МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЧЕРНІГІВСЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНОЛОГІЧНИЙ УНІВЕРСИТЕТ

ТЕХНОЛОГІЇ ПРОЕКТУВАННЯ ПРОГРАМНИХ СИСТЕМ

МЕТОДИЧНІ ВКАЗІВКИ
до виконання лабораторних робіт та самостійної роботи
для студентів спеціальності 123 – «Комп'ютерна інженерія»

Обговорено і рекомендовано
на засіданні кафедри
інформаційних та комп'ютерних
систем
Протокол № 2
від 27 вересня 2017 р.

Чернігів ЧНТУ 2018

Технології проектування програмних систем. Методичні вказівки до виконання лабораторних робіт та самостійної роботи для студентів спеціальності 123 – «Комп'ютерна інженерія». / Укл.: Казимир В.В., Пріла О.А. – Чернігів: ЧНТУ, 2018. - 61 с., укр. мовою.

Укладачі: КАЗИМИР ВОЛОДИМИР ВІКТОРОВИЧ, доктор технічних наук, професор кафедри інформаційних та комп'ютерних систем
ПРІЛА ОЛЬГА АНАТОЛІЇВНА, доцент кафедри інформаційних та комп'ютерних систем

Відповідальний за випуск: ЗАЙЦЕВ СЕРГІЙ ВАСИЛЬОВИЧ, завідувач кафедри інформаційних та комп'ютерних систем, доктор технічних наук, доцент

Рецензент: НІКІТЕНКО ЄВГЕНІЙ ВАСИЛЬОВИЧ, кандидат фізико-математичних наук, доцент кафедри інформаційних та комп'ютерних систем Чернігівського державного технологічного університету

Зміст

Вступ

Курс «Технології проектування програмних систем» присвячений вивченню сучасних підходів до проектування і реалізації корпоративних програмних систем (КПС), що характеризуються складністю бізнес-логіки, а також високими вимогами до рівня надійності, продуктивності, розширюваності, масштабованості, безпеки і захищеності.

У методичних вказівках представлений поетапний процес розробки програмної системи рівня підприємства. Розглядаються типові підходи до проектування і реалізації корпоративних програмних систем, а також практичні особливості використання технологій розробки КПС. Курс орієнтований на використання специфікації Java EE 7. Приділяється увага питанням тестування програмного забезпечення.

Звіт з лабораторних робіт має бути чітко та якісно оформлений та включати такі розділи: постановка задачі, результати проектування/реалізації, висновки.

Методичні вказівки можуть використовуватися при виконанні дипломної роботи і проходження переддипломної практики.

При підготовці методичних вказівок використовувалися навчальні матеріали та розробки Ткача Ю. Е., який викладав дисципліну «Технології проектування програмних систем» на кафедрі інформаційних і комп'ютерних систем з 2007 по 2009 р.

1 Лабораторна робота №1 Вивчення принципів модульного тестування з використанням бібліотеки JUnit 4

1.1 Мета роботи

Навчитися розробляти Unit-тести з використанням інструментальної бібліотеки JUnit. Розробити тести для одного зі стандартних класів JDK.

1.2 Теоретичні відомості

Тестування програмного забезпечення - процес виявлення помилок в програмному забезпеченні. Виділяють види тестування згідно таких критеріїв: ступень ізольованості компонентів (модульне, інтеграційне, системне тестування); об’єкт тестування (функціональне, тестування навантаження, тестування інтерфейсу; тестування безпеки); наявність інформації про структуру та функціонування системи (тестування «чорного», «білого» ящику); час проведення (альфа-, бета-тестування, приймальне тестування, регрес тестування); ступінь автоматизації (ручне, автоматизоване, напівавтоматизоване); виконання кода (статичне та динамічне тестування).

Кінцевою метою будь-якого процесу тестування є забезпечення якості програмного забезпечення. При цьому жоден вид тестування не надає 100% гарантії працездатності програми.

Планування тестових сценаріїв має бути виконано таким чином, щоб перевірялися різні варіанти використання програми.

Модульне тестування передбачає перевірку працездатності найменших складових програми: зазвичай це окремий клас і його методи. У випадках, якщо при виконанні методів тестованого класу відбувається виклик методів стороннього класу, необхідно відокремити тестовані класи один від одного, замінити їх за допомогою mock-об'єктів, заглушок, імітацій. Наприклад, якщо тестується метод класу, який приймає ім'я і пароль користувача, а потім звертається до бази даних для перевірки правильності введених користувачем даних аутентифікації (інший клас), то потрібно замінити клас, який працює з реальними даними «імітацією», яка не виконує звернення до бази даних, а зберігає дані користувачів всередині самого класу. Імітація повинна бути досить проста, щоб не містити помилок - в іншому випадку говорити про модульному тестуванні не можна.

JUnit - популярний в даний час інструмент модульного тестування java-додатків. JUnit 4 заснований на використанні анотацій.

Наприклад, для тестування правильності виконання методів класу Calculator, необхідно створити наступний тестовий клас TestCalculator:

```
import org.junit.Test;
import static org.junit.Assert.*;

public class TestCalculator {
```

```
    public Calculator calculator;

    @Before
    public void prepareTestData() {
        calculator = new Calculator ();
        ...
    }

    @Before
    public void setupMocks() { ... }

    @After
    public void cleanupTestData() { ... }

    @BeforeClass
    public static void setupDatabaseConnection() { ... }

    @AfterClass
    public static void teardownDatabaseConnection() { ... }

    @Test
    public void testAdd() {
        assertEquals(4, calculator.add(1, 3) );
        assertEquals(2, calculator.add( -3, 5 ) );
        assertEquals(0, calculator.add(0, 0) );
        assertEquals(-10, calculator.add(-4, -6) );
    …

    }
    @Test
    public void testDivide() {
    …..

    }
    @Test(expected=ArithmeticException.class)
    public void testDivisionByZero() {
        calculator.divide(4, 0);
    }
    …
}
```

Рекомендується (але необов'язково) використання стандарту іменування тестового класу і його методів (префікс test).

Для позначення тестового методу використовується анотація @Test. Метод assertEquals перевіряє рівність очікуваного і отриманого значення. Інші подібні методи - assertTrue, assertFalse, assertNull, assertNotNull, assertSame.

Анотацією @Before позначають методи, які повинні бути запущені перед запуском кожного тестового методу, анотацією @After - після запуску кожного тестового методу відповідно. Якщо в тестовому класі визначе-

но кілька методів з анотацією @Before, порядок запуску методів може бути довільним.

Анотаціями @BeforeClass і @AfterClass позначають методи, які повинні бути виконані до і відповідно після виконання всього набору тестів класу. Ці методи повинні бути оголошені як статичні.

Плани тестів повинні включати і «неправильні» варіанти використання системи, наприклад, неприпустимі вхідні параметри методу. Тестування того, що код коректно працює в виняткових ситуаціях, також важливо, як і тестування функціональності коду. Для тестування коректності роботи методу в виняткової ситуації необхідно оголосити очікуване виключення в анотації @Test.

Параметр timeout анотації @Test дозволяє вказати тимчасові обмеження виконання тестового методу:

```
@Test(timeout=5000)
public void testOperation() {   ...   }.
```

У деяких випадках потрібно відключення тестового методу класу. Для цього використовується анотація @Ignore. Як параметр бажано вказати текст повідомлення, яке буде виведено при запуску тестових методів класу.

```
@Ignore("Not running because <fill in a good reason
here>")
@Test
public void testOperation(){   ...   }.
```

Набор тестов можно определить следующим образом:

```
@RunWith(value=Suite.class)
@SuiteClasses(value={CalculatorTest.class,
AnotherTest.class})
public class AllTests {
...
}
```

JUnit 4 вводить можливість визначення різних класів для запуску тестів. Анотація @RunWith визначає використання класу Suite для запуску тестів. У свою чергу, анотація @SuiteClasses використовується класом Suite для визначення тестів, які входять в даний набір. При визначенні набору тестів допускається можливість використання методів, анотованих за допомогою @BeforeClass і @AfterClass, які будуть викликатися перед початком запуску всіх тестів і після виконання останнього тесту з набору відповідно. Таким чином, вона може змінювати необхідних параметрів відразу для цілого набору тестів.

Клас Parameterized бібліотеки JUnit 4 дозволяє запускати одні і ті ж тести, але з різними наборами даних. Розглянемо це на прикладі. Нижче наведено приклад тесту для методу, який обчислює факторіал заданого числа n:

```
@RunWith(value=Parameterized.class)
public class TestFactorial {
    private long expected;
    private int value;
    @Parameters
    public static Collection data() {
        return Arrays.asList( new Object[][] {
                        { 1, 0 },   // expected, value
                        { 1, 1 },
                        { 2, 2 },
                        { 24, 4 },
                        { 5040, 7 },
                        });
    }
public TestFactorial(long expected, int value) {
     this.expected = expected;
     this.value = value;
}
@Test
public void factorial() {
     Calculator calculator = new Calculator();
     assertEquals(expected,
calculator.factorial(value));
}
}
```

Клас Parameterized запускає всі тести класу FactorialTest (в прикладі він тільки один), використовуючи при цьому дані методів, промаркованих анотацією @Parameters. В даному випадку маємо список з п'ятьма елементами. Кожен елемент містить масив, який буде використаний в якості аргументів конструктора класу FactorialTest. Тест factorial () використовує ці дані при виклику методу assertEquals (). Тест буде виконаний 5 разів з використанням наступних даних:

```
factorial#0: assertEquals(1,calculator.factorial(0));
factorial#1: assertEquals(1,calculator.factorial(1));
factorial#2: assertEquals(2,calculator.factorial(2));
factorial#3: assertEquals(24,calculator.factorial(4));
factorial#4:assertEquals(5040,calculator.factorial(7));
```

1.3 Порядок виконання роботи

Створити тестовий клас і тести для перевірки належного функціонування методів класу. Перевірці підлягають не менше 10 методів класу на вибір.

Класи для тестування вибираються відповідно таблиці 1.1.

Таблиця 1.1 – Вибір класу для тестування

Група 1		**Група 2**	
№ п/п	Клас	№ п/п	Клас
1	java.util.ArrayList	1	java.lang.Enum
2	java.lang.Byte	2	java.util.HashMap
3	java.io.BufferedReader	3	java.util.Scanner
4	java.math.BigInteger	4	java.util.Vector
5	java.text.BreakIterator	5	java.lang.Long
6	java.util.Arrays	6	java.io.FileOutputStream
7	java.lang.Boolean	7	java.util.LinkedList
8	java.io.BufferedWriter	8	java.io.ObjectInputStream
9	java.text.DecimalFormat	9	java.io.ObjectOutputStream
10	java.util.BitSet	10	java.util.LinkedHashSet
11	java.lang.Character	11	java.util.HashTable
12	java.io.DataInputStream	12	java.util.jar.Manifest
13	java.math.BigDecimal	13	java.lang.reflect.Constructor
14	java.text.SimpleDateFormat	14	java.util.jar.Attributes
15	java.util.Calendar	15	java.io.FileInputStream
16	java.util.StringTokenizer	16	java.lang.reflect.Method
17	java.lang.Double	17	java.math.BigDecimal
18	java.io.DataOutputStream	18	java.io.File
19	java.io.File	19	java.text.BreakIterator
20	java.util.HashSet	20	java.io.StringReader
21	java.lang.StringBuffer	21	java.lang.Float
22	java.util.Collections	22	java.util.PriorityQueue
23	java.text.StringCharacterIterator	23	java.lang.Math
24	java.lang.String	24	java.io.DataOutputStream
25	java.lang.Math	25	java.io.StringReader
26	java.io.LineNumberReader	26	java.util.ArrayList
27	java.io.StringReader	27	java.util.Calendar
28	java.util.PriorityQueue	28	java.util.BitSet
29	java.util.Stack	29	java.util.Collections
30	java.lang.Integer	30	java.util.Stack
31	java.util.LinkedHashMap	31	java.lang.Double
32	java.lang.Float	32	java.io.BufferedWriter
33	java.lang.Short	33	java.util.Arrays
34	java.lang.reflect.Field	34	java.lang.Short
Група 3			
№ п/п	Клас	№ п/п	Клас
1	java.util.jar.Attributes	18	java.lang.String

Продовження таблиці 1.1

№ п/п	Клас	№ п/п	Клас
2	java.util.Stack	19	java.lang.Short
3	java.util.PriorityQueue	20	java.lang.reflect.Field
4	java.util.LinkedHashMap	21	java.lang.Math
5	java.util.HashSet	22	java.lang.Integer
6	java.util.Collections	23	java.lang.reflect.Constructor
7	java.util.Calendar	24	java.lang.Double
8	java.util.BitSet	25	java.util.jar.Manifest
9	java.util.Arrays	26	java.lang.Byte
10	java.util.ArrayList	27	java.lang.Boolean
11	java.text.StringCharacterIterator	28	java.io.StringReader
12	java.text.SimpleDateFormat	29	java.io.LineNumberReader
13	java.text.DecimalFormat	30	java.io.File
14	java.text.BreakIterator	31	java.io.DataOutputStream
15	java.lang.reflect.Method	32	java.io.DataInputStream
16	java.math.BigDecimal	33	java.io.BufferedWriter
17	java.lang.StringBuffer	34	java.io.BufferedReader

2 Лабораторна робота №2 Проектування слою бізнес-логіки корпоративної програмної системи

2.1 Мета роботи

Вивчення і застосування типових підходів до організації бізнес-логіки корпоративного програмного застосування.

2.2 Теоретичні відомості

2.2.1 Призначення шару бізнес-логіки в архітектурі застосування

Логіка домену (бізнес-логіка або логіка предметної області) описує основні функції програми, призначені для реалізації поставленої перед нею мети. До таких функцій відносяться обчислення на основі введених і збережених даних, перевірка всіх елементів даних і обробка команд, що надходять від шару уявлення, а також передача інформації шару джерела даних.

Бізнес-логіка зазвичай схильна до частих змін, тому дуже важливою є можливість найпростішої модифікації і тестування цього шару коду. Звідси випливає нагальна необхідність знижувати ступінь залежності моделі предметної області від інших верств системи. Саме ця вимога є основоположним аспектом багатьох типових рішень, що мають відношення до "розшарування" системи.

2.2.2 Вибір типового рішення реалізації бізнес-логіки

Шаблон сценарій транзакцій. Найпростіший підхід до опису бізнес-логіки. Принцип дії - процедура, яка отримує на вхід інформацію від шару уявлення, обробляє її, проводячи необхідні перевірки і обчислення, зберігає в базі даних і активізує операції інших систем. Потім процедура повертає шару уявлення певні дані, можливо, здійснюючи допоміжні операції для форматування результату.

Бізнес-логіка в цьому випадку може бути охарактеризована процедур, по одній на кожну (складову) операцію, яку здатне виконувати додаток. Таким чином, шаблон можна трактувати як сценарій дії, або бізнес-транзакцію.

При використанні типового рішення сценарій транзакції логіка предметної області розподіляється по транзакціях, виконуваних в системі. Якщо, наприклад, користувачеві необхідно замовити номер у готелі, відповідна процедура повинна передбачати дії по перевірці наявності відповідного номера, обчислення суми оплати і фіксації замовлення в базі даних.

Перевагами застосування є такі:

- Простота реалізації. Саме такий вид організації логіки, ефективний з точки зору сприйняття і продуктивності, характерний і природний для невеликих додатків.

- Чіткі межі транзакцій. Годі й враховувати наявність і варіанти функціонування інших паралельних транзакцій. Завдання методу, який займається обробкою запит, - отримати вхідну інформацію, опитати базу даних, зробити висновки і зберегти результати.

Недоліки:

- Проблеми з реалізацією складної логіки. В міру ускладнення бізнес-логіки стає все важче утримувати її в добре структурованому вигляді.
- Схильність до дублювання коду. Якщо кількох транзакціях необхідно виконувати схожі функції, виникає ймовірність дублювання фрагментів коду. Винесення загальних підпрограм часто повністю не вирішує питань дублювання коду.
- Складність застосування ГО прийомів і шаблонів.

