Andrzej Marciniak

JavaServer Faces i Eclipse Galileo

Tworzenie aplikacji WWW

Jak projektować estetyczne i wygodne interfejsy użytkownika aplikacji WWW?

Od czego zacząć projekt wykorzystujący JavaServer Faces?

Co oferuje środowisko Eclipse Galileo, a co narzędzia Web Tools Platform?

Helion

helion.pl

Idź do

- Spis treści
- Przykładowy rozdział

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

JavaServer Faces i Eclipse Galileo. Tworzenie aplikacji WWW

Autor: Andrzej Marciniak
ISBN: 978-83-246-2656-4
Format: 158×235, stron: 384

- Jak projektować estetyczne i wygodne interfejsy użytkownika aplikacji WWW?
- Od czego zacząć projekt wykorzystujący JavaServer Faces?
- Co oferuje środowisko Eclipse Galileo, a co narzędzia Web Tools Platform?

Dobry interfejs aplikacji WWW to połowa jej sukcesu. Osiągnij go z JavaServer Faces! Język Java od lat zdobywa i ugruntowuje swoją popularność wśród programistów i twórców aplikacji WWW, a rozmaite platformy i rozwiązania, w których jest on wykorzystywany, zostały na stałe włączone do pakietu narzędzi stosowanych przez wielu z nich na co dzień. Jednym z najbardziej popularnych tego typu narzędzi jest JavaServer Faces. Można dzięki niemu w prosty sposób tworzyć interfejsy użytkownika aplikacji, wykorzystujące platformę Java EE. Ten spójny i kompletny szkielet programistyczny jest obecnie najbardziej elastycznym, najlepiej dopracowanym i najprostszym w użyciu rozwiązaniem, opartym na technologii serwletów.

Jednak „najprostszy" wcale nie musi oznaczać „prosty", o czym z pewnością miało okazję przekonać się wielu studentów kierunków informatycznych i profesjonalnych programistów, którzy postanowili praktycznie zapoznać się z możliwościami tej technologii. Nieocenioną pomocą okaże się dla nich książka „JavaServer Faces i Eclipse Galileo. Tworzenie aplikacji WWW", dzięki której można uniknąć wielu typowych błędów i nauczyć się biegle korzystać z JSF, zdobywając przy tym kompletną wiedzę na temat mechanizmów i rozwiązań zapewniających działanie tej platformy. Co więcej, opisano tu nie tylko samą technologię, lecz również sposób jej praktycznego wykorzystania w konkretnych projektach, co w przyszłości zaowocuje z pewnością opracowaniem niejednej doskonałej i cieszącej oko aplikacji WWW.

- Mechanizmy działania aplikacji WWW i sposoby ich projektowania w oparciu o język Java
- Podstawowe informacje na temat szkieletu programistycznego JSF
- Realizacja praktycznego projektu z wykorzystaniem JavaServer Faces
- Rozszerzanie standardowej implementacji JSF i tworzenie niestandardowych interfejsów użytkownika
- Opis środowiska programistycznego Eclipse Galileo oraz pakietu narzędzi Web Tools Platform

Naucz się szybko i sprawnie tworzyć rozbudowane interfejsy użytkownika aplikacji WWW za pomocą szkieletu programistycznego JavaServer Faces

Kontakt

Helion SA
ul. Kościuszki 1c
44-100 Gliwice
tel. 32 230 98 63
e-mail: helion@helion.pl
© Helion 1991–2010

Spis treści

Wstęp **7**
Geneza książki 7
Cele 9
Czytelnicy 10
Układ zagadnień 10
Materiały pomocnicze 11
Podziękowania 12

Rozdział 1. Przed przystąpieniem do pracy z JSF **13**
1.1. Dlaczego JSF? 13
1.1.1. Wersje JSF 15
1.2. JSF i Java EE (J2EE) 17
1.2.1. Serwlety i strony JSP 19
1.2.2. Technologie prezentacji 23
1.2.3. MVC w aplikacjach webowych 27
1.2.4. Implementacja MVC w JSF 32
1.2.5. Kluczowe elementy JSF — podsumowanie 38
1.3. Pierwsza aplikacja JSF w środowisku Eclipse 38
1.3.1. Instalacja kontenera serwletów 40
1.3.2. Instalacja środowiska Eclipse Galileo 40
1.3.3. Integracja kontenera serwletów ze środowiskiem Eclipse 43
1.3.4. Tworzenie projektu JSF 44
1.3.5. Praca z JSF w oknie roboczym 48
1.3.6. Definiowanie modelu 53
1.3.7. Definiowanie komponentu wspierającego 54
1.3.8. Utworzenie strony do wprowadzania danych 56
1.3.9. Utworzenie strony do wyświetlania wyniku 59
1.3.10. Deklarowanie reguł nawigacji 62
1.3.11. Uruchamianie aplikacji 63
1.3.12. Podsumowanie przykładu 65

Rozdział 2. Planowanie, modelowanie i projektowanie aplikacji JSF na platformie Java EE **67**
2.1. Modelowanie interfejsu użytkownika przy wykorzystaniu techniki scenopisu 68
2.1.1. Przypadki użycia 69
2.1.2. Model ekranów interfejsu użytkownika 70
2.1.3. Model nawigacji 72
2.1.4. Prototypy ekranów 73
2.1.5. Diagram maszyny stanowej 75

2.2. Warstwa biznesowa ... 77
2.2.1. Rola i elementy warstwy biznesowej ... 77
2.2.2. Interfejs do warstwy trwałości danych ... 85
2.3. Organizacja kodu aplikacji w Eclipse ... 95
2.3.1. Projekty i moduły Java EE w Eclipse ... 95
2.3.2. Zastosowanie projektu użytkowego do przechowywania kodu warstwy biznesowej ... 96
2.3.3. Moduły zależne ... 99

Rozdział 3. Używanie JSF ... 101
3.1. Komponenty zarządzane ... 101
3.1.1. Deklaracja komponentu ... 102
3.1.2. Zasięg komponentów ... 105
3.1.3. Inicjalizacja właściwości komponentów ... 106
3.1.4. Odwołania do komponentów zarządzanych ... 113
3.1.5. Komponenty referowane ... 115
3.1.6. Komponenty wspierające ... 116
3.1.7. Język wyrażeń UEL ... 118
3.1.8. Konfiguracja komponentów i innych zasobów ... 123
3.2. Obsługa zdarzeń ... 127
3.2.1. Typy zdarzeń i metody ich obsługi ... 127
3.2.2. Natychmiastowe wykonywanie metod obsługi zdarzeń ... 136
3.2.3. Parametryzacja komponentów UI ... 137
3.3. Nawigacja ... 140
3.3.1. Definiowanie reguł nawigacji ... 140
3.3.2. Nawigacja statyczna i dynamiczna ... 143
3.3.3. Rozstrzyganie niejednoznaczności w regułach ... 144
3.3.4. Przekierowanie i nawigacja poza JSF ... 144
3.4. Przetwarzanie pakietów zasobów i internacjonalizacja ... 146
3.4.1. Lokalizatory ... 147
3.4.2. Pakiety zasobów ... 150
3.4.3. Pakiety komunikatów ... 154
3.4.4. Internacjonalizacja w warstwie biznesowej ... 157
3.5. Konwersja ... 159
3.5.1. Standardowe konwertery ... 159
3.5.2. Obsługa błędów konwersji ... 162
3.6. Walidacja ... 164
3.6.1. Implementacja metod walidacji ... 165
3.6.2. Standardowe walidatory ... 168
3.6.3. Kombinowanie różnych walidatorów ... 168
3.6.4. Wymuszanie wprowadzania danych ... 169
3.6.5. Pomijanie weryfikacji ... 169
3.7. Standardowe znaczniki i komponenty UI ... 169
3.7.1. Komponenty i znaczniki biblioteki html ... 170
3.7.2. Komponenty i znaczniki biblioteki core ... 201

Rozdział 4. Dopasowywanie JSF ... 211
4.1. Przetwarzanie żądań ... 211
4.1.1. Scenariusze obsługi żądań ... 211
4.1.2. Standardowy cykl przetwarzania żądania JSF ... 214
4.1.3. Kolejka zdarzeń ... 225
4.1.4. Kolejka komunikatów ... 228
4.2. Konfigurowanie i rozszerzanie standardowej implementacji ... 229
4.2.1. Infrastruktura aplikacji JSF ... 229
4.2.2. Mechanizm nawigacji ... 238

4.2.3. Mechanizm zarządzania widokami ... 238
4.2.4. Mechanizm zarządzania stanem ... 239
4.2.5. Mechanizm przetwarzania wyrażeń EL ... 239
4.2.6. Mechanizm obsługi akcji ... 243
4.3. Model komponentów ... 243
4.3.1. Struktura klas ... 244
4.3.2. Identyfikatory komponentów ... 251
4.4. Praktyczne zastosowania obiektów PhaseListener ... 255
4.4.1. Wzorzec POST-Redirect-GET ... 256
4.4.2. Generowanie danych binarnych ... 258
4.5. Złożone tabele ... 260
4.5.1. Modele danych ... 260
4.5.2. Przykład tabeli interaktywnej ... 261
4.6. JSF i bezpieczeństwo ... 264
4.6.1. Bezpieczeństwo zarządzane przez kontener ... 265
4.6.2. Bezpieczeństwo zarządzane przez aplikację ... 270

Rozdział 5. Tworzenie komponentów użytkownika ... 275
5.1. Komponenty interfejsu użytkownika (UI) ... 275
5.1.1. Implementacja komponentu ... 276
5.1.2. Przechowywanie stanu komponentu ... 280
5.1.3. Rejestracja komponentu ... 282
5.1.4. Generowanie i obsługa zdarzeń ... 282
5.1.5. Integracja z JSP ... 284
5.2. Renderery ... 289
5.2.1. Podstawy implementacji rendererów ... 290
5.2.2. Renderery i JavaScript ... 291
5.2.3. Rejestracja rendererów ... 295
5.3. Konwertery ... 296
5.3.1. Podstawy implementacji konwerterów ... 297
5.3.2. Rejestracja konwerterów ... 300
5.3.3. Integracja konwerterów z JSP ... 302
5.4. Walidatory ... 304
5.4.1. Podstawy implementacji walidatorów ... 305
5.4.2. Rejestracja walidatorów ... 306
5.4.3. Integracja walidatorów z JSP ... 308
5.5. Komponenty JSF i AJAX ... 309
5.5.1. Podstawy AJAX-a ... 310
5.5.2. Biblioteka AJAX4JSF ... 313
5.5.3. Biblioteki AJAX-a ... 315

Rozdział 6. Narzędzia wsparcia i integracja z innymi szkieletami ... 327
6.1. Narzędzia Web Tools Platform ... 327
6.1.1. Zarządzanie serwerem aplikacji ... 328
6.1.2. Monitorowanie portów ... 330
6.1.3. Narzędzia ułatwiające pracę z bazami danych ... 333
6.2. Technologie widoku ... 338
6.2.1. Facelets ... 339
6.2.2. Apache Tiles 2 ... 344
6.2.3. Smile ... 350

Dodatek A Standardy kodowania IANA ... 353

Dodatek B Założenia JSF 2.0 ... 355

Dodatek C Wspólne elementy konfiguracji ... 357

Dodatek D Komunikaty o błędach ... 359

Komunikaty o błędach komponentów ... 359
Komunikaty o błędach konwerterów ... 359
Komunikaty o błędach walidatorów ... 360

Dodatek E Atrybuty przekazywane HTML i zdarzenia DHTML ... 361

Literatura ... 363

Skorowidz ... 365

Rozdział 2.

Planowanie, modelowanie i projektowanie aplikacji JSF na platformie Java EE

Po zapoznaniu się z podstawowymi wiadomościami na temat architektury i możliwości szkieletu JSF możemy przejść do zagadnień związanych z wytwarzaniem realnych aplikacji WWW w środowisku programistycznym Eclipse Galileo. W niniejszym rozdziale wybrane fazy wytwarzania aplikacji są zilustrowane na przykładzie **Internetowego Systemu Recenzowania Publikacji** (w skrócie **ISRP**), przeznaczonego do zapewnienia obsługi informatycznej pracy redakcji czasopism naukowych oraz wspomagania organizacji konferencji naukowych. Zakres funkcjonalności projektowanego systemu obejmuje obsługę rejestracji artykułów zgłaszanych przez autorów, proces recenzji artykułów oraz administrowanie kontami użytkowników.

Nie jest intencją autora zamieszczenie w tym miejscu pełnej dokumentacji systemu ISRP jako złożonego studium przypadku użycia JSF (ang. *case study*), ale raczej przedstawienie pewnych praktyk projektowych związanych z architekturą i projektowaniem aplikacji JSF na platformie Java EE. Prezentowane rozwiązania są wypadkową doświadczeń wielu osób w zakresie wykorzystania szkieletu JSF do wytwarzania warstwy prezentacji w aplikacjach o architekturze wielowarstwowej. Szczególna uwaga poświęcona jest wybranym aspektom związanym z czynnościami fazy przedimplementacyjnej projektu, takim jak modelowanie interfejsu użytkownika oraz projektowanie aplikacji oparte na architekturze wielowarstwowej. Zamieszczone w tym rozdziale model i założenia projektowe będą rozwijane w dalszej części książki i stanowić jednocześnie odniesienie dla kolejnych przykładów ilustrujących elementy technologii JSF.

2.1. Modelowanie interfejsu użytkownika przy wykorzystaniu techniki scenopisu

Jakość modelu projektowanego systemu często decyduje o powodzeniu projektu informatycznego, a jego najbardziej pożądaną cechą jest kompromisowość. Model powinien być jednocześnie zrozumiały dla klientów, którzy na jego podstawie oceniają funkcjonalność systemu, jak też zawierać wszystkie niezbędne dane dla zespołu projektowego, który z niego korzysta w fazie projektowania. Z jednej strony model powinien być dopasowany do technologii stosowanych w fazach projektowania i implementacji (przy założeniu, że są one znane wcześniej i wynikają np. z wymagań niefunkcjonalnych), a jednocześnie nie powinien wymagać rozległej wiedzy na ich temat od analityków wytwarzających model.

Ze względu na komponentowy charakter szkieletu JSF, **scenopis** wydaje się techniką modelowania, która doskonale spełnia opisane wyżej kryteria. Scenopis specyfikuje model zawierający analizę funkcjonalną, komponenty funkcjonalne oraz interakcje pomiędzy nimi na ekranach oraz przepływy i projekty ekranów. Największą zaletą scenopisu jest to, że jest on zrozumiały dla osób spoza branży informatycznej, co zapewniają diagramy przypadków użycia i prototypy ekranów. W dalszej części rozdziału zaprezentowane zostaną poszczególne elementy scenopisu, ilustrowane wybranymi fragmentami modelu aplikacji ISRP.

