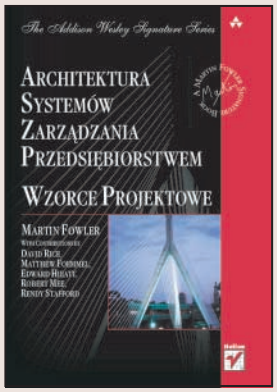
Architektura systemów zarządzania przedsiębiorstwem. Wzorce projektowe



Autor: Martin Fowler
Tłumaczenie: Paweł Koronkiewicz (wstęp, rozdz. 1 – 12), Piotr Rajca (rozdz. 13 – 18, dod. A)
ISBN: 83-7361-715-9
Tytuł oryginału: Patterns of Enterprise Application Architecture
Format: B5, stron: 496

Wykorzystaj wzorce projektowe w pracy nad oprogramowaniem

- Zaprojektuj aplikacje o architekturze trójwarstwowej
- Dobierz odpowiednią technologię
- Stwórz moduły aplikacji

Systemy informatyczne służące do zarządzania przedsiębiorstwem to zwykle ogromne aplikacje. Operują na milionach rekordów, przesyłają gigabajty danych i są obsługiwane przez dziesiątki użytkowników. Sprawne działanie takiej aplikacji jest niezwykle istotne dla funkcjonowania przedsiębiorstwa, dlatego musi ona być stabilna, a przed wdrożeniem – gruntownie przetestowana. Przy tworzeniu aplikacji tego typu wykorzystuje się opracowane już rozwiązania, zwane wzorcami projektowymi. Wzorce projektowe to modele poszczególnych komponentów aplikacji — należy jedynie zaimplementować je w wybranym języku programowania.

Książka „Architektura systemów zarządzania przedsiębiorstwem. Wzorce projektowe" to przegląd wzorców wykorzystywanych przy projektowaniu aplikacji korporacyjnych. Opisuje zasady podziału aplikacji na warstwy i zasady współpracy pomiędzy warstwami; przedstawia także modele komponentów wchodzących w skład każdej z nich.

- Warstwy w aplikacjach biznesowych
- Wzorce logiki aplikacji
- Wzorce architektury źródła danych
- Wzorce mapowania obiektowo-relacyjnego
- Wzorce prezentacji
- Wzorce dystrybucji
- Wzorce stanu sesji
- Wzorce podstawowe

Korzystając z zawartych w książce wzorców, stworzysz stabilne i wydajne aplikacje korporacyjne.

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści

8.

9.

10.

13.

Wzorce odwzorowań obiektów i relacyjnych metadanych 277

14.

Wzorce prezentacji internetowych 299

15.

Wzorce dystrybucji 353

16.

Wzorce współbieżności autonomicznej 379

17.

Wzorce stanu sesji 415

18.

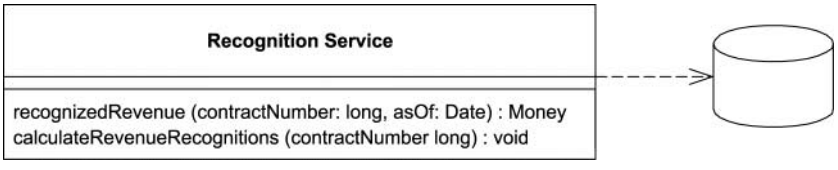
Wzorce podstawowe 425

9

Wzorce logiki dziedziny

Transaction Script (Skrypt transakcji)

Porządkuje logikę dziedziny w procedury, gdzie każda procedura obsługuje pojedyncze żądanie warstwy prezentacji.



Pracę większości aplikacji biznesowych można rozpatrywać jako przetwarzanie sekwencji transakcji. Transakcja może polegać na przeglądaniu danych w pewien określony sposób lub wprowadzaniu w nich zmian. Każda interakcja między systemem klienckim a systemem serwera obejmuje pewną część logiki aplikacji. W pewnych przypadkach interakcja może być tak prosta jak wyświetlanie przechowywanych w bazie danych. W innym może wymagać wielu operacji sprawdzania poprawności i obliczeń.

Wzorzec *Transaction Script* porządkuje logikę takiej interakcji jako, ogólnie rzecz biorąc, pojedynczą procedurę, która wywołuje bazę danych bezpośrednio lub za pośrednictwem prostego kodu osłaniającego. Każda transakcja ma własny skrypt. Dodatkową optymalizacją może być łączenie wspólnych fragmentów kodu w podprocedury.

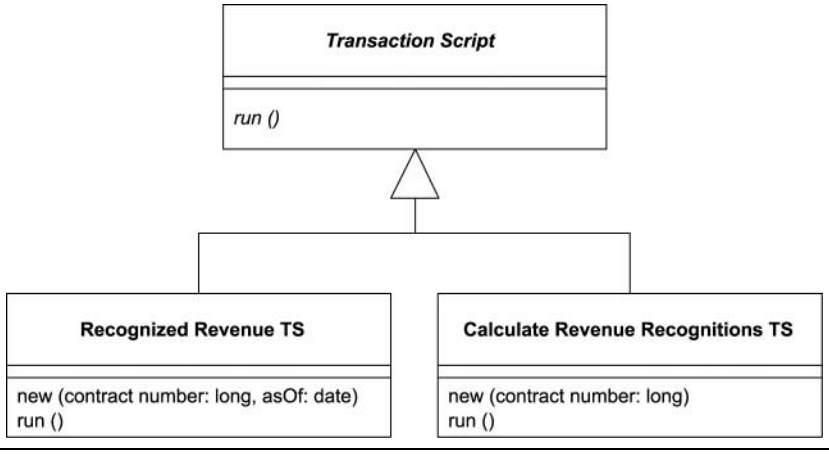
Na czym polega

Gdy stosujemy wzorzec *Transaction Script*, logika domeny porządkowana jest zasadniczo według wykonywanych w systemie transakcji. Gdy celem transakcji jest wprowadzenie rezerwacji pokoju hotelowego, kod pojedynczej procedury Zarezerwuj Pokój obejmuje sprawdzenie dostępności pokoi, określenie wysokości opłaty i zaktualizowanie bazy danych.

W prostych przypadkach sposób porządkowania skryptów transakcji jest naturalny. Oczywiście, podobnie jak w każdym programie, powinniśmy wprowadzić przejrzystą strukturę modułów. O ile transakcja nie jest szczególnie skomplikowana, jest to bardzo proste. Jedną z zalet skryptów transakcji jest to, że nie ma dla nich znaczenia, jakie operacje wykonują inne transakcje. Zadaniem implementowanego kodu jest przyjęcie danych wejściowych, odczytanie danych z bazy, przekształcenie ich, a następnie zapisanie wyników w bazie.

O tym, gdzie umieścimy skrypt transakcji, decyduje przyjęty system warstw aplikacji. Lokalizacją taką może być strona serwera, skrypt CGI lub rozproszony obiekt sesji. Dobrze jest rozdzielać skrypty tak dalece, jak to możliwe. Absolutnym minimum są osobne procedury. Jeszcze lepsze są klasy, oddzielone również od kodu prezentacji i źródła danych. Dodatkowo, w skryptach transakcji nie wprowadzamy żadnych wywołań logiki prezentacji. Ułatwia to ich modyfikowanie i testowanie.

Skrypty transakcji można porządkować w klasy dwoma sposobami. Najbardziej typowym podejściem jest łączenie w jednej klasie grupy skryptów. Każda klasa odpowiada wtedy pewnemu zakresowi tematycznemu. Układ taki jest przejrzysty i sprawdza się w praktyce. Inną możliwością jest umieszczanie każdego skryptu w osobnej klasie (patrz rysunek 9.1). Odpowiada to wzorcowi Command *Gang of Four*. Definiujemy wtedy również supertyp takich poleceń, który określa pewną metodę uruchamiania skryptów. Pozwala to operować skryptami jako obiektami. Trzeba przyznać, że potrzeba taka pojawia się stosunkowo rzadko, ze względu na prostotę systemów, gdzie stosuje się wzorzec *Transaction Script*. Oczywiście, wiele języków pozwala zapomnieć o klasach i definiować funkcje globalne. Nie można jednak zapominać, że tworzenie egzemplarzy obiektów pomaga izolować dane różnych wątków przetwarzania.



RYSUNEK 9.1. Skrypty transakcji jako polecenia

Termin „skrypt transakcji” jest o tyle uzasadniony, że w większości przypadków każdy z nich odpowiada pojedynczej transakcji bazy danych. Nie jest to reguła obowiązująca we wszystkich przypadkach, ale jest stosunkowo dobrym przybliżeniem.

Kiedy używamy

Wielką zaletą skryptów transakcji jest ich prostota. Porządkowanie logiki przy ich użyciu jest bardzo naturalne, gdy cała aplikacja zawiera niewiele kodu. Nie wprowadzamy dodatkowych poziomów złożoności ani elementów obniżających wydajność.

Gdy logika biznesowa staje się bardziej skomplikowana, utrzymanie porządku w skryptach transakcji staje się coraz bardziej skomplikowane. Podstawowym problemem są powtórzenia kodu. Ponieważ każdy skrypt ma obsłużyć dokładnie jedną transakcję, powtórne wprowadzanie takich samych lub podobnych fragmentów jest nieuniknione.

Uważny programista uniknie większości takich problemów, jednak większe aplikacje wymagają zbudowania modelu dziedziny. Wzorzec *Domain Model* (109) zapewnia znacznie więcej opcji strukturalizowania kodu, większą przejrzystość i ograniczenie problemu powtórzeń.

Trudno określić poziom złożoności, na którym skrypty transakcji nie mogą być zastosowane. Będzie to jeszcze trudniejsze, jeżeli przyzwyczailiśmy się do jednego z wzorców. Można oczywiście przekształcić skrypty transakcji w model dziedziny, ale nie jest to zmiana prosta, więc dobrze byłoby zastanowić się nad właściwym podejściem już na samym początku.

Bez względu na to, jak bardzo lubimy podejście obiektowe, nie powinniśmy skreślać wzorca *Transaction Script*. Jest wiele prostych problemów i stosowanie dla nich prostych rozwiązań pozwala szybciej zakończyć pracę.

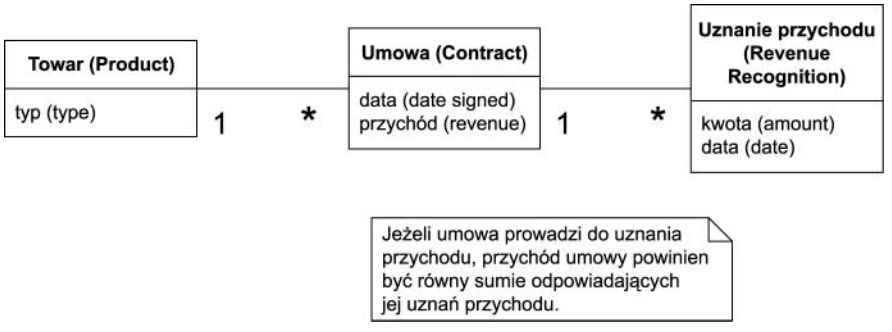
Problem obliczania przychodu

Do zilustrowania tego i dalszych wzorców logiki dziedziny będę używał tego samego problemu. Poniżej przedstawiam jego opis, którego już przy kolejnych wzorcach nie będę niepotrzebnie powtarzał.

Obliczanie uznania przychodu (ang. *revenue recognition*) to dość typowe dla systemów biznesowych zagadnienie. Podstawowym problemem jest to, kiedy można zaksięgować otrzymany przychód (ang. *revenue*). Gdy sprzedaję filiżankę kawy, sytuacja jest prosta: wydaję kawę, biorę pieniądze i natychmiast wprowadzam odpowiednią kwotę do książki przychodów. Nie zawsze jednak jest tak łatwo. Wyobraźmy sobie, że dostaję zaliczkę na ten rok. Nawet jeżeli jest to niewielka kwota, jej natychmiastowe wprowadzenie do książki może nie być możliwe, ponieważ usługa będzie wykonana dopiero na przestrzeni tego roku. Jedną z możliwości może być wprowadzanie jednej dwunastej tej kwoty w każdym kolejnym miesiącu, na wypadek, gdyby umowa została nagle rozwiązana.

Jest wiele różnych i często zmiennych reguł obliczania przychodu na potrzeby urzędu skarbowego. Część ustala ustawa, inne wynikają ze standardów księgowości, jeszcze inne — z wewnętrznych zasad firmy. Śledzenie przychodu może być bardzo złożonym zagadnieniem.

Nie będziemy się tutaj wgłębiać w szczegóły tego procesu. Wyobraźmy sobie jedynie firmę, która sprzedaje trzy rodzaje towarów: edytory tekstu, bazy danych i arkusze kalkulacyjne. Zgodnie z przyjętymi zasadami, całość przychodu z umowy sprzedaży (ang. *contract*) edytora tekstu może zostać zaksięgowana od razu. W przypadku arkusza kalkulacyjnego, jedną trzecią przychodu wprowadzamy od razu, jedną trzecią po sześćdziesięciu dniach i pozostałą jedną trzecią po dziewięćdziesięciu dniach. Gdy sprzedajemy bazę danych, jedną trzecią wprowadzamy od razu, jedną trzecią po trzydziestu dniach i jedną trzecią po sześćdziesięciu dniach. Oczywiście zasady te niewiele mają wspólnego z rzeczywistością i mają służyć jedynie zilustrowaniu omawianych zagadnień.



RYSUNEK 9.2. Uproszczony model obliczania przychodu. Każda umowa obejmuje wiele wartości przychodu, powiązanych z określeniem, kiedy różne części przychodu trafiają do ksiąg

Przykład: uznanie przychodu (Java)

W niniejszym przykładzie stosujemy dwa skrypty transakcji: jeden do obliczania uznań przychodu dla danej umowy i drugi do określenia, jaka kwota przychodu z danej umowy została uznana do określonego dnia. Baza danych ma trzy tabele: towarów (ang. *products*), umów (ang. *contracts*) i uznań przychodu (ang. *revenue recognitions*).

```
CREATE TABLE products (ID int primary key, name varchar, type varchar)
CREATE TABLE contracts (ID int primary key, product int, revenue decimal, dateSigned
date)
CREATE TABLE revenueRecognitions (contract int, amount decimal, recognizedOn date,
PRIMARY KEY (contract, recognizedOn))
```

Pierwszy skrypt oblicza kwotę uznania na dany dzień. Można ją wyliczyć w dwóch krokach: w pierwszym wybieramy wiersze tabeli uznań przychodu, w drugim sumujemy kwoty.

Projekty oparte na wzorcu *Transaction Script* korzystają często ze skryptów, które operują bezpośrednio na bazie danych, wykorzystując kod SQL. W tym przykładzie korzystamy z prostej bramy *Table Data Gateway* (133) jako osłony zapytań SQL. Ze względu na prostotę przykładu, użyjemy tylko jednej, wspólnej bramy, nie tworząc osobnych dla każdej tabeli. Definiujemy w jej obrębie odpowiednią metodę wyszukiwania, `findRecognitionsFor`.

```
class Gateway...

   public ResultSet findRecognitionsFor(long contractID, MfDate asof)
   throws SQLException{
      PreparedStatement stmt = db.prepareStatement(findRecognitionsStatement);
      stmt.setLong(1, contractID);
      stmt.setDate(2, asof.toSqlDate());
      ResultSet result = stmt.executeQuery();
      return result;
   }
   private static final String findRecognitionsStatement =
      "SELECT amount " +
      "  FROM revenueRecognitions " +
      "  WHERE contract = ? AND recognizedOn <= ?";
   private Connection db;
```

Drugi skrypt służy do podsumowania danych przekazanych przez bramę.

```
class RecognitionService...

   public Money recognizedRevenue(long contractNumber, MfDate asof){
      Money result = Money.dollars(0);
      try{
         ResultSet rs = db.findRecognitionsFor(contractNumber, asof);
         while(rs.next()){
            result = result.add(Money.dollars(rs.getBigDecimal("amount")));
         }
         return result;
      }catch(SQLException e){throw new ApplicationException(e);
      }
   }
```

W przypadku tak prostych obliczeń, procedura w języku Java mogłaby również zostać zastąpiona wywołaniem SQL, które sumuje wartości przy użyciu funkcji agregującej.

Podobny podział stosujemy przy obliczaniu uznań przychodu dla wybranej umowy. Skrypt warstwy dziedziny realizuje logikę biznesową.

```
class RecognitionService...

   public void calculateRevenueRecognitions(long contractNumber){
      try{
         ResultSet contracts = db.findContract(contractNumber);
         contracts.next();
         Money totalRevenue = Money.dollars(contracts.getBigDecimal("revenue"));
         MfDate recognitionDate = new MfDate(contracts.getDate("dateSigned"));
         String type = contracts.getString("type");
         if (type.equals("S")){
            Money[] allocation = totalRevenue.allocate(3);
            db.insertRecognition
               (contractNumber, allocation[0], recognitionDate);
            db.insertRecognition
               (contractNumber, allocation[1], recognitionDate.addDays(60));
            db.insertRecognition
               (contractNumber, allocation[2], recognitionDate.addDays(90));
         }else if (type.equals("W")){
            db.insertRecognition(contractNumber, totalRevenue, recognitionDate);
         }else if (type.equals("D")){
            Money[] allocation = totalRevenue.allocate(3);
            db.insertRecognition
               (contractNumber, allocation[0], recognitionDate);
            db.insertRecognition
               (contractNumber, allocation[1], recognitionDate.addDays(30));
            db.insertRecognition
               (contractNumber, allocation[2], recognitionDate.addDays(60));
         }
      } catch(SQLException e){throw new ApplicationException(e);
      }
   }
```

Zwróćmy uwagę na użycie klasy *Money* (446) do alokacji. Zapobiega ona gubieniu pojedynczych groszy (lub centów), o co łatwo przy dzieleniu kwot pieniężnych przez trzy.

Obsługa SQL jest zrealizowana w formie wzorca *Table Data Gateway* (133). Pierwsza procedura znajduje umowę.

```
class Gateway...

   public ResultSet findContract(long contractID) throws SQLException{
      PreparedStatement stmt = db.prepareStatement(findContractStatement);
      stmt.setLong(1, contractID);
      ResultSet result = stmt.executeQuery();
      return result;
   }
   private static final String findContractStatement =
      "SELECT * " +
      "  FROM contracts c, products p " +
      "  WHERE ID = ? AND c.products = p.ID";
```

Druga procedura to osłona instrukcji `INSERT`.

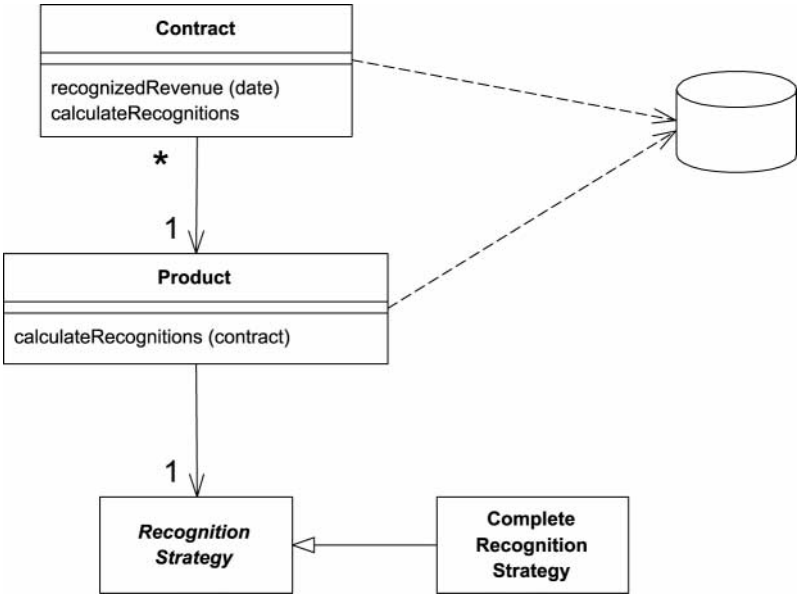
```
class Gateway...

   public void insertRecognition(long contractID, Money amount, MfDate asof)
   throws SQLException{
      PreparedStatement stmt = db.prepareStatement(insertRecognitionStatement);
      stmt.setLong(1, contractID);
      stmt.setBigDecimal(2, amount.amount());
      stmt.setDate(3, asof.toSqlDate());
      stmt.executeUpdate();
   }
   private static final String insertRecognitionStatement =
      "INSERT INTO revenueRecognition VALUES (?, ?, ?)";
```

Gdy używamy języka Java, usługa obliczania uznań dochodu może mieć postać tradycyjnej klasy lub obiektu bean sesji.

Czytelnik, który nie jest przyzwyczajony do modelowania dziedziny, uzna prawdopodobnie przedstawioną tu implementację za znacznie prostszą niż przedstawiona w opisie wzorca *Domain Model* (109). Niestety, nieco trudniej w zwięzły sposób przedstawić (lub wyobrazić sobie), co stanie się, gdy reguły biznesowe będą bardziej skomplikowane. Stosowane w praktyce reguły obliczania uznań przychodu nie są proste, a różnice między nimi wyznacza nie tylko produkt, którego dotyczą, ale i data operacji („jeżeli umowa została podpisana przed 15 kwietnia, obowiązuje taka a taka reguła"). Gdy poziom złożoności jest duży, utrzymanie spójnej konstrukcji skryptów transakcji jest bardzo trudne, co doskonale uzasadnia przywiązanie miłośników podejścia obiektowego do stosowania modeli dziedziny.

Domain Model (model dziedziny)

Obiektowy model dziedziny, obejmujący wymagane zachowania i dane.



Logika biznesowa może osiągnąć bardzo duży poziom złożoności. Reguły i logika opisują wiele różnych przypadków i wariantów zachowań, a właśnie rozwiązanie problemu złożoności było główną przesłanką stworzenia koncepcji obiektów. Model dziedziny to sieć takich połączonych ze sobą obiektów, gdzie każdy obiekt reprezentuje pewien znaczący element lub czynnik. Elementy te mogą być tak znaczne jak przedsiębiorstwo lub tak niewielkie jak pojedynczy wiersz formularza zamówienia.

Na czym polega

Wprowadzenie w aplikacji wzorca *Domain Model* (109) wymaga stworzenia dość rozbudowanej koncepcyjnie warstwy obiektów, które modelują rozwiązywany problem. Należą do nich obiekty reprezentujące dane przedsiębiorstwa i obiekty odpowiadające regułom jego pracy. Dane i procesy najczęściej łączy się ze sobą, aby skupić czynności przetwarzania i informacje, którymi operują.

Obiektowy model dziedziny może przypominać model bazy danych, zawsze jednak dzielą je istotne różnice. Model dziedziny łączy dane i procesy, ma wielowartościowe atrybuty, złożoną sieć asocjacji i wykorzystuje dziedziczenie.

Można wyróżnić dwa rodzaje modeli dziedziny. Prosty model przypomina projekt bazy danych i dominuje w nim układ „jeden obiekt modelu-jedna tabela bazy danych". Rozbudowany model dziedziny znacznie odbiega od struktury bazy i wykorzystuje dziedziczenie, strategie i inne wzorce *Gang of Four*, jak również złożone sieci niewielkich, połączonych ze sobą obiektów. Takie

podejście jest lepsze, gdy logika jest bardziej skomplikowana, trudniej jednak wówczas przeprowadzić mapowanie do bazy. W prostym modelu można korzystać z aktywnych rekordów (*Active Record* (147)), rozbudowany wymaga mechanizmu *Data Mapper* (152).

Ponieważ funkcje biznesowe wymagają zawsze wielu późniejszych zmian, ważna jest możliwość łatwego modyfikowania, kompilowania i testowania warstwy, w której są implementowane. Musimy więc dbać o to, aby sprzężeń między modelem a innymi warstwami systemu było jak najmniej. Łatwo zauważyć, że podstawą wielu wzorców układu warstwowego jest właśnie utrzymanie jak najmniejszej zależności między warstwą dziedziny a innymi.

Rozwiązania obsługi modelu dziedziny mogą być różne. Najprostszym przypadkiem jest aplikacja dla jednego użytkownika, gdzie cały graf obiektów zostaje odczytany z pliku i załadowany do pamięci. O ile sprawdza się to w przypadku aplikacji biurowych, nie jest raczej stosowane w wielowarstwowych aplikacjach systemów informacyjnych z tej prostej przyczyny, że obiektów jest wtedy zbyt wiele. Pamięć jest zbyt mała, aby takie obciążenie było uzasadnione, a samo ładowanie obiektów trwa zbyt długo. Urok obiektowych baz danych polega właśnie na tym, że zapewniają iluzję wykonywania takiej operacji, podczas gdy w rzeczywistości jedynie zarządzają przenoszeniem obiektów pomiędzy pamięcią a dyskiem.

Brak obiektowej bazy danych zmusza do samodzielnego projektowania podobnych rozwiązań. W typowym przypadku w każdej sesji ładowany jest graf obiektów, których ta sesja wymaga. Nie są to jednak nigdy wszystkie obiekty aplikacji i zazwyczaj nie wszystkie stosowane klasy. Przykładowo, gdy przeglądany jest zbiór umów, z dysku ładowane są tylko obiekty odpowiadające produktom, do których te umowy się odwołują. Gdy przeprowadzane są obliczenia operujące umowami i uznaniami przychodu, obiekty reprezentujące produkty mogą nie być ładowane w ogóle. O tym, co zostanie załadowane do pamięci, decydują obiekty zarządzające mapowaniem do bazy danych.

Gdy pojawia się potrzeba utrzymania tego samego grafu obiektów pomiędzy kolejnymi wywołaniami serwera, niezbędne jest zachowanie danych stanu. To zagadnienie omawiamy w rozdziale poświęconym stanowi sesji (strona 79).

Typowym problemem związanym z logiką dziedziny jest nadmierne „puchnięcie" obiektów. W trakcie projektowania ekranu do zarządzania zamówieniami możemy zauważyć, że niektóre funkcje zamówień służą wyłącznie do jego obsługi. Przypisanie tych funkcji zamówieniu może doprowadzić do niepotrzebnej rozbudowy odpowiedniej klasy. Niepotrzebnej, ponieważ wiele funkcji wykorzystanych zostaje tylko w jednym przypadku użycia. Wielu programistów zwraca więc pilnie uwagę na to, czy pewne funkcje mają charakter ogólny (i mogą być implementowane w klasie Zamówienie), czy są specyficzne dla określonych operacji. W tym ostatnim przypadku powinny być implementowane w pewnej klasie związanej z zastosowaniem obiektu. Może to oznaczać skrypt transakcji lub warstwę prezentacji.

Problem oddzielania zachowań specyficznych dla zastosowań obiektu jest powiązany z zagadnieniem duplikacji kodu. Funkcję oddzieloną od zamówienia trudniej znaleźć, łatwo więc — na pewnym etapie projektu — o przeoczenie prowadzące do ponownej implementacji tego samego zachowania. Powtórzenia kodu z kolei prowadzą do szybkiego zwiększania jego złożoności i utraty spójności. Z moich doświadczeń wynika, że „puchnięcie" obiektów nie jest tak częstym zjawiskiem, jak mogłoby się na pierwszy rzut oka wydawać. Choć nie można zaprzeczyć jego występowaniu, łatwo je wykryć i wprowadzić niezbędne korekty. Zalecałbym więc raczej powstrzymanie się od oddzielania funkcji od obiektów i implementowanie ich w tych klasach, gdzie w naturalny sposób pasują. „Odchudzanie" obiektów stosujemy dopiero wtedy, gdy faktycznie stwierdzimy, że jest konieczne.

Implementacja w języku Java

Implementowanie modelu dziedziny w J2EE wzbudza mnóstwo emocji. Wiele materiałów szkoleniowych i podręczników zaleca stosowanie do tego celu obiektów *entity bean*. Podejście takie wiąże się jednak z pewnymi poważnymi trudnościami. Być może zostaną one usunięte w przyszłych wersjach specyfikacji (w chwili pisania tej książki obowiązuje wersja 2.0).

Obiekty *entity bean* najlepiej sprawdzają się, gdy stosujemy mechanizm Container Managed Persistance (CMP, kontenerowo zarządzane składowanie). Można wręcz stwierdzić, że w innych rozwiązaniach stosowanie obiektów entity bean nie ma uzasadnienia. CMP jest jednak dość ograniczoną formą mapowania obiektowo-relacyjnego i nie pozwala stosować wielu wzorców potrzebnych w rozbudowanych modelach dziedziny.

Obiekty *entity bean* nie powinny mieć charakteru wielobieżnego, co znaczy, że jeżeli obiekt *entity bean* wywołuje inny obiekt, ten inny obiekt (ani żaden inny w łańcuchu dalszych wywołań) nie może wywołać pierwszego obiektu *entity bean*. Jest to o tyle kłopotliwe, że rozbudowane modele dziedziny często wykorzystują wielobieżność. Co gorsza, zachowania tego rodzaju są trudne do wykrycia. Prowadzi to do popularnego zalecenia, aby obiekty *entity bean* nie wywoływały się wzajemnie. Pozwala to co prawda uniknąć wielobieżności, ale znacznie ogranicza korzyści ze stosowania wzorca *Domain Model*.

Model dziedziny powinien opierać się na obiektach z interfejsami o dużej ziarnistości (podobna cecha powinna charakteryzować samą strukturę obiektów). Jednak zdalne wywoływanie obiektów z interfejsami o dużej ziarnistości prowadzi do bardzo niskiej wydajności systemu. W konsekwencji, pomimo tego, że obiekty *entity bean* mogą być dostosowane do wywołań zdalnych (we wcześniejszych wersjach specyfikacji była to wymagana cecha), w modelu dziedziny powinniśmy stosować wyłącznie interfejsy lokalne.

Aby korzystać z obiektów *entity bean*, niezbędny jest kontener i połączenie z bazą danych. Zwiększa to czas kompilacji i czas niezbędny do wykonania testów, bo obiekty te muszą korzystać z bazy danych. Również debugowanie obiektów *entity bean* nie należy do najprostszych.

Alternatywą jest stosowanie zwykłych obiektów języka Java, nawet jeżeli takie podejście może być dla wielu osób zaskakujące — zadziwiające, jak wielu programistów jest utwierdzonych w przekonaniu, że w kontenerze EJB nie mogą pracować zwykłe obiekty. Doszedłem kiedyś do wniosku, że zwykłe obiekty Java idą w zapomnienie, bo nie mają ładnej nazwy. Dlatego właśnie, przygotowując się do pewnej dyskusji w 2000 roku, razem z Rebeccą Parsons i Joshem Mackenzie wymyśliliśmy nazwę POJO (ang. *plain old Java objects*, zwykłe obiekty języka Java). Model dziedziny oparty na obiektach POJO stosunkowo łatwo jest opracować, szybko się kompiluje, a można go uruchamiać i testować poza kontenerem EJB. Jest on zasadniczo niezależny od EJB (co być może jest przyczyną, dla której producenci związani z EJB nie zachęcają do takich rozwiązań).

Reasumując, wydaje mi się, że zastosowanie obiektów entity bean do implementacji modelu dziedziny sprawdza się wtedy, gdy logika dziedziny jest tylko umiarkowanie rozbudowana. Wówczas model może mieć prosty związek z bazą danych, oparty na ogólnej zasadzie przypisania jednej klasy entity bean do jednej tabeli bazy danych. Bardziej rozbudowana logika, wykorzystująca dziedziczenie, strategie i inne wyrafinowane wzorce, wymaga modelu opartego na obiektach POJO i wzorcu *Data Mapper* (152). Ten ostatni możemy wygenerować korzystając ze specjalnego, zakupionego narzędzia.

Mnie osobiście najbardziej zniechęca do korzystania z EJB fakt, że rozbudowany model dziedziny jest sam w sobie wystarczająco skomplikowany, aby skłaniać do utrzymywania jak największej niezależności od środowiska implementacji. EJB wymusza pewne schematy działania, co sprawia, że musimy zajmować się nie tylko dziedziną, ale i jednocześnie środowiskiem EJB.

Kiedy używamy

O ile odpowiedź na pytanie „jak” jest trudna ze względu na obszerność tematu, odpowiedź na pytanie „kiedy” nie jest łatwa ze względu na swoją ogólność i prostotę. Sprowadza się bowiem do rozważenia poziomu złożoności funkcji systemu. Skomplikowane i zmienne reguły biznesowe,

obejmujące sprawdzanie poprawności, obliczenia i derywacje, zdecydowanie powinny skłaniać do zastosowania modelu obiektowego. Z drugiej strony, proste weryfikacje typu „not null" i obliczanie kilku podsumowań to zadanie idealne dla skryptów transakcji.

Jednym z czynników jest doświadczenie zespołu pracującego nad aplikacją, a konkretniej — jak radzi on sobie z operowaniem obiektami dziedziny. Projektowanie i praca z modelem dziedziny to coś, czego trzeba się nauczyć. Stąd wiele artykułów opisujących tę „zmianę paradygmatu" w rozwiązaniach obiektowych. Proces nauki jest długi, a zdobycie praktyki wymaga czasu. Ukoronowaniem nauki jest rzeczywista zmiana sposobu myślenia, kiedy projektant przestaje stosować inne wzorce warstwy dziedziny i nie chce używać skryptów transakcji w żadnych aplikacjach poza, ewentualnie, najprostszymi.

Gdy stosujemy model dziedziny, podstawowym schematem interakcji z bazą danych jest *Data Mapper* (152). Pomaga on utrzymać niezależność modelu od bazy i jest najlepszym podejściem, gdy model dziedziny i schemat bazy danych znacznie różnią się od siebie.

Uzupełnieniem modelu dziedziny może być warstwa usług (*Service Layer* (124)), zapewniająca modelowi przejrzysty interfejs API.

Warto przeczytać

Niemal każda książka traktująca o projektowaniu obiektowym porusza zagadnienie modeli dziedziny. Decyduje o tym fakt, że w powszechnym rozumieniu programowanie obiektowe opiera się właśnie na takim podejściu.

Czytelnikom poszukującym wprowadzenia do projektowania obiektowego polecam obecnie książkę *Larman*. Przykłady modelu dziedziny znajdziemy w *Fowler AP*. *Hay* zawiera wiele przykładów ukierunkowanych na kontekst relacyjny. Zbudowanie dobrego modelu dziedziny wymaga dobrej znajomości teorii projektowania obiektowego. Jest ona doskonale wyłożona w *Martin and Odell*. Wyczerpujący przegląd wzorców charakterystycznych dla rozbudowanych modeli dziedziny, a także innych systemów obiektowych, znajdziemy w *Gang of Four*.

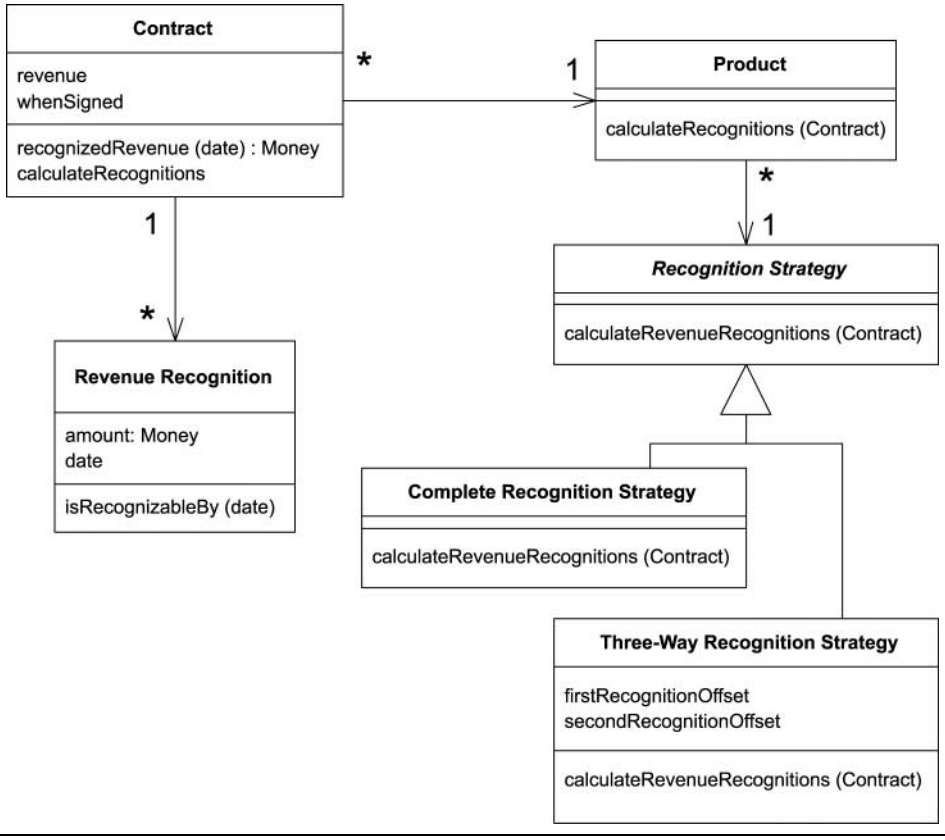
Eric Evans pisze obecnie książkę *Evans*, traktującą właśnie o budowaniu modeli dziedziny. Do chwili pisania tych słów miałem okazję zetknąć się tylko z jej wstępną wersją, ale wyglądała ona bardzo obiecująco.

Przykład: uznanie przychodu (Java)

Opisywanie zasad modelowania dziedziny jest dość niewdzięcznym zajęciem, ponieważ każdy przykład, który można przedstawić, jest z konieczności bardzo uproszczony. Uproszczenia te skutecznie ukrywają wszelkie mocne strony tego rodzaju rozwiązań. Można je naprawdę docenić tylko wtedy, gdy rozpatrzymy naprawdę złożoną dziedzinę, na co oczywiście nie ma tu miejsca.

Choć przykład nie może być wystarczająco dobrym dowodem wielkich korzyści, jakie zapewnia modelowanie dziedziny, może przynajmniej dać Czytelnikowi pewne pojęcie o tym, jak taki model może wyglądać. Korzystam tutaj z tego samego przykładu (strona 105), który posłużył do zilustrowania wzorca *Transaction Scripts* (skryptów transakcji).

Pierwszą rzeczą, którą zauważymy, będzie to, że każda klasa obejmuje zarówno zachowania, jak i dane (patrz rysunek 9.3). Nawet prosta klasa `RevenueRecognition` zawiera metodę służącą do określania, czy w danym dniu wartość obiektu została już zaksięgowana.



RYSUNEK 9.3. Diagram klas dla przykładu modelu dziedziny

```
class RevenueRecognition...

   private Money amount;
   private MfDate date;
   public RevenueRecognition(Money amount, MfDate date) {
      this.amount = amount;
      this.date = date;
   }
   public Money getAmount() {
      return amount;
   }
   Boolean isRecognizableBy(MfDate asOf) {
      return asOf.after(date) || asOf.equals(date);
   }
```

Obliczanie wielkości przychodu uznanego na dany dzień wymaga klas umowy i uznania przychodu.

```
class Contract...

   private List revenueRecognitions = new ArrayList();
```

```
public Money recognizedRevenue(MfDate asOf) {
   Money result = Money.dollars(0);
   Iterator it = revenueRecognitions.iterator();
   while (it.hasNext()) {
      RevenueRecognition r = (RevenueRecognition) it.next();
      if (r.isRecognizableBy(asOf))
         result = result.add(r.getAmount());
   }
   return result;
}
```

Charakterystyczny dla modelów dziedziny jest sposób, w jaki wiele klas współpracuje ze sobą w realizacji nawet najprostszych zadań. To właśnie prowadzi często do konkluzji, że w programach obiektowych ogromną ilość czasu spędzamy na przeszukiwaniu kolejnych klas, szukając tej, która jest nam w danej chwili potrzebna. Konkluzja taka jest niewątpliwie słuszna. Wartość takiego rozwiązania doceniamy, gdy decyzja o uznaniu przychodu w określonym dniu staje się bardziej skomplikowana. Musimy też rozważyć informacje dostępne dla innych obiektów. Zamknięcie zachowania w obiekcie, który potrzebuje określonych danych, pozwala uniknąć powtórzeń kodu i ogranicza sprzężenia między obiektami.

Analiza sposobu obliczania wartości i tworzenia obiektów reprezentujących uznanie przychodu pozwala zauważyć charakterystyczne dla modelu dziedziny liczne obiekty o niewielkich rozmiarach. W przedstawionym przykładzie, obliczenia i tworzenie obiektu rozpoczynają się od klienta i są przekazywane poprzez produkt do hierarchii strategii. Wzorzec strategii *Gang of Four* to popularny wzorzec projektowania obiektowego, który umożliwia połączenie grupy operacji w hierarchię niewielkich klas. Każdy egzemplarz produktu jest połączony z pojedynczym egzemplarzem strategii obliczania uznania, określającej, który algorytm zostanie użyty do obliczenia uznania. W tym przypadku mamy dwie podklasy strategii obliczania uznania, reprezentujące dwa różne przypadki. Struktura kodu wygląda następująco:

```
class Contract...

   private Product product;
   private Money revenue;
   private MfDate whenSigned;
   private Long id;
   public Contract(Product product, Money revenue, MfDate whenSigned) {
      this.product = product;
      this.revenue = revenue;
      this.whenSigned = whenSigned;
   }

class Product...

   private String name;
   private RecognitionStrategy recognitionStrategy;
   public Product(String name, RecognitionStrategy recognitionStrategy) {
      this.name = name;
      this.recognitionStrategy = recognitionStrategy;
   }
```

```
   public static Product newWordProcessor(String name) {
      return new Product(name, new CompleteRecognitionStrategy());
   }
   public static Product newSpreadsheet(String name) {
      return new Product(name, new ThreeWayRecognitionStrategy(60,90));
   }
   public static Product newDatabase(String name) {
      return new Product(name, new ThreeWayRecognitionStrategy(30,60));
   }

class RecognitionStrategy...

   abstract void calculateRevenueRecognitions(Contract contract);

class CompleteRecognitionStrategy...

   void calculateRevenueRecognitions(Contract contract) {
      contract.addRevenueRecognition(new RevenueRecognition(contract.getRevenue(),
contract.getWhenSigned()));
   }

class ThreeWayRecognitionStrategy...

   private int firstRecognitionOffset;
   private int secondRecognitionOffset;
   public ThreeWayRecognitionStrategy(int firstRecognitionOffset, int
secondRecognitionOffset)
   {
      this.firstRecognitionOffset = firstRecognitionOffset;
      this.secondRecognitionOffset = secondRecognitionOffset;
   }
   void calculateRevenueRecognitions(Contract contract) {
      Money[] allocation = contract.getRevenue().allocate(3);
      contract.addRevenueRecognition(new RevenueRecognition
            (allocation[0], contract.getWhenSigned()));
      contract.addRevenueRecognition(new RevenueRecognition
            (allocation[1], contract.getWhenSigned().addDays(firstRecognitionOffset)));
      contract.addRevenueRecognition(new RevenueRecognition
            (allocation[2], contract.getWhenSigned().addDays(secondRecognitionOffset)));
   }
```

Wielką zaletą strategii jest to, że zapewniają dobrze zintegrowane punkty rozbudowy aplikacji. Dodawanie nowego algorytmu uznawania przychodu wymaga więc utworzenia nowej podklasy, z własną metodą `calculateRevenueRecognitions`. Znacznie ułatwia to wprowadzanie do systemu nowych algorytmów.

Gdy tworzymy obiekty reprezentujące produkty, łączymy je z odpowiednimi obiektami strategii. Implementuję to w kodzie testu.

```
class Tester...

   private Product word = Product.newWordProcessor("Thinking Word");
```

```
   private Product calc = Product.newSpreadsheet("Thinking Calc");
   private Product db = Product.newDatabase("Thinking DB");
```

Gdy wszystkie elementy są gotowe, obliczanie uznania przychodu nie wymaga znajomości podklas strategii.

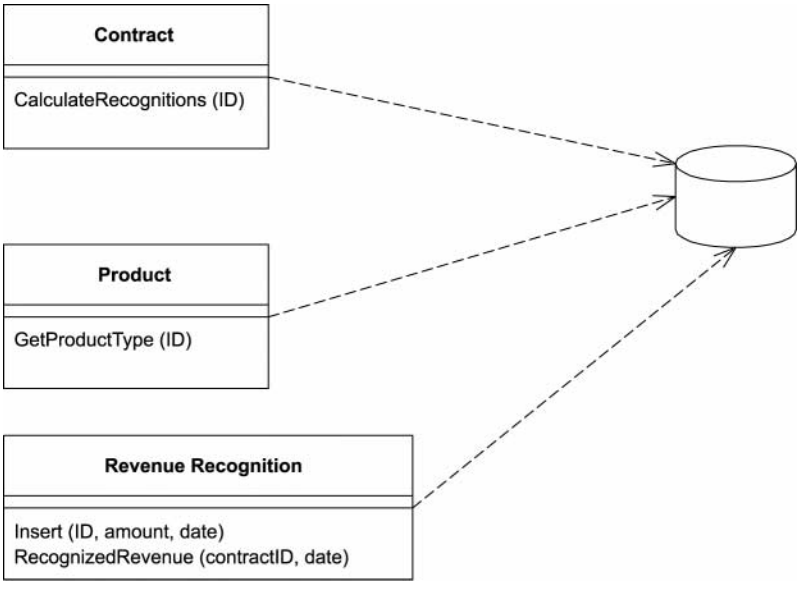
```
class Contract...

   public void calculateRecognitions() {
      product.calculateRevenueRecognitions(this);
   }

class Product...

   void calculateRevenueRecognitions(Contract contract) {
      recognitionStrategy.calculateRevenueRecognitions(contract);
   }
```

Obiektowa zasada przekazywania od obiektu do obiektu przenosi zachowanie do tego z nich, który jest najbardziej uprawniony do obsługi tego zachowania, a co więcej, realizuje większość funkcji warunkowych. Można zauważyć, że w obliczeniach nie ma żadnych instrukcji warunkowych. Wprowadzony układ obiektów sprawia, że algorytmy w naturalny sposób podążają właściwą ścieżką. Modele dziedziny sprawdzają się bardzo dobrze, gdy w systemie jest wiele podobnych warunków, bo wówczas warunki te mogą zostać zrealizowane przez samą strukturę obiektów. Przenosi to złożoność z algorytmów do samych związków między obiektami. Im bardziej podobna logika, tym częściej okazuje się, że ta sama sieć związków jest wykorzystywana przez różne części systemu. Każdy algorytm zależny od sposobu obliczania uznania przychodu może korzystać z wprowadzonego układu.

Pragnę zwrócić uwagę Czytelnika na fakt, że w tym przykładzie nie przedstawiam żadnego opisu sposobów pobierania i zapisywania obiektów do bazy danych. Wynika to z kilku przyczyn. Po pierwsze, mapowanie modelu dziedziny do bazy danych jest dość skomplikowane, więc zwyczajnie uciekam przed trudami tego opisu. Po drugie, samym celem wprowadzenia modelu dziedziny jest ukrycie bazy danych, tak przed warstwami wyższymi, jak i przed osobami, które pracują z modelem. Pominięcie opisu interakcji z bazą danych jest więc odbiciem tego, jak w rzeczywistości wygląda programowanie w środowisku opartym na wzorcu *Domain Model*.

Table Module (moduł tabeli)

Pojedynczy egzemplarz obsługuje logikę biznesową dla wszystkich wierszy tabelilub widoku bazy danych.



Jedną z podstawowych zasad podejścia obiektowego jest wiązanie danych z funkcjami (zachowaniami), które na tych danych operują. Tradycyjny schemat opiera się na obiektach o określonej tożsamości. Rozwiązania tego rodzaju reprezentuje wzorzec *Domain Model* (109). Gdy więc mamy do czynienia z klasą Pracownik, każdy egzemplarz tej klasy odpowiada pewnemu pracownikowi. Systemy tego rodzaju sprawdzają się dobrze w praktyce, ponieważ posiadanie odwołania do pracownika umożliwia wykonywanie związanych z nim operacji, podążanie za odwołaniami i gromadzenie danych o pracowniku.

Jednym z problemów charakterystycznych dla modelu dziedziny jest implementacja interfejsu relacyjnej bazy danych. Można powiedzieć, że baza relacyjna jest w takich rozwiązaniach jak ciotka-wariatka, zamknięta na strychu i wspominana w rozmowach jak najrzadziej i tylko wtedy, kiedy jest to absolutnie konieczne. Wynikiem tego są niezwykłe akrobacje, do których programista jest zmuszony w sytuacji, kiedy pojawia się potrzeba pobrania lub zapisania danych w bazie. Transformacje pomiędzy dwiema reprezentacjami danych okazują się czasochłonnym i wymagającym fragmentem pracy nad aplikacją.