Шаблон «Модуль таблиці». Передбачає створення окремого класу для кожної таблиці або запиту з методами для їх обробки. Єдиний екземпляр класу містить всю логіку обробки даних таблиці.

Сильна сторона рішення модуль таблиці полягає в можливості ефективного використання ресурсів реляційної бази даних.

Модулю таблиці відповідає певна табличная структура даних. Подібна інформація зазвичай є результатом виконання SQL-запиту і зберігається у вигляді безлічі записів. Об'єкт модуля таблиці не містить ідентифікаційного ознаки об'єкта, окремі екземпляри записів бази даних відсутні. Логіка обробки даних прив'язана до реляційної моделі зберігання даних.

Можливе створення модулів таблиці для складних SQL-запитів і віртуальних таблиць.

Переваги:

- ефективне використання ресурсів СКБД;
- наявність інструментальних засобів.

Недоліки:

- складність застосування об'єктного підходу до реалізації складної логіки.

Шаблон «Модель предметної області (домен)». Типове рішення модель предметної області передбачає створення мережі взаємопов'язаних об'єктів, кожен з яких представляє якусь осмислену сутність. Об'єктна модель домену, що охоплює поведінку і дані. Кожне поняття моделі предметної області представлено окремим класом, можлива реалізація складних взаємозв'язків між об'єктами.

Об'єктно-орієнтована модель предметної області часто нагадує схему відповідної бази даних, однак в моделі предметної області змішуються дані і функції, допускаються багатозначні атрибути, створюються складні мережі асоціацій і використовуються зв'язку успадкування.

У сфері КПС можна виділити два різновиди моделей предметної області.

"Проста" багато в чому схожий на схему бази даних і містить, як правило, по одному об'єкту домена у розрахунку на кожну таблицю. "Складна" модель може відрізнятися від структури бази даних і містити ієрархії успадкування, стратегії та інші типові рішення, а також складні мережі дрібних взаємопов'язаних об'єктів. Складна модель більш адекватно представляє заплутану бізнес-логіку, але важче піддається відображенню в реляційну схему бази даних.

Переваги:

- можливість реалізації складної бізнес-логіки;
- простота тестування;
- висока ступінь повторного використання.

Недоліки:

- складність забезпечення взаємодії з базою даних.

Якщо логіка додатка проста, модель предметної області не завжди доцільна до застосування, оскільки витрати на її реалізацію не окупаються. Але із зростанням складності альтернативні підходи стають все менш прийнятними: трудомісткість поповнення додатки новими функціями збільшується відповідно до експоненціальним законом.

Ефективність модуля таблиці в значній мірі залежить від рівня підтримки структури безлічі записів в конкретної інструментальної середовищі. Підхід не застосовний для реалізації складної бізнес-логіки.

Домен - практично кращий вибір в будь-якій ситуації за винятком дуже простих додатків або при наявності спеціальних інструментальних засобів, орієнтованих на модуль таблиць.

2.2.3 Шар сервісів в архітектурі застосування

Шар сервісів визначає межі застосування та безліч операцій, що надаються для інтерфейсних клієнтських шарів коду. Він інкапсулює бізнес-логіку додатка, управляє транзакціями та координує реакції на дії.

Мається на увазі рознесення бізнес-логіки за двома категоріями: логіка домену (domain logic) - працює з предметною областю (прикладом можуть служити стратегії нарахування відсотків за депозитом і т.д.) і логіка застосування (application logic) - визначає сферу відповідальності застосування (наприклад, повідомлення користувачів і сторонніх додатків про виконання будь-якого процесу). Логіку додатку часто називають також "логікою робочого процесу".

Розміщення логіки додатка в "чистих" класах домену небажано. По-перше, класи домену допускають меншу ймовірність повторного використання, якщо вони реалізують специфічну логіку програми та залежать від тих чи інших прикладних інструментальних пакетів. По-друге, змішування логіки обох категорій в контексті одних і тих же класів ускладнює можливість нової реалізації логіки додатка. З цих причин шар служб передбачає розподілення "різної" логіки по окремих шарах, що забезпечує тра-

диційні переваги розшарування, а також велику ступінь свободи застосування класів домену в різних додатках.

Перевагою використання шару служб є можливість визначення набору спільних операцій, доступних для застосування багатьма категоріями клієнтів, і координація відгуків додатку на виконання кожної операції. У складних випадках відгуки можуть включати в себе логіку додатка, передану в рамках атомарних транзакцій з використанням декількох ресурсів. Таким чином, якщо у бізнес-логіки додатка є більше однієї категорії клієнтів, а відгуки на варіанти використання передаються через кілька ресурсів транзакцій, використання шару служб з транзакціями, керованими на рівні контейнера, стає необхідним, навіть якщо архітектура програми не є розподіленою.

Шар служб не потрібно використовувати, якщо:

– у логіки додатка є тільки одна категорія клієнтів;

– бізнес-логіка реалізована як сценарій транзакції;

– логіка дуже проста.

2.2.4 Типові рішення реалізації шару сервісів

Шаблон «Інтерфейс доступу до домену». Шар служб представлений набором "тонких" інтерфейсів, розміщених "поверх" моделі предметної області. У класах, що реалізують інтерфейси, ніяка бізнес-логіка відображення не знаходить – вона зосереджена виключно в контексті моделі предметної області. Тонкі інтерфейси встановлюють границі і визначають безліч операцій, за допомогою яких клієнтські шари взаємодіють з додатком, виявляючи тим самим характерні властивості шару служб.

Приклад реалізації:

```
interface IStudentService {
function void moveStudent(Student s, Group g);
}

class StudentServiceImpl {
     function void moveStudent(Student s, Group g) {
          s.moveStudent(g);
     }
}
```

Шаблон «Сценарій операції». Шар служб реалізується як безліч більш "товстих" класів, які безпосередньо включають в собі логіку додатка, але за бізнес-логікою звертаються до класів домену. Операції, що надаються клієнтам шару служб, реалізуються у вигляді сценаріїв, що створюються групами в контексті класів, кожний з яких визначає певну частину відповідної логіки. Подібні класи формують "служби" додатка.

Приклад:

```
class StudentServiceImpl {
     function void moveStudent(Student s, Group g) {
          Transactio tx = new Transaction();
          tx.start();
          // Найти текущую группу
```

```
            // Удалить студента
            // Добавить в g
            tx.commit();
        }
}
```

В назвах класів сервісів прийнято використовувати суфікс "Service".

2.2.5 Визначення необхідних служб та операцій

У якості відправної точки для визначення набору операцій шару служб слід розглядати модель варіантів використання і призначений для користувача інтерфейс програми - кожному варіанту використання системи повинен бути поставлений у відповідність метод класу шару сервісів.

Як правило, більшість варіантів використання корпоративних застосувань складають CRUD-операції над об'єктами домену (create, read, update, delete - створити, зчитати, оновити, видалити). Створення, оновлення або видалення в об'єкта домену, а також перевірка правильності його вмісту, вимагає відправки повідомлень користувачам або іншим інтегрованим застосуванням. Координацією відгуків і відправленням їх в рамках атомарних транзакцій займаються операції шару служб.

Існують різні підходи до формування абстракцій шару служб додатку. Невеликому додатку цілком може вистачити однієї абстракції. Більші програми зазвичай розбиваються на кілька підсистем, кожна з яких включає в себе вертикальний зріз всіх наявних архітектурних шарів. В цьому випадку створюють по одній однойменній абстракції для кожної підсистеми.

Можливе створення абстракцій, що відображають основні складові моделі предметної області, якщо такі відрізняються від підсистем (наприклад, StudentService, CourseService) або абстракцій, названих відповідно до типів поведінки (наприклад, AddService, UpdateService).

2.2.6 Віддалений доступ. Шаблон «Об’єкт переносу даних»

Інтерфейс будь-якого класу шару служб найчастіше володіє низьким ступенем деталізації, оскільки в ньому оголошується набір прикладних операцій, відкритих для зовнішніх інтерфейсних клієнтських шарів. Тому класи шару служб добре застосовані для віддаленого доступу.

Рекомендується організація локального доступу до сервісів додатку, включення можливості віддаленого виклику тільки при необхідності, розміщуючи "поверх" шару служб відповідні інтерфейси віддаленого доступу (Remote Facade).

Кожне звернення до інтерфейсу віддаленого доступу пов'язано з великими витратами. Тому клієнтові необхідно мінімізувати число віддалених викликів, а отже, кожен виклик повинен повертати якомога більше інформації. Можна використовувати об'єкт перенесення даних, який буде містити в собі всі дані, які повертаються клієнтові за один виклик.

Об'єкт перенесення даних (Data Transfer Object) застосовується для перенесення даних між процесами з метою зменшення кількості віддалених викликів (рисунок 2.1).

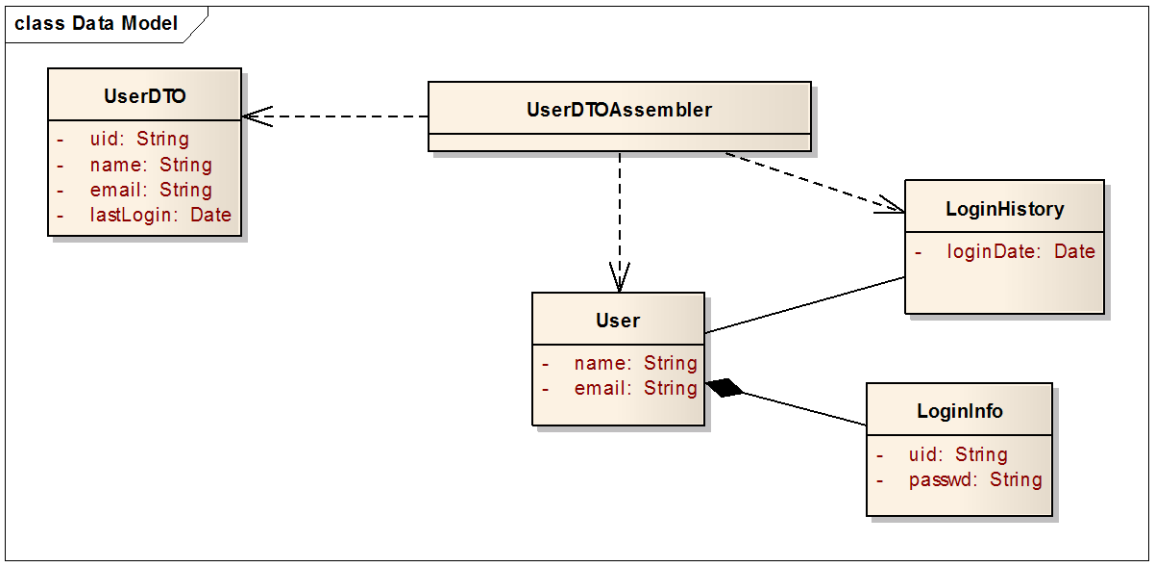

Рисунок 2.1 – Формування об’єкту переносу даних із об’єктів домену

Для переміщення даних між об'єктом переносу даних і об'єктами домену, як правило, застосовується об'єкт-зборщик, розташований на стороні сервера. Об'єкт перенесення повинен містити в собі всі дані, які можуть знадобитися віддаленому об'єкту через деякий час. Оскільки віддалені виклики пов'язані з великими витратами, краще відправити більше даних, ніж виконувати більшу кількість викликів. Найчастіше об'єкт перенесення даних містить набагато більше інформації, ніж звичайний серверний об'єкт. Таким чином, якщо віддалений об'єкт запросить дані про об'єкт замовлення, об'єкт перенесення даних поверне дані про замовлення, покупця, пунктах замовлення, умови доставки - інакше кажучи все, що мало відношення до запитуваного замовлення.

Об'єкт перенесення даних не може передавати об'єкти з моделі предметної області. Як правило, ці об'єкти пов'язані між собою складною системою відносин, яку складно серіалізувати. Крім того, об'єкти домену взагалі не слід розміщати на стороні клієнта. Тому для перенесення даних зазвичай використовують кілька спрощену форму об'єктів домену. Поля об'єктів перенесення даних досить прості. Зазвичай вони містять значення стандартних типів на зразок рядків або дат, а також інші об'єкти перенесення даних. Будь-які зв'язки між такими об'єктами повинні укладатися в рамки простого графа, як правило, ієрархії, на противагу складним структурам, які можна спостерігати в моделі предметної області.

Крім підтримки серіалізації, атрибути об'єктів перенесення даних повинні "розпізнаватися" і передавальною і приймаючою стороною. Тому класи об'єктів перенесення даних, а також класи, на які вони посилаються, мають бути присутні на обох кінцях з'єднання.

Виникає питання: чи потрібно використовувати загальний об'єкт перенесення даних для обробки всіх наявних взаємодій або слід створювати

окремі об'єкти для кожного запиту? Використання різних об'єктів перенесення даних дозволяє простіше відстежити, які дані передаються в кожному виклику, однак вимагає створення безлічі об'єктів. Рекомендується використовувати один об'єкт перенесення даних, особливо якщо передані дані містять багато спільного, однак можливе використання декількох об'єктів для конкретних запитів при необхідності.

Схоже питання пов'язане з тим, чи потрібно використовувати загальний об'єкт перенесення даних для запитів і відповідей або ж слід скористатися окремими об'єктами для кожного з них. Загального правила немає. Якщо дані, що передаються в запиті і у відповіді, схожі, доцільно використовувати один об'єкт перенесення даних. Якщо ж дані занадто різні, використовують два об'єкти. Можливе використання «незмінних» об'єктів перенесення даних. У цьому випадку додаток отримує від клієнта один екземпляр об'єкта перенесення даних, а повертає інший, навіть якщо вони належать одному і тому ж класу. Забезпечення можливості зміни об'єкта перенесення даних, дозволяє наповнювати об'єкт даними, навіть якщо в якості відповіді на запит буде створений новий об'єкт.

2.2.7 Шаблон проектування «Стратегія»

Стратегія (Strategy) - поведінковий шаблон проектування, призначений для визначення сімейства алгоритмів, інкапсуляції кожного з них і забезпечення їх взаємозаміни. Це дозволяє обирати алгоритм шляхом визначення відповідного класу. Шаблон Strategy дозволяє змінювати обраний алгоритм незалежно від об'єктів-клієнтів, які його використовують. Передбачає відділення процедури вибору алгоритму від його реалізації. Це дозволяє зробити вибір на підставі контексту.

Приклад застосування патерну «Стратегія» наведено для реалізації різних депозитних програм і відповідних схем розрахунку процентних ставок залежно від типу вкладу і характеристик клієнта (рисунок 2.2).

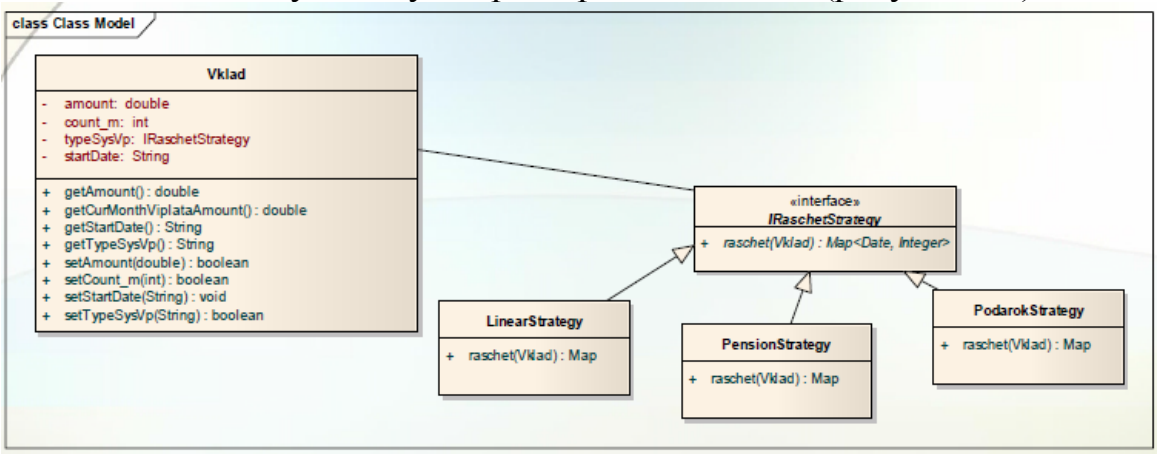

Рисунок 2.2 – Приклад реалізації патерну «Стратегія»

2.2.8 Шаблок «Супертип шару»

Часто виникає ситуація, коли одні і ті ж властивості і методи дублюються в усіх об'єктах шару, наприклад, збереження і обробка полів

ідентифікації. З метою усунення дублювання коду шаблон «супертіп шару» передбачає створення суперкласу для всіх об'єктів шару. При необхідності можлива реалізація декількох супертипів шару.

Приклад реалізації паттерна «супертипу шару» для доменних об'єктів наведено на рисунку 2.3.

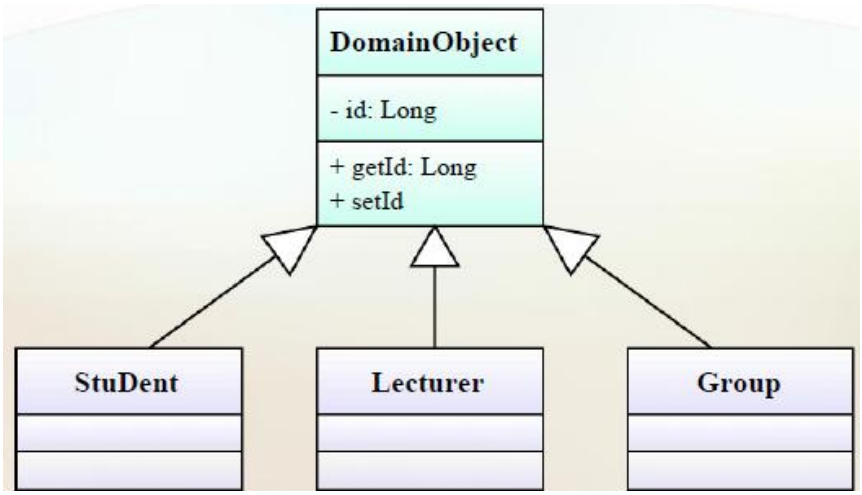

Рисунок 2.3 – Приклад використання шаблону супертип шару

2.3 Порядок виконання роботи

1. Провести аналіз предметної області, визначити групи користувачів програми та їх функціональні можливості, визначити функціональні вимоги до системи. Результати проведення аналізу необхідно представити у вигляді UML-діаграм (Use Case діаграми, діаграма сутностей) з описом.

2. Вивчити типові рішення реалізації бізнес-логіки додатка.

3. Розробити діаграму класів доменних об'єктів.

4. Розробити діаграму інтерфейсів сервісів додатки. Інтерфейси сервісів обов'язково повинні бути визначені з метою забезпечення гнучкості внесення змін.

5. Розробити алгоритми методів, що реалізують бізнес-логіку програми. Діаграми послідовності / дій для опису відповідних алгоритмів.

2.4 Завдання для самостійної роботи

Вивчення структурних шаблонів проектування і їх застосування для реалізації бізнес-логіки додатка.

3 Лабораторна робота №3 Програмна реалізація та тестування компонентів бізнес-логіки застосування

3.1 Мета роботи

Вивчення принципів інверсії контролю; отримання практиче-ських навичок використання технології EJB при реалізації бізнес-логіки додатка.

3.2 Теоретичні відомості

3.2.1 Реалізація об'єктів домену

Доменні об'єкти додатки повинні бути реалізовані як POJO об'єкти (Plain Old Java Object) - простий Java-об'єкт, який не успадкований від будь-якого специфічного об'єкта і не реалізує жодних службових інтерфейсів окрім тих, які потрібні для реалізації бізнес-моделі програми.

На даному етапі розробки питання інтеграції з базою даних (відображення доменних об'єктів і зв'язків між ними в схему бази даних) не розглядаються, відповідно доменні об'єкти не потрібно реалізовувати у вигляді об'єктів сталості (entity).

3.2.2 Реалізація шару сервісів з використанням технології Enterprise JavaBeans

Компоненти Enterprise Beans є основою розробки розподілених корпоративних JavaEE додатків. EJB компонент представляє програмний модуль, який самостійно або взаємодіючи з іншими EJB-компонентами, реалізує і виконує бізнес-логіку додатка на стороні Java EE сервера. З точки зору програмування, Enterprise JavaBeans являють собою спеціальним чином анотовані Java-класи. При використанні EJB вище 3 версії не потрібно спадкування будь-яких додаткових базових класів.

Для виконання EJB-компонентів потрібна наявність оточення EJB-контейнера. EJB-контейнер реалізує сервіси системного рівня, такі як управління транзакціями і управління безпекою, що спрощує процес розробки розподілених додатки. При цьому реалізація низькорівневих сервісів залишається прозорою для розробника. Використання EJB-технологій при розробці програмних систем доцільно в наступних випадках.

1. Додаток має бути масштабованим, тобто повинна забезпечуватися можливість розподілу компонентів програми на кілька серверів у разі збільшення навантаження. EJB-компоненти можуть виконуватися на декількох серверах, при цьому їх розташування залишається прозорим для клієнтів.

2. При виконанні транзакцій повинна забезпечуватися цілісність даних. EJB-технології реалізують механізми синхронізації паралельного доступу до поділюваних об'єктів.

3. Додаток має безліч клієнтів. Надаються механізми простого виявлення EJB-компонентів програми.

Виділяють такі типи EJB-компонентів.

1. Сесійна компонент (session bean). Відповідає сесії виконання запиту користувача. При завершенні обробки клієнтського запиту, сесійний компонент і його дані видаляються.

2. Компонент, орієнтований на обробку повідомлень. Реалізують асинхронну обробку запитів додатка найчастіше у вигляді JMS повідомлень. Діє як слухач JMS-повідомлень. Клієнт відправляє запит за допомогою відправки JMS повідомлення до приймаючого об'єкт (чергу), для якого MDB-компонент є слухачем. Повідомлення можуть бути відправлені будь-яким Java EE компонентом (клієнтську програму, інший EJB-компонент або веб-компонент), JMS-додатком або системою, не використовує Java EE. MDB-компонент не містить даних і інформації про сесію певного клієнта, всі сутності компонентів еквівалентні, що дозволяє контейнеру призначати обробку повідомлення будь-якого з компонентів. Контейнер створює пул об'єктів для одночасної обробки потоків повідомлень.

Виділяють такі типи сесійних компонентів: зберігають стан сесії (stateful), що не зберігають стану (stateless) і singleton-компоненти. Для визначення EJB-компонента певного типу використовуються відповідні анотації @Stateless, @Stateful, @Singleton.

Компоненти без стану є найбільш використовуваним варіантом реалізації сеансових компонентів розподіленого додатка. Як випливає з назви, особливістю таких компонентів є відсутність зберігається стану. Поняття стан об'єкта тісно пов'язане з активізацією і пулами об'єктів. З метою підвищення ефективності використання ресурсів сервера контейнер EJB виконує автоматичну активізацію об'єктів Session Beans. Активізація означає, що об'єкт Session Bean не завжди існує в пам'яті сервера, а може бути завантажений при запиті клієнта і вивантажено в разі відсутності потреби в даному об'єкті. Активізація EJB виконується повністю автоматично і не вимагає участі розробника. Вивантаження об'єкта має на увазі втрату значень всіх атрибутів об'єкта. Для stateless об'єктів може бути створений пул об'єктів, що містить деяку кількість заздалегідь створених контейнером об'єктів одного типу. Оскільки всі об'єкти без стану абсолютно ідентичні, то для обробки запиту клієнта контейнер може вибрати будь-який з вільних об'єктів в пулі. Після обробки запиту об'єкт повертається назад в пул. Розмір пулу управляється контейнером (на підставі налаштувань адміністратора) в залежності від поточної завантаженості сервера. Подібний підхід дозволяє підвищити масштабованість корпоративних додатків.

При необхідності збереження стану сесії користувача (наприклад, вмісту кошика покупця) і збереження сеансових даних у властивостях об'єкта, використовуються EJB-компоненти, що зберігають сесію. Для таких об'єктів контейнер EJB виконує автоматичне збереження значень атрибутів протягом всього сеансу, крім атрибутів, позначених ключовим словом transient.

У разі оголошення EJB-компонента як Singleton, створюється один екземпляр об'єкта, який існує в процесі всього життєвого циклу програми. Використовується з метою забезпечення одночасного доступу клієнтів додатки до єдиного разделяемому компоненту. При цьому розробником повинні враховуватися аспекти синхронізації одночасного доступу за допомогою керуванням рівнем блокування об'єкта або окремих його властивостей і методів (анотація @Lock) і часом блокування об'єкта (@AccessTimeout).

Існує механізм управління життєвим циклом EJB компонентів. Для визначення дій, які повинні бути виконані на певних етапах життєвого циклу компонента, при оголошенні методу використовуються відповідні анотації @ PostConstruct, @ PreDestroy, @ PostActivate, @ PrePassivate, @Remove.

Існує механізм визначення тимчасових подій для виконання тих чи інших дій, які визначаються методом будь-якого EJB-компонента, крім Stateful. Наприклад:

```
@Schedule(dayOfWeek="Sun", hour="0")
public void cleanupWeekData() { ... }

@Schedules ({
@Schedule(dayOfMonth="Last"),
@Schedule(dayOfWeek="Fri", hour="23")
})
public void doPeriodicCleanup() { ... }
```

Для отримання доступу до EJB¬-компонентів програми може використовуватися принцип інверсії контролю або JNDI-пошук. Для віддалених клієнтів може використовуватися тільки JNDI-пошук.

Рішення щодо типу клієнтського доступу до EJB-компонентів - локальний, віддалений або веб-сервіс, має враховувати такі фактори.

1. Ступінь зв'язності між компонентами програми. Розміщення тісно пов'язаних компонентів додатка усередині одного контейнера дозволить підвищити продуктивність.

2. Тип клієнта. Якщо клієнтом може бути сторонній додаток, віддалений доступ повинен бути дозволений. Якщо клієнтами є веб-компоненти або інші EJB-компоненти, вибір залежить від розподіленості компонентів.

3. Розподіленість компонентів. Необхідністю розподілу компонентів програми на кілька серверів вимагає наявності віддаленого доступу.

Можливий варіант підтримки і локального, і віддаленого доступу. При цьому повинні бути визначені кілька бізнес-інтерфейсів з відповідними анотаціями @Local і @Remote.

У разі організації віддаленого доступу, повинні враховуватися аспекти ізольованості і гранулярності даних. Ізольованість даних передбачає, що віддалений клієнт і EJB-компонент працюють з різними копіями об'єкта-параметра. Подібний принцип ізоляції дозволяє забезпечити захищеність додатки. Гранулярність даних має на увазі використання крупнозернистих

параметрів (coarse-grained) з метою зменшення кількості віддалених викликів.

3.2.3 Типи JavaEE-контейнерів компонентів застосування

Java EE сервер виконує управління життєвим циклом компонентів програми та включає наступні типи контейнерів:

- EJB-контейнер (Enterprise JavaBeans) - управляє виконанням EJB-компонентів додатка;
- Web-контейнер - управляє виконанням веб-сторінок, сервлетів та деяких EJB-компонентів.

Розміщення компонентів програми на сервері зображено на рисунку 3.1.

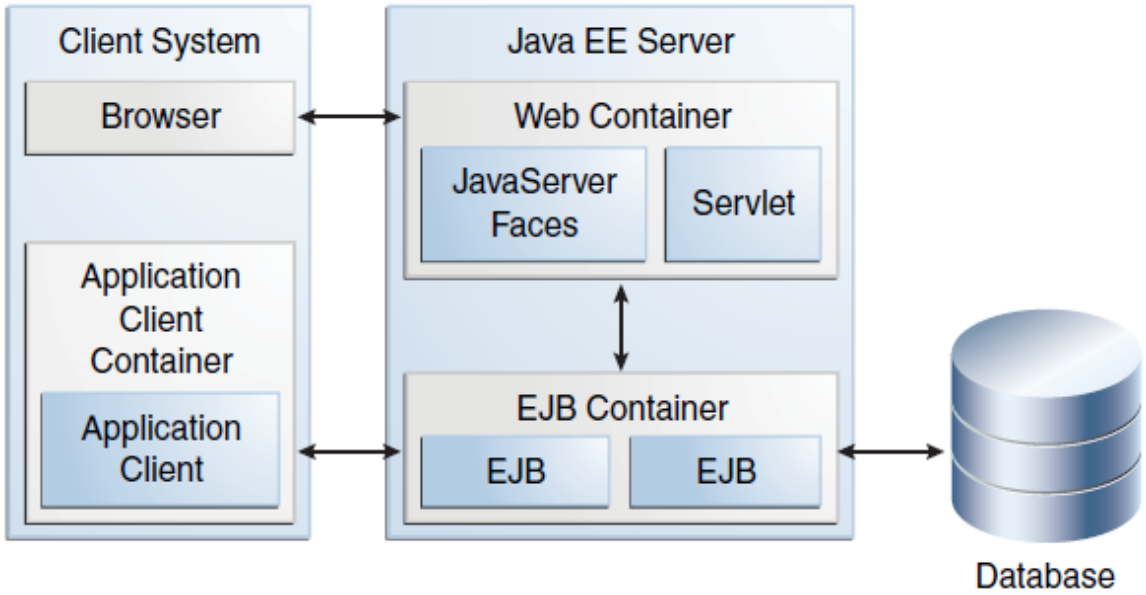

Рисунок 3.1 – Функції сервера застосувань

3.2.4 Понятті інверсії контролю. Реалізація в Java EE 7

Основна ідея патерну «інверсія контролю» (Inversion of Control, IoC) – усунення залежності компонентів додатку від конкретних реалізацій допоміжних інтерфейсів і делегування повноважень з управління створенням потрібних реалізацій IoC контейнеру. Контейнер управляє ЖЦ об'єктів, управляє залежностями.

Таким чином, компонент не створює залежності, а автоматично отримує їх в процесі життєвого циклу.

Переваги використання принципів IoC:

декларативне управління залежностями;

– спрощення повторного використання класів або компонентів при зміні реалізації допоміжних класів;

– забезпечення можливості незалежного модульного тестування компонентів додатка;

– «чистий» код: компоненти не займаються инициализацией допоміжних об'єктів.

Однак не слід керувати створенням абсолютно всіх об'єктів через контейнер IoC. В управління контейнера найкраще виносити ті інтерфейси, реалізація яких може бути змінена.

Існують альтернативні підходи, що дозволяють усунути залежність від реалізації. Альтернативою IoC є добре відомі шаблони ServiceLocator / Factory. Однак дані підходи припускають звернення до фабрики об'єктів для створення допоміжного об'єкта з коду класу компонента. IoC підхід передбачає декларативне опис залежностей за допомогою визначення конфігураційних файлів або анотацій.

Розглянемо приклад використання принципів IoC, реалізованих в рамках специфікації JavaEE 7. Принципи IoC в Java EE (починаючи з 5 версії) реалізовані сервісами CDI (Contexts and Dependency Injection). CDI дозволяє інвертувати різні типи об'єктів, в тому числі в не керував на рівні контейнера об'єкти. Наведений нижче простий Java-клас є компонентом (bean) і може бути інвертований в інший клас:

```
package greetings;
public class Greeting {
     public String greet(String name) {
     return "Hello, " + name + ".";
     }
}
```

Наприклад, наведений нижче сервлет звертається до методів об'єкта, який реалізує інтерфейс Message:

```
@WebServlet("/cdiservlet")
public class NewServlet extends HttpServlet {
     private Message message;
     @Override
     public void init() {
     message = new MessageB();
     }
     @Override
     public     void     doGet(HttpServletRequest     request,
HttpServletResponse response)throws IOException {
          response.getWriter().write(message.get());
     }
}
public interface Message {
     public String get();
}
public class MessageB implements Message {
     public MessageB() { }
     @Override
     public String get() {
     return "message B";
     }
}
```

В даному випадку сервлет самостійно створює необхідний екземпляр об'єкта MessageB.

У разі використання IoC, залежність NewServlet від об'єкта Message може бути визначена наступним чином:

```
@WebServlet("/cdiservlet")
public class NewServlet extends HttpServlet {
     @Inject private Message message;
     @Override
     public void doGet(HttpServletRequest request, HttpS-
ervletResponse response)
     throws IOException {
     response.getWriter().write(message.get());
     }
}
```

У цьому випадку залежність буде встановлена автоматично засобами контейнера в процесі виконання.Якщо в додатку є більше однієї реалізації інтерфейсу Message, CDI надає механізм визначення конкретної необхідної реалізації. Для цього використовується поняття специфікатор (анотація @Qualifier).

При інвертуванні слід враховувати область видимості вкладеного об'єкта (scope).

3.2.5 Особливості тестування класів EJB-компонентів

EJB-компоненти програми виконуються всередині EJB-контейнера, відповідно вони можуть бути протестовані тільки при наявності EJB-контейнера. Нижче наведено приклад реалізації Unit-тестів сесійного EJB-компонента з використанням вбудованого контейнера:

```
     @Stateless
     @Local
     public class UserFacade extends AbstractFacade<User> im-
plements IUserFacade {

         @PersistenceContext(name = "IASBP-test_PU")
         private EntityManager em;
         …
         }

     public class UserFacadeTest {

         private static EJBContainer container;
         private static IUserFacade instance;

         public UserFacadeTest() {
         }

         @BeforeClass
         public static void setUpClass() throws Exception {
             container                                    =
javax.ejb.embeddable.EJBContainer.createEJBContainer();
             instance              =              (IUserFacade)
contain-
er.getContext().lookup("java:global/classes/UserFacade");
```

```
    }

    @AfterClass
    public static void tearDownClass() throws Exception {
        container.close();
    }
    @Test
    public void testCreate() throws Exception {
        System.out.println("create");
        User entity = new User ("test","test");
        instance.create(entity);
        Long id = entity.getId();
        assertEquals(entity, instance.find(id));
     }
}
```

3.3 Порядок виконання роботи

1. Реалізація доменних об'єктів і бізнес-методів предметної області.
2. Модульне тестування доменних об'єктів.
3. Тестова реалізація інтерфейсів сервісів додатки. Тестова реалізація передбачає імітування нижчого рівня інтеграції за допомогою mock-об'єктів.
4. Модульне тестування логіки додатка.

3.4 Завдання для самостійної роботи

1. Вивчення особливостей реалізації та використання ком-нентов Message-Driven Bean.
2. Вивчення принципів реалізації та використання інверсії контролю в Spring фреймворку.

4 Лабораторна робота №4 Проектування слою інтеграції

4.1 Мета роботи

Изучение и применение типовых подходов к организации слоя интеграции корпоративного программного приложения.

4.2 Теоретичні відомості

4.2.1 Роль слою інтеграції в архітектурі застосування

Роль шару джерела даних полягає в тому, щоб забезпечити можливість взаємодії додатку з різними компонентами інфраструктури (інформаційні ресурси програми, сервіси віддаленого програми і ін.) Для виконання необхідних функцій.

4.2.2 Типові рішення реалізації шару інтеграції

Шаблон «Шлюз таблиці даних». Типове рішення передбачає створення об'єкта, що виконує роль шлюзу до таблиці бази даних і містить методи пошуку, вилучення, вставки, оновлення та видалення даних з таблиці або подання. Методи шлюзу таблиці даних використовуються іншими компонентами програми для взаємодії з базою даних. Об'єкт, що виконує роль шлюзу до таблиці, не має станів, його призначення полягає в передачі даних з / в базу даних. Передбачається, що методи пошуку містять множинні результати.

Підхід пропонує найбільш простий варіант організації доступу до бази даних і є найкращим рішенням у випадку реалізації бізнес-логіки у вигляді «Модуля таблиці».

Шаблон «Шлюз запису даних». Об'єкт, що виконує роль шлюзу до окремого запису джерела даних. Кожному рядку джерела даних відповідає свій екземпляр об'єкта.

Оскільки екземпляр об'єкта, що реалізує шлюз записи даних, відповідає одному запису таблиці бази даних, для організації методів пошуку рекомендується створювати окремі класи. Таким чином, кожній таблиці реляційної бази даних ставиться у відповідність клас, що містить методи для виконання операцій вибірки даних і клас шлюзу записи даних, що містить поля, відповідні атрибутам записи таблиці бази даних і методи додавання, модифікації і видалення. Шлюз записи даних містить тільки логіку доступу до даних і не реалізує бізнес-логіку обробки даних, можливе виконання перетворення даних. У разі організації бізнес-логіки у вигляді «Сценарію транзакції» є ідеальним рішенням.

Шаблон «Активна запис». Об'єкт, що виконує роль оболонки для запису таблиці бази даних. Об'єкт інкапсулює доступ до бази даних і додає до даних логіку домену.

Згідно шаблоном «Активна запис» до функцій об'єкта, що виконує роль шлюзу записи даних, додається реалізація бізнес-логіки додатка. Можливий варіант реалізації всієї бізнес-логіки додатка на рівні компонентів шару інтеграції. У цьому випадку шар інтеграції фактично відсутня.

Структура даних активної записи повинна в точності відповідати структурі таблиці бази даних: кожне поле об'єкта має відповідати одному стовпцю таблиці. Значення полів слід залишати такими ж, якими вони були отримані в результаті виконання SQL-команд; ніякого перетворення на цьому етапі робити не потрібно.

Як правило, типове рішення активний запис включає в себе методи для виконання наступних операцій:

- створення екземпляра активної записи на основі рядка, отриманої в результаті виконання SQL-запиту;
- створення нового екземпляра активної записи для подальшої вставки в таблицю;
- статичні методи пошуку, які виконують стандартні SQL-запити і повертають активні записи;
- оновлення бази даних і вставка в неї даних з активної записи;
- витягування та установлення значень полів (get- і set-методи);
- реалізація деяких фрагментів бізнес-логіки.

Активна запис добре підходить для реалізації не надто складною логіки домена, зокрема операцій створення, зчитування, оновлення та видалення, а також виконання валідації даних.

Шаблон «Перетворювач даних» (Data mapper). Типове рішення перетворювач даних являє собою шар програмного забезпечення, який здійснює передачу даних між об'єктами і базою даних, зберігаючи останніх незалежними один від одного і від самого перетворювача.

Модель об'єктів предметної області і реляційні СУБД використовують різні механізми структурування даних. У реляційних базах даних не відображаються багато характеристик об'єктів, зокрема колекції і успадкування. При побудові об'єктної моделі з великим об'ємом бізнес-логіки механізми ООП дозволяють краще організувати дані і соот¬ветствующее їм поведінку. З іншого боку, використання подібних механізмів призводить до розбіжностей об'єктної і реляційної схем. Останнє значно ускладнює завдання взаємодії та обміну даними між об'єктною моделлю і реляційної СУБД.

Типове рішення «перетворювач даних» дозволяє забезпечити повну ізоляцію об'єктів предметної області та бази даних програми - об'єкти, розташовані в оперативній пам'яті, можуть навіть не «підозрювати» про сам факт присутності бази даних. Їм не потрібен SQL-інтерфейс і тим більше схема бази даних. У свою чергу, схема бази даних не «знає» про об'єкти, які її використовують.

Більш того, перетворювач даних повинен бути повністю прихований від рівня домену. Слід уникати залежності об'єктів домену від перетворю-

вача, отже, уникати використання і викликів перетворювача з об'єктів домену.

Реалізація шару інтеграції у вигляді «перетворювача даних» складних і ресурсномістких через складність взаємного відображення об'єктної і реляційної схем.

4.2.3 Об’єктно-реляційне відображення

Об'єктно-релляціонное відображення - це автоматичне і прозоре збереження і отримання об'єктів програми в таблиці релляціонной бази даних, використовуючи мета дані, які описують відображення між об'єктами і відносинами.

Будь-яке ORM рішення складається з 4-х компонент:

- API для виконання CRUD операцій;
- мова або API для виконання запитів;
- механізм визначення мета даних відображень;
- техніки для транзакцій, завантаження по вимоги, і інших оптимізацій.

Застосування ORM технологій дозволяє зосередитися на реалізації складної бізнес-логіки з використанням принципів ООП, що не реалізуючи механізми відображення об'єктів. Оскільки доменні об'єкти в разі використання ORM є «чистими», при необхідності зміни бази даних вносити зміни в шар бізнес-логіки не доводиться.

JPA (Java Persistence API) є стандартом об'єктно-реляційного відображення згідно специфікації Java EE. Існують різні реалізації даного стандарту (Hibernate, EclipseLink, Apache OpenJPA, ін.).

4.2.4 Поняття транзакції. Керування транзакціями

Під транзакцією розуміють обмежену послідовність дій з явно певними початковій і завершальній операціями, яка або виконується цілком, або не виконується зовсім.

Виділяють наступні властивості транзакцій:

- atomicity (атомарность) - в контексті транзакції або виконуються всі дії, або не виконується жодна з них;
- consistency (узгодженість) - системні ресурси повинні перебувати в цілісному і несуперечливою стані як до початку транзакції, так і після її закінчення;
- isolation (ізольованість) - проміжні результати транзакції повинні бути закриті для доступу з боку будь-якої іншої діючої транзакції до моменту їх фіксації;
- durability (стійкість) - результат виконання завершеної транзакції не повинен бути втрачений ні за яких умов.

Виділяють 4 рівня ізоляції транзакцій. Більш високий рівень ізоляції транзакцій забезпечує більшу ймовірність відсутності конфліктних ситуацій при виконанні паралельних транзакцій (аномалії транзакцій), однак при цьому можуть погіршуватися параметри продуктивності додатка. У табли-

ці 4.1 наведені можливі аномалії транзакцій, що відповідають певному рівню ізоляції.

Таблиця 4.1 – Рівні ізоляції транзкацій

Аномалії	Рівень ізоляцій			
	Читання нефіксованих даних	Читання фіксованих даних	Повторюване читання	Упорядковане читання
Втрачені зміни	-	-	-	-
Грязне читання	+	-	-	-
Неповторюване читання	+	+	-	-
Фантомна вставка	+	+	+	-

Слід застосовувати різні рівні блокування для різних ресурсів програми в залежності від специфіки предметної області, тобто при необхідності встановлювати різні рівні ізоляції для різних таблиць додатка. Наприклад, аномалію «брудне читання» можна дозволити для ресурсів програми, читання яких виконується дуже рідко.

При управлінні транзакціями корпоративних додатків під транзакцією частіше мають на увазі бізнес-транзакцію - набір операцій користувача, які виконують різні дії згідно з визначеним сценарієм, що включають, можливо, кілька системних транзакцій. При цьому всі дії бізнес-транзакції розглядаються як «єдина» транзакція, а бізнес-транзакція повинна володіти всіма властивостями системної транзакції.

4.3 Порядок виконання роботи

1. Вивчити типові рішення реалізації шару інтеграції корпоративних додатків.
2. Провести вибір сервера баз даних програми.
3. Провести вибір ORM-провайдера.

4.4 Завдання для самостійної роботи

Вивчити можливі аномалії транзакцій і відповідні рівні ізоляції.

5 Лабораторна робота №5 Програмна реалізація та тестування компонентів слою інтеграції

5.1 Мета роботи

Вивчення стандарту JPA; отримання практичних навичок використання при реалізації компонентів шару інтеграції додатку.

5.2 Теоретичні відомості

5.2.1 Поняття класу сутності (Entity)

Згідно Java EE сутності описуються звичайними Java класами (entity classes). При цьому класи сутностей мають такі особливості:

- клас повинен мати анотацію @Entity;
- клас повинен мати public або protected конструктор без параметрів (можлива наявність інших конструкторів);
- класом сутності не може бути enum або interface;
- ні клас сутності, ні будь-який метод або атрибут класу, що зберігається, не повинен бути оголошений як final;
- якщо передбачається передача об'єктів класу сутності за значенням (наприклад, через віддалений інтерфейс), то клас повинен реалізовувати інтерфейс Serializable.

У всьому іншому класи сутностей нічим не відрізняються від звичайних Java класів. Сутнісні класи допускають спадкування і поліморфізм, при цьому клас сутності може бути нащадком звичайного класу або іншого класу сутності. Звичайний Java клас також може бути нащадком класу сутності.

Стан сутності, що зберігається, складається з полів і властивостей. Полем вважається прихована (private, protected або package) змінна-член сутнісного класу. Властивість (property) - це комбінація з (зазвичай прихованої) змінної члена класу і відкритих методів доступу до неї:

```
private T propertyValue;
public T getProperty() {return propertyValue;}
public void setProperty(T t) {propertyValue = t;}
```

Зберігання підлягають всі поля і властивості, не зазначені аннотацією @Transient. Збережені поля можуть мати (необов'язкову) анотацію. Місце розміщення анотації вказує на тип доступу: якщо анотація розташована перед описом змінної-члена класу, то використовується доступ через цю змінну; якщо анотація розташована перед описом методу getProperty (), то доступ здійснюється через властивість.

Приклад опису класу сутності:

```
@Entity
public class Customer implements Serializable {
private Long id;
private String name;
```

```
private Address address;
// No-arg constructor
public Customer() {}
@Id // property access is used
public Long getId() {
return id;
}
public void setId(Long id) {
this.id = id;
}
public String getName() {
return name;
}
public void setName(String name) {
this.name = name;
}
public Address getAddress() {
return address;
}
public void setAddress(Address address) {
this.address = address;
}
}
```

При відображенні класу сутності, визначеного вище, за замовчуванням буде створена одна таблиця, в якій кожному полю або властивості класу відповідає окремий стовпець. Поле, відзначений-ве анотацією @Id, визначає первинний ключ таблиці.

Для визначення автоматичної генерації значення первинного ключа використовується анотація @GeneratedValue. Існує можливість визначення різних стратегій генерації значення первинного ключа, які визначаються за допомогою параметра анотації @GeneratedValue (strategy): strategy = GenerationType.AUTO, strategy = GenerationType.IDENTITY, strategy = GenerationType.SEQUENCE, strategy = GenerationType.TABLE.

5.2.2 Стратегії відображення наслідування в JPA

Класи сутностей підтримують успадкування, поліморфні зв'язку і поліморфні запити. Класи сутностей можуть успадковувати звичайні класи і звичайний (non-entity) клас може успадковувати клас сутності. Клас сутності може бути абстрактним. Запити можуть формуватися до абстрактних сутностей, при цьому запит буде виконаний над усіма конкретними класами, що наслідують абстрактний.

Приклад оголошення абстрактної сутності:

```
@Entity
public abstract class Employee {
@Id
protected Integer employeeId;
...
}
@Entity
public class FullTimeEmployee extends Employee {
```

```
protected Integer salary;
...
}
@Entity
public class PartTimeEmployee extends Employee {
protected Float hourlyWage;
}
```

Клас сутності може успадковувати суперклас (абстрактний або конкретний), який має збережене стан і метадані відображення, але не є сутністю (не відображається в таблицю бази даних, MappedSuperclass). Такий підхід найчастіше використовується, коли є стан і метадані відображення, загальні для багатьох класів сутностей. Формування запитів до подібних суперкласів не допускається.

Приклад оголошення:

```
@MappedSuperclass
public class Employee {
@Id
protected Integer employeeId;
...
}
@Entity
public class FullTimeEmployee extends Employee {
protected Integer salary;
...
}
@Entity
public class PartTimeEmployee extends Employee {
protected Float hourlyWage;
...
}
```

Клас сутності може успадковувати клас, який не є сутністю. Базовий клас може бути оголошений як абстрактний або конкретний. Поля базового класу не зберігаються, і будь-яку властивість, успадковане класом суті від базового класу, не є його записала. Базовий клас не може використовуватися при формуванні запитів.

Правила відображення успадкування визначаються за допомогою анотації javax.persistence.Inheritance над базовим класом. Існують такі стратегії відображення успадкування:

1) одна таблиця для ієрархії класів @Inheritance (strategy = InheritanceType.SINGLE_TABLE), використовується за умовчанням);

2) таблиця для кожного підкласу @Inheritance (strategy = InheritanceType.JOINED);

3) таблиця для конкретного класу @Inheritance (strategy = InheritanceType.TABLE_PER_CLASS).

Для структури успадкованих класів, наведеної на малюнку 5.1, перша стратегія (SINGLE_TABLE) передбачає відображення всіх класів ієрархії в одну таблицю, яка містить додатковий атрибут - discriminator column,

що визначає приналежність конкретної записи таблиці того чи іншого підкласу (рисунок 5.2).

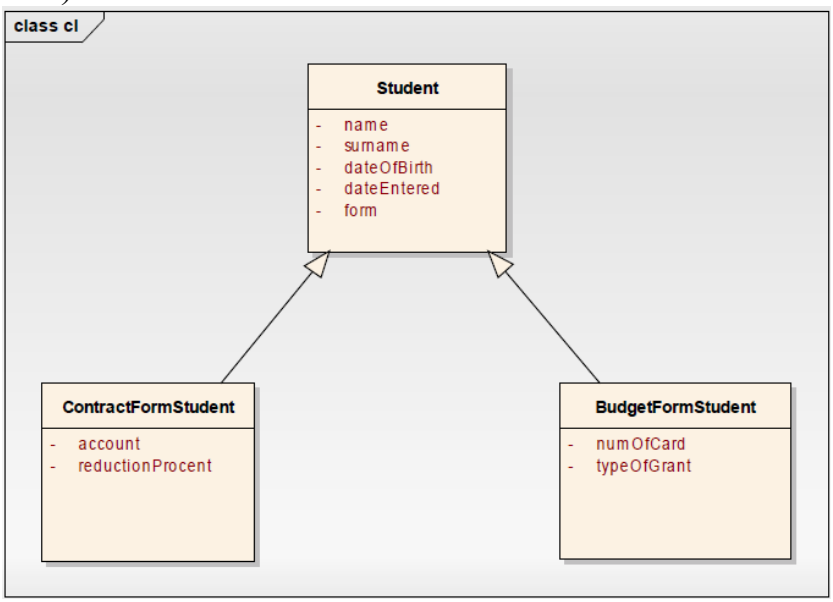

Рисунок 5.1 – Ієрархія класів

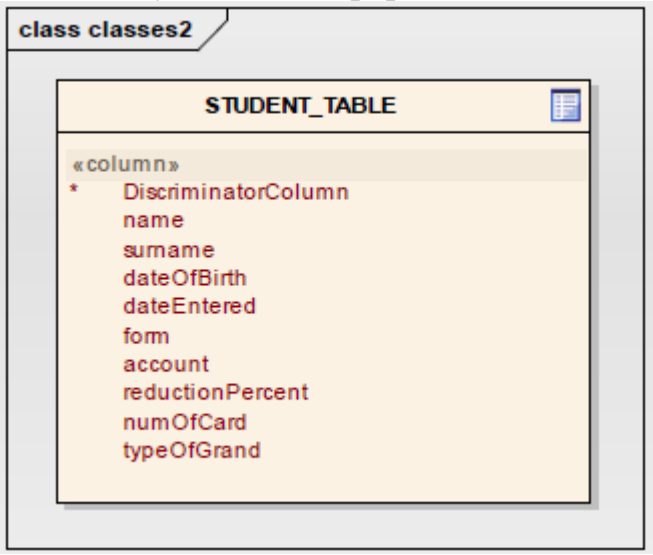

Рисунок 5.2 – Відображення наслідування згідно стратегії SINGLE_TABLE

Тип і принцип формування атрибуту ідентифікації визначається за допомогою анотації javax.persistence.DiscriminatorColumn над базовим класом ієрархії. Анотація javax.persistence.DiscriminatorValue використовується для визначення значення атрибута ідентифікації для об'єктів конкретного підкласу.

Переваги стратегії SINGLE_TABLE:

- в структуру бази даних додається тільки одна таблиця;
- для отримання даних не потрібно виконувати з'єднання;
- зміна полів суперкласів і підкласів не вимагає зміни декількох таблиць.

Недоліки:

- проблема not-null полів (наявність обов'язкового поля одного конкретного класу, яке відсутнє в іншому підкласі);

- проблеми синхронізації одночасно доступу до даних таблиці і блокувань;
- пусте місце в невикористовуваних столбцях.

JOINED-стратегія передбачає створення окремої таблиці бази даних для кожного підкласу (рисунок 5.3).

Переваги JOINED-стратегії:

- простота реалізації поліморфних зв'язків;
- немає "непотрібних" порожніх полей;
- реляційна модель повністю нормалізована.

Недоліки:

- необхідність виконання JOIN-операцій при виконанні запитів, знижує продуктивність системи (в разі складних ієрархій);
- переміщення полів з підкласу в суперклас вимагає зміни структури БД;
- частий доступ до таблиць суперкласів.

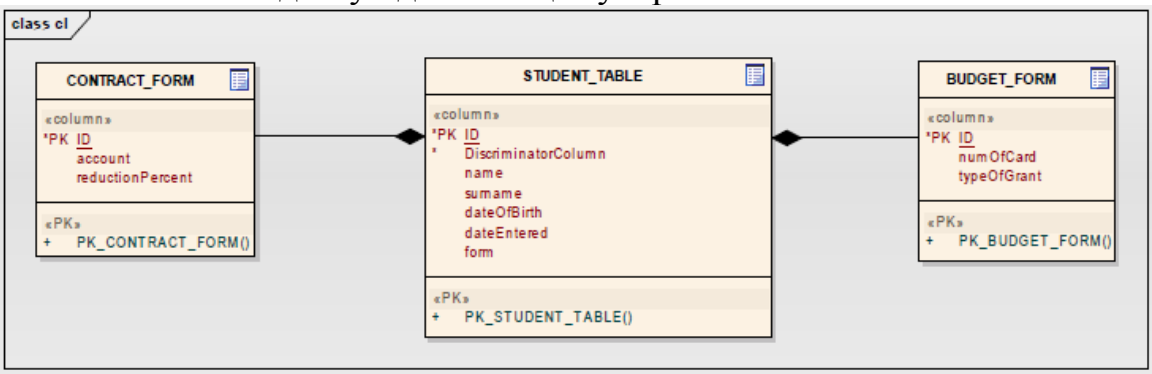

Рисунок 5.3 – Відображення наслідування згідно JOINED-стратегії

Стратегія TABLE_PER_CLASS передбачає створення окремих таблиць для конкретних класів (рисунок 5.4).

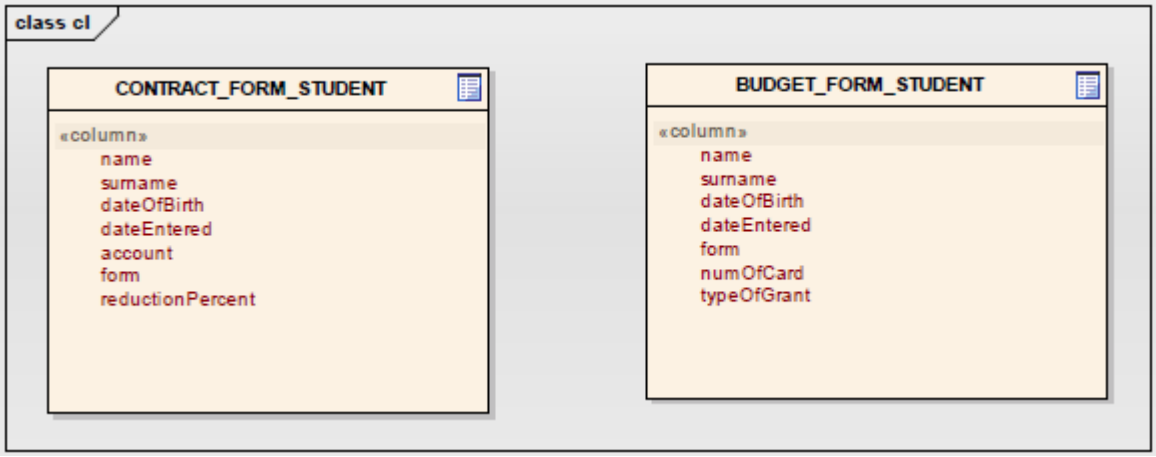

Рисунок 5.4 – Відображення наслідування згідно стратегії TABLE_PER_CLASS

Переваги стратегії TABLE_PER_CLASS:

- кожна таблиця замкнута і не містить непотрібних полів;
- при зчитуванні даних не потрібно виконувати операції з'єднання.

Недоліки:

- погана підтримка поліморфних зв'язків;
- для запитів, що покривають всю ієрархію, потрібна операція UNION або окремі SELECT-запити для кожного підкласу;
- зміна полів суперкласів вимагає зміни декількох таблиць.

5.2.3 Менеджер сутностей JPA (Entity Manager). Підключення в класи шару сервісів застосування

Інтерфейс EntityManager використовується для управління збереженими сутностями (створення, пошук, модифікація, видалення і т.д.) Основними методами інтерфейсу EntityManager є:

- public void persist (Object entity) - зберегти сутність, представлену об'єктом класу суті в БД;
- public void remove (Object entity) - видалити сутність;
- public <T> T find (Class <T> entityClass, Object primaryKey) - визначити сутність по відомому первинному ключу, T - тип сутності;
- public Query createQuery (String qlString) - створити запит БД на мові запитів Java persistence Query Language (JPQL);
- public Query createNamedQuery (String name) - створити заздалегідь визначений (статичний) запит до БД (name - назва визначеного запиту);
- public Query createNativeQuery (String sqlString) - створити запит з використанням мови запитів СУБД (SQL).

Існує поняття контексту зберігання (persistence context) - безліч сутностей, керованих менеджером сутностей. Контекст зберігання і відповідний менеджер сутностей надається контейнером EJB-компоненту за допомогою інверсії контролю або JNDI пошуку.

Виділяють поняття менеджера сутностей, керованого на рівні контейнера і керованого на рівні додатку. В останньому випадку екземпляр менеджера сутностей і пов'язаний з ним контекст зберігання створюється і знищується безпосередньо з коду програми. При цьому кожен примірник менеджера сутностей инициализирует новий, незалежний контекст зберігання. Управління контекстом транзакцій JTA в цьому випадку також має здійснюватися на рівні додатку.

Приклад підключення менеджера сутностей, керованого на рівні контейнера, з використанням механізму інверсії контролю:

```
@Stateless
public class UserService {
     @PersistenceContext(unitName = "unit")
     private EntityManager em;
     ...
}
```

Приклад створення менеджера сутностей, керованого на рівні програми:

```
     @PersistenceUnit
     EntityManagerFactory emf;
     EntityManager em = emf.createEntityManager();
```

Використання модулів зберігання (Persistent Unit) дозволяє логічно згрупувати такі компоненти:

- менеджери сутностей (Entity Managers);
- безліч класів сутностей, керованих менеджерами сущностей;
- метадані (у формі анотацій або XML-метаданих), що визначають правила відображення сутнісних класів в БД.

Модуль зберігання може являти собою окремий JAR-архів або набір класів в складі JAR-архіву компонента додатку-ня. Модуль зберігання має унікальне ім'я і описується XML-дескриптором persistence.xml. Приклад визначення модуля зберігання:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0" xmlns="http://java.sun.com/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
  <persistence-unit name="IASBP-test_PU" transaction-type="JTA">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <jta-data-source>jdbc/sample</jta-data-source>
    <class>entity.UsrTwoTest</class>
    <exclude-unlisted-classes>false</exclude-unlisted-classes>
    <properties>
      <property name="eclipselink.ddl-generation" value="drop-and-create-
tables"/>
    </properties>
  </persistence-unit>
</persistence>
```

5.2.4 Виконання запитів в JPA з використанням JPQL та Criteria API

Для виконання запитів до БД через сервіс зберігання використовується інтерфейс Query. Основні методи інтерфейсу Query:

- public List getResultList () - виконує запит типу SELECT, який повертає безліч сутностей;
- public Object getSingleResult () - виконує запит типу SELECT, який повертає єдину сутність;
- public int executeUpdate () - виконує запит на зміну або видалення сутностей, повертає число було змінено або видалено сутностей;
- public Query setParameter (String name, Object value) - дозволяє задати значення іменованого параметра в параметризрвані запиті;
- public Query setMaxResults (int maxResult) - задати максимальну кількість повертаються сутностей.

Інтерфейс Query може використовуватися як для виконання заздалегідь визначених іменованих запитів, так і для генерації динамічних запитів. Використовується мова опису об'єктних (а не реляційних) запитів Java Persistence Query Language (JPQL), синтаксис мови багато в чому схожий з SQL.

При безпосередньому виконанні запиту вираз на JPQL транслюється в SQL (або іншу мову запитів використовуваної СУБД). Є також можливість безпосередньо задавати вираження на мові використовується (native) СУБД, проте, такий підхід не має переносимість.

Приклад створення динамічного запиту:

```
public List findWithName(String name) {
return em.createQuery(
"SELECT c FROM Customer c WHERE c.name LIKE :custName")
.setParameter("custName", name)
.setMaxResults(10)
.getResultList();
```

Іменовані запити можуть формуватися з використанням метаданих. Для формування іменованих запитів можуть використовуватися JPQL або SQL.

Приклад оголошення іменованого запиту:

```
@NamedQuery(
name="findAllCustomersWithName",
query="SELECT c FROM Customer c WHERE c.name LIKE :custName"
)
```

Приклад подальшого використання іменованого запиту:

```
@PersistenceContext
public EntityManager em;
...
customers = em.createNamedQuery("findAllCustomersWithName")
.setParameter("custName", "Smith")
.getResultList();
```

У разі використання для формування запитів мови JPQL запит подається у вигляді рядка. Наприклад, оголошений наступний JPQL-запит:

```
EntityManager em = ...;
String jpql = "select p from Person where p.age > 20";
Query query = em.createQuery(jpql);
List result = query.getResultList();
```

Незважаючи на те, що JPQL-рядок запиту синтаксично некор-ректно (правильний варіант String jpql = "select p from Person p where p.age> 20";), код буде успішно скомпільовано, згенерує виняток тільки на етапі виконання.

Одним з ключових переваг API Criteria є те, що він дозволяє створювати типобезпечні запити і не дозволяє створювати синтаксично некоректні запити. Приклад визначення того ж запиту з використанням Criteria API:

```
EntityManager em = ...
QueryBuilder qb = em.getQueryBuilder();
CriteriaQuery<Person> c = qb.createQuery(Person.class);
Root<Person> p = c.from(Person.class);
Predicate condition = qb.gt(p.get(Person_.age), 20);
c.where(condition);
TypedQuery<Person> q = em.createQuery(c);
List<Person> result = q.getResultList();
```

Criteria-запити вимагають реалізації метамоделі - класу, що визначає метаінформацію для класу сутності. Наприклад, класу сутності Person, відповідає клас метамоделі Person_:

```
package domain;
@Entity
public class Person {
  @Id
  private long id;
  private string name;
  private int age;
  …
```

```
}

package domain;
import javax.persistence.metamodel.SingularAttribute;

@javax.persistence.metamodel.StaticMetamodel(domain.Person.class)

public class Person_ {
  public static volatile SingularAttribute<Person,Long> ssn;
  public static volatile SingularAttribute<Person,String> name;
  public static volatile SingularAttribute<Person,Integer> age;
}
```

Все зберігаються атрибути класу domain.Person присутні в класі метамоделі в вигляді відкритих статичних змінних-членів типу SingularAttribute <Person,?>. Завдяки цьому можна посилатися на атрибути domain.Person (наприклад, age), використовуючи відповідні статичні змінні, зокрема, Person_.age. В цьому випадку компілятор може перевірити сумісність типів, оскільки він знає оголошений тип атрибута age.

Механізм визначення метаінформації є альтернативним по відношенню до механізму рефлексії Java (Java Reflection API). При цьому між ними є принципова відмінність: звернення до метаданих об'єкта Person.class, отриманим за допомогою рефлексії, не може контролюватися компілятором. Наприклад, механізм рефлексії дозволяє звернутися до поля age в Person.class таким чином:

Field field = Person.class.getField("age");

Однак подібний підхід має ті ж недоліки, що і використання строкових запитів на мові JPQL. Компіляція цього фрагмента коду проходить успішно, проте компілятор не може гарантувати, що при його виконанні не виникне помилок.

5.2.5 Керування областю дії транзакції в JPA

Управління кордонами транзакцій може виконуватися на рівні контейнера (Container-Managed Transaction) або за допомогою прямого звернення до методів класу javax.transaction.UserTransaction з коду EJB-компонента (Bean-Managed Transaction).

За замовчуванням, контейнер відкриває нову транзакцію перед стартом кожного методу EJB-компонента, і закриває відповідно після завершення методу, перед виходом з нього. Отже, кожен методу EJB-компонента за замовчуванням виконується в рамках однієї транзакції. Вкладені і множинні транзакції усередині методу не допускаються.

При зверненні до методу EJB-компонента з іншого методу EJB-компонента (рисунок 5.5) в JPA використовується схема визначення меж транзакції в залежності від встановленого атрибуту транзакції, наведена в таблиці 5.1.

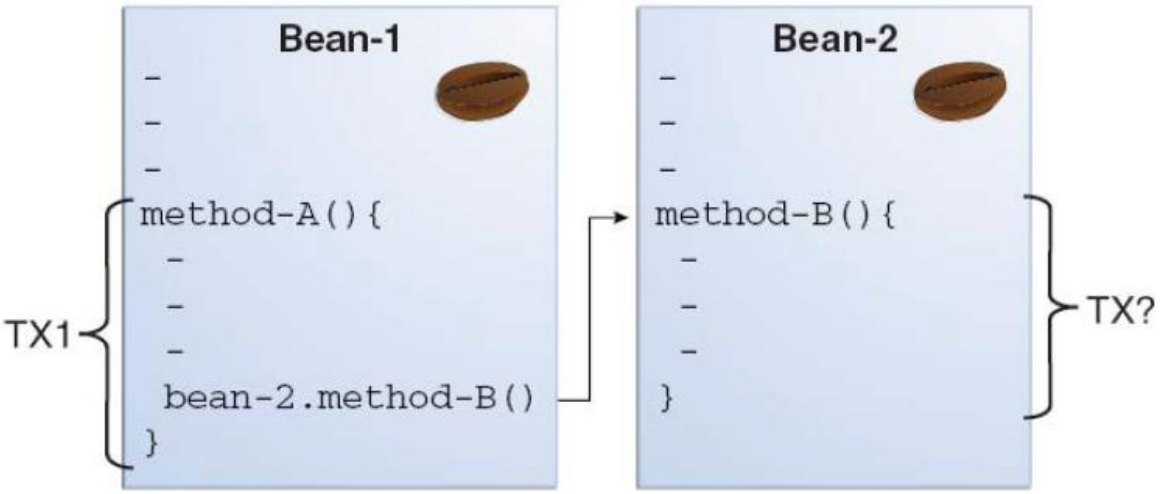

Рисунок 5.5 – Визначення області дії транзакції при виклику методу EJB-компонента з іншого методу

Таблиця 5.1 – Область дії транзакції в залежності від встановленого атрибуту транзакції

Атрибут транзакции	Клиентская транзакция	Транзакция бизнес-метода
Required	None	T2
	T1	T1
RequiresNew	None	T2
	T1	T2
Mandatory	None	error
	T1	T1
NotSupported	None	None
	T1	None
Supports	None	None
	T1	T1
Never	None	None
	T1	Error

Для визначення атрибута транзакції використовується анотація @TransactionAttribute, яка може бути визначена як над конкретним методом EJB-компонента, так і над усім класом. Значення атрибута транзакції, певне над методом, переопределяет певне над класом.

Приклад визначення атрибута транзакції:

```
@TransactionAttribute(NOT_SUPPORTED)
@Stateful
public class TransactionBean implements Transaction {
...
@TransactionAttribute(REQUIRES_NEW)
public void firstMethod() {...}
@TransactionAttribute(REQUIRED)
public void secondMethod() {...}
public void thirdMethod() {...}
public void fourthMethod() {...}
```

```
}
```

Відповідно, при виклику методу firstMethod () буде відкрита нова транзакція. При виклику secondMethod () нова транзакція буде відкрита, якщо викликав метод виконувався поза транзакції; якщо викликав метод виконувався в транзакції, метод secondMethod () буде виконуватися всередині відкритої поточної транзакції. При виклику методів thirdMethod () і fourthMethod () буде застосована наступна схема: якщо метод виконувався в транзакції, то клієнтська транзакція буде припинена перед виконанням методу і відновлена після його завершення; якщо метод виконувався поза транзакції, нова транзакція відкрита не буде.

5.2.6 Визначення рівня блокування при виконанні транзакцій в JPA

У JPA існує можливість визначення рівня блокування не тільки конкретної таблиці, але і окремого запису таблиці, відповідної примірника об'єкта. За замовчуванням, провайдери JPA виконують модифікують записи в базу даних після завершення бізнес-транзакції, крім ситуацій примусової синхронізації (виклик методу flush () менеджера сталості з програми).

Існує поняття оптимістичної (використовується за замовченням) і песимістичній блокування. У разі оптимістичній блокування дані не блокуються, перед фіксацією змін бізнес-транзакції проводиться перевірка, чи не були змінені дані іншого транзакцією в процесі виконання поточної. Це реалізовано за допомогою додавання поля VERSION в таблицю, яка автоматично інкрементується при кожній зміні даних таблиці.

У разі песимістичній блокування дані блокуються на весь час виконання транзакції, що запобігає можливість модифікації даних іншими транзакціями. Песимістичне блокування ефективне в разі можливої частої модифікації даних паралельними транзакціями.

Можливо блокування об'єкта як на запис (модифікацію, режим READ), так і на читання (режим WRITE).

Існує можливість завдання тимчасових обмежень блокування (властивість javax.persistence.lock.timeout).

Рівень блокування транзакції в JPA може бути визначений за допомогою:

1) виклику методу EntityManager.lock () і передачі в якості параметра режиму блокування:

```
EntityManager em = ...;
Person person = ...;
em.lock(person, LockModeType.OPTIMISTIC);
```

2) виклику одного з методів EntityManager.find, який приймає режим блокування в якості параметра:

```
EntityManager em = ...;
String personPK = ...;
Person person = em.find(Person.class, personPK, LockMode-
Type.PESSIMISTIC_WRITE);
```

3) виклику одного з методів EntityManager.refresh і передачі рівня блокування в якості параметра:

```
EntityManager em = ...;
String personPK = ...;
Person person = em.find(Person.class, personPK);
...
em.refresh(person, LockModeType.OPTIMISTIC_FORCE_INCREMENT);
```

4) виклику методу Query.setLockMode або TypedQuery.setLockMode з передачею режиму блокування в якості параметра:

```
Query q = em.createQuery(...);
q.setLockMode(LockModeType.PESSIMISTIC_FORCE_INCREMENT);
```

5) зазначення режиму блокування при визначенні статичного запиту:

```
@NamedQuery(name="lockPersonQuery",
query="SELECT p FROM Person p WHERE p.name LIKE :name",
lockMode=PESSIMISTIC_READ).
```

5.3 Порядок виконання роботи

1. Реалізувати відображення доменних об'єктів і зв'язків між ними в схему БД.

2. Реалізувати інтерфейси сервісів з використанням компонентів шару інтеграції з базою даних.

3. Реалізація інтеграційних тестів.

5.4 Завдання для самостійної роботи

1. Визначення збережених полів і властивостей сутностей. Валідація значень.

2. Використання колекцій в полях і властивості сутностей. Рання (EAGER) і пізня (LAZY) вивантаження об'єктів.

3. Визначення первинних ключів збережених об'єктів. Складені первинні ключі. Стратегії генерації первинних ключів.

4. Відображення відносин між сутностями. Виконання каскадних операцій.

5. Використання кеш другого рівня для зберігаються об'єктів в JPA.

6 Лабораторна робота №6 Проектування шару відображення

6.1 Мета роботи

Проаналізувати особливості реалізації шару уявлення веб-додатки. Вивчити типові рішення реалізації.

6.2 Теоретичні відомості

6.2.1 Форми відображення програми Web-сервера. Шаблон «модель – відображення – контролер»

Функції Web-сервера полягають в інтерпретації адреси URL запиту і передачі керування відповідною програмою. Існує дві основні форми подання програми Web-сервера - сценарій (script) і сторінка сервера (serverpage).

Сценарій складається з функцій або методів, призначених для обробки запитів HTTP. Типовими прикладами можуть служити сценарії CGI і Java сервлети. Сценарій часто розбивається на підпрограми і користується сторонніми службами. Він отриміє дані з Web-сторінки, перевіряючи строковий об'єкт HTTP-запиту і виокремлюючи з нього регулярні вирази. Наприклад, при використанні сервлетів Java, програміст отримує доступ до інформації запиту через інтерфейс ключових слів. Результатом роботи Web-сервера служить інша - відповідна - рядок, що утворюється сценарієм із залученням звичайних функцій поточного виведення.

Завдання формування коду HTML за допомогою команд поточного виведення ускладнює роботу дизайнерів. Це відповідним чином підводить до моделі сторінок сервера, де функції програми зводяться до повернення порції текстових даних. Сторінка містить текст HTML з "вкрапленнями" виконуваного коду. Подібний підхід, що реалізується, наприклад, в PHP, ASP і JSP, особливо зручний, якщо потрібна незначна додаткова обробка тексту з урахуванням реакції користувача.

Оскільки модель сценаріїв краще підходить для інтерпретації запитів, а схема сторінок сервера - для форматування відповідей, цілком розумно застосовувати їх спільно. Такий підхід був вперше реалізований досить давно в призначених для користувача інтерфейсів на основі типового рішення модель-представлення-контролер (Model View Controller).

Вхідний контролер приймає запит і витягує з нього інформацію. Потім він передає бізнес-логіку належному об'єкту моделі, який звертається до історника даних і виконує дії, передбачені в запиті, включаючи збір інформації, необхідної для відповіді. По завершенні функцій він передає управління вхідного контролера, який, аналізуючи отриманий результат, приймає рішення щодо вибору варіанта уявлення відповіді. Управління та відповідні дані передаються поданням. Взаємодія вхідного контролера та подання часто здійснюється не у вигляді прямих викликів, а за посеред-

ництва деякого об'єкту HTTP-сеансу, який служить для передачі даних в обох напрямках.

Типове рішення модель-відображення-контролер передбачає повне відмежування моделі від Web-відображення. Це спрощує можливості модифікації існуючих і додавання нових типів відображень. А розміщення логіки в окремих об'єктах сценарію транзакції (Transaction Script) і моделі предметної області (Domain Model) полегшує їх тестування. Це особливо важливо, коли в якості уявлення використовується сторінка сервера.

6.2.2 Типові рішення реалізації контролеру

Існує два типових підходу до організації вхідних контролерів - контролер сторінок (Page Controller) і контролер запитів (Front Controller).

Найбільш загальний підхід полягає в створенні об'єкта вхідного контролера для кожної веб-сторінки. У найпростішому випадку подібний контролер сторінок можна оформити у вигляді сторінки сервера, поєднуючи в ньому функції уявлення та вхідного контролера. У багатьох ситуаціях, проте, легше виділити вхідний контролер в самостійний об'єкт. Об'єкт, який реалізує контролер сторінок, обробляє запит до конкретної веб-сторінці або виконувати певну дію на сайті. На вхідних контролер покладаються дві основні обов'язки: обробка HTTP-запиту і прийняття рішення про подальшу опрацювання запиту, які часто має сенс розділити, доручивши першу функцію сторінці сервера, а другу - допоміжному об'єкту. Призначення контролера сторінок: аналіз параметрів запиту; створення об'єктів моделі і передача їм даних; визначення потрібного уявлення і передача йому моделі. Подібний підхід зазвичай використовується для сторінок сервера (php, jsp, asp) і добре підходить для реалізації простої логіки і схеми навігації.

Типове рішення контролер запитів передбачає використання єдиного об'єкта, призначеного для обробки всіх запитів. Оброблювач інтерпретує отриманий адресу URL, визначає відповідний тип запиту і створює окремий об'єкт для подальшої обробки запиту. Таким чином, вдається централізувати діяльність з обробки всіх HTTP-запитів в рамках єдиного об'єкта і уникнути необхідності зміни конфігурації Web-сервера в разі модифікації структури дій сайту.

6.2.3 Типові рішення реалізації відображення

Існують три основних типових рішення щодо реалізації шару уявлення: уявлення з перетворенням (Transform View), уявлення за шаблоном (Template View) і двоетапне уявлення (Two Step View). По суті, вибір зводиться до одного з двох варіантів - подання з перетворенням або подання за шаблоном в базовій одноетапною версії щоб створювати труднощі до рівня двоетапного уявлення.

Відображення за шаблоном дозволяє оформляти подання відповідно до структури сторінки і вставляти в неї спеціальні маркери, що відзначають позиції фрагментів динамічного вмісту. Подібний підхід підтримуєть-

ся рядом платформ, багато з яких засновані на моделі сторінок сервера (наприклад, ASP, JSP і PHP) і дозволяють впроваджувати в текст сторінки код на повнофункціональному мовою програмування. Таке рішення відрізняється потужністю і гнучкістю; якщо код складний і заплутаний, завдання супроводу системи надзвичайно важко. Тому використання технологій сторінок сервера вимагає акуратності і послідовності в відокремленні логіки коду від структури сторінки, яке найчастіше досягається за допомогою допоміжного (helper) об'єкта.

Підхід реалізації «уявлення з перетворенням» передбачає послідовну обробку елементів даних домену та перетворення їх в код HTML. Прикладом реалізації уявлення з перетворенням може служити XSLT. Ця технологія виявляється досить ефективною, якщо дані домену зберігаються в форма¬те XML або допускають швидке перетворення в XML. Вхідний контролер вибирає потрібну таблицю стилів XSLT і застосовує її до XML-коду, що описує модель.

Одноетапний варіант реалізації уявлення передбачає переважно по одному компоненту представлення для кожного інтерфейсного екрану програми. Код уявлення отримує дані домену і перетворює їх в формат HTML.

Двохетапне уявлення розділяє процес на дві стадії: на першій на основі даних домену формується логічний екран, який на другій стадії транс¬формируется в код HTML. Для кожного екрану існує одне подання першого етапу, але для додатка в цілому - тільки одне подання другого етапу.

Перевага двоетапного уявлення полягає в тому, що рішення щодо варіант перетворення в HTML приймається централізовано. Це істотно полегшує завдання внесення глобальних змін, оскільки для модифікації кожного екрану досить відредагувати дані єдиного об'єкта. Зрозуміло, скористатися такою перевагою вдасться тільки в тому випадку, коли логічне уявлення залишається постійним, тобто різні екрани компонуються за одним принципом. Сайти, спроектовані по надмірно складними схемами, одноманітністю логічної структури зазвичай не відрізняються.

Типове рішення двоетапне відображення добре проявляє себе в ситуаціях, де служби Web-додатки використовуються численними клієнтами (наприклад, відвідувачами сайту системи бронювання авіаквитків). Задовольняючи вимогам компонування одного логічного екрана, кожна з версій клієнтської програми, що відповідає певному варіанту реалізації другого етапу уявлення, може мати інший зовнішній вид. Таким же чином двоетапне відображення може бути використано і для обслуговування різних пристроїв виводу, коли необхідно передбачити окремі реалізації другого етапу відображення.

6.3 Порядок виконання роботи

1. Вивчити типові рішення реалізації компонентів шару уявлення.
2. Вибір фреймворка для реалізації шару уявлення.

3. Розробка карти сайту, діаграми переходів.

4. Розробка прототипу інтерфейсу і тестування основних сценаріїв. Переробка прототипу інтерфейсу з урахуванням результатів тестування.

6.4 Завдання для самостійної роботи

Вивчення основних принципів використання існуючих MVC-фреймворків.

7 Лабораторна робота №7 Програмна реалізація та тестування компонентів шару відображення

7.1 Мета роботи

Отримання практичних навичок використання технології JSF. Реалізація шару уявлення корпоративного програми. Оцінка продуктивності і тестування програми.

7.2 Теоретичні відомості

7.2.1 Особливості використання технології JSF, конфігурація застосування

JSF (JavaServer Faces) є специфікацією, а також технологією і фреймворком, яка надає:

- API для подання компонент для користувача інтерфейсу і управління їх станом; обробка подій; валідація та обробка даних; визначення правил навігації; підтримка інтернаціоналізації та визначення прав доступу;
- бібліотеки тегів для додавання компонентів на веб-сторінки і зв'язку з серверними об'єктами.

JSF реалізує MVC модель, зв'язок між шарами моделі і представлення здійснюється через керовані об'єкти BackingBeans. Відповідно в об'єкти BackingBean, прив'язані до JSF, бажано не розміщувати ні бізнес логіку, ні логіку збереження даних, а делегувати ці функції в шар моделі самого додатка. До об'єктів додатки можна отримувати доступ через властивості керованих об'єктів.

Facelets є технологією управління поданням для JSF2.0. Технологія JSP, яка використовувалася раніше для управління поданням до JSF, не підтримує нових функцій JSF 2.0 і вважається застарілою технологією відображення для JSF 2.0.

Структура стандартного JSF-додатка наступна:

- веб-сторінки з розташованими на них компонентами;
- бібліотека тегів компонентів;
- компоненти backing beans, що визначають властивості і функції компонентів сторінки;
- дескриптор розгортання (web.xml);
- один або більше файлів конфігурації (faces-config.xml або ін.), які можуть використовуватися для визначення правил навігації і конфігурації bean-ів та інших компонентів додатка;
- власні (призначені для користувача) компоненти, валідатори, конвертори або слухачі;
- власні (призначені для користувача) теги для відображення користувальницьких компонентів на сторінці.

Переваги використання JSF:

- повне розділення поведінки від подання;
- незалежна робота дизайнерів і розробників (зв'язок з серверними компонентами за допомогою тегів, без написання скриптів);
- відсутність обмежень на використання певних скриптових технологій або мов розмітки; JSF API розміщено поверх Servlet API (рисунок 7.1);
- використання різних технологій відображення, створення власних компонент, генерація виведення для різних клієнтів;
- технологія Facelets (як частина JSF2.0) надає можливість повторного використання коду, розширення компонентів за допомогою функцій шаблонирования і composite-компонентів;
- автоматична реєстрація backing bean-ів як ресурсів JSF-додатки при використанні анотацій JSF;
- швидке конфігурація сторінки навігації за допомогою визначення "неявних" правил навігації;
- багата архітектура для управління складовими компонентами, управління даними, перевірки введених даних і обробки подій.

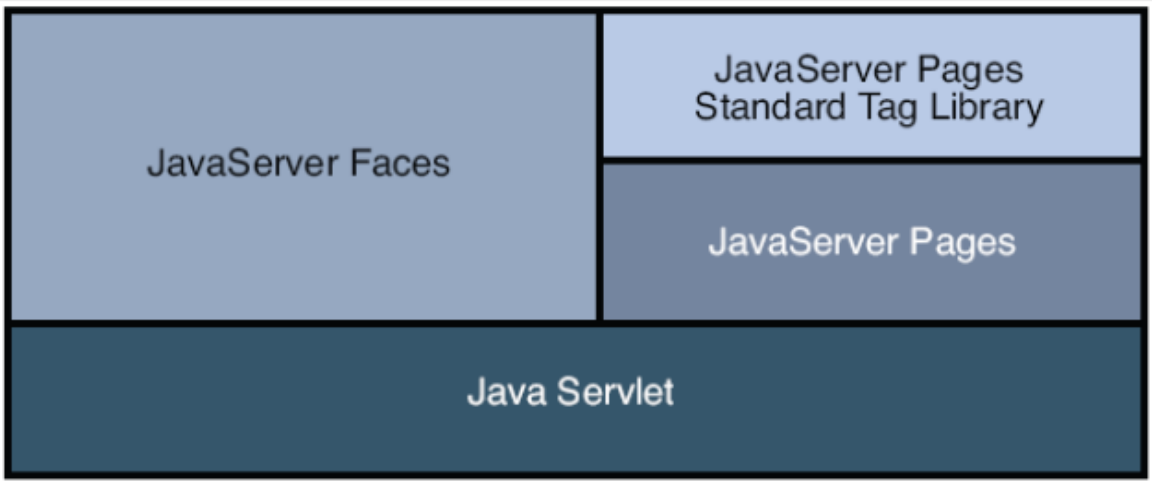

Рисунок 7.1 – Особливості реалізації технології JSF

Дескриптор розгортання web.xml повинен містити елементи, обов'язкові для Facelets застосування.

1. Параметр контексту, який визначає стадію розробки проекту:

```
<context-param>
<param-name>javax.faces.PROJECT_STAGE</param-name>
<param-value>Development</param-value>
</context-param>
```

Наприклад, при встановленому значенні параметра контексту Development налагоджувальна інформація буде автоматично генерує-тися і виводитися. Значення за замовчуванням - Production.

2. Визначення та відображення сервлета FacesServlet:

```
<servlet>
  <servlet-name>Faces Servlet</servlet-name>
  <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>
```

```
<servlet-mapping>
  <servlet-name>Faces Servlet</servlet-name>
  <url-pattern>/faces/*</url-pattern>
</servlet-mapping>
```

Таким чином, замість написання власного сервлету, використовується стандартний JSF-сервлет для обробки запитів. Сервлет повинен викликатися для кожного запиту зі сторінок, що використовують JSF. Згідно відображенню в web.xml, JSF контейнер буде викликати сервлет Faces для обробки всіх запитів, відносний URI яких починається з / faces /. Завдяки цьому буде коректно инициализироваться контекст Faces і кореневий елемент вистави перед показом сторінок JSF. Кореневий елемент містить дерево компонентів, а контекст Faces служить для взаємодії додатку з JSF

3. Стартова сторінка програми:

```
<welcome-file-list>
    <welcome-file>faces/index.xhtml</welcome-file>
</welcome-file-list>
```

За замовчуванням сервлет Faces використовує конфігураційний файл faces-config.xml, розташований в директорії WEB-INF Web-додатки. У файлі конфігурації можуть визначатися ресурси програми, правила навігації, компоненти ManagedBean і ін.

Наприклад, для глобального управління повідомленнями, необобхідно визначити набір ресурсів (resource bundle) в файлі faces-config.xml і використовувати його для зберігання текстів повідомлень.

```
<?xml version="1.0" encoding="UTF-8"?>
<faces-config xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-facesconfig_1_2.xsd"
    version="1.2">
    <application>
        <message-bundle>messages</message-bundle>
    </application>
  ...
```

Вміст файла message.properties:

```
javax.faces.component.UIInput.REQUIRED_detail=required
javax.faces.converter.IntegerConverter.INTEGER_detail=not a valid
number
```

В цьому випадку для зміни повідомлення про помилку конвертації або валідації для будь-якого поля введення досить поміняти текст в глобальному файлі ресурсів.

Можливе використання додаткових конфігураційних файлів. Для цього використовується параметр javax.faces.application.CONFIG_FILES, в якому можна вказати імена файлів, розділені комами.

Динамічна складова jsf-сторінок визначається за допомогою властивостей і методів керованих компонентів (backing bean). Зв'язування компонентів jsf-сторінок з відповідними властивостями / методами об'єктів backing bean виконується з використанням JSF Expression Language (EL).

EL пов'язує поля введення / виведення з відповідними значення-ми властивостей об'єктів. Зв'язок двунаправлений, тобто якщо значення властивості було 100, то в поле виведення буде виведено 100 при відображенні сторінки. Якщо користувач ввів 200, то 200 буде збережено у властивості

об'єкта (не враховуючи можливості виникнення помилок при конвертації та валідації даних).

Для визначення backing bean використовується анотація @Named із зазначенням в якості параметра імені компонента (за замовчуванням в якості імені використовується назва класу):

```
import javax.enterprise.context.RequestScoped;
import javax.inject.Named;

@Named
@RequestScoped
public class Hello {

    private String name;

    public Hello() {
    }

    public String getName() {
        return name;
    }

    public void setName(String user_name) {
        this.name = user_name;
    }
}
```

Поняття Scope визначає зону видимості керованого jsf-компонента. Можливі такі значення:

- @ RequestScoped - час життя об'єкта відповідає часу обробки одиничного HTTP запиту;
- @SessionScoped - час життя об'єкта відповідає часу користувальницької сесії, зберігає стан між HTTP за-просив сесії;
- @ApplicationScoped - зберігає стан між множествен-ними HTTP запитами різних призначених для користувача сесій.

Приклад звернення до властивості оголошеного backing bean з компонента введення jsf-сторінки:

```
<h:body>
  <h:form>
   <h:inputText id="username" value="#{hello.name}"
      required="true" requiredMessage="Error: A name is required."
     maxlength="25" />
…
  </h:form>
</h:body>
```

Атрибути required і requiresMessage визначають обов'язковість заповнення поля і відповідне виведене повідомлення.

Аналогічно обробка дії може бути пов'язана з викликом методу backing bean допомогою EL-виразів. Таким чином, при натисканні на кнопку буде викликаний відповідний метод (якщо не виникло помилок конвертації або валідації).

За замовчуванням JSF виробляє затвердження даних форми перед викликом методу. Якщо встановлено значення атрибута immediate в true, JSF викличе метод негайно.

7.2.2 Шаблони та составні компоненти

Шаблонування є функцією технології Facelets, що дозволяє створювати сторінку, яка буде виступати в якості бази або шаблону для інших сторінок додатку. Шаблонування дозволяє забезпечити можливість повторного використання коду, а також забезпечує стандартний вид «великого» додатка з великою кількістю сторінок.

Використовуються такі ключові слова для шаблонирования:<ui:component/>, <ui:composition/>, <ui:debug/>, <ui:decorate/>, <ui:define/>, <ui:fragment/>, <ui:include/>, <ui:insert/>, <ui:param/>, <ui:repeat/>, <ui:remove/>.

Приклад шаблону template.xhtml:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
xmlns:ui="http://xmlns.jcp.org/jsf/facelets"
xmlns:h="http://xmlns.jcp.org/jsf/html">
      <h:head>
            <meta http-equiv="Content-Type"
                  content="text/html; charset=UTF-8" />
            <h:outputStylesheet library="css" name="default.css"/>
            <h:outputStylesheet library="css" name="cssLayout.css"/>
            <title>Facelets Template</title>
      </h:head>
      <h:body>
            <div id="top" class="top">
                  <ui:insert name="top">Top Section</ui:insert>
            </div>
            <div>
            <div id="left">
                  <ui:insert name="left">Left Section</ui:insert>
            </div>
            <div id="content" class="left_content">
                  <ui:insert name="content">Main Content</ui:insert>
            </div>
            </div>
      </h:body>
</html>
```

Наведений шаблон визначає три секції сторінки: top, left, main і визначає стилі секцій. Використання наведеного шаблону для іншої сторінки додатка проводиться за допомогою тега <ui: composition />. Підключення та визначення секцій виконується за допомогою тега <ui: define />.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
xmlns:ui="http://xmlns.jcp.org/jsf/facelets"
xmlns:h="http://xmlns.jcp.org/jsf/html">
      <h:body>
            <ui:composition template="./template.xhtml">
                  <ui:define name="top">
                        Welcome to Template Client Page
                  </ui:define>
                  <ui:define name="left">
                        <h:outputLabel value="You are in the Left Sec-
                  tion"/>
                  </ui:define>
                  <ui:define name="content">
```

```
                  <h:graphicImage val-
            ue="#{resource['images:wave.med.gif']}"/>
                  <h:outputText value="You are in the Main Content
            Section"/>
            </ui:define>
      </ui:composition>
</h:body>
</html>
```

JSF вводить поняття складеного (composite) компонента. Під компонентом розуміється повторно використовуваний код, який виконує певну функціональність (наприклад, inputText). Складовий компонент являє собою особливий тип шаблону, який виступає в ролі повторно використовуваного для користувача компонента і складається з набору інших компонентів і власних розширень; може мати пов'язані з ним валідатори, конвертори і слухачі. Будь-яка xhtml-сторінка може бути конвертована в складовий компонент.

Теги, використовувані для визначення складових компонентів: <composite: interface />, <composite: implementation />, <composite: attribute />, <composite: insertChildren />, <composite: valueHolder />, <composite: editableValueHolder / >, <compo-site: actionSource />.

Приклад реалізації composite компонента, який представляє компонент введення email адреси:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
xmlns:composite="http://xmlns.jcp.org/jsf/composite"
xmlns:h="http://xmlns.jcp.org/jsf/html">
<h:head>
      <title>This content will not be displayed</title>
</h:head>
<h:body>
      <composite:interface>
            <composite:attribute name="value" required="false"/>
      </composite:interface>
      <composite:implementation>
            <h:outputLabel value="Email id: "></h:outputLabel>
            <h:inputText value="#{cc.attrs.value}"></h:inputText>
      </composite:implementation>
</h:body>
</html>
```

Вираз # {cc.attrs.attribute-name} використовується для отримання доступу до атрибутів, певним інтерфейсом компонента. У наведеному вище прикладі атрибутом є value. Наведений компонент збережений як email.xhtml в каталозі resources / emcomp, який розглядається JSF як бібліотека компонентів.

Для використання складеного компонента email.xhtml, посилання на бібліотеку компонента повинна бути підключена при визначенні простору імен xml:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html"
      xmlns:em="http://xmlns.jcp.org/jsf/composite/emcomp">
<h:head>
      <title>Using a sample composite component</title>
```

```
</h:head>
<body>
      <h:form>
            <em:email value="Enter your email id" />
      </h:form>
</body>
</html>
```

Звернення до компоненту відбувається за допомогою тега <em:email/>.

7.2.3 Визначення правил навігації

Виділяють поняття явної і неявної навігації. Останнє перед-вважає визначення правил навігації поза файлу конфігурації jsf-додатки.

Прикладами неявної навігації є наступні:

- сторінка переходу вказується в якості значення action-властивості компонента
  <H: commandButton value = "submit" action = "response">
- сторінка переходу вказується як результат виконання методу backing bean-a, що виконує обробку події:

```
    public String addUser() {
        User usr = new User(this.login, this.passwd);
        User usrExists = userFacade.getUserByLogin(login);
        if (usrExists != null) {
            FacesMessage facesMessage = new FacesMessage("User with such
name already exists");
            facesMessage.setSeverity(facesMessage.SEVERITY_ERROR);
            FacesContext.getCurrentInstance().addMessage("registerForm",
facesMessage); //$NON-NLS-1$
            return "addUser";
        }
        userFacade.create(usr);
        User user = userFacade.getUserByLoginAndPassword(login, passwd);
        this.id = user.getId();

        return "welcome";

    }

   <h:commandButton id="addButton" value="Add"
                    action="#{userBean.addUser}"/>
```

Останній варіант забезпечує можливість визначення схеми навігації в залежності від певних умов.

Явна визначення правил навігації передбачає визначення строкового параметра переходу з однієї сторінки на іншу в файлі конфігурації jsf (faces-config.xml або ін.):

```
<navigation-rule>
      <from-view-id>/greeting.xhtml</from-view-id>
      <navigation-case>
            <from-outcome>success</from-outcome>
            <to-view-id>/response.xhtml</to-view-id>
      </navigation-case>
</navigation-rule>
```

Відповідно, значення action-властивості компонента або ре-зультат виконання методу backing bean-а повинні бути прив'язані до встановлених в файлі конфігурації строковим параметрам переходу.

7.2.4 Механізми керування безпекою

Компоненти JavaEE додатків можуть містити як захищені, так і незахищені ресурси. Доступ до захищених ресурсів додатки повинен бути організований відповідно до прав користувачів, які визначаються політикою безпеки програми.

Процес авторизації (authorization) забезпечує керований доступ до захищених ресурсів додатки. Авторизація заснована на процесах ідентифікації (identification) і аутентифікації (authentication).

Ідентифікація є процесом визначення (впізнання, розпізнавання) сутності системою; аутентифікація є процесом перевірки автентичності користувача, пристрою або іншої сутності в комп'ютерній системі, результати аутентифікації використовуються на етапі авторизації для надання доступу до ресурсів системи.

Процеси авторизації і аутентифікації необов'язкові для надання доступу до незахищених ресурсів додатки (анонімний доступ).

Виділяють наступні аспекти інформаційної безпеки програми:

- аутентифікація - процес взаємної перевірки автентичності клієнта і сервера;
- авторизація або контроль доступу - надання доступу до ресурсів відповідно до політики безпеки програми з метою забезпечення цілісності та конфіденційності даних;
- цілісність даних (data integrity) - забезпечення можливості виявлення підміни інформації «третьою особою», які не є джерелом інформації. Так, наприклад, одержувач даних, переданих через публічну мережу, повинен мати можливість виявлення і видалення інформації, яка була змінена «третьою особою» після відправки джерелом. Це гарантує можливість зміни даних тільки авторизованими користувачами;
- конфіденційність даних (confidentiality, or data privacy) - забезпечення доступу до конфіденційної інформації тільки для авторизованих користувачів;
- стійкість виконання операцій (non-repudiation) - результати виконання успішно завершених користувачем дій не повинні бути втрачені або скасовані;
- якість обслуговування (Quality of Service) - забезпечення якісного обслуговування клієнтських запитів з урахуванням параметрів мережі передачі даних;
- облік виконаних дій (auditing) - облік виконання операцій доступу до захищених ресурсів додатки з метою оцінки ефективності використовуваних політики і механізмів забезпечення безпеки.

JavaEE надає наступні розгортаються сервіси забезпечення інформаційної безпеки програми:

- Java Authentication and Authorization Service (JAAS);
- Java Generic Security Services (Java GSS-API);
- Java Cryptography Extension (JCE);
- Java Secure Sockets Extension (JSSE);
- Simple Authentication and Security Layer (SASL).

Сервіси безпеки надаються на рівні JavaEE контейнера компонентів. Можливо декларативне і програмне керування безпекою додатки на рівні контейнера. Декларативне передбачає визначення політики безпеки за допомогою дескриптора розгортання програми (web.xml, ejb-jar.xml) або за допомогою визначення анотацій в класах компонентів; програмне - використання методів інтерфейсів EJBContext і HttpServletRequest.

Виділяють такі рівні управління інформаційною безпекою програми:

- управління на рівні додатку (application-layer security);
- управління на транспортному рівні (transport-layer security);
- управління на рівні повідомлень (message-layer security).

Управління безпекою веб-додатки на рівні контейнера включає наступні етапи.

1. Визначення механізму аутентифікації.

Механізм аутентифікації визначає спосіб взаємної перевірки автентичності клієнта і сервера. JavaEE підтримує такі механізми аутентифікації:

- basic authentication;
- form-based authentication;
- digest authentication;
- client authentication;
- mutual authentication.

Механізми аутентифікації basic і form-based вважаються найменш надійними. У разі basic authentication дані аутентифікації користувача (логін і пароль) відправляються у вигляді тексту в кодуванні Base64. Form-based authentication передбачає відправку даних аутентифікації у вигляді звичайного тексту. В обох випадках аутентифікація з боку серверу не передбачається. Уразливість зазначених механізмів аутентифікації може бути знижена за рахунок використання захищених протоколів передачі інформації.

Приклад використання form-based аутентифікації:

```
<form method="POST" action="j_security_check">
<input type="text" name="j_username">
<input type="password" name="j_password">
</form>
```

Використання form-based аутентифікації дозволяє стандартизувати стиль форм аутентифікації додатки.

Digest аутентифікація, також як і basic, має на увазі використання логін / пароль в якості даних аутентифікації користувача. Однак, в разі digest-аутентифікації, замість відправки пароля користувача передбачається від-

правка од-ностороннего криптографічного хеша пароля і додаткових даних.

Client аутентифікація передбачає використання цифрових сертифікатів X.509 як даних аутентифікації користувача.

Mutual аутентифікація передбачає двосторонню взаємну аутентифікацію клієнтської і серверної сторони. Дані аутентифікації можуть бути представлені як цифровими сертифікатами, так і у вигляді логін / пароль.

Механізм аутентифікації визначається за допомогою елемента <login-config /> (подтег <auth-method />) дескриптора розгортання програми.

Приклад визначення form-based authentication:

```
<login-config>
      <auth-method>FORM</auth-method>
      <realm-name>file</realm-name>
      <form-login-config>
            <form-login-page>/login.xhtml</form-login-page>
            <form-error-page>/error.xhtml</form-error-page>
      </form-login-config>
</login-config>
```

Приклад визначення digest authentication:

```
<login-config>
      <auth-method>DIGEST</auth-method>
</login-config>
```

Приклад визначення client authentication:

```
<login-config>
      <auth-method> CLIENT-CERT</auth-method>
</login-config>
```

2. Налаштування політики безпеки розгорнутого додатки. Ролі користувачів додатка можуть бути визначені за допомогою елемента <security-role /> дескриптора розгортання програми.

Захищені ресурси програми, а також права доступу користувальницьких ролей до захищених ресурсів додатки визначаються за допомогою елемента <security-constraint />.

```
<security-constraint>
      <web-resource-collection>
            <web-resource-name>Protected Area</web-resource-name>
            <url-pattern>/security/protected/*</url-pattern>
            <http-method>PUT</http-method>
            <http-method>DELETE</http-method>
            <http-method>GET</http-method>
            <http-method>POST</http-method>
      </web-resource-collection>
      <auth-constraint>
            <role-name>manager</role-name>
      </auth-constraint>
</security-constraint>

<!-- Security roles used by this web application -->
<security-role>
      <role-name>manager</role-name>
</security-role>
<security-role>
      <role-name>employee</role-name>
</security-role>
```

3. Визначення авторизованих користувачів і груп (визначення реалма додатки).

Реалм є доменом політики безпеки, який визначається для веб-сервера або сервера додатків. Реалм веб-додатки зазвичай представлений базою даних, що містить інформацію про користувачів і групи, що визначають коректних користувачів додатка (одного або декількох).

JavaEE сервер може бути налаштований на використання декількох реалмов.

Glassfish сервер за замовчуванням автоматично налаштований на використання реалмов наступних типів: file realm, admin-realm, certificate realm.

File realm передбачає збереження даних аутентифікації користувачів локально на сервері у файлі keyfile; admin-realm - збереження даних аутентифікації користувачів, що має права адміністратора, локально на сервері у файлі admin-keyfile; certificate realm - дані аутентифікації користувачів зберігаються в базі даних сертифікатів X.509 (в цьому випадку поняття групи користувачів не може використовуватися).

4. Відповідність користувальницьких ролей додатки певним користувачам і групам.

Кожному авторизованому користувачеві, визначеному в реалмі контейнера, ставиться у відповідність роль користувача додатку, що визначає права доступу користувача до захищених ресурсів додатки згідно політики безпеки (рисунок 7.2).

Відображення Security Roles додатки в Users / Groups сервера додатків може бути визначено в консолі адміністрування сервера додатків або в дескрипторі розгортання, відповідному використовуваному сервера додатків.

Наприклад, в разі використання сервера glassfish настройки відображення Security Roles повинні бути визначені в файлі конфігурації сервера glassfish-web.xml:

```
<security-role-mapping>
      <role-name>USERS</role-name>
      <group-name>USERS</group-name>
</security-role-mapping>
<security-role-mapping>
      <role-name>ADMINS</role-name>
      <group-name>ADMINS</group-name>
</security-role-mapping>
```

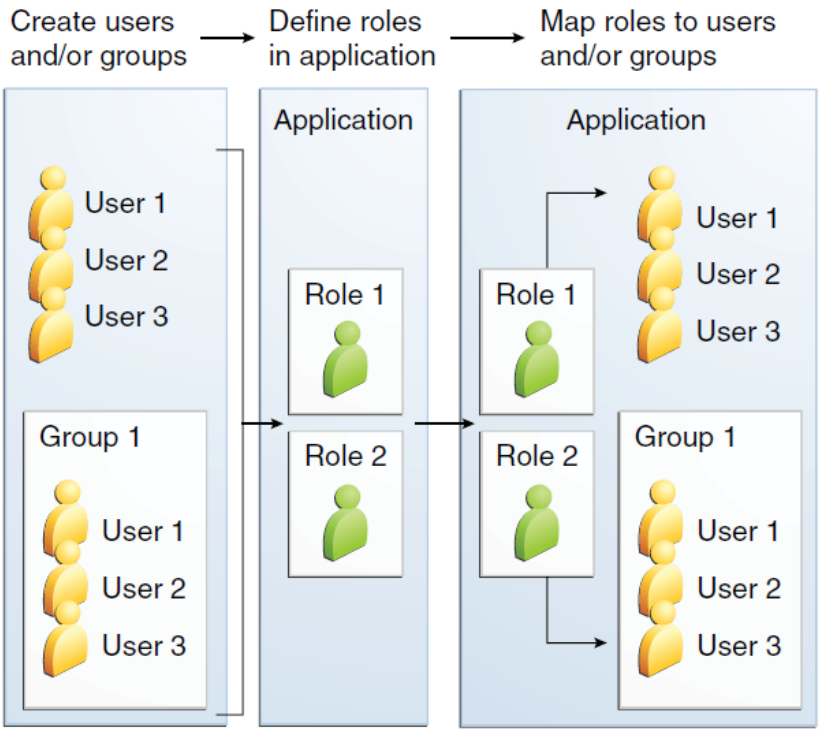

Рисунок 7.2 – Відповідність ролі користувача додатки користувачам і групам, визначеним у реалме сервера додатків

Декларативне управління безпекою дозволяє визначити права доступу різних призначених для користувача ролей додатки до компонентів програми (EJB і web-компонентів). Для цього використовуються анотації @DeclareRoles - визначити всі існуючі ролі користувачів додатка; @RolesAllowed - визначити ролі користувачів, які мають право на виклик методів компонента; @ DenyAll / PermitAll - заборонити / дозволити доступ усім користувачам програми. Права доступу можуть бути визначені як над класом EJB-компонента, так і над конкретними його методами. Права доступу, певні над методом, скасовують права доступу, визначені над класом компонента.

Приклад декларативного обмеження доступу до методів EJB-компонента:

```
import javax.annotation.security.*;
...
@DeclareRoles({"DEPT-ADMIN", "DIRECTOR"})
@Stateless
public class PayrollBean implements Payroll {
      @RolesAllowed("DEPT-ADMIN")
      public void reviewEmployeeInfo(EmplInfo info) {
            // ...
      }
      @RolesAllowed("DIRECTOR")
      public void updateEmployeeInfo(EmplInfo info) {
            // ...
      }
...
}
```

Для обмеження прав доступу до сервлету може використовуватися анотація @HttpConstraint всередині анотації @ServletSecurity, наприклад:

```
@WebServlet(name = "PayrollServlet", urlPatterns = {"/payroll"})
@ServletSecurity(
```

```
@HttpConstraint(transportGuarantee = TransportGuarantee.CONFIDENTIAL,
      rolesAllowed = {"DEPT-ADMIN", "DIRECTOR"}))
public class GreetingServlet extends HttpServlet {
      ….
}
```

Для налаштування і конфігурації SSL-підтримки сервером додатків необхідно:

1) визначити connector-елемент в дескрипторі розгортання;

2) розгорнути сховище коректних ключа та сертифікатів (keystore and certificate files);

3) розміщення сховища ключа і пароль до нього повинні бути вказані в дескрипторі розгортання сервера.

Подтег <transport-guarantee> елемента <user-data-constraint> дескриптора розгортання може використовуватися для визначення необхідності використання захищеного з'єднання для всіх URL і HTTP-методів, визначених у security constraint.

```
<security-constraint>
      <web-resource-collection>
            <web-resource-name>view dept data</web-resource-name>
            <url-pattern>/hr/employee/*</url-pattern>
            <http-method>GET</http-method>
            <http-method>POST</http-method>
      </web-resource-collection>
      <auth-constraint>
            <role-name>DEPT_ADMIN</role-name>
      </auth-constraint>
      <user-data-constraint>
            <transport-guarantee>CONFIDENTIAL</transport-guarantee>
      </user-data-constraint>
</security-constraint>
```

Можливі значення <transport-guarantee> визначають рівень захищеності з'єднання:

- CONFIDENTIAL - пересилається інформація не може бути переглянута «третьої» стороною;
- INTEGRAL - пересилається інформація не може бути модифікована «третьої» строной;
- NONE - контейнер забезпечує підтримку як захищених, так і незахищених з'єднань.

Поняття поширення SecurityIdentity використовується в разі, коли при обробці запиту з бізнес-методу поточного EJB-компонента відбувається виклик бізнес-методу іншого EJB-компонента, керованого стороннім контейнером (рисунок 7.3).

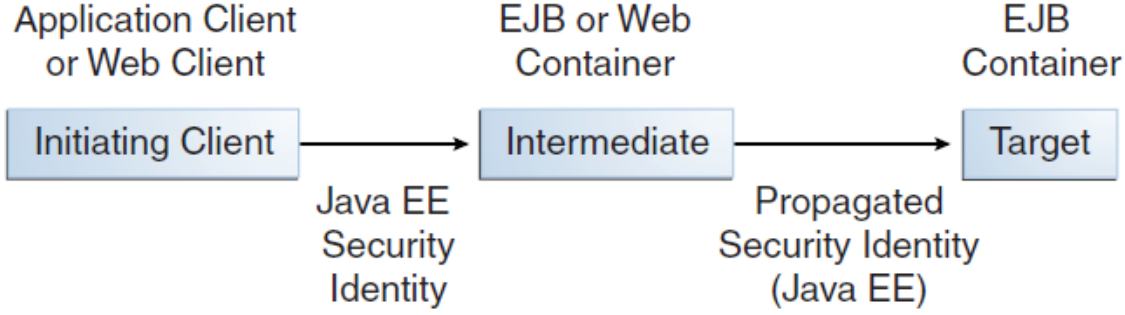

Рисунок 7.3 – Пощирення SecurityIdentity

SecurityIdentity при виконанні другого запиту може відповідати:

- ідентифікатором сутності, яка ініціювала перший запит (за замовчуванням);
- специфічного ідентифікатором, що приймається цільовим контейнером і визначається за допомогою анотації @RunAs над Классс EJB-компонента або конкретним методом:

```
@RunAs("Admin")
public class Calculator {
    //....
}
```

7.3 Порядок виконання роботи

1. Конфігурація веб-модуля додатка.

2. Розробка веб-сторінок додатку і відповідних когось тами BackingBean.

3. Визначення правил навігації програми.

4. Тестування основних сценаріїв використання веб-додатки.

4. Планування і проведення навантажувального тестування прирозкладання. Оцінка результатів.

7.4 Завдання до самостійної роботи

1. Використання стандартних і власних конверторів, слухачів і валідаторів в JSF.

2. Розробка власних компонентів JSF.

3. Вивчення принципів використання AJAX.

Рекомендована література

1. Фаулер, М. Архитектура корпоративных программных приложений: пер. с англ. – М.: Издательский дом «Вильямс», 2006. – 544 с.
2. Eric Jendrock, Ricardo Cervera-Navarro, Ian Evans and others. The Java EE 7 Tutorial. – August 2013.
3. Martin Fowler. Inversion of Control Containers and the Dependency Injection pattern [Електронний ресурс]. – Режим доступу: URL: http://martinfowler.com/articles/injection.html.
4. Java Platform, Enterprise Edition (Java EE) | Oracle Technology Network [Електронний ресурс]. – Режим доступу: URL: http://www.oracle.com/technetwork/java/javaee/overview/index.html.