Scenopis i metodyki modelowania, projektowania i dokumentowania systemów informatycznych

Metodyki wytwarzania systemów informatycznych stanowią jeden z głównych nurtów inżynierii oprogramowania. Obecnie standardowym narzędziem do modelowania systemów informatycznych jest język UML w wersji 2.1, który stanowi jednocześnie system wizualizacji, specyfikowania oraz dokumentowania elementów systemu informatycznego. UML jest używany wraz z jego reprezentacją graficzną — jego elementom przypisane są symbole, które wiązane są ze sobą na diagramach. Ze względu na to że UML został stworzony z myślą o opisie świata w analizie obiektowej, jego specyfikacja nie definiuje gotowych rozwiązań dla specyficznych zastosowań, takich jak modelowanie interfejsu użytkownika. Ponieważ nie powstał zunifikowany standard dla modelowania zachowania i struktury aplikacji WWW, np. w zakresie modelowania nawigacji, przepływów sterowania na ekranach czy projektów ekranów, różne organizacje i firmy forsują własne rozszerzenia języka UML w tym zakresie.

Opierając się na UML, stworzono metodykę projektowania oprogramowania RUP (*Rational Unified Process*), która definiuje szablon procesu iteracyjnego wytwarzania oprogramowania opracowany przez firmę Rational Software Corporation. Szablon RUP pozwala na korzystanie z jego elementów w zależności od potrzeb projektowych. Spośród najlepszych praktyk i zasad inżynierii oprogramowania, które są w nim zawarte, warto wskazać dość istotne z punktu widzenia specyfiki aplikacji tworzonych

w JSF zarządzanie wymaganiami (ang. *Requirement Management*), używanie architektury bazującej na komponentach (ang. *Component-based Architecture*) czy graficzne projektowanie oprogramowania.

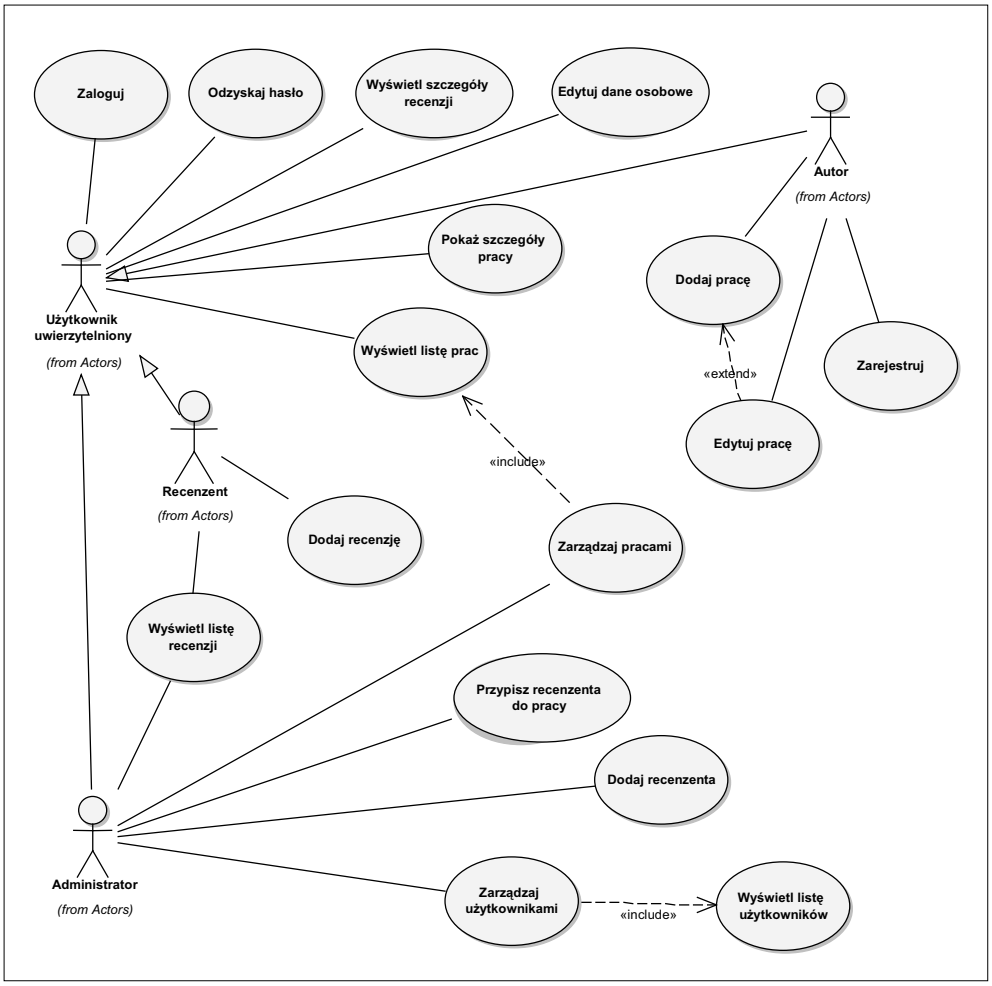
Scenopis (ang. *storyboard*) jest modelem projektowym, zwykle tworzonym przez analityków biznesowych i projektantów interfejsu użytkownika, dostarczonym zespołowi deweloperów jako część specyfikacji funkcjonalnej. Idea scenopisu wywodzi się z przedstawionej wyżej metodyki RUP, która definiuje m.in. *model projektowania interakcji oparty na doświadczeniu użytkownika* (ang. *User-eXperienc*e — *UX*) w postaci ekranów, ich dynamicznej zawartości i sposobu nawigacji. Typowy scenopis zawiera pięć elementów:

- przypadki użycia opisujące wymagania funkcjonalne i przepływy zdarzeń,
- model ekranów UI specyfikujący ekrany, komponenty i elementy UX,
- diagram nawigacji ilustrujący związki pomiędzy ekranami (przede wszystkim reguły nawigacji), oparty na notacji graficznej diagramu klas,
- makiety ekranów (ang. *mock-up*) pokazujące szczegółowo elementy funkcjonalne interfejsu GUI, tj. kontrolki i ich rozmieszczenie,
- diagramy maszyny stanowej dla każdego ekranu ilustrujące interakcję warstwy kontrolera ekranu z warstwą modelu aplikacji.

2.1.1. Przypadki użycia

Wygodnym narzędziem stosowanym na potrzeby identyfikacji i dokumentacji wymagań funkcjonalnych są diagramy **przypadków użycia** (ang. *Use Case* — *UC*) w języku UML. Ilustrują one funkcjonalność systemu poprzez wyszczególnienie i przedstawienie za pomocą symboli graficznych: aktorów inicjujących działania, przypadków użycia systemu oraz występujących pomiędzy nimi związków. Dzięki swojej prostocie oraz abstrahowaniu od rozwiązań technicznych, diagramy przypadków użycia są zrozumiałe nawet dla osób, które nie posiadają wykształcenia informatycznego. Z założenia stanowią one czytelne narzędzie pozwalające na komunikację pomiędzy twórcami systemu i jego potencjalnymi odbiorcami czy inwestorami. Podsumowując, przypadki użycia opisują, co system powinien robić, a nie jak ma to robić.

Na rysunku 2.1 przedstawiono diagram przypadków użycia modelujący aktorów, przypadki użycia oraz powiązania pomiędzy nimi, występujące w systemie ISRP. Schematyczny rysunek postaci ludzkiej złożony z kilku kresek jest symbolem graficznym *aktora*, definiującego spójny zbiór ról odgrywanych przez użytkowników przypadku użycia w czasie jego realizacji. Przypadki użycia oznaczone są przy wykorzystaniu wypełnionych elips, a ich realizacje zwykłymi liniami. Relacje rozszerzania i zawierania dla przypadków użycia opisane są przy zastosowaniu stereotypów umieszczonych w cudzysłowach ostrokątnych. Relacje uogólniania (w sensie związku taksonomicznego, opisującego dziedziczenie cech) przedstawione zostały za pomocą linii zakończonych strzałką z pustym grotem.



Rysunek 2.1. *Diagram przypadków użycia dla ISRP*

Dokładny opis wykorzystania diagramów UC do modelowania funkcjonalności systemu Czytelnik znajdzie w książce *Język UML 2.0 w modelowaniu systemów informatycznych*, a wyczerpujący opis scenariuszy przypadków użycia (tekstowej formy prezentacji UC) znajduje się w książce *Writing Effective Use Cases*[1].

2.1.2. Model ekranów interfejsu użytkownika

Model ekranów UI stanowi centralny punkt wyjściowy dla równoległego rozwoju aplikacji, prowadzonego w zespołach projektantów stron WWW i programistów aplikacji, stanowiąc jednocześnie interfejs umożliwiający integrację kodu wytwarzanego przez

[1] Polskie wydanie: Alistair Cockburn, *Jak pisać efektywne przypadki użycia*, tłum. Krzysztof Stencel, Warszawa 2004.

oba zespoły. Model ekranów UI definiuje zbiór ekranów (wraz z ich zawartością) niezbędnych do zrealizowania wymagań funkcjonalnych określonych przez przypadki użycia. W przypadku zawartości ekranów model definiuje wyłącznie elementy funkcjonalne, jakie powinny znaleźć się na nich, bez określania szczegółów dotyczących typów kontrolek (np. pola tekstowego albo listy rozwijanej) czy ich rozmieszczenia.

W technice scenopisów standardowym elementom UML nadawane są nowe znaczenia poprzez użycie stereotypów przedstawionych w tabeli 2.1.

Tabela 2.1. *Stereotypy modelu UI wykorzystywane w technice scenopisów*

Stereotyp	Rozszerzany element UML	Reprezentuje
Screen	Class	Ekran (w postaci strony WWW).
Compartment	Class	Fragment ekranu (strony WWW) przeznaczony do wykorzystania na wielu stronach aplikacji WWW (definiujący sekcję strony np. stopkę, nagłówek itp.).
Form	Class	Formularz strony WWW.
Input	Attribute	Pole wejściowe formularza.
Submit	Operation	Element zatwierdzający formularz i generujący akcję.

Statyczne informacje takie jak etykiety, obrazki czy panele nie są w ogóle reprezentowane w modelu UI. Dane biznesowe są przedstawiane na ekranie przy użyciu **atrybutów**. Jeśli atrybut opatrzony jest stereotypem `<<input>>`, oznacza to, że służy on do wprowadzania danych. Pełny opis atrybutu prezentowany jest według następującego formatu: *widoczność nazwa :typ liczebność =domyślna_wartość {właściwości}*. Załóżmy, że mamy ekran służący do tworzenia konta nowego użytkownika, dostępny wyłącznie dla administratora systemu. Pole służące do wprowadzania uprawnień dla nowego użytkownika może zostać opisane w następujący sposób:

```
<<input>> +role :String 1 =reviewer {user.role==admin}
```

Powyższy zapis informuje nas, że atrybut `role` reprezentuje pole wejściowe dla danych typu String, wypełniane przez użytkowników z uprawnieniami administratora. Widoczność elementu określa, czy jego zawartość jest wyświetlana na ekranie (+), czy też jest ukryta (-). Nazwa atrybutu powinna identyfikować pole wprowadzania danych (np. wartość atrybutu `name` dla odpowiedniego pola formularza). Typem atrybutu może być dowolny typ prymitywny lub obiektowy, włącznie z abstrakcyjnymi typami danych, reprezentującymi złożone struktury danych (takie jak listy czy mapy, które mogą być reprezentowane przez klasy abstrakcyjne `List` bądź `Map`). Pozostałe właściwości atrybutu są opcjonalne i nie muszą być definiowane. Liczebność określa, ile instancji elementu może być renderowanych na ekranie, wartość domyślna jest wartością używaną dla elementów, które nie są inicjowane, a nawiasy klamrowe umożliwiają zamieszczenie dodatkowych właściwości bądź ograniczeń dotyczących atrybutu, np. w zakresie dostępu do pola danych.

Zdarzenia generowane przez interfejs użytkownika oraz akcje użytkownika reprezentowane są za pomocą **operacji**. Jeśli operacja jest dodatkowo opatrzona stereotypem `<<submit>>`, oznacza to, że wskazany element UI służy do zatwierdzania formularza

i generowania akcji, której wynik może wpływać na nawigację aplikacji. Pełny opis operacji prezentowany jest według następującego formatu: *widoczność nazwa (lista_parametrów) :zwracany_typ {właściwości}*. Na przykład element wyzwalający akcję polegającą na przejściu do okna edycji bieżącego użytkownika może być opisany w następujący sposób:

```
<<submit>> + edit ():String {current.user==notEmpty}
```

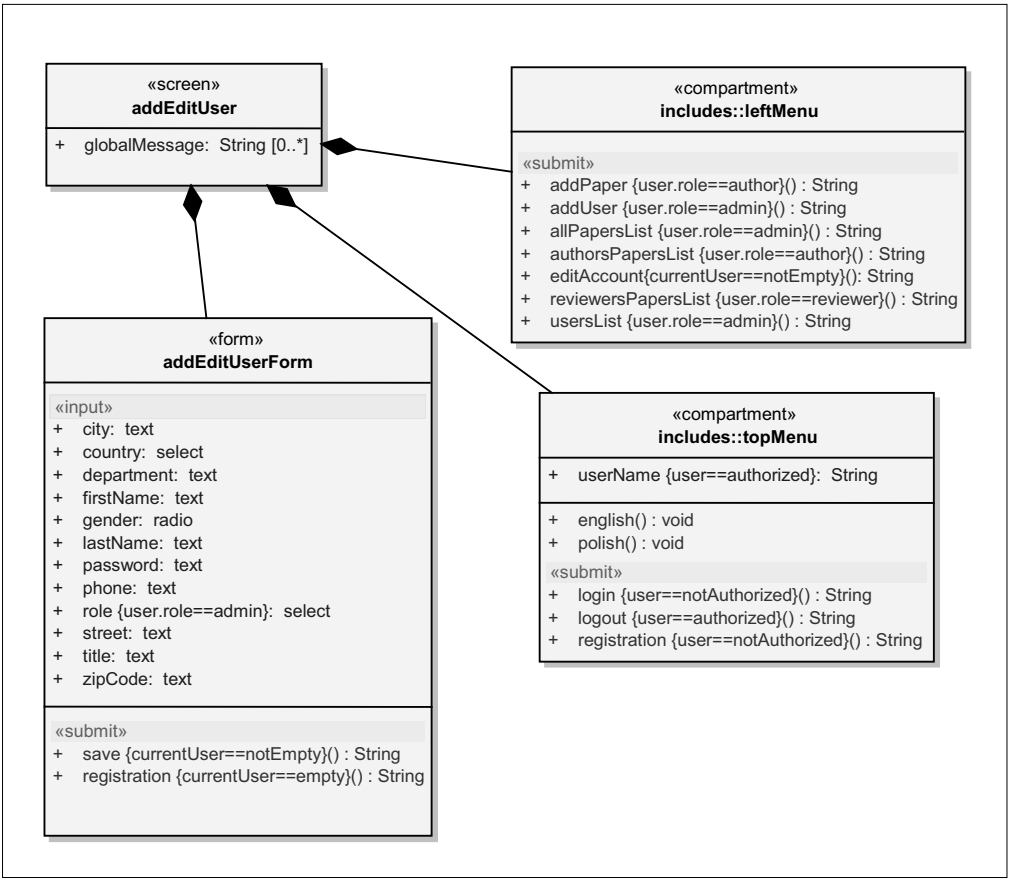
Nazwa operacji powinna być powiązana z nazwą metody służącej do obsługi zdarzenia wygenerowanego przez operację lub nazwą powiązanego z operacją pola (np. gdy operacja dotyczy walidacji danych wprowadzanych do pola wejściowego po stronie klienta). Widoczność elementu określa, czy akcja ma być podjęta po stronie klienta (-), czy po stronie serwera (+). Właściwości dodatkowo charakteryzujące operacje zamieszczane są w opcjonalnych nawiasach klamrowych.