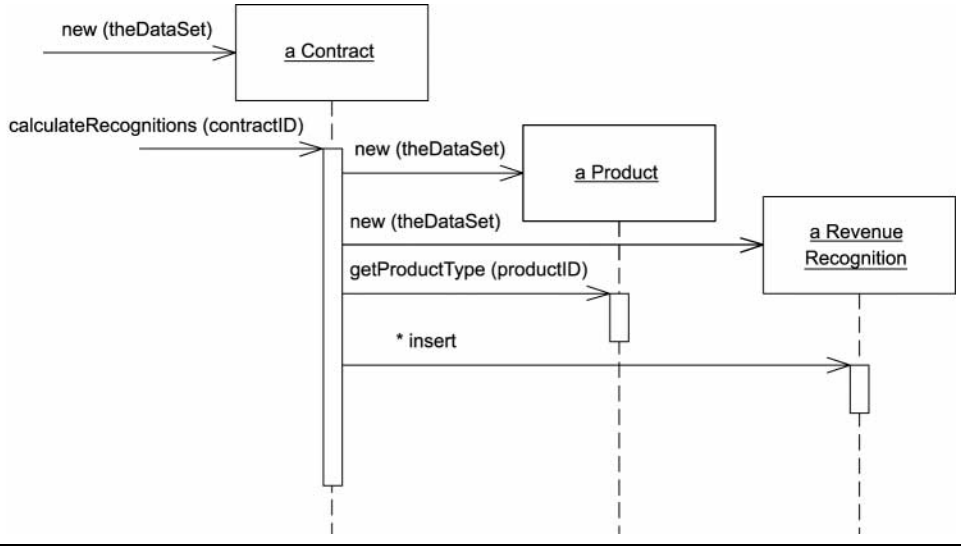
Gdy korzystamy ze wzorca *Table Module*, logika dziedziny zostaje uporządkowana w klasy, które odpowiadają poszczególnym tabelom bazy danych. Pojedynczy egzemplarz takiej klasy zawiera różnorodne procedury operujące danymi tabeli. Głównym wyróżnikiem wzorca *Domain Model* (109) jest to, że gdy mamy do czynienia z wieloma zamówieniami, jednemu zamówieniu odpowiada jeden obiekt zamówienia. Gdy korzystamy ze wzorca *Table Module*, jeden obiekt obsługuje wszystkie zamówienia.

Na czym polega

Zaletą wzorca *Table Module* jest to, że umożliwia połączenie ze sobą danych i zachowań bez utraty wartości reprezentowanych przez relacyjną bazę danych. Z wierzchu moduł tabeli wygląda jak zwykły obiekt. Główną różnicą jest brak powiązania go z tożsamością bytów, na których operuje. Aby uzyskać adres pracownika, używamy metody w rodzaju `modulPracownika.pobierzAdres(long idPracownika)`. Za każdym razem, gdy zamierzamy wykonać operację na wybranym pracowniku, musimy przekazać pewnego rodzaju odwołanie do jego tożsamości. Najczęściej jest to klucz główny tabeli bazy danych.

Wzorzec *Table Module* stosujemy zazwyczaj z pewną strukturą składowania danych opartą na tabelach. Ułożone w tabele dane są najczęściej wynikiem wywołania SQL i są przechowywane w obiekcie *Record Set* (465), którego zachowania są podobne do zachowań tabeli SQL. Zadaniem modułu tabeli jest zapewnienie interfejsu danych opartego na metodach. Grupowanie zachowań według tabel zapewnia wiele zalet hermetyzacji — funkcje pozostają blisko związane z danymi, na których operują.

Wykonanie pewnych operacji często wymaga użycia funkcji wielu modułów tabel. Niejednokrotnie więc można się spotkać z wieloma modułami, które operują na tym samym obiekcie *Record Set* (465) (patrz rysunek 9.4).



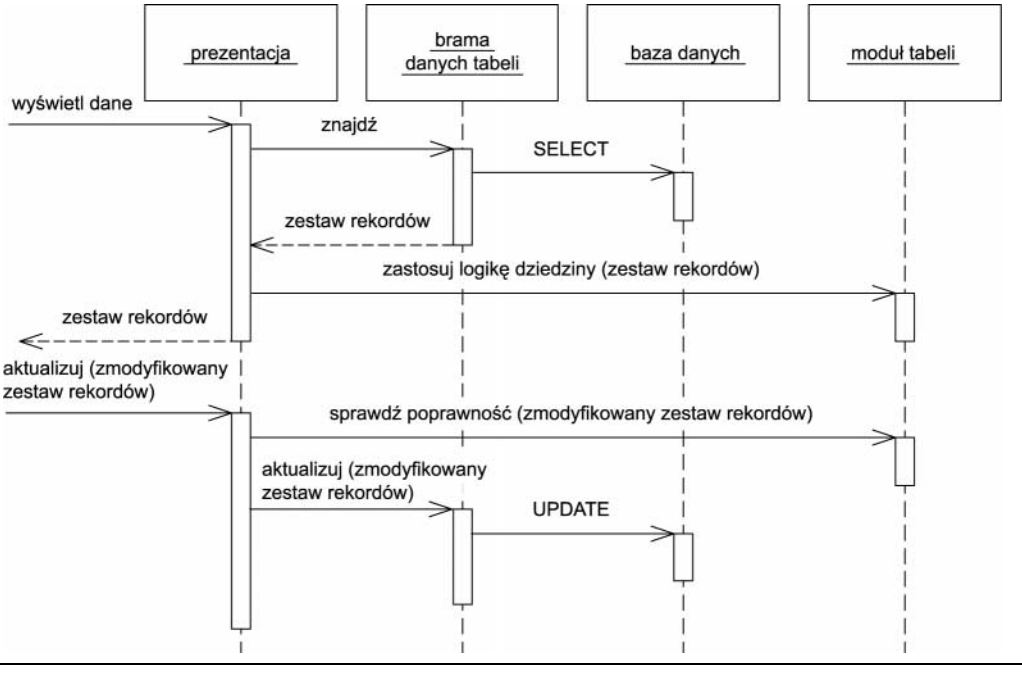
RYSUNEK 9.4. Kilka modułów tabeli może korzystać z tego samego obiektu Record Set (465)

Najbardziej przejrzystym przykładem rozwiązania opartego na modułach tabel będzie sytuacja, kiedy każdy z modułów odpowiada pojedynczej tabeli bazy danych. Jeżeli jednak w bazie danych zdefiniowane zostały pewne użyteczne zapytania i widoki, również i one mogą mieć własne moduły.

Moduł tabeli może być egzemplarzem klasy lub grupą metod statycznych. Zaletą stosowania egzemplarzy jest to, że można wówczas inicjalizować moduł tabeli przy użyciu zbioru rekordów (który może być wynikiem zapytania). Egzemplarz modułu służy w takiej sytuacji do wykonywania operacji na wierszach zestawu rekordów. Egzemplarze pozwalają również korzystać z dziedziczenia, można więc stworzyć moduł dla wybranej grupy umów, który zawiera zachowania nie mające odniesienia do każdej z nich.

Zapytania w module tabeli mogą przyjąć postać metod fabrykujących (ang. *factory methods*). Alternatywą jest wzorzec *Table Data Gateway* (133), choć wówczas w projekcie pojawia się dodatkowa klasa i związany z nią mechanizm. Zaletą takiego rozwiązania jest możliwość korzystania z pojedynczego modułu tabeli do obsługi danych z różnych źródeł, ponieważ każde z nich może obsługiwać inna brama *Table Data Gateway* (133).

Gdy używamy bramy *Table Data Gateway* (133), aplikacja wykorzystuje ją przede wszystkim do zbudowania obiektu *Record Set* (465). Obiekt ten staje się wówczas argumentem konstruktora modułu tabeli. Gdy niezbędne jest wykorzystanie wielu modułów tabeli, można utworzyć je przy użyciu tego samego zestawu rekordów. Moduł tabeli realizuje wówczas operacje logiki biznesowej, po czym przekazuje zmodyfikowany obiekt *Record Set* (465) do warstwy prezentacji, która wyświetla dane zestawu rekordów i umożliwia wprowadzanie zmian. W warstwie prezentacji można wówczas stosować widżety dostosowane do operowania danymi tabel. Widżety te nie odróżniają zestawów rekordów pobranych bezpośrednio z relacyjnej bazy danych od tych, które zostały zmodyfikowane przez moduł tabeli. Po zmianach, wprowadzonych przy użyciu graficznego interfejsu użytkownika, dane powracają do modułu tabeli w celu sprawdzenia poprawności, po czym są zapisywane w bazie danych. Jedną z zalet takiego podejścia jest możliwość testowania modułu tabeli przez stworzenie obiektu *Record Set* (465) w pamięci, bez użycia bazy danych (rysunek 9.5).



RYSUNEK 9.5. Typowe interakcje warstw otaczających moduł tabeli

Słowo „tabela” w nazwie wzorca sugeruje powiązanie każdego modułu z pojedynczą tabelą bazy danych. Jest to zasadniczo prawdą, ale nie obowiązującą regułą. Można stosować moduły tabeli dla bardziej znaczących widoków i innych zapytań. Struktura modułu tabeli nie jest ściśle uwarunkowana strukturą tabel bazy. Można więc korzystać z tabel wirtualnych, które powinny być widoczne dla aplikacji, czyli właśnie widoków i zapytań.

Kiedy używamy

Wzorzec *Table Module* opiera się na danych uporządkowanych w tabele. Korzystanie z niego jest uzasadnione, gdy korzystamy z takich danych przy użyciu obiektów *Record Set* (465). Obiekty takie stają się osią kodu aplikacji, więc dostęp do nich musi być stosunkowo prosty.

Wzorzec *Table Module* nie zapewnia pełnego wykorzystania koncepcji programowania obiektowego przy dobrze uporządkowanej, złożonej logice aplikacji. Nie można tworzyć bezpośrednich relacji między egzemplarzami obiektów. Polimorfizm również nie sprawdza się w takich rozwiązaniach. Gdy poziom złożoności jest duży, budowanie modelu dziedziny będzie lepszym podejściem. Wybór pomiędzy wzorcami *Table Module* a *Domain Model* (109) sprowadza się w zasadzie do wyboru pomiędzy potencjałem obsługi bardzo złożonej logiki a prostotą integracji z opartymi na tabelach strukturami danych.

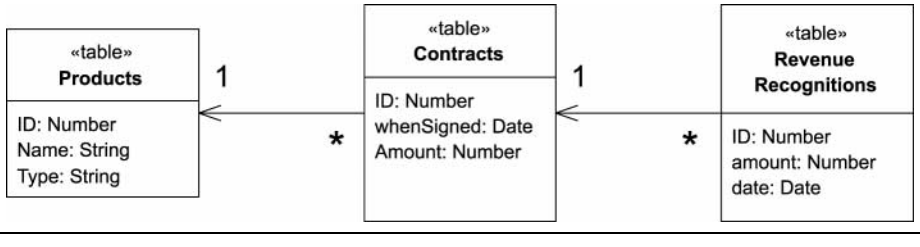
Jeżeli obiekty modelu dziedziny i tabele bazy danych mogą opierać się na podobnej organizacji, warto rozważyć połączenie wzorca *Domain Model* (109) z obiektami *Active Record* (147). Zalety modułów tabeli przewyższają korzyści z takiej kombinacji w sytuacji, gdy inne części aplikacji korzystają ze wspólnej, tabelarycznej struktury danych. Stąd bierze się niewielka popularność wzorca *Table Module* (117) w środowisku Java. I to jednak może się zmienić wraz z coraz szerszym stosowaniem mechanizmu zestawów wierszy (ang. *row set*).

Najczęściej spotykanym zastosowaniem wzorca *Table Module* są projekty oparte na mechanizmach Microsoft COM. W środowisku COM (i .NET) zestaw rekordów (*Record Set* (465)) jest podstawowym typem repozytorium danych aplikacji. Zestawy rekordów mogą być przekazywane do interfejsu użytkownika, gdzie specjalne widżety wyświetlają zawarte w nich dane. Biblioteki Microsoft ADO zapewniają niezbędny mechanizm dostępu do danych struktur relacyjnych. W takim środowisku moduły tabeli umożliwiają efektywne porządkowanie logiki biznesowej bez utraty ułatwień w obsłudze tabel, jakie zapewniają różne dostępne programiście aplikacji elementy.

Przykład: uznanie przychodu (C#)

Pora powrócić do przykładu aplikacji obliczającej uznania przychodu (opis na stronie 105), której implementacje służyły nam do zilustrowania wcześniej omawianych wzorców warstwy dziedziny. Dla przypomnienia, celem jest tu obliczenie uznania przychodu dla zamówień w sytuacji, gdy reguły obliczania różnią się w zależności od produktu, którego dane uznanie dotyczy. Mamy do czynienia z trzema produktami: edytorami tekstu, arkuszami kalkulacyjnymi i bazami danych.

System modułów tabeli opiera się na pewnym schemacie danych, który zazwyczaj jest modelem relacyjnym (chociaż w przyszłości w podobnych schematach spotkamy się zapewne z modelami XML). W tym przypadku podstawą jest schemat relacyjny przedstawiony na rysunku 9.6.



RYSUNEK 9.6. Schemat bazy danych dla przykładu obliczania uznań przychodu

Klasy operujące danymi uporządkowane są bardzo podobnie; każdej tabeli odpowiada jeden moduł tabeli. W architekturze .NET reprezentację struktury bazy danych w pamięci zapewnia obiekt zbioru danych (ang. *data set*). Właśnie na takich obiektach powinny operować tworzone klasy. Każda klasa modułu tabeli ma pole typu `DataTable`. Jest to klasa, która w systemie .NET odpowiada pojedynczej tabeli zbioru danych. Mogą z niej korzystać wszystkie moduły tabeli i można ją traktować jako wzorzec *Layer Supertype* (434).

```
class TableModule...

   protected DataTable table;
   protected TableModule(DataSet ds, String tableName) {
      table = ds.Tables[tableName];
   }
```

Konstruktor podklasy wywołuje konstruktor superklasy korzystając ze wskazanej nazwy tabeli.

```
class Contract...

   public Contract (DataSet ds) : base (ds, "Contracts") {}
```

Umożliwia to utworzenie nowego modułu tabeli przez proste przekazanie zbioru danych do konstruktora modułu.

```
contract = new Contract(dataset);
```

Utrzymujemy w ten sposób kod odpowiedzialny za tworzenie zbioru danych poza modułami tabeli, co odpowiada zasadom korzystania z ADO.NET.

Wygodną cechą języka C# jest indeksator (ang. *indexer*), który umożliwia dostęp do wybranego wiersza danych tabeli w oparciu o klucz główny.

```
class Contract...

   public DataRow this [long key] {
   get {
      String filter = String.Format("ID = {0}", key);
      return table.Select(filter)[0];
      }
   }
```

Pierwszy fragment kodu, w którym implementujemy zachowania, oblicza uznanie przychodu dla umowy, aktualizując odpowiednio tabelę uznań. Uznawana kwota zależy od produktu, którego dotyczy. Ponieważ korzystamy przy tym głównie z danych tabeli umów, włączamy odpowiednią metodę do klasy umów.

```
class Contract...

   public void CalculateRecognitions (long contractID) {
      DataRow contractRow = this[contractID];
      Decimal amount = (Decimal)contractRow["amount"];
      RevenueRecognition rr = new RevenueRecognition (table.DataSet);
      Product prod = new Product(table.DataSet);
      long prodID = GetProductId(contractID);
```

```
      if (prod.GetProductType(prodID) == ProductType.WP) {
         rr.Insert(contractID, amount, (DateTime) GetWhenSigned(contractID));
      } else if (prod.GetProductType(prodID) == ProductType.SS) {
         Decimal[] allocation = allocate(amount, 3);
         rr.Insert(contractID, allocation[0], (DateTime) GetWhenSigned(contractID));
         rr.Insert(contractID, allocation[1], (DateTime)
            GetWhenSigned(contractID).AddDays(60));
         rr.Insert(contractID, allocation[2], (DateTime)
            GetWhenSigned(contractID).AddDays(90));
      } else if (prod.GetProductType(prodID) == ProductType.DB) {
         Decimal[] allocation = allocate(amount, 3);
         rr.Insert(contractID, allocation[0], (DateTime) GetWhenSigned(contractID));
         rr.Insert(contractID, allocation[1], (DateTime)
            GetWhenSigned(contractID).AddDays(30));
         rr.Insert(contractID, allocation[2], (DateTime)
            GetWhenSigned(contractID).AddDays(60));
      } else throw new Exception("nieznany identyfikator produktu");
   }
   private Decimal[] allocate(Decimal amount, int by) {
      Decimal lowResult = amount / by;
      lowResult = Decimal.Round(lowResult, 2);
      Decimal highResult = lowResult + 0.01m;
      Decimal[] results = new Decimal[by];
      int remainder = (int) amount % by;
      for (int i = 0; i < remainder; i++) results[i] = highResult;
      for (int i = remainder; i < by; i++) results[i] = lowResult;
      return results;
   }
```

Choć we wcześniejszych przykładach używałem w tym miejscu obiektu *Money* (446), tutaj dla zróżnicowania wprowadziłem typ `Decimal`. Metoda alokacji jest podobna jak w przypadku klasy *Money* (446).

Do przeprowadzenia takich operacji niezbędne są pewne funkcje zdefiniowane w innych klasach. Musimy mieć możliwość pobrania typu każdego produktu. Możliwość tę zapewnimy sobie, wprowadzając enumerację typów i metodę pobierającą odpowiednią wartość.

```
   public enum ProductType {WP, SS, DB};

class Product...

   public ProductType GetProductType (long id) {
      String typeCode = (String) this[id]["type"];
      return (ProductType) Enum.Parse(typeof(ProductType), typeCode);
   }
```

Metoda `GetProductType` hermetyzuje dane tabeli. Ogólnie rzecz biorąc, bezpośredni odczyt sumy umowy z kolumny tabeli (jak w przykładzie powyżej) nie jest najlepszym podejściem. Zasada hermetyzacji powinna objąć poszczególne kolumny danych. Rozwiązanie takie wybrałem ze względu na założenie, że pracujemy w środowisku, w którym różne części systemu korzystają z bezpośredniego dostępu do zbioru danych. Gdy zbiór danych jest przekazywany do interfejsu użytkownika, hermetyzacja również nie jest stosowana. Funkcje dostępu do kolumn stosujemy tylko wtedy, gdy ma to służyć wprowadzeniu dodatkowej funkcjonalności, takiej jak konwersja ciągu na typ `ProductType`.

Warto w tym miejscu wspomnieć również o tym, że choć w przykładzie stosujemy zbiór danych o nieokreślonych typach (ponieważ jest to częściej stosowane na różnych innych platformach), w środowisku .NET zaleca się ścisłe określanie typów (strona 466).

Kolejną niezbędną funkcją jest wstawianie nowego rekordu uznania przychodu.

```
class RevenueRecognition...

   public long Insert (long contractID, Decimal amount, DateTime date) {
      DataRow newRow = table.NewRow();
      long id = GetNextID();
      newRow["ID"] = id;
      newRow["contractID"] = contractID;
      newRow["amount"] = amount;
      newRow["date"] = String.Format("{0:s}", date);
      table.Rows.Add(newRow);
      return id;
   }
```

Również ta metoda służy nie tyle hermetyzacji wiersza danych, co przede wszystkim wprowadzeniu metody w miejsce kilku wierszy kodu, które musiałyby być powtarzane w różnych miejscach aplikacji.

Drugą funkcją jest sumowanie wszystkich przychodów umowy uznanych do określonego dnia. Ponieważ korzysta ona z tabeli uznań przychodów, metodę definiujemy w odpowiadającej tej tabeli klasie.

```
class RevenueRecognition...

   public Decimal RecognizedRevenue (long contractID, DateTime asOf) {
      String filter = String.Format("ContractID = {0} AND date <= #{1:d}#",
      contractID, asOf);
      DataRow[] rows = table.Select(filter);
      Decimal result = 0m;
      foreach (DataRow row in rows) {
         result += (Decimal)row["amount"];
      }
      return result;
   }
```

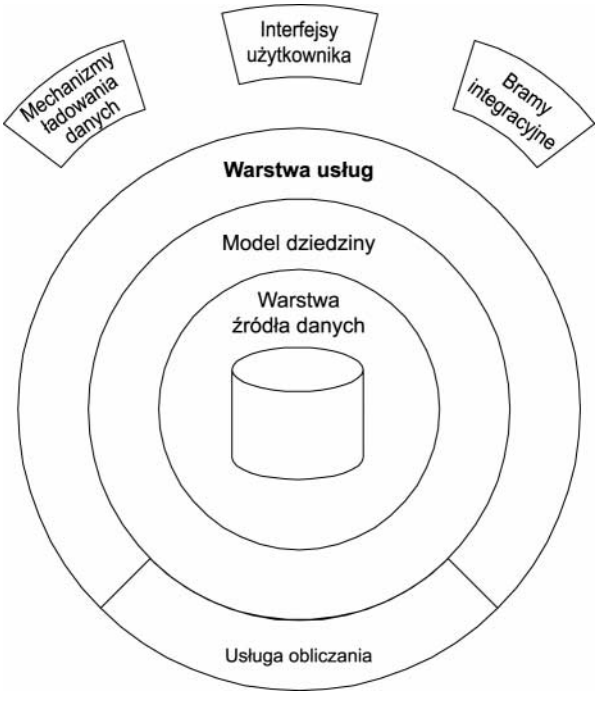
Ten fragment korzysta z bardzo wygodnego mechanizmu ADO.NET, który umożliwia (bez użycia języka SQL) definiowanie klauzuli `WHERE` i wybieranie podzbioru danych tabeli, na których mają zostać wykonane operacje. W praktyce, w tak prostym przykładzie można pójść jeszcze dalej i użyć funkcji agregującej.

```
class RevenueRecognition...

   public Decimal RecognizedRevenue2 (long contractID, DateTime asOf) {
      String filter = String.Format("ContractID = {0} AND date <= #{1:d}#",
      contractID, asOf);
      String computeExpression = "sum(amount)";
      Object sum = table.Compute(computeExpression, filter);
      return (sum is System.DBNull) ? 0 : (Decimal) sum;
   }
```

Service Layer (warstwa usług)

Randy Stafford

Definiuje granicę aplikacji przez wprowadzenie warstwy, która określa zbiór dostępnych operacji i koordynuje odpowiedzi aplikacji dla każdej z nich.



Aplikacje dla przedsiębiorstw wymagają często różnych interfejsów danych, na których operują, i logiki, którą implementują: mechanizmów ładowania danych, interfejsów użytkownika, bram integracji i innych. Pomimo, że są przeznaczone do różnych celów, interfejsy te często wymagają tych samych mechanizmów interakcji z aplikacją, niezbędnych, aby uzyskać dostęp i operować danymi, jak również do wywoływania funkcji logiki biznesowej. Interakcje te mogą być bardzo złożone, mogą obejmować transakcje operujące na różnych zasobach i wymagać koordynacji wielu odpowiedzi aplikacji na pojedyncze wywołanie. Kodowanie logiki interakcji w każdym z interfejsów niezależnie prowadzi wówczas do znacznej ilości powtórzeń.

Warstwa usług definiuje granicę aplikacji *Cockburn PloP* i zbiór operacji dostępnych warstwom klienckim. Hermetyzuje ona logikę biznesową, zapewniając sterowanie transakcjami i koordynację odpowiedzi aplikacji, generowane przez właściwe implementacje operacji.

Na czym polega

Warstwa usług może być implementowana kilkoma sposobami, z których każdy odpowiada przedstawionej powyżej charakterystyce. Różnice występują w podziale funkcjonalności wspierającej interfejs tej warstwy. Zanim przedstawię bliżej różne możliwości implementacji, zapoznajmy się z podstawą koncepcyjną wzorca *Service Layer*.

Typy logiki biznesowej

Podobnie jak *Transaction Script* (103) i *Domain Model* (109), wzorzec *Service Layer* (124) służy do porządkowania logiki biznesowej. Wielu projektantów — wśród nich i ja — dzieli logikę biznesową (ang. *business logic*) na dwa rodzaje: logikę dziedziny (ang. *domain logic*), związaną wyłącznie z dziedziną problemu (jak strategie obliczania uznania przychodu z umowy), i logikę aplikacji (ang. *application logic*), związaną z funkcjami aplikacji *Cockburn UC* (jak powiadamianie administratorów umów i aplikacji zintegrowanych o obliczeniach uznań przychodu). Logikę aplikacji określa się czasem terminem „logika pracy" (ang. *workflow logic*), choć termin „praca" (albo „przepływ pracy") bywa różnorodnie interpretowany.

Model dziedziny ma tę przewagę na skryptami transakcji, że dzięki zastosowaniu klasycznych wzorców projektowych sprzyja unikaniu duplikacji kodu dziedziny i ułatwia zarządzanie złożonością. Jednak umieszczenie logiki aplikacji w klasach obiektów dziedziny ma kilka niepożądanych konsekwencji. Po pierwsze, klasy obiektów dziedziny, które zawierają logikę specyficzną dla określonej aplikacji i są zależne od jej pakietów, nie mogą być używane w innych aplikacjach. Po drugie, połączenie obu rodzajów logiki w tych samych klasach utrudnia powtórne zaimplementowanie, gdy przyjdzie taka potrzeba, logiki aplikacji (na przykład w narzędziu do zarządzania przepływem pracy). Wprowadzenie warstwy usług umożliwia oddzielenie dwóch rodzajów logiki biznesowej. Oznacza to nie tylko korzyści typowe dla podziału warstwowego aplikacji, ale również czyste klasy obiektów dziedziny, które łatwiej przenosić między aplikacjami.

Możliwości implementacji

Dwie podstawowe opcje implementacji to fasada dziedziny i skrypty operacji. **Fasada dziedziny** (ang. *domain facade*) to zbiór prostych osłon modelu dziedziny. Implementujące je klasy nie zawierają logiki biznesowej, która w całości pozostaje w modelu dziedziny. Osłony wyznaczają granicę aplikacji i zbiór operacji, które warstwy klienckie mogą wykorzystywać do interakcji z nią.

Skrypty operacji (ang. *operation script*) to zbiór nieco bardziej rozbudowanych klas, które bezpośrednio implementują logikę aplikacji, delegując zarazem logikę dziedziny do hermetycznych klas obiektów dziedziny. Operacje dostępne klientom warstwy usług są implementowane jako skrypty uporządkowane według obszarów tematycznych logiki. Każda taka klasa stanowi „usługę" aplikacji, stąd często umieszczane w ich nazwach słowo „Service" (usługa). Zbiór takich klas tworzy warstwę usług. Powinny one być rozszerzeniem klasy *Layer Supertype* (434), która ogólnie definiuje zakres ich funkcji i wspólne zachowania.

Wywołania zdalne

Interfejs warstwy usług ma niską ziarnistość. Wynika to w oczywisty sposób stąd, że jest to zestaw operacji aplikacji, które są dostępne dla korzystających z niej warstw klienckim. W konsekwencji, klasy warstwy usług dość dobrze nadają się do realizacji wywołań zdalnych.

Z drugiej strony, zdalne wywołania wiążą się z kosztem rozproszenia obiektów. Wprowadzenie do warstwy usług obiektów *Data Transfer Object* (366) wymaga zazwyczaj dość dużo pracy. Ten dodatkowy koszt warto potraktować poważnie, zwłaszcza gdy model dziedziny jest skomplikowany, a interfejs użytkownika rozbudowany pod kątem obsługi złożonych przypadków aktualizacji

danych. Jest to praca znacząca i czasochłonna, porównywalna chyba tylko z mapowaniem obiektowo-relacyjnym. Nigdy nie wolno więc zapominać o „pierwszym prawie projektowania dla obiektów rozproszonych" (strona 87).

Najrozsądniejszym podejściem do budowy warstwy usług będzie więc stworzenie metod przeznaczonych do wywołań lokalnych, których sygnatury operują obiektami dziedziny. Możliwość wywołań zdalnych można dodać dopiero wtedy, gdy okaże się faktycznie niezbędna. Korzystamy wówczas ze wzorca *Remote Facade* (353), który uzupełnia gotową warstwę, przystosowaną do użytku lokalnego (można też wprowadzić interfejsy zdalne bezpośrednio do obiektów warstwy usług). Jeżeli aplikacja jest wyposażona w interfejs przeglądarki WWW lub usług WWW, nie oznacza to jeszcze, że logika biznesowa musi pracować w innym procesie niż strony serwera czy usługi sieci Web. W rzeczywistości, można oszczędzić sobie pracy i skrócić czas reakcji aplikacji stosując rozwiązanie kolokowane. Nie oznacza to wcale ograniczenia skalowalności.

Identyfikacja usług i operacji

Identyfikowanie operacji, które powinna zapewniać wyznaczana przez warstwę usług granica, jest stosunkowo proste. Wyznaczają je potrzeby klientów, z których najbardziej znaczącym (i pierwszym) jest najczęściej interfejs użytkownika. Ponieważ jest on zaprojektowany pod kątem przypadków użycia, punktami wyjścia stają się potrzeby aktorów, opracowane wcześniej przypadki użycia i projekt interfejsu użytkownika aplikacji.

Wiele przypadków użycia aplikacji korporacyjnej to dość nudne operacje „utwórz", „odczytaj", „aktualizuj", „usuń" — dla różnych obiektów dziedziny. Tworzymy więc jeden obiekt pewnego rodzaju, odczytujemy kolekcję innych, aktualizujemy jeszcze inne itp. Doświadczenie wykazuje, że właśnie takie przypadki użycia mają swoje miejsce jako wzór dla funkcji warstwy usług.

Choć przypadki użycia prezentują się stosunkowo prosto, niezbędne do ich realizacji czynności aplikacji są zazwyczaj znacznie ciekawsze. Poza sprawdzaniem poprawności danych oraz tworzeniem, aktualizowaniem i usuwaniem obiektów dziedziny, coraz częściej wymagane jest powiadamianie o wykonanych operacjach zarówno osób, jak i innych aplikacji. Tego rodzaju zdarzenia wymagają koordynacji i całościowego podejścia do ich przetwarzania. To właśnie jest zadaniem warstwy usług.

Nie jest łatwo wyznaczyć zasady grupowania powiązanych operacji warstwy usług. Trudno tu mówić o gotowych regułach postępowania. W przypadku niewielkiej aplikacji wystarczyć może jedna taka grupa, nazwana tak samo jak aplikacja. Większe aplikacje dzieli się na kilka „podsystemów", z których każdy zawiera kompletny „pion" elementów poszczególnych warstw architektury. W takim przypadku każdemu podsystemowi można przypisać jedną abstrakcję warstwy usług, dziedziczącą nazwę po podsystemie, do którego dostęp zapewnia. Alternatywą może być grupowanie według partycji modelu dziedziny (np. `ContractsService`, `ProductsService`) albo zagadnień, z którymi związane są realizowane zachowania (np. `RecognitionService`).

Implementacja w języku Java

Zarówno fasada dziedziny, jak i skrypty operacji mogą być implementowane przy użyciu obiektów POJO lub bezstanowych obiektów bean sesji. Wybieramy tutaj pomiędzy łatwością testowania a łatwością sterowania transakcjami. Zwykłe obiekty języka Java jest łatwiej testować, ponieważ nie wymagają do pracy kontenera EJB. Trudniej je natomiast powiązać z rozproszonymi, zarządzanymi kontenerowo usługami transakcji, zwłaszcza przy wywołaniach pomiędzy usługami. Obiekty EJB zapewniają obsługę kontenerowo zarządzanych transakcji rozproszonych, ale każde uruchomienie lub test musi odbywać się w środowisku kontenera. Wybór nie jest więc łatwy.

Moją ulubioną techniką implementacji warstwy usług w J2EE są bezstanowe obiekty *session bean* EJB 2.0, wyposażone w interfejsy lokalne. Korzystam przy tym ze skryptów operacji, które delegują do klas obiektów dziedziny, zaimplementowanych jako obiekty POJO. Rozproszone, zarządzane kontenerowo transakcje, jakie zapewnia środowisko EJB, znakomicie ułatwiają implementowanie warstwy usług przy użyciu bezstanowych obiektów *session bean*. Wprowadzone w EJB 2.0 interfejsy lokalne umożliwiają warstwie usług wykorzystanie cennych usług transakcji, nie zmuszając zarazem do wprowadzania obiektów rozproszonych.

Omawiając implementację w języku Java, warto zwrócić uwagę na różnice między wzorcem *Service Layer* a wzorcem *Session Facade*, opisywanym w literaturze J2EE (*Alur et al.* i *Marinescu*). *Session Facade* ma na celu uniknięcie obniżenia wydajności, wynikającego z nadmiernej liczby zdalnych wywołań obiektów *entity bean*. Stąd osłonięcie obiektów *entity bean* obiektami *session bean*. *Service Layer* to wzorzec, który dzieli implementację w celu uniknięcia powtórzeń kodu i ułatwienia jego powtórnego użycia. Jest to wzorzec architektury niezależny od wykorzystywanej technologii. Warto przypomnieć, że opisany w *Cockburn PloP* wzorzec granicy aplikacji, który był inspiracją dla wzorca *Service Layer*, powstał trzy lata wcześniej niż środowisko EJB. Wzorzec *Session Facade* może przypominać w swojej idei warstwę usług, nie jest jednak tym samym.

Kiedy używamy

Podstawowe korzyści z wprowadzenia warstwy usług to definicja jednolitego zbioru operacji aplikacji, który jest dostępny wielu rodzajom klientów, oraz koordynacja odpowiedzi aplikacji. Odpowiedź taka może wymagać logiki aplikacji, która zapewnia całościowe przetwarzanie operacji, oparte na wielu zasobach transakcyjnych. Aplikacja, która ma więcej niż jednego klienta korzystającego z jej logiki biznesowej, i realizuje złożone odpowiedzi oparte na wielu zasobach transakcyjnych, jest niewątpliwie dobrym kandydatem do wprowadzenia warstwy usług z kontenerowo zarządzanymi transakcjami, nawet jeżeli nie korzysta z architektury rozproszonej.

Prościej jest prawdopodobnie odpowiedzieć na pytanie, kiedy nie warto wprowadzać warstwy usług. Nie jest ona zazwyczaj uzasadniona, gdy logika biznesowa aplikacji ma tylko jednego klienta, na przykład interfejs użytkownika, a jej przypadki użycia nie przewidują korzystania z wielu zasobów transakcyjnych. W takich przypadkach do kontroli transakcji i koordynowania odpowiedzi może posłużyć mechanizm *Page Controller* (302). Może on też delegować bezpośrednio do warstwy źródła danych.

Gdy tylko pojawia się koncepcja wprowadzenia drugiego klienta lub drugiego zasobu transakcyjnego, warto wprowadzić warstwę usług już na początku projektu.

Warto przeczytać

Niewiele napisano dotąd o wzorcu *Service Layer* (124). Jego pierwowzorem był wzorzec granicy aplikacji, przedstawiony przez Alistair Cockburn (*Cockburn PloP*). Praca *Alpert et al.* omawia rolę fasad w systemach rozproszonych. Dla porównania warto zapoznać się z różnymi opisami wzorca *Session Facade — Alur et al.* i *Marinescu*. Z zagadnieniem funkcji aplikacji, które muszą być koordynowane przez warstwę usług, powiązany jest opis przypadków użycia jako kontraktu zachowań, przedstawiony w *Cockburn UC*. Wcześniejszą pracą teoretyczną jest *Coleman et al.*, gdzie mowa o rozpoznawaniu „operacji systemowych" w metodologii Fusion.

Przykład: uznanie przychodu (Java)

Przedstawię teraz kolejną wersję przykładu, który służy nam jako ilustracja od początku rozdziału. Tym razem zaprezentuję na nim zastosowanie wzorca *Service Layer*, a więc podział na logikę aplikacji i logikę dziedziny. Warstwa usług będzie miała postać skryptu operacji, który zaimplementujemy najpierw przy użyciu obiektów POJO, a następnie obiektów EJB.

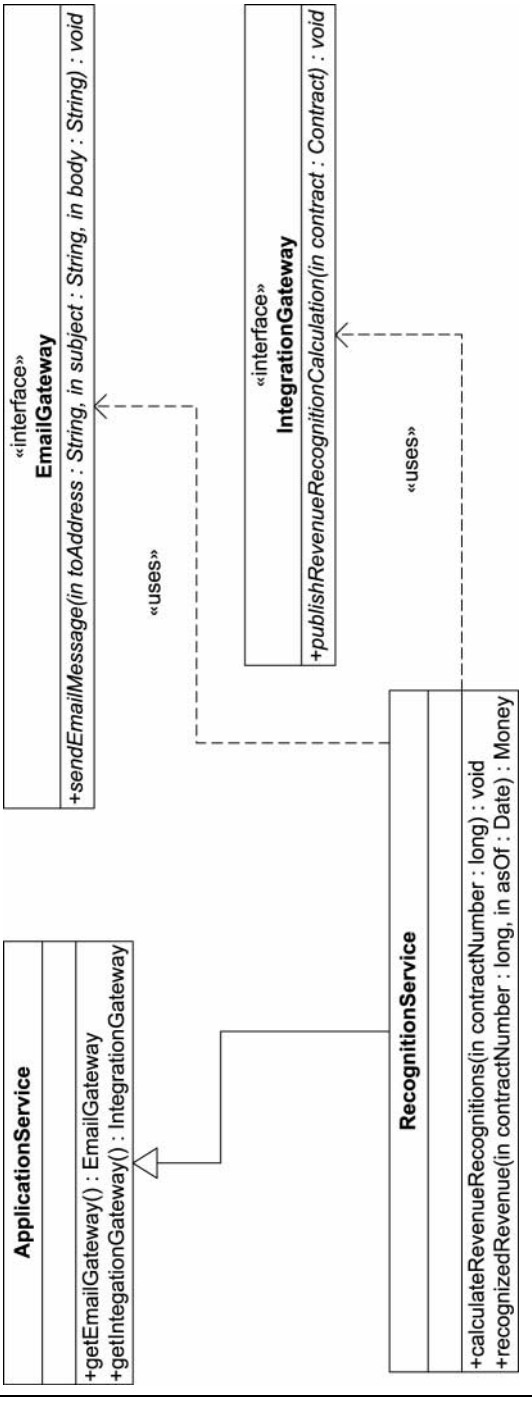
Rozwiniemy teraz nieco podstawowy scenariusz, aby wprowadzić do niego elementy logiki aplikacji. Załóżmy, że przypadki użycia aplikacji wymagają, aby obliczaniu uznań przychodu dla umowy towarzyszyło przesłanie powiadomienia e-mail do wskazanego „administratora umów" oraz opublikowanie przy użyciu oprogramowania typu middleware wiadomości, która zapewni przekazanie informacji aplikacjom zintegrowanym.

Rozpoczynamy od zmodyfikowania klasy `RecognitionService` z przykładu ilustrującego wzorzec *Transaction Script* (103) tak, aby rozszerzyć klasę *Layer Supertype* (434) i wprowadzić kilka klas *Gateway* (425), które posłużą do realizacji logiki aplikacji. Będzie to odpowiadać diagramowi klas przedstawionemu na rysunku 9.7. Klasa `RecognitionService` stanie się implementacją warstwy usług opartą na obiektach POJO, a jej metody będą reprezentować dwie dostępne na granicy aplikacji operacje.

Metody klasy `RecognitionService` realizują logikę aplikacji, delegując logikę dziedziny do klas obiektów dziedziny, wziętych z przykładu ilustrującego wzorzec *Domain Model* (109).

```
public class Application Service {
   protected EmailGateway getEmailGateway() {
      //zwraca egzemplarz bramy poczty elektronicznej
   }
   protected IntegrationGateway getIntegrationGateway() {
      //zwraca egzemplarz bramy integracji
   }
}
public interface EmailGateway {
   void sendEmailMessage(String toAddress, String subject, String body);
}
public interface IntegrationGateway {
   void publishRevenueRecognitionCalculation(Contract contract);
}
public class RecognitionService
extends ApplicationService {
   public void calculateRevenueRecognitions(long contractNumber) {
      Contract contract = Contract.readForUpdate(contractNumber);
      contract.calculateRecognitions();
      getEmailGateway().sendEmailMessage(
         contract.getAministratorEmailAddress(),
         "Odp: Umowa #" + contractNumber,
         contract + ". Obliczono uznania przychodu.");
      getIntegrationGateway().publishRevenueRecognitionCalculation(contract);
   }
   public Money recognizedRevenue(long contractNumber, Date asOf) {
      return Contract.read(contractNumber).recognizedRevenue(asOf);
   }
}
```

W naszym przykładzie nie będziemy zajmować się kwestiami magazynowania danych, ograniczając się do stwierdzenia, że klasa `Contract` implementuje statyczne metody do odczytywania umów z warstwy źródła danych według ich numerów. Nazwa jednej z tych metod (`readForUpdate`) sygnalizuje zamiar aktualizacji odczytywanej umowy, co umożliwia mechanizmowi *Data Mapper* (152) zarejestrowanie odczytywanego obiektu przy użyciu, na przykład, schematu *Unit of Work* (169).



RYSUNEK 9.7. Diagram klas POJO dla usługi obliczania uznań przychodu

Nie będziemy również opisywać szerzej sterowania transakcjami. Metoda `calculateRevenueRecognitions()` jest z natury transakcyjna, ponieważ w trakcie jej wykonywania modyfikowane są trwałe obiekty umów (przed dodaniem uznań przychodu); do oprogramowania middleware przekazywane są wiadomości; wysyłane są wiadomości pocztowe. Wszystkie te odpowiedzi aplikacji muszą być przetwarzane jako całość, bo nie chcemy wysyłać wiadomości e-mail lub publikować wiadomości dla innych aplikacji, jeżeli zmiany w umowie nie zostały trwale zapisane.

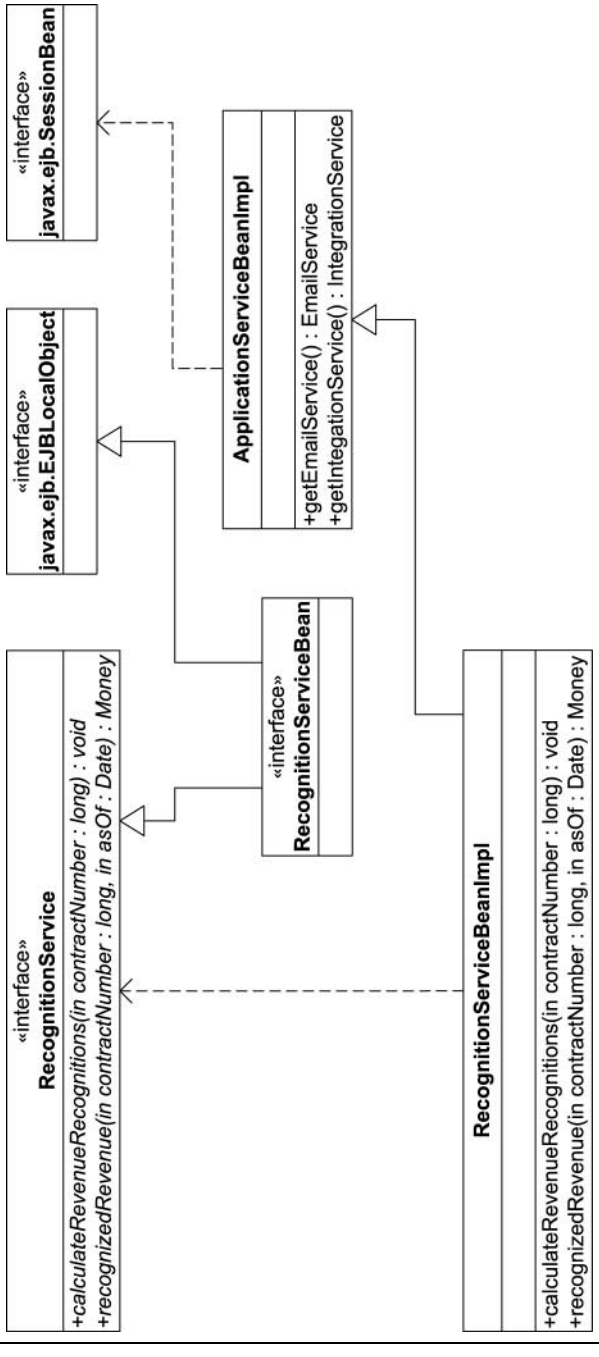
Na platformie J2EE zarządzanie transakcjami rozproszonymi można pozostawić kontenerowi EJB. W tym celu implementujemy usługi aplikacji (i obiekty *Gateway* (425)) jako bezstanowe obiekty *session bean*, korzystające z zasobów transakcyjnych. Rysunek 9.8 przedstawia diagram klas implementacji usługi obliczania uznań przychodu, korzystającej z interfejsów lokalnych EJB 2.0 i idiomu „interfejsu biznesowego" (ang. *business interface*). W tej implementacji wciąż korzystamy z klasy *Layer Supertype* (434), zapewniającej metody obiektów bean, których wymaga mechanizm EJB, oraz metody specyficzne dla aplikacji. Jeżeli założymy, że interfejsy `EmailGateway` i `IntegrationGateway` są również „interfejsami biznesowymi" odpowiednich bezstanowych obiektów *session bean*, to sterowanie transakcją rozproszoną uzyskamy przez zadeklarowanie metod `calculateRevenueRecognitions`, `sendEmailMessage` i `publishRevenueRecognitionCalculation` jako transakcyjnych. Metody `RecognitionService` z przykładu opartego na obiektach POJO zostają przeniesione w niezmienionej postaci do klasy `RecognitionServiceBeanImpl`.

Ważnym elementem w tym przykładzie jest fakt, że warstwa usług używa do koordynowania transakcyjnych odpowiedzi aplikacji zarówno skryptów operacji, jak i klas obiektów dziedziny. Metoda `calculateRevenueRecognitions` realizuje logikę aplikacji wymaganą przez przypadki użycia, ale deleguje logikę dziedziny do klas obiektów dziedziny. Zastosowanych jest też kilka technik unikania powtórzeń kodu w skryptach operacji, które tworzą warstwę usług. Część funkcji jest przeniesiona do osobnych obiektów *Gateway* (425), które mogą być ponownie wykorzystywane przez zastosowanie delegacji. *Layer Supertype* (434) zapewnia wygodny dostęp do tych obiektów.

Czytelnik mógłby stwierdzić, że zastosowanie wzorca *Observer* i *Gang of Four* pozwalałoby uzyskać bardziej elegancką implementację skryptu operacji. Jednak wzorzec *Observer* byłby trudny do zaimplementowania w bezstanowej i wielowątkowej warstwie usług. Otwarty kod skryptu operacji okazuje się bardziej przejrzysty i prostszy.

Można również stwierdzić, że funkcje logiki aplikacji mogłyby zostać zaimplementowane w metodach obiektów dziedziny, takich jak `Contract.calculateRevenueRecognitions()`, lub nawet w warstwie źródła danych, co wyeliminowałoby potrzebę stosowania osobnej warstwy usług. Taka alokacja funkcji wydaje się jednak niepożądana z kilku powodów. Po pierwsze, klasy obiektów dziedziny gorzej poddają się próbom ponownego użycia w innych aplikacjach, gdy zawierają logikę specyficzną dla jednego rozwiązania (i pozostają zależne od specyficznych dla aplikacji obiektów (*Gateway* (425)). Ich zadaniem jest modelowanie części dziedziny problemu, z którymi aplikacja jest związana, co jednak nie oznacza wszystkich funkcji wyznaczanych przez przypadki użycia tejże aplikacji. Po drugie, hermetyzacja logiki aplikacji w „wyższej", przeznaczonej wyłącznie do tego celu warstwie (czego nie można powiedzieć o warstwie źródła danych) ułatwia wprowadzanie w niej zmian. Ich celem może być choćby wprowadzenie motoru zarządzania przepływem pracy.

Jako wzorzec organizacji warstwy logicznej aplikacji korporacyjnej, *Service Layer* łączy w sobie skrypty i klasy obiektów dziedziny, korzystając z najlepszych cech obu podejść. Jego implementacja może być przeprowadzona różnymi metodami: przy użyciu fasad dziedziny lub skryptów operacji, obiektów POJO lub obiektów bean sesji (albo jednych i drugich), z orientacją na wywołania lokalne lub zdalne (albo oba rodzaje). Co najważniejsze, niezależnie od tego, którą implementację wybierzemy, będzie ona hermetyczną realizacją logiki biznesowej aplikacji, zapewniającą spójny interfejs tej logiki różnym warstwom klienckim.



RYSUNEK 9.8. Diagram klas EJB dla usługi obliczania uznań przychodu