МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ "ЧЕРНІГІВСЬКА ПОЛІТЕХНІКА"

# ТЕХНОЛОГІЇ ПРОЄКТУВАННЯ ПРОГРАМНИХ СИСТЕМ

**МЕТОДИЧНІ ВКАЗІВКИ**

до виконання лабораторних робіт та самостійної роботи
для здобувачів першого (бакалаврського) рівня вищої освіти
спеціальності 123 – «Комп'ютерна інженерія»

Обговорено і рекомендовано
на засіданні кафедри
інформаційних та комп'ютерних
систем
*Протокол № 5*
*від 30 квітня 2023 р.*

Чернігів 2023

Технології проєктування програмних систем. Методичні вказівки до виконання лабораторних робіт та самостійної роботи для здобувачів першого (бакалаврського) рівня вищої освіти спеціальності 123 – «Комп'ютерна інженерія». Частина І. / Укл.: Казимир В.В., Пріла О.А., Хижняк А.В., Чорноног О.А. – Чернігів: НУ «Чернігівська політехніка», 2023. – 115 с., укр. мовою.

Укладачі: КАЗИМИР ВОЛОДИМИР ВІКТОРОВИЧ, доктор технічних наук, професор кафедри інформаційних та комп'ютерних систем;
ПРІЛА ОЛЬГА АНАТОЛІЇВНА, кандидат технічних наук, доцент кафедри інформаційних та комп'ютерних систем;
ХИЖНЯК АНДРІЙ ВАСИЛЬОВИЧ, старший викладач кафедри інформаційних та комп'ютерних систем;
ЧОРНОНОГ ОЛЬГА АНАТОЛІЇВНА, асистент кафедри інформаційних та комп'ютерних систем

Відповідальний за випуск: БАЗИЛЕВИЧ ВОЛОДИМИР МАРКОВИЧ, завідувач кафедри інформаційних та комп'ютерних систем Національного університету "Чернігівська політехніка", кандидат економічних наук, доцент

Рецензент: БИВОЙНО ПАВЛО ГЕОРГІЙОВИЧ, кандидат технічних наук, доцент кафедри інформаційних та комп'ютерних систем Національного університету "Чернігівська політехніка"

## Зміст

Вступ ... 5
1 Загальні теоретичні відомості ... 6
1.1 Поняття корпоративних програмних систем (систем рівня підприємства) ... 6
1.2 Проєктування багатошарової архітектури застосування ... 7
1.2.1 Особливості використання архітектурних шарів ... 7
1.2.2 Класична модель розшарування системи ... 8
1.2.3 Вертикальна та горизонтальна декомпозиція системи ... 9
1.2.4 Шар сервісів в класичній моделі розшарування ... 10
1.2.5 Взаємодія між шарами ... 11
1.2.6 Розміщення шарів ... 11
1.3 Концепції «чистої» архітектури (“Clean” architecture). Domain-Driven Design (DDD) та принципи проєктування S.O.L.I.D ... 12
1.3.1 Принцип Single Responsibility Principle ... 13
1.3.2 Принцип Open-Closed Principle ... 17
1.3.3 Принцип Liskov Substitution Principle ... 20
1.3.4 Принцип Interface Segregation Principle ... 23
1.3.5 Inversion of Control та Dependency Inversion ... 24
1.4 Висновки ... 28
2 Реалізація корпоративної програмної системи з використанням стандарту Jakarta EE ... 29
2.1 Лабораторна робота № 1 Проєктування архітектури та слою бізнес-логіки корпоративного програмного застосування ... 30
2.1.1 Мета роботи ... 30
2.1.2 Теоретичні відомості ... 30
2.1.3 Порядок виконання роботи ... 37
2.1.4 Завдання для самостійної роботи ... 37
2.1.5 Структура звіту ... 37
2.2 Лабораторна робота №2 Програмна реалізація та тестування компонентів бізнес-логіки застосування ... 39
2.2.1 Мета роботи ... 39
2.2.2 Теоретичні відомості ... 39
2.2.3 Порядок виконання роботи ... 45
2.2.4 Завдання для самостійної роботи ... 45
2.3 Лабораторна робота №3 Проєктування слою інтеграції ... 46
2.3.1 Мета роботи ... 46
2.3.2 Теоретичні відомості ... 46
2.3.3 Порядок виконання роботи ... 49
2.3.4 Завдання для самостійної роботи ... 49
2.4 Лабораторна робота №4 Програмна реалізація та тестування компонентів шару інтеграції ... 50
2.4.1 Мета роботи ... 50

2.4.2 Теоретичні відомості .......... 50
2.4.3 Порядок виконання роботи .......... 62
2.4.4 Завдання для самостійної роботи .......... 62
2.5 Лабораторна робота №5 Проєктування шару відображення .......... 63
2.5.1 Мета роботи .......... 63
2.5.2 Теоретичні відомості .......... 63
2.5.3 Порядок виконання роботи .......... 65
2.5.4 Завдання для самостійної роботи .......... 66
2.6 Лабораторна робота №6 Програмна реалізація та тестування компонентів шару відображення .......... 67
2.6.1 Мета роботи .......... 67
2.6.2 Теоретичні відомості .......... 67
2.6.3 Порядок виконання роботи .......... 80
2.6.4 Завдання до самостійної роботи .......... 80
3 Реалізація корпоративної програмної системи з використанням фреймворку Spring .......... 81
3.1 Лабораторна робота № 1 Знайомство з фреймворком Spring .......... 82
3.1.1 Мета роботи .......... 82
3.1.2 Теоретичні відомості .......... 82
3.1.3 Порядок виконання роботи .......... 83
3.2 Лабораторна робота № 2 Spring Data .......... 88
3.2.1 Мета роботи .......... 88
3.2.2 Теоретичні відомості .......... 88
3.2.3 Порядок виконання роботи .......... 89
3.3 Лабораторна робота № 3 Spring MVC .......... 96
3.3.1 Мета роботи .......... 96
3.3.2 Теоретичні відомості .......... 96
3.3.3 Порядок виконання роботи .......... 96
3.4 Лабораторна робота № 4 Реалізація мікросервісної архітектури застосування .......... 99
3.4.1 Мета роботи .......... 99
3.4.2 Теоретичні відомості .......... 99
3.4.3 Порядок виконання роботи .......... 99
Рекомендована література .......... 115
Інформаційні ресурси .......... 115

# Вступ

Різновиди програмних систем визначаються різним рівнем складності. Курс «Технології проєктування програмних систем» присвячений вивченню сучасних підходів до проєктування та реалізації корпоративних програмних систем або систем рівня підприємства (enterprise application, КПС). При цьому некоректно інтерпретувати поняття «корпоративної програмної системи» як синонім терміну «велика програмна система», хоча зазвичай подібні застосування є великі за обсягом коду. Більш того, загально відомо, що гарною практикою є зменшення коду «великого застосування» за рахунок удосконалення архітектури системи. Розуміння поняття «корпоративної програмної системи» як такої, що забезпечує діяльність корпорації та некоректне функціонування якої може значно вплинути на відповідну діяльність корпорації, є більш чітким. Отже, визначальними є значущість системи для корпорації, а не обсяг її коду.

У даних методичних вказівках розглядаються етапи процесу проєктування та розробки корпоративних програмних систем, типові підходи та практичні засоби реалізації. Курс орієнтований на використання стандарту Jakarta EE 9.

Курсом передбачено виконання лабораторних робіт з подальшим виконанням курсового проєкту. Лабораторні роботи та курсовий проєкт можут виконуватись в командах складом 1-3 студенти.

Методичні вказівки можуть використовуватися при виконанні дипломної роботи та під час проходження переддипломної практики.

# 1 Загальні теоретичні відомості

## *1.1 Поняття корпоративних програмних систем (систем рівня підприємства)*

Корпоративна програмна система – це програмна система, яка оперує великими обсягами даних за допомогою певних бізнес-правил, орієнтована на конкретний вид діяльності підприємства та від її функціонування часто залежить діяльність підприємства.

Корпоративні системи мають ряд особливостей: вони істотно відрізняються від вбудованих систем, систем керування, телекомунікаційних систем, операційних систем тощо. Прикладом може бути порівняння корпоративної системи із програмною системою сфери телекомунікацій. З певної точки зору, корпоративні системи є набагато простішими, ніж телекомунікаційне програмне забезпечення, оскільки не вирішують проблем забезпечення багатопоточного функціонування, інтеграції апаратних і програмних компонентів. Однак, в іншому КПС є значно складнішими, оскільки передбачають обробку даних (що зазвичай великі за обсягом та передбачають довготривале зберігання) з використанням бізнес-правил, що дуже часто змінюються.

Отже, основними аспектами розробки корпоративних додатків є:

1) Корпоративні програми зазвичай включають необхідність довготривалого (іноді протягом десятиліть) зберігання даних. Дані часто здатні "пережити" кілька поколінь реалізацій прикладних програм, призначених для їх обробки, апаратних засобів, операційних систем. Упродовж цього часу структура даних може піддаватися численним змінам з метою додавання нових даних без впливу на старі. У разі внесення змін до програмно-апаратного забезпечення, дані не знищуються, а переносяться в нове середовище. Обсяг даних, як правило, є великий і завдання маніпуляції цими даними виростає в одну з основних функцій програмної системи.

2) Значна кількість користувачів звертається до даних паралельно, що обов'язково необхідно враховувати для забезпечення продуктивності системи при визначенні нефункціональних вимог. Зрозуміло, що більшість завдань з виконання паралельних низькорівневих транзакцій вирішуються на рівні диспетчера транзакцій систем керування базами даних, проте управління рівнем ізоляції бізнес-транзакцій виконується на рівні прикладного застосування та суттєво впливає на продуктивність. Високий рівень ізоляції транзакцій підвищує надійність інформаційної системи, усуває ймовірність виникнення конфліктних ситуацій, проте значно знижує продуктивність системи.

3) Велика кількість ролей користувачів застосування передбачає необхідність реалізації різних варіантів екранного інтерфейсу та можливість подання даних в різних формах, зручних для різних груп користувачів.

4) Бізнес-логіка корпоративних систем складна та характеризується постійним внесенням змін. Це, наприклад, різні умови оплати кредиту для клієнтів

з урахуванням різних критеріїв. Велика кількість таких умов роблять бізнес-логіку програмної системи складною, заплутаною і такою, що складно піддається сприйняттю. Для порівняння, операційної системи має строгу логіку реалізації.

5) Некоректне функціонування системи або невчасна реалізація функціоналу та впровадження системи в експлуатацію означає значні фінансові затрати.

До числа корпоративних додатків відносяться, наприклад, системи бронювання квитків, програми забезпечення фінансових процесів, системи торгового, бухгалтерського обліку, збереження та обробки медичних карт пацієнтів, системи документообігу і т.і. Корпоративними додатками не є засоби обробки тексту, керування пристроями, телекомунікаційні системи, операційні системи, компілятори, ігри і т.д.

Зазначені особливості корпоративних програмних систем визначають такі нефункціональні вимоги до них: надійність, захищеність, масштабованість, гнучкість інтеграції та супроводження.

## *1.2 Проєктування багатошарової архітектури застосування*

Обираючи архітектуру застосування та шаблони реалізації, слід брати до уваги особливості конкретної системи. Обране типове рішення найчастіше вимагає модернізації для конкретного випадку.

При проєктуванні архітектури застосування ефективним є застосування типових підходів до виділення основних компонентів, архітектурних шарів і схеми взаємодії між ними.

### 1.2.1 Особливості використання архітектурних шарів

Концепція шарів (layers) – одна з загальновживаних, що використовується розробниками програмного забезпечення для розділення складних систем на прості частини. Відрізняють поняття горизонтальної та вертикальної декомпозиції систем.

Шар вищого рівня зазвичай «знає» про особливості реалізації шару нижнього рівня та може користуватися сервісами нижнього шару. Однак нижні шари не знають про особливості реалізації верхніх шарів та не можуть звертається до їх сервісів. Визначають поняття чіткого та нечіткого розшарування системи. Чітке розмежування передбачає, що кожний проміжний шар "приховує" нижній шар від верхнього: наприклад, шар 4 користується послугами шару 3, який звертається до шару 2, але шар 4 не знає про існування шару 2. Поняття нечіткого розмежування передбачає можливість звернення із даного шару до сервісів будь-якого нижнього шару.

Розподілення системи на шари надає ряд переваг:

- окремий шар можна розглядати як єдине самодостатнє ціле;
- можна вибирати альтернативну реалізацію базових шарів;
- залежності між шарами можна звести до мінімуму;
- кожен шар є вдалим кандидатом на стандартизацію.

Розшарування архітектури має й певні недоліки:

1) шари здатні інкапсулювати особливості реалізації застосування, однак модифікація одного шару зазвичай пов'язана з необхідністю внесення каскадних змін в інші шари. Наприклад, поле, додане в таблицю бази даних, зазвичай має бути відтвореним в графічному інтерфейсі та знайти відповідне відображення в кожному проміжному шарі.

2) Наявність надлишкових шарів знижує продуктивність системи. При передачі об’єктів від шару до шару вони зазвичай потребують перетворення з одного вигляду до іншого. Однак, незважаючи на це, інкапсуляція нижніх шарів дозволяє досягти істотних переваг. Наприклад, оптимізація шару транзакцій зазвичай призводить до підвищення продуктивності верхніх шарів.

Найважче при використанні архітектурних шарів – це визначення меж відповідальності кожного шару.

### 1.2.2 Класична модель розшарування системи

Класична архітектура передбачає наявність 3 основних шарів: шар представлення (presentation), шар бізнес-логіки (домен, domain) та шар інтеграції (джерело даних, data source).

До основних функцій шару представлення відносяться відображення інформації та інтерпретація команд, що вводяться користувачем, з перетворенням їх у відповідні запити в контексті домену (бізнес-логіки) та джерела даних.

Шар джерела даних реалізує підмножину функцій, що забезпечують взаємодію застосування зі сторонніми системами, які реалізують функції, що використовуються додатком. Код цієї категорії несе відповідальність за моніторинг транзакцій, взаємодію з іншими застосуваннями, обмін повідомленнями та ін. Для більшості корпоративних додатків основна частина низькорівневої логіки джерела даних зосереджена в коді СУБД та відповідному провайдері доступу до БД.

Логіка домену (бізнес-логіка або логіка предметної області) описує основні функції застосування, призначені для досягнення поставленої перед ним мети. До таких функцій відносяться обчислення для даних, перевірка даних та обробка запитів, що надходять від шару представлення, а також передача інформації шару джерела даних.

Класична архітектура корпоративної програмної системи, що характеризується складністю бізнес-логіки, представлена на рисунку 1.1.

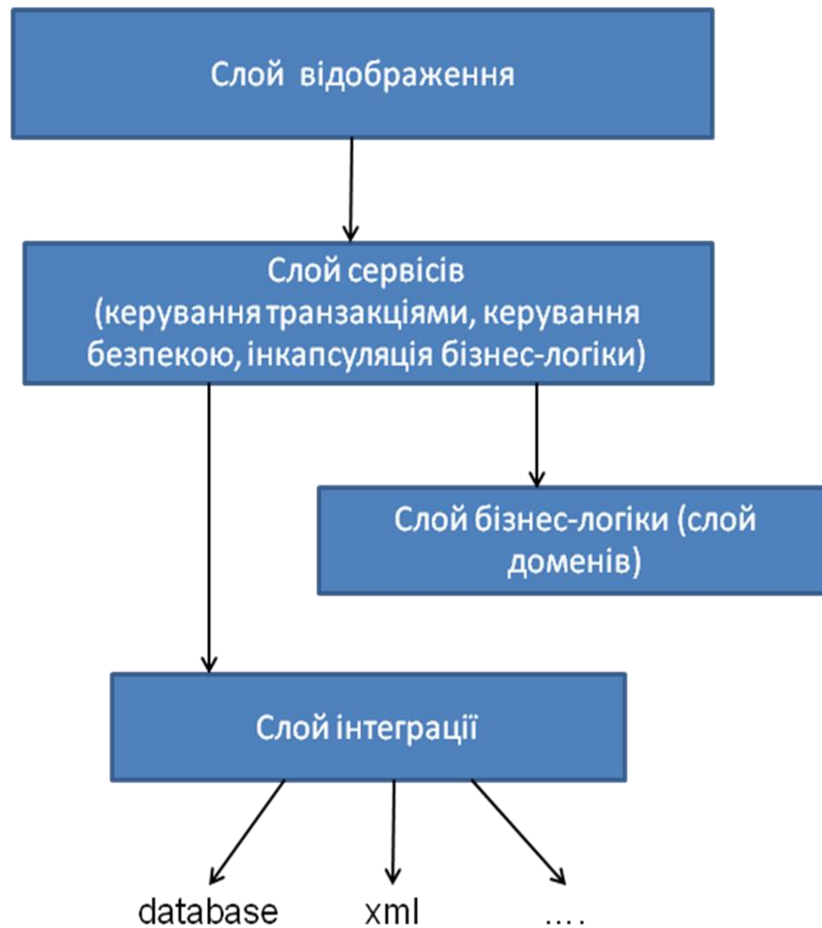


Рисунок 1.1 – Класична архітектура корпоративної програмної системи

Модель трьох шарів – не єдина, існують альтернативні схеми "розшарування" коду. Хоча три основні шари – відображення, бізнес-логіка і джерело даних – можна виявити в будь-якому корпоративному додатку, спосіб їх поділу залежить від ступеня складності цього додатка. Форма розшарування може бути довільною, але в будь-якому корпоративному додатку шари повинні бути ідентифіковані.

Часто застосування має декілька варіантів реалізації певного шару.

#### 1.2.3 Вертикальна та горизонтальна декомпозиція системи

Архітектура застосування повинна відображати не тільки вертикальну, але і горизонтальну декомпозицію системи. Вертикальна декомпозиція передбачає виділення архітектурних шарів, в той час як горизонтальна – визначення функціональних модулей в межах одного архітектурного шару. Вертикальна декомпозиція не відображає функціональних особливостей і предметної області застосування, в більшості випадків є стандартна для багатьох корпоративних програмних застосувань.

### 1.2.4 ШАР СЕРВІСІВ В КЛАСИЧНІЙ МОДЕЛІ РОЗШАРУВАННЯ

При проєктуванні корпоративних систем та при виконанні лабораторних робіт та курсового проєкту, відповідно, пропонується відділення логіки застосування від «чистої» логіки домену та введення додаткового шару сервісів (або шару служб).

Проміжні шари – часто, але не обов'язково корисні. Загальний підхід: якщо який-небудь з трьох базових шарів переходить розумну межу складності, модель можна доповнити додатковим шаром, який приймає на себе надлишок функцій.

Ідея виокремлення шару служб з шару предметної області передбачає можливість відмежування логіки процесу від "чистої" бізнес-логіки. Рівень служб зазвичай охоплює логіку, яка відноситься до конкретного варіанту використання системи або забезпечує взаємодію з іншими інфраструктурами (наприклад, за допомогою механізму повідомлень).

Шар служб визначає межі застосування та API сервісів, що надаються шару представлення та іншим клієнтам застосування. Шар інкапсулює бізнес-логіку додатка, здійснює керування транзакціями та керує безпекою застосування. В даному випадку бізнес-логіку розносять по двох категоріях: логіка домену (domain logic), що має справу лише з предметною областю застосування, і логіка застосування (application logic), що визначає допоміжну логіку додатку (наприклад, надсилання повідомлень користувачам або стороннім застосуванням). Такий підхід є ефективним, оскільки розміщення допоміжної логіки додатку в "чистих" класах домену небажано, тому що, по-перше, класи домену допускають меншу ймовірність повторного використання, якщо вони реалізують специфічну логіку застосування та залежать від тих чи інших прикладних інструментальних пакетів, та, по-друге, змішування логіки обох категорій в контексті одних і тих же класів ускладнює можливість нової реалізації логіки застосування, якщо необхідність такого кроку стає очевидною. З цих причин шар служб передбачає розподілення "різної" логіки по окремих шарах, що забезпечує традиційні переваги розшарування, та забезпечує можливість гнучкого використання компонентів шару доменних об'єктів в інших додатках.

Компоненти шару служб можуть використовуватись також для розміщення логіки управління транзакціями та керування безпекою і зазвичай реалізують відповідні сервіси визначення меж транзакцій, перевірки прав доступу користувачів.