Na rysunku 2.2 zaprezentowano model ekranu realizującego dwa przypadki użycia systemu ISRP — rejestrację nowego użytkownika i edycję danych użytkownika uwierzytelnionego. Obie funkcjonalności systemu są dość podobne i operują na jednym zbiorze danych, więc do ich obsługi zamodelowany został jeden ekran. Ekran składa się z trzech części: nagłówka (`topMenu`) zawierającego przyciski nawigacji, menu dla uwierzytelnionych użytkowników (`leftMenu`) i formularza danych użytkownika. W przypadku rejestracji nowego użytkownika elementy menu *leftMenu* oraz przycisk zapisywania formularza nie są widoczne, podobnie jak przyciski nawigacji (patrz rysunek 2.4). Z kolei w przypadku korzystania z ekranu przez administratora wszystkie jego elementy są widoczne.

2.1.3. Model nawigacji

W technice scenopisów diagramy nawigacji nie koncentrują się na konkretnym przepływie zdarzeń, ale pokazują związki pomiędzy ekranami. Model nawigacji zawiera graficzną ilustrację przepływów nawigacji dla wszystkich przypadków użycia. Przepływy nawigacji dla poszczególnych przypadków mogą być prezentowane na oddzielnych diagramach bądź jednym diagramie zbiorczym, jak pokazano na rysunku 2.3. Jak można zauważyć, przedstawiony diagram nawigacji jest bardziej szczegółowy niż opisana w rozdziale 1. graficzna reprezentacja reguł nawigacji w Eclipse WTP. Niemniej podczas tworzenia diagramu można wykorzystać gotowe obiekty ekranów, stanowiące zawartość utworzonego wcześniej modelu ekranów interfejsu użytkownika.

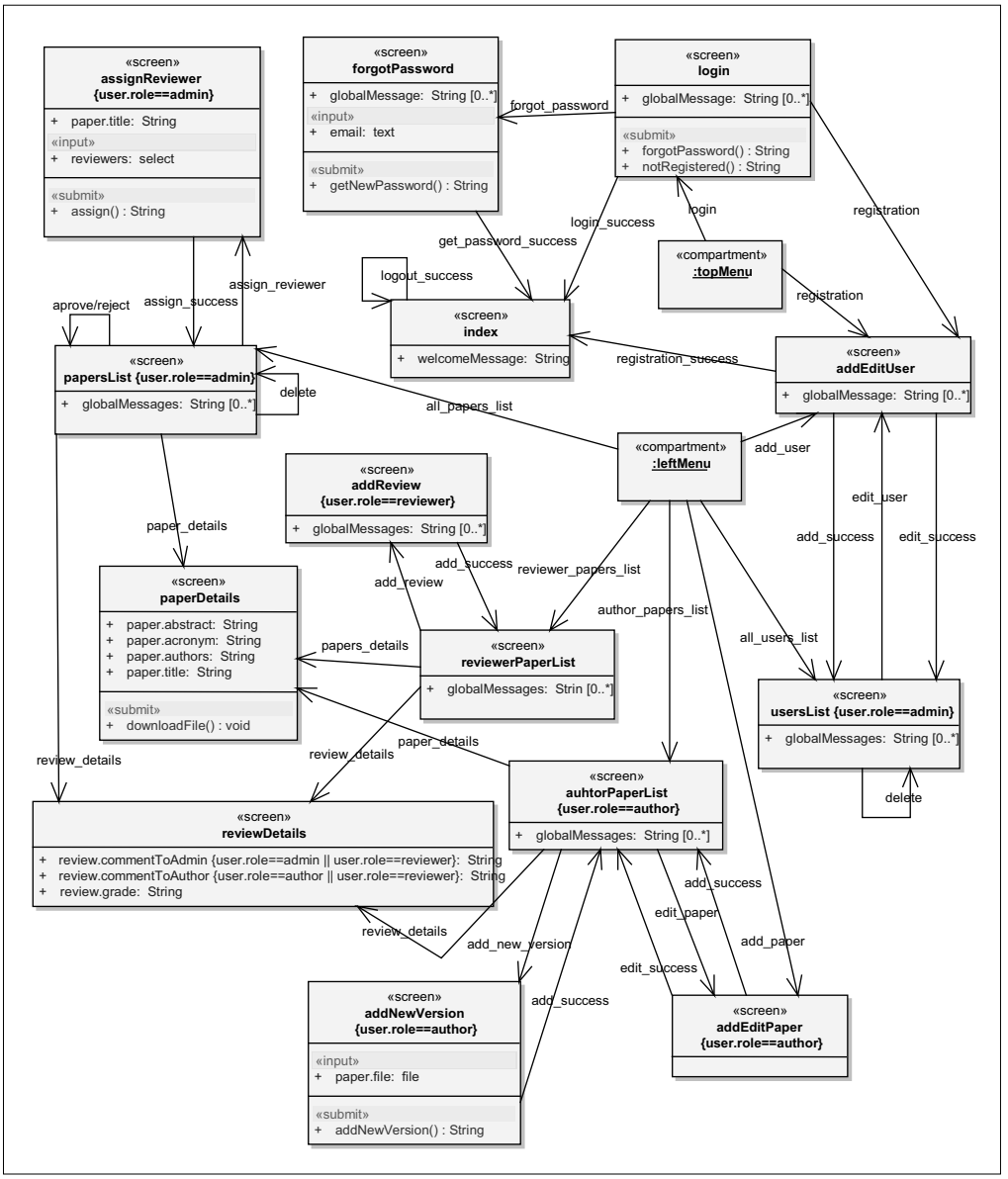
Występujące na diagramie asocjacje skierowane wskazują kierunek przepływu sterowania pomiędzy dwoma ekranami. W szczególnych przypadkach asocjacje mogą być wyprowadzane do formularza (`<<form>>`), ale zawsze miejscem docelowym jest ekran. Każda asocjacja powinna posiadać powiązaną z docelowym ekranem unikatową nazwę, która będzie mogła zostać wykorzystana podczas definiowania reguł nawigacji JSF (jako wartość elementu `<from-outcome>`).



Rysunek 2.2. *Model ekranu rejestracji i edycji danych użytkownika*

2.1.4. Prototypy ekranów

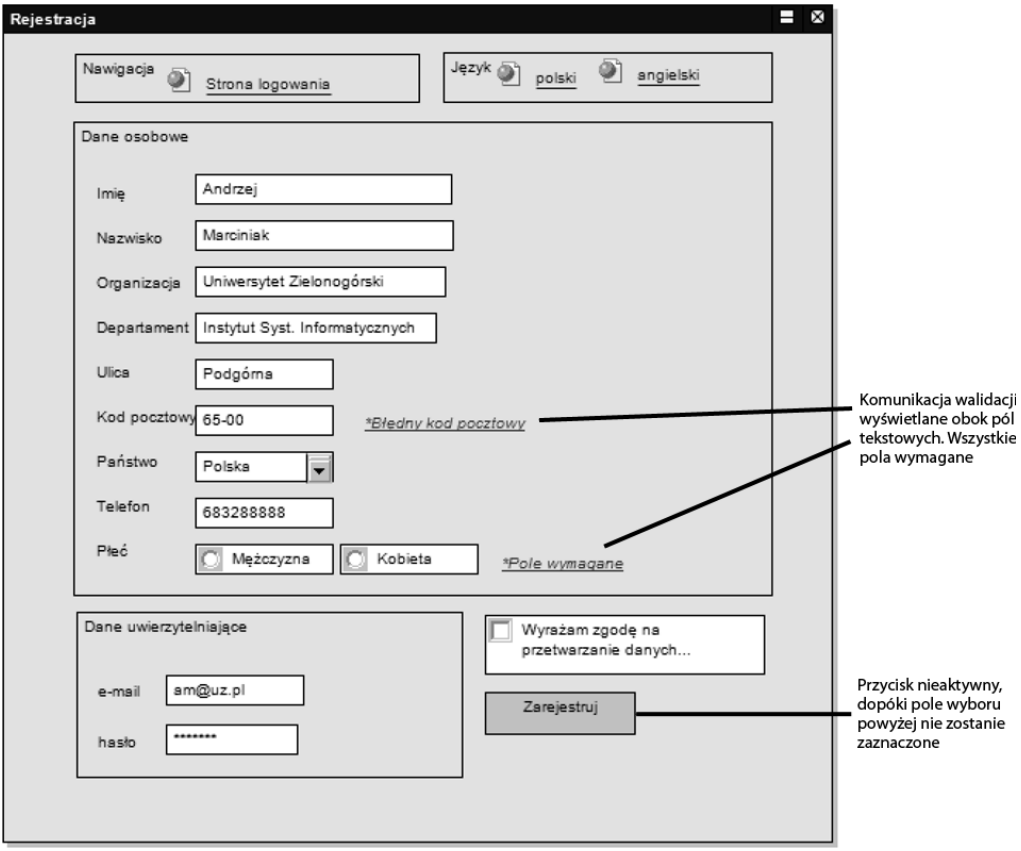
Zrozumiały, ergonomiczny i przejrzysty interfejs użytkownika jest jednym z kluczowych czynników decydujących o sukcesie aplikacji WWW. Konsultowanie wyglądu interfejsu przez potencjalnych użytkowników systemu w fazie przedimplementacyjnej lub w trakcie implementacji pozwala ten sukces osiągnąć. Celem przeanalizowania preferencji użytkowników w zakresie funkcjonalności interfejsu stosuje się tzw. prototypowanie, polegające na wytwarzaniu makiet ekranów (ang. *mock-up*). Pojęcie makiety jest mało precyzyjne i charakteryzuje wszelkie projekty ekranów, począwszy od odręcznych szkiców, poprzez *projekty szkieletowe ekranu* (ang. *wireframes*), aż po gotowe interaktywne prototypy ekranów posiadające bogatą szatę graficzną i umożliwiające testowanie ekranu na fałszywych danych. Projekty szkieletowe nie uwzględniają szaty graficznej, a ich celem jest przedstawienie ogólnej funkcjonalności poszczególnych ekranów oraz sposobu prezentacji danych biznesowych.



Rysunek 2.3. *Diagram nawigacji dla aplikacji ISRP*

W technice scenopisów generalnie zaleca się stosowanie projektów szkieletowych, które mogą być wytwarzane zarówno odręcznie, z wykorzystaniem narzędzi graficznych (dość popularny jest pakiet MS PowerPoint), jak i za pomocą specjalnych narzędzi dostępnych w pakietach do modelowania, np. Microsoft Visio Toolkit for Wireframe, MockupScreens czy Screen Architect for Sparx Enterprise Architect. W środowisku Eclipse warto rozważyć wykorzystanie edytora stron WWW pakietu WTP — utworzone w ten sposób projekty mogą być później stosowane podczas implementacji.

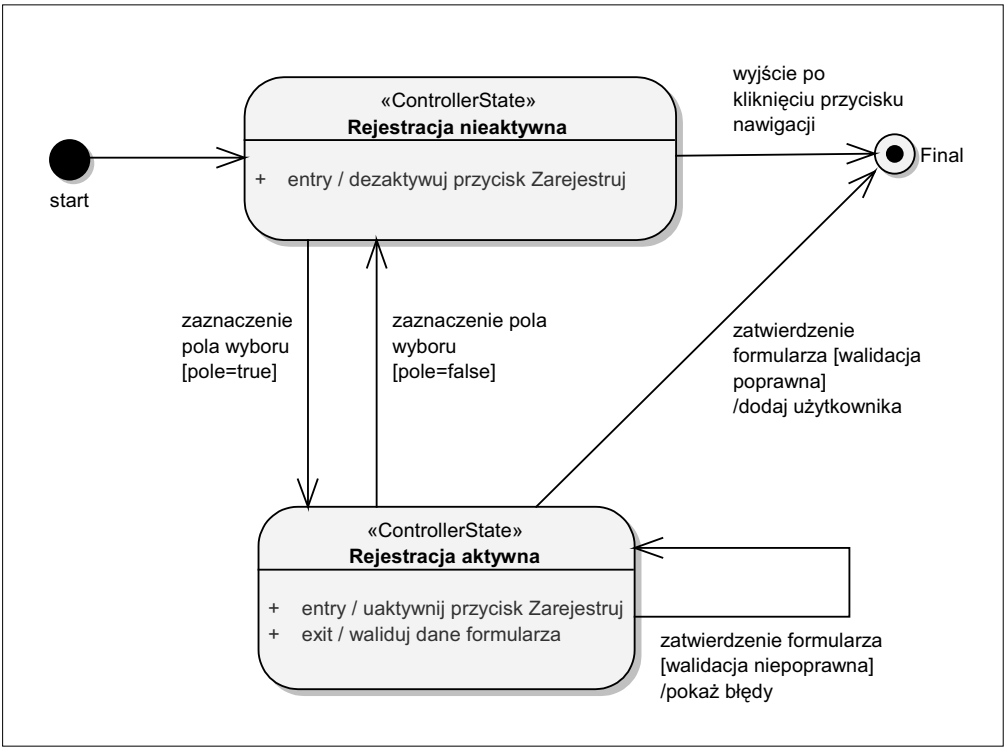
Przykład projektu szkieletowego dla ekranu rejestracji użytkownika w systemie ISRP został pokazany na rysunku 2.4. Projekt stanowi rozwinięcie modelu ekranu zamieszczonego na rysunku 2.2 zarówno w zakresie specyfikacji kontrolek realizujących elementy funkcjonalne, jak i rozmieszczenia elementów wizualnych, mających ułatwić korzystanie ze strony.



Rysunek 2.4. *Projekt szkieletowy ekranu rejestracji aplikacji ISRP*

2.1.5. Diagram maszyny stanowej

W technice scenopisów diagram maszyny stanowej służy do opisania zachowania ekranu w trakcie realizacji przypadku użycia. Diagram modeluje stan znajdującego się po stronie serwera kontrolera ekranu oraz jego zmiany następujące w wyniku interakcji użytkownika z ekranem. Na rysunku 2.5 zamieszczony został przykład diagramu maszyny stanowej, ilustrujący zmiany stanów kontrolera ekranu rejestracji (przedstawionego na rysunku 2.4).



Rysunek 2.5. *Diagram maszyny stanowej dla ekranu rejestracji*

Stan początkowy reprezentuje stan kontrolera po utworzeniu jego instancji i jest oznaczony przez strzałkę wychodzącą od elementu *start*. Każde przejście (tranzycja) na diagramie definiuje relację między dwoma stanami kontrolera, wskazującą, że kontroler znajdujący się w pierwszym stanie wykona pewne akcje i przejdzie do drugiego stanu, ilekroć zajdzie określone zdarzenie i będą spełnione określone warunki. Opis przejścia składa się z trzech opcjonalnych elementów: *wyzwalacza* (ang. *trigger*), *dozoru* (ang. *guard*) i *aktywności* (ang. *activity*), które opisywane są według następującej składni: `wyzwalacz [dozór] /aktywność`. Wyzwalaczem jest zazwyczaj pojedyncze zdarzenie, które może spowodować przejście i zmianę stanu. Dozór określa warunek logiczny, jaki musi być spełniony, aby przejście zostało wykonane; warunek ten jest obliczany w momencie pojawienia się wyzwalacza. Aktywność definiuje operację wykonywaną w momencie przejścia ze stanu do stanu; nawet jeżeli akcja przejścia jest złożona z wielu akcji elementarnych, jest ona wykonywana niepodzielnie.

Przejścia są oznaczane za pomocą strzałek łączących dwa stany kontrolera i mogą być postrzegane jako zmiany na ekranie, które nie powodują uaktywnienia reguł nawigacji skutkujących wyświetleniem innego ekranu. Podobnie jak w zwykłym diagramie maszyny stanowej, również tutaj mogą występować szczególne przypadki przejść, których wykonanie nie prowadzi do zmiany danego stanu, tj. przejścia wewnętrzne lub zwrotne. W przypadku przejścia zwrotnego są wykonywane czynności wejściowe oraz wyjściowe danego stanu, w przypadku przejść wewnętrznych są one pomijane.

Przejścia wewnętrzne nie mają odrębnego oznaczenia graficznego, lecz są specyfikowane w obszarze stanu, obok definicji następujących czynności:

- `entry` — czynności wykonywane w momencie, gdy obiekt przyjmuje dany stan;
- `do` — czynności wykonywane nieprzerwanie w czasie, gdy obiekt przebywa w danym stanie;
- `exit` — czynności wykonywane w momencie opuszczenia stanu.

Przejścia wewnętrzne lub zwrotne występują w sytuacjach, gdy kompozycja ekranu zmienia się tylko nieznacznie lub wcale. Stan końcowy (ang. *final*) wskazuje, że maszyna stanów jest kompletna, a obiekt kontrolera oraz wszystkie przechowywane w nim zmienne mogą zostać usunięte.

2.2. Warstwa biznesowa

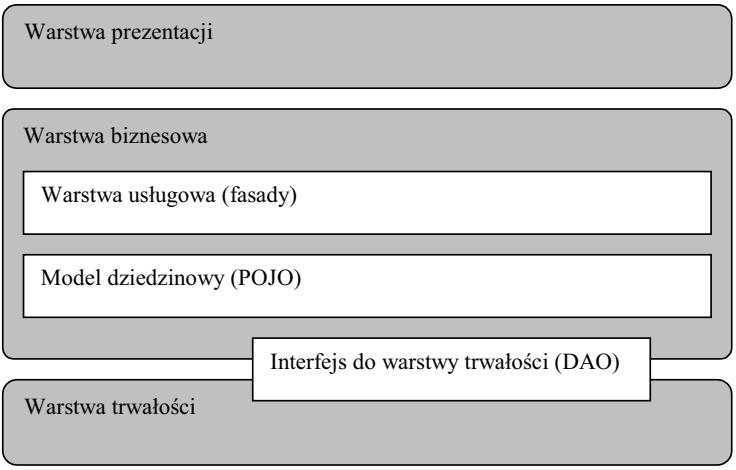
W rozdziale 1.2.4, „Implementacja MVC w JSF", przedstawiono ogólny schemat implementacji wzorca MVC w aplikacjach opartych na JSF oraz korzyści płynące z izolowania warstwy biznesowej i prezentacji. W tym punkcie zostaną omówione dobre praktyki projektowe, rozwiązania i technologie związane z tworzeniem samej warstwy biznesowej.

2.2.1. Rola i elementy warstwy biznesowej

Pojęcie **warstwy biznesowej** dość sugestywnie określa rolę tej warstwy, w której modelowane są obiekty i reguły biznesowe. Stanowi ona zazwyczaj rdzeń aplikacji i jest najbardziej rozległą jej warstwą, z jednej strony świadcząc usługi dla warstwy prezentacji, z drugiej strony korzystając z zewnętrznych źródeł danych, takich jak bazy danych, korporacyjne systemy informacyjne (ang. *Enterprise Information Systems — EIS*) czy zewnętrzne systemy informatyczne. Separowanie warstwy biznesu od prezentacji ułatwia implementację reguł biznesowych i umożliwia użycie obiektów biznesowych w dowolnym środowisku uruchomieniowym (strona 29 „Korzyści wynikające z izolowania warstw widoku i logiki biznesowej"). Ze względu na rozległość warstwy biznesowej dobrze jest podzielić ją na podwarstwy, które są od siebie logicznie oddzielone i posiadają ściśle zdefiniowane zakresy odpowiedzialności, tj. model dziedzinowy, warstwę usług i interfejs do warstwy trwałości danych (patrz rysunek 2.6), omówione w kolejnych punktach.

Najpopularniejsze podejścia do tworzenia warstwy biznesowej w aplikacjach WWW oparte są na wykorzystaniu:

- zarządzanych przez kontener komponentów,
- zwykłych obiektów Javy (ang. *Plain Old Java Objects — POJO*).



Rysunek 2.6. *Warstwowy model architektury wykorzystany w aplikacji ISRP*

Programowanie komponentowe jest naturalnym rozszerzeniem programowania zorientowanego obiektowo — nie posiada ono zupełnie nowych atrybutów w stosunku do programowania obiektowego, a jedynie konsekwentnie i w pełni wykorzystuje cechy obiektowości. Komponenty oferują hermetycznie opakowane, wysokopoziomowe funkcje realizujące logikę biznesową. Zarządzaniem komponentami zajmuje się kontener, który kontroluje różne aspekty ich życia, m.in. tworzenie instancji, wprowadzanie zależności pomiędzy nimi czy też zarządzanie cyklem ich życia. Kontener pozwala na ograniczenie zadań stojących przed komponentami do realizacji jedynie funkcji biznesowych, dzięki czemu są one łatwe do zrozumienia, mają czytelne interfejsy i mogą być rozproszone w sieci. Programowanie komponentowe boryka się jednak z wieloma problemami implementacyjnymi, przede wszystkim z dużą różnorodnością protokołów komponentowych i języków interfejsów, problemami ze wzajemnym porozumiewaniem się komponentów czy złożonością infrastruktur środowisk uruchomieniowych. Popularnymi technologiami komponentów są Enterprise Java Beans czy Spring.

Do implementacji warstwy biznesowej w zawartych w książce przykładach wykorzystywane jest podejście oparte na klasach POJO. Zaletą tego podejścia jest brak zależności od technologii EJB czy Java EE oraz możliwość użycia i testowania jej kodu bez potrzeby korzystania ze środowiska uruchomieniowego serwera.

Klasy POJO i komponenty JavaBean

Termin *Plain Old Java Objects* określa obiekty zwykłych klas Javy, które nie są ograniczone żadnymi specjalnymi restrykcjami i nie muszą implementować żadnych specjalnych zachowań w odróżnieniu od np. komponentów EJB 2.x. Klasy POJO nie powinny dziedziczyć po wcześniej predefiniowanych klasach, implementować predefiniowanych interfejsów czy zawierać predefiniowanych adnotacji, jeśli te nie są związane z ich podstawową funkcjonalnością (choć nawet tych wymagań nie należy traktować zbyt rygorystycznie). Pojęcie POJO zostało wprowadzone przez Martina Fowlera, Rebeccę Parsons i Josha McKenziego w 2000 roku, którzy poprzez wprowadzenie atrakcyjnie brzmiącej nazwy dla zwykłych obiektów Javy chcieli wyróżnić ideę wykorzystania prostych technik projektowych w programowaniu obiektowym.

Ponieważ w aplikacjach korporacyjnych Java EE (J2EE) wymagane są dodatkowe usługi, takie jak zarządzanie transakcjami, utrzymywanie bezpieczeństwa czy trwałości danych (w alternatywnym podejściu usługi te zapewniają kontenery EJB), powstały specjalne lekkie szkielety programistyczne (ang. *lightweight frameworks*), udostępniające wybrane usługi obiektom POJO. Przykładami lekkich szkieletów są Spring, Hibernate czy iBatis. Pozytywne doświadczenia ze stosowania lekkich szkieletów i klas POJO skłoniły również projektantów technologii EJB do gruntownej przebudowy ich rozwiązania. Począwszy od wersji EJB 3.0, komponenty EJB są w zasadzie klasami POJO, które zawierają adnotacje specyfikujące dodatkowe właściwości obiektów tych klas (poza kontenerem EJB adnotacje nie są interpretowane, więc można stwierdzić, że EJB 3.0 to POJO).

Komponenty JavaBean są to napisane w odpowiedniej konwencji klasy POJO, będące niezależnymi modułami wielokrotnego użytku w programach Javy. Narzędzia do tworzenia oprogramowania są w stanie łączyć i wykorzystywać komponenty JavaBeans oraz manipulować nimi dzięki przyjęciu odpowiednich konwencji w nazywaniu metod, konstrukcji klasy i jej zachowania, a mianowicie:

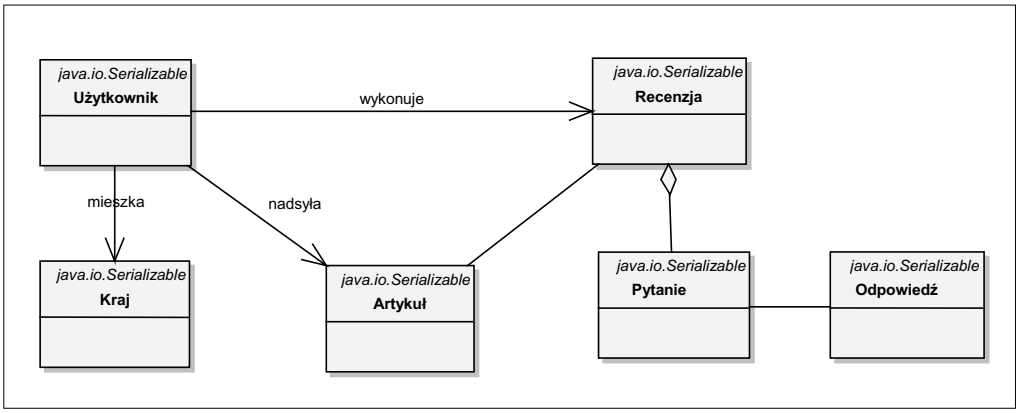
- klasa musi posiadać publiczny i bezparametrowy konstruktor;
- hermetyzowane właściwości klasy muszą być dostępne do odczytu i modyfikacji za pomocą specjalnych metod publicznych (tzw. *getters & setters*), które muszą mieć odpowiednio skonstruowane nazwy, zawierające przedrostki `get-`, `set-` i opcjonalnie `is-` dla typu `boolean`, np.: `public int getTemperature()`, `public void setTemperature(int t)`, `public boolean isFinished()` — dla właściwości `temperature` i `finished`;
- klasa powinna być serializowalna (implementować interfejs `Serializable` lub `Externalizable`);
- klasa nie powinna zawierać metod obsługi zdarzeń.

Przykładowymi komponentami JavaBean są komponenty wspierające.

2.2.1.1. Model dziedzinowy

Na podstawie analizy przypadków użycia możemy zidentyfikować podstawowe klasy konceptualne (pojęcia, rzeczy bądź obiekty) występujące w projektowanym systemie. Model konceptualny stanowi punkt wyjściowy generowania obiektowego *modelu dziedzinowego* oraz *modelu relacyjnego* (fizycznego). Diagram konceptualny na rysunku 2.7 przedstawia podstawowe pojęcia występujące w systemie ISRP.

Dokładne odwzorowanie obiektowe koncepcji biznesowych występujących w modelowanym środowisku zapewnia tzw. model drobnoziarnisty (ang. *fine-grained model*), oparty na wykorzystaniu wielu prostych obiektów POJO, reprezentujących istotne koncepcje biznesowe. Modele drobnoziarniste znakomicie spełniają swoje funkcje, o ile nie występuje potrzeba rozproszenia obiektów na różnych serwerach. W modelu drobnoziarnistym występuje mnóstwo wzajemnych odwołań pomiędzy obiektami, co stanowi problem w systemach rozproszonych, gdyż zdalne wywołania są dość kosztowne.



Rysunek 2.7. *Diagram konceptualny systemu ISRP*

W takich systemach podejście komponentowe może być przydatniejsze, a kontenery Java EE mogą dodatkowo zapewnić bezpieczeństwo, transakcyjność czy połączenia w systemach heterogenicznych.

Model dziedzinowy aplikacji ISRP zaimplementowany jest w postaci klas POJO. Przykład obiektu biznesowego dla koncepcji użytkownika został przedstawiony na listingu 2.1.

Listing 2.1. *Kod klasy User*

```
package isrp.hibernate.model.businessobject;
import java.util.Date;

public class User implements java.io.Serializable {
   private Integer id;
   private int countryId;
   private String title;
   private String firstName;
   private String lastName;
   private String organization;
   private String department;
   private String street;
   private String zipCode;
   private String city;
   private String email;
   private String password;
   private String phone;
   private boolean gender;
   private Date entryDate;
   private Byte role;

   public User() { //konstruktor na potrzeby serializacji
   }


   public User(int countryId, String title, String firstName, String lastName,
         String organization, String department, String street,
         String zipCode, String city, String email, String password,
         String phone, boolean gender, Date entryDate, Byte role) {
```

```
        this.countryId = countryId;
        this.title = title;
        this.firstName = firstName;
        this.lastName = lastName;
        this.organization = organization;
        this.department = department;
        this.street = street;
        this.zipCode = zipCode;
        this.city = city;
        this.email = email;
        this.password = password;
        this.phone = phone;
        this.gender = gender;
        this.entryDate = entryDate;
        this.role = role;
    }

    public Integer getId() {return this.id;}

    public void setId(Integer id) {this.id = id;}

/* ...Pozostałe gettery i settery ...*/
}
```

Uwaga

Podejście polegające na wyprowadzeniu implementacji logiki biznesowej poza model dziedzinowy i obiekty reprezentujące dane określane jest w literaturze mianem *anemicznego modelu dziedzinowego*. Wprowadził je Martin Fowler, który traktuje takie rozwiązanie jako antywzorzec. Najważniejszym zarzutem, jaki stawia Fowler takiemu modelowi, jest naruszenie zasad OOP w zakresie enkapsulacji i hermetyzacji kodu — przetwarzaniem stanu obiektów dziedzinowych czy też ich walidacją zajmują się zewnętrzne klasy. Inne istotne mankamenty to zmniejszenie czytelności kodu oraz zwiększenie możliwości występowania duplikacji w kodzie. Mimo tych wad (wyjaśnionych dokładnie w artykule pod adresem *http://www.martinfowler.com/bliki/AnemicDomainModel.html*) rozwiązanie to jest często i chętnie stosowane, zwłaszcza gdy obiektowy model dziedzinowy jest generowany automatycznie, za pomocą narzędzi ORM, na podstawie istniejącego wcześniej modelu fizycznego.

2.2.1.2. Warstwa usługowa

Bardzo ważne na etapie projektowania jest rozróżnienie dwóch aspektów funkcjonalności systemu występujących w warstwie biznesowej: *modelu dziedzinowego* i *logiki biznesowej*. Model dziedzinowy reprezentuje koncepcje biznesowe i ich zachowanie, podczas gdy logika biznesowa reprezentuje procesy i przepływy sterowania odpowiadające scenariuszom przypadków użycia. Oddzielenie logiki biznesowej od modelu dziedzinowego jest możliwe poprzez wprowadzenie warstwy usługowej (ang. *service layer*) odpowiedzialnej za:

- zdefiniowanie uproszczonego interfejsu dla klientów modelu dziedzinowego,
- implementację złożonych obiektów usług zdefiniowanych przez wyżej wymieniony interfejs.

Obiekty usług mogą zapewniać fasady dla modelu dziedzinowego (strona 83 „Wzorzec fasady”) oraz implementować procesy biznesowe.

Na przykład na diagramie przypadków użycia systemu ISRP występuje przypadek ***Przypisz recenzenta do pracy***, który reprezentuje złożony proces biznesowy. W ramach tego procesu zachodzi następująca sekwencja czynności: redaktor z uprawnieniami administratora wybiera z listy recenzenta, do którego automatycznie wysyłane jest zawiadomienie (pocztą elektroniczną — SMTP), i zapisywany jest stan procesu w bazie danych. Zamiast wywoływać wiele metod dla obiektów modelu dziedzinowego w warstwie prezentacji, możemy wywołać jedną metodę `void assignReviewer(Paper paper,int reviewerId)`, zdefiniowaną w fasadzie `PaperService`, której kod został zamieszczony na listingu 2.2. Zgodnie z dobrą praktyką programowania obiektowego, polegającą na przedkładaniu interfejsów nad implementacje, wszystkie metody wymagane dla realizacji procesów biznesowych związanych z przetwarzaniem artykułów konferencyjnych są określone w interfejsie `PaperService`.

Listing 2.2. *Kod źródłowy interfejsu PaperService*

```
package isrp.model.service;
import java.util.List;
import org.apache.myfaces.custom.fileupload.UploadedFile;
import isrp.hibernate.model.businessobject.Paper;
import isrp.hibernate.model.businessobject.User;
import isrp.hibernate.model.util.email.EmailConfiguration;

public interface PaperService {

   boolean addPaperNewVersion(Paper paper,User author,UploadedFile
   ➥uploadedFile,String uploadPath,EmailConfiguration emailConfiguration);
   void update(Paper paper);
   void assignReviewer(Paper paper,int reviewerId,EmailConfiguration
   ➥emailConfiguration);
   boolean addPaper(Paper paper,User author,UploadedFile uploadedFile,String
   ➥uploadPath,EmailConfiguration emailConfiguration);
   Paper findById(int id);
   void delete(Paper paper);
   void approvePaper(Paper paper,EmailConfiguration emailConfiguration);
   void rejectPaper(Paper paper,EmailConfiguration emailConfiguration);
   List<Paper> getReviewerPapers(int reviewerId, String sortColumn);
   List<Paper> getAuthorPapers(int userId, String sortColumn);
   List<Paper> getAllPapers(String sortColumn);
   List<Paper> getAllPapers(String sortColumn,Integer userId, Integer role);
}
```

Implementacja metod interfejsu `PaperService` znajduje się w klasie `PaperServiceImpl`. Ponieważ rozmiar całego kodu klasy jest obszerny, na listingu 2.3 zamieszczony został wyłącznie jego fragment, zawierający deklarację klasy oraz implementację metody odpowiedzialnej za przypisanie recenzenta i wysłanie powiadomienia pocztą elektroniczną.

Listing 2.3. *Implementacja interfejsu PaperService*

```
public class PaperServiceImpl implements PaperService{

   public PaperServiceImpl() {}// bezargumentowy konstruktor

   public void assignReviewer(Paper paper, int reviewerId,
      EmailConfiguration emailConfiguration) {
```

```
        /* pobranie obiektów dostępowych do tabel — patrz listing 2.11*/
        PaperDao pDao = DaoFactory.getInstance().getPaperDao();
        UserDao uDao = DaoFactory.getInstance().getUserDao();
        pDao.saveOrUpdate(paper);
        User author=(User) uDao.findById(paper.getAuthorId());//pobranie obiektu autora
        User reviewer=(User) uDao.findById(reviewerId);//pobranie obiektu recenzenta
        /* wysłanie potwierdzenia pocztą elektroniczną*/
        EmailSender eSender = new EmailSender(emailConfiguration);
        eSender.sendEMail(EmailMessagesUtil
            .assignReviewerMessageToReviewer(reviewer, paper));
    }
/* ... implementacje pozostałych metod interfejsu PaperService*/
}
```

Wzorzec fasady

Wzorzec fasady jest jednym z podstawowych i jednocześnie najprostszych strukturalnych wzorców projektowych. Zapewnia on dostęp do złożonych systemów, prezentując uproszczony lub uporządkowany interfejs programistyczny. Wzorzec fasady pozwala:

- ukryć złożoność tworzonego systemu poprzez dostarczenie prostego interfejsu API,
- uprościć interfejs istniejącego systemu poprzez stworzenie nakładki, która dostarcza nowy interfejs API.

Obiekt fasady jest prostym obiektem pośredniczącym pomiędzy obiektem klasy klienta żądającym określonej funkcjonalności a klasami dostarczającymi elementów do uzyskania tej funkcjonalności.

2.2.1.3. Lokalizatory usług

Usługi biznesowe powinny być łatwo dostępne dla wszelkiego typu klientów, niezależnie od tego, czy są one klientami lokalnymi, czy też zdalnymi. Gdy w warstwie biznesowej występuje wiele fasad, korzystanie z usług biznesowych możemy ułatwić poprzez zastosowanie uproszczonej wersji wzorca lokalizatora usług (częściowo opisanego w rozdziale 1., w ramce „Wzorzec delegata biznesowego”). Obiekt lokalizatora tworzy jeden punkt dostępowy do kodu warstwy biznesowej, jednocześnie zmniejszając zależność klienta od implementacji mechanizmów wyszukiwania usług. Zazwyczaj lokalizator jest implementowany jako pojedynczy obiekt (singleton), chyba że stosowany jest w środowisku rozproszonym.

Na przykład w systemie ISRP istnieją zdefiniowane odrębne fasady dla usług związanych z przetwarzaniem artykułów, użytkowników, recenzentów, formularzy z pytaniami oraz danych słownikowych. Przedstawiony na listingu 2.4 interfejs `ServiceLocator` zawiera deklaracje metod, które udostępniają obiekty implementujące interfejsy tych fasad. Operowanie na interfejsach pozwala na uniezależnienie się od jednej konkretnej implementacji zestawu fasad, dzięki czemu łatwo możemy dostarczyć implementacje zoptymalizowane pod kątem wydajności czy bezpieczeństwa.

Listing 2.4. *Kod źródłowy interfejsu lokalizatora usług*

```
package isrp.model.service.factory;
import isrp.model.service.CountryService;
import isrp.model.service.PaperService;
import isrp.model.service.QuestionService;
import isrp.model.service.ReviewService;
import isrp.model.service.UserService;

public interface ServiceLocator {
   PaperService getPaperService();
   UserService getUserService();
   ReviewService getReviewService();
   CountryService getCountryService();
   QuestionService getQuestionService();
}
```

Na listingu 2.5 przedstawiono kod klasy lokalizatora `ServiceLocatorImpl`, definiującego metody dostępowe zwracające obiekty fasad. Aby tak prosty lokalizator wykonywał swoje zadanie, należy właściwie zainicjować i przechować jego instancję w kontenerze, co jest opisane w następnym punkcie.

Listing 2.5. *Kod źródłowy lokalizatora usług ServiceLocatorImpl*

```
package isrp.model.service.factory;
import org.apache.commons.logging.*;
import isrp.model.service.*;
import isrp.model.service.impl.*;

public class ServiceLocatorImpl implements ServiceLocator{

   private PaperService paperService;
   private UserService userService;
   private ReviewService reviewService;
   private CountryService countryService;
   private QuestionService questionService;

   private Log log = LogFactory.getLog(this.getClass());

   public ServiceLocatorImpl() {
      this.paperService = new PaperServiceImpl();
      this.userService = new UserServiceImpl();
      this.reviewService = new ReviewServiceImpl();
      this.countryService = new CountryServiceImpl();
      this.questionService = new QuestionServiceImpl();
      this.log.info("Service locator bean is initialized");
   }
   public PaperService getPaperService() {return paperService;}
   public UserService getUserService() {return userService;}
   public ReviewService getReviewService() {return reviewService;}
   public CountryService getCountryService() {return countryService;}
   public QuestionService getQuestionService() {return questionService;}
}
```

2.2.1.4. Integracja warstwy usługowej z warstwą prezentacji JSF

Wygodnym sposobem integracji warstwy usługowej z warstwą prezentacji aplikacji JSF jest wykorzystanie mechanizmu komponentów zarządzanych. Na przykład obiekt lokalizatora usług może zostać zarejestrowany w aplikacji jako komponent zarządzany o zasięgu aplikacji. Będzie on wówczas automatycznie tworzony w momencie pierwszego odwołania się do niego, a następnie przechowywany w kontekście aplikacji JSF aż do końca jej działania.

Deklaratywny sposób rejestracji obiektu lokalizatora w pliku *faces-config.xml* został zaprezentowany na listingu 2.6. Pokazane na przykładzie rozwiązanie umożliwia dostęp do obiektu lokalizatora poprzez wykorzystanie jego identyfikatora — `service ↪LocatorImpl` lub właściwości komponentu użytkowego `utilBean` (opis komponentów zarządzanych znajduje się w rozdziale 3.). W omawianym przykładzie lokalizatorem jest obiekt zaprezentowanej na listingu 2.5 klasy `ServiceLocatorImpl`, ale dzięki stosowaniu interfejsów fasad może on zostać łatwo zamieniony na inny obiekt będący lokalizatorem, do czego wystarczy zmiana wartości elementu `<managed-bean-class>`. Może się to okazać przydatne, np. gdy będziemy chcieli zapewnić buforowanie przesyłanych danych czy zdalny dostęp do usług.

Listing 2.6. *Integracja warstwy usługowej w pliku faces-config.xml*

```
<managed-bean>
   <description> Komponent lokalizatora </description>
   <managed-bean-name>serviceLocatorImpl</managed-bean-name>
   <managed-bean-class>isrp.model.service.factory.ServiceLocatorImpl</managed-bean-class>
   <managed-bean-scope>application</managed-bean-scope>
</managed-bean>
<managed-bean>
   <managed-bean-name>utilBean</managed-bean-name>
   <managed-bean-class>isrp.viewbeans.UtilBean</managed-bean-class>
   <managed-bean-scope>application</managed-bean-scope>
   <managed-property>
      <property-name>serviceLocator</property-name>
      <value>#{serviceLocatorImpl}</value>
   </managed-property>
    <!-- pozostałe właściwości -->
</managed-bean>
```

2.2.2. Interfejs do warstwy trwałości danych

Choć współczesne aplikacje WWW mogą korzystać z wielu możliwości w zakresie utrwalania danych, to wciąż najpopularniejszą z nich jest stosowanie relacyjnych baz danych. Interfejs do warstwy trwałości powinien całkowicie oddzielać szczegóły technologii relacyjnych od warstwy biznesu, jednocześnie zapewniając następujące operacje:

- funkcje CRUD, tj. ang. *Create, Retrieve, Update, Delete* (utwórz, odczytaj, modyfikuj i usuń obiekty w pamięci trwałej),
- tworzenie i uruchamianie zapytań do bazy danych,
- zarządzanie połączeniami, transakcjami, buforowaniem i wydajnością.

Korzystając z relacyjnych baz danych, należy zapewnić odwzorowania obiektów Javy na tabele, kolumny i rekordy, podobnie jak mapowanie wzajemnych relacji pomiędzy obiektami na powiązania tabel i rekordów. W tym celu można zastosować:

- interfejs JDBC API,
- komponenty encji dostępne w technologii Java Persistence API,
- technologie odwzorowań obiektowo-relacyjnych (ORM).

W zasadzie dwa ostatnie rozwiązania są dość podobne — JPA było wzorowane na szkieletach ORM Hibernate i TopLink. Technologie ORM pozwalają zarówno utrwalać obiekty POJO w bazie danych, jak i generować model dziedzinowy na podstawie schematu bazy danych (tzw. *reverse engineering*). Dzięki temu stanowią wygodne rozwiązanie, które możemy wykorzystać niezależnie od scenariusza, według jakiego wytwarzana jest warstwa trwałości, a mianowicie gdy:

- istnieje kod Javy i należy zapewnić przechowywanie obiektów Javy w bazie danych,
- istnieje zaprojektowana (lub odziedziczona z innego systemu) baza danych i należy napisać kod aplikacji. Model dziedzinowy w Javie jest dopasowywany do schematu bazy danych,
- istnieje zarówno baza danych, jak i część kodu Javy, które należy powiązać.

W dalszych punktach przybliżone zostaną najważniejsze informacje dotyczące wykorzystania biblioteki Hibernate w celu zapewnienia dostępu do danych.

2.2.2.1. Interfejs warstwy danych oparty na Hibernate

Hibernate jest darmowym szkieletem programistycznym, służącym do realizacji warstwy dostępu do danych i zapewniającym translację danych pomiędzy relacyjną bazą danych a światem obiektowym. Użycie Hibernate powoduje, że z punktu widzenia programisty baza danych zawiera zwykłe obiekty Javy (POJO). Hibernate zapewnia połączenia z bazą danych i obsługę podstawowych operacji na nich z uwzględnieniem przenośności między różnymi dialektami SQL.

Zasada działania Hibernate opiera się na zastosowaniu plików konfiguracyjnych definiowanych w języku XML oraz użyciu mechanizmu refleksji do pobierania obiektów i właściwości klas. Plik konfiguracyjny o nazwie *hibernate.cfg.xml* zawiera globalne ustawienia parametrów szkieletu oraz połączenia do bazy danych. Przykład pliku konfiguracyjnego dla aplikacji ISRP został przedstawiony na listingu 2.7. Znaczenie poszczególnych elementów opisane jest w komentarzach do kodu.

Listing 2.7. *Plik konfiguracji Hibernate*

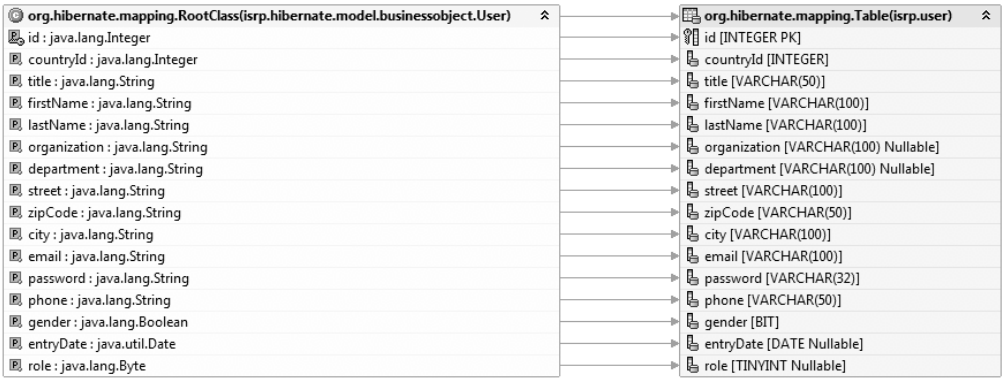
```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
      "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
      "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
   <session-factory>
```

```
        <!-- parametry sterownika i połączenia do bazy danych -->
        <property name="hibernate.connection.driver_class">com.mysql.
        ➥jdbc.Driver</property>
        <property name="hibernate.connection.url">jdbc:mysql:
        ➥//localhost/isrp</property>
        <property name="hibernate.connection.useUnicode">yes</property>
        <property name="hibernate.connection.characterEncoding">UTF-8</property>
        <property name="hibernate.connection.username">root</property>
        <property name="hibernate.dialect">org.hibernate.dialect.
         ➥MySQLDialect</property>
        <property name="current_session_context_class">thread</property>
        <!-- podgląd generowanych wyrażeń SQL na konsoli -->
        <property name="hibernate.show_sql">true</property>
        <!-- pliki mapowań obiektowo-relacyjnych -->
        <mapping resource="isrp/hibernate/model/businessobject/Answer.hbm.xml" />
        <mapping resource="isrp/hibernate/model/businessobject/User.hbm.xml" />
        <mapping resource="isrp/hibernate/model/businessobject/Country.hbm.xml" />
        <mapping resource="isrp/hibernate/model/businessobject/Paper.hbm.xml" />
        <mapping resource="isrp/hibernate/model/businessobject/Question.hbm.xml" />
        <mapping resource="isrp/hibernate/model/businessobject/Review.hbm.xml" />
        <mapping resource="isrp/hibernate/model/businessobject/
        ➥Answersquestions.hbm.xml" />
    </session-factory>
</hibernate-configuration>
```

Pliki mapowań Hibernate odpowiadają za translację danych znajdujących się w tabeli na obiekty Javy i odwrotnie. Oznaczane są przyrostkiem *.hbm.xml*, gdzie pierwszy element przyrostka jest skrótem od wyrażenia „HiBernate Mapping". Poniżej zamieszczono fragment pliku mapowania *User.hbm.xml*. Diagram pełnego mapowania przykładowej klasy `User` do tabeli `user`, wygenerowany przy użyciu narzędzia HibernateTools, zaprezentowano na rysunku 2.8.

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<!-- Generated 2009-10-11 21:28:51 by Hibernate Tools 3.2.5.Beta -->
<hibernate-mapping>
    <class name="isrp.hibernate.model.businessobject.User" table="user" catalog="isrp">
        <id name="id" type="java.lang.Integer">
            <column name="id" />
            <generator class="identity" />
        </id>
        <property name="countryId" type="int">
            <column name="countryId" not-null="true" />
        </property>
    </class>
</hibernate-mapping>
```

Instalacja Hibernate sprowadza się do pobrania i rozpakowania pakietów instalacyjnych, dostępnych na stronie *http://sourceforge.net/projects/hibernate/files/hibernate3/*. Po rozpakowaniu pliki bibliotek powinny zostać umieszczone w ścieżce przeszukiwania klas projektu. Najważniejsze klasy Hibernate znajdują się w pliku *hibernate3.jar*, a klasy pomocnicze w plikach bibliotek: *commons-collections.jar*, *commons-logging.jar*, *dom4j.jar*, *hsqldb.jar*, *jta.jar*, *c3p0.jar*, *ehcache.jar*, *cglib.jar* i *antlr.jar*.



Rysunek 2.8. *Przykładowy diagram mapowania wygenerowany w HibernateTools*

Hibernate Tools

Wsparcie korzystania z biblioteki Hibernate w środowisku Eclipse zapewnia wtyczka programistyczna Hibernate Tools, udostępniana za darmo przez Red Hat Inc. Hibernate Tools dostarcza zestaw widoków i kreatorów zawierający m.in.:

- kreator dla realizacji procesu inżynierii wstecznej (ang. *reverse engineering*), umożliwiający wygenerowanie modelu obiektowego na podstawie tabel i kolumn istniejącej bazy danych,
- kreator konfiguracji *hibernate.cfg.xml*,
- w pełni funkcjonalny edytor dla języka Hibernate HQL,
- okno komend języka SQL,
- podpowiadanie składni SQL,
- w pełni funkcjonalny edytor plików odwzorowań *.hbm.xml*.

Wtyczkę Hibernate Tools możemy zainstalować według opisu w ramce „Wtyczki programistyczne w Eclipse”. Wówczas w polu *Work with* menedżera instalacji wpisujemy następujący adres:

http://download.jboss.org/jbosstools/updates/stable.

2.2.2.2. Obiekty dostępu do danych DAO

Obiekty implementujące interfejs do warstwy trwałości są zazwyczaj określane jako obiekty dostępu do danych (ang. *Data Access Objects — DAO*). Dobrą praktyką projektową wynikającą ze stosowania tego wzorca jest tworzenie pojedynczego obiektu DAO dla każdej utrwalanej klasy. Generalnie do najważniejszych zadań DAO należą tworzenie (pobieranie) sesji Hibernate oraz zapisywanie (pobieranie) obiektów.

Ponieważ pewne operacje (zapis, odczyt czy wyszukiwanie) powtarzają się we wszystkich DAO, celem uniknięcia powielania kodu możemy zdefiniować wspólny interfejs oraz klasę abstrakcyjną zawierającą definicję powtarzających się dla różnych typów operacji. Generyczny interfejs został zaprezentowany na listingu 2.8, a implementująca jego metody klasa abstrakcyjna `GenericDao` na listingu 2.9.

Listing 2.8. *Interfejs definiujący podstawowe operacje dla DAO*

```
package isrp.hibernate.model.dao;
import java.util.List;

public interface GenericDaoInterface {
   public abstract Object findById(int id);//wyszukiwanie po id
   public abstract void save(Object object);//zapisywanie nowego obiektu
   public abstract void saveOrUpdate(Object object);//zapisywanie lub modyfikowanie
                                                    //obiektu, jeśli istnieje
   public abstract void delete(Object object);//usuwanie obiektu
   public abstract List findAll();//wyszukiwanie wszystkich obiektów
}
```

Listing 2.9. *Implementacje podstawowych operacji dostępnych w DAO*

```
package isrp.hibernate.model.dao;
import isrp.hibernate.model.util.HibernateUtil;
import java.util.List;
import org.hibernate.SessionFactory;
import org.hibernate.Session;
import org.hibernate.Criteria;

public abstract class GenericDao implements GenericDaoInterface {
   protected Class persistedClass;
   protected Session session;

   public GenericDao(Class persistedClass) {
       /* pobranie sesji Hibernate z obiektu klasy użytkowej, zaprezentowanej na listingu 2.14*/
      SessionFactory factory = HibernateUtil.getSessionFactory();
      this.session = factory.getCurrentSession();
      this.persistedClass = persistedClass;
   }
   public Object findById(int id) {
      Object object = (Object) session.get(persistedClass, id);
      return object;
   }
   public void save(Object object) {
      session.save(object);
   }
   public void saveOrUpdate(Object object) {
      session.saveOrUpdate(object);
   }
   public void delete(Object object) {
      session.delete(object);
   }
   public List findAll() {
      Criteria criteria = session.createCriteria(persistedClass);
      return criteria.list();
   }
}
```

Zastosowanie klasy abstrakcyjnej `GenericDao` pozwala na znaczne uproszczenie implementacji konkretnych obiektów DAO. Na listingu 2.10 znajduje się kod źródłowy klasy `UserDao`, w którym zaimplementowane zostały wyłącznie operacje specyficzne dla utrwalania obiektów `User`.

Listing 2.10. *Kod źródłowy klasy UserDao*

```
package isrp.hibernate.model.dao;
import java.util.List;
import org.hibernate.Query;
import org.hibernate.criterion.Order;
import org.hibernate.criterion.Restrictions;
import isrp.hibernate.model.businessobject.User;
import isrp.hibernate.model.util.Md5HashCode;

public class UserDao extends GenericDao {

   public UserDao() {
      super(User.class);//
   }
   public String getAdministratorEmail(){
      User user = null;
      Query query = session.createQuery("FROM User where role=?")
         .setByte(0, new Byte("1"));
      user = (User) query.uniqueResult();
      return user.getEmail();
   }
   public List getReviewers(){
      return  session.createCriteria(persistedClass)
      .add(Restrictions.eq("role", new Byte("3")))
      .list();
   }
   public User authenticate(String email, String password) {
      password = Md5HashCode.getMd5HashCode(password);
      return (User) session.createCriteria(persistedClass)
         .add(Restrictions.eq("email", email))
         .add(Restrictions.eq("password", password))
         .uniqueResult();
   }
   public  User checkIfEmailExistInDB(String email) {
      Query query = session.createQuery("FROM User where email=?")
         .setString(0, email);
      return (User) query.uniqueResult();
   }
   public List<User> getUsers(String sortColumn){
      if (sortColumn==null){
         sortColumn="firstName";
      }
      return (List<User>) session.createCriteria(persistedClass)
         .addOrder(Order.asc(sortColumn))
         .list();
   }
}
```

Automatyczne generowanie DAO w Hibernate Tools

Hibernate umożliwia automatyczne generowanie obiektów DAO z wykorzystaniem narzędzia Ant (opis znajduje się w dokumentacji Hibernate) bądź wymienionej już w tym rozdziale wtyczki Hibernate Tools. Korzystając z wtyczki Hibernate Tools, w środowisku Eclipse w grupie menu Run pojawi się nowa opcja — *Hibernate Code Generation Configurations*, będąca konfiguracją uruchomieniową automatycznego generowania kodu przy użyciu Hibernate. Wybór tej opcji powoduje wyświetlenie okna kreatora *Create, manage and run configurations*, które w zakładce *Exporters* pozwala ustawić opcję automatycznego generowania kodu źródłowego klas DAO (opcja *DAO code*). Automatycznie generowane w ten sposób klasy będą jednak opierać się na wykorzystaniu interfejsu JNDI i obiektów klasy `javax.naming.InitialContext`. Nie jest to problemem w przypadku serwerów aplikacji, ale przy korzystaniu z prostych kontenerów serwletów, takich jak Tomcat, stanowi to utrudnienie, gdyż należy samodzielnie zapewnić kontekst początkowy. Tomcat wprawdzie dostarcza własny obiekt kontekstowy JNDI, ale nie można go modyfikować z poziomu kodu aplikacji (jest tylko do odczytu), co powoduje, że nie może on być wykorzystany przez Hibernate. Rozwiązaniem tego problemu jest użycie biblioteki Sun FS Context, dostępnej w pliku *fscontext.jar*, która umożliwia przechowanie kontekstu aplikacji w postaci plików.

2.2.2.3. Fabrykowanie obiektów DAO

Analogicznie do problemu lokalizacji fasad w warstwie usługowej w przypadku użycia DAO również istotnym problemem jest zapewnienie łatwej lokalizacji obiektów poprzez centralizację dostępu. W przypadku interfejsu do warstwy trwałości centralizacja ma za zadanie ułatwienie tworzenia obiektów DAO oraz zamiany ich implementacji. Dostarczenie interfejsu do tworzenia różnych obiektów implementujących wspólny interfejs bez specyfikowania ich konkretnych klas zapewnia wzorzec projektowy fabryki abstrakcyjnej (ang. *abstract factory*). Umożliwia on jednemu obiektowi tworzenie różnych, powiązanych ze sobą reprezentacji podobiektów, określając ich typy podczas działania programu.

Przykładowa fabryka abstrakcyjna dla obiektów DAO użyta w aplikacji ISRP zaprezentowana została na listingu 2.11, a jej implementacja na listingu 2.12. Przykłady wykorzystania fabryki obiektów do tworzenia DAO w warstwie usługowej zaprezentowano na listingu 2.13.

Listing 2.11. *Fabryka abstrakcyjna obiektów DAO*

```
package isrp.hibernate.model.dao.factory;
import isrp.hibernate.model.dao.*;

public abstract class DaoFactory {
   private static DaoFactory instance = new HibernateDaoFactory();

   public static DaoFactory getInstance() {
      return instance;
   }
```

```
   public abstract UserDao getUserDao();
   public abstract CountryDao getCountryDao();
   public abstract PaperDao getPaperDao();
   public abstract QuestionDao getQuestionDao();
   public abstract ReviewDao getReviewDao();
   public abstract AnswersquestionsDao getAnswersquestionsDao();
}
```

Listing 2.13. *Implementacja fabryki obiektów DAO*

```
package isrp.hibernate.model.dao.factory;
import isrp.hibernate.model.dao.*;

public class HibernateDaoFactory extends DaoFactory {

   public UserDao getUserDao() {return new UserDao();}
   public CountryDao getCountryDao() {return new CountryDao();}
   public PaperDao getPaperDao() {return new PaperDao();}
   public QuestionDao getQuestionDao() {return new QuestionDao();}
   public ReviewDao getReviewDao() {return new ReviewDao();}
   public AnswersquestionsDao getAnswersquestionsDao() {return new
   ↪AnswersquestionsDao();}
}
```

Listing 2.13. *Kod filtru umożliwiającego automatyczne zarządzanie sesją Hibernate*

```
package isrp.filters;
import isrp.util.HibernateUtil;
import java.io.IOException;
import javax.servlet.*;
import org.apache.commons.logging.*;
import org.hibernate.*;

public class HibernateSessionFilter implements Filter {

   private SessionFactory factory = null;
   private static final Log log = LogFactory.getLog(HibernateSessionFilter.class);
   public void doFilter(ServletRequest request, ServletResponse response,
            FilterChain chain) throws IOException, ServletException {
      Session session = factory.getCurrentSession();
      try {
         session.beginTransaction();
         log.debug("<< HIBERNATE START");
         chain.doFilter(request, response);
         log.debug(">> HIBERNATE END");
         session.getTransaction().commit();
      } catch (Throwable e) {
         try {
            if (session.getTransaction().isActive()) {
            session.getTransaction().rollback();
            }
         } catch (HibernateException e1) {
            log.error("Niemożliwe wykonanie operacji rollback", e1);
            e1.printStackTrace();
         }
         throw new ServletException(e);
```

```
        }
    }

    public void destroy() {}

    public void init(FilterConfig filterConfig) throws ServletException{
        factory = HibernateUtil.getSessionFactory();
        if (filterConfig == null) {
        log.error("HibernateSessionFilter: Nie zainicjowano filtru");
        throw new ServletException();
        }
    }
}
```

2.2.2.4. Zarządzanie sesją Hibernate w warstwie widoku

W celu zapewnienia zarządzania połączeniem z bazą danych (tzw. sesją Hibernate) możemy wykorzystać wzorzec *Open Session in View* (*OSIV*), którego idea opiera się na założeniu, że zawsze do wygenerowania odpowiedzi HTTP niezbędne są informacje przechowywane w bazie danych. Sesja Hibernate otwierana jest po pojawieniu się żądania HTTP i jest dostępna przez cały proces generowania strony widoku. Wprowadzone zmiany do bazy zatwierdzane są dopiero po realizacji całego żądania, tj. wygenerowaniu dokumentu odpowiedzi HTTP. Nie jest to jedyna możliwa do zastosowania tutaj strategia; warto wspomnieć o rozwiązaniach polegających na wprowadzaniu zmian do bazy danych w trybie automatycznego zatwierdzania transakcji (*autocommit*) czy też inicjowaniu sesji i pobieraniu danych niezbędnych do generowania warstwy widoku jeszcze przed przekazaniem sterowania do niej. Strategie zarządzania sesją Hibernate opisane są wyczerpująco w książkach *POJOs in Action. Developing Enterprise Applications with Lightweight Frameworks*[2] oraz *Java Persistence with Hibernate*[3]. Wzorzec OSIV pozwala na realizację wielu operacji w trakcie jednej sesji z Hibernate. Na listingu 2.13 przedstawiono użyty w aplikacji ISRP filtr implementujący wzorzec OSIV. Filtr przechwytuje wszystkie żądania płynące do aplikacji i dla każdego z nich rozpoczyna nową transakcję, w której realizowane są wszystkie operacje na bazie danych, realizowane w trakcie przetwarzania żądania HTTP. Transakcja jest zatwierdzana dopiero po zakończeniu przetwarzania żądania. Filtr zawiera obsługę wszelkich wyjątków i błędów, które mogą wystąpić podczas przetwarzania żądania. Jeśli istnieje aktywna sesja Hibernate, to po wystąpieniu błędów lub wyjątków zmiany w bazie danych są wycofywane (*rollback*). W kodzie do pobrania bieżącej sesji Hibernate użyto klasy pomocniczej `HibernateUtil`, zaprezentowanej na listingu 2.14. Interakcje występujące pomiędzy elementami implementacji wzorca OSIV przedstawione zostały na rysunku 2.9.

Rejestracja filtru w pliku *web.xml* w najprostszym schemacie będzie mieć następującą postać:

[2] Chris Richardson, *POJOs in Action. Developing Enterprise Applications with Lightweight Frameworks*, Manning 2006.

[3] Christian Bauer, Gavin King, *Java Persistence with Hibernate*, Manning 2006.

```
<filter>
      <filter-name>HibernateSessionFilter</filter-name>
      <filter-class>isrp.filters.HibernateSessionFilter</filter-class>
   </filter>
   <filter-mapping>
      <filter-name>HibernateSessionFilter</filter-name>
      <url-pattern>/*</url-pattern>
   </filter-mapping>
```

Jak łatwo zauważyć, w powyższym kodzie wykonano mapowanie filtru na wszystkie zasoby URL, co jest rozwiązaniem wysoce niewydajnym — sesja Hibernate jest tworzona zawsze, również dla tych żądań HTTP, których przetwarzanie nie wymaga połączenia do bazy danych. Użytkownik w tym miejscu w zależności od potrzeb może zmienić mapowanie na wybrane żądania lub filtrację żądań obsługiwanych tylko przez wybrany serwlet (np. poprzez wstawienie w miejsce znaczników `<url-pattern>` znacznika `<servlet-name>Faces Servlet</servlet-name>` do obsługi żądań przez aplikację JSF).

Listing 2.14. *Kod klasy użytkowej HibernateUtil*

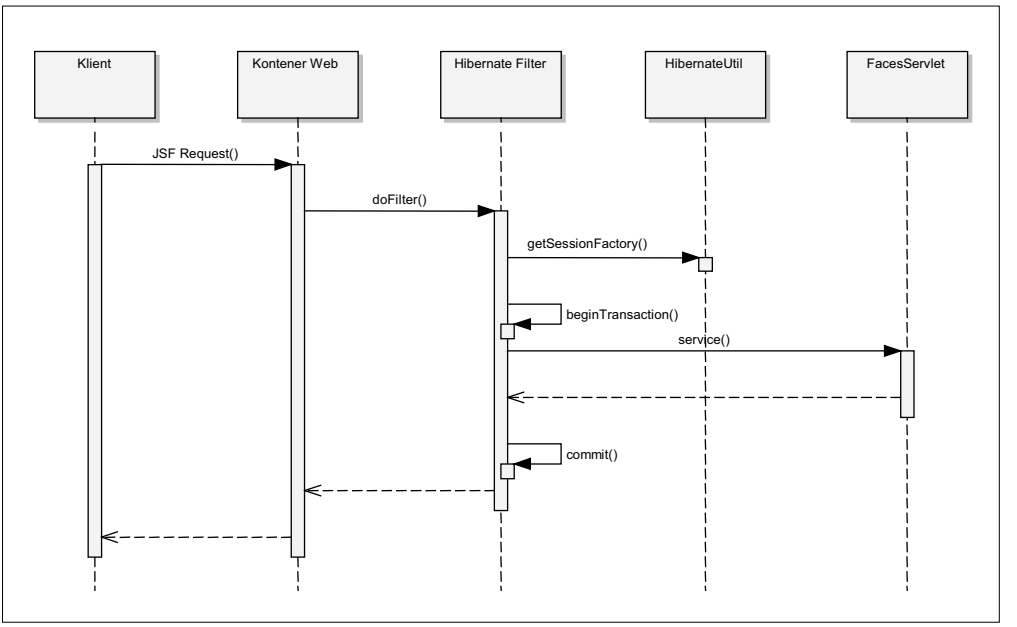
```
package isrp.hibernate.model.util;
import org.hibernate.cfg.AnnotationConfiguration;
import org.hibernate.SessionFactory;

public class HibernateUtil {
   private static final SessionFactory sessionFactory;
   private static final Log log = LogFactory.getLog(HibernateUtil.class);

   static {
      try {// Tworzy SessionFactory na podstawie konfiguracji w pliku hibernate.cfg.xml
         sessionFactory = new AnnotationConfiguration()
            .configure().buildSessionFactory();
      } catch (Throwable ex) {
         log.error("Nie można pobrać sesji Hibernate " + ex);
         throw new ExceptionInInitializerError(ex);
      }
   }
   public static SessionFactory getSessionFactory() {
      return sessionFactory;
   }
}
```

Uwaga

Organizując kod aplikacji, należy pamiętać, że `HibernateSessionFilter` jest elementem warstwy widoku aplikacji WWW, podczas gdy klasa `HibernateUtil` zapewnia dostęp do warstwy trwałości.



Rysunek 2.9. *Diagram sekwencji wzorca OSIV dla aplikacji JSF*

2.3. Organizacja kodu aplikacji w Eclipse

Kod aplikacji WWW opartych na technologii Java EE zazwyczaj jest wytwarzany w postaci wielu modułów, rozwijanych niezależnie przez różne zespoły projektowe. Rozproszenie kodu aplikacji w wielu projektach ułatwia m.in.:

- zarządzanie kodem źródłowym i plikami zasobów,
- podział pracy w zespołach,
- zautomatyzowanie procesu wytwarzania i testowania aplikacji.

W dalszej części rozdziału znajduje się opis rozwiązań, jakie zapewnia Eclipse WTP w zakresie organizacji kodu źródłowego oraz integracji i wdrażania skompilowanych modułów aplikacji na serwerze.

2.3.1. Projekty i moduły Java EE w Eclipse

Zanim zajmiemy się planowaniem rozmieszczenia kodu aplikacji ISRP, warto przybliżyć podstawowe koncepcje związane z wytwarzaniem i wdrażaniem aplikacji Java EE w środowisku Eclipse Web Tools Platform, a mianowicie *projekt* i *moduł aplikacji*.

Wszystkie pliki źródłowe oraz inne zasoby aplikacji bądź biblioteki muszą być umieszczone w projekcie składającym się z jednego lub wielu zagnieżdżonych katalogów. Projekt stanowi podstawową jednostkę organizacyjną kodu aplikacji i jest zapisywany w **przestrzeni projektów** (ang. *workspace*). Projekt opisywany jest za pomocą metadanych, które zawierają pomocnicze informacje o wymaganych zasobach zewnętrznych, sposobie kompilacji czy środowisku uruchomieniowym. Logiczna struktura prezentowanych w oknie *Project Explorer* katalogów i zasobów projektu jest zoptymalizowana pod kątem ułatwienia pracy deweloperom.

Moduł aplikacji (ang. *module*) reprezentuje gotowy do wdrożenia na serwerze aplikacji kod wynikowy projektu (powstały po jego skompilowaniu), który może być reprezentowany przy użyciu plików archiwalnych, np. WAR lub JAR. Moduły Java EE posiadają określoną przez specyfikację Java EE fizyczną strukturę katalogów oraz plików. Moduły mogą być wdrażane w kontenerze Java EE zarówno w postaci spakowanych plików archiwum, jak też rozpakowanych struktur katalogów. Eclipse WTP wspiera wytwarzanie i wdrażanie następujących modułów Java EE (w nawiasach podano typy archiwum oraz nazwy odpowiadających im projektów Eclipse po myślniku):

- *Enterprise application* (EAR) — *Enterprise application Project*,
- *Enterprise application client* (JAR) — *Application Client Project*,
- *Enterprise JavaBean* (JAR) — *EJB Project*,
- *Web application* (WAR) — *Dynamic Web Project*,
- *Resource adapter for JCA* (RAR) — *Connector Project*,
- *Utility module* (JAR) — *Utility Project*.

Pierwszych pięć wymienionych typów modułów jest określonych w specyfikacji Java EE, natomiast ostatni typ został wprowadzony przez środowisko Eclipse i stanowi **moduł użytkowy** (*Utility module*) ogólnego przeznaczenia, umożliwiający podział kodu aplikacji na oddzielne projekty. Każdy projekt WTP może generować tylko jeden moduł aplikacji, ale w kodzie źródłowym projektu można odwoływać się do kodu umieszczonego w innych modułach. Odwoływanie się do kodu znajdującego się w innych modułach wymaga jednak jawnego zdefiniowania zależności pomiędzy projektami, co jest zilustrowane przykładem w dalszej części rozdziału.

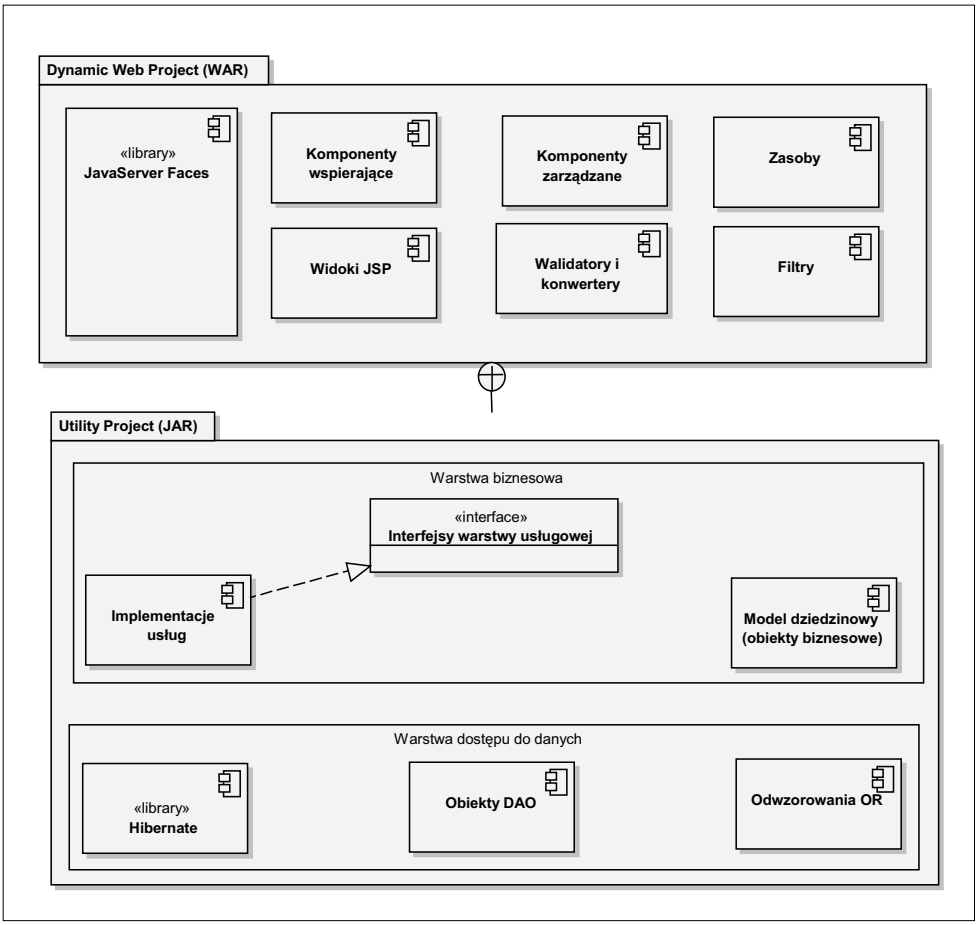
2.3.2. Zastosowanie projektu użytkowego do przechowywania kodu warstwy biznesowej

Załóżmy, że chcemy umożliwić rozwijanie aplikacji ISRP w dwóch projektach, tak by kod warstwy prezentacji opartej na szkielecie JSF znajdował się w jednym projekcie, a kod pozostałych warstw (biznesowej i dostępu do danych) w innym. Zaletą takiego podziału będzie możliwość wielokrotnego wykorzystania modułu zawierającego warstwę biznesową w aplikacjach zawierających interfejsy oparte na różnych technologiach (np. JSF i Swing).

Uwaga

W materiałach pomocniczych do książki Czytelnik znajdzie m.in. kod modułu użytkowego ISRPModel, zawierający implementację warstwy biznesowej, udostępniony w postaci zarchiwizowanego pliku projektu Eclipse oraz skompilowanej wersji modułu w pliku JAR. Czytelnik, który chciałby pominąć zagadnienia wytwarzania warstwy biznesowej, może skorzystać z gotowego do użycia modułu ISRPModel i skupić się wyłącznie na implementacji interfejsu użytkownika według wskazówek zamieszczonych w następnych rozdziałach.

Na rysunku 2.10 przedstawiono strukturę aplikacji ISRP, która składa się z modułów Web Application oraz Utility Module. Moduł Web Application zawiera kod warstwy prezentacji i jest tworzony w środowisku Eclipse przy użyciu projektu typu Dynamic Web Project (przykład takiego projektu został zaprezentowany w rozdziale 1.). Moduł Utility Module przechowuje warstwę biznesową aplikacji ISRP i jest tworzony w środowisku Eclipse przy użyciu projektu typu Utility Project.



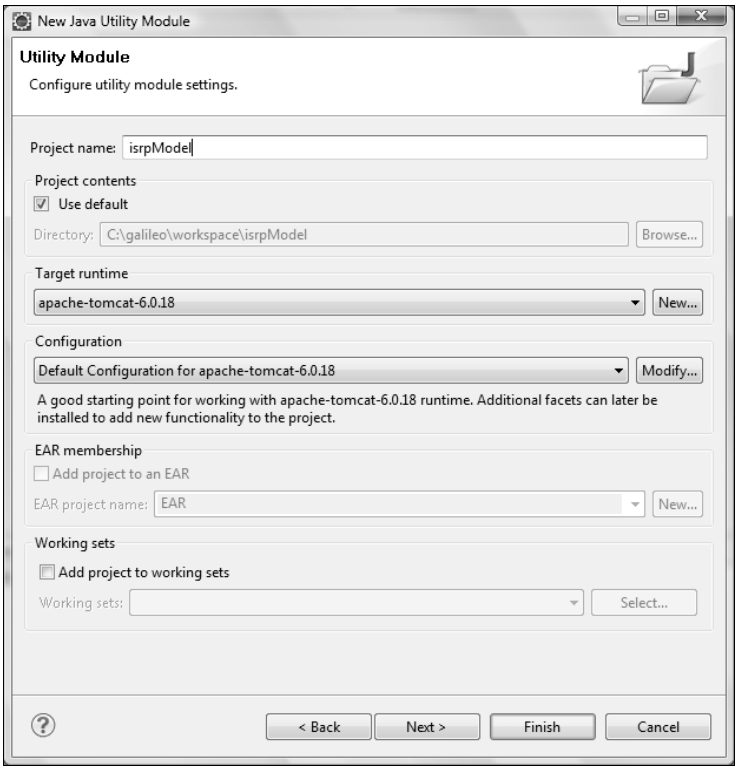
Rysunek 2.10. *Struktura modułów aplikacji ISRP oraz zależności występujące pomiędzy nimi*

Projekt typu Java EE Utility Project jest bardzo podobny do zwykłego projektu Javy w środowisku Eclipse, ale posiada dodatkowy pakiet rozszerzeń (tzw. *facet*) — Utility Module (idea pakietów rozszerzeń Eclipse została omówiona w rozdziale 1., w ramce „JSF Facet"). Utility Module stanowi część konfiguracji uruchomieniowej projektu i umożliwia wykorzystanie skompilowanego kodu projektu w innych modułach aplikacji korporacyjnej (Java EE). Gdy zmiany w projekcie zostaną zapisane, Eclipse automatycznie go skompiluje i spakuje kod wynikowy do pliku JAR, który następnie jest publikowany (bądź aktualizowany) na serwerze oraz w innych projektach, które z niego korzystają.

Ze względu na opisane wyżej właściwości Utility Project znakomicie nadaje się do przechowywania kodu warstwy biznesowej oraz interfejsu warstwy trwałości, oczywiście pod warunkiem, że model dziedzinowy nie jest oparty na komponentach EJB (w takim przypadku wymagany jest EJB Module).

W celu utworzenia nowego projektu Java EE Utility Project należy wykonać następujące kroki:

1. Z menu aplikacji wybieramy opcję *File/New/Project*. Pojawi się okno kreatora *New Project*, w którym wyświetli się lista typów projektów.
2. Na liście wskazujemy grupę projektów korporacyjnych (*Java EE*). Po rozwinięciu elementów grupy wybieramy opcję *Utility Project* i zatwierdzamy przyciskiem *Next*. Wyświetli się okno kreatora *New Java Utility Module*, pokazane na rysunku 2.11.



Rysunek 2.11. *Tworzenie projektu typu Utility Project*

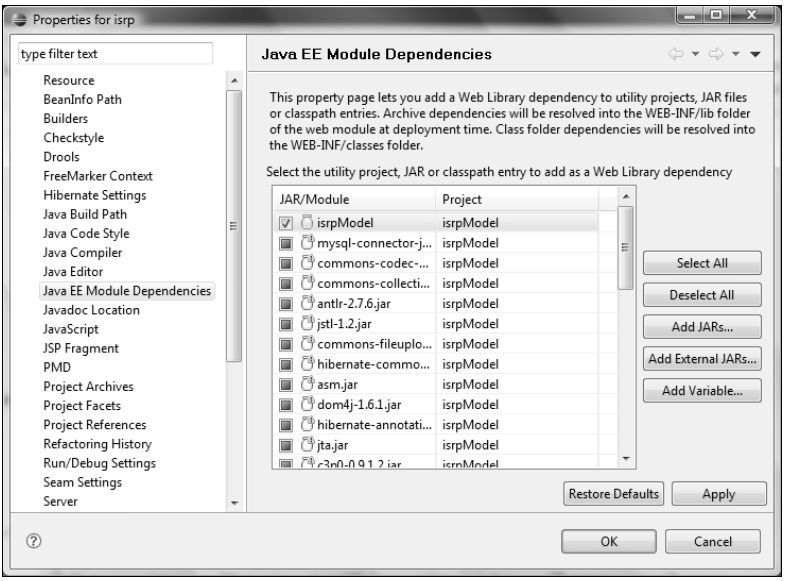
3. W polu *Project name* wpisujemy nazwę projektu. Następnie z listy rozwijanej *Target Runtime* wybieramy środowisko uruchomieniowe serwera, na którym ma być publikowany kod wynikowy projektu. Aby dodać kolejne pakiety rozszerzeń Eclipse do projektu (np. dla szkieletu Spring), należy w sekcji *Configuration* kliknąć przycisk *Modify* i w nowym oknie kreatora zaznaczyć odpowiednie pola wyboru. Wszystko zatwierdzamy przyciskiem *Next* i przechodzimy do kolejnego okna, w którym możemy załączyć do projektu dodatkowe źródła. Całość zatwierdzamy przyciskiem *Finish*.

2.3.3. Moduły zależne

Istnieją dwa sposoby wykorzystania gotowego modułu użytkowego w aplikacji, a mianowicie:

- bezpośrednie wdrożenie modułu użytkowego w środowisku uruchomieniowym serwera (opisane w poprzednim punkcie),
- ustawienie zależności pomiędzy projektami i wykorzystanie modułu użytkowego podczas budowy innego modułu.

W drugim przypadku we właściwościach projektu Utility Project ustawiamy wartość *<None>* w polu *Target Runtime*. Aby dołączyć moduł Utility Module do dowolnego innego projektu Java EE, w oknie *Project Explorer* zaznaczamy projekt docelowy i z menu kontekstowego wybieramy opcję *Properties*. Następnie w oknie właściwości wybieramy stronę *Java EE Module Dependencies*, po czym zaznaczamy pole wyboru przy nazwie projektu, którego skompilowany kod chcemy dołączyć, jak pokazano na rysunku 2.12.



Rysunek 2.12. *Ustawianie zależności pomiędzy projektami Java EE*

Na przykład do budowy aplikacji ISRP wykorzystamy dwa projekty — projekt Dynamic Web Project (DWP) do rozwijania kodu warstwy prezentacji oraz Utility Project (UP) do rozwijania warstwy biznesowej oraz warstwy dostępu do danych. Ponieważ oczekiwanym modułem wynikowym aplikacji ma być aplikacja WWW, projekt DWP będzie zależny od UP. Po każdej kompilacji wynikowy moduł Utility Module będzie automatycznie dodawany (zamieniany) do wynikowego pliku WAR oraz sekcji Web App Libraries w ścieżce przeszukiwania projektu DWP. Jednocześnie każda zmiana pliku WAR będzie powodować automatyczną publikację nowej wersji aplikacji WWW na serwerze, chyba że użytkownik zmieni domyślne ustawienia dotyczące publikowania (patrz rozdział 6., rysunek 6.2).

Wszystkie klasy i dołączone do projektu użytkowego biblioteki są widoczne w projekcie zależnym. Klasy i biblioteki projektu zależnego nie są jednak widoczne w projekcie użytkowym. Niestety, nie możemy zdefiniować odwrotnej zależności, gdyż spowodowałoby to powstanie *cyrkularności*, która jest przez środowisko Eclipse automatycznie wykrywana i traktowana jako błąd. Dość powszechnym rozwiązaniem tego problemu jest umieszczenie kodu niezbędnych dla obu projektów klas w oddzielnym projekcie użytkowym (Utility Project). W takim przypadku dwa projekty są zależne od trzeciego i nie powstaje pomiędzy nimi zależność cyrkularna.

**Dobry interfejs aplikacji WWW to połowa jej sukcesu.
Osiągnij go z JavaServer Faces!**

Język Java od lat zdobywa i ugruntowuje swoją popularność wśród programistów i twórców aplikacji WWW, a rozmaite platformy i rozwiązania, w których jest on wykorzystywany, zostały na stałe włączone do pakietu narzędzi stosowanych przez wielu z nich na co dzień. Jednym z najbardziej popularnych tego typu narzędzi jest JavaServer Faces. Można dzięki niemu w prosty sposób tworzyć interfejsy użytkownika aplikacji, wykorzystujące platformę Java EE. Ten spójny i kompletny szkielet programistyczny jest obecnie najbardziej elastycznym, najlepiej dopracowanym i najprostszym w użyciu rozwiązaniem, opartym na technologii serwletów.

Jednak „najprostszy" wcale nie musi oznaczać „prosty", o czym z pewnością miało okazję przekonać się wielu studentów kierunków informatycznych i profesjonalnych programistów, którzy postanowili praktycznie zapoznać się z możliwościami tej technologii. Nieocenioną pomocą okaże się dla nich książka „JavaServer Faces i Eclipse Galileo. Tworzenie aplikacji WWW", dzięki której można uniknąć wielu typowych błędów i nauczyć się biegle korzystać z JSF, zdobywając przy tym kompletną wiedzę na temat mechanizmów i rozwiązań zapewniających działanie tej platformy. Co więcej, opisano tu nie tylko samą technologię, lecz również sposób jej praktycznego wykorzystania w konkretnych projektach, co w przyszłości zaowocuje z pewnością opracowaniem niejednej doskonałej i cieszącej oko aplikacji WWW.

- **Działanie aplikacji WWW i sposoby ich projektowania w oparciu o język Java**
- **Podstawowe informacje na temat szkieletu programistycznego JSF**
- **Realizacja praktycznego projektu z wykorzystaniem JavaServer Faces**
- **Rozszerzanie standardowej implementacji JSF i tworzenie niestandardowych interfejsów użytkownika**
- **Opis środowiska programistycznego Eclipse Galileo oraz pakietu narzędzi Web Tools Platform**

Naucz się szybko i sprawnie tworzyć rozbudowane interfejsy użytkownika aplikacji WWW za pomocą szkieletu programistycznego JavaServer Faces.

Andrzej Marciniak od 2002 roku jest adiunktem w Instytucie Sterowania i Systemów Informatycznych Uniwersytetu Zielonogórskiego. Jest autorem bądź współautorem ponad 30 opracowań naukowych w dziedzinie informatyki – książek, artykułów, referatów wydanych w kraju i za granicą. Ma również bogate doświadczenie w projektowaniu i wdrażaniu komercyjnych systemów informatycznych opartych na technologiach Java i Java EE.

Cena: 54,90 zł

Nr katalogowy: 5616

Księgarnia internetowa: http://helion.pl

Zamówienia telefoniczne: 0 801 339900
0 601 339900

Wydawnictwo Helion
ul. Kościuszki 1c, 44-100 Gliwice
44-100 Gliwice, skr. poczt. 462
32 230 98 63
http://helion.pl
e-mail: helion@helion.pl

helion.pl księgarnia internetowa

ISBN 978-83-246-2656-4
9788324626564

Informatyka w najlepszym wydaniu