe) zawiera naruszone reguły biznesowe
   /// </podsumowanie>
   protected abstract bool CheckIfSubClassStateIsValid
   {
      get;
   }
   /// <podsumowanie>
   /// Podklasy potrzebują możliwości dodawania opisu naruszonej reguły
   /// </podsumowanie>
   protected void AddBrokenRule(string rule)
   {
      _brokenRules.Add(rule);
   }

   /// <podsumowanie>
   /// Zwróć opisy wszystkich naruszonych reguł
   /// </podsumowanie>
   public List<string> BrokenRules
   {
      get { return _brokenRules; }
   }

   /// <podsumowanie>
   /// Czy obiekt biznesowy jest poprawny
   /// </podsumowanie>
   public bool IsValid
   {
      get
      {
         _brokenRules.Clear();
         return CheckIfSubClassStateIsValid;
      }
   }

   /// <podsumowanie>
   /// Czy obiekt biznesowy był modyfikowany od czasu ostatniego zapisu
   /// </podsumowanie>
   protected bool IsModified
   {
      get { return _isModified; }
      set { _isModified = value; }
   }
```

```
        /// <podsumowanie>
        /// Czy ten obiekt biznesowy jest nowy, czy może
        /// zawiera dane, które już istnieją w bazie
        /// </podsumowanie>
        protected bool IsNew
        {
           get { return _isNew; }
           set { _isNew = value; }
        }

    }
}
```

Nie zamieszczam kodów dla wszystkich klas z rysunku 11.6 (chociaż można je pobrać z mojej witryny WWW pod adresem *http://www.randyconnolly.com/core*), jednak w listingu 11.2 prezentuję kod klasy biznesowej `PublisherBO`.

Listing 11.2. *PublisherBO.cs*

```
using System;
using System.Data;

using ThreeLayer.DataAccess;

namespace ThreeLayer.Business
{
   /// <podsumowanie>
   /// Obiekt biznesowy do przechowywania informacji o wydawcy
   /// </podsumowanie>
   public class PublisherBO: AbstractBO
   {
      // Dane składowe
      private int _id = DEFAULT_ID;
      private string _name = "";

      // Konstruktory
      public PublisherBO() { }
      public PublisherBO(int id, string name)
      {
         _id = id;
         _name = name;
         IsNew = false;
         IsModified = true;
      }

      /// <podsumowanie>
      /// Wypełnia obiekt danymi na podstawie przekazanego ID
      /// </podsumowanie>
      /// <zwraca>
      /// Wartość true, jeśli wypełnienie danymi powiodło się,
      /// w przeciwnym razie wartość false
      /// </zwraca>
      public bool Load(int id)
      {
         PublisherDA da = new PublisherDA();
         DataTable table = da.GetById(id);
```

```
    // Jeśli w tabeli nie ma danych, wypełnienie nie powiodło się
    if (table.Rows.Count == 0)
    {
        AddBrokenRule("Wydawca o numerze id=" + id +
            " nie został odnaleziony");
        return false;
    }

    // Przypisz pobrane dane danym składowym
    Id = (int)table.Rows[0]["PublisherId"];
    Name = (string)table.Rows[0]["PublisherName"];
    IsNew = false;

    // Sprawdź, czy pobrane dane są zgodne z regułami
    return IsValid;
}

/// <podsumowanie>
/// Zapisuje dane obiektu. Jeśli obiekt jest nowy,
/// dane zostaną wstawione; w przeciwnym razie, jeśli dane zmieniły się,
/// zostaną zaktualizowane
/// </podsumowanie>
public void Save()
{
    if (IsNew)
        Insert();
    else
        Update();
}

/// <podsumowanie>
/// Tylko aktualizuje dane
/// </podsumowanie>
public void Update()
{
    if (IsValid)
    {
        if (IsModified)
        {
            PublisherDA da = new PublisherDA();
            da.UpdatePublisher(Id, Name);
            IsModified = false;
        }
    }
}

/// <podsumowanie>
/// Tylko wstawia dane
/// </podsumowanie>
public void Insert()
{
    IsNew = true;
    if (IsValid)
    {
        PublisherDA da = new PublisherDA();
        da.InsertPublisher(Name);
    }
}
```

```
/// <podsumowanie>
/// Usuwa dane
/// </podsumowanie>
public void Delete()
{
   PublisherDA da = new PublisherDA();
   da.DeletePublisher(Id);
}

/// <podsumowanie>
/// Sprawdza, czy stan wewnętrzny obiektu
/// biznesowego jest prawidłowy
/// </podsumowanie>
protected override bool CheckIfSubClassStateIsValid
{
   get
   {
      bool valid = true;
      if (Name.Length == 0)
      {
         AddBrokenRule("Nazwa wydawcy nie może być pusta");
         valid = false;
      }
      if (Id < 0)
      {
         AddBrokenRule(
            "Id wydawcy nie może być mniejszy niż zero");
         valid = false;
      }
      // Oto przykład bardziej złożonej reguły
      if (IsNew)
      {
         // Sprawdź, czy tytuł już nie istnieje
         PublisherDA da = new PublisherDA();
         DataTable dt = da.GetByName(Name);
         if (dt.Rows.Count > 0)
         {
            AddBrokenRule("Nazwa wydawcy już istnieje");
            valid = false;
         }
      }
      return valid;
   }
}

// Właściwości

public int Id
{
   get { return _id; }
   set {
      _id = value;
      IsModified = true;
   }
}
```

```
        public string Name
        {
            get { return _name; }
            set {
                _name = value;
                IsModified = true;
            }
        }
    }
}
```

Ta klasa hermetyzuje dane pojedynczego wydawcy i metody niezbędne do jego ładowania, aktualizacji, wstawiania oraz usuwania (za pomocą klasy dostępu do danych podobnej do pokazanej pod koniec rozdziału 9.). Klasa biznesowa potrzebuje także możliwości określania, czy jej dane naruszają którąś z jej reguł biznesowych. Zabezpieczona właściwość `CheckIfSubClassStateIsValid` sprawdza, czy nazwa wydawcy nie jest pusta i czy ID wydawcy nie jest mniejsze od zera. Oprócz tego podczas dodawania wydawcy właściwość ta sprawdza, czy wydawca o tej samej nazwie już nie istnieje. Wszystkie naruszone reguły są dodawane do zbioru `BrokenRules` (zdefiniowanego w klasie `AbstractBO`).

> Wyjaśniając omawiane zagadnienia w powyższym przykładzie oraz w innych przykładach zamieszczonych w niniejszym rozdziale, znacząco uprościłem kod. W klasie powinno być na przykład więcej kodu sprawdzającego błędy, wsparcia dla transakcji oraz być może rozłączony mechanizm definiowania reguł.

Używanie przestrzeni nazw

Warto zauważyć, że w przykładowych listingach użyto słowa kluczowego języka C# `namespace`. Jest ono stosowane w celu organizowania klas za pomocą definiowania globalnie unikalnych definicji typów. Biblioteka klas platformy .NET korzysta z przestrzeni nazw, aby zagwarantować brak konfliktów między nazwami. Klasa `Image` istnieje na przykład w przestrzeniach nazw `System.Drawing` oraz `System.Web.UI.WebControls`. Kiedy na początku klas jest dodawana dyrektywa `using`, wskazuje ona, że odwołania będą dotyczyć klas z tej właśnie przestrzeni nazw bez podawania jej w pełni kwalifikowanej nazwy.

Jeśli zatem zachodzi potrzeba użycia klasy `PublisherBO` w klasie z kodem ukrytym formularza WWW, należy dodać odpowiednie odwołanie:

```
using ThreeLayer.Business;
…
PublisherBO pub = new PublisherBO();
```

W przeciwnym razie konieczne byłoby użycie w pełni kwalifikowanej nazwy:

```
ThreeLayer.Business.PublisherBO pub = new
ThreeLayer.Business.PublisherBO();
```

Programowe używanie obiektu biznesowego

Aby z obiektu biznesowego można było korzystać w formularzu WWW, można umieścić go w kontrolce `ObjectDataSource` albo użyć go programowo. Aby na przykład w kontrolce `GridView` wyświetlić zbiór wydawców, można powiązać programowo jej zbiór tylko do odczytu z właściwością kontrolki `DataSource`.

```
PublisherCollectionBO pubs = new PublisherCollectionBO();

grdPublishers.DataSource = pubs.ReadOnlyCopy();
grdPublishers.DataBind();
```

Przyjmijmy, że ta sama kontrolka `GridView` ma w każdym wierszu przycisk służący do wybierania wiersza. Kiedy użytkownik wybierze wiersz, zostaną wyświetlone dane wybranego wydawcy. Kod wykonujący to zadanie może wyglądać następująco:

```
// Określ ID wybranego wydawcy
int pubId = (int)grdPublishers.SelectedDataKey.Values[0];

// Utwórz obiekt biznesowy wydawcy i wczytaj jego dane
PublisherBO pub = new PublisherBO();
if (pub.Load(pubId))
{
   panPublisher.Visible = true;
   labId.Text = pub.Id.ToString();
   txtName.Text = pub.Name;
}
```

W powyższym przykładzie jest tworzony obiekt biznesowy wydawcy, który wczytuje dane odpowiadające wybranemu identyfikatorowi `PublisherId`. Jeśli wczytywanie danych zakończy się pomyślnie, wartości pól będzie można odczytywać za pośrednictwem właściwości obiektu biznesowego.

W celu aktualizacji danych w bazie można skorzystać z tej samej klasy biznesowej. Na poniższym przykładzie pokazano, jak obiekt klasy `PoblisherBO` jest tworzony i zapełniany danymi pochodzącymi z formularza WWW. Następnie obiekt ma sam siebie zaktualizować. Jeśli aktualizacja nie powiodła się na skutek naruszenia reguły biznesowej, w kontrolce formularza zostanie wyświetlony zbiór łańcuchów z regułami.

```
// Pobierz wartości ID oraz tytułu
int id = Convert.ToInt32(labId.Text);
string name = txtName.Text;

// Utwórz obiekt biznesowy wydawcy i postaraj się go zaktualizować
PublisherBO pub = new PublisherBO(id, name);
pub.Update();

// Sprawdź, czy aktualizacja udała się
if (!pub.IsValid)
{
   // Aktualizacja nie udała się, a zatem pokaż naruszone reguły biznesowe
   grdErrors.DataSource = pub.BrokenRules;
   grdErrors.DataBind();
}
```

Używanie obiektów biznesowych z kontrolką ObjectDataSource

Jak już pokazano w poprzednim rozdziale — używając kontrolki `ObjectDataSource`, można powiązać kontrolki danych z dowolna klasą. Rzeczywiście, za pomocą kontrolki `ObjectDataSource` stosunkowo łatwo można wyświetlić wydawców w klasie `PublisherCollectionBO`:

```
<asp:ObjectDataSource ID="dsBusiness" runat="server"
   TypeName="ThreeLayer.Business.PublisherCollectionBO"
   SelectMethod="ReadOnlyCopy" >
```

Używanie jednak obiektu biznesowego `PublisherId` z kontrolką `ObjectDataSource` rodzi pewne problemy. Niestety, sposób, w jaki zaprojektowano tę kontrolkę, może powodować trudności w używaniu jej z typowym, jednoelementowym obiektem biznesowym. Pierwszy problem, na jaki można się natknąć, polega na tym, że w przypadku używania kontrolki `ObjectDataSource` do wyświetlania obiektów biznesowych i manipulowania nimi korzystanie z własnych funkcji obiektu służących do ładowania, aktualizowania i wstawiania może być trudne. Podczas projektowania obiektu biznesowego zazwyczaj hermetyzuje się nie tylko jego dane, ale też funkcje służące do ładowania i zapisywania danych (używając klas pochodzących z warstwy dostępu do danych). Takie typowe rozwiązanie może powodować problemy podczas próby użycia tych funkcji w kontrolce `ObjectDataSource`.

Załóżmy na przykład, że istnieje formularz WWW z kontrolką `DetailsView`, która wyświetla i aktualizuje dane z klasy `PublisherBO`. Każdy obiekt tej klasy może zapełniać swoje dane składowe informacjami z bazy danych za pośrednictwem metody `Load`. Warto jednak przypomnieć, że metoda określona we właściwości `SelectMethod` kontrolki `ObjectDataSource` musi zwracać potrzebne dane, które zostaną wyświetlone w kontrolce. To oznacza, że jest potrzebna metoda, która zwraca zapełniony już obiekt klasy `PublisherBO`. Jest zatem potrzebna jakaś *inna* klasa, która zwróci potrzebny, zapełniony obiekt (taki typ klasy można nazwać ***adapterem***, z uwagi na wzorzec projektowy adaptera). Alternatywnie można dodać do obiektu biznesowego metodę statyczną, która zwraca zapełniony obiekt biznesowy, jak to pokazano poniżej:

```
public class PublisherBO: AbstractBO
{
   …
   public static PublisherBO RetrievePublisher(int pubId)
   {
      PublisherBO pub = new PublisherBO();
      if (pub.Load(pubId))
         return pub;
      else
         return null;
   }
}
```

Podobny problem zaistnieje w przypadku aktualizacji. Każdy obiekt klasy `PublisherBO` ma możliwość zapisywania lub aktualizacji bazy danych przy użyciu metody `Save` albo `Update`. Kontrolka `ObjectDataSource` oczekuje jednak, że jej określona metoda pobierze jako parametr obiekt klasy `PublisherBO` albo listę parametrów, które odpowiadają aktualizowanym wartościom danych. Tutaj też będzie potrzebna jakaś inna klasa z metodą aktualizującą, która

jako parametr pobiera obiekt klasy PublisherBO (i która po prostu wywołuje własną metodę aktualizującą obiektu biznesowego) albo konieczne będzie dodanie do obiektu biznesowego metody statycznej, która aktualizuje przekazany obiekt tej klasy.

```
public class PublisherBO: AbstractBO
{
   …
   public static void UpdatePublisher(PublisherBO pub)
   {
      pub.Update();
   }
}
```

Kontrolka ObjectDataSource może zatem być podobna do kontrolki pokazanej w poniższym przykładzie (korzysta ona z parametru łańcucha zapytania w celu określenia, którego wydawcę wyświetlić).

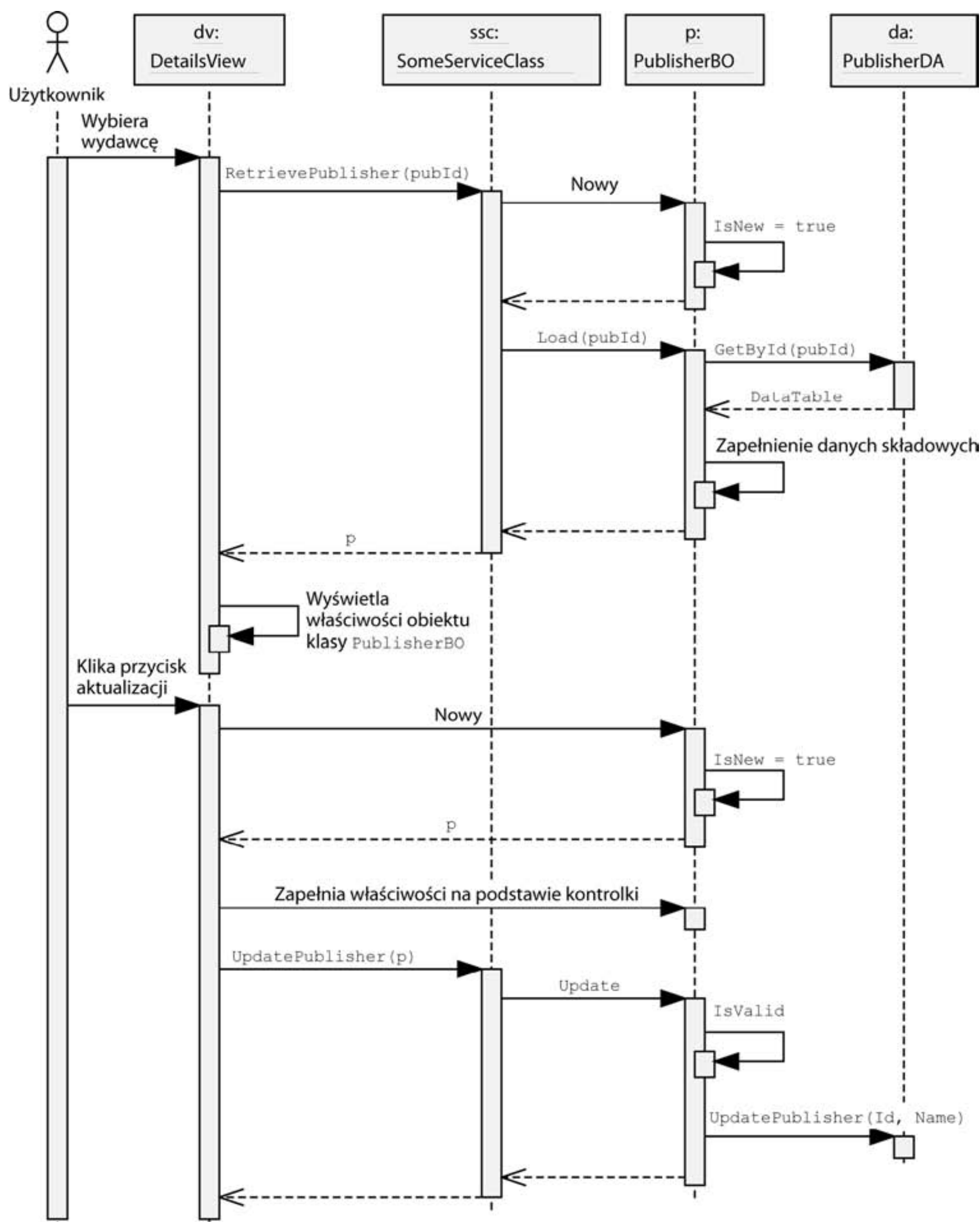
```
<asp:ObjectDataSource ID="dsBusiness" runat="server"
   TypeName="ThreeLayer.Business.PublisherBO"
   SelectMethod="RetrievePublisher"
   UpdateMethod="UpdatePublisher"
   DataObjectTypeName="ThreeLayer.Business.PublisherBO">

   <SelectParameters>
      <asp:QueryStringParameter Name="pubId"
         QueryStringField="id" />
   </SelectParameters>

</asp:ObjectDataSource>
```

W powyższym przykładzie należy także określić właściwość DataObjectTypeName kontrolki. Ta właściwość jest używana do określania nazwy klasy obiektu, z którego korzysta kontrolka w operacjach aktualizacji, wstawiania oraz usuwania.

W pokazanym przykładzie metody wybierania i aktualizacji zawierają się w obiekcie biznesowym. Można też utworzyć odrębną klasę adapterową zawierającą metody RetrievePublisher oraz UpdatePublisher, co zmieniałoby kontrolkę ObjectDataSource w następujący sposób:

```
<asp:ObjectDataSource ID="dsBusiness" runat="server"
TypeName="SomeAdapterClass"
SelectMethod="RetrievePublisher"
UpdateMethod="UpdatePublisher"
DataObjectTypeName="ThreeLayer.Business.PublisherBO">
```

Inny potencjalny problem związany z obiektami biznesowymi oraz kontrolką ObjectDataSource polega na tym, że wewnętrzna logika typowego obiektu biznesowego może nie integrować się dobrze z kontrolką. Zazwyczaj sposób postępowania z obiektami biznesowymi polega na przykład na użyciu niestandardowej klasy fabrycznej w celu tworzenia i dostosowywania różnych obiektów biznesowych w aplikacji. Kontrolka ObjectDataSource tworzy jednak wszystkie obiekty używane w aplikacji, korzystając z domyślnego konstruktora obiektu. Ponadto, podczas używania właściwości DataObjectTypeName w celu przekazania obiektu biznesowego do metody aktualizującej, wstawiającej lub usuwającej jest tworzona nowa instancja obiektu biznesowego, niektóre jej właściwości są zapełniane, po czym obiekt ten zostaje przekazany do właściwej metody (rysunek 11.7).



Rysunek 11.7. *Interakcja między kontrolką z listą, kontrolką ObjectDataSource oraz obiektem biznesowym*

Takie zachowanie może być przyczyną problemów w przypadku obiektów biznesowych, które używają wewnętrznej logiki, aby decydować, czy dane mają zostać zaktualizowane, czy też wstawione. Obiekt biznesowy może na przykład decydować o potrzebie aktualizacji istniejącego rekordu albo o wstawieniu nowego rekordu na podstawie stanu swojej właściwości `IsNew`, która ze szkodą dla zgodności z kontrolką `ObjectDataSource` jest ustawiana przez domyślnego konstruktora na wartość `true`.

Oczywiście, można przeprojektować obiekty biznesowe, aby lepiej współpracowały z kontrolką `ObjectDataSource`. W rzeczywistości należy tak postąpić w celu umożliwienia stronicowania i sortowania w obiektach biznesowych opartych na kolekcjach. Niektórzy jednak projektanci oprogramowania mogą uważać, że zmiana projektu warstwy biznesowej w celu dostosowania jej do potrzeb warstwy prezentacji jest niczym ciążąca nad nimi klątwa. W innych sytuacjach mogą istnieć obiekty biznesowe, których projektu nie da się zmienić bez szkody dla innych aplikacji. W takim przypadku należy utworzyć fasadę albo klasę adapterową, która udostępni kontrolce `ObjectDataSource` lepiej pasujący interfejs. Można oczywiście podjąć decyzję o całkowitej rezygnacji z używania kontrolki `ObjectDataSource` i przeprowadzać wiązanie oraz przetwarzanie danych w „staroświecki" sposób znany ze środowiska ASP.NET 1.1, który polega na korzystaniu z krótkich fragmentów kodu w plikach z kodem ukrytym.

Alternatywą do korzystania z modelu trójwarstwowego jest użycie modelu czterowarstwowego. Model czterowarstwowy ma wiele zalet, z których jedna polega na tym, że taki model jest potencjalnie łatwiejszy do zintegrowania z kontrolką `ObjectDataSource`.

Model czterowarstwowy

W podstawowym modelu trójwarstwowym warstwa biznesowa nie jest szczególnie spójna, ponieważ może hermetyzować dane i role biznesowe, a także logikę i procesy aplikacji. Aby uczynić warstwę biznesową bardziej spójną, niektórzy programiści umieszczają logikę i procesy aplikacji w warstwie prezentacji, co niestety często sprawia, że warstwa ta staje się bardzo chaotyczna i trudna w modyfikacji.

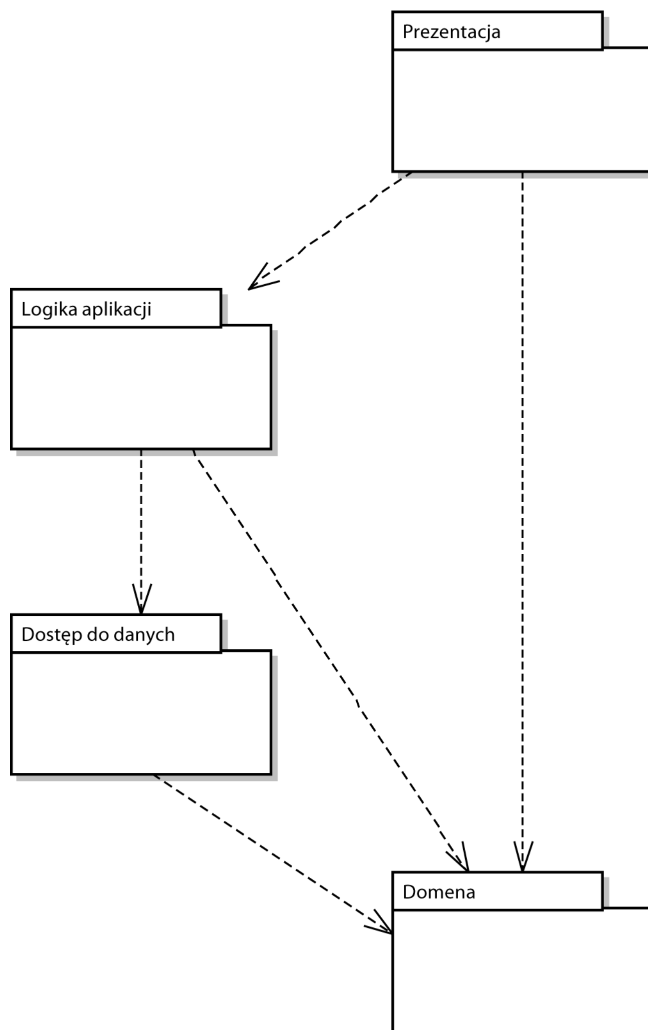
Projektowanie architektury czterowarstwowej

Lepsze rozwiązanie polega na odseparowaniu logiki i procesów aplikacji zarówno od warstwy prezentacji, jak i danych oraz operacji biznesowych, co pokazano na rysunku 11.8.

Na przedstawionym schemacie warstwy prezentacji oraz dostępu do danych są niemal takie same jak w schemacie trójwarstwowym. Różnica kryjąca się w modelu czterowarstwowym polega na sposobie, w jaki warstwa biznesowa została podzielona na dwie części: warstwę domeny oraz warstwę logiki aplikacji. Jest to zgodne z często spotykanym schematem polegającym na umieszczeniu tych dwóch rodzajów logiki (logiki domeny i logiki aplikacji, która jest czasami nazywana logiką przepływu pracy) w dwóch odrębnych warstwach.

Warstwa logiki aplikacji jest odpowiedzialna za koordynację i kontrolę logiki i procesów aplikacji, czyli sprawdza te zadania oprogramowania, które oprogramowanie ma wykonywać. Operacje w tej warstwie często są zbieżne z konkretnymi przypadkami użycia. Warstwa ta może być również nazywana warstwą obsługi (Fowler albo Nilsson), warstwą kontrolera, warstwą zarządzającą albo warstwą przepływu pracy biznesu (Microsoft).

Rysunek 11.8. *Model czterowarstwowy*



Obiekty warstwy logiki aplikacji nie powinny zawierać danych stanu dotyczących biznesu, mogą jednak zawierać dane stanu dotyczące procesów aplikacji. Większość pracy związanej z danymi biznesowymi jest przekazana ***warstwie domeny***, która jest odpowiedzialna za reprezentowanie danych i reguł biznesu lub aplikacji.

Ponieważ warstwa domeny zawiera właśnie te reguły i dane, znajduje się ona w centrum uwagi na wstępnym etapie projektowania aplikacji. Klasy tej warstwy często są nazywane ***encjami***. Chociaż klasy warstwy domeny mogą być zbieżne z tabelami w bazie danych (co można zauważyć na rysunku 11.10), nie jest to w żadnym wypadku konieczne. Zamiast tego klasy tej warstwy mają reprezentować koncepcje w domenie problemu aplikacji. Rzeczywiście, klasy domeny można było spotkać już w klasie `Customer`, w rozdziale 8.

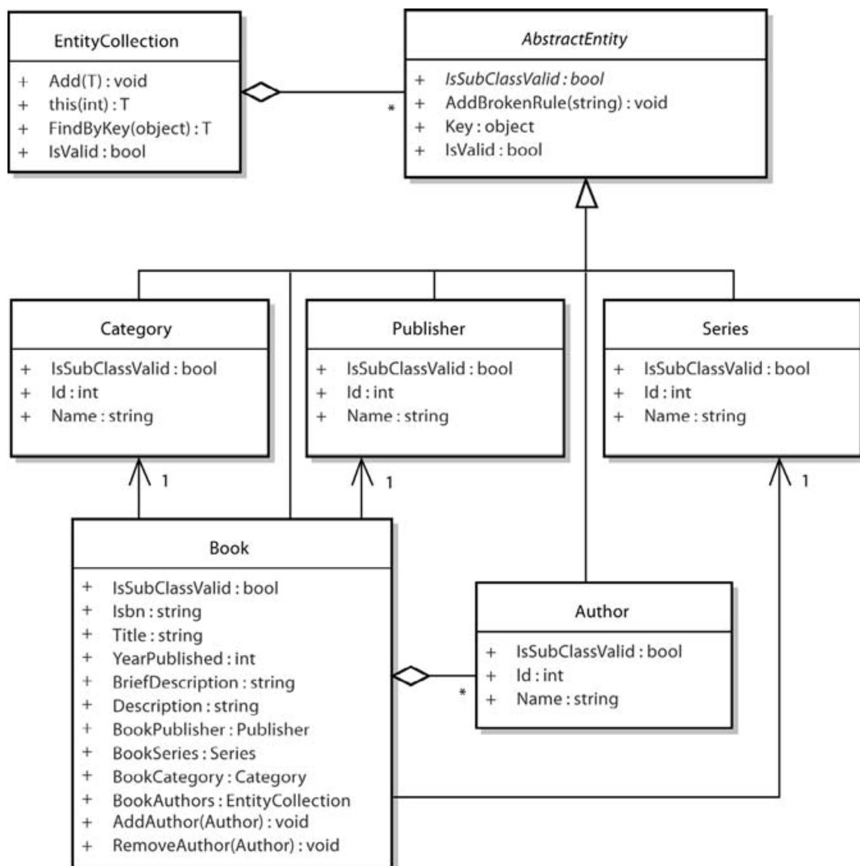
Ostatnią rzeczą, na którą należy zwrócić uwagę w czterowarstwowym modelu pokazanym na rysunku 11.8, jest fakt, że klasy warstwy domeny są używane przez pozostałe trzy warstwy. Jest tak dlatego, że klasy encji w tym modelu są używane w celu przechowywania

danych, gdy są one przekazywane między warstwami. Pełnią one zatem rolę, którą w modelach dwuwarstwowym oraz trójwarstwowym odgrywała klasa `DataTable`. Przypomnijmy, że w modelach dwuwarstwowym i trójwarstwowym różne klasy dostępu do danych umieszczały dane uzyskiwane z bazy danych w obiekcie klasy `DataTable` i zwracały je do zgłaszającego żądanie. W rezultacie użytkownik dowolnej klasy dostępu do danych musiał manipulować klasą ADO.NET `DataTable`, chociaż ona sama nie była klasą dostępu do danych.

Jako rozwiązanie alternatywne w modelu czterowarstwowym, pokazanym na rysunku 11.8, klasy dostępu do danych same tworzą i zapełniają klasy encji. W sytuacjach, w których są pobierane wielokrotne rekordy danych, klasy dostępu do danych tworzą wielokrotne instancje odpowiedniej klasy encji, a następnie umieszczają je w obrębie klasy `EntityCollection` — tej samej klasy pojemnikowej ogólnego zastosowania, która została utworzona w rozdziale 8.

Na rysunku 11.9 przedstawiono niektóre przykładowe klasy z warstwy domeny. Należy zauważyć, że każda klasa domeny jest podklasą klasy `AbstractEntity`, która jest podobna do klasy opisanej w rozdziale 8. i z którą ma części wspólne, jak na przykład `AbstractBO` — klasę bazową dla wszystkich klas biznesowych w modelu trójwarstwowym.

Rysunek 11.9.
Klasy warstwy domeny



Różne klasy domeny mają wspólną część zespołu funkcji, podobnie jak klasy obiektów biznesowych pokazane na rysunku 11.5. Główna różnica polega na tym, że klasy domeny nie mogą wchodzić w interakcję z bazą danych. W rezultacie jest usprawiedliwione uważanie tych klas za odpowiednik czegoś, co niektórzy programiści znający język Java nazywają ***obiektem transferu danych***. Określenie to odnosi się do klasy, która zawiera wyłącznie dane składowe z właściwościami służącymi uzyskiwaniu dostępu do danych. Najważniejsza różnica między obiektem transferu danych a naszą klasą domeny kryje się w tym, że klasa domeny zawiera również zachowania służące do sprawdzania i walidacji reguł biznesowych.

Na koniec warto zauważyć związki zachodzące między różnymi klasami domen na rysunku 11.9. Podobnie jak w przypadku obiektów biznesowych w modelu trójwarstwowym, encje w modelu czterowarstwowym reprezentują koncepcje lub rzeczy w domenie problemu aplikacji. W przykładowej aplikacji z książkami książka ma wydawcę, serię, kategorię i kilku autorów. Encja `Book` ma zatem encję `Publisher`, encję `Series`, encję `Category` oraz zbiór encji `Author`. Chociaż związki te są odzwierciedleniem modelu danych w bazie, nie są one takie same. Encja książki ma na przykład obiekt encji `Publisher`, podczas gdy kluczem obcym rekordu `Books` jest pole `PublisherId`. Encja książki ma zbiór obiektów encji `Author`, gdy tymczasem autorzy danej książki zawierają się w tabelach `AuthorBooks` oraz `Authors`.

Opisywanie najlepszego sposobu projektowania modelu domeny wykracza poza zakres niniejszego rozdziału. Czytelnicy chcący poszerzyć swoje wiadomości w przedstawionym temacie mogą zajrzeć do książek Evansa, Fowlera i Nilssona wymienionych w części „Odnośniki”, pod koniec tego rozdziału.

Modyfikowanie warstwy dostępu do danych

W modelu czterowarstwowym klasy encji biznesowej są używane w celu przenoszenia danych między warstwami. W rezultacie należy wprowadzić pewne zmiany w klasach dostępu do danych, które zostały utworzone pod koniec poprzedniego rozdziału. W dalszym ciągu są w nich zaimplementowane podstawowe funkcje CRUD (tworzenia, odczytywania, aktualizacji oraz usuwania), a także dziedziczą one z bazowej klasy abstrakcyjnej. Różnica występująca w nowej wersji klas polega na tym, że klasy te w roli parametrów pobierają encje i zwracają dane w tych samych encjach.

Dwuwarstwowa klasa `BookDA` (której wersja została przedstawiona pod koniec rozdziału 9.) zawierała na przykład metodę `UpdateBook`. Metodzie tej przekazywano wszystkie wartości danych, które powinny być zachowane w pokazany poniżej sposób:

```
public void UpdateBook(string isbn, string title,
   int publisherId, int categoryId,
   int seriesId, int yearPublished,
   string briefDescription, string description)
```

W podejściu czterowarstwowym metodzie tej jest po prostu przekazywany wypełniony obiekt encji `Book`:

```
public void Update(Book book)
```

W podobny sposób zostanie zmieniona także metoda służąca do odczytywania danych. W poprzedniej klasie `BookDA` istniała następująca metoda, która zwracała obiekt klasy `DataTable` zawierający jeden wiersz danych:

```
public DataTable GetBookByIsbn(string isbn)
```

Nowa wersja zwraca po prostu wypełnioną encję `Book`:

```
public Book GetBookByIsbn(string isbn)
```

Przejście na encje powinno mieć wpływ na uproszczenie warstwy dostępu do danych. Modyfikacja metody `GetAll` wiąże się jednak z pewnym stopniem złożoności, o który zostanie wzbogacona nasza warstwa. W warstwie dostępu do danych w modelu dwuwarstwowym metoda `GetAll` była zaimplementowana w klasie bazowej `AbstractDA`. Dla odpowiedniej instrukcji `SELECT` zwracała ona po prostu wypełniony obiekt klasy `DataTable`:

```
public DataTable GetAll()
{
   string sql = SelectStatement;
   return GetDataTable(sql, null);
}
```

Oto na czym polega problem: co takiego zwraca ta metoda w modelu czterowarstwowym? Najlepiej, gdyby zwracała silnie typizowaną wersję klasy `EntityCollection`, która zawiera odpowiednie jednostki encji. Kłopot jednak w tym, że ta metoda została zdefiniowana w klasie bazowej. Ponieważ stara wersja tylko zwracała wypełniony obiekt klasy `DataTable`, typ zwracanych danych nie musiał być jej znany. Obecnie jednak, jeśli ma być zwracany silnie typizowany zbiór, klasa bazowa musi „wiedzieć", jaki typ danych powinien zostać zwrócony. Można oczywiście zaimplementować metodę `GetAll` we wszystkich klasach dostępu do danych; spowoduje to jednak konieczność dodania niemal identycznego kodu do każdej z tych klas.

Rozwiązanie problemu polega na użyciu typów ogólnych języka C#, omówionych w rozdziale 8. łącznie ze ***wzorcem Template Method***. Wzorzec ten jest często stosowany w celu eliminacji powtarzającego się w podklasach kodu i polega na zdefiniowaniu głównych kroków algorytmu w klasie bazowej, a następnie zaimplementowaniu w podklasach tylko tych części algorytmu, które są różne w różnych podklasach.

Spójrzmy na algorytm, który należy zaimplementować w metodzie `GatAll`. Podstawowe kroki algorytmu są następujące:

1. Utworzenie obiektu połączenia
2. Skonfigurowanie łańcucha polecenia wyboru
3. Utworzenie i skonfigurowanie obiektu polecenia
4. Otwarcie połączenia
5. Uruchomienie czytnika danych
6. Utworzenie obiektu ze zbiorem encji
7. Dla każdego rekordu w czytniku danych:

8.1. Pobranie wartości pól z rekordu

8.2. Utworzenie odpowiedniej encji obiektu biznesowego

8.3. Wypełnienie encji wartościami pól

8.4. Dodanie encji do zbioru encji

9. Zamknięcie czytnika oraz połączenia

10. Zwrócenie kolekcji encji

Które z powyższych kroków są różne dla różnych podklas dostępu do danych? Są to kroki 2. oraz kroki od 8.1 do 8.3. Wszystkie pozostałe kroki algorytmu są stałe i nie zależą od pobieranych danych. W istniejącej klasie `AbstractDA` zmieniający się krok 2. został obsłużony w następujący sposób:

```
string sql = SelectStatement;
```

W rozdziale 9. właściwość `SelectStatement` została zdefiniowana w klasie `AbstractDA` oraz zaimplementowana we wszystkich podklasach. Jest to przykład wzorca Template Method! Wszystko, czego jeszcze potrzeba, to zdefiniowanie abstrakcyjnej metody w klasie `AbstractDA`, która wykona kroki 8.1, 8.2 i 8.3, a następnie zaimplementowanie jej we wszystkich podklasach. Ponieważ ta metoda jest odpowiedzialna za tworzenie pojedynczej encji oraz wypełnienie jej wartościami pól bieżącego rekordu, należy zdefiniować ją następująco:

```
protected abstract AbstractEntity CreateAndFillEntity(DbDataReader
   reader)
```

Jak może wyglądać taka metoda, gdy zostanie już zaimplementowana? Na poniższym przykładzie pokazano, jak może ona wyglądać w klasie dostępu do danych na temat wydawców.

```
protected override AbstractEntity CreateAndFillEntity(
   DbDataReader recordJustRead)
   {
   // Pobierz wartości rekordów i umieść je
   // w tymczasowych zmiennych
   int id = (int)recordJustRead["PublisherId"];
   string name = (string)recordJustRead["PublisherName"];

   // Utwórz obiekt i zapełnij na podstawie danych
   Publisher pub = new Publisher(id, name);
   return pub;
}
```

> W celu uproszczenia powyższego fragmentu kodu, a także innych fragmentów zamieszczonych w niniejszym rozdziale, pominięto w nich obsługę wyjątków oraz inne szczegóły. Pełne wersje są dostępne na mojej witrynie pod adresem *http://www.randyconnolly.com/core*.

Gdy przykładowa wersja szablonu metody jest już zaimplementowana, można zacząć implementację algorytmu `GetAll` klasy `AbstractDA`. Aby było łatwiej, zajmiemy się krokami od 5. do 10.

```
EntityCollection<AbstractEntity> collection;
Collection = new EntityCollection<AbstractEntity>();

DbDataReader reader = cmd.ExecuteReader();
while (reader.Read())
{
   collection.Add( CreateAndFillEntity(reader) );
}
reader.Close();
conn.Close();
return(collection);
```

Ponieważ jest zwracany zbiór, a nie obiekt klasy `DataTable`, nie ma potrzeby używania klasy `DataAdapter` w celu jego wypełnienia. Zamiast tego można skorzystać z o wiele szybszej klasy `DataReader`. Wystarczy po prostu przebiec w pętli przez zwrócony czytnik danych, aby każda podklasa odczytała rekord, utworzyć i wypełnić odpowiednią encję, a następnie dodać ją do zbioru.

W poprzednim przykładzie kryje się jednak pewien problem. W rozdziale 8. klasa `EntityCollection` została zdefiniowana za pomocą typów ogólnych, aby była zbiorem silnie typizowanym, a w przytoczonym przykładzie jest zwracany zbiór obiektów klasy `AbstractEntity`. Oznacza to, że każda klasa, która przetwarza wspomniany zbiór, musi rzutować encje na odpowiedni typ w pokazany poniżej sposób:

```
PublisherDA dao = new PublisherDA();
EntityCollection<AbstractEntity> collection = dao.GetAll();
Publisher pub = (Publisher)collection[0];
```

Ponieważ klasa `EntityCollection` obsługuje typy ogólne, należy zmodyfikować klasę `AbstractDA` w taki sposób, aby również korzystała z tych typów. Najpierw można zmienić jej definicję klasy, dzięki czemu będzie akceptować ograniczony typ parametru, który jest podobny do parametru użytego w klasie `EntityCollection` w rozdziale 8.:

```
public abstract class AbstractDA<T> where T : AbstractEntity
```

W klasie `AbstractDA` należy użyć parametru typu w miejscach, w których wcześniej następowało odwołanie do klasy `AbstractEntity`. Definicja klasy abstrakcyjnej `CreateAndFillEntity` zmieni się zatem następująco:

```
protected abstract T CreateAndFillEntity(DbDataReader reader);
```

Tworzenie zbioru (czyli krok 5. algorytmu) również ulegnie zmianie:

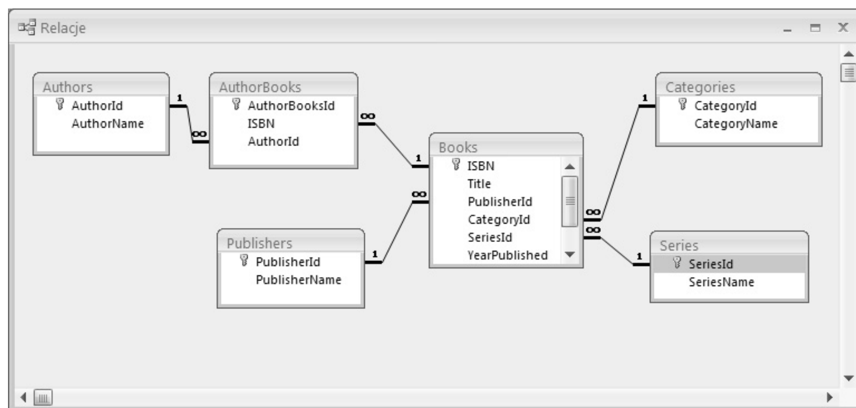
```
EntityCollection<T> collection = new EntityCollection<T>();
```

Na koniec należy zmodyfikować wszystkie podklasy, aby udostępniały odpowiednią nazwę klasy encji za każdym razem, gdy w klasie `AbstractDA` zostanie użyty parametr `T`:

```
public class PublisherDA: AbstractDA<Publisher>
{
   protected override Publisher CreateAndFillEntity(…)
   {
      …
   }
   …
}
```

Tworzenie złożonej encji domeny

Jak widać, do zadań szablonu metody CreateAndFillEntity należy tworzenie odpowiedniego typu encji, pobieranie wartości rekordu, a następnie zapełnianie encji tymi wartościami. Przykład takiej metody dla klasy PublisherDA był całkiem prosty. Wersja metody dla klasy BooksDA musi być jednak nieco bardziej skomplikowana, ponieważ encja Book zawiera nie tylko proste typy danych, ale też inne encje biznesowe (rysunek 11.9). Obsługa encji Publisher, Category i Series należących do encji Book jest łatwa, ponieważ tabela Book znajduje się w relacji „wiele do jednego" z tabelami Publisher, Category oraz Series (rysunek 11.10).



Rysunek 11.10. *Relacje między tabelami*

Jeśli pola dla każdej z tych dodatkowych tabel zostaną dołączone do instrukcji SELECT dotyczącej książek (tak jak w następnym przykładzie), będzie można utworzyć i wypełnić encje Publisher, Category oraz Series po odczytaniu rekordu książki.

```
protected override string SelectStatement
{
   get {
      return "SELECT ISBN,Title,YearPublished,BriefDescription,
      Description,PublisherName,Books.PublisherId As
      BookPublisherId,SeriesName,Books.SeriesId As
      BookSeriesId,CategoryName,Books.CategoryId As
      BookCategoryId FROM Series INNER JOIN
      (Publishers INNER JOIN (Categories INNER JOIN
      Books ON Categories.CategoryId = Books.CategoryId)
      ON Publishers.PublisherId = Books.PublisherId)
      ON Series.SeriesId = Books.SeriesId";
   }
}
```

Obsługa autorów książek jest nieco bardziej skomplikowanym zagadnieniem, ponieważ tabele Books oraz Authors wiąże relacja „wiele do wielu". Metoda GetAuthors klasy AuthorDA jest używana w celu uzyskania zbioru obiektów klasy Author na podstawie bieżącej wartości pola ISBN. Każda encja Author w tym zbiorze jest następnie dodawana do encji Book w sposób pokazany poniżej:

```
protected override Book CreateAndBuildEntity(DbDataReader
   recordJustRead)
{
   // Pobierz wartości rekordów i umieść je
   // w tymczasowych zmiennych
   string isbn = (string)recordJustRead["ISBN"];
   string title = (string)recordJustRead["Title"];
   int yearPub = (int)recordJustRead["YearPublished"];
   string brief = (string)recordJustRead["BriefDescription"];
   string desc = (string)recordJustRead["Description"];
   string pubName = (string)recordJustRead["PublisherName"];
   int pubId = (int)recordJustRead["BookPublisherId"];
   string seriesName = (string)recordJustRead["SeriesName"];
   int seriesId = (int)recordJustRead["BookSeriesId"];
   string catName = (string)recordJustRead["CategoryName"];
   int catId = (int)recordJustRead["BookCategoryId"];

   // Utwórz i zapełnij obiekt na podstawie danych
   Book book = new Book(isbn, title, yearPub, brief, desc);

   // Utwórz encje potomne
   Publisher publisher = new Publisher(pubId, pubName);
   Category category = new Category(catId, catName);
   Series series = new Series(seriesId, seriesName);

   // Dodaj encje potomne do książki
   book.BookPublisher = publisher;
   book.BookCategory = category;
   book.BookSeries = series;

   // Pobierz wszystkich autorów danej książki
   AuthorDA dao = new AuthorDA();
   EntityCollection<Author> authors =
   dao.GetAuthorsForIsbn(isbn);

   // Dodaj każdego autora do książki
   foreach (Author author in authors)
   {
      book.AddAuthor(author);
   }
   return book;
}
```

Metoda `GetAuthorsForIsbn` klasy `AuthorDA` pobiera tabele `AuthorBooks` oraz `Author`, a zwraca listę encji `Author` zgodnych z przekazaną wartością pola `ISBN`.

```
public EntityCollection<Author> GetAuthorsForIsbn(string isbn)
{
   EntityCollection<Author> collection =
     new EntityCollection<Author>();

   using (DbConnection conn =
       DatabaseActions.Factory.CreateConnection())
   {
      conn.ConnectionString =
         DatabaseActions.ConnectionSetting.ConnectionString;
      conn.Open();
```

```
            DbCommand cmd = DatabaseActions.Factory.CreateCommand();
            cmd.Connection = conn;
            cmd.CommandText =
               "SELECT AuthorBooks.AuthorId,AuthorName,ISBN
                 FROM Authors INNER JOIN AuthorBooks ON
                 Authors.AuthorId = AuthorBooks.AuthorId ";
            cmd.CommandText += " WHERE ISBN=@ISBN";
            cmd.CommandType = CommandType.Text;

            // Dodaj parametry
            cmd.Parameters.Add(DatabaseActions.MakeParameter(
               "@ISBN",isbn));

            DbDataReader reader = cmd.ExecuteReader();
            if (reader != null)
            {
               while (reader.Read())
               {
                  // Utwórz encję autora i dodaj ją do zbioru
                  collection.Add(CreateAndBuildEntity(reader));
               }
               reader.Close();
               conn.Close();
            }
         }
         return collection;
      }
```

Na koniec należy w podobny sposób zmodyfikować metody Update oraz Insert klasy BookDA, aby tabele potomne także podlegały aktualizacji.

Tworzenie warstwy logiki aplikacji

Teraz, gdy zostały już rozpatrzone niektóre klasy z warstwy encji biznesu oraz warstwy dostępu do danych, można utworzyć warstwę logiki aplikacji. Przypomnę, że ta warstwa powinna hermetyzować niezbędne w aplikacji funkcje i procesy i w związku z tym odzwierciedla funkcje określone w przypadkach użycia. Metody klas w tej warstwie mają zazwyczaj niewielkie rozmiary, ponieważ na ogół przekazują większość swoich funkcji odpowiednim warstwom domeny i dostępu do danych.

Załóżmy, że jest projektowana warstwa aplikacji dla aplikacji, która będzie używana nie tylko z encjami pokazanymi na rysunku 11.9, ale też z innymi encjami, takimi jak na przykład Customer, CustomerAddress, Order, Payment, CreditCard oraz niektórymi innymi klasami mającymi związek ze składaniem zamówienia bądź klientem. Można podjąć decyzję, że warstwa aplikacji będzie oparta na trzech klasach, które hermetyzują trzy podstawowe podsystemy funkcji aplikacji: BookCatalogLogic, CustomerServiceLogic oraz OrderLogic.

Po przeanalizowaniu przypadków użycia można dojść do wniosku, że w klasie BookCatalogLogic będą potrzebne następujące funkcje:

- Pobranie listy wszystkich wydawców
- Pobranie listy wszystkich kategorii książek
- Pobranie listy wszystkich serii książek
- Pobranie listy wszystkich książek
- Pobranie książki na podstawie jej numeru ISBN
- Pobranie listy wszystkich książek od określonego wydawcy
- Pobranie listy wszystkich książek z określonej serii
- Pobranie listy wszystkich książek z określonej kategorii
- Aktualizacja książki

Jeśli są to jedyne potrzebne funkcje, klasę `BookCatalogLogic` można zaprojektować w sposób pokazany na rysunku 11.11.

Rysunek 11.11. *Projekt warstwy aplikacji*

> Przedstawiona warstwa logiki aplikacji jest przede wszystkim warstwą usługową, ponieważ nie zawiera żadnych przykładów bardziej złożonej logiki aplikacji ani procesów. Jeśli w niniejszym rozdziale pokazano by klasę `OrderLogic`, byłoby widocznych więcej metod zorientowanych na procesy, takich jak na przykład `CheckOutPipeline` albo `MoveProductFromDistributorToInventory`.

Jak już wspomniano, klasy w warstwie logiki aplikacji prawie w ogóle nie zawierają stanu i przekazują większość swoich zadań innym warstwom. Na listingu 11.3 przedstawiono przykładową implementację wymienionych powyżej funkcji w klasie `BookCatalogLogic`.

Listing 11.3. *BookCatalogLogic.cs*

```
using System;
using System.Data;
using System.Collections.Generic;

using FourLayer.BusinessEntity;
using FourLayer.DataAccessObject;

namespace FourLayer.ApplicationLogic
{
   /// <podsumowanie>
   /// Obsługuje całość logiki aplikacji dla katalogu książek
   /// </podsumowanie>
   public class BookCatalogLogic
   {
      // -------------------------------------------------
      // Metody związane z książkami
      // -------------------------------------------------
      public static EntityCollection<Book> GetAllBooks()
      {
         BookDAO dao = new BookDAO();
         return dao.GetAll();
      }

      public static Book GetBookByIsbn(string isbn)
      {
         BookDAO dao = new BookDAO();
         Book b = dao.GetByKey(isbn);
         return b;
      }

      public static EntityCollection<Book>
         GetBooksByPublisher(Publisher pub)
      {
         BookDAO dao = new BookDAO();
         return dao.GetByCriteria("PublisherId", "=", pub.Id);
      }

      public static EntityCollection<Book> GetBooksByCategory(
         int catId)
      {
         BookDAO dao = new BookDAO();
         return
            dao.GetByCriteria("Books.CategoryId", "=", catId);
      }

      public static EntityCollection<Book>
         GetBooksBySeries(int seriesId)
      {
         BookDAO dao = new BookDAO();
         return
            dao.GetByCriteria("Books.SeriesId", "=", seriesId);
      }
```

```
        public static void UpdateBook(Book book)
        {
           if (book.IsValid)
           {
              BookDAO dao = new BookDAO();
              dao.Update(book);
           }
        }

        // --------------------------------------------------
        // Metody związane z wydawcami, seriami, kategoriami i autorami
        // --------------------------------------------------
        public static EntityCollection<Publisher>
           GetAllPublishers()
        {
           PublisherDAO dao = new PublisherDAO();
           return dao.GetAll();
        }

        public static EntityCollection<Series> GetAllSeries()
        {
           SeriesDAO dao = new SeriesDAO();
           return dao.GetAll();
        }

        public static EntityCollection<Category> GetAllCategories()
        {
           CategoryDAO dao = new CategoryDAO();
           return dao.GetAll();
        }
     }
  }
```

Użycie architektury w warstwie prezentacji

Teraz, gdy infrastruktura warstw została już utworzona, można ją zastosować w formularzach WWW. W przykładach od 11.1 do 11.5 użyto tych warstw w celu utworzenia strony *Bookportal.aspx*, pokazanej na rysunku 11.12. Strona ta składa się z czterech kontrolek `GridView` i jednej kontrolki `DetailsView`. Wszystkie interakcje z danymi odbywają się za pośrednictwem pewnej liczby kontrolek `ObjectDataSource`, które komunikują się z warstwą logiki aplikacji.

Łącza w lewej kolumnie służą do zmieniania listy książek widocznej w kontrolce `GridView` u góry prawej kolumny. Wybranie książki z listy spowoduje wyświetlenie w kontrolce `DetailsView`, która znajduje się w prawej dolnej kolumnie, pełnych informacji o książce. Kontrolka ta umożliwia również edycję książek przez użytkownika, co pokazano na rysunku 11.12.

Rysunek 11.12. *BookPortal.aspx*

Przykład 11.1. Konfigurowanie kontrolek ObjectDataSource

W pierwszym ćwiczeniu zostaną skonfigurowane kontrolki `ObjectDataSource`, które będą wchodzić w interakcję z klasą `BookCatalogLogic`. Reszta strony to przede wszystkim dodatkowe znaczniki formatowania.

1. Utwórz formularz WWW o nazwie *BookPortal.aspx*.
2. Dodaj do formularza następujące kontrolki `ObjectDataSource`. Kontrolki te są używane w kontrolkach `GridView`.

```
<asp:ObjectDataSource ID="dsBooks" runat="server"
   TypeName="FourLayer.ApplicationLogic.BookCatalogLogic" />

<asp:ObjectDataSource ID="dsSeries" runat="server"
   TypeName="FourLayer.ApplicationLogic.BookCatalogLogic"
   SelectMethod="GetAllSeries" />
```

```
<asp:ObjectDataSource ID="dsPublishers" runat="server"
   TypeName="FourLayer.ApplicationLogic.BookCatalogLogic"
   SelectMethod="GetAllPublishers" />

<asp:ObjectDataSource ID="dsCategories" runat="server"
   TypeName="FourLayer.ApplicationLogic.BookCatalogLogic"
   SelectMethod="GetAllCategories" />
```

3. Utwórz dodatkową kontrolkę `ObjectDataSource`.

```
<asp:ObjectDataSource ID="dsBookSingle" runat="server"
   TypeName="FourLayer.ApplicationLogic.BookCatalogLogic"
   DataObjectTypeName="FourLayer.BusinessEntity.Book"
   SelectMethod="GetBookByIsbn"
   UpdateMethod="UpdateBook">

   <SelectParameters>
      <asp:ControlParameter ControlID="grdBooks"
         Name="isbn" Type="string"
         PropertyName="SelectedDataKey.Values[0]" />
   </SelectParameters>
</asp:ObjectDataSource>
```

Ta kontrolka jest używana w kontrolce `DetailsView`, która wyświetla pełne dane pojedynczej książki. Warto zauważyć, że jej metoda wybierająca książkę (`GetBooksByIsbn`) wymaga parametru łańcuchowego o nazwie `isbn` zawierającego numer ISBN książki, której dane mają zostać pobrane. Kontrolka zapełnia ten parametr na podstawie wartości ISBN wiersza wybranego w kontrolce `GridView` (która zostanie jeszcze zdefiniowana). Właściwość `DataObjectTypeName` także musi zostać ustawiona, ponieważ określonej metodzie aktualizującej jest przekazywany wypełniony obiekt klasy `Book`.

> W ćwiczeniach jedynie zarysowano kontrolki oraz ich funkcje. Wybór stylu kontrolek pozostawiono Czytelnikowi. Ukończoną wersję strony, łącznie z nadanym stylem, można pobrać z mojej witryny pod adresem *http://www.randyconnolly.com/core*.

Przykład 11.2. Dodawanie kontrolek GridView do formularza WWW

Po zdefiniowaniu kontrolek, które będą wchodzić w interakcję z warstwą logiki aplikacji można skonfigurować interfejs użytkownika formularza WWW. W poniższym przykładzie do formularza zostaną dodane cztery kontrolki `GridView`.

1. Dodaj do formularza kontrolkę `GridView`, która będzie wyświetlać listę książek.

```
<asp:GridView ID="grdBooks" runat="server"
   DataSourceID="dsBooks" DataKeyNames="Isbn"
   EmptyDataText="No books found"
      AutoGenerateColumns="False" >

<Columns>
   <asp:CommandField
      SelectImageUrl="images/btn_select.gif"
      ButtonType="Image" ShowSelectButton="true" />
```

```
      <asp:BoundField DataField="Isbn" HeaderText="Isbn" />
      <asp:BoundField DataField="Title" HeaderText="Tytuł" />
      <asp:BoundField DataField="YearPublished"
         HeaderText="Rok" />
      </Columns>
   </asp:GridView>
```

2. Dodaj do formularza kontrolkę `GridView`, która będzie wyświetlać listę wszystkich możliwych wydawców książek.

```
<asp:GridView ID="grdPublisher" runat="server"
   DataSourceID="dsPublishers" DataKeyNames="Id"
   AutoGenerateColumns="False"
   OnSelectedIndexChanged=
      "grdPublisher_SelectedIndexChanged" >

   <Columns>
      <asp:TemplateField HeaderText="Publishers">
         <ItemTemplate>
            <asp:LinkButton ID="btnSelectPublisher"
               runat="server"
               Text='<%# Eval("Name") %>'
               CommandName="Select" />
         </ItemTemplate>
      </asp:TemplateField>
   </Columns>
</asp:GridView>
```

Należy zauważyć, że w kontrolce `GridView` użyto funkcji obsługi zdarzenia wyboru. Kiedy użytkownik wybierze wydawcę, funkcja obsługi zdarzenia wyboru (która zostanie jeszcze zdefiniowana) zmienia książki wyświetlane w kontrolce `GridView` na takie książki, których wydawca jest zgodny z wybranym wydawcą.

W omawianym przykładzie kolumna wyboru w kontrolce `GridView` nie została wyodrębniona. Zamiast tego w kontrolce `GridView` znajduje się jedna kolumna, w której nazwy wydawców są łączami. Łącza te działają jak przyciski wyboru. Niestety, nie można użyć kolumny `CommandField`, ponieważ w każdym wierszu jest potrzebna inna właściwość `SelectText`. Taki efekt można osiągnąć, używając kolumny `TemplateField` zawierającej obiekty klasy `LinkButton` z właściwością `CommandName` ustawioną na wartość `Select`, co pokazano w przykładowym kodzie.

3. Dodaj do formularza kontrolkę `GridView`, która będzie wyświetlać listę możliwych nazw serii książek.

```
<asp:GridView ID="grdSeries" runat="server"
   DataSourceID="dsSeries" DataKeyNames="Id"
   AutoGenerateColumns="False"
   OnSelectedIndexChanged="grdSeries_SelectedIndexChanged" >
   <Columns>
      <asp:TemplateField HeaderText="Serie">
         <ItemTemplate>
            <asp:LinkButton ID="btnSelectSeries" runat="server"
               Text='<%# Eval("Name") %>'
               CommandName="Select" />
         </ItemTemplate>
      </asp:TemplateField>
   </Columns>
</asp:GridView>
```

4. Dodaj do formularza kontrolkę, która będzie wyświetlać listę możliwych nazw kategorii książek.

```
<asp:GridView ID="grdCategories" runat="server"
   DataSourceID="dsCategories" DataKeyNames="Id"
   AutoGenerateColumns="False"
   OnSelectedIndexChanged="grdCategory_SelectedIndexChanged">

   <Columns>
      <asp:TemplateField HeaderText="Kategorie">
         <ItemTemplate>
            <asp:LinkButton ID="btnSelectCategory"
               runat="server"
               Text='<%# Eval("Name") %>'
               CommandName="Select" />
         </ItemTemplate>
      </asp:TemplateField>
   </Columns>
</asp:GridView>
```

5. Dodaj łącze, którego kliknięcie spowoduje ponownie zażądanie strony. Spowoduje to przełączenie strony do stanu, w którym nie pracuje już ona w trybie postback.

```
<a href="BookPortal.aspx">Zobacz wszystkie książki</a>
```

Przykład 11.3. Dodanie funkcji obsługi zdarzeń do formularza WWW

Po utworzeniu kontrolek można dodać funkcje obsługi zdarzeń wyboru. Każda z tych funkcji zmienia programowo kontrolkę `ObjectDataSource` używaną w kontrolce `GridView` do pracy z danymi książek.

1. Dodaj do formularza następujące metody z funkcjami obsługi zdarzeń:

```
protected void grdPublisher_SelectedIndexChanged(
   object sender, EventArgs e)
{
   dsBooks.SelectMethod = "GetBooksByPublisher";
   dsBooks.SelectParameters.Clear();
   Parameter p = new ControlParameter("pubId", "grdPublisher",
      "SelectedDataKey.Values[0]");
   dsBooks.SelectParameters.Add(p);
}

protected void grdSeries_SelectedIndexChanged(object sender,
   EventArgs e)
{
   dsBooks.SelectMethod = "GetBooksBySeries";
   dsBooks.SelectParameters.Clear();
   Parameter p = new ControlParameter("seriesId", "grdSeries",
      "SelectedDataKey.Values[0]");
   dsBooks.SelectParameters.Add(p);
}

protected void grdCategory_SelectedIndexChanged(
   object sender, EventArgs e)
{
```

```
    dsBooks.SelectMethod = "GetBooksByCategory";
    dsBooks.SelectParameters.Clear();
    Parameter p = new ControlParameter("catId", "grdCategories",
       "SelectedDataKey.Values[0]");
    dsBooks.SelectParameters.Add(p);
}
```

Należy zauważyć, że każda funkcja obsługi zdarzenia zmienia po prostu metodę wybierającą kontrolki `ObjectDataSource` danej książki na odpowiednią metodę `GetBooksBy`*`X`* klasy `BookCatalogLogic`. Ponieważ metoda `GetBooksBy`*`X`* pobiera parametr, każda funkcja obsługi zdarzenia musi zapełnić ten parametr na podstawie wybranej wartości kontrolki `GridView` wydawcy, serii albo kategorii.

2. Dodaj do formularza następującą metodę `Page_Load`:

```
protected void Page_Load(object sender, EventArgs e)
{
    if (!IsPostBack)
    {
       dsBooks.SelectMethod = "GetAllBooks";
    }
}
```

Przy pierwszym załadowaniu strony (i za każdym razem, gdy użytkownik kliknie łącze *Zobacz wszystkie książki*) kontrolka `ObjectDataSource` kontrolki `GridView` książek powinna użyć metody `GetAllBooks`, która zwraca zbiór wszystkich książek obecnych w bazie danych.

3. Teraz można już przetestować stronę w przeglądarce. Powinny być wyświetlane listy z wydawcami, seriami oraz kategoriami, a także lista książek. Lista książek powinna ulegać zmianie po wybraniu wydawcy, serii albo kategorii.

Przykład 11.4. Dodanie możliwości przeglądania oraz edycji danych wybranej książki

Kontrolka `DetailsView` zostanie użyta w celu wyświetlania i edycji wszystkich danych wybranej książki.

1. Dodaj do formularza WWW następującą kontrolkę `DetailsView`:

```
<asp:DetailsView ID="dvEditBook" runat="server"
    DataSourceID="dsBookSingle" DataKeyNames="Isbn"
    AutoGenerateRows="False" >

    <HeaderTemplate>Book Details</HeaderTemplate>
    <Fields>
       <asp:BoundField DataField="Isbn" HeaderText="Isbn"
          ReadOnly="true"/>

       <asp:TemplateField HeaderText="Tytuł">
          <ItemTemplate><%# Eval("Title")%></ItemTemplate>
          <EditItemTemplate>
             <asp:TextBox ID="txtTitle" runat="server"
                Columns="55"
                Text='<%# Bind("Title") %>' />
          </EditItemTemplate>
       </asp:TemplateField>
```

```
<asp:TemplateField HeaderText="Autorzy">
   <ItemTemplate>
      <asp:GridView id="grdAuthors" runat="server"
         AutoGenerateColumns="false" ShowHeader="false"
         DataSource='<%# Bind("Authors") %>'>

         <Columns>
            <asp:BoundField DataField="Name" />
         </Columns>

      </asp:GridView>
   </ItemTemplate>
   <EditItemTemplate>
      <a href='EditAuthors.apx?book=<%#
         Eval("ISBN") %>'>
      Edit Authors
      </a>
   </EditItemTemplate>
</asp:TemplateField>

<asp:BoundField DataField="YearPublished"
   HeaderText="Rok" />

<asp:TemplateField HeaderText="Opis">
   <ItemTemplate>
      <%# Eval("BriefDescription")%>
   </ItemTemplate>
   <EditItemTemplate>
      <asp:TextBox ID="txtBrief" runat="server"
        TextMode="multiLine" Columns="45" Rows="7"
        Text='<%# Bind("BriefDescription") %>'/>
   </EditItemTemplate>
</asp:TemplateField>

<asp:TemplateField HeaderText="Wydawca">
   <ItemTemplate>
      <%# Eval("BookPublisher.Name") %>
   </ItemTemplate>
   <EditItemTemplate>
      <asp:DropDownList ID="drpPublisher" runat="server"
        DataSourceID="dsPublishers" DataValueField="Id"
        DataTextField="Name"
        SelectedValue='<%# Bind("PublisherId") %>' />
   </EditItemTemplate>
</asp:TemplateField>
<asp:TemplateField HeaderText="Seria">
   <ItemTemplate>
      <%# Eval("BookSeries.Name") %>
   </ItemTemplate>
   <EditItemTemplate>
      <asp:DropDownList ID="drpSeries" runat="server"
        DataSourceID="dsSeries" DataValueField="Id"
        DataTextField="Name"
        SelectedValue='<%# Bind("SeriesId") %>' />
   </EditItemTemplate>
</asp:TemplateField>
```

```
        <asp:TemplateField HeaderText="Kategoria">
           <ItemTemplate>
              <%# Eval("BookCategory.Name") %>
           </ItemTemplate>
           <EditItemTemplate>
              <asp:DropDownList ID="drpCategory" runat="server"
                DataSourceID="dsCategories" DataValueField="Id"
                DataTextField="Name"
                SelectedValue='<%# Bind("CategoryId") %>' />
           </EditItemTemplate>
        </asp:TemplateField>

        <asp:CommandField ButtonType="Image"
           ShowEditButton="true" ShowCancelButton="true"
           CancelImageUrl="images/btn_cancel.gif"
           EditImageUrl="images/btn_edit.gif"
         UpdateImageUrl="images/btn_update.gif" />

     </Fields>
  </asp:DetailsView>
```

Należy zauważyć, że autorzy książek są wyświetlani w zagnieżdżonej kontrolce GridView, ale już ich edycja odbywa się w odrębnym formularzu WWW. Edytowanie autorów na osobnej stronie jest łatwiejsze, ponieważ jest potrzebna nie tylko możliwość wybierania różnych autorów, ale też ich dodawania oraz usuwania.

Oprócz tego w kontrolce użyto kilku potomnych właściwości obiektu Book. Nazwa wydawcy jest na przykład odczytywana w wyrażeniu wiązania danych przy zastosowaniu następującej notacji z kropką:

```
<%# Eval("BookPublisher.Name") %>
```

Przykład 11.5. Obsługa nieprawidłowych reguł biznesowych

Przed aktualizacją informacji o książce należy sprawdzić, czy dane wprowadzone przez użytkownika nie naruszają jakichkolwiek reguł biznesowych dla encji Book. Ponieważ kontrolka ObjectDataSource książki wywołuje metodę aktualizującą, należy sprawdzić, czy w jej zdarzeniu OnUpdating nie zostały naruszone reguły. Zdarzenie to jest uruchamiane przed wywołaniem metody aktualizującej, ale już po utworzeniu i zapełnieniu obiektu danych, który będzie do tej metody przekazany.

1. Dodaj do strony poniższą kontrolkę GridView. Wyświetla ona naruszone reguły biznesowe.

```
<asp:GridView ID="grdErrors"
   runat="server" ShowHeader="false" />
```

2. Dodaj następujący kod do kontrolki ObjectDataSource:

```
<asp:ObjectDataSource ID="dsBookSingle" runat="server"
   TypeName="FourLayer.ApplicationLogic.BookCatalogLogic"
   DataObjectTypeName="FourLayer.BusinessEntity.Book"
   SelectMethod="GetBookByIsbn"
   UpdateMethod="UpdateBook"
   OnUpdating="dsBookSingle_Updating" >
```

3. Dodaj do formularza poniższą funkcję obsługi zdarzenia:

```
/// <podsumowanie>
/// Uruchamiana, gdy kontrolka ObjectDataSource książki próbuje zaktualizować dane
/// </podsumowanie>
protected void dsBookSingle_Updating(object sender,
   ObjectDataSourceMethodEventArgs e)
{
   // Pobierz zapełniony parametr aktualizujący
   Book bk = (Book)e.InputParameters[0];

   // Zapytaj encję, czy istnieją naruszone reguły biznesowe
   if (!bk.IsValid)
   {
      // Reguły zostały naruszone; anuluj aktualizację
      e.Cancel = true;
   }
   // Wyświetl naruszone reguły biznesowe (które, jeśli są
   // poprawne, będą puste)
   grdErrors.DataSource = bk.BrokenRules;
   grdErrors.DataBind();
}
```

Funkcja obsługi zdarzenia pobiera wypełnioną encję książki, która zostanie przekazana metodzie `Update` w parametrze `ObjectDataSourceMethodEventArgs`. Następnie pyta encję książki o naruszone reguły biznesowe. Jeśli takie reguły istnieją, wiąże listę komunikatów naruszonych reguł z kontrolką `GridView` błędów.

4. Przetestuj formularz WWW w przeglądarce. Aktualizacja powinna zostać anulowana, jeśli zostały naruszone jakiekolwiek reguły biznesowe, a w kontrolce `GridView` powinny zostać wyświetlone komunikaty o naruszonych regułach, co pokazano na rysunku 11.13.

Korzystanie z niezależnych zasobów projektowych

Opracowanie w pełni funkcjonalnej, wielowarstwowej architektury aplikacji nie należy do trywialnych zadań. Niektórzy programiści wolą zamiast tego skorzystać z zasobów udostępnianych przez niezależnych producentów, co ułatwia konstruowanie infrastruktury aplikacji.

Istniejące biblioteki z architekturami dostępu do danych oraz aplikacji, takie jak Microsoft Data Access Application Block (znana także pod nazwą Microsoft Enterprise Library) albo NHibernate, mogą być dołączone do istniejącej albo nowej aplikacji sieciowej. Narzędzia generujące kod, takie jak LLBLGen lub MyGenerator, mogą generować profesjonalne klasy dostępu do danych albo logiki biznesowej. Niektórzy programiści z kolei w celu obsługi połączenia modelu domeny i relacyjnej bazy danych wolą korzystać z techniki ORM (Object-Relational Mapper). Do popularnych programów ORM dla środowiska .NET można zaliczyć EasyObjects.NET, ORM.NET, EntityBroker albo DataObjects.

Przegląd zagadnień związanych z używaniem tego typu niezależnych zasobów można znaleźć w artykule Boumy, wymienionym w podrozdziale „Odnośniki”.

Rysunek 11.13.
Naruszone reguły biznesowe na stronie BookPortal.aspx

Podsumowanie

W niniejszym rozdziale skupiono się na implementacji warstwowych architektur aplikacji sieciowych. Fakt, że utworzenie aplikacji sieciowej może być skomplikowanym przedsięwzięciem, stanowi uzasadnienie dla dobrze zaprojektowanej infrastruktury aplikacji. „Prawdziwa” aplikacja sieciowa może zawierać dziesiątki, a nawet setki opisanych przypadków użycia, co sprawia, że niezbędne staje się połączenie wysiłku wielu programistów. To właśnie w takim rodzaju aplikacji sieciowych prawidłowe zasady projektowania oprogramowania są tak ważne. W niniejszym rozdziale omówiono jedną z najważniejszych idei wykorzystywanych podczas projektowania nowoczesnego, złożonego oprogramowania: pojęciowe

rozbicie klas aplikacji na powiązane, ale jednocześnie niezależne warstwy. W szczególności opisano, a następnie zaimplementowano aplikację dwuwarstwową, trójwarstwową, a następnie czterowarstwową.

W następnym rozdziale zostaną omówione niektóre dodatkowe możliwości środowiska ASP.NET, które są istotne w każdej aplikacji sieciowej: zarządzanie stanem oraz korzystanie z bufora ASP.NET.

Ćwiczenia

Rozwiązania poniższych ćwiczeń można znaleźć na mojej stronie WWW pod adresem *http://www.randyconnolly.com/core*. Znajdują się tam także dodatkowe ćwiczenia, które są dostępne wyłącznie dla nauczycieli i wykładowców.

1. Utwórz obiekt biznesowy `Book`, przyjmując za model przykład klasy `PublisherBO` z listingu 11.2. W celu zaimplementowania interakcji z bazą danych możesz skorzystać z obiektu dostępu do danych `BookDA`. Utwórz stronę testową, na której zademonstrujesz nową funkcję klasy.
2. Dodaj następującą funkcję do klasy `BookCatalogLogic`: pobranie listy książek z określonym identyfikatorem autora, aktualizacja wydawcy, wstawienie wydawcy i usunięcie wydawcy. Utwórz stronę, na której zademonstrujesz nową funkcję klasy.

W następnych dwóch ćwiczeniach jest używana baza danych *ModernEyeCatalog*.

3. Zaprojektuj i zaimplementuj dla tej bazy warstwę obiektów biznesowych. Jako wskazówki możesz użyć przykładu pokazanego w listingu 11.2. Nie zapomnij o utworzeniu obiektu biznesowego o nazwie `ArtWorkBO`.
4. Zaprojektuj i zaimplementuj dla tej bazy model czterowarstwowy.

Najważniejsze zagadnienia

- Adapter
- CRUD (tworzenie, odczytywanie, aktualizacja i usuwanie)
- Domena
- Encja
- Logika aplikacji
- Obiekt biznesowy
- Obiekt transferu danych
- Poziom

- Przypadki użycia
- Reguły biznesowe
- Skojarzenia
- Spójność
- Warstwa
- Warstwa biznesowa
- Warstwa domeny
- Warstwa dostępu do danych
- Warstwa logiki aplikacji
- Warstwa prezentacji
- Wzorzec Template Method (wzorzec metody szablonu)
- Zależność

Odnośniki

Frank Bouma, „Solving the Data Access Problem: to O/R Map or Not to O/R Map", *http://weblogs.asp.net/fbouma*.

Frank Buschmann i inni, *Pattern-Oriented Software Architecture: A System of Patterns*, John Wiley & Sons 1996.

Eric Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*, Addison-Wesley 2004.

Martin Fowler, *Architektura systemów zarządzania przedsiębiorstwem. Wzorce projektowe*, Helion 2005.

Craig Larman, *Agile and Iterative Development: A Manager's Guide*, Addison-Wesley 2003.

Microsoft, *Application Architecture for .NET: Designing Applications and Services*, Microsoft 2003.

Jimmy Nilsson, *Applying Domain-Driven Design and Patterns*, Addison-Wesley 2006.