Найскладнішим при проєктуванні бізнес-логіки є, ймовірно, вибір того, що саме слід відносити до того чи іншого шару. Наприклад, якщо для web-застосування додається принципово відмінний шар, як-от, інтерфейс командного рядку. Якщо існують сервіси, які доведеться продублювати в даному випадку, то логіка домену "перетікає" в шар представлення. Або можна навести інший приклад: чи потрібно повторювати бізнес-логіку компонентів застосування при необхідності заміни реляційної бази даних XML-файлом?

Уявіть собі додаток, що відображає червоним кольором назви товарів, обсяги продажів яких зросли більш ніж на 10% в порівнянні з рівнем минулого місяця. Припустимо, що відповідна логіка розміщена безпосередньо в шарі представлення, що передбачає зіставлення рівню продажів поточного й минулого місяця та відповідної зміни кольору, якщо різниця перевищує заданий поріг. В даному випадку до шару відображення вводиться зайва логіка предметної області. Для належного розділення шарів потрібно реалізувати метод бізнес-логіки, що визначає факт перевищення рівня продажів певного продукта на задану величину – метод, що здійснює порівняння рівнів продажів за два місяці, та повертає відповідне значення. Надалі у коді шару представлення викликати відповідний метод бізнес-логіки та, в залежності від повернутого значення, прийняти рішення про колір відображення. В цьому випадку логіку сервісу буде розділено на дві частини: логіка виявлення необхідності зміни кольору, і, власне, зміна при відображенні.

### 1.2.5 Взаємодія між шарами

Крім необхідності поділу на шари, існують правила взаємодії між шарами: залежність бізнес-логіки і джерела даних від рівня відображення не допускається, тобто в коді застосування не повинно бути викликів функцій відображення з коду бізнес-логіки або джерела даних. Такий підхід дозволяє спростити можливість адаптації шару уявлення або заміни його альтернативним варіантом зі збереженням базових модулей застосування. Зв'язок між бізнес-логікою і джерелом даних, однак, не настільки однозначний і багато в чому визначається вибором типових рішень для архітектури джерела даних.

Іноді шари організують таким чином, щоб бізнес-логіка повністю приховувала джерело даних від уявлення. Найчастіше, однак, код уявлення може звертатися до джерела даних безпосередньо. Хоча такий варіант менш бездоганний з теоретичної точки зору, в практичному відношенні він нерідко більш зручний і доцільний: код уявлення може інтерпретувати команду користувача, активізувати функції джерела даних для вилучення відповідних порцій інформації з бази даних, звернутися до засобів бізнес-логіки для аналізу цієї інформації і здійснення необхідних розрахунків і потім відобразити результат на екрані.

### 1.2.6 Розміщення шарів

Вище йшлося про логічних шарах, тобто про розчленування системи на окремі частини. Подібне розділення корисно навіть тоді, коли всі шари функціонують на одній машині. Втім, існують ситуації, в яких відмінності в поведінці системи можуть бути обумовлені принципами її фізичної організації. У більшості випадків існує тільки два варіанти розміщення і виконання компонентів корпоративних додатків – на стороні клієнта і на стороні сервері. Найчастіше найпростішим є функціонування коду всіх верств системи на сервері, при цьому максимально спрощуються процедури виправлення помилок і оновлення версій. В цьому випадку не доводиться турбуватися про внесення відповідних змін на всіх

комп'ютерах, про їх сумісність з іншими додатками і синхронізації з серверними компонентами. Загальні аргументи на користь розміщення будь-яких шарів на комп'ютері клієнта полягають в підвищенні швидкості реагування (responsiveness) додатку і в забезпеченні можливості локальної роботи.

Шар джерела даних краще завжди розташовувати на сервері. Виняток становить випадок, коли функції сервера дублюються в коді "дуже товстого" клієнта для забезпечення засобів локального функціонування системи. При цьому передбачається, що зміни, що вносяться до окремих джерел даних на клієнтській машині і на сервері, підлягають синхронізації за допомогою механізму реплікації.

Рішення про те, де повинен функціонувати шар уявлення, здебільшого залежить від переваг у виборі типу призначеного для користувача інтерфейсу. Застосування інтерфейса товстого клієнта автоматично тягне за собою необхідність розміщення шару відображення на клієнтській машині. Використання Web-інтерфейсу означає, що логіка уявлення зосереджена на сервері. Можливий також змішаний варіант розміщення логіки подання – частково на сервері, частково на клієнті.

Розщеплення безлічі бізнес-функцій між сервером і клієнтом виглядає як найгірше рішення, оскільки в загальному випадку ускладнює ідентифікацію того чи іншого фрагмента логіки. Основна причина, що спонукає застосовувати подібну архітектуру, може полягати в тому, що клієнту необхідно володіти тільки якоюсь частиною бізнес-логіки. Головне – ізолювати цю порцію коду в окремому модулі, який не залежить від інших частин системи. Це дасть можливість активізувати код і на комп’ютері клієнта, і на сервері, якщо така потреба виникне пізніше.

Як вже було зазначено, існують різні варіанти реалізації кожного з архітектурних шарів. Існують наступні «класичні» шаблони реалізації бізнес-логіки програми: «Сценарій транзакцій», «Модуль таблиці», «Модель предметної області (домен)».

При виборі варіанта реалізації шару сервісів додатки рекомендуються до розгляду шаблони «Інтерфейс доступу до домену», «Сценарій операції».

Реалізація шару бізнес-логіки у вигляді «Моделі домену» і шару сервісів як «Інтерфейс доступу до домену» дозволяє реалізувати об'єкти бізнес-логіки «чистими», в т. ч. незалежними від реалізації шару інтеграції.

При проєктуванні і розробці компонентів додатка повинен бути застосований принцип інверсії контролю (Inversion of Control, IoC), що буде розглянутий надалі.

## *1.3 Концепції «чистої» архітектури (“Clean” architecture). Domain-Driven Design (DDD) та принципи проєктування S.O.L.I.D*

Найзагальніші концепції, яких слід дотримуватись при розробці корпоративних програмних систем – концепції «чистої» архітектури та Domain-Driven Design, які передбачають:

- Сутності та варіанти використання – ядро застосування;
- Незалежність від фреймворків;
- Незалежність від деталей реалізації (UI, база даних, бібліотеки, ін.);
- Можливість тестування компонентів бізнес-логіки незалежно від інших компонентів застосування;
- Масштабованість;
- Розширюваність.

Принципи проєктування S.O.L.I.D визначають, як функції та дані слід об'єднувати у компоненти/класи та організувати взаємодію між класами. Поняття класу не зводиться до використанням лише об'єктно-орієнтованих технологій. Під класом розуміється модуль, який групує тісно зв'язані між собою функції та дані. SOLID принципи використовуються на рівні цих модулів.

Головна мета принципів – створення компонентів середнього рівня, які:

- Гнучкі до змін;
- Легкі до сприйняття;
- Є компонентами, що можуть бути перевикористані в багатьох програмних системах.

Під компонентами середнього рівня розуміється, що принципи застосовуються на рівні програмного коду окремих модулів, і визначають підходи щодо реалізації всередині модуля. Однак деякі з підходів, як будемо бачити, знаходять відображення на високому архітектурному рівні також.

#### 1.3.1 Принцип Single Responsibility Principle

Один з принципів, що, скоріш за все через назву, найчастіше невірно розуміється розробниками. Single Responsibility Principle згідно назви сприймається як те, що кожен модуль має виконувати певний окремий функціонал. Наприклад, конкретна функція має реалізовувати одну задачу, а не змішувати реалізацію декількох різних задач. Такий підхід є правильним, його використовуємо, коли виконуємо рефакторинг з перетворення великих функцій на декілька окремих, тобто розносячи окремі задачі по окремих функціях. Але не це мається на увазі під Single Responsibility Principle.

Спершу формулювання принципу було таким:

*Модуль повинен мати лише одну причину для внесення змін*

Внесення змін до програмного забезпечення відбувається найчастіше з метою задоволення потреб користувачів та замовників. Саме користувачі та замовники є цією «причиною для внесення змін». Тому є *відповідне перефразування*.

A module should be responsible to one, and only one, user or stakeholder.

Але під «користувачем» мається на увазі група користувачів, які мають певні права доступу до сервісів застосування та певну роль. *A module should be responsible to one, and only one, actor.*

Під модулем розуміємо набір функцій та даних, тісно зв'язаних між собою.

І відповідно розглянемо проблеми, які пов'язані з порушенням цього принципу.

Проблема 1: Випадкове дублювання Accidental Duplication

Наприклад, при проєктуванні модуля обліку відпрацьованих робітником годин та розрахунку відповідних нарахувань заробітної плати основні методи реалізації цієї логіки та метод save() збереження даних до сховища даних, розробниками розміщені в класі Employee (рисунок 1.2).

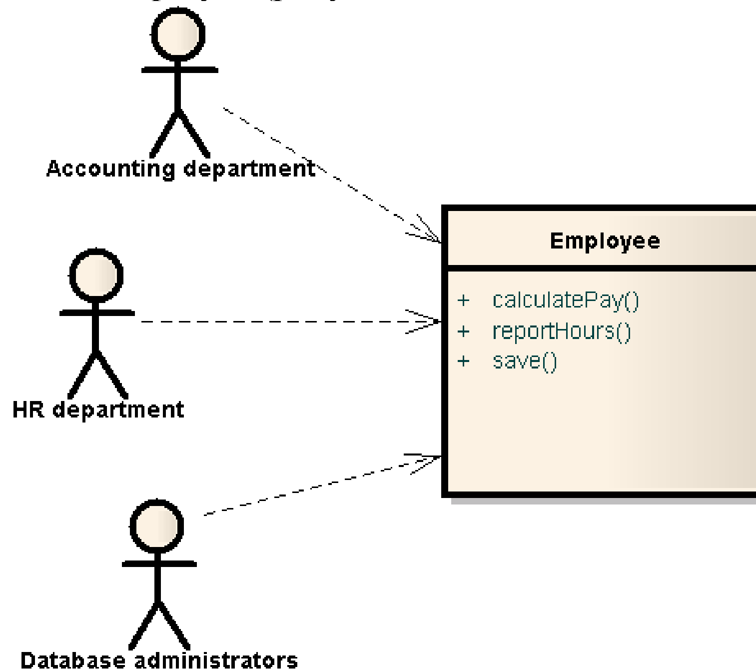


Рисунок 1.2 — Проєктування модулю нарахувань заробітної плати

Але кожен з методів відповідає за реалізацію сервісів для різних груп користувачів. Так, алгоритм реалізації логіки з нарахування заробітної плати, метод calculatePay() може змінюватися під впливом представників фінансового відділу (Accounting department), логіка генерування звіту з відпрацьованих годин – налаштовуватися відділом HR (HR department), і логіка збереження даних відповідно адміністраторами бд (Database administrators). Припустимо, що функції calculatePay () та reportHours () використовують загальний алгоритм розрахунку не понаднормових годин. Відповідно, з метою уникнення дублювання коду, розробники винесли цей алгоритм розрахунку в окрему функцію regularHours () (рисунок 1.3).

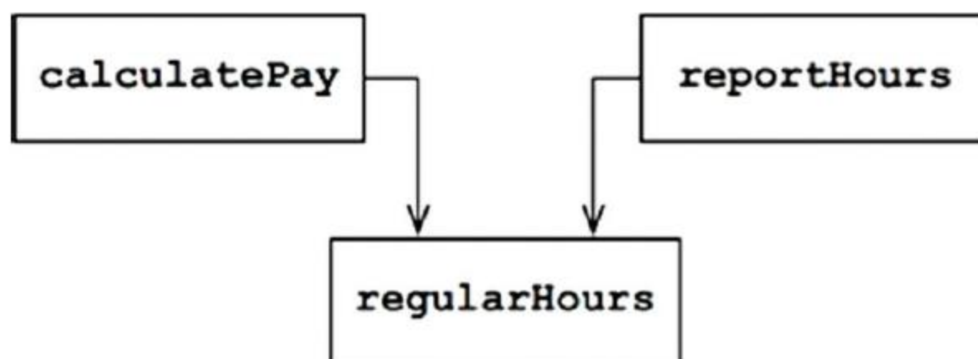


Рисунок 1.3 — Винесення алгоритму розрахунку годин в окрему функцію

Відділ Accounting department вирішує внести певні зміни та налаштування до алгоритму розрахунку не понаднормових годин. При цьому відділ HR не потребує змін до алгоритму, тому що використовує не понаднормові години для іншої задачі.

Розробник отримує відповідну задачу на внесення змін, і знаходить зручний для цього метод regularHours(), який використовує метод calculatePay(). При цьому, розробник забуває/не помічає, що цей метод також використовується із методу reportHours(). Робить необхідні зміни, успішно тестує їх. Відділ Accounting department тестує правильність роботи нового алгоритму нарахування, та оновлена система розгортається на сервері.

HR відділ не знає про внесення відповідних змін, продовжує використовувати звіти, що генерує функція reportHours (), та отримує некоректні дані, які виявляються згодом, і відповідно утворюють фінансові проблеми підприємства.

Подібні речі відбуваються доволі часто.

Проблемою є те, що клас, або будь-який модуль, який включає тісно інтегровані функції та дані, стає відповідальним за реалізацію сервісів для трьох різних ролей користувачів застосування, і може змінюватися під їх впливом.

Проблема 2: Автоматизовані Merges

Зрозуміло, що операція merge буде виконуватися досить часто для модулей програмного коду, що включають велику кількість різних методів. Особливо у випадку, коли методи реалізують сервіси для різних категорій акторів, відповідно модифікуються під впливом різних користувачів, зміни вносяться різними розробниками, можливо навіть з різних команд. Це є аспекти використання систем керування версіями проєкту та використання репозиторію проєкту.

Наприклад, адміністратори бд вносять певні зміни до структури таблиці Employee. В той же час відділ HR вносить зміни до формату годин для генерації звіту.

Відповідно, два різні розробники, можливо з двох різних команд, клонують клас Employee та вносять до нього зміни. Бачимо, що клонування відбувається майже одночасно, розробник з команди адміністраторів бд фіксує зміни в центральному репозиторії раніше. Розробник відділу HR після коміту в локальний репозиторій, оновлення локальної гілки та поєднання власних змін, отримує автоматичний merge, який загалом становить певний ризик, тому що в такому випадку розробник зазвичай не бачить зміни, внесені іншими розробниками в той самий модуль, до якого він фіксує. Оскільки зміни стосуються різних частин модуля, вони об’єднуються та фіксуються автоматично. У випадку виникнення конфлікту, коли розробник має переглянути конфліктуючі зміни, та зробити відповідну модифікацію, цієї проблеми не буде. Але у випадку автоматичних merge вона є.

Такі автоматичні merge будуть відбуватися досить часто, звісно, що логіка роботи сервісів Accounting department також може бути некоректно модифікована.

Знов, проблема в тому, що різні актори впливають на функціональні модулі, що тісно пов'язані між собою.

Рішення: Існують різні підходи щодо вирішення цієї проблеми. По-перше, очевидним є необхідність відокремлення даних від функцій, винесення EmployeeData в окремий клас, що включає в себе лише дані, з розподіленим доступом до нього (рисунок 1.4).

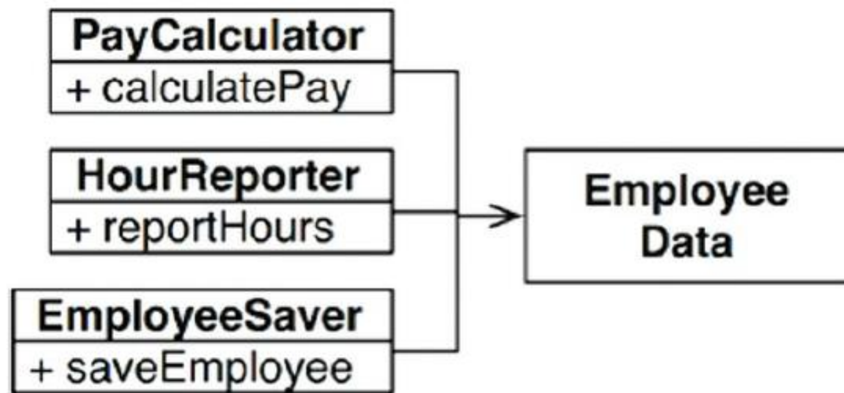


Рисунок 1.4 – Рішення: відокремлення даних від функцій

Рознесення методів в три окремі незалежні класи, які не пов'язані між собою та не знають один про одного. Таким чином, будь-яке випадкове дублювання не допускається.

Недоліком є те, що розробники мають створювати та відслідковувати об'єкти трьох різних класів. Окрім того, маємо проблему ефективності виконання застосування, яка дещо знижується при створення в пам'яті великої кількості допоміжних об'єктів. Але цю проблему зараз не розглядаємо, вирішується шляхом використання відповідних апаратних ресурсів.

Рішенням може бути використання фасаду (рисунок 1.5).

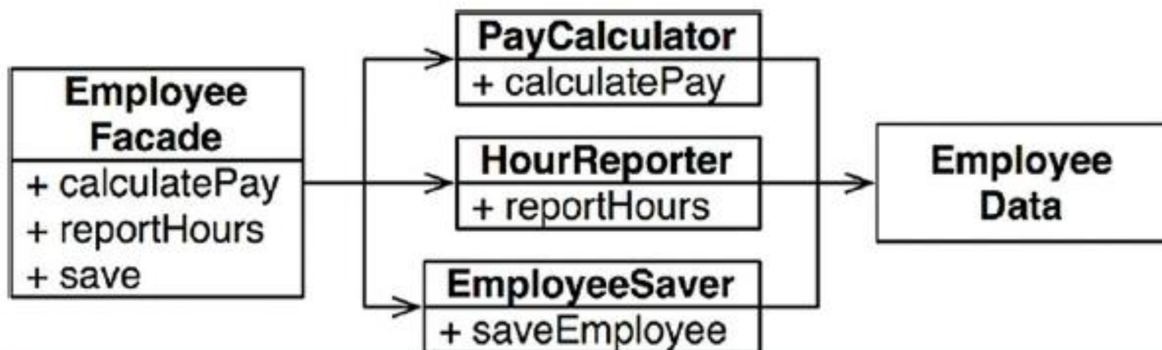


Рисунок 1.5 – Використання фасаду

EmployeeFacade містить дуже мало коду, він лише відповідає за ініціалізацію та перенаправлення запросів до методів певних класів.

Інколи найважливіші бізнес-методи розміщують ближче до даних. Так, клас Employee може включати в себе дані та реалізацію основної бізнес-логіки (в даному випадку, метод calculatePay), та використовуватися в якості фасаду для доступу до інших функцій (рисунок 1.6).

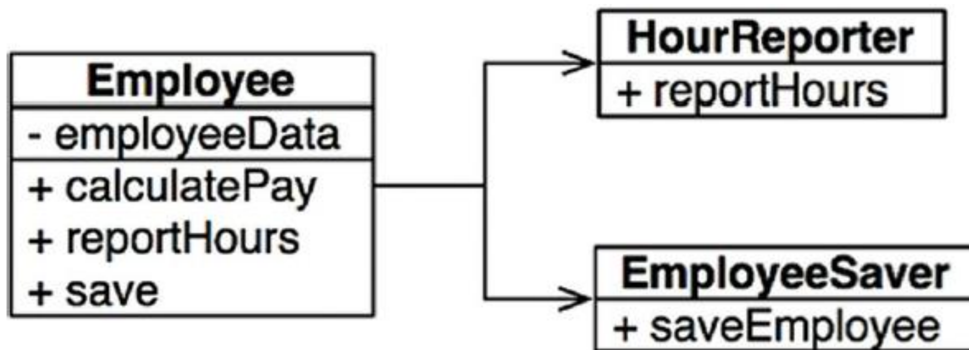

Рисунок 1.6 – Розміщення бізнес-методів «ближче» до даних

Звісно, розміщення в класі лише одного методу є абстрактним. Реалізація кожного з бізнес-методів буде передбачати декілька методів, але приватних, в межах певного класу.

Розглянутий принцип має застосування на більш високому архітектурному рівні також для визначення функціоналу та меж архітектурних шарів.

### 1.3.2 Принцип Open-Closed Principle

Розширення функціоналу, тобто реалізація нових сервісів існуючого програмного забезпечення має бути можливим без необхідності внесення змін до існуючого коду. Це вважається фундаментальною задачею проєктування архітектури. Якщо незначне розширення вимог до ПЗ потребує внесення значних змін до існуючого коду системи, це сигналізує про проблеми в архітектурі.

Наприклад, є програмне застосування, яке виводить результуючі фінансові звіти на веб-сторінку. На веб-сторінці для виводу даних є скролінг, від'ємні числа мають відображатися наприклад червоним кольором.

Через певний час замовник просить виводити ті самі дані фінансових звітів, але у вигляді звіту, який може бути роздрукований на чорно-білому принтері. При цьому звіт має бути строго відформатований, мати певні відступи, формати заголовків. Від'ємні числа мають виводитися в дужках. Звісно, має бути написаний новий код. Питання в тому, наскільки багато змін треба внести до існуючого коду. У випадку правильної архітектури буде забезпечено мінімум змін. В ідеалі, повна відсутність необхідності змін існуючого коду.

Використовуючи принцип Single Responsibility, отримаємо представлений потік даних. Деяке бізнес-правило Financial Analyzer аналізує дані та генерує дані для звіту Financial Report Data, які потім форматуються відповідним чином сервісами Web Reporter та Print Reporter. Головна ідея полягає в тому, що генерація звіту включає дві незалежні задачі: розрахунок даних для звіту та відображення даних в різних форматах – в даному випадку веб-формат та формат для роздрукування на принтері (рисунок 1.7).

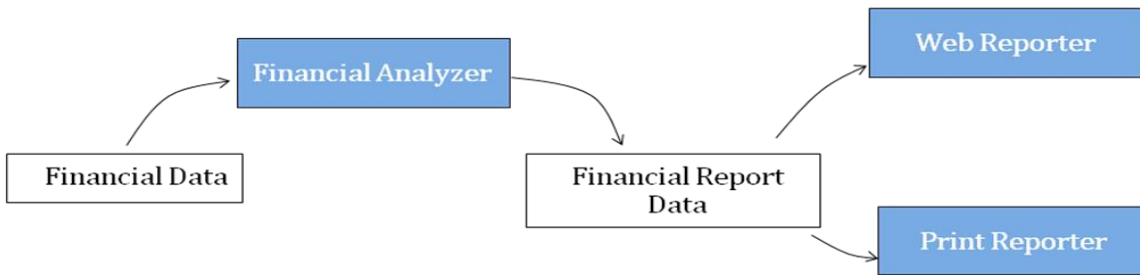


Рисунок 1.7 – Застосування принципу Single Responsibility

Залежності між класами на рівні коду мають бути організовані таким чином, щоб внесення змін до одного з модулей не спричиняло змін в інших модулях. Також, розширення логіки, тобто додавання нового функціоналу, має бути реалізованим без модифікації існуючих модулей.

Представлено організацію компонентів архітектури застосування відповідно підходів чистої архітектури (рисунок 1.8).



Рисунок 1.8 – Приклад реалізації архітектури

Клас Financial Data Mapper знає про існування та реалізує інтерфейс Financial Data Gateway, але Financial Data Gateway не знає про існування Financial Data Mapper.

Введення додаткових компонентів в представленій архітектурі зроблено з метою забезпечення потрібного напрямку залежностей між компонентами. Наприклад, інтерфейс FinancialDataGateway між класами FinancialReportGenerator та FinancialDataMapper існує для інвертування залежності, яка в іншому випадку була б спрямована від компоненту Iterator до компоненту Database. Таке саме призначення і інтерфейсу FinancialReportPresenter та двох інтерфейсів View.

Інтерфейс FinancialReportRequester існує не лише для забезпечення незалежності Interactor від змін у Controller, але й для незалежності Controller від внутрішньої реалізації компонентів Iterator, залежність лише від інтерфейсу компонента Iterator. Тобто, якщо відбуваються зміни у внутрішній реалізації бізнес-логіки, інтерфейс FinancialReportRequester залишається без змін, відповідно не потрібно змінювати Controller. Принцип стабільності, тобто незмінності інтерфейсу, будемо обговорювати далі. Звісно, інколи зміни у бізнес-логіці можуть потребувати змін до її інтерфейсу, але такі ситуації необхідно намагатися обходити.

Варто звернути увагу, що всі зв'язки між компонентами є односпрямованими. І є спрямованими в сторону тих компонентів, що мають бути захищеними від внесення змін.

Якщо компонент А має бути захищеним від змін в компоненті B, то компонент B має бути залежним від компонента А, не навпаки (рисунок 1.9).

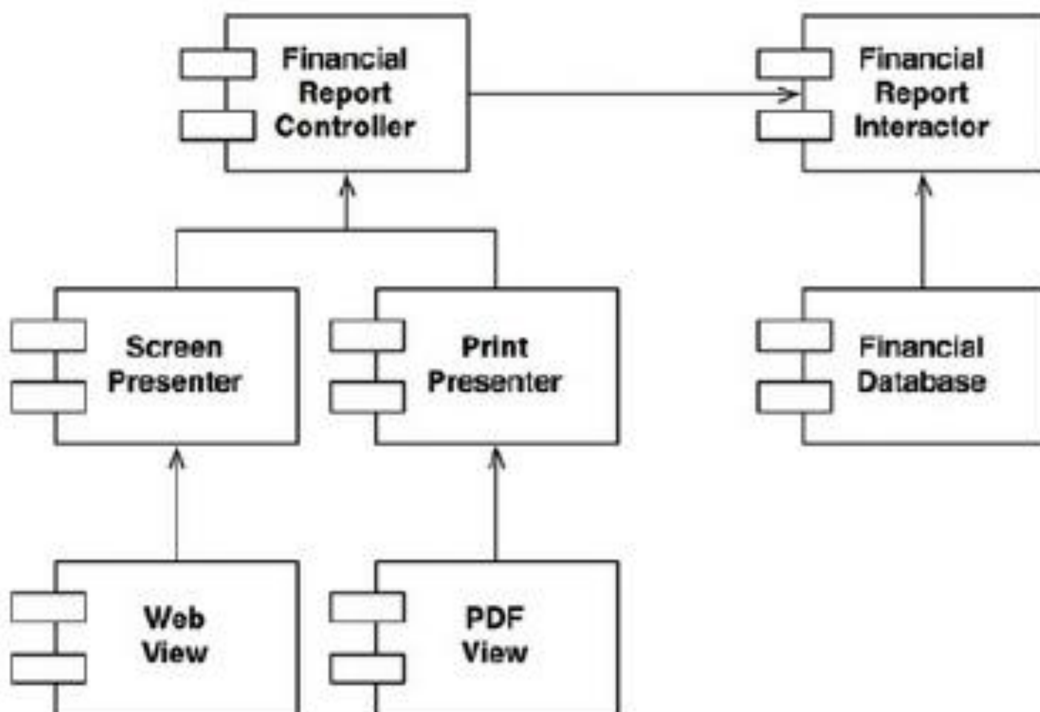


Рисунок 1.9 – Зв'язки між компонентами архітектури

Controller має бути захищений від змін в класах Presenters. Класи Presenters мають бути захищені від змін в компонентах View. Interactor має бути незалежним від змін в будь-яких інших компонентах. Компонент Interactor є ядром (бізнес-логікою) системи. Головний принцип проєктування архітектури полягає в тому, що зміни в організації сховища даних, або відображення даних, не мають

впливати на класи бізнес-логіки системи. Інтерфейс системи може мати різні реалізації, сховище даних може замінюватись якоюсь альтернативною реалізацією, при цьому ядро системи не потребує змін. Інтерфейс системи, сховище даних, використовувані специфічні бібліотеки, фреймворки є деталями реалізації та можуть замінюватися альтернативною реалізацією.

Таким чином, отримуємо ієрархію залежностей між компонентами та ієрархію захищеності від змін на рівні шарів архітектури.

Забезпечення гнучкої розширюваності систем из мінімальною кількістю змін до існуючого коду є важливим принципом проєктування архітектури системи. Визначення незалежних модулей застосування базується на тому, як, коли та чому можуть відбуватися зміни у віповідних модулях та подальшого визначення ієрархії залежностей між компонентами. Високорівневі компоненти в ієрархії мають бути захищені від змін в компонентах нижчого рівня.

### 1.3.3 Принцип Liskov Substitution Principle

Був визначений в 1988р. Барбарою Лісков, амер. вченою, лауретом премії Тьюринга. Принцип визначає правило заміщення типу. Якщо для кожного об'єкту o1 типу S існує об'єкт o2 типу T такий, що для всіх програм P визначених для об'єктів типу T, поведінка програм не зміниться при заміні o2 на o1, то тип S є підтипом типу T.

Приклад, використання наслідування. Два підтипи класу License: PersonalLicense та BusinessLicense, реалізують різні алгоритми розрахунку вартості ліцензії (метод calcFee) (рисунок 1.10).

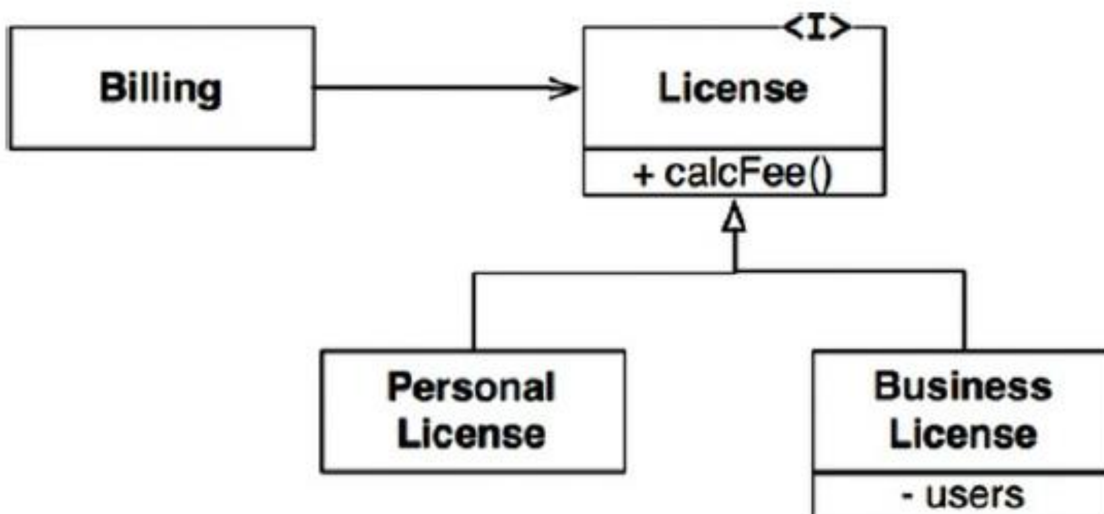


Рисунок 1.10 – Приклад використання наслідування

Клас License використовується в клієнті Billing. Це є приклад використання відомого патерну «стратегія» для визначення різних варіантів реалізації бізнес-правил. Таке проєктування відповідає принципам LSP. Обидва підтипи PersonalLicense та BusinessLicense можуть бути використані для заміщення типу License, поведінка клієнтського застосування Billing від цього не зміниться.

Відомою є проблема квадрата/прямокутника. Клас квадрата Square не є коректним підтипом класу прямокутника Rectangle, тому що логіка зміни основних, з точки зору бізнес-логіки, параметрів класу висоти та ширини, не є однаковою. Поле висота та ширина класу Rectangle змінюються незалежно один від одного; навпаки, ці параметри для об'єктів класу Square мають змінюватися одночасно. І це є основна з точки зору бізнес-логіки поведінка цих об'єктів, яка не підлягає наслідуванню.

Тому при використанні представленої на рисунку 1.11 ієрархії класів із клієнтського коду можуть виникати конфліктні ситуації.

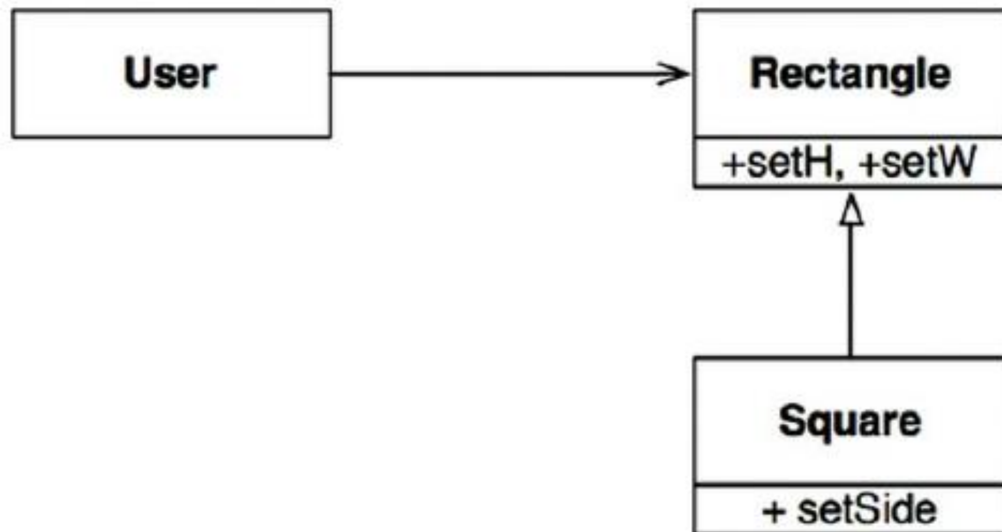


Рисунок 1.11 – Некоректне використання зв'язку наслідування

Наприклад, якщо у клієнті визначено тестовий метод перевірки алгоритму розрахунку площі фігури. Метод розрахунку площі реалізований у класі Rectangle, і в тестовому методі визначено тип Rectangle. Але у випадку заміщення типу Rectangle на підтип Square при ініціалізації об'єкту, відповідний тест розрахунку площі не пройде успішно.

```
Rectangle r = …
r.setW (5);
r.setH (2);
assert (r.area () ==10);
```

Тобто, заміна базового типу на підтип, призводить до некоректної роботи класу User. Рішенням може бути використання перевірки if для визначення типу створеного об'єкту:

```
if (r.getClass () ==..Rectangle) …{assert ...}
if (r.getClass () ==.. Square ) …{assert ...}
```

Але велика кількість if для визначення контексту суперечить принципам чистого коду. Отже визначення класу Square підтипом класу Rectangle є некоректним, оскільки ці типи не можуть бути замінені.

Є багато суперечок з приводу виникнення зазначеної проблеми та можливості її усунення. Загалом кажучи, можуть бути визначені окремі тестові методи для обчислення площі прямокутника та квадрата.

Але можна навести інший приклад виникнення конфліктної ситуації. Якщо, наприклад, необхідно реалізувати метод зі збільшення проці фігури у 10 разів, то маємо різну логіки реалізації в залежності від конкретного типу об'єкту. І знов різну логіку можна реалізувати або засобами перевірок через конструкцію if, або шляхом перевизначення методу для класу підтипу. Перший випадок вже казали, не відповідає принципам чистої архітектури. В другому випадку, виникає питання, чи є спільнє між класами Rectangle та Square з точки зору реалізації функціоналу, якщо функціонал має бути повністю перевизначений у класі підтипу. Відповідно, питання коректності зв'язку наслідування, який є тривіальним на рівні реальних об'єктів, зв'язок наслідування є недоцільним та проблемним з точки зору реалізації.

Принцип підтипу, If you made Square derive from Rectangle, then a Square should be usable anywhere you expect a rectangle, не витримується.

Високорівневим прикладом використання підходу є використання на рівні визначення інтерфейсів реалізацій. Під інтерфейсом може розумітися Java-інтерфейс, що має реалізацію декількома класами. Або це можуть бути сервіси, що реалізують певний REST інтерфейс.

Наприклад, є система пошуку та замовлення таксі клієнтом, що об'єднує різні диспетчерські служби таксі. Після вибору таксі клієнтом, система виконує замовлення шляхом виклику restful сервісу відповідної диспетчерської служби.

З бази даних отримуємо URI відповідної служби, відправляємо відповідний запит для виклику сервісу замовлення, наприклад: `PUT /pickupAddress/24 Maple St. /pickupTime/%s /destination/%s`.

. Відповідно отримуємо ситуацію, коли всі диспетчерські служби мають відповідати однаковому REST інтерфейсу, тобто однаково приймати та обробляти параметри pickupAddress, pickupTime та destination.

Але, уявімо, якщо одна з компаній наприклад змінює назву параметра destination на dest. Відповідно, необхідна спеціальна обробка випадку виклику сервісу замовлення через цю компанію. Напростіший спосіб, додати відповідний if… І знову ж жоден архітектор не дозволить використання такої конструкції, коли текстова назва зазначається на рівні коду застосування. Окрім отримання проблем з точки зору архітектури, в цьому випадку отримує проблему і з точки зору безпеки застосування. Рішенням має бути реалізація модуля формування відповідного запиту, що використовує конфігураційні дані для певного URI. Відповідний приклад конфігураційних даних:

| URI | Dispatch Format |
|---|---|
| kruiz.com | /pickupAddress/%s/pickupTime/%s/dest/%s |
| *.* | /pickupAddress/%s/pickupTime/%s/destination/%s |

Таким чином, реалізовано механізм, який враховує факт невідповідності інтерфейсів різних restful сервісів. Принцип може знаходити багато інших застосувань на архітектурному рівні шляхом включення в архітектуру інших подібних механізмів.

#### 1.3.4 Принцип Interface Segregation Principle

Принцип передбачає рознесення методів, що реалізують не інтегровану функціональність для різних груп користувачів, між різними інтерфейсами (рисунок 1.12, рисунок 1.13)

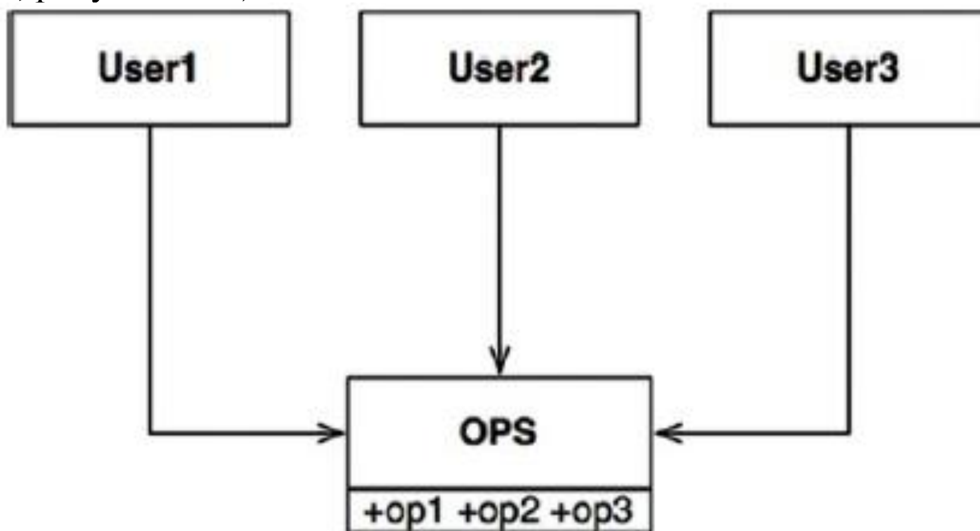


Рисунок 1.12 – Приклад проєктування класів

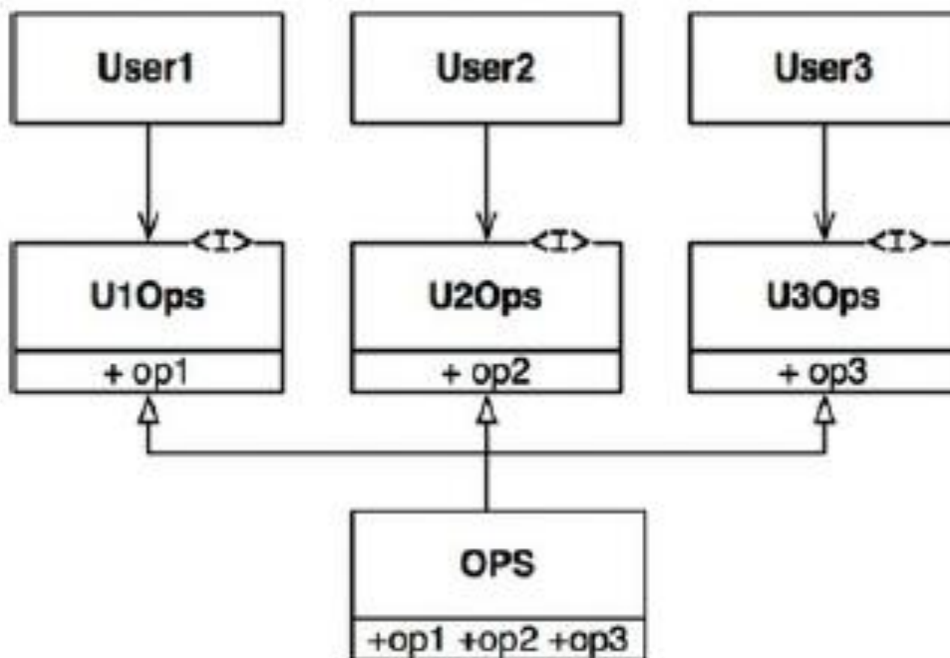


Рисунок 1.13 – Застосування принципу розділення інтерфейсів

### 1.3.5 INVERSION OF CONTROL ТА DEPENDENCY INVERSION

Головна ідея **Dependency Inversion Principle** є те, що найбільш гнучкі системи є такі, які на рівні вихідного коду визначають залежності тільки від абстракцій, не від конкретних реалізацій.

В мовах програмування зі статичною типізацію таких як Java, це значить, що use, import, include statements мають посилатися лише на модулі, що включають інтерфейси, абстрактні класи або інші абстракції, а не на конкретні реалізації. Такий же підхід має використовуватися у мовах з динамічною типізацією типу Ruby, Python, але в цьому випадку мається на увазі залежність від будь-якого модуля, що включає реалізацію функцій, що викликаються.

Звичайно, абсолютне дотримання цього правила не є реалістичним. Наприклад, клас String є конкретною реалізацією і його складно визначити через абстракцію. І це не є потрібним, тому що клас String є стабільним. Зміни до класу відбуваються дуже рідко, та строго контролюються. Розробникам немає потреби враховувати часті та непередбачувані зміни до класу String. Тому, виконання принципу інверсії може бути проігноровroutine для стабільних платформ та утиліт. Можна розглядати їх як такі, що не будуть змінюватися.

Принцип стосується залежності від нестабільних модулей, що найчастіше є складовими компонентами системи або зовнішні бібліотеки, які знаходяться в процесі розробки та підлягають частим змінам.

Стабільні абстракції. Кожна зміна до абстрактного інтерфейсу потребує внесення змін до його реалізації. Але навпаки, внесення змін до конкретної реалізації практично ніколи не вимагає змін до інтерфейсу, який вона реалізує. Забезпечення стабільності інтерфейсів є основним правилом розробки ПЗ.

Таким чином, стабільне ПЗ уникає залежності від нестабільних реалізацій, а використовує стабільні абстрактні інтерфейси. Такий підхід зводиться до реалізацій наступних coding practices:

- Don’t refer to volatile concrete classes.
  Стосується всіх мов програмування, вводить обмеження на механізм ініціалізації об’єктів. Рекомендується використання окремих компонентів (Abstract Factories) для реалізації задач створення об’єктів застосування;
- Don’t derive from volatile concrete classes.
  Обережне введення зв’язку наслідування, оскільки це є однією з найстрогіших залежностей на рівні коду.
- Don’t override concrete functions.
  При перевизначенні реалізації конкретного методу, отримуємо додаткові залежності на рівні коду. Для гнучкого керування залежностями визначайте функцію/метод як абстрактний, і створюйте різні його реалізації.
- Never mention the name of anything concrete and volatile.

> Будь-які конкретні значення, назви конкретних реалізацій тощо, не мають визначатися явно на рівні програмного коду, має використовуватися декларативний підхід для їх визначення - тобто зовнішні конфігураційні файли, значення яких може змінюватися незалежно від програмного коду.

Проблема керування залежностями між компонентами породжує проблему створення екземплярів нестабільних конкретних реалізацій, оскільки задача створення об'єкту вимагає залежності даного модуля від модуля визначення конкретної реалізації класу об'єкту. Відомим підходом в ООП мовах є використання **абстрактної фабрики** для гнучкого керування залежностями між компонентами застосування.

Напр., клас Application (рисунок 1.14) використовує клас Concrete Impl через інтерфейс Service. Але в класс Application якимось чином необхідно виконати ініціалізацію екземпляру класу ConcreteImpl. Для реалізації цього без визначення залежності від класу ConcreteImpl, клас Application викликає метод інтерфейсу фабрики для створення екземпляру сервісу. Цей метод реалізовано у класі реалізації інтерфейсу фабрики. Метод створює об'єкт класу ConcreteImpl та повертає його як реалізацію інтерфейсу Service.

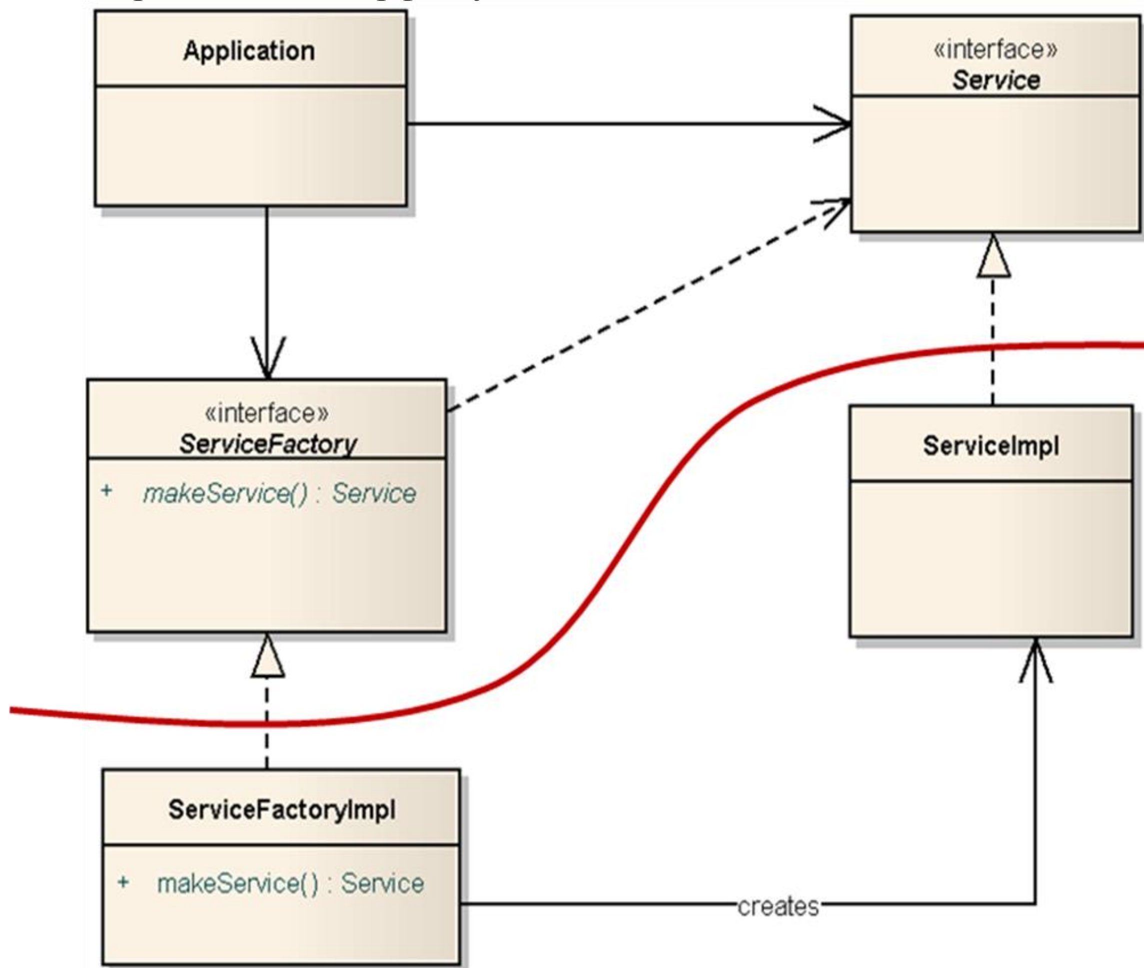


Рисунок 1.14 — Застосування принципу інверсії контролю

Бачимо лінію, яка визначає архітектурну межу, що відокремлює абстрактні реалізації від конкретних. Всі залежності на рівні коду перетинають цю лінію в одному напрямку, спрямованому до абстрактних компонентів. Абстрактні компоненти включають високорівневе API сервісів застосування. Конкретні компоненти містять всі деталі реалізації відповідних сервісів та бізнес-правил.

Варто звернути увагу, що лінія контролю між об'єктами спрямована в напрямку, протилежному до напрямку залежностей на рівні коду. Залежності на рівні програмного коду інвертовані (спрямовані в протилежному напрямку) до напрямку лінії контролю – звідси назва, інверсія залежностей.

Якщо розглядати на рівні загальної архітектури системи, то такий підхід дозволяє усунути велику кількість зв'язків між компонентами архітектури, яка б існувала без застосування принципу інверсії залежностей. В такому випадки все компоненти застосування для отримання екземплярів інших компонентів, звертаються до окремого модуля фабрики, який реалізує механізм створення потрібних об'єктів. Таким чином, замість великої кількості залежностей, отримуємо лише залежність від інтерфейсу фабрики.

Зрозуміло, що абсолютне дотримання інверсії залежностей для всіх компонентів застосування не є можливим. Певні компоненти (але це не компоненти ядра системи, а компоненти зовнішніх, неосновних шарів системи, що є деталями реалізації) можуть включати в себе прямі залежності від конкретних реалізацій. Так, в даному випадку має бути якийсь головний модуль (якась main function), яка виконує ініціалізацію об'єкту конкретної реалізації інтерфейсу фабрики, і надає доступ до цього об'єкту іншим компонентам застосування.

Правило перетину архітектурної межі таке, що всі залежності однаково спрямовані, в напрямку більш абстрактних компонентів, визначає правило залежностей на рівні всієї архітектури системи.

Відомим є також термін інверсії контролю. Основна ідея принципу «інверсія контролю» (**Inversion of Control**, IoC) – вирішення тієї ж проблеми створення екземплярів конкретних класів без визначення залежностей від конкретних реалізацій шляхом використання спеціального IoC контейнеру компонентів, який керує життєвим циклом об'єктів застосування, і відповідно реалізує механізми створення необхідних реалізацій об'єктів при керуванні залежностями між ними.

Таким чином, компонент не створює екземпляри об'єктів, від яких він залежить, самостійно (рисунок 1.15), а отримує їх ззовні (рисунок 1.16). Це реалізовано сервісами контейнеру, на якому розгорнуті та виконуються компоненти застосування.

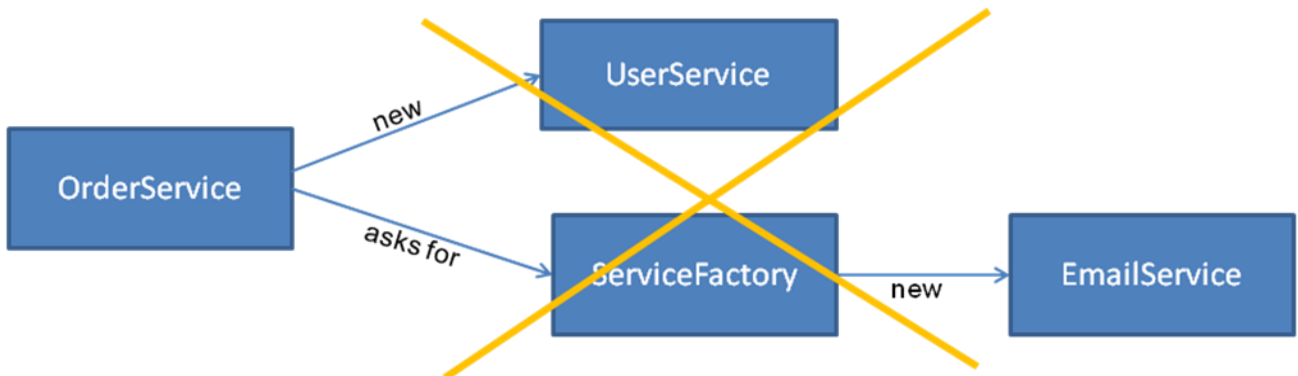


Рисунок 1.15 – Відсутність принципу інверсії залежностей

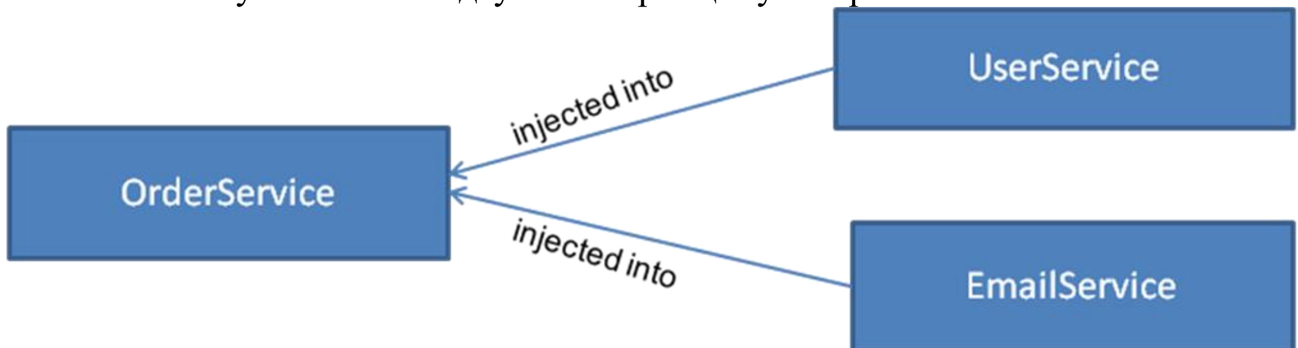


Рисунок 1.16 – Застосування принципу інверсії залежностей

Фактично, замість власної реалізації сервісів фабрики компонентів, використовується готова реалізація сервісами певного фреймворку, що реалізує IoC.

Для керування залежностями використовується принцип декларативного визначення залежностей. Тобто, в залежності від фреймворку, за допомогою анотацій на рівні коду, або у зовнішніх файлах конфігурації, визначаються залежності даного компонента від конкретних реалізацій сторонніх компонентів. У випадку наявності декількох реалізацій одного й того ж компоненту, визначення необхідної реалізації так само виконується декларативно, не на рівні коду, а в зовнішніх файлах конфігурації.

На відміну від використання шаблону фабрики, коли залишається залежність від методів компоненту фабрики, залишається код ініціалізації, який передбачає виклик відповідних методів фабрики та передачі необхідного контексту, для визначення типу конкретної реалізації, необхідної для створення. У випадку використання IoC контейнеру отримуємо переваги чистого коду, коли компоненти взагалі не виконують задачу ініціалізації сторонніх об’єктів.

Забезпечення можливості гнучкого модульного тестування логіки компонентів, і заміни при цьому сторонніх об’єктів на тестові mock-реалізації для незалежного тестування конкретного модуля бізнес-логіки. Наприклад, для забезпечення чистого тестування компонентів контролерів без неявного тестування при цьому компонентів шару інтеграції з базою даних, відповідні компоненти шару інтеграції замінюються на mock-реалізацію бази даних засобами колекцій. Для інтеграційного тестування та при розгортанні системи декларативно замінюємо реалізацію на production.

### *1.4 Висновки*

Дотримання концепцій чистої архітектури та наяність класичної моделі розшарування системи (або модифікацій даної моделі) є обов'язковою умовою виконання лабораторних робіт та курсового проєкту з дисципліни.

Реалізацію застосування можна виконувати з використанням мови програмування Java та відповідних технологій стандарту Jakarta EE або фреймворку Spring або іншої мови програмування з використанням альтернативних технологій та фреймворків реалізації систем рівня підприємства. Нижче наведено приклади реалізації веб-застосувань з використанням технологій стандарту Jakarta EE та фреймворку Spring.

# 2 Реалізація корпоративної програмної системи з використанням стандарту Jakarta EE

Стандарт Jakarta EE — це набір стандартних API для розробки серверних Java-застосувань, в тому числі традиційних корпоративних застосувань та мікросервісів.

У 2017 році Oracle оголосила про передачу Java EE фонду Eclipse Foundation, однією з умов переходу було перейменування технології. Спільнота обрала Jakarta EE новою назвою для Java EE під керівництвом Eclipse Foundation.

Основна перевага Java EE / Jakarta EE полягає в тому, що це є стандарт, для якого існує декілька реалізацій, тому код, написаний на основі однієї реалізації, можна легко перенести в іншу реалізацію з невеликими змінами або без них.

Перенесення Java EE на незалежну від постачальника базу, таку як Eclipse, має перевагу в тому, що жодна компанія чи організація не має впливу на стандарт. Крім того, раніше організації, зацікавлені в сертифікації своїх продуктів як «сертифікованих Java EE», повинні були заплатити значні суми, щоб отримати доступ до інструменту сертифікації Technology Compatibility Kit (TCK), TCK став відкритий і розміщений у вільному доступі, що надає можливість розробки додаткових продуктів, сумісних із Jakarta EE. Завдяки цим та іншим перевагам перехід Java EE до Eclipse Foundation було підтримано спільнотою Java.

## *2.1 Лабораторна робота № 1*
## *Проєктування архітектури та слою бізнес-логіки корпоративного програмного застосування*

### 2.1.1 МЕТА РОБОТИ

Огляд та застосування типових підходів до проєктування архітектури та бізнес-логіки програмних систем рівня підприємства.

### 2.1.2 ТЕОРЕТИЧНІ ВІДОМОСТІ

**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, валідація всіх елементів даних і обробка команд, що надходять від шару представлення, передача даних шару джерела даних.

Бізнес-логіка зазвичай схильна до частих змін, тому важливим є забезпечення можливості найпростішої модифікації і тестування цього шару коду. Звідси випливає нагальна необхідність знижувати ступінь залежності моделі предметної області та відповідних компонентів системи від інших шарів системи. Саме ця вимога є головним аспектом багатьох типових рішень, що визначають "розшарування" архітектури системи.

**Вибір типового рішення реалізації бізнес-логіки**

**Шаблон сценарій транзакцій**

Найпростіший підхід щодо проєктування та реалізації бізнес-логіки системи. Передбачає реалізацію у вигляді процедурного підходу, коли окрема процедура (або декілька процедур) виконує отримання даних від шару відображення, валідацію, необхідні обчислення, обробку та збереження даних, звернення за необхідності до сервісів інших систем, з подальшим форматуванням та поверненням даних шару відображення.

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

При використанні даного типового рішення логіка предметної області розподіляється по транзакціях, виконуваних в системі. Якщо, наприклад, користувачеві необхідно замовити номер у готелі, відповідна процедура повинна передбачати перевірку наявності відповідного номера, обчислення суми оплати та фіксацію замовлення в базі даних.

Перевагами застосування патерну є такі:

- Простота реалізації. Такий вид організації логіки є ефективний з точки зору сприйняття та продуктивності, зазвичай використовується для реалізації нескладної логіки невеликих додатків.
- Чіткі межі транзакцій, відсутність необхідності враховувати наявність і варіанти функціонування інших паралельних транзакцій.

Недоліки:

- При ускладненні бізнес-логіки її  складніше структурувати. Отже, складність реалізації складної бізнес-логіки.
- Схильність до дублювання коду. Транзакції зазвичай описують виконання однакових невеликих функцій, тому виникає велика ймовірність дублювання фрагментів коду. Винесення загальних алгоритмів в окремі функції часто повністю не вирішує питань дублювання коду.
- Складність застосування об’єктно-орієнтованих 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.

Переваги модуля таблиці полягають в можливості ефективного використання ресурсів реляційної бази даних.

В даному випадку класу бізнес-логіки відповідає певна таблична 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 (за умови наявності та використання певних інструментальних засобів).

Недоліки:

- складність застосування об'єктно-орієнтованого підходу до реалізації складної логіки.

**Шаблон «Модель предметної області (домен)»**

Типове рішення модель предметної області передбачає створення ієрархії об'єктів, кожен з яких представляє являє сутність. Об'єктна модель домену охоплює поведінку та дані. Кожне поняття моделі предметної області представлено окремим класом, можлива реалізація складних взаємозв'язків між об'єктами.

Ієрархія класів моделі предметної області зазвичай нагадує схему відповідної бази даних, однак у випадку використання реляційної бази даних, принципово від неї відрізняється, в першу чергу, використанням складних зв’язків між об'єктами (асоціацій, спадкування), об'єднанням даних та функцій, наявністю багатозначних атрибутів та ін.

Виділяють два різновиди моделей предметної області.

"Проста" багато в чому схожа на схему бази даних і містить, як правило, по одному об'єкту домена на кожну таблицю.
"Складна" модель може відрізнятися від структури бази даних і містити ієрархії

успадкування, стратегії та інші типові рішення, а також складні мережі взаємопов'язаних об'єктів. Складна модель надає можливість більш гнучко та просто представити заплутану бізнес-логіку, але важче піддається відображенню в реляційну модель бази даних.

Переваги використання:

- проста реалізації складної бізнес-логіки;
- просте тестування окремих компонентів;
- висока ступінь повторного використання.

Недоліки:

- складність забезпечення взаємодії з реляційною моделлю бази даних.

Модель предметної області не завжди доцільно використовувати, якщо бізнес-логіка застосування дуже проста. Однак варто враховувати можливість зростання складності бізнес-логіки застосування та змін трудомісткості розширення функціоналу застосування відповідно до експоненціального закону у випадку використання інших підходів. Ефективність патерну «модуль таблиці» в значній мірі залежить від рівня підтримки структури множини записів (що є зазвичай результатом виконання SQL-запиту) в певному інструментальному середовищі. Підхід не застосовується для реалізації складної бізнес-логіки.

Модель домен – кращий вибір для реалізації практично будь-якого застосування, крім випадку занадто простої логіки застосування або випадку використання інструментальних засобів, орієнтованих на модуль таблиць.

**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.

Передбачає рознесення бізнес-логіки за двома категоріями: логіка домену (domain logic) – працює з предметною областю (наприклад, стратегії нарахування відсотків за депозитом і т.д.) та логіка застосування (application logic) – визначає сферу відповідальності застосування (наприклад, повідомлення користувачів або сторонніх додатків про виконання певних процесів). Логіку застосування 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 традиційні 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.

**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".

**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 становлять CRUD-операції над об'єктами домену (create, read, update, delete - створити, зчитати, оновити, видалити). Створення, оновлення або видалення об'єкта домену застосування, 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. Невеликий додаток може включати одну абстракцію. Більші застосування зазвичай розбиваються на декілька підсистем, кожна з яких включає в себе вертикальний зріз всіх наявних архітектурних шарів. В цьому випадку створюють окремі абстракції для кожної з підсистем.

Отже, можливе створення абстракцій шару сервісів:

1) одна абстракція шару сервісів для всього застосування;
2) окрема абстракція для кожної з підсистем застосування;
3) абстракції сервісів для кожного домену моделі предметної області, якщо вони відрізняються від підсистем (наприклад, StudentService, CourseService);
4) окремі абстракції для різних типів поведінки (наприклад, AddService, UpdateService).

**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 найчастіше використовуються для віддаленого доступу.

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

Звернення до інтерфейсу віддаленого доступу пов'язано з витратами продуктивності. Тому варто дотримуватися мінімізації кількості віддалених викликів за рахунок повернення більшої інформації. Для реалізації таких абстракцій використовується патерн «об'єкт перенесення даних», що містить в собі більше даних, ніж вимагає клієнт, але можуть знадобитися надалі. Об'єкт перенесення даних (Data Transfer Object) застосовується для перенесення даних між процесами з метою зменшення кількості віддалених викликів (рисунок 2.1).

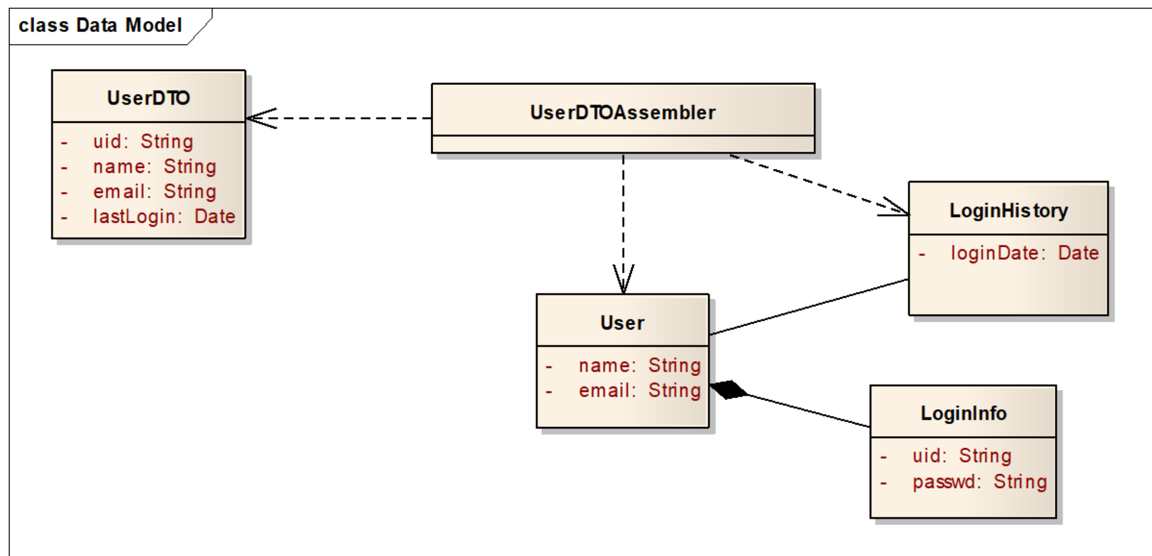


Рисунок 2.1 – Формування об’єкту переносу даних із об’єктів домену

Для переміщення даних між об'єктом переносу даних і об'єктами домену, як правило, застосовується «об'єкт-зборщик», що розташований на стороні сервера. Об'єкт перенесення містить в собі всі дані, які можуть знадобитися віддаленому клієнту 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 один об'єкт перенесення даних для запитів і відповідей або окремі об'єкти 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 дані 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.

**Шаблон проєктування «Стратегія»**

Стратегія (Strategy) - поведінковий шаблон проєктування, 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).

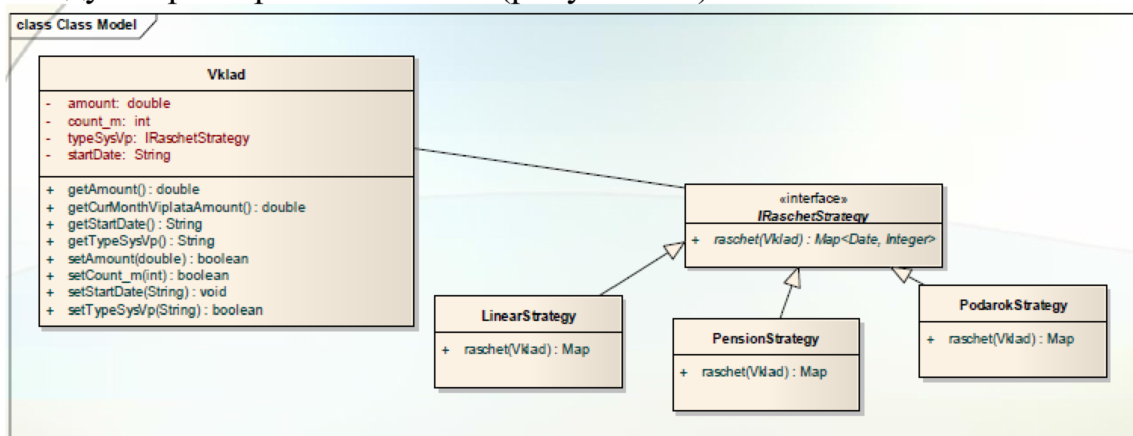


Рисунок 2.2 – Приклад реалізації патерну «Стратегія»

**Шаблок «Супертип шару»**

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

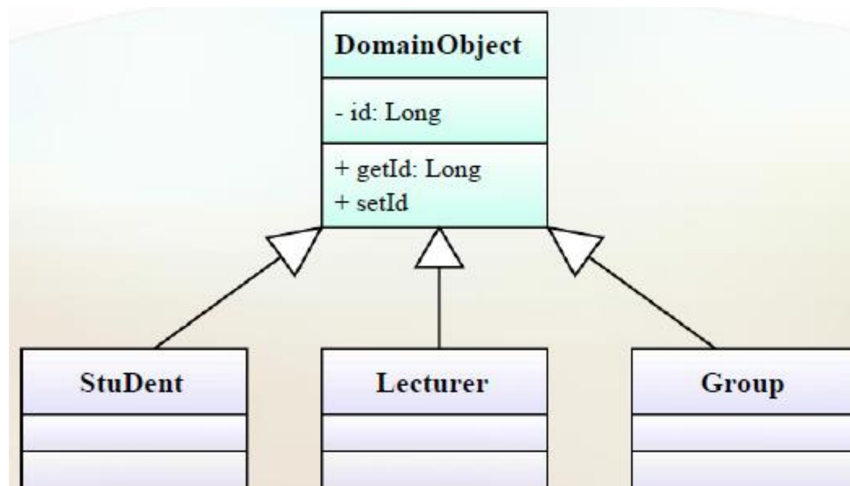


Рисунок 2.3 – Приклад використання шаблону супертип шару

### 2.1.3 Порядок виконання роботи

1. Провести аналіз предметної області: визначити функціональні та нефункціональні вимоги до системи; визначити ієрархію ролей користувачів додатку та їхні функціональні можливості. Результати проведення аналізу необхідно представити у вигляді UML-діаграм (Use Case діаграма, діаграма наслідування ролей користувачів, діаграма сутностей) з поясненнями.

2. Спроєктувати та представити архітектуру застосування.

2. Визначити типові рішення реалізації бізнес-логіки додатка.

3. Розробити та представити діаграму класів доменних об'єктів.

4. Розробити та представити діаграму інтерфейсів сервісів додатку. Інтерфейси шару сервісів обов'язково мають бути визначені з метою забезпечення можливості віддаленого доступу та гнучкості внесення змін.

5. Розробити алгоритми основних функціональних методів, що реалізують бізнес-логіку застосування. Представити діаграми послідовностей з поясненнями для відповідних алгоритмів.

### 2.1.4 Завдання для самостійної роботи

Вивчення структурних шаблонів проєктування та їх застосування для реалізації бізнес-логіки застосування.

### 2.1.5 Структура звіту

– назва, тема та мета лабораторної роботи;
– UML-діаграми варіантів використання додатку;
– архітектура додатку;
– опис використовуваних типових рішень;
– діаграма сутностей системи;
– діаграма класів доменних об'єктів системи;
– діаграма інтерфейсів сервісів додатків;
– діаграми послідовностей/дій для основних методів бізнес-логіки застосування;

– висновки.

UML-діаграми мають супроводжуватися описом.

## *2.2 Лабораторна робота №2*
## *Програмна реалізація та тестування компонентів бізнес-логіки застосування*

### 2.2.1 Мета роботи

Реалізація компонентів шару бізнес-логіки застосування з використанням технології EJB. Вивчення особливостей застосування SOLID-принципів.

### 2.2.2 Теоретичні відомості

**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).

**Реалізація шару сервісів з використанням технології Enterprise JavaBeans**

Компоненти Enterprise Beans є основою розробки розподілених корпоративних Jakarta EE додатків. EJB компонент представляє програмний модуль, який самостійно або взаємодіючи з іншими EJB-компонентами, реалізує і виконує бізнес-логіку додатка на стороні сервера застосувань. Enterprise JavaBeans являють собою спеціальним чином анотовані Java-класи. При використанні EJB вище 3 версії не потрібно успадковувати будь-які додаткові базові класи.

Для виконання EJB-компонентів потрібна наявність оточення EJB-контейнера, який реалізує сервіси системного рівня, такі як управління транзакціями та управління безпекою, що спрощує процес розробки розподілених додатків. При цьому реалізація низькорівневих сервісів залишається прозорою для розробника. Використання EJB-технологій при розробці програмних систем є доцільним в таких випадках:

1. Додаток має бути масштабованим, тобто повинна забезпечуватися можливість розгортання компонентів програми на декілька 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 розподіленого доступу до розподілюваних об'єктів.

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

Виділяють такі типи EJB-компонентів.

1. Сесійний 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'єкта (чергу), для якого MDB-компонент є слухачем. Повідомлення можуть бути відправлені будь-яким Jakarta EE компонентом (клієнт, інший EJB-компонент або веб-компонент), JMS-застосуванням або системою, що не використовує Jakarta EE. MDB-компонент не містить даних та інформації про сесію певного клієнта, всі сутності компонентів еквівалентні, що дозволяє контейнеру призначати обробку повідомлення будь-якому з компонентів. Контейнер створює пул об'єктів для одночасної обробки потоків повідомлень.

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

Компоненти, що не зберігають стан, є найбільш використовуваним варіантом реалізації сеансових компонентів розподіленого застосування. Як випливає з назви, особливістю таких компонентів є відсутність збереження 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 виконанні запиту і вивантажений 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 додатку до єдиного

розподілений 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) та 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 застосування 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 та гранулярності даних. Ізольованість даних передбачає, що віддалений клієнт і EJB-компонент працюють з різними копіями об'єкта-параметра. Подібний принцип ізоляції дозволяє забезпечити захищеність застосування. Гранулярність даних передбачає використання крупнозернистих параметрів (coarse-grained) з метою зменшення кількості віддалених викликів.

**Типи контейнерів компонентів застосування Jakarta EE**

Сервер застосування 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 додатку на сервері застосувань зображено на рисунку 2.4.

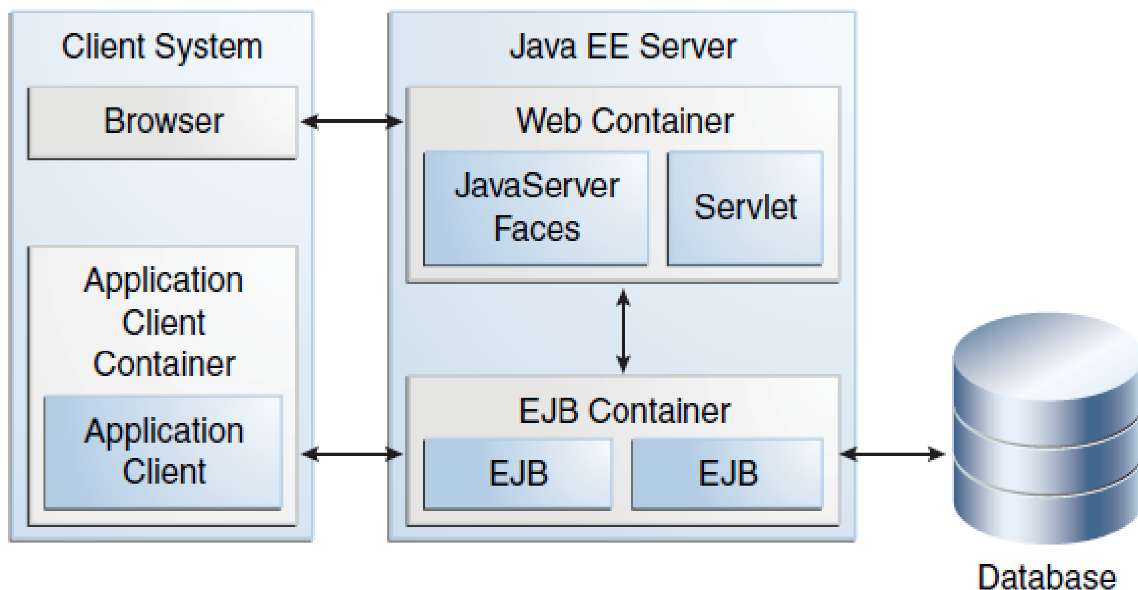


Рисунок 2.4 – Функції сервера застосувань

**Понятті інверсії контролю. Реалізація в Jakarta EE 9**

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

нак дані підходи припускають звернення до фабрики об'єктів для створення допоміжного об'єкта з коду класу компонента. IoC підхід передбачає декларативний опис залежностей за допомогою визначення конфігураційних файлів або анотацій.

Розглянемо приклад використання принципів IoC, реалізованих в рамках специфікації Jakarta EE 9. 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 не керованы 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).

**Особливості тестування класів EJB-компонентів**

EJB-компоненти 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> implements
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)
container.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));
     }
}
```

### 2.2.3 Порядок виконання роботи

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

### 2.2.4 Завдання для самостійної роботи

Вивчення особливостей реалізації та використання компонентів Message-Driven Bean.

## *2.3 Лабораторна робота №3*
## *Проєктування слою інтеграції*

### 2.3.1 Мета роботи

Дослідження особливостей застосування типових підходів до проєктування шару інтеграції корпоративного програмного застосування.

### 2.3.2 Теоретичні відомості

**Роль шару інтеграції в архітектурі застосування**

Роль шару інтеграції полягає в забезпеченні взаємодії з різними компонентами інфраструктури застосування (наприклад, інформаційні ресурси, віддалені сервіси та ін.) з метою виконання необхідних функцій.

**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 дані у вигляді множинних результатів.

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 для доступу до окремого запису джерела даних. Кожному рядку джерела даних відповідає окремий екземпляр шлюзу.

Оскільки екземпляр об'єкта, що реалізує шлюз запису даних, відповідає одному запису таблиці бази даних, то для визначення методів пошуку зазвичай 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 запису даних містить лише логіку доступу до даних і не реалізує бізнес-логіку обробки даних, можливе лише перетворення даних. У разі організації бізнес-логіки у вигляді «Сценарію транзакції» для організації шару інтеграції шаблон «Шлюз запису даних» є ідеальним рішенням.

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

Згідно з шаблоном «Активний запис» до функцій об'єкта, що виконує роль шлюзу запису даних, додається реалізація бізнес-логіка додатка. Можливим є варіант реалізації всієї бізнес-логіки додатка на рівні компонентів шару інтеграції. У цьому випадку шар інтеграції фактично відсутній.

Структура даних активного запису повинна повністю відповідати структурі таблиці бази даних: поле об'єкта відповідає певному стовпцю таблиці. Значення полів слід залишати такими ж, якими вони були отримані в результаті виконання SQL-команд, перетворення даних на цьому етапі робити не потрібно.

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

- створення екземпляра активного запису на основі рядка, отриманого в результаті виконання SQL-запиту;
- створення нового екземпляра активного запису для подальшого додавання в базу даних;
- статичні методи пошуку, які виконують стандартні SQL-запити та повертають активні записи;
- оновлення бази даних і додавання даних з активного запису;
- одержання та встановлення значень полів (get- і set-методи);
- реалізація деяких фрагментів бізнес-логіки.

Даний шаблон добре підходить для реалізації не надто складної 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 перетворення та передачу даних між об'єктами домену та базою даних, зберігаючи останніх незалежними один від одного та від самого перетворювача.

Об'єктно-орієнтована модель предметної області та реляційна модель даних СКБД використовують різні механізми структурування даних. У реляційних базах даних не відображаються багато характеристик об'єктів, зокрема, колекції, успадкування, двоспрямовані зв'язки та ін. При проєктуванні моделі доменних об'єктів застосування ООП механізмів дозволяє гнучко спроєктувати дані та поведінку об'єктів для складної бізнес-логіки. Однак, з іншого боку, застосування подібних механізмів призводить до розбіжностей об'єктної та реляційної моделей даних, що значно ускладнює завдання взаємодії та обміну даними між об'єктною моделлю та реляційною СКБД.

Типове рішення «Перетворювач даних» дозволяє забезпечити ізоляцію об'єктів предметної області та бази даних застосування – об'єкти «не добізнані» про навність бази даних, SQL, схему бази даних і т.д. 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 перетворювача, отже, уникати використання та викликів перетворювача з об'єктів домену.

Реалізація шару інтеграції у вигляді «перетворювача даних» достатньо складна та ресурсномістка через складність взаємного відображення об'єктної та реляційної моделей.

**Об’єктно-реляційне відображення**

Об'єктно-реляційне відображення (ORM - ) – це автоматичне та прозоре збереження та отримання об'єктів застосування в (з) таблиці реляційної бази даних з використанням метаданих, що визначають відображення між об'єктами та відносинами.

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

- API для виконання CRUD операцій;
- мова або API для виконання запитів;
- механізм визначення мета даних відображень;
- техніки для керування транзакціями, завантаження за вимогою та інших оптимізацій.

Застосування ORM технологій дозволяє зосередитися на реалізації складної бізнес-логіки з використанням принципів ООП, не виконуючи реалізації механізмів 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) – стандарт об'єктно-реляційного відображення згідно специфікації Jakarta EE [4]. Існують різні реалізації даного стандарту (Hibernate, EclipseLink, Apache OpenJPA, ін.).

**Поняття транзакції. Керування транзакціями**

Під транзакцією розуміють обмежену послідовність дій з чітко визначеними початковою та завершальною операціями, яка або виконується повністю, або не виконується зовсім.

Визначають наступні властивості транзакцій:

- atomicity (атомарність) – виконуються всі дії транзакції, або не виконується жодна з них;
- consistency (узгодженість) – системні ресурси повинні бути в цілісному та несуперечливому стані як до початку виконання транзакції, так і після її завершення;
- isolation (ізольованість) – проміжні результати транзакції повинні бути закритими 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. Вищий рівень ізоляції транзакцій забезпечує більш високу ймовірність відсутності конфліктних ситуацій при виконанні паралельних транзакцій (аномалії транзакцій), однак при цьому зазвичай можуть погіршуватися параметри продуктивності додатка. У таблиці 2.1 наведені аномалії транзакцій, що можуть виникати при певному рівню ізоляції транзакцій.

Таблиця 2.1 – 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 застосування в залежності від логіки застосування, тобто, за необхідності, встановлювати різні рівні ізоляції транзакцій для різних таблиць застосування. Наприклад, аномалію «брудне читання» можна залишати для ресурсів застосування, для яких передбачено, що читання буде виконуватися 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, декілька системних низькорівневих транзакцій. При цьому всі дії бізнес-транзакції розглядають як «єдина» транзакція, а бізнес-транзакція повинна відповідати всім властивостям системної транзакції.

### 2.3.3 Порядок виконання роботи

1. Вивчити типові рішення реалізації шару інтеграції застосування.
2. Виконати вибір сервера баз даних застосування та ORM-провайдера.

### 2.3.4 Завдання для самостійної роботи

Визначити можливі аномалії транзакцій для сценаріїв використання застосування та відповідні рівні ізоляції транзакцій.

## *2.4 Лабораторна робота №4*
## *Програмна реалізація та тестування компонентів шару інтеграції*

### 2.4.1 Мета роботи

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

### 2.4.2 Теоретичні відомості

**Поняття класу сутності (Entity)**

Згідно Jakarta 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 та відкритих методів доступу до неї:

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

Збереженню підлягають всі поля і властивості, не зазначені аннота-цією @Transient. Поля, що підлягають збереженню, можуть мати (необов'язково) додаткові анотації. 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, відзначене 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.

**Стратегії відображення наслідування в JPA**

Класи сутностей підтримують успадкування, поліморфні зв'язки 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 та 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-яка властивість, успадкована класом сутності від базового класу, не зберігається. Базовий клас не може використовуватися при формуванні запитів.

Правила відображення наслідування 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 рисунку 2.5, перша стратегія (SINGLE_TABLE) передбачає відображення всіх класів ієрархії в одну таблицю, яка містить додатковий атрибут – Discriminator column, що визначає приналежність конкретного запису таблиці того чи іншого підкласу (рисунок 2.5).

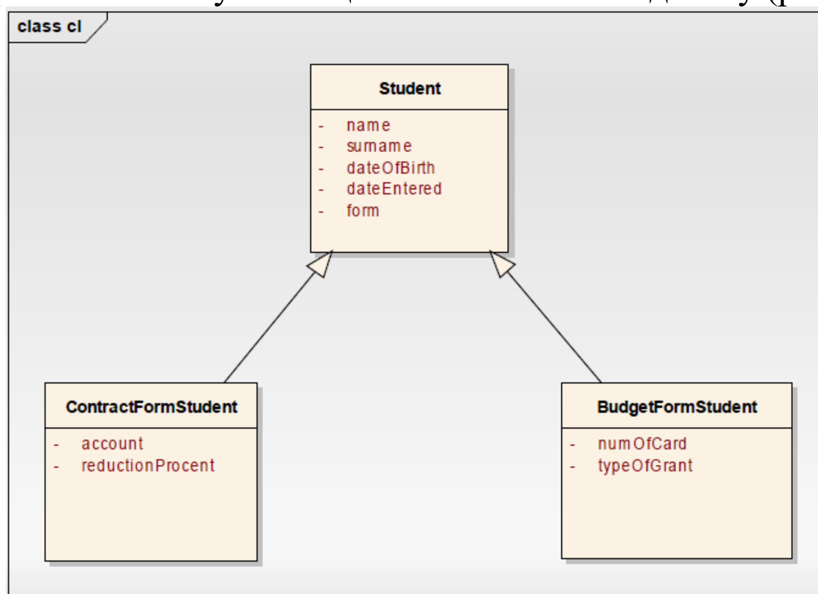


Рисунок 2.5 – Ієрархія класів

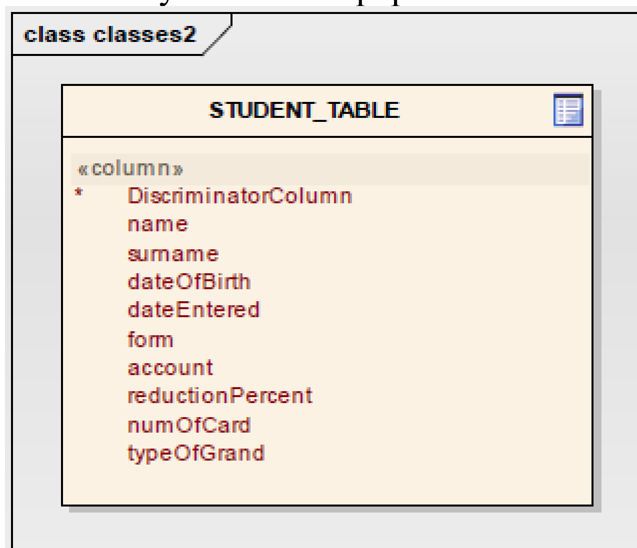


Рисунок 2.6 – Відображення наслідування згідно стратегії SINGLE_TABLE

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 та блокувань;
- пусте місце в невикористовуваних стовпцях.

JOINED-стратегія передбачає створення окремої таблиці бази даних для кожного підкласу (рисунок 2.7).

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 (у разі наявних складних ієрархій об’єктів);
- переміщення полів з підкласу в суперклас вимагає зміни структури БД;
- частий доступ до таблиць суперкласів.

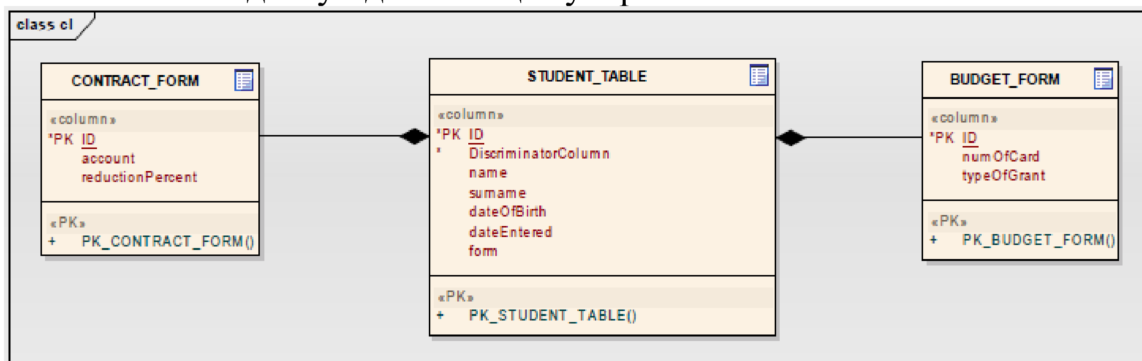


Рисунок 2.7 – Відображення наслідування згідно JOINED-стратегії

Стратегія TABLE_PER_CLASS передбачає створення окремих таблиць для конкретних класів (рисунок 2.8).

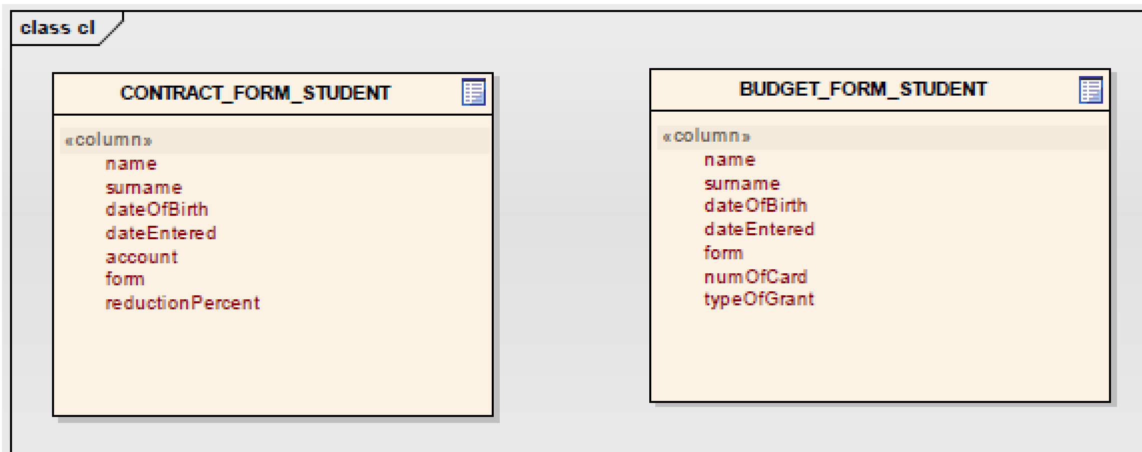

Рисунок 2.8 – Відображення наслідування згідно стратегії TABLE_PER_CLASS

Переваги стратегії TABLE_PER_CLASS:

– кожна таблиця замкнута та не містить непотрібних полей;
– при зчитуванні даних не потрібно виконувати операції з'єднання.

Недоліки:

– відсутня 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.

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

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

- public void persist (Object entity) – зберегти сутність, представлену об'єктом класу сутності, в БД;
- public void remove (Object entity) – видалити сутність;
- public <T> T find (Class <T> entityClass, Object primaryKey) - знайти сутність первинним ключем, T - тип сутності;
- public Query createQuery (String qlString) - створити запит БД з використанням мови 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 СКБД (SQL).

Існує поняття контексту зберігання (persistence context) – набір 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 та на рівні додатку. В останньому випадку екземпляр менеджера сутностей та

пов'язаний з ним контекст створюється та знищується безпосередньо з коду додатку. При цьому кожен екземпляр менеджера сутностей ініціалізує новий, незалежний контекст збереження. Управління контекстом транзакцій JTA в цьому випадку також має здійснюватися на рівні коду 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 коду застосування:

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

Використання модулів визначення контекстів збереження (Persistent Unit) дозволяє логічно згрупувати такі компоненти:

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

Модуль контексту збереження може являти собою окремий JAR-архів або набір класів в складі JAR-архіву компонента застосування. Модуль збереження має унікальне ім'я та визначається XML-дескриптором persistence.xml. Приклад визначення модуля збереження:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence  version="2.0"  xmlns="http://java.sun.com/xml/ns/per-
sistence"     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.PersistencePro-
vider</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>
```

**Виконання запитів в JPA з використанням JPQL та Criteria API**

Для виконання запитів до БД через сервіс збереження 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, що повертає одну сутність;
- public int executeUpdate () - виконує запит на зміну або видалення сутностей, повертає число сутностей, що було змінено або видалено;
- public Query setParameter (String name, Object value) - дозволяє задати значення іменованого параметра в параметризованому запиті;
- public Query setMaxResults (int maxResult) - задати максимальну кількість сутностей, що повертаються.

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

При безпосередньому виконанні запиту вираз на JPQL транслюється в SQL (або іншу мову запитів використовуваної СКБД). Є також можливість безпосередньо визначати запит на мові СКБД, що використовується, (native), проте, такий підхід не відповідає критерію переносимості.

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, якщо 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 є синтаксично некоректний (правильний варіант String jpql = "select p from Person p where p.age> 20";), код буде успішно скомпільовано. При цьому виключення буде згенероване тільки на етапі виконання.

Одною з ключових переваг Criteria API є те, що він дозволяє створювати типобезпечні запити та не дозволяє створювати синтаксично некоректні запити. Приклад визначення такого 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;
}
```

Всі атрибути класу domain.Person, що зберігаються, присутні в класі мета-моделі 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 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 й використання строкових запитів на мові JPQL. Компіляція цього фрагмента коду проходить успішно, проте компілятор не може гарантувати, що при його виконанні не виникне помилок.

**Керування областю дії транзакції в JPA**

Керування межами 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 початком виконання кожного методу EJB-компонента, і закриває відповідно після завершення методу, перед виходом з нього. Отже, кожен метод EJB-компонента за замовчуванням виконується в межах однієї транзакції. Вкладені та множинні транзакції всередині методу не допускаються.

При зверненні до методу EJB-компонента з іншого методу EJB-компонента (рисунок 2.9) в JPA використовується схема визначення меж транзакції в залежності від встановленого атрибуту транзакції, наведена в таблиці 2.2.

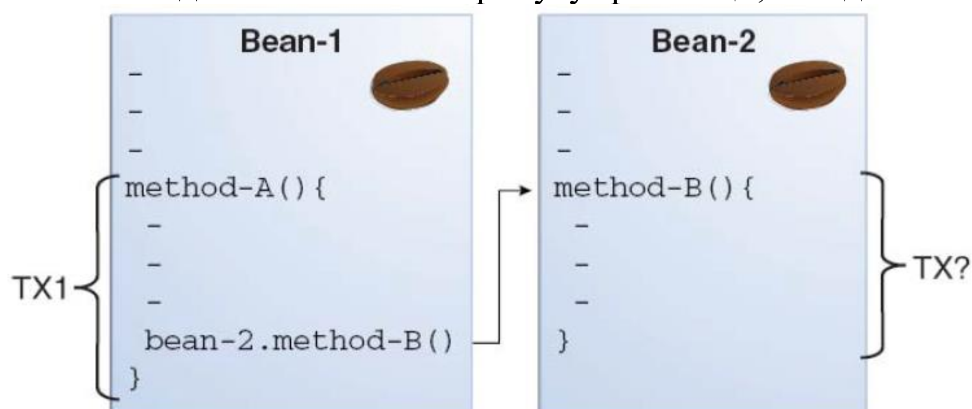


Рисунок 2.9 – Визначення області дії транзакції при виклику методу EJB-компонента з іншого методу

Таблиця 2.2 – Область дії транзакції в залежності від встановленого атрибуту транзакції

| **Transaction Attribute** | **Client's Transaction** | **Business Method's Transaction** |
|---|---|---|
| `Required` | None | T2 |
| `Required` | T1 | T1 |
| `RequiresNew` | None | T2 |
| `RequiresNew` | T1 | T2 |
| `Mandatory` | None | Error |
| `Mandatory` | T1 | T1 |
| `NotSupported` | None | None |
| `NotSupported` | T1 | None |
| `Supports` | None | None |
| `Supports` | T1 | T1 |
| `Never` | None | None |
| `Never` | 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, визначене над методом, перевизначає визначене 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 метод, що зробив виклик, виконувався поза транзакції;
- якщо викликав метод, що зробив виклик, виконувався в транзакції, метод secondMethod () буде виконуватися всередині відкритої поточної транзакції.

При виклику методів thirdMethod () і fourthMethod () буде застосована така послідовність:

- якщо метод виконувався в транзакції, то клієнтська транзакція буде припинена перед виконанням методу та відновлена після його завершення;
- якщо метод виконувався поза транзакцією, нова транзакція відкрита не буде.

**Визначення рівня блокування при виконанні транзакцій в JPA**

У JPA існує можливість визначення рівня блокування не тільки конкретної таблиці, але й окремого запису таблиці, що відповідного певному екземпляру об'єкта. За замовчуванням, провайдери JPA виконують модифікацію запису в таблиці бази даних після завершення бізнес-транзакції, крім ситуацій виконання примусової синхронізації – виклик методу flush () менеджера сутностей.

Існує поняття оптимістичного, що використовується за замовченням, та песимістичного блокування. У разі оптимістичного блокування записи таблиці не блокуються, при цьому перед фіксацією змін бізнес-транзакції виконується перевірка, чи не були змінені дані іншого транзакцією в процесі виконання поточної. Це реалізовано за допомогою додавання поля VERSION в таблиці, що автоматично інкрементується при кожному внесенні змін до запису таблиці.

У разі песимістичного блокування записи таблиці блокуються на весь час виконання транзакції, що запобігає можливості модифікації даних іншими транзакціями. Песимістичне блокування ефективне в разі наявної можливості частої модифікації даних паралельними транзакціями.

Можливо блокування запису як для запису (внесення змін, режим READ), так і для читання (режим WRITE).

Існує можливість визначення обмежень блокування даних за часом шляхом визначення відповідної анотації @Timeout (властивість javax.persistence.lock.timeout).

Рівень блокування транзакції в JPA може бути визначений за допомогою:

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

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

2) виклику одного з методів EntityManager.find () та передачі режиму блокування в якості параметра:

```
EntityManager em = ...;
String personPK = ...;
Person person = em.find(Person.class, personPK, LockModeType.PESSI-
MISTIC_WRITE);
```

3) виклику одного з методів EntityManager.refresh () та передачі режиму 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) визначення режиму блокування при визначенні статичного запиту:

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

### 2.4.3 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. Реалізувати інтеграційні тести компонентів шару сервісів та шару інтеграції.

### 2.4.4 Завдання для самостійної роботи

1. Визначення та валідація значень полей (або властивостей) сутностей, що зберігаються.
2. Використання колекцій для визначення полей і властивостей сутностей. Раннє (EAGER) та пізнє (LAZY) вивантаження об'єктів.
3. Визначення первинних ключів об'єктів, що зберігаються. Складені первинні ключі. Стратегії генерації первинних ключів.
4. Відображення відносин між сутностями. Виконання каскадних операцій.
5. Використання кешу при збереженні об'єктів з використанням JPA.

## *2.5 Лабораторна робота №5*
## *Проєктування шару відображення*

### 2.5.1 Мета роботи

Проаналізувати типові рішення реалізації шару відображення веб-застосувань. Виконати проєктування компонетів шару відображення.

### 2.5.2 Теоретичні відомості

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

Функції Web-сервера полягають в інтерпретації адреси URL запиту та передачі керування відповідному методу. 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 отримує дані з Web-сторінки, перевіряючи строковий об'єкт HTTP-запиту та 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 інший рядок відповіді, що утворюється сценарієм із залученням звичайних функцій поточного виведення.

Завдання формування коду HTML за допомогою команд поточного виведення ускладнює роботу дизайнерів. Це призвели до появи 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, то доцільним є одночасне застосовування двох моделей. Такий підхід був вперше реалізований досить давно для реалізації інтерфейсів користувача на основі типового рішення модель-представлення-контролер (Model View Controller).

Вхідний контролер приймає запит та отримує з нього вхідні дані запиту. Потім він передає бізнес-логіку належному об'єкту моделі, який звертається до джерела даних і виконує дії, передбачені в запиті, включаючи отримання даних для формування відповіді. По завершенні функцій він передає управління вхідному контролеру, який, аналізуючи отриманий результат, приймає рішення щодо вибору варіанта відображення відповіді. Управління та відповідні дані передаються шару відображення. Взаємодія вхідного контролера та відображення часто здійснюється не у вигляді прямих викликів, а з використанням деякого об'єкту HTTP-сеансу, який слугує для передачі даних в обох напрямках.

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

**Типові рішення реалізації контролеру**

Існують два типові підходи до організації вхідних контролерів – **контролер сторінок (Page Controller)** та **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 відображення 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 виконання певної дії на сайті. На вхідний контролер покладаються два основні обов'язки: обробка HTTP-запиту та прийняття рішення щодо подальшого 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 та передача їм даних; визначення потрібного відображення та 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 отриману адресу URL, визначає відповідний тип запиту та створює окремий об'єкт для подальшої обробки запиту. Таким чином, вдається централізувати діяльність з обробки всіх HTTP-запитів в рамках єдиного об'єкта та уникнути необхідності зміни конфігурації Web-сервера в разі модифікації структури дій сайту.

**Типові рішення реалізації відображення**

Існують три основні типові рішення щодо реалізації шару відображення: **відображення з перетворенням (Transform View), відображення за шаблоном (Template View) та двоетапне відображення (Two Step View)**. Вибір типового рішення зводиться до одного з двох варіантів – відображення з перетворенням або відображення за шаблоном в базовій одноетапній реалізації або ускладнення до рівня двоетапного відображення.

Відображення за шаблоном дозволяє оформлювати відображення відповідно до структури сторінки та вставляти в неї спеціальні маркери, що відзначають позиції фрагментів динамічного вмісту. Подібний підхід підтримується рядом платформ. Багато з яких побудовані на моделі сторінок сервера (наприклад, ASP, JSP і PHP) та дозволяють впроваджувати в текст сторінки коду з використанням повнофункціональної мови програмування. Таке рішення відрізняється

потужністю та гнучкістю, однак якщо код складний і заплутаний, завдання супроводу системи надзвичайно ускладнюється. Тому використання технологій сторінок сервера вимагає акуратності та послідовності в відокремленні логіки коду від структури сторінки, яке найчастіше досягається за допомогою допоміжного (helper) об'єкта.

Підхід реалізації «відображення з перетворенням» передбачає послідовну обробку елементів даних домену та перетворення їх в код HTML. Прикладом реалізації відображення з перетворенням є технологія XSLT. Ця технологія виявляється досить ефективною, якщо дані домену зберігаються в форматі XML або допускають швидке перетворення в XML. Вхідний контролер обирає потрібну таблицю XSLT стилів та застосовує її до XML-коду, що описує модель.

Одноетапний варіант реалізації відображення передбачає реалізацію переважно по одному компоненту відображення для кожного екрану інтерфейсу застосування. Код відображення отримує дані домену та перетворює їх в формат HTML.

Двоетапне відображення розділяє процес на дві стадії: на першій на основі даних домену формується логічний екран, який на другій стадії трансформується в код HTML. Для кожного екрану існує одне відображення першого етапу, але для додатка в цілому – єдине відображення другого етапу.

Перевага двоетапного уявлення полягає в тому, що рішення щодо варіанту 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 лише в тому випадку, коли логічне відображення залишається постійним, тобто різні екрани компонуються за одним принципом. Сайти, що спроєктовані за надмірно складними схемами, одноманітністю логічної структури зазвичай не відрізняються.

Типове рішення двоетапне відображення може ефективно використовуватися в ситуаціях, коли сервіси Web-застосування використовуються численними клієнтами (наприклад, відвідувачами сайту системи бронювання авіаквитків). Задовольняючи вимогам компонування одного логічного екрана, кожна з версій клієнтського застосування, що відповідає певному варіанту реалізації другого етапу відображення, 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.

### 2.5.3 Порядок виконання роботи

1. Проаналізувати типові рішення реалізації компонентів шару відображення.
2. Обрати технології реалізації шару відображення.
3. Розробити карту сайту, діаграми переходів.

4. Розробити прототип інтерфейсу користувача та виконати тестування прототипу інтерфейсу для основних сценаріїв. Переробити прототип інтерфейсу з урахуванням результатів тестування.

#### 2.5.4 Завдання для самостійної роботи

Дослідження архітектури та особливостей реалізації існуючих MVC-фреймворків.

## *2.6 Лабораторна робота №6*
## *Програмна реалізація та тестування компонентів шару відображення*

### 2.6.1 Мета роботи

Отримання практичних навичок використання технології Jakarta Faces. Реалізація шару відображення корпоративного застосування. Оцінка продуктивності та тестування веб-застосування.

### 2.6.2 Теоретичні відомості

**Особливості використання технології Jakarta Faces, конфігурація застосування**

Технологія Jakarta Faces є специфікацією та фреймворком, що надає:

- API для подання компонент для користувача інтерфейсу та 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 та зв'язку з серверними об'єктами.

JSF реалізує MVC модель, зв'язок між шарами моделі та відображенням здійснюється через об'єкти керування BackingBeans. Відповідно в об'єктах BackingBeans бажано не розміщувати бізнес-логіку та логіку збереження даних, а делегувати ці функції відповідно шарам бізнес-логіки та інтеграції застосування. До компонентів шарів застосування можна отримувати доступ через властивості об'єктів керування BackingBeans.

Facelets є технологією відображення для Jakarta Faces. Технологія JSP, яка використовувалася раніше для управління відображенням для Jakarta Faces, не підтримує нових функцій і вважається застарілою технологією відображення для Jakarta Faces.

Структуру стандартного Jakarta Faces-застосування наведено нижче:

- веб-сторінки з розташованими на них компонентами;
- бібліотека тегів компонентів;
- компоненти Backing Beans, що визначають властивості та функції компонентів сторінки;
- дескриптор розгортання (web.xml);
- один або більше файлів конфігурації (faces-config.xml або ін.), які можуть визначати правила навігації та конфігурації об'єктів BackingBeans та інших компонентів застосування;
- власні компоненти, валідатори, конвертори або слухачі;
- власні теги для відображення користувальницьких компонентів на сторінці.

Переваги використання Jakarta Faces:

- повне розділення логіки поведінки від логіки відображення;

– незалежна робота дизайнерів та розробників (зв'язок із серверними компонентами за допомогою тегів, без написання скриптів);
– відсутність обмежень на використання певних скриптових технологій або мов розмітки; Jakarta Faces API розміщено поверх Servlet API (рисунок 2.10);
– використання різних технологій відображення, створення власних компонент, генерація виведення для різних клієнтів;
– технологія Facelets (як частина Jakarta Faces) надає можливість повторного використання коду, розширення компонентів за допомогою функцій шаблонування та composite-компонентів;
– автоматична реєстрація backing bean-ів як ресурсів Jakarta Faces-застосування при використанні анотацій Jakarta Faces;
– швидке конфігурація сторінки навігації за допомогою визначення "неявних" правил навігації;
– багата архітектура для управління складовими компонентами, управління даними, перевірки введених даних і обробки подій.

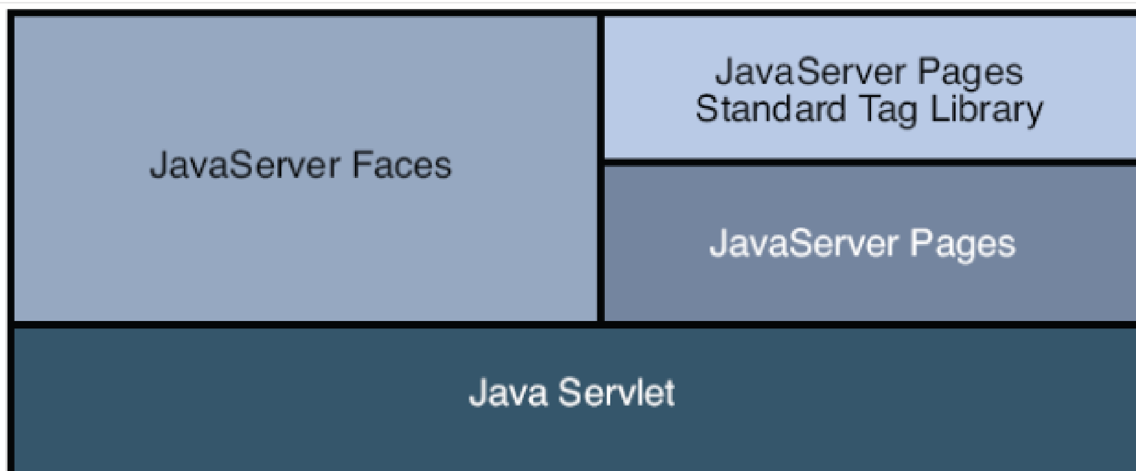


Рисунок 2.10 – Особливості реалізації технології JSF

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

1. Параметр контексту, який визначає стадію розробки проєкту:

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

Наприклад, при встановленому значенні параметра контексту Development інформація для відлагодження буде автоматично генеруватися та 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 Jakarta Faces-сервлет для обробки запитів. Сервлет повинен викликатися для кожного запиту зі сторінок, що використовують Jakarta Faces. Згідно відображенню в web.xml, Jakarta Faces контейнер буде викликати FacesServlet для обробки всіх запитів, відносний URI яких починається з / faces /. Завдяки цьому буде коректно ініціалізуватися контекст Faces і кореневий елемент відображення перед показом сторінок Jakarta Faces. Кореневий елемент містить дерево компонентів, а контекст Faces служить для взаємодії додатку з Jakarta Faces.

3. Стартова сторінка програми:

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

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

Наприклад, для глобального управління повідомленнями, необ-обхідно визначити ресурс (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 будь-якого поля введення достатньо змінити 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.

Динамічні елементи Jakarta Faces-сторінок визначаються за допомогою властивостей і методів компонентів керування Backing Beans. Зв'язування компонентів Jakarta Faces-сторінок з відповідними властивостями / методами об'єктів Backing Beans виконується з використанням Jakarta Faces Expression Language (EL).

EL пов'язує поля введення / виведення з відповідними значеннями 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 область визначення компонентів керування Jakarta Faces. 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 запитами сесії;
- @ApplicationScoped – зберігає стан між HTTP запитами різних сесій різних користувачів.

Приклад звернення до властивості визначеного backing bean з компонента введення Jakarta Faces-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 за допомогою EL-виразів. Таким чином, при натисканні на кнопку буде викликаний відповідний метод (якщо не виникло помилок конвертації або валідації).

За замовчуванням Jakarta Faces виконує затвердження даних форми перед викликом методу. Якщо встановлено значення атрибута immediate в true, метод буде викликано відразу.

**Шаблони та составні компоненти**

Шаблонування є функцією технології Facelets, що дозволяє створити 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 теги для шаблонування:

<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 виконується 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 Section"/>
                  </ui:define>
                  <ui:define name="content">
                        <h:graphicImage value="#{resource['im-
                  ages:wave.med.gif']}"/>
                        <h:outputText value="You are in the Main Content Sec-
                  tion"/>
                  </ui:define>
            </ui:composition>
      </h:body>
</html>
```

Jakarta Faces надає можливість створення складеного (composite) компонента. Під складеним компонентом розуміється повторно використовуваний компонент, який виконує певну функціональність (наприклад, inputText). Складений компонент являє собою особливий тип шаблону, який виступає в ролі повторно використовуваного компонента та 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 складений компонент.

Теги, використовувані для визначення складених компонентів: <composite: interface />, <composite: implementation />, <composite: attribute />, <composite: insertChildren />, <composite: valueHolder />, <composite: editableValueHolder / >, <composite: 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 Jakarta Faces 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/>.

**Визначення правил навігації**

Виділяють поняття явної та неявної навігації. Останнє передбачає визначення правил навігації не з використанням файлу конфігурації Jakarta Faces-додатку.

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-компонента, 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", fac-
esMessage); //$NON-NLS-1$
            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.

Явне визначення правил навігації передбачає визначення строкового параметра переходу з однієї сторінки на іншу в файлі конфігурації Jakarta Faces (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 результат виконання методу backing bean-компонента повинні бути визначені відповідно до встановлених в файлі конфігурації строкових параметрів переходу.

**Механізми керування безпекою**

Компоненти JakartaEE додатків можуть містити як захищені, так і незахищені ресурси. Доступ до захищених ресурсів додатку 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) та 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 застосування:

- аутентифікація – процес взаємної перевірки автентичності клієнта та 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 та 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 додатка з метою оцінки ефективності використовуваних політики та механізмів забезпечення безпеки.

JakartaEE 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 JakartaEE контейнера компонентів. Можливе декларативне та програмне керування безпекою додатку 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 – визначення політики безпеки з використанням методів інтерфейсів EJBContext і HttpServletRequest.

Виділяють такі рівні управління інформаційною безпекою застосування:

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

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

1. Визначення механізму аутентифікації.

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

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

Механізми аутентифікації basic та form-based вважаються менш надійними. У випадку 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 додатку.

Digest аутентифікація, як і basic, передбачає використання логін / пароль в якості даних аутентифікації користувача. Однак, в разі digest-аутентифікації, замість відправки пароля користувача передбачається відправка одностороннього криптографічного хеша пароля та 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 застосування.

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 додатку. Ролі користувачів додатка можуть бути визначені за допомогою елемента <security-role /> дескриптора розгортання застосування.

Захищені ресурси програми, а також права доступу різних ролей користувачів до захищених ресурсів додаткк 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 додатку).

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

JakartaEE сервер може бути налаштовано на використання декількох реалмів.

Glassfish сервер за замовчуванням автоматично налаштований на використання реалмів 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 ролей користувачів додатку конкретним 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 2.11).

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 налаштування 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>
```

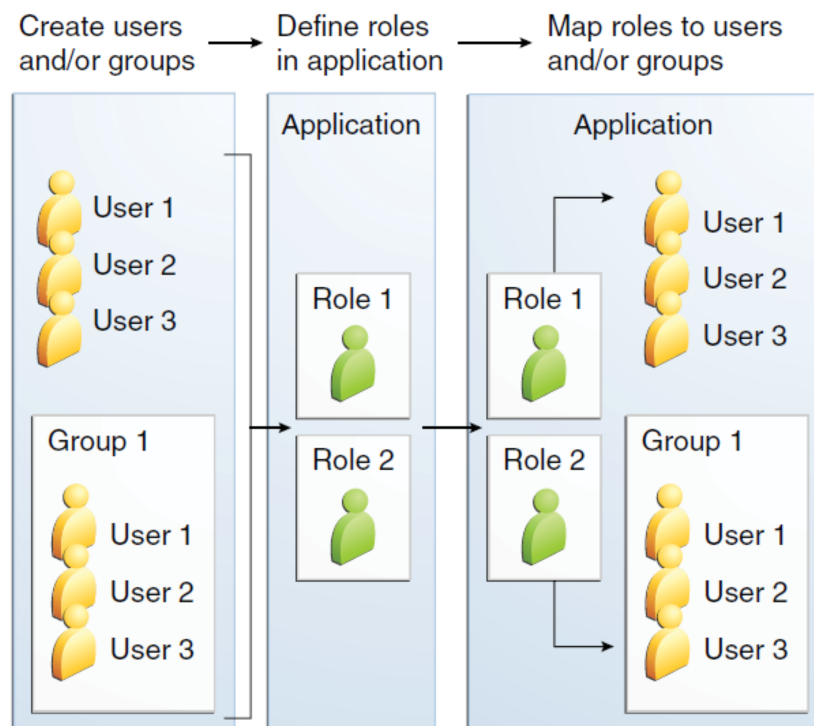


Рисунок 2.11 – Відповідність ролі користувача додатку користувачам і групам, визначеним в реалмі сервера застосування

Декларативне управління безпекою дозволяє встановити права доступу різних визначених ролей користувачів додатку до компонентів застосування (EJB і web-компонентів). Для цього використовуються анотації @DeclareRoles – визначити доступ для всіх існуючих ролец 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, визначені 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 та конфігурації підтримки SSL сервером застосування необхідно:

1) визначити connector-елемент в дескрипторі розгортання;

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

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

Підтег <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>, що визначають рівень захищеності з'єднання:

- CONFIDENTIAL –інформація, що пересилається, не може бути переглянута «третьої» стороною;
- INTEGRAL – інформація, що пересилається, 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 2.12).

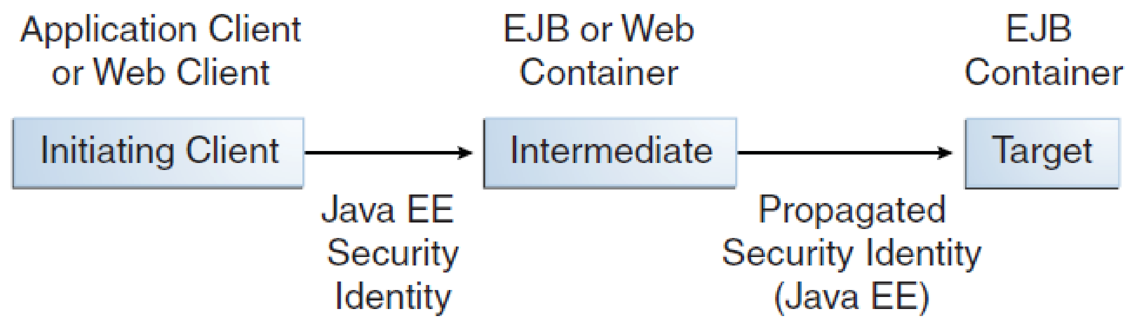

Рисунок 2.12 – Поширення SecurityIdentity

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 класом EJB-компонента або певним методом:

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

### 2.6.3 Порядок виконання роботи

1. Конфігурація веб-модуля додатка.
2. Розробка веб-сторінок додатку та відповідних компонентів BackingBean.
3. Визначення правил навігації застосування.
4. Тестування основних сценаріїв використання веб-додатку.
4. Планування та проведення навантажувального тестування застосування. Оцінка результатів.

### 2.6.4 Завдання до самостійної роботи

1. Використання стандартних і власних конверторів, слухачів і валідаторів в Jakarta Faces.
2. Розробка власних компонентів Jakarta Faces.
3. Застосування технології AJAX в Jakarta Faces.

# 3 Реалізація корпоративної програмної системи з використанням фреймворку Spring

Spring — це найпопулярніший фреймворк для розробки додатків на мові програмування Java. Мільйони розробників у всьому світі використовують Spring Framework для створення високопродуктивного коду, який легко тестувати та використовувати повторно.

Основні функції Spring Framework можна використовувати для розробки будь-якої програми на мові Java. Нижче наведено список кількох великих переваг використання Spring:

- Spring організована за модульним принципом. Незважаючи на те, що кількість пакунків і класів є значною, вам доведеться турбуватися лише про ті, які вам потрібні, і ігнорувати решту.
- Тестування програми, написаної за допомогою Spring, просте, оскільки код, що залежить від середовища, переміщується в цю структуру.
- Spring надає зручний API для перекладу специфічних для технології винятків (наприклад, JDBC, Hibernate).
- Spring забезпечує послідовний інтерфейс керування транзакціями, який можна масштабувати до локальної транзакції (наприклад, за допомогою єдиної бази даних) і масштабувати до глобальних транзакцій.

## *3.1 Лабораторна робота № 1*
## *Знайомство з фреймворком Spring*

### 3.1.1 Мета роботи

Знайомство з фреймворком Spring та отримання практичних навичок.

### 3.1.2 Теоретичні відомості

**Структура Spring**

Spring — це не один якийсь конкретний фреймворк. Це скоріше загальна назви для цілого ряду невеликих фреймворків, кожен із яких виконує якусь свою роботу (рисунок 3.1).

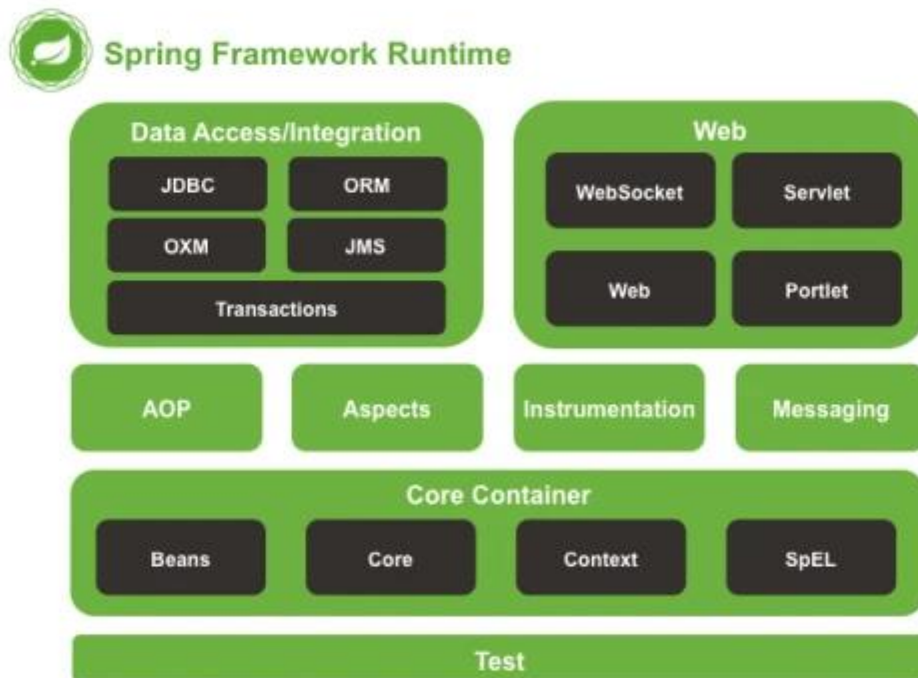


Рисунок 3.1 – Структура Spring

Як видно, Spring має модульну структуру. Це дозволяє підключати тільки ті модулі, які нам потрібні для нашої програми і не підключати ті, якими ми не будемо користуватися.

На зображенні видно, що Spring фреймворк складається з кількох модулів:

- Data access – робота з реляційними системами керування базами даних на платформі Java, за допомогою Java Database Connectivity (JDBC), та інструментів об’єктно-реляційного відображення, та з базами даних NoSQL.
- WEB – для роботи в мережі та для створення веб-додатків.
- Core – це базовий модуль Spring і надає spring контейнери (BeanFactory і ApplicationContext).

- Messaging: конфігураційна реєстрація об’єктів прослуховування повідомлень для прозорого споживання повідомлень із черг повідомлень через службу повідомлень Java (JMS), покращення надсилання повідомлень через стандартні API JMS.
- Test: допоміжні класи для написання модульних тестів та інтеграційних тестів.
- Крім того, є ще так-звана ціла Spring-інфраструктура: безліч інших проєктів, які не входять у сам фреймворк офіційно, але при цьому безкоштовно інтегруються у ваш проєкт на Spring (наприклад, spring security для роботи з авторизацією користувачів на сайті).

**Spring Bean та Inversion of Control (IoC)**

Bean є ключовою концепцією Spring Framework. Отже, розуміння цього поняття має вирішальне значення, щоб зрозуміти структуру та використовувати її ефективним способом.

Spring об’єкти, які утворюють основу вашої програми та якими керує контейнер Spring Inversion of Control (IoC), називаються компонентами. Компонент — це об’єкт, який створюється, збирається та іншим чином керується контейнером Spring IoC.

Ретельна обробка контейнера IoC Spring Framework супроводжується повним охопленням технологій Spring’s Aspect-Oriented Programming (AOP). Spring Framework має власну структуру AOP, концептуально просту для розуміння та успішно задовольняє 80% вимог до AOP у корпоративному програмуванні Java.

### 3.1.3 Порядок виконання роботи

Розглянемо приклад використання Bean та IoC.

```
@Component
public class Company {
    private Address address;
    public Company(Address address) {
        this.address = address;
    }
}

@Configuration
@ComponentScan(basePackageClasses = Company.class)
public class Config {
    @Bean
    public Address getAddress() {
        return new Address("High Street", 1000);
    }
}
```

Клас конфігурації створює Bean типу Address. Він також містить анотацію @ComponentScan, яка вказує контейнеру шукати компоненти в пакеті, що містить клас Company.

Коли контейнер Spring IoC створює об’єкти цих типів, усі об’єкти називаються компонентами Spring, оскільки ними керує контейнер IoC.

Оскільки ми визначили компоненти в класі конфігурації, нам знадобиться екземпляр класу AnnotationConfigApplicationContext для створення контейнера:

```
ApplicationContext context = new AnnotationConfigApplicationContext(Config.class);
```

Швидкий тест перевіряє існування та значення властивостей наших компонентів:

```
Company company = context.getBean("company", Company.class); assertEquals("High Street",
company.getAddress().getStreet()); assertEquals(1000, company.getAddress().getNumber());
```

**Spring Boot**

Spring Boot — це доповнення до платформи Spring, яке легко розпочати, і дуже корисне для створення автономних програм виробничого рівня з мінімальними зусиллями.

Нижче приведений код, точки входу в додаток.

```
@SpringBootApplication
@RestController
public class ExampleApplication {
      public static void main(String[] args) {
            SpringApplication.run(ExampleApplication.class, args);
            }
      }
```

Зверніть увагу, як ми використовуємо @SpringBootApplication як основний клас конфігурації програми. За кадром це еквівалентно @Configuration, @EnableAutoConfiguration і @ComponentScan разом.

**Тестування в Spring**

Ми можемо використати @SpringBootTest, щоб завантажити контекст програми та перевірити відсутність помилок під час запуску програми:

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class SpringContextTest {
      @Test public void contextLoads() {
      }
}
```

Далі додамо JUnit, який перевіряє виклики API, які ми написали, використовуючи REST Assured.

REST-assured розроблено для спрощення тестування та перевірки REST API, і на нього сильно впливають методи тестування, що використовуються в динамічних мовах.

І тепер ми можемо додати тест:

```
public class SpringBootBootstrapLiveTest {
    private static final String API_ROOT
    = "http://localhost:8081/api/books";
    private Book createRandomBook() {
        Book book = new Book(); book.setTitle(randomAlphabetic(10));
        book.setAuthor(randomAlphabetic(15));
        return book;
        }
    private String createBookAsUri(Book book) {
        Response response = RestAssured.given()
        .contentType(MediaType.APPLICATION_JSON_VALUE) .body(book)
        .post(API_ROOT);
        return API_ROOT + "/" + response.jsonPath().get("id");
        }
}
```

По-перше, ми можемо спробувати знайти книги за допомогою різних методів:

```
@Test
public void whenGetAllBooks_thenOK() {
    Response response = RestAssured.get(API_ROOT); assertEquals(HttpStatus.OK.value(),
    response.getStatusCode());
}
@Test
public void whenGetBooksByTitle_thenOK() {
    Book book = createRandomBook();
    createBookAsUri(book);
    Response response = RestAssured.get( API_ROOT + "/title/" + book.getTitle());
    assertEquals(HttpStatus.OK.value(), response.getStatusCode()); assertTrue(response.as(List.class)
    .size() > 0);
}
@Test
public void whenGetCreatedBookById_thenOK() {
    Book book = createRandomBook(); String location = createBookAsUri(book);
    Response response = RestAssured.get(location); assertEquals(HttpStatus.OK.value(),
    response.getStatusCode()); assertEquals(book.getTitle(), response.jsonPath() .get("title"));
    }
@Test
public void whenGetNotExistBookById_thenNotFound() {
```

```
        Response response = RestAssured.get(API_ROOT + "/" + randomNumeric(4));
        assertEquals(HttpStatus.NOT_FOUND.value(), response.getStatusCode());
}
```

Далі ми спробуємо створити нову книгу:

```
@Test
public void whenCreateNewBook_thenCreated() {
    Book book = createRandomBook();
    Response response = RestAssured.given() .contentType(MediaType.APPLICATION_JSON_VALUE)
    .body(book) .post(API_ROOT); assertEquals(HttpStatus.CREATED.value(), response.getStatusCode());
}
@Test
public void whenInvalidBook_thenError() {
    Book book = createRandomBook();
    book.setAuthor(null);
    Response response = RestAssured.given() .contentType(MediaType.APPLICATION_JSON_VALUE)
    .body(book)
    .post(API_ROOT);
    assertEquals(HttpStatus.BAD_REQUEST.value(), response.getStatusCode());
}
```

Потім ми оновимо наявну книгу:

```
@Test
public void whenUpdateCreatedBook_thenUpdated() {
    Book book = createRandomBook();
    String location = createBookAsUri(book);
    book.setId(Long.parseLong(location.split("api/books/")[1])); book.setAuthor("newAuthor");
    Response response = RestAssured.given() .contentType(MediaType.APPLICATION_JSON_VALUE)
    .body(book)
    .put(location);
    assertEquals(HttpStatus.OK.value(),
    response.getStatusCode());
    response = RestAssured.get(location); assertEquals(HttpStatus.OK.value(),
    response.getStatusCode());
    assertEquals("newAuthor", response.jsonPath() .get("author"));
}
```

І ми можемо видалити книгу:

```
@Test
public void whenDeleteCreatedBook_thenOk() {
```

```
    Book book = createRandomBook();
    String location = createBookAsUri(book);
    Response response = RestAssured.delete(location); assertEquals(HttpStatus.OK.value(),
    response.getStatusCode());
    response = RestAssured.get(location); assertEquals(HttpStatus.NOT_FOUND.value(),
    response.getStatusCode());
}
```

## *3.2 Лабораторна робота № 2*
## *Spring Data*

### 3.2.1 МЕТА РОБОТИ

Вивчення Spring Data (Spring Data JDBC та Spring Data JPA); отримання практичних навичок використання при реалізації компонентів шару інтеграції програми.

### 3.2.2 ТЕОРЕТИЧНІ ВІДОМОСТІ

**Spring Data**

Місія Spring Data полягає в тому, щоб надати знайому та узгоджену модель програмування на основі Spring для доступу до даних, зберігаючи при цьому особливі риси основного сховища даних.

Це полегшує використання технологій доступу до даних, реляційних і нереляційних баз даних, фреймворків для скорочення карт і хмарних служб даних. Це комплексний проєкт, який містить багато підпроєктів, які є специфічними для даної бази даних. Розглянемо функції Spring Data:

- Потужний репозиторій і спеціальні абстракції відображення об'єктів;
- Виведення динамічного запиту з імен методів репозиторію;
- Базові класи домену реалізації, що забезпечують базові властивості;
- Підтримка прозорого аудиту (створено, останні зміни);
- Можливість інтеграції власного коду репозиторію;
- Легка інтеграція Spring через JavaConfig і власні простори імен XML;
- Розширена інтеграція з контролерами Spring MVC.

**Spring Data JDBC**

Spring Data JDBC, частина сімейства Spring Data, дозволяє легко впроваджувати репозиторії на основі JDBC. Цей модуль стосується розширеної підтримки рівнів доступу до даних на основі JDBC. Це полегшує створення додатків на основі Spring, які використовують технології доступу до даних.

Spring Data JDBC прагне бути концептуально простим. Щоб досягти цього, він не пропонує кешування, відкладене завантаження, запис позаду або багато інших функцій JPA.

Spring Data JDBC доступний для програм Spring Boot із запуском залежностей JDBC. Однак цей засіб запуску залежностей не містить драйвер бази даних. Це рішення повинен прийняти програміст.

Як ми вже згадували раніше, Spring Data JDBC не пропонує створення схем. У такому випадку ми можемо створити спеціальний файл schema.sql, який міститиме команди SQL DDL для створення об'єктів схеми. Автоматично Spring Boot вибере цей файл і використає його для створення об'єктів бази даних.

**Додавання Entities**

Як і в інших проєктах Spring Data, ми використовуємо анотації для зіставлення POJO з таблицями бази даних. У Spring Data JDBC сутність повинна мати @Id. Spring Data JDBC використовує анотацію @Id для ідентифікації сутностей.

Spring Data JDBC за замовчуванням використовує стратегію іменування, яка зіставляє сутності Java з таблицями реляційної бази даних і атрибути з іменами стовпців. Назви сутностей і атрибутів зіставляються з назвами регістрів таблиць і стовпців відповідно. Наприклад, сутність Java під назвою AddressBook відображається в таблиці бази даних під назвою address_book.

Крім того, ми можемо явно зіставляти сутності та атрибути з таблицями та стовпцями, використовуючи анотації @Table і @Column.

### 3.2.3 Порядок виконання роботи

Нижче ми визначили сутність, яку будемо використовувати в цьому прикладі:

```
public class Person {
    @Id
    private long id;
    private String firstName;
    private String lastName;
    // constructors, getters, setters
}
```

Нам не потрібно використовувати анотацію @Table або @Column у класі Person. Стратегія іменування за замовчуванням Spring Data JDBC неявно виконує всі зіставлення між сутністю та таблицею.

**Оголошення інтерфейсів JDBC**

Ми можемо створити інтерфейс Spring Data JDBC, розширивши його Repository, CrudRepository або PagingAndSortingRepository.

Впроваджуючи CrudRepository, ми отримуємо впровадження найбільш часто використовуваних методів, таких як збереження, видалення та findById, серед інших.

Давайте створимо інтерфейс JDBC, який ми будемо використовувати в нашому прикладі:

```
@Repository
public interface PersonRepository extends CrudRepository<Person, Long> { }
```

Якщо нам потрібні функції розбиття на сторінки та сортування, найкращим вибором було б розширити інтерфейс PagingAndSortingRepository.

**Налаштування інтерфейса JDBC**

Незважаючи на вбудовані методи CrudRepository, нам потрібно створювати наші методи для конкретних випадків.

Тепер давайте налаштуємо наш PersonRepository за допомогою запиту, що не змінює, і запиту, що змінює:

```
@Repository
```

```
public interface PersonRepository extends CrudRepository<Person, Long> {
	List<Person> findByFirstName(String firstName);
	@Modifying
	@Query("UPDATE person SET first_name = :name WHERE id = :id")
	boolean updateByFirstName(@Param("id") Long id, @Param("name") String name);
}
```

Починаючи з версії 2.0, Spring Data JDBC підтримує методи запитів. Тобто, якщо ми назвемо наш метод запиту, включаючи ключові слова, наприклад, findByFirstName, Spring Data JDBC автоматично згенерує об'єкт запиту.

Однак для модифікаційного запиту ми використовуємо анотацію @Modifying, щоб анотувати метод запиту, який змінює сутність. Крім того, ми прикрашаємо його анотацією @Query.

Одна річ, яку ми повинні мати на увазі, полягає в тому, що Spring Data JDBC не підтримує посилання на параметри з номерами індексів. Ми можемо посилатися на параметри лише за назвою.

**Заповнення бази даних**

Нарешті, нам потрібно заповнити базу даних, ці дані будуть служити для тестування репозиторію Spring Data JDBC, який ми створили вище. Отже, ми збираємося створити базу даних, яка вставлятиме фіктивні дані. Давайте додамо реалізацію бази даних для цього прикладу:

```
@Component
public class DatabaseSeeder {
	@Autowired
	private JdbcTemplate jdbcTemplate;
	public void insertData() {
		jdbcTemplate.execute("INSERT INTO Person(first_name,last_name) VALUES('Victor',
		'Hugo')"); jdbcTemplate.execute("INSERT INTO Person(first_name,last_name)
		VALUES('Dante', 'Alighieri')");
		jdbcTemplate.execute("INSERT INTO Person(first_name,last_name) VALUES('Stefan',
		'Zweig')"); jdbcTemplate.execute("INSERT INTO Person(first_name,last_name)
		VALUES('Oscar', 'Wilde')"); }
}
```

Як видно вище, ми використовуємо Spring JDBC для виконання операторів INSERT. Зокрема, Spring JDBC обробляє з'єднання з базою даних і дозволяє нам виконувати команди SQL за допомогою JdbcTemplates. Це рішення є дуже гнучким, оскільки ми маємо повний контроль над виконуваними запитами.

**Spring Data JPA**

Spring Data JPA, частина сімейства Spring Data, дозволяє легко впроваджувати репозиторії на основі JPA. Цей модуль стосується розширеної підтримки

рівнів доступу до даних на основі JPA. Це полегшує створення додатків на основі Spring, які використовують технології доступу до даних.

У разі використання Spring Data JPA для реалізації рівня збереження, репозиторій зазвичай повертає один або більше екземплярів кореневого класу. Однак найчастіше нам не потрібні всі властивості повернутих об’єктів.

У таких випадках нам може знадобитися отримати дані як об’єкти налаштованих типів. Ці типи відображають часткові перегляди кореневого класу, що містить лише ті властивості, які нам важливі. Розглянемо функції Spring Data JPA:

- Складна підтримка створення репозиторіїв на основі Spring і JPA.
- Підтримка розбиття на сторінки, динамічне виконання запитів, можливість інтегрувати власний код доступу до даних.
- Перевірка анотованих запитів @Query під час початкового завантаження.
- Підтримка відображення сутностей на основі XML.
- Конфігурація сховища на основі JavaConfig шляхом представлення @EnableJpaRepositories.

**Класи сутностей Spring Data JPA**

Давайте визначимо два класи сутностей:

```
@Entity
public class Address {
	@Id
	private Long id;
	@OneToOne
	private Person person;
	private String state;
	private String city;
	private String street;
	private String zipCode;
	// getters and setters
}

@Entity
public class Person {
	@Id
	private Long id;
	private String firstName;
	private String lastName;
	@OneToOne(mappedBy = "person")
	private Address address;
```

```
    // getters and setters
}
```

Зв'язок між Person та Address є двонаправленим один до одного.

**Скрипти SQL**

Ми використаємо сценарій projection-insert-data.sql для заповнення обох опорних таблиць:

```
INSERT INTO person(id,first_name,last_name) VALUES (1,'John','Doe');
INSERT INTO address(id,person_id,state,city,street,zip_code)  VALUES (1,1,'CA', 'Los Angeles', 'Standford Ave', '90001');
```

Щоб очистити базу даних після кожного тестового запуску, ми можемо використовувати інший сценарій, projection-clean-up-data.sql:

```
DELETE FROM address;
DELETE FROM person;
```

**Тестовий клас**

Щоб підтвердити, що прогнози створюють правильні дані, нам потрібен тестовий клас:

```
@DataJpaTest
@RunWith(SpringRunner.class)
@Sql(scripts = "/projection-insert-data.sql")
@Sql(scripts = "/projection-clean-up-data.sql", executionPhase = AFTER_TEST_METHOD)
public class JpaProjectionIntegrationTest {
    // injected fields and test methods
}
```

За допомогою наданих анотацій Spring Boot створює базу даних, вводить залежності, заповнює та очищає таблиці до та після виконання кожного тестового методу.

**Закриті проекції**

Озираючись на клас Address, ми бачимо, що він має багато властивостей, але не всі вони корисні. Наприклад, іноді для позначення адреси достатньо поштового індексу.

Давайте оголосимо інтерфейс проекції для класу Address:

```
public interface AddressView {
    String getZipCode();
}
```

Потім ми використаємо його в інтерфейсі сховища:

```
public interface AddressRepository extends Repository<Address, Long> {
    List<AddressView> getAddressByState(String state);
}
```

Легко побачити, що визначення методу репозиторію за допомогою інтерфейсу проекції майже таке ж, як і для класу сутності.

Єдина відмінність полягає в тому, що інтерфейс проекції, а не клас сутності, використовується як тип елемента у поверненій колекції.

Давайте швидко перевіримо проекцію Address:

```
@Autowired
private AddressRepository addressRepository;
@Test
public void whenUsingClosedProjections_thenViewWithRequiredPropertiesIsReturned() {
	AddressView addressView = addressRepository.getAddressByState("CA").get(0);
	assertThat(addressView.getZipCode()).isEqualTo("90001");
	// ...
}
```

За лаштунками Spring створює екземпляр проксі-сервера інтерфейсу проекції для кожного об'єкта сутності, і всі виклики проксі-сервера перенаправляються на цей об'єкт.

Ми можемо використовувати проекції рекурсивно. Ось інтерфейс проекції для класу Person:

```
public interface PersonView {
	String getFirstName();
	String getLastName();
}
```

Тепер ми додамо метод із типом повернення PersonView, вкладену проекцію, у проекцію Address:

```
public interface AddressView {
	// ...
	PersonView getPerson();
}
```

Зверніть увагу, що метод, який повертає вкладену проекцію, повинен мати те саме ім'я, що й метод у кореневому класі, який повертає пов'язану сутність.

Ми перевіримо вкладені проекції, додавши кілька тверджень до щойно написаного методу тестування:

```
// ...
PersonView personView = addressView.getPerson();
assertThat(personView.getFirstName()).isEqualTo("John");
assertThat(personView.getLastName()).isEqualTo("Doe");
```

Зауважте, що рекурсивні проекції працюють, лише якщо ми переходимо від власної сторони до зворотної сторони. Якщо ми зробимо це навпаки, вкладена проекція матиме значення null.

**Відкриті проекції**

До цього моменту ми розглядали закриті проекції, які вказують на інтерфейси проекцій, методи яких точно відповідають іменам властивостей сутності.

Існує також інший вид проекції на основі інтерфейсу, відкриті проекції. Ці проекції дозволяють нам визначати методи інтерфейсу з невідповідними іменами та зі значеннями, що повертаються, обчисленими під час виконання.

Давