helion.pl

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

J2EE. Wzorce projektowe



Autorzy: Deepak Alur, John Crupi, Dan Malks
Tłumaczenie: Maciel Gorywoda
ISBN: 83-7197-988-6
Tytuł oryginału: Core J2EE Patterns
Format: B5, stron: 360

W ciągu ostatnich kilku lat Java 2 Enterprise Edition stała się standardową platformą do budowy skomplikowanych aplikacji. Ponieważ jest ona rozbudowanym środowiskiem programistycznym, pozwalającym projektować i programować potężne aplikacje, nie jest łatwa w obsłudze. W książce, którą trzymasz w ręku, doświadczeni architekci Sun Java Center, organizacji konsultingowej, działającej w ramach Sun Java, dzielą się z czytelnikiem swoim doświadczeniem w projektowaniu aplikacji opartych na technologii J2EE.

Autorzy skupiają się na wzorcach projektowych, opartych na najważniejszych technologiach J2EE, takich jak Java Server Pages (JSP), serwlety, Enterprise JavaBeans (EJB) oraz Java Message Service (JMS). Omawiane są również inne technologie Javy, ważne dla prawidłowej implementacji wzorców, takie jak JDBC i JNDI. Katalog zawierający wzorce projektowe J2EE dokumentuje i prezentuje najlepsze sposoby wykorzystania tych technologii. To jednak nie wszystko. Książka omawia również:

- wiele strategii przydatnych przy projektowaniu warstw prezentacyjnej i biznesowej;
- identyfikację błędnych technik znajdywanych w warstwach prezentacyjnej, biznesowej i integracyjnej oraz propozycje rozwiązań wykorzystujących wzorce projektowe oraz sposoby przebudowy systemów;
- metody przebudowy dla poszczególnych warstw aplikacji oraz techniki zastępowania błędnych implementacji prawidłowymi;
- przykładowy kod programów opartych na omawianych strategiach i wzorcach projektowych.

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

Spis treści

Podziękowania 9

Przedsłowie I 11

Przedsłowie II 12

Wstęp 13

Część I Wzorce i J2EE 19

Rozdział 1. Wprowadzenie 21

Czym jest J2EE? 21
Czym są wzorce? 22
Historia 22
Definicja wzorca 23
Kategorie wzorców 24
Katalog wzorców J2EE 25
Nieprzerwana ewolucja 25
Jak korzystać z katalogu wzorców J2EE 26
Korzyści wynikające z używania wzorców 27
Wzorce, struktury oraz powtórne wykorzystanie 29
Podsumowanie 29

Rozdział 2. Omówienie platformy J2EE 31

Krótko o historii 31
Serwery aplikacji — nowe pokolenie 32
Właściwości technologii Javy 33
Powstanie platformy J2EE 34
Ocena wartości J2EE 34
Platforma J2EE 35
Architektura J2EE 35
Java 2 Standard Edition 36
Komponenty i pojemniki aplikacji J2EE 37
Serwisy standardowe 38
Role w platformie J2EE 39
Pliki opisów rozmieszczenia 40
Wzorce J2EE a platforma J2EE 41
Podsumowanie 42

Część II Elementy projektów, niewłaściwe techniki oraz przebudowa systemu 43

Rozdział 3. Elementy projektów oraz niewłaściwe techniki warstwy prezentacyjnej 45

Elementy projektowania warstwy prezentacyjnej ... 45
Zarządzanie sesją ... 45
Kontrola praw dostępu klienta ... 48
Walidacja ... 52
Właściwości klas pomocniczych — integralność i konsekwencja ... 54
Niewłaściwe techniki zarządzania warstwą prezentacyjną ... 56
Kontrola kodu w widokach wielokrotnych ... 57
Udostępnianie struktur danych warstwy prezentacyjnej warstwie komercyjnej ... 57
Udostępnianie struktur danych warstwy prezentacyjnej obiektom rodzimym ... 58
Powielanie wysyłanych formularzy ... 58
Bezpośrednie udostępnianie ważnych zasobów ... 59
Założenie, że <jsp:setProperty> będzie resetować własności komponentu ... 59
Tworzenie „grubych" sterowników ... 60

Rozdział 4. Elementy projektów oraz niewłaściwe techniki warstwy komercyjnej 61

Elementy projektowania warstwy komercyjnej ... 61
Korzystanie z komponentów sesyjnych ... 61
Korzystanie z komponentów jednostkowych ... 65
Pamięć podręczna dla odległych referencji i uchwytów EnterpriseBeans ... 67
Niewłaściwe techniki zarządzania warstwą komercyjną ... 67
Mapowanie modelu obiektu bezpośrednio na model jednostki ... 67
Mapowanie modelu relacji bezpośrednio na model jednostki ... 68
Mapowanie każdego rodzaju wykorzystania na komponent sesyjny ... 69
Obsługa wszystkich atrybutów komponentu poprzez metody zmiany lub dostępu ... 69
Umieszczenie serwisu aktualizującego w aplikacjach klienckich ... 69
Używanie jednostki w charakterze obiektu tylko do odczytu ... 70
Używanie jednostek w charakterze obiektów wyspecjalizowanych ... 71
Przechowywanie pełnej struktury obiektów zależnych od jednostki ... 72
Niepotrzebne ujawnianie wyjątków związanych z EJB ... 72
Używanie metod przeszukujących komponentów jednostkowych do zwracania dużych bloków danych ... 73
Klient sortujący dane dla komponentów komercyjnych ... 73
Używanie EnterpriseBeans do obsługi transakcji czasochłonnych ... 74
Niezachowujące stanu komponenty odtwarzają stan konwersacji dla każdego wywołania ... 74

Rozdział 5. Sposoby przebudowy J2EE 77

Sposoby przebudowy warstwy prezentacyjnej ... 77
Korzystanie ze sterownika ... 77
Korzystanie z synchronizatora ... 79
Wyodrębnienie procedur rozproszonych ... 83
Ukrycie danych warstwy prezentacyjnej przed warstwą komercyjną ... 89
Usuwanie konwersji z widoku ... 92
Ukrywanie zasobów przed klientem ... 95
Sposoby przebudowy warstwy komercyjnej i integracyjnej ... 98
Zamknięcie komponentów jednostkowych w sesji ... 98
Korzystanie z delegatów komercyjnych ... 100
Łączenie komponentów sesyjnych ... 101
Ograniczanie komunikacji międzyjednostkowej ... 102
Przesunięcie procedur komercyjnych do sesji ... 104

Ogólne sposoby przebudowy ... 105
Separacja kodu dostępu do danych ... 105
Przebudowa architektury warstw ... 107
Korzystanie z zestawu połączeń ... 109

Część III Katalog wzorców J2EE 111

Rozdział 6. Omówienie wzorców J2EE 113

Czym jest wzorzec? ... 113
Identyfikacja wzorca ... 114
Podejście warstwowe ... 115
Wzorce J2EE ... 117
Wzorce warstwy prezentacyjnej ... 117
Wzorce warstwy komercyjnej ... 117
Wzorce warstwy integracyjnej ... 117
Wprowadzenie do katalogu ... 118
Terminologia ... 118
Korzystanie z UML ... 121
Szablon wzorców ... 122
Relacje pomiędzy wzorcami J2EE ... 123
Relacje do innych znanych wzorców ... 126
Mapa wzorców ... 126
Podsumowanie ... 130

Rozdział 7. Wzorce warstwy prezentacyjnej 131

Filtr przechwytujący ... 131
Kontekst ... 131
Problem ... 131
Siły ... 132
Rozwiązanie ... 132
Konsekwencje ... 145
Relacje z innymi wzorcami ... 146
Sterownik frontalny ... 146
Kontekst ... 146
Problem ... 146
Siły ... 147
Rozwiązanie ... 147
Konsekwencje ... 156
Relacje z innymi wzorcami ... 156
Pomocnik widoku ... 157
Kontekst ... 157
Problem ... 157
Siły ... 157
Rozwiązanie ... 158
Konsekwencje ... 169
Relacje z innymi wzorcami ... 169
Widok złożony ... 170
Kontekst ... 170
Problem ... 170
Siły ... 170
Rozwiązanie ... 170

Konsekwencje ... 177
Przykładowy kod ... 178
Relacje z innymi wzorcami ... 179
Struktura usługa-pracownik ... 180
Kontekst ... 180
Problem ... 180
Siły ... 180
Rozwiązanie ... 181
Konsekwencje ... 185
Przykładowy kod ... 186
Relacje z innymi wzorcami ... 191
Widok przekaźnika ... 191
Kontekst ... 191
Problem ... 192
Siły ... 192
Rozwiązanie ... 192
Konsekwencje ... 197
Przykładowy kod ... 198
Relacje z innymi wzorcami ... 202

Rozdział 8. Wzorce warstwy komercyjnej 203

Delegat komercyjny ... 203
Kontekst ... 203
Problem ... 203
Siły ... 204
Rozwiązanie ... 204
Konsekwencje ... 208
Przykładowy kod ... 209
Relacje z innymi wzorcami ... 213
Obiekt wartości ... 213
Kontekst ... 213
Problem ... 213
Siły ... 214
Rozwiązanie ... 214
Konsekwencje ... 223
Przykładowy kod ... 224
Relacje z innymi wzorcami ... 235
Fasada sesji ... 235
Kontekst ... 235
Problem ... 235
Siły ... 237
Rozwiązanie ... 237
Konsekwencje ... 241
Przykładowy kod ... 243
Relacje z innymi wzorcami ... 249
Jednostka złożona ... 250
Kontekst ... 250
Problem ... 250
Siły ... 252
Rozwiązanie ... 253
Konsekwencje ... 259

Przykładowy kod ... 260
Relacje z innymi wzorcami ... 269
Asembler obiektów wartości ... 269
Kontekst ... 269
Problem ... 269
Siły ... 272
Rozwiązanie ... 273
Konsekwencje ... 276
Przykładowy kod ... 277
Relacje z innymi wzorcami ... 281
Uchwyt listy wartości ... 281
Kontekst ... 281
Problem ... 282
Siły ... 282
Rozwiązanie ... 283
Konsekwencje ... 286
Przykładowy kod ... 287
Relacje z innymi wzorcami ... 292
Lokalizator usług ... 292
Kontekst ... 292
Problem ... 293
Siły ... 294
Rozwiązanie ... 295
Konsekwencje ... 302
Przykładowy kod ... 302
Relacje z innymi wzorcami ... 306

Rozdział 9. Wzorce warstwy integracyjnej 307

Obiekt dostępu do danych ... 307
Kontekst ... 307
Problem ... 307
Siły ... 308
Rozwiązanie ... 308
Konsekwencje ... 312
Przykładowy kod ... 314
Relacje z innymi wzorcami ... 320
Aktywator usług ... 320
Kontekst ... 320
Problem ... 320
Siły ... 321
Rozwiązanie ... 321
Konsekwencje ... 324
Przykładowy kod ... 325
Relacje z innymi wzorcami ... 330

Dodatki 331

Epilog 333

Bibliografia 345

Skorowidz 349

5

Sposoby przebudowy J2EE

Sposoby przebudowy warstwy prezentacyjnej

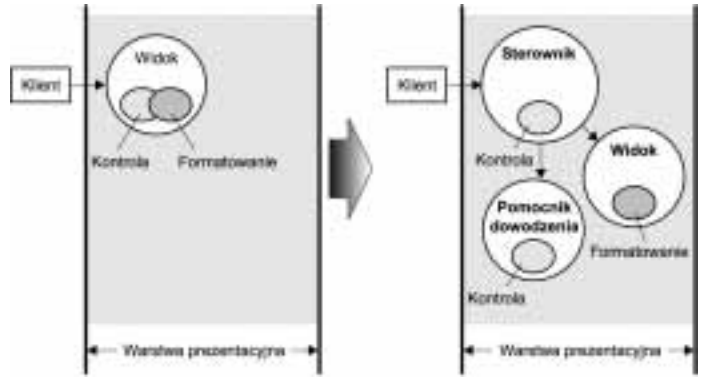
W tym podrozdziale zajmiemy się sposobami przebudowy warstwy prezentacyjnej.

Korzystanie ze sterownika

Procedury sterowania są rozproszone po całym kodzie aplikacji — zazwyczaj powielane są w wielu widokach stron serwerowych Javy (*Java Server Pages*, JSP).

Procedury sterowania należy wyodrębnić w przynajmniej jednej klasie sterowników. Klasy te odgrywają główną rolę w procesie obsługi żądań klientów.

Rysunek 5.1. *Korzystanie ze sterownika*



Motywacja

Powielany na różnych stronach JSP kod sterowania musi być przez każdą z nich utrzymywany. Wyodrębnienie takiego kodu w jednej klasie lub w większej ilości klas sterownika zwiększa przejrzystość oraz możliwości wielokrotnego wykorzystania aplikacji, a także ułatwia jej utrzymanie.

Mechanika

- Klasę sterownika należy utworzyć za pomocą wzorca klasy wydobycia (*Extract Class* [Fowler]) oraz wzorca sterownika frontalnego. Pozwala to na przeniesienie procedur sterowania z poszczególnych stron JSP do powstałego w ten sposób sterownika.
  - Przejdź do akapitu „Sterownik frontalny" (strona 146).
  - Należy pamiętać, że klasa sterownika stanowi punkt wyjścia w procesie sterowania obsługą żądań. Strukturę kodu aplikacji musi cechować modularność oraz możliwość wielokrotnego wykorzystania. Czasami zamiast tworzenia pojedynczego sterownika konieczne jest utworzenie obiektów pomocniczych i rozmieszczenie w nich kodu sterowania. Przejdź do akapitu „Tworzenie „grubych" sterowników" (strona 60).
- Przy korzystaniu ze wzorca polecenia (*Command* [GoF]) kod sterowania można również wyodrębnić w obiektach poleceń skoordynowanych ze sterownikiem.
  - Przejdź do akapitów „Sterownik frontalny" (strona 146).

Przykład

Załóżmy, że wiele stron JSP przykładowej aplikacji ma strukturę taką, jak w listingu 5.1.

Listing 5.1. *Korzystanie ze sterownika — struktura JSP*

```
<HTML>
<BODY>
  <control:grant_access/>
.
.
.
</BODY>
</HTML>
```

W wykropkowanej części listingu znajduje się właściwy kod każdej strony JSP, którego tu nie pokazano, a który jest różny w poszczególnych stronach JSP. Nie zmienia się jednak komponent pomocniczy, zaimplementowany na początku kodu każdej strony jako dodatkowo zdefiniowany znacznik. Komponent ten odpowiada za kontrolę dostępu do strony. Jest to kontrola typu „wszystko albo nic", co oznacza, że klient może uzyskać całkowity dostęp do strony albo nie uzyskać go w ogóle.

Aby zmienić strukturę poprzez wprowadzenie sterownika, zgodnie z opisem zamieszczonym w poprzedniej sekcji, należy usunąć ze stron JSP znacznik `<control:grant_access/>`, pokazany w listingu 5.1.

Funkcję tego znacznika, który do tej pory zawarty był we wszystkich stronach JSP, będzie pełnił scentralizowany sterownik, obsługujący kontrolę dostępu. Listing 5.2 ukazuje fragment kodu sterownika, zaimplementowanego jako serwlet.

Listing 5.2. *Korzystanie ze sterownika — struktura sterownika*

```
if (grantAccess())
{
    dispatchToNextView();
}
else
{
    dispatchToAccessDeniedView();
}
```

Oczywiście w pewnych sytuacjach komponenty pomocnicze dobrze spełniają rolę kodu sterowania. Jeżeli na przykład tylko ograniczona liczba stron JSP wymaga tego rodzaju kontroli dostępu, to rozsądnym rozwiązaniem jest użycie komponentu pomocniczego w postaci dodatkowo zdefiniowanego znacznika na tych wybranych stronach. Użycie dodatkowo definiowanych znaczników w poszczególnych stronach JSP jest również uzasadnione, gdy chcemy kontrolować dostęp do określonych podwidoków widoku złożonego (przejdź do akapitu „Widok złożony" w rozdziale 7.).

Nawet jeżeli korzystamy już ze sterownika, warto rozważyć zastosowanie omawianego tu rozwiązania scentralizowanego, ponieważ liczba stron wymagających takiego zabezpieczenia może się z czasem zwiększyć. Jeżeli więc sterownik został już utworzony, należy wyodrębnić z widoków kod sterowania i dodać go do istniejącego sterownika. Oznacza to, że konieczne będzie przeniesienie metod (za pomocą wzorca metody przesunięcia — *Move Method* [Fowler]), a nie wyodrębnienie nowej klasy.

Korzystanie z synchronizatora

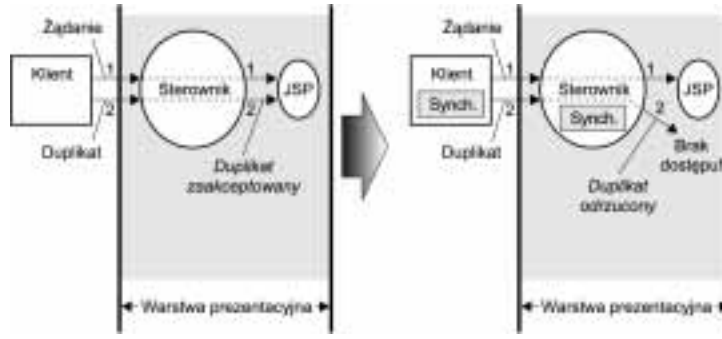
Żądania dostępu do zasobów, wysyłane przez klientów, wymagają monitorowania i kontroli. Pewnym problemem może być powtarzalność takich żądań. Klienci mogą też żądać dostępu do pewnych widoków w różnej kolejności, powracając do poprzednio odwiedzanych, zaznaczonych stron.

Skorzystanie z synchronizatora (ang. synchronizer token) umożliwia kontrolę przepływu żądań oraz dostępu klientów do określonych zasobów.

Motywacja

Istnieje szereg sytuacji, które wymagają kontrolowania przychodzących żądań. Są to najczęściej żądania wysyłane powtórnie przez klienta. Powtórzenie żądania ma miejsce na przykład wtedy, gdy użytkownik kliknie w przeglądarce przycisk *Wstecz* lub *Zatrzymaj* i ponownie prześle dany formularz.

Rysunek 5.2. *Korzystanie z synchronizatora*



Chociaż chodzi tu głównie o sterowanie kolejnością i przepływem żądań, należy również wspomnieć o zagadnieniach kontroli dostępu opartej na zezwoleniach. Użycie kontroli dostępu opartej na zezwoleniach omówiono w podrozdziale „Ukrywanie zasobów przed klientem".

Mechanika

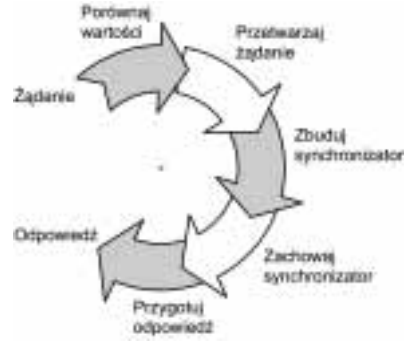
- Należy utworzyć jedną lub więcej klas pomocniczych, obsługujących generowanie i porównywanie jednorazowych, unikatowych synchronizatorów.
  - Rozwiązanie to można również dołączyć do istniejących komponentów sterowania.
  - Za pomocą takich obiektów pomocniczych komponent zarządzający żądaniami (zazwyczaj jest to sterownik, ale może to również być strona JSP) obsługuje tymczasowe przechowywanie aktualnego synchronizatora dla każdego żądania przesyłanego przez klienta.
  - Kopię synchronizatora przechowuje dla każdego użytkownika zarówno serwer, jak też przeglądarka klienta. Przeglądarka przechowuje synchronizator zazwyczaj w postaci ukrytego pola, a serwer w sesji użytkownika.

Generowanie, zachowywanie i porównywanie synchronizatorów

Zanim odebrane żądanie zostanie przetworzone, sprawdzana jest zgodność synchronizatora. Po przetworzeniu żądania generowany i zapisywany jest nowy synchronizator. Dopiero wtedy odpowiedź zostaje przygotowana i odesłana do klienta.

Dalsze informacje na ten temat zawiera podrozdział „Korzystanie z synchronizatora" oraz rysunek 5.3.

Rysunek 5.3. *Cykl „życia" synchronizatora*



- Dołącz kod sprawdzający, czy synchronizator, przesyłany wraz z żądaniem klienta, odpowiada synchronizatorowi sesji użytkownika.
  - Wartość synchronizatora przesyłanego przez klienta w bieżącym żądaniu powinna być taka sama, jak wartość synchronizatora, który serwer przesłał do klienta w ostatniej odpowiedzi. Zgodność tych dwóch wartości daje gwarancję, że żądanie nie zostało powtórnie przesłane, podczas gdy brak zgodności może oznaczać, że miało miejsce powtórzenie żądania.
  - Jak już wspomniano, możliwe są również inne przyczyny braku zgodności tych wartości — użytkownik mógł na przykład udać się bezpośrednio na odwiedzoną wcześniej, zaznaczoną stronę. Najczęstszym jednak powodem pozostaje powtórne przesłanie żądania. Klasa sterownika zazwyczaj obsługuje generowanie oraz porównywanie synchronizatorów. Jeżeli aplikacja nie posiada sterownika, to warto rozważyć jego utworzenie.
  - Zobacz „Korzystanie ze sterownika" na stronie 77.
  - W przypadku braku sterownika centralnie obsługującego generowanie i porównywanie synchronizatorów konieczne jest zapewnienie takiej obsługi z poziomu każdej strony JSP.
  - Zazwyczaj strony JSP wykorzystują w tym celu komponent pomocniczy, zaimplementowany w postaci obiektu JavaBean lub dodatkowo zdefiniowanego znacznika (zobacz „Pomocnik widoku" — strona 157), który zapewnia odpowiednie zarządzanie synchronizatorami.

Fragmenty kodu źródłowego, przedstawione w sekcji „Korzystanie z synchronizatora", zostały użyte zgodnie z warunkami licencji *Apache Software License, Version 1.1*.

Przykład

W szkielecie warstwy prezentacji *Struts* zastosowano kilka wzorców i sposobów przebudowy J2EE, między innymi omawiany tu rodzaj kontroli przepływu żądań. W poniższym przykładzie wykorzystamy fragmenty kodu źródłowego tego szkieletu.

W szkielecie *Struts* nie jest tworzona oddzielna klasa, obsługująca generowanie i porównywanie synchronizatorów — funkcjonalność ta jest dodana do utworzonej wcześniej klasy, która stanowi część mechanizmu sterowania szkieletu. Jest to klasa o nazwie `Action` („działanie"), będąca wspólną klasą nadrzędną wszystkich działań. Działania są obiektami typu `Command` („polecenie"), stanowiącymi rozszerzenie funkcjonalności sterownika. Jest to aplikacja zgodna ze wzorcem sterownika frontalnego oraz strategią sterownika i polecenia.

Listing 5.3 pokazuje, w jaki sposób metoda `saveToken()`, zdefiniowana w klasie `Action`, generuje i zachowuje wartości synchronizatorów.

Listing 5.3. *Generowanie i zachowywanie synchronizatorów*

```
/**
* Zapisanie synchronizatora nowej transakcji
* w bieżącej sesji użytkownika. W razie
* potrzeby tworzona jest nowa sesja.
```

```
 *
 * @param request Przetwarzane żądanie serwletu
 * /
protected void saveToken(HttpServletRequest request) {

HttpSession session = request.getSession();
String token = generateToken(request);
if (token != null)
  session.setAttribute(TRANSACTION_TOKEN_KEY, token);
}
```

Copyright © 1999 The Apache Software Foundation. **Wszelkie prawa zastrzeżone.**

Metoda ta generuje unikatowy synchronizator w oparciu o ID sesji i bieżący czas, a następnie zapisuje uzyskaną w ten sposób wartość w sesji użytkownika.

Jeszcze przed wygenerowaniem kodu HTML dla klienta przesyłającego żądanie, którego powtórzenia chcemy uniknąć (zazwyczaj jest to formularz odsyłany do serwera), w opisany wcześniej sposób ustawiana jest wartość synchronizatora. Wartość ta zostaje ustawiona poprzez wywołanie metody `saveToken()`:

```
saveToken(request);
```

Dodatkowo strona JSP, generująca wyświetlany kod HTML, zawiera również kod, który — przy wykorzystaniu klasy pomocniczej — generuje ukryte pole, zawierające wartość synchronizatora. Dzięki temu strona odsyłana do klienta (zazwyczaj z formularzem odsyłanym do serwera) zawiera takie ukryte pole:

```
<input type="hidden"
name="org.apache.struts.taglib.html.TOKEN"
value="8d2c392e93a39d299ec45a22">
```

Atrybut `value` tego ukrytego pola zawiera wartość synchronizatora, wygenerowanego przez metodę `saveToken()`.

Gdy klient przesyła stronę zawierającą to ukryte pole, sterownik — poprzez obiekt `Command` (będący podklasą klasy `Action`) — porównuje wartość synchronizatora w sesji użytkownika z wartością parametru `request`, przesyłanego wraz z ukrytym polem strony. Do porównania tych wartości obiekt `Command` wykorzystuje metodę pokazaną w listingu 5.4, a zaczerpniętą z jego klasy nadrzędnej (również w tym przypadku jest to klasa `Action`).

Listing 5.4. *Sprawdź poprawność synchronizatora*

```
/**
* Zwraca wartość <code>true</code>, jeżeli w bieżącej sesji użytkownika przechowywany
* jest synchronizator transakcji, którego wartość jest zgodna z wartością przekazaną
* jako parametr request.
*
* Zwraca wartość <code>false</code> w przypadku zaistnienia któregoś z poniższych
* warunków:
* <ul>
* <li>żądanie nie jest przypisane do żadnej sesji,</li>
* <li>w sesji nie zapisano żadnego synchronizatora,</li>
* <li>nie istnieje parametr request zawierający wartość synchronizatora,</li>
* <li>wartość załączonego synchronizatora transakcji różni się od wartości
* synchronizatora transakcji, przechowywanej w sesji użytkownika.</li>
```

```
 * </ul>
 *
 * @param request Przetwarzane żądanie serwletu
 */

   protected boolean isTokenValid(HttpServletRequest request) {

   // Pobranie synchronizatora transakcji, zapisanego w bieżącej sesji:
   HttpSession session = request.getSession(false);
   if (session == null)
       return (false);
   String saved = (String)
       session.getAttribute(TRANSACTION_TOKEN_KEY);
   if (saved == null)
       return (false);
   // Pobranie synchronizatora transakcji załączonego w bieżącym żądaniu
   String token = (String)
       request.getParameter(Contants.TOKEN_KEY);
   if (token == null)
       return (false);

   // Sprawdzenie zgodności obu wartości:
   return (saved.equals(token));
 }
```

Copyright © 1999 The Apache Software Foundation. **Wszelkie prawa zastrzeżone.**

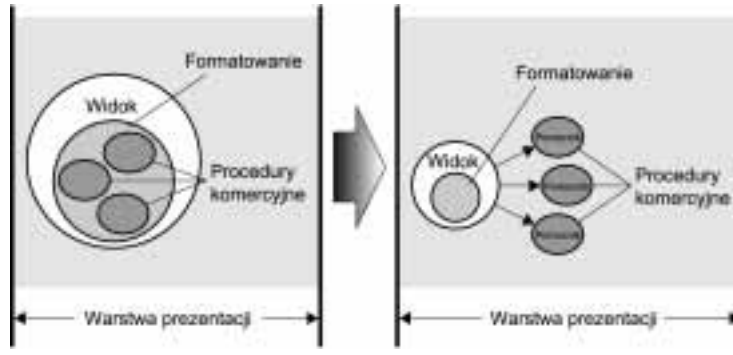
Jeżeli porównywane wartości są identyczne, to mamy pewność, że dane żądanie nie zostało przesłane powtórnie. Jeżeli natomiast wartości synchronizatorów są różne, pozwala to podjąć odpowiednie czynności obsługi powtórzonych żądań.

Wyodrębnienie procedur rozproszonych

Procedury komercyjne oraz kod prezentacji (formatowania) są ze sobą przemieszane w poszczególnych widokach stron JSP.

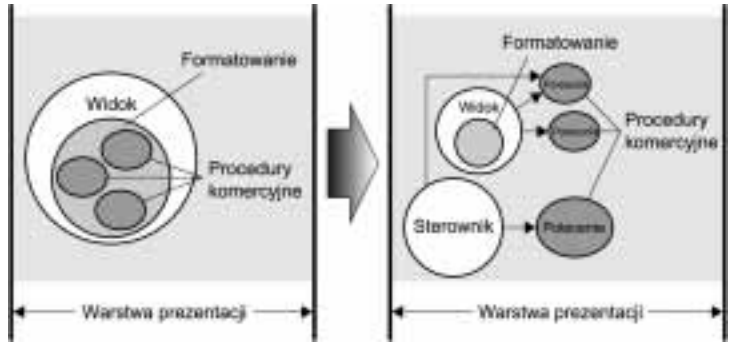
Procedury komercyjne należy wyodrębnić w jednej lub większej ilości klas pomocniczych, z których będzie korzystać strona JSP lub sterownik.

Rysunek 5.4 ukazuje wyodrębnienie procedur komercyjnych z widoku do klas pomocniczych.



Rysunek 5.4. *Wyodrębnienie procedur rozproszonych: przebudowa wstecz*

Rysunek 5.5 ukazuje wyodrębnienie logiki komercyjnej z widoku do sterownika, obiektu `Command` oraz klas pomocniczych.



Rysunek 5.5. *Wyodrębnienie procedur rozproszonych: przebudowa naprzód*

Motywacja

Celem jest zwiększenie przejrzystości i spójności tworzonych aplikacji przy jednoczesnym zmniejszeniu ilości powielanych fragmentów kodu, co zwiększy modularność i możliwość wielokrotnego wykorzystania kodu aplikacji. Przejrzysta struktura elementów aplikacji pozwala również na wyraźny podział pracy pomiędzy twórcami kodu formatowania aplikacji WWW a programistami, zajmującymi się procedurami komercyjnymi.

Mechanika

- W oparciu o wzorce klasy wydobycia (*Extract Class* [Fowler]) oraz pomocnika widoku należy utworzyć nowe klasy pomocnicze, przesuwając do nich procedury komercyjne ze stron JSP.
- Klasy te należy udostępnić z poziomu stron JSP.
  - Przejdź do akapitu „Pomocnik widoku" (strona 157).
  - Punktem wyjściowym w procesie obsługi żądania może być widok, jak to pokazano na diagramie przebudowy wstecz na rysunku 5.4. Przejdź do akapitu. „Widok przekaźnika" (strona 191).
- Warto rozważyć skorzystanie ze sterownika, jeżeli aplikacja go nie posiada.
  - Przejdź do akapitu „Korzystanie ze sterownika" na stronie 77.
  - Jak to przedstawiono na diagramie przebudowy naprzód (rysunek 5.5), sterownik może korzystać z obiektu pomocniczego `Command`.
  - Punktem wyjściowym w procesie obsługi żądania klientów może być sterownik, jak to pokazuje diagram przebudowy naprzód. Przejdź do akapitu „Struktura usługa-pracownik" (strona 180).

Przykład

Rozważmy kod, przedstawiony w widoku w listingu 5.5. Jest to strona JSP, zawierająca fragmenty skryptletu oraz kodu procedur komercyjnych.

Listing 5.5. *Strona JSP zawierająca kod skryptletu*

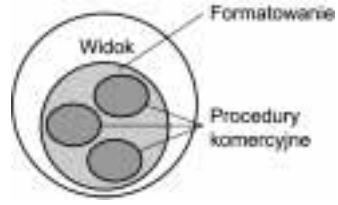
```
<html>
<head><title>Lista pracowników</title></head>
<body>
<%-- Wyświetlenie nazwisk wszystkich pracowników danego działu, których zarobki nie
przekraczają podanego limitu --%>

<%

    // Określenie nazwy działu, którego pracownicy mają
    // zostać wyświetleni:
    String deptidStr = request.getParameter(
        Constants.REQ_DEPTID);

    // Określenie limitu zarobków:
    String salaryStr = request.getParameter(
        Constants.REQ_SALARY);

    // Weryfikacja parametrów.

    // Jeżeli wysokość zarobków lub dział nie zostały określone,
    // wyświetlana jest strona z informacją o błędzie:
    if ( (deptidStr == null) || (salaryStr == null ) )
    {
       request.setAttribute(Constants.ATTR_MESSAGE,
        "Brak wymaganych parametrów wyszukiwania" +
        "(Dział i Zarobki)");
       request.getRequestDispatcher("/error.jsp").
         forward(request, response);
    }

    // Konwersja parametrów (pobranych jako łańcuchy znaków)
    // na typy numeryczne:
    int deptid = 0;
    float salary = 0;
    try
    {
        deptid = Integer.parseInt(deptidStr);
        salary = Float.parseFloat(salaryStr);
    }
    catch(NumberFormatException e)
    {
       request.setAttribute(Constants.ATTR_MESSAGE,
          "Niewłaściwe wartości wyszukiwania" +
          "(department id and salary )");
       request.getRequestDispatcher("/error.jsp").
          forward(request, response);
}
```

```
    // Sprawdzenie poprawności zakresu parametrów:
    if ( salary < 0  )
    {
      request.setAttribute(Constants.ATTR_MESSAGE,
        " Niewłaściwe wartości wyszukiwania" +
        "(department id and salary )");
      request.getRequestDispatcher("/error.jsp").
        forward(request, response);
    }

%>

<h3><center> Lista pracowników w dziale #
  <%=deptid%> zarabiających do <%= salary %>. </h3>

<%
    Iterator employees = new EmployeeDelegate().
                           getEmployees(deptid);
%>

<table border="1" >
    <tr>
        <th> Imię </th>
        <th> Nazwisko </th>
        <th> Stanowisko </th>
        <th> Numer identyfikacyjny </th>
        <th> Odliczenia od podatku </th>
        <th> Uwagi na temat wydajności </th>
        <th> Roczne zarobki </th>
    </tr>
<%
    while ( employees.hasNext() )
    {
        EmployeeVO employee = (EmployeeVO)
                               employees.next();

        // lista pracowników zostanie wyświetlona,
        // jeżeli określono poprawne kryteria wyszukiwania:
        if ( employee.getYearlySalary() <= salary )
        {
%>
        <tr>
          <td> <%=employee.getFirstName()%></td>
          <td> <%=employee.getLastName()%></td>
          <td> <%=employee.getDesignation()%></td>
          <td> <%=employee.getId()%></td>
          <td> <%=employee.getNoOfDeductibles()%></td>
          <td> <%=employee.getPerformanceRemarks()%>
                </td>
          <td> <%=employee.getYearlySalary()%></td>
        </tr>
<%
        }
    }
%>
</table>
```

```
<%@ include file="/jsp/trace.jsp" %>
<P> <B>Procedury komercyjne oraz kod formatowania (warstwa prezentacyjna) są
przemieszane w tym widoku JSP. </B>

</body>
</html>
```

Przedstawiony powyżej kod JSP generuje tabelę HTML, zawierającą listę pracowników o określonych zarobkach. Zawarto tu zarówno procedury komercyjne, jak też kod formatowania, co zilustrowano na rysunku 5.6.

Rysunek 5.6. *Widok, w którym przemieszano logikę komercyjną z kodem formatowania*



W listingu 5.6 pokazano, w jaki sposób w oparciu o wzorzec pomocnika widoku można zmienić strukturę tak, aby wyodrębnić kod skryptletu z widoku JSP.

Listing 5.6. *Strona JSP po wyodrębnieniu kodu skryptletu*

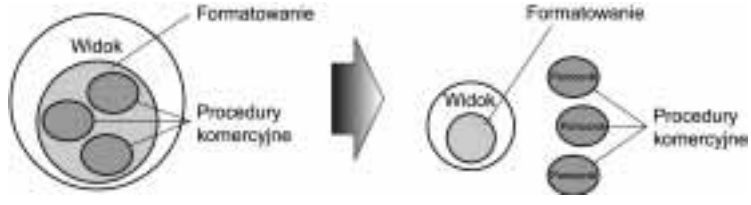
```
<%@ taglib uri="/WEB-INF/corepatternstaglibrary.tld"
    prefix="corepatterns" %>
<html>
<head><title>Lista pracowników</title></head>
<body>

<corepatterns:employeeAdapter />

<h3><center>Lista pracowników działu
  <corepatterns:department attribute="id"/>
  - przykład użycia obiektu pomocniczego w dodatkowo zdefiniowanym znaczniku. </h3>

<table border="1" >
    <tr>
        <th> Imię </th>
        <th> Nazwisko </th>
        <th> Stanowisko </th>
        <th> Numer identyfikacyjny pracownika </th>
        <th> Odliczenia od podatku </th>
        <th> Uwagi na temat wydajności </th>
        <th> Roczne zarobki </th>
    </tr>
    <corepatterns:employeelist id="employeelist_key">
    <tr>
      <td><corepatterns:employee
            attribute="FirstName"/></td>
      <td><corepatterns:employee
            attribute="LastName"/></td>
      <td><corepatterns:employee
            attribute="Designation"/> </td>
        <td><corepatterns:employee
            attribute="Id"/></td>
```

```
            <td><corepatterns:employee
                attribute="NoOfDeductibles"/></td>
            <td><corepatterns:employee
                attribute="PerformanceRemarks"/></td>
            <td><corepatterns:employee
                attribute="YearlySalary"/></td>
            <td>
          </tr>
        </corepatterns:employeelist>
    </table>

    </body>
    </html>
```

Poza tym zdefiniowano dwa obiekty pomocnicze, zaimplementowane w dodatkowo zdefiniowanych znacznikach, co pozwoliło na wyodrębnienie procedur komercyjnych oraz formatowania poprzez oparcie modelu danych na kolumnach i wierszach tabeli HTML.

Dwa obiekty pomocnicze, o których mowa, to znaczniki `<corepatterns:employeelist>` oraz `<corepatterns:employee>`.

Rysunek 5.7 ukazuje transformację projektu: od struktury pokazanej po lewej stronie do struktury pokazanej po prawej stronie.



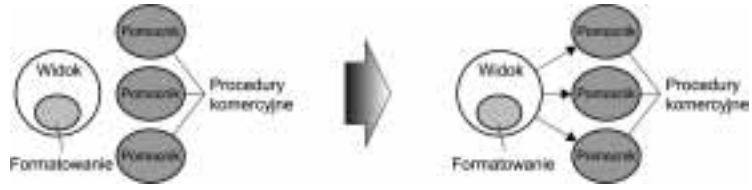
Rysunek 5.7. *Wyodrębnienie logiki komercyjnej w klasach pomocniczych*

Procedury komercyjne zostały wyodrębnione w klasach pomocniczych, dzięki czemu nie znajdują się już bezpośrednio na stronie JSP. Klasy te pełnią szereg funkcji, takich jak pobieranie określonej zawartości, kontrolowanie dostępu, oraz wyświetlanie stanu modelu dla użytkownika. W przypadku ostatniej z wymienionych funkcji klasa pomocnicza hermetyzuje część logiki prezentacji, np. formatowanie zbioru wyniku zapytania w postaci tabeli HTML (zobacz także „Usuwanie konwersji z widoku"). Takie rozwiązania pozwalają spełnić postawiony wcześniej cel wyodrębnienia jak największej części logiki programowania z widoku. Dzięki temu z poziomu JSP możliwe jest, poprzez użycie klas pomocniczych, uzyskanie tabeli z określonymi danymi w miejsce kodu skryptletu generującego taką tabelę, zamieszczanego w kodzie JSP.

Komponenty pomocnicze można implementować w postaci obiektów JavaBean lub też jako dodatkowo definiowane znaczniki (przejdź do akapitu „Pomocnik widoku" w rozdziale 7.). Komponenty pomocnicze JavaBean sprawdzają się najlepiej przy wyodrębnianiu logiki pobierania zadanej zawartości oraz przechowywania wyników zapytań. Natomiast dodatkowo zdefiniowanych znaczników należy używać do omówionej powyżej konwersji modelu przy wyświetlaniu wyniku, czyli na przykład przy tworzeniu tabeli z zwróconego zbioru wyniku. Obie te opcje mają ze sobą wiele wspólnego, więc wybór techniki implementacji komponentów pomocniczych będzie zależał od doświadczenia programisty i względów praktycznych.

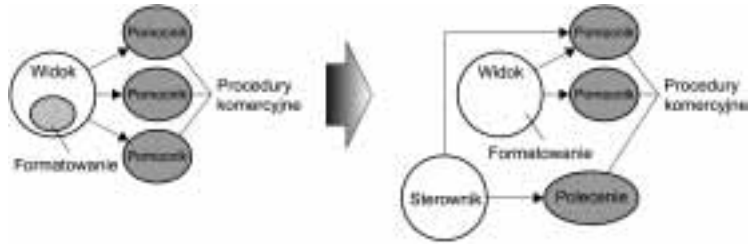
Rysunek 5.8 ilustruje drugi punkt podrozdziału „Mechanika", czyli przekazywanie zadań do klas pomocniczych z poziomu JSP.

Rysunek 5.8. *Przekazywanie zadań do klas pomocniczych*



Z poziomu widoku JSP wywoływane są klasy pomocnicze, które odpowiadają za przetwarzanie i generowanie widoku. Zazwyczaj przed stroną JSP używany jest sterownik jako punkt wyjściowy w procesie obsługi żądań klientów (przejdź do akapitów „Sterownik frontalny" w rozdziale 7. oraz „Korzystanie ze sterownika"). Sterownik udostępnia widok, ale zanim do tego dojdzie, może również przekazać zadania do komponentów pomocniczych (przejdź do akapitu „Struktura usługa-pracownik" w rozdziale 7.). Zmiany w strukturze po zastosowaniu sterownika przedstawiono na rysunku 5.9.

Rysunek 5.9. *Korzystanie ze sterownika*

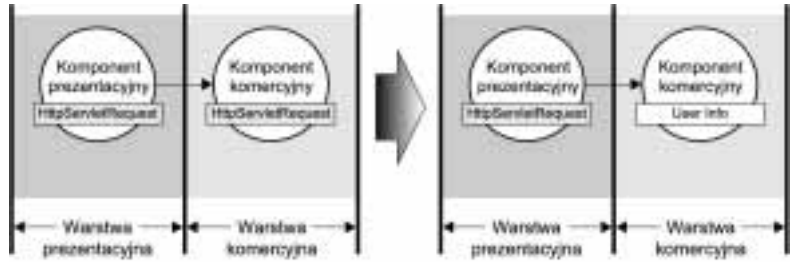


Ukrycie danych warstwy prezentacyjnej przed warstwą komercyjną

Obsługa żądań oraz (lub) związane z obsługą protokołu struktury danych są udostępniane z warstwy prezentacyjnej dla warstwy komercyjnej.

Z warstwy komercyjnej należy usunąć wszystkie referencje do obsługi żądań oraz do struktur danych warstwy prezentacyjnej, związanych z obsługą protokołu. Wartości należy przekazywać pomiędzy warstwami za pomocą ogólnych struktur danych.

Rysunek 5.10. *Wyodrębnienie kodu warstwy prezentacji z warstwy komercyjnej*



Motywacja

Fragmenty kodu implementacji właściwe dla jednej warstwy nie powinny być umieszczane w innej warstwie. Interfejs API usługi, udostępniany przez warstwę komercyjną warstwie prezentacyjnej, będzie najprawdopodobniej wykorzystywany przez różnych klientów. Jeżeli

API usługi przyjmuje parametry szczegółowego typu, np. *HttpServletRequest*, to każdy klient będzie musiał hermetyzować dane w strukturach danych typu *HttpServletRequest*. To z kolei radykalnie zmniejsza możliwość wielokrotnego wykorzystania danej usługi.

Mechanika

- Wszystkie referencje do struktur danych warstwy prezentacji, zamieszczone w warstwie komercyjnej, należy zastąpić referencjami do bardziej ogólnych struktur i typów danych.
  - Chodzi tu zazwyczaj o zastąpienie metod warstwy komercyjnej przyjmujących parametry szczegółowego typu, np. *HttpServletRequest*, parametrami bardziej ogólnych typów, takich jak *String*, *int* czy też *UserInfo*.
- W warstwie prezentacji należy zmodyfikować kod kliencki, który zawiera wywołania takich metod.
  - Do metod warstwy komercyjnej można przesyłać rozdrobnione struktury danych warstwy prezentacji. Jeżeli na przykład obiekt żądania *HttpServletRequest* ma parametry `x`, `y` oraz `z`, to metoda w warstwie komercyjnej zamiast parametrów typu *HttpServletRequest* może przyjmować te trzy argumenty oddzielnie — jako parametry typu `String`. Wadą takiego przesyłania rozdrobnionych argumentów jest to, że technika ta zwiększa powiązania pomiędzy kodem warstwy prezentacji a API serwisu komercyjnego. Oznacza to, że przy zmianie stanu wymaganego przez usługę musi również zostać zmieniony API serwisu.
  - Nieco bardziej elastyczne rozwiązanie polega na skopiowaniu odpowiedniego stanu ze struktury danych warstwy prezentacji do bardziej ogólnych struktur danych, na przykład do obiektów przyjmujących wartość, które są potem przekazywane do warstwy komercyjnej. W takim przypadku API serwisu nadal przyjmuje obiekt tego typu, nawet jeżeli zmieniła się jego implementacja.
- Inne rozwiązanie polega na implementacji strategii nałożenia typu interfejsu, jeżeli na przykład zastosowano szkielet warstwy prezentacji taki, jak często stosowany projekt *Struts* [Struts].
  - Podczas obsługi żądania szkielety warstwy tworzą zazwyczaj szereg struktur danych. Polega to na przykład na tym, że szkielet kopiuje odpowiedni stan ze struktury danych typu *HttpServletRequest* do bardziej ogólnych struktur danych, dopasowując parametry żądania do typu danych właściwego dla określonego szkieletu. Chociaż wspomniany typ danych może spełniać taką samą funkcję, jaką spełnia obiekt posiadający wartość (typu ogólnego), to jednak jest to typ właściwy dla danego szkieletu. Oznacza to, że przesyłanie takiej struktury danych do warstwy komercyjnej wymaga wytworzenia ścisłej zależności pomiędzy szkieletem obsługującym żądania a serwisami komercyjnymi. Pewnym rozwiązaniem może być w tej sytuacji właśnie skopiowanie struktury danych właściwej dla szkieletu do bardziej ogólnej struktury, przed przekazaniem jej do warstwy komercyjnej. Bardziej wydajne jest jednak utworzenie ogólnego typu interfejsu, który będzie zawierał metody odpowiadające metodom typu właściwego dla szkieletu. Nałożenie takiego typu interfejsu na obiekt właściwy

dla szkieletu umożliwi korzystanie z tego obiektu bez konieczności wytwarzania ścisłego powiązania z określonym szkieletem.

- Jeżeli na przykład szkielet tworzy instancję podklasy obiektu `a.framework.MyStateBean`, to będzie ona typu `StateBean`:

```
// Uwaga: tworzenie instancji odbywa się zazwyczaj poprzez przebudowę.
// Uwaga: dla uproszczenia pominięto parametry.
a.framework.StateBean bean = new my.stuff.MyStateBean(...);
```

- Gdyby warstwa komercyjna przyjmowała przedstawiony komponent JavaBean jako parametr, byłby on typu `StateBean`:

```
public void aRemoteBizTierMethod(a.framework.StateBean bean)
```

- Zamiast przekazywać komponent JavaBean typu `StateBean` do warstwy komercyjnej, należy utworzyć nowy interfejs, np. o nazwie *my.stuff.MyStateVO*, zaimplementowany przez obiekt `my.stuff.MyStateBean`:

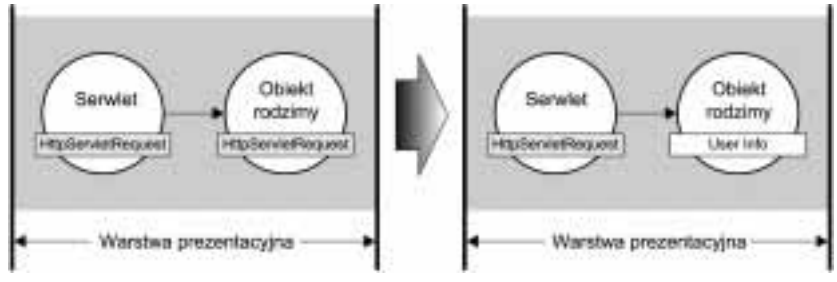
```
public class MyStateBean extends a.framework.StateBean implements MyStateVO
```

- W ten sposób w warstwie komercyjnej można użyć następującej sygnatury metody:

```
public void aRemoteBizTierMethod(my.stuff.MyStateVO bean)
```

- Nie ma potrzeby kopiowania parametrów do obiektu ogólnego typu, posiadającego wartość, a typ właściwy dla szkieletu nie jest już udostępniany w różnych warstwach jednocześnie.

- Należy wreszcie zaznaczyć, że zależności pomiędzy logicznie niezwiązanymi ze sobą częściami można zredukować poprzez zastosowanie opisywanego tu sposobu przebudowy dla obiektów warstwy prezentacyjnej.
  - Ilustracja podobnego przykładu znajduje się na rysunku 5.11.



Rysunek 5.11. *Oddzielenie kodu warstwy prezentacji od obiektów domeny*

- Ponieważ należy zachowywać możliwość wielokrotnego użycia obiektów rodzimych, takich jak `Customer`, również w tym przypadku obowiązuje opisywana powyżej motywacja i mechanika.
- Oznacza to, że referencje do struktur danych związanych z protokołem są skoncentrowane wokół komponentów obsługujących żądania, takich jak sterownik. Listingi 5.7 oraz 5.8, zamieszczone poniżej, pokazują, w jaki sposób można oddzielić obiekt `HttpServletRequest` od obiektu rodzimego.

Przykład

Przedstawiona w listingu 5.7 klasa `Customer` przyjmuje jako parametr instancję obiektu domeny `HttpServletRequest`, którego typ charakteryzuje się wysokim stopniem specjalizacji. Oznacza to, że klient spoza Internetu, który chce skorzystać z klasy `Customer`, musiałby najpierw utworzyć w jakiś sposób obiekt `HttpServletRequest`, a takich rozwiązań należy unikać.

Listing 5.7. *Ścisłe powiązanie obiektu domeny z obiektem HttpServletRequest*

```
/** Poniższy fragment kodu ukazuje obiekt rodzimy,
który został zbyt ściśle powiązany z obiektem HttpServletRequest. **/
public class Customer
{
  public Customer ( HttpServletRequest request )
  {
       firstName = request.getParameter("firstname");
       lastName = request.getParameter("lastname ");
  }
}
```

Zamiast udostępniać obiekt typu *HttpServletRequest* ogólnemu obiektowi `Customer`, należy je od siebie oddzielić, jak to pokazano w listingu 5.8.

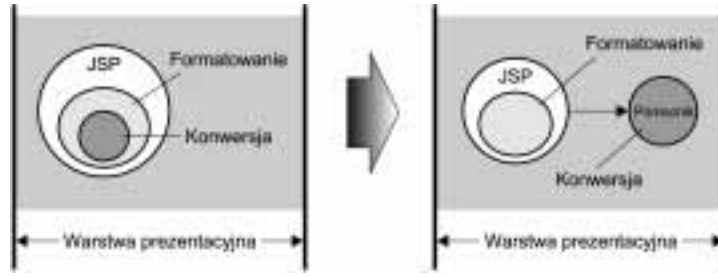
Listing 5.8. *Oddzielenie obiektu domeny od obiektu HttpServletRequest*

```
// Przykład oddzielenia obiektu domeny od obiektu HttpServletRequest.
public class Customer
{
  public Customer ( String first, String last )
  {
    firstName = first;
    lastName = last;
  }
}
```

Usuwanie konwersji z widoku

Fragmenty modelu podlegają w komponencie widoku konwersji, która umożliwia ich wyświetlenie.

Cały kod konwersji należy usunąć z widoku,
umieszczając go w jednej lub większej ilości klas pomocniczych.



Rysunek 5.12. *Usunięcie konwersji z widoku*

Motywacja

Bezpośrednie osadzenie w widoku JSP kodu dokonującego konwersji modelu w celu jego wyświetlenia ogranicza modularność i możliwość wielokrotnego wykorzystywania aplikacji. Ponieważ tego rodzaju konwersje mogą być konieczne na wielu stronach JSP, kod konwersji w nich umieszczany musiałby się powtarzać, a utrzymanie takich stron polegałoby na pracochłonnym wycinaniu i wklejaniu fragmentów kodu.

Mechanika

- Za pomocą wzorca klasy wydobycia (*Extract Class* [Fowler]) oraz pomocnika widoku należy utworzyć klasy pomocnicze i przenieść do nich procedury konwersji z poszczególnych stron JSP.
  - Przykładem może być konwersja zbioru wyników z bazy danych do postaci tabeli HTML, w oparciu o odpowiedni kod aplikacji.
- Klasy te należy udostępnić z poziomu stron JSP, dzięki czemu — w razie potrzeby — możliwe będzie dokonanie konwersji i adaptacji.
  - Konwersja jest przeprowadzana przez klasę pomocniczą, wywoływaną z poziomu strony JSP.

Przykład

W przedstawionym tu przykładzie zajmiemy się procedurą, która przeprowadza konwersję zbioru elementów (zbioru wyników z bazy danych) do postaci tabeli HTML. Jest to w pewnym sensie procedura formatowania, ale chodzi tu również o kod konwersji, generujący tabelę z modelu pośredniego. Kod takiej dynamicznej konwersji może być wielokrotnie wykorzystywany, jeżeli zostanie zaimplementowany w dodatkowo zdefiniowanych znacznikach, zamiast bezpośrednio w kodzie strony JSP.

W listingu 5.9 przedstawiono stronę JSP, w której kodzie bezpośrednio zamieszczono procedurę konwersji.

Listing 5.9. *Procedura konwersji umieszczona bezpośrednio w widoku*

```
<html>
<head><title>Lista pracowników</title></head>
<body>

<h3><head><center> Lista pracowników</h3>

<%
    String firstName =
      (String)request.getParameter("firstName");
    String lastName  =
      (String)request.getParameter("lastName");
    if ( firstName == null )
      // w razie potrzeby pobrane zostaną wszystkie elementy
      firstName = "";
```

```
    if ( lastName == null )
      lastName = "";

    EmployeeDelegate empDelegate = new
            EmployeeDelegate();
    Iterator employees =
        empDelegate.getEmployees(
          EmployeeDelegate.ALL_DEPARTMENTS);
%>

<table border="1" >
    <tr>
        <th> Imię </th>
        <th> Nazwisko </th>
        <th> Stanowisko </th>
    </tr>
<%
    while ( employees.hasNext() )
    {
        EmployeeVO employee = (EmployeeVO)
                              employees.next();

        if ( employee.getFirstName().
              startsWith(firstName) &&
             employee.getLastName().
              startsWith(lastName) ) {
%>
 <tr>
  <td><%=employee.getFirstName().toUpperCase() %></td>
  <td> <%=employee.getLastName().toUpperCase() %></td>
  <td> <%=employee.getDesignation()%></td>
 </tr>
<%
        }
    }
%>
</table>
```

Najpierw taki kod należy wyodrębnić w klasach pomocniczych. Klasy takie należy w tym przypadku zaimplementować w dodatkowo zdefiniowanych znacznikach, co umożliwi usunięcie możliwie największej ilości kodu skryptletu ze strony JSP. Następnie w kodzie strony JSP należy umieścić odwołania do klas pomocniczych, przeprowadzających konwersję. W listingu 5.10 przedstawiono stronę JSP po wprowadzeniu takich zmian.

Listing 5.10. *Wyodrębnienie procedury w klasach pomocniczych*

```
<html>
<head><title>Lista pracowników po przebudowie </title>
</head>
<body>

<h3> <center>Lista pracowników</h3>

<corepatterns:employeeAdapter />

<table border="1" >
```

```
    <tr>
        <th> Imię </th>
        <th> Nazwisko </th>
        <th> Stanowisko </th>
    </tr>

<corepatterns:employeelist id="employeelist"
    match="FirstName, LastName">
<tr>

    <td><corepatterns:employee attribute= "FirstName"
  case="Upper" /> </td>
    <td><corepatterns:employee attribute= "LastName"
  case="Upper" /></td>
    <td><corepatterns:employee attribute=
  "Designation" /> </td>
    <td>
 </tr>
</corepatterns:employeelist>
</table>
```

Warto też rozważyć inny rodzaj konwersji. W niektórych przypadkach fragmenty modelu podlegają konwersji do HTML poprzez transformacje XSL. W tym celu również można wykorzystać klasy pomocnicze, wywoływane za pomocą dodatkowo zdefiniowanych znaczników. Również takie rozwiązanie pozwala wyodrębnić procedurę ze strony JSP, zwiększając w ten sposób modularność i możliwość wielokrotnego wykorzystania komponentów. Poniżej ukazano przykład strony JSP, w której konwersja wykonywana jest poprzez wywołanie za pomocą dodatkowych znaczników klas pomocniczych, a nie bezpośrednio w kodzie strony:

```
%@taglib uri="http://jakarta.apache.org/taglibs/xsl-1.0" prefix="xsl" %
<xsl:apply nameXml="model" propertyXML="xml" xsl="/stylesheet/transform.xsl"/>
```

Znacznik XSL `apply` [JakartaTaglibs] odpowiada za wygenerowanie całego wyniku z tej strony. Za jego pomocą można utworzyć w podobny sposób poszczególne składniki strony. Wywołanie za pomocą znacznika jest możliwe, ponieważ w zakresie strony („modelu") istnieje komponent JavaBean z właściwością o nazwie „xml". Innymi słowy, w zakresie strony znajduje się instancja komponentu, który zawiera metodę o następującej sygnaturze:

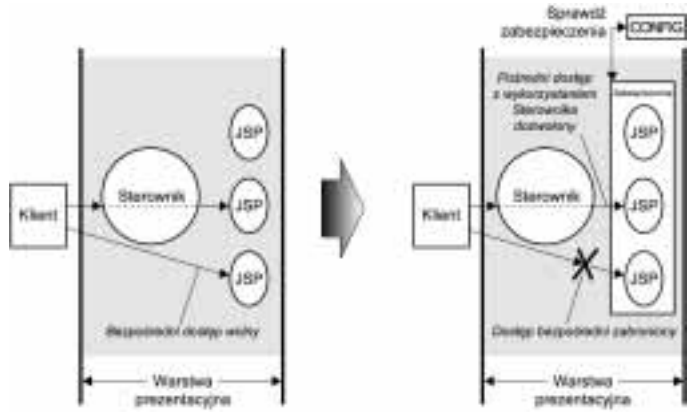
```
public String getXML()
```

Warto zaznaczyć, że tego typu konwersje można przeprowadzać całkowicie niezależnie od JSP. Wybór tego rozwiązania będzie zależał od szeregu czynników, takich jak format przechowywania zawartości oraz użycie dotychczasowo stosowanych technologii.

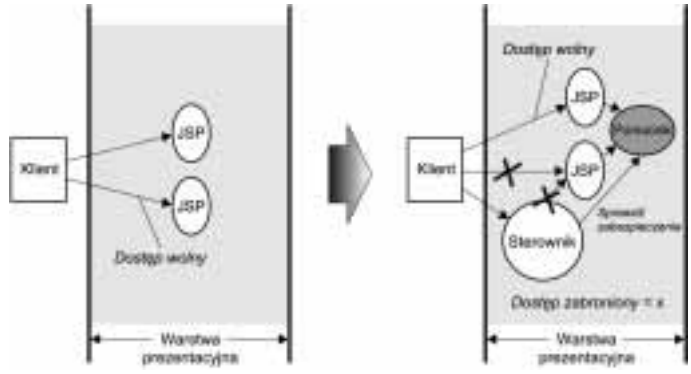
Ukrywanie zasobów przed klientem

Czasami zachodzi potrzeba ograniczenia dostępu klientów do zasobów takich, jak widoki JSP, do których mają oni bezpośredni dostęp.

Dostęp do zasobów można ograniczyć poprzez skonfigurowanie kontenerów lub użycie sterownika.



Rysunek 5.13. *Ograniczenie dostępu poprzez skonfigurowanie kontenerów*



Rysunek 5.14. *Ograniczenie dostępu za pomocą komponentu sterowania*

Motywacja

Często zachodzi potrzeba kontroli przychodzących żądań. Omawiany tu sposób przebudowy umożliwia wprowadzenie opartej na zezwoleniach kontroli i ochrony zasobów.

Jeżeli istnieje potrzeba wprowadzenia kontroli kolejności lub przepływu żądań, należy zastosować rozwiązania omówione w podrozdziale „Korzystanie z synchronizatora" (rozdział 5.).

Mechanika

- Dostęp do pewnych zasobów (na przykład zasobów WWW, serwletów itp.) należy ograniczać poprzez odpowiednią konfigurację, polegającą na przeniesieniu takich zasobów do odpowiednich podkatalogów w podkatalogu */WEB-INF/* aplikacji WWW.
  - Aby na przykład zablokować bezpośredni dostęp przeglądarki do widoku o nazwie *info.jsp* w aplikacji *securityissues*, można umieścić plik źródłowy JSP w następującym podkatalogu: */securityissues/WEB-INF/internalaccessonly/info.jsp*.
- Dostęp można też ograniczać za pomocą komponentu kontrolnego.
  - W tym celu można wykorzystać wzorzec przedstawiony w sekcji „Korzystanie ze sterownika" (rozdział 5.), gdzie pokazano, w jaki sposób sterownik może kontrolować dostęp do chronionych zasobów.

- Dodatkowo, wszystkie zasoby podlegające ochronie mogą same kontrolować dostęp, wywołując odpowiednie klasy pomocnicze.

■ Należy utworzyć jedną lub więcej klas pomocniczych.

- W zależności od implementacji sterownik lub poszczególne strony JSP wywołują klasy pomocnicze, które sprawdzają, czy dany zasób powinien zostać udostępniony.

Przykład

Ograniczenie dostępu poprzez konfigurację pojemnika

Aby dostęp klienta do przykładowej strony JSP *info.jsp* był możliwy tylko poprzez sterownik, należy przenieść stronę JSP do katalogu */WEB-INF/*.

W przykładowej aplikacji WWW o nazwie *corepatterns* skonfigurowano następujący katalog w głównym katalogu serwera:

```
/corepatterns/secure_page.jsp
```

Taka konfiguracja domyślnie umożliwia klientom bezpośredni dostęp do tego zasobu, np. za pomocą poniższego adresu URL:

```
http://localhost:8080/corepatterns/secure_page.jsp
```

Aby zastrzec bezpośredni dostęp, należy przenieść plik JSP do podkatalogu katalogu */WEB-INF/*. Ścieżka dostępu w głównym katalogu serwera będzie wyglądała wtedy następująco:

```
/corepatterns/WEB-INF/privateaccess/secure_page.jsp
```

Zawartość katalogu */WEB-INF/* jest dostępna tylko pośrednio poprzez żądania wewnętrzne, wysyłane na przykład przez sterownik oraz obiekt `RequestDispatcher`. Oznacza to, że adres URL, pod którym przeglądarka może uzyskać dostęp do tego pliku, będzie wyglądał mniej więcej następująco:

```
http://localhost:8080/corepatterns/controller?view=/corepatterns/
➥WEB-INF/privateaccess/secure_page.jsp
```

Uwaga: podany powyżej adres URL ilustruje jedynie omawiane zagadnienia i nie stanowi zalecanej metody przekazywania informacji do serwera. Parametr zapytania widoku nie powinien ujawniać struktury katalogów serwera.

Jeżeli żądanie dotarło do sterownika serwletu, to może on je przekazać do strony *secure_page.jsp*, korzystając z obiektu `RequestDispatcher`.

Natomiast przy próbie uzyskania bezpośredniego dostępu do zasobu:

```
http://localhost:8080/corepatterns/WEB-INF/privateaccess/secure_page.jsp
```

serwer odsyła odpowiedź z informacją o braku dostępu do żądanego zasobu, co pokazano na rysunku 5.15.



Rysunek 5.15. *Ograniczenie dostępu poprzez odpowiednią konfigurację plików*

Ograniczenie dostępu poprzez użycie komponentu kontrolnego

Innym sposobem ograniczania dostępu jest wywołanie komponentu kontrolującego, tak jak pokazano to na rysunku 5.14 i w listingu 5.11.

Listing 5.11. *Kontrola dostępu przy użyciu komponentu*

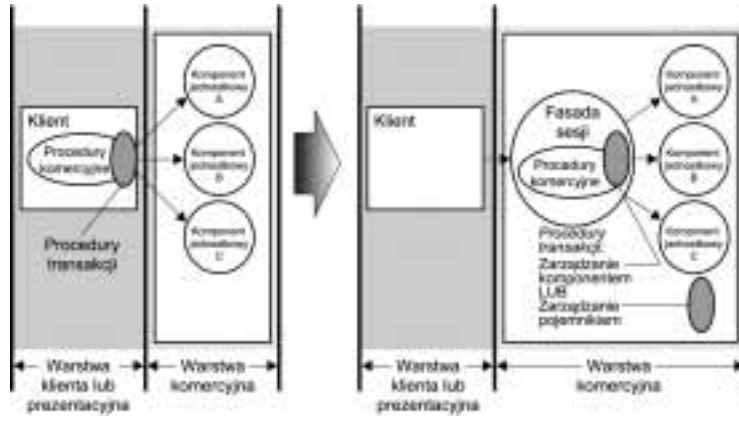
```
<%@ taglib uri="/WEB-INF/corepatternstaglibrary.tld"
  prefix="corepatterns" %>
<corepatterns:guard/>
<html>
<head><title>Ograniczenie dostępu klienta do zasobów</title></head>
<body>

<h2>Klient uzyskuje dostęp do tego widoku, tylko
za pośrednictwem komponentu kontrolnego. Widok ten
wywołuje mechanizm kontroli dostępu za pomocą znacznika
kontrolnego użytego na początku strony.</h2>
</body>
</html>
```

Sposoby przebudowy warstwy komercyjnej i integracyjnej

Zamknięcie komponentów jednostkowych w sesji

Komponenty jednostkowe z warstwy komercyjnej są udostępniane klientom w innej warstwie.

Komponenty jednostkowe należy wyodrębnić poprzez użycie fasady sesji.



Rysunek 5.16. *Zamknięcie komponentów jednostkowych w sesji*

Motywacja

Komponenty jednostkowe (ang. *entity beans*) to rozproszone, trwałe obiekty ogólnego zastosowania. Udostępnienie komponentu jednostkowego klientom w innej warstwie wywołuje przeciążenie sieci i obniżenie wydajności. Każde żądanie klienta oznacza kosztowne pod względem wydajności wywołanie metody odległej.

Użycie komponentów jednostkowych oznacza wprowadzenie transakcji zarządzanej przez pojemniki. Udostępnienie komponentu jednostkowego klientom może wymagać od programisty aplikacji klienckiej przeanalizowania, zaprojektowania oraz demarkacji transakcji, zwłaszcza w przypadku użycia wielu komponentów jednostkowych. Programista taki musi uzyskać dostęp do transakcji użytkownika od obiektu zarządzania transakcjami oraz opracować kod interakcji z komponentami jednostkowymi (interakcja ta ma się odbywać w kontekście transakcji). Ponieważ klient zarządza transakcją, nie jest możliwe wykorzystanie demarkacji transakcji zarządzanej przez pojemniki.

Mechanika

- Procedury komercyjne, które powinny współdziałać z komponentami jednostkowymi, należy wyodrębnić z kodu klienta aplikacji.
  - Do wyodrębnienia procedur z kodu klienta można wykorzystać wzorzec klasy wydobycia (`Extract Class` [Fowler]).
- Następnie należy użyć komponentu sesyjnego jako fasady dla jednostki.
  - Taki komponent sesyjny może zawierać procedury interakcji z komponentem jednostkowym oraz inne potrzebne procedury.
  - Przejdź do akapitu „Fasada sesji”, (rozdział 8.).
- Należy potem zaimplementować komponenty sesyjne w celu utworzenia skonsolidowanej warstwy dostępu do komponentów jednostkowych — w oparciu o wzorzec fasady sesji.
  - Liczne interakcje pomiędzy klientem a jednostkami zostają w ten sposób przesunięte do fasady sesji w warstwie komercyjnej.

- Dzięki temu zredukowana zostaje liczba wywołań metod odległych, dokonywanych przez klienta.

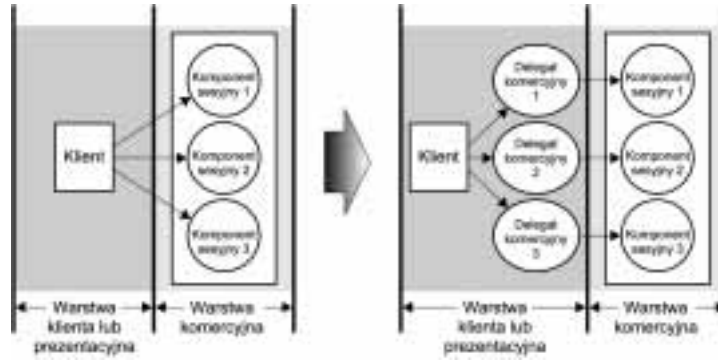
- Następnie należy zaimplementować procedury transakcji w komponentach sesyjnych, jeżeli używane są transakcje zarządzane przez komponenty JavaBean. W przypadku transakcji zarządzanych przez pojemniki należy określić atrybuty transakcji dla komponentu sesyjnego w obiekcie opisu wdrażania.
  - Ponieważ komponent sesyjny współdziała z komponentami jednostkowymi, klient nie jest już odpowiedzialny za demarkację transakcji.
  - Za demarkację transakcji odpowiada teraz albo komponent sesyjny, albo pojemnik — w zależności od tego, czy wybrano model transakcji zarządzanych przez użytkownika czy też przez pojemnik.

Korzystanie z delegatów komercyjnych

Komponenty sesyjne w warstwie komercyjnej są udostępniane klientom w innych warstwach.

Za pomocą delegatów komercyjnych należy rozdzielić warstwy i wyodrębnić szczegóły implementacji.

Rysunek 5.17. *Użycie delegacji komercyjnej*



Motywacja

Komponenty sesyjne umożliwiają implementację fasad dla komponentów jednostkowych. Komponenty sesyjne pozwalają na utworzenie interfejsów dla ogólnych serwisów komercyjnych. Należy jednak pamiętać, że bezpośrednie udostępnienie komponentu sesyjnego dla klienta aplikacji wymusza ścisłe powiązania pomiędzy kodem klienta aplikacji a komponentem sesyjnym.

Udostępnienie komponentu sesyjnego dla klienta aplikacji zwiększa liczbę wywołań komponentu sesyjnego w kodzie klienta. Każda zmiana w interfejsie komponentu sesyjnego ma wpływ na wszystkie wywołania komponentu sesyjnego w kodzie klienta aplikacji, co oznacza problemy z utrzymaniem kodu. Przy korzystaniu z komponentów EJB klienci są dodatkowo narażeni na wyjątki, występujące na poziomie usługi. Może to być szczególnie kłopotliwe w przypadku aplikacji używanej przez różnego rodzaju klientów, z których każdy korzysta z interfejsu komponentu sesyjnego w celu uzyskania określonej usługi.

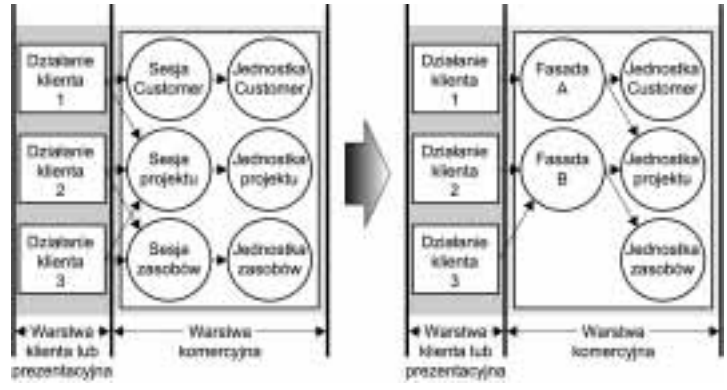
Mechanika

- Dla każdego komponentu sesyjnego, który jest udostępniany klientom bezpośrednio w warstwie, należy zastosować delegata komercyjnego.
  - Delegaci komercyjni to standardowe klasy Javy, w których wyodrębniono szczegóły implementacji warstwy komercyjnej. Klasy te pośredniczą pomiędzy klientami a komponentem sesyjnym, przejmując wyjątki powstające na poziomie obsługi.
  - Przejdź do akapitu „Delegat komercyjny", rozdział 8.
- Należy utworzyć delegacje komercyjne, które będą obsługiwały poszczególne komponenty sesyjne, zazwyczaj jako fasady. Każdej delegacji komercyjnej odpowiada tylko jedna fasada sesji.
  - Delegaci komercyjni redukują ścisłe powiązania pomiędzy warstwą klienta a serwisami komercyjnymi (komponentami sesyjnymi) poprzez wyodrębnienie szczegółów implementacji.
  - Klienci współpracują z delegatami komercyjnymi za pomocą lokalnych wywołań znajdujących się w nich metod.
- Kod usług wyszukiwania oraz zachowywania w pamięci podręcznej należy wyodrębnić w delegacjach komercyjnych.
  - Delegaci komercyjni wyszukują serwisy komercyjne za pomocą lokalizatora usług.
  - Przejdź do akapitu „Lokalizator usług", rozdział 8.

Łączenie komponentów sesyjnych

Pomiędzy komponentami sesyjnymi a komponentami jednostkowymi zachodzi relacja typu „jeden do jednego".

Ogólne serwisy komercyjne należy odwzorować w komponentach sesyjnych. Komponenty sesyjne, które pełnią tylko rolę obiektów proxy komponentów jednostkowych, należy wyeliminować lub też połączyć w pojedynczy komponent sesyjny.



Rysunek 5.18. *Łączenie komponentów sesyjnych*

Motywacja

Relacja typu „jeden do jednego" pomiędzy komponentem sesyjnym a komponentem jednostkowym jest rozwiązaniem niekorzystnym. Takie odwzorowanie wprowadza jedynie warstwę komponentów sesyjnych, pełniących rolę obiektów proxy. Ma to zazwyczaj miejsce wtedy, gdy programista tworzy komponenty sesyjne, które nie stanowią reprezentacji serwisów ogólnego zastosowania, a jedynie wysuwają naprzód komponenty jednostkowe (pośredniczą pomiędzy nimi). W rezultacie komponenty sesyjne nie stanowią fasad, a jedynie obiekty proxy. Wady bezpośredniego udostępniania komponentów jednostkowych klientom omówiono w podrozdziale „Zamknięcie komponentów jednostkowych w sesji".

Na rysunku 5.18 różni klienci obsługują różne rodzaje interakcji. W każdej takiej interakcji bierze udział jeden lub więcej komponentów jednostkowych. W przypadku rozwiązania, polegającego na utworzeniu relacji typu „jeden do jednego" pomiędzy komponentami sesyjnymi a jednostkami, klient współdziała z poszczególnymi komponentami sesyjnymi, które pośredniczą pomiędzy nim a komponentami jednostkowymi. W takiej sytuacji komponenty sesyjne są obiektami proxy, co oznacza, że klienci mają w zasadzie bezpośredni dostęp do komponentów jednostkowych.

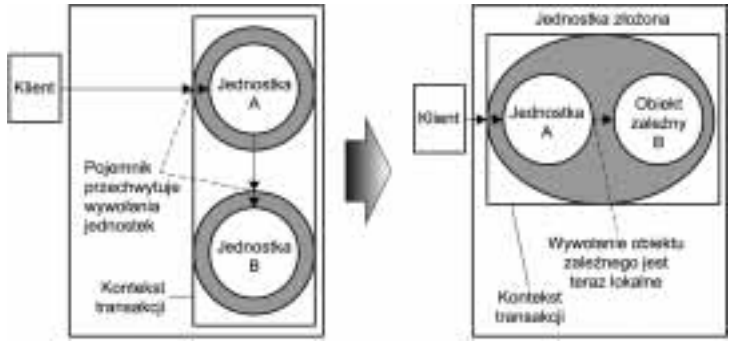
Mechanika

- Komponenty sesyjne należy zaimplementować jako fasadę komponentów jednostkowych. Dzięki temu każdy komponent sesyjny będzie zapewniał klientom interfejs serwisu komercyjnego ogólnego zastosowania.
- Wyspecjalizowane komponenty sesyjne oraz komponenty sesyjne, stanowiące obiekty proxy dla komponentów jednostkowych, należy połączyć w pojedynczy komponent sesyjny.
  - Komponenty sesyjne będą stanowiły reprezentację serwisu komercyjnego ogólnego zastosowania.
  - Komponenty jednostkowe będą stanowiły reprezentację zachowanych informacji, dotyczących transakcji.
  - Przejdź do akapitu „Fasada sesji", rozdział 8.
- Aby uniknąć implementacji każdej interakcji w osobnym komponencie sesyjnym, należy połączyć zbiór powiązanych ze sobą interakcji, w których bierze udział jeden lub więcej komponentów sesyjnych, w pojedynczą fasadę sesji.
  - Pozwala to ograniczyć rolę komponentów sesyjnych, umożliwiających dostęp jednolitych serwisów komercyjnych do komponentów jednostkowych.
  - Liczba fasad sesji powinna zależeć od sposobu pogrupowania interakcji, a nie od liczby jednostek.

Ograniczanie komunikacji międzyjednostkowej

Wiele zależności między komponentami jednostkowymi prowadzi do obniżenia wydajność modelu.

Liczbę zależności pomiędzy komponentami jednostkowymi należy obniżyć poprzez użycie komponentu jednostkowego (jednostki złożonej) oraz obiektów zależnych.



Rysunek 5.19. *Ograniczenie komunikacji między komponentami jednostkowymi*

Motywacja

Komponenty jednostkowe pochłaniają dużo więcej czasu procesora niż zwykłe obiekty Javy. Wywołania metod jednostek są wywołaniami odległymi, przez co obniżają wydajność sieci. Dodatkowo, komponenty jednostkowe muszą współdziałać z zewnętrznym źródłem danych.

Nawet, jeżeli dane dwa komponenty jednostkowe znajdują się w tym samym pojemniku, w momencie, gdy jeden z nich wywołuje drugi komponent, nadal obowiązuje schemat zdalnych wywołań metod (pojemnik uczestniczy w komunikacji). Co prawda w niektórych implementacjach pojemników zoptymalizowano tego rodzaju wywołania poprzez rozróżnianie wywołań pochodzących od obiektów, znajdujących się w tym samym pojemniku, ale nie są to rozwiązania, które można stosować uniwersalnie (obowiązują tylko w określonej grupie produktów danej firmy).

Należy również pamiętać, że jednostka nie jest w stanie dokonać demarkacji transakcji. W przypadku korzystania z komponentów jednostkowych możliwe są tylko transakcje zarządzane przez pojemniki. Oznacza to, że — w zależności od ustawionego atrybutu transakcji metody w komponencie jednostkowym — pojemnik może rozpocząć nową transakcję, wziąć udział w bieżącej transakcji lub też nie wykonać żadnej z tych czynności. Kiedy klient wywołuje metodę w komponencie jednostkowym, do kontekstu transakcji zostaje włączony szereg jednostek zależnych. Powoduje to obniżenie ogólnej wydajności komponentów jednostkowych, ponieważ każda transakcja może oznaczać zablokowanie wielu komponentów jednostkowych, a w rezultacie może doprowadzić do zakleszczenia.

Mechanika

- Jednostki należy zaprojektować jako obiekty ogólnego przeznaczenia, złożone z obiektów głównych i obiektów zależnych.
  - Relacje międzyjednostkowe należy zastąpić relacjami typu jednostka — obiekt zależny.
  - Obiekty zależne nie są komponentami jednostkowymi. Są to obiekty, zawierające się w komponencie jednostkowym. Pomiędzy komponentem

jednostkowym a znajdującymi się w nim obiektami zależnymi występuje relacja lokalna, która nie obciąża sieci.

- Przeprowadzane przez jednostki złożone operacje ładowania oraz przechowywania danych należy zoptymalizować, korzystając ze wzorców: strategii oszczędnego wczytywania oraz strategii optymalizacji zapisu.
- Przejdź do akapitu „Jednostka złożona", rozdział 8.

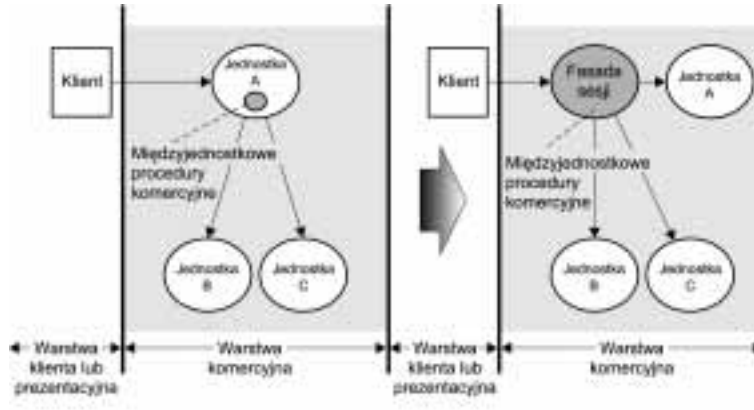
■ Z komponentu jednostkowego należy wyodrębnić procedury współdziałania z innymi jednostkami i przenieść je do komponentu sesyjnego.

- W oparciu o wzorzec metody wydobycia (`Extract Method` [Fowler]) i (lub) metody przesunięcia (`Move Method` [Fowler]) należy przenieść taką procedurę komercyjną do komponentu sesyjnego, stosując w tym celu wzorzec fasady sesji.
- Przejdź do akapitu „Fasada sesji", rozdział 8.

Przesunięcie procedur komercyjnych do sesji

Duża liczba zależności pomiędzy jednostkami obniża wydajność modelu.

Operacje związane z zależnościami pomiędzy komponentami jednostkowymi należy wyodrębnić w komponencie sesyjnym (tworząc fasadę sesji).



Rysunek 5.20. *Przesunięcie procedur komercyjnych do sesji*

Motywacja

W podrozdziale „Ograniczanie komunikacji międzyjednostkowej" omówiono problemy związane z bezpośrednimi zależnościami między komponentami jednostkowymi. Problemy te wynikały z faktu, iż komponent jednostkowy może zawierać procedury komercyjne, które zostały utworzone do obsługi innych komponentów jednostkowych. Takie rozwiązanie wymusza bezpośrednie i pośrednie zależności z innymi jednostkami. Problemy omawiane w powyższym podrozdziale mają również zastosowanie tutaj.

Mechanika

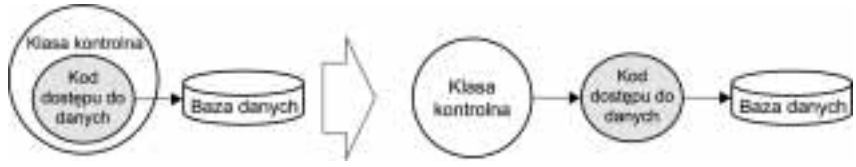
- Procedury komercyjne, związane z interakcją między jednostkami, należy przenieść z komponentu jednostkowego do komponentu sesyjnego.
  - W oparciu o wzorzec metody wydobycia (`Extract Method` [Fowler]) i (lub) metody przesunięcia (`Move Method` [Fowler]) należy przenieść procedury komercyjne do komponentu sesyjnego, z zastosowaniem wzorca fasady sesji.
  - Przejdź do akapitu „Fasada sesji", rozdział 8.
  - Przejdź do akapitu „Zamknięcie komponentów jednostkowych w sesji", rozdział 5.

Ogólne sposoby przebudowy

Separacja kodu dostępu do danych

Kod dostępu do danych jest umieszczany bezpośrednio w klasach, które obsługują operacje niezwiązane z dostępem do danych.

Kod dostępu do danych należy wyodrębnić w nowej klasie, którą następnie należy przesunąć logicznie i (lub) fizycznie bliżej źródła danych.



Rysunek 5.21. *Separacja kodu dostępu do danych*

Motywacja

Tworząc aplikacje, należy zadbać o ich przejrzystą strukturę oraz spójność. Jednocześnie należy unikać zbyt ścisłych zależności pomiędzy różnymi komponentami, zwiększając w ten sposób ich modularność oraz możliwość wielokrotnego wykorzystania.

Mechanika

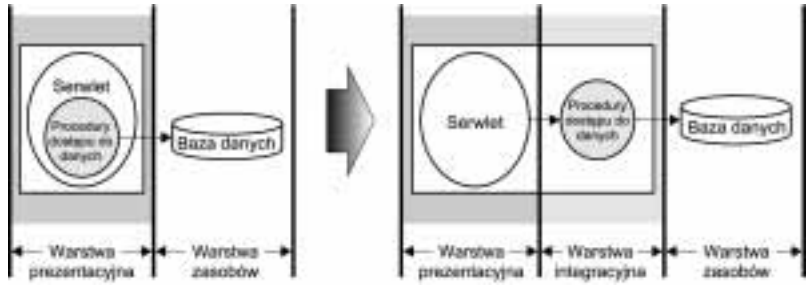
- Procedury dostępu do danych należy wydzielić z obiektu sterowania.
  - W oparciu o wzorzec klasy wydobycia (`Extract Class` [Fowler]) należy utworzyć nową klasę obiektu dostępu do danych i przenieść do niej kod dostępu z określonej klasy.
  - W nazwie utworzonej w ten sposób klasy warto zaznaczyć, że jest to obiekt dostępu do danych.

- Przejdź do akapitu „Obiekt dostępu do danych", rozdział 9.

- Dostęp do danych ze sterownika należy uzyskiwać za pomocą nowej klasy.
- Dalsze informacje na temat podziału struktury aplikacji można znaleźć w podrozdziale „Przebudowa architektury warstw" (rozdział 5.).

Przykład

Za przykład weźmy serwlet, zawierający kod dostępu do danych, którymi są informacje o użytkownikach. Wprowadzenie zmian, opisanych w pierwszych dwóch punktach poprzedniej sekcji, pokazano na rysunku 5.22.

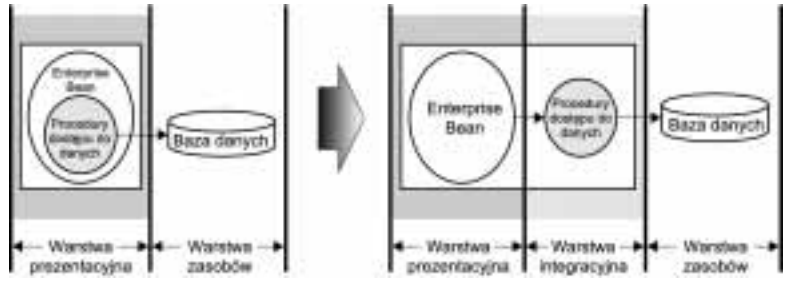


Rysunek 5.22. *Wydzielenie kodu dostępu do danych — przykład serwletu*

W ten sposób uzyskujemy dwie klasy: pierwsza z nich to klasa serwletu, która pełni rolę sterownika, natomiast druga to klasa o nazwie `UserDAO`, pełniąca rolę obiektu dostępu do danych, umożliwiającego dostęp do informacji o użytkownikach. Wszelkie szczegóły implementacji zamieszczone w serwlecie zostały zastąpione przez kod połączenia z bazą danych (JDBC), wyodrębniony w obiekcie `UserDAO`. W rezultacie kod serwletu został znacznie uproszczony.

Inny przykład może stanowić komponent EJB, zawierający kod procedur zachowujących dane. Takie połączenie kodu przechowywania danych z kodem komponentów EJB znacznie obniża przejrzystość kodu oraz niezależność poszczególnych komponentów. Jeżeli kod procedur przechowywania informacji został zawarty w komponencie EJB, to każda zmiana w składzie obiektów trwałych wymaga dokonania zmiany w kodzie trwałości, zawartym w komponencie EJB. Takie zbyt ścisłe powiązanie kodu sprawia, że kod komponentów EJB jest trudny w utrzymaniu. Omawiany w tym podrozdziale sposób przebudowy może stanowić rozwiązanie tego problemu.

Przebudowując aplikację, zmieniamy nasz projekt w sposób przedstawiony na rysunku 5.23.

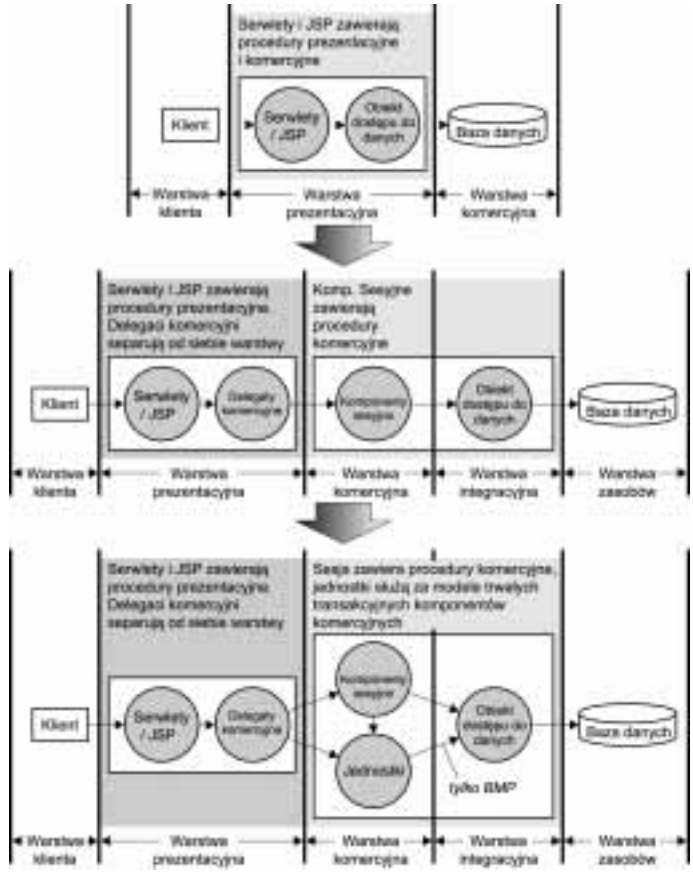


Rysunek 5.23. *Wydzielenie kodu dostępu do danych — przykład komponentu EJB*

Przebudowa architektury warstw

W miarę dodawania kolejnych warstw do aplikacji konieczna staje się zmiana rozmieszczenia procedur przetwarzania oraz dostępu do danych.

Kod dostępu do danych należy przesunąć logicznie i (lub) fizycznie bliżej źródła danych. Logikę przetwarzania należy usunąć z klienta oraz z warstwy prezentacji i wyodrębnić ją w warstwie komercyjnej.



Rysunek 5.24. *Przebudowa architektury warstw*

Motywacja

W podrozdziale „Separacja kodu dostępu do danych" (rozdział 5.) pokazano sposób przebudowy procedur dostępu do danych. W tym podrozdziale omówiliśmy przebudowę pozostałych procedur komercyjnych aplikacji.

W platformie J2EE obowiązuje jasny podział ról komponentów na serwlety, strony JSP oraz komponenty EJB, dzięki czemu możliwe jest podwyższenie poziomu elastyczności, transakcji, bezpieczeństwa i innych aspektów działania aplikacji.

W miarę rozbudowywania komercyjnej funkcjonalności aplikacji zachodzi potrzeba odpowiedniego potraktowania problemów związanych z zagadnieniami trwałości, transakcji, bezpieczeństwa oraz skalowalności usług komercyjnych. Po osiągnięciu pewnego poziomu złożoności konieczne jest wprowadzenie komponentów sesyjnych i jednostkowych, które umożliwiają scentralizowane korzystanie z usług komercyjnych przez wszystkich klientów. Konieczne staje się także wykorzystanie możliwości kontenerów EJB.

Należy pamiętać, że użycie „ciężkich" komponentów, takich jak komponenty EJB, nie zawsze znajduje uzasadnienie w wymaganiach stawianych danej aplikacji. Skorzystanie z możliwości takich komponentów może być jednak konieczne, jeżeli tworzona aplikacja ma spełniać wysokie kryteria bezpieczeństwa, transakcji, elastyczności czy też przetwarzania rozproszonego.

Mechanika

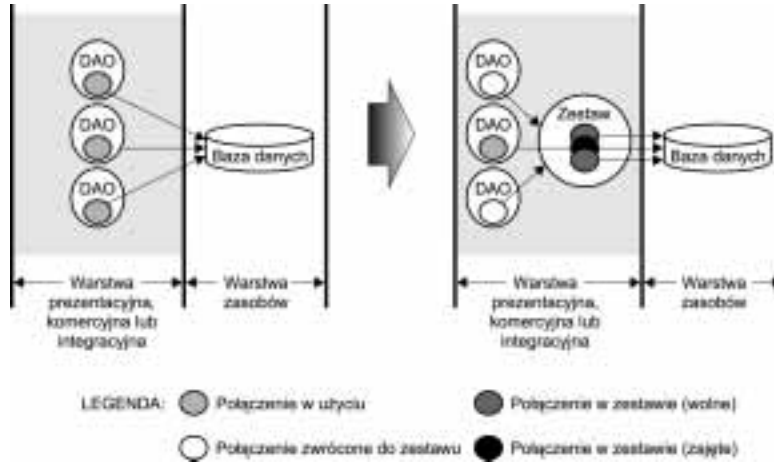
- Kod dostępu do danych należy usunąć z obiektów kontrolnych i jednostkowych, a następnie wyodrębnić go w obiektach dostępu do danych (*Data Access Objects*, DAO).
  - Zobacz „Separacja kodu dostępu do danych", rozdział 5.
- Rozdzielić kod przetwarzania prezentacji oraz przetwarzania komercyjnego. Przetwarzanie komercyjne powinno być obsługiwane przez komponenty sesyjne. Kod przetwarzania prezentacji powinien pozostać w serwletach oraz stronach JSP.
  - Takie rozwiązanie należy zastosować, jeżeli aplikacja jest na tyle złożona, że wymaga takiego rozdzielenia oraz jeżeli zachodzi potrzeba skonsolidowania logiki w warstwie komercyjnej tak, aby wszyscy klienci (a nie tylko klienci warstwy prezentacji) mogli korzystać z tych samych usług komercyjnych.
  - Taką funkcjonalność zapewnia wprowadzenie komponentów sesyjnych, których zadaniem będzie przetwarzanie usług komercyjnych. Komponenty sesyjne uzyskują dostęp do składu obiektów trwałych poprzez obiekty dostępu do danych (DAO).
  - W razie potrzeby dla komponentów sesyjnych można zastosować demarkację, zarządzaną przez komponenty lub przez pojemniki.
  - Przejdź do akapitu „Fasada sesji", rozdział 8.
- Komponenty jednostkowe należy zastosować dla współdzielonych przez model, transakcyjnych, trwałych obiektów komercyjnych ogólnego zastosowania. Punkt ten należy pominąć, jeżeli ich użycie nie znajduje odzwierciedlenia w wymaganiach stawianych aplikacji.
  - Rozwiązanie to należy zastosować, gdy zwiększa się stopień złożoności trwałych obiektów komercyjnych i zachodzi potrzeba skorzystania z możliwości komponentów jednostkowych, takich jak transakcje oraz trwałość zarządzana przez pojemniki.
  - Komponenty jednostkowe umożliwiają demarkację transakcji, zarządzaną przez pojemniki. Pozwala to na deklaratywne programowanie demarkacji transakcji, bez konieczności kodowania procedur transakcji na stałe w komponentach jednostkowych.
  - Przejdź do akapitu „Obiekt wartości" oraz „Jednostka złożona", rozdział 8.

- Warstwę prezentacji należy oddzielić od warstwy komercyjnej za pomocą delegatów komercyjnych.
  - Delegacje komercyjne umożliwiają oddzielenie komponentów warstwy prezentacji od komponentów warstwy komercyjnej i wyodrębnienie z nich kodu złożonych operacji wyszukiwania i innych szczegółów implementacji.
  - Przejdź do akapitu „Delegat komercyjny", rozdział 8.

Korzystanie z zestawu połączeń

Połączenia z bazą danych nie są współdzielone przez klientów. Przy wywoływaniu bazy danych klienci sami zarządzają połączeniami.

Wprowadzenie zestawu połączeń pozwala na wstępne zainicjowanie określonej liczby połączeń, co korzystnie wpływa na elastyczność i wydajność aplikacji.



Rysunek 5.25. *Korzystanie z zestawu połączeń*

Motywacja

Otwarcie połączenia z bazą danych to dość kosztowna operacja, której przeprowadzenie wymaga odpowiedniej ilości czasu i zasobów. Ma to wpływ zarówno na wydajność, jak i na efektywność. Ponieważ w sytuacji, w której każdy klient posługuje się własnym połączeniem, liczba połączeń jest ograniczona, zazwyczaj szybko osiąga ona swą maksymalną wartość.

Problem ten powstaje w warstwie prezentacji projektów, w których technologia EJB jest wprowadzana fazami. W takim przypadku komponenty w warstwie prezentacji początkowo współdziałają bezpośrednio z bazą danych, a następnie kod dostępu do danych zostaje przesunięty do warstwy komercyjnej i wyodrębniony w warstwie EJB. Przejdź do akapitów „Separacja kodu dostępu do danych" oraz „Przebudowa architektury warstw" (rozdział 5.).

Mechanika

- Należy utworzyć interfejs zarządzania połączeniami, którego metody będą umożliwiały pobieranie i zwracanie połączeń.
- W oparciu o wzorce klasy wydobycia (*Extract Class* [Fowler]) i (lub) metody przesunięcia (*Move Method* [Fowler]) należy przesunąć istniejący kod przydzielania połączeń do klasy, która implementuje (utworzony w poprzednim punkcie) interfejs zarządzania połączeniami.
  - W miejscach, z których usunięto kod połączeń, należy wstawić wywołania instancji nowej klasy (i jej metod), na przykład:

    `connectionMgr.getConnection()` oraz `connectionMgr.returnConnection(conn)`.[1]
  - Należy pamiętać, że w wersji 2. specyfikacji JDBC opisano standardowy mechanizm tworzenia zestawu połączeń. Jeżeli mechanizm ten jest dostępny, to zalecane jest zastosowanie go przy tworzeniu zestawu połączeń. W wersji 2. specyfikacji JDBC interfejs zarządzania połączeniami nosi nazwę *javaxsql.DataSource* i pozwala na przebudowę obiektów połączeń (`Connection`).
  - Na tym etapie standaryzacja dotyczy jedynie struktury oraz interfejsu. Nie zmienia się natomiast funkcjonalność.
  - Implementacja zestawów połączeń jest możliwa po zastosowaniu zalecanej przebudowy JDBC 2.0 DataSource.
- Aby zaimplementować zestawy połączeń, należy odpowiednio dostosować implementację metod zwracających połączenia (znajdujących się w klasie zarządzającej połączeniami), inicjując w ten sposób wstępnie pewną liczbę obiektów połączeń (`Connection`), a następnie rozdzielić je między klientów.
  - Istnieje wiele powszechnie dostępnych implementacji, które można wykorzystać w tym celu.
  - Klientami utworzonych w ten sposób instancji klasy zarządzającej połączeniami są zazwyczaj obiekty DAO. Przejdź do podrozdziału „Separacja kodu dostępu do danych", rozdział 5.
  - W miarę rozbudowywania projektu kod dostępu do danych jest zwykle przesuwany w kierunku bazy danych. Przejdź do podrozdziału „Przebudowa architektury warstw", rozdział 5.

[1] **gdzie** `connectionMgr` **to nazwa klasy zarządzającej połączeniami, a** `getConnection()` **oraz** `returnConnection()` **to, kolejno, metody pobierające i zwracające połączenie. Parametrem drugiej z tych metod jest połączenie** — `conn` — *przyp. tłum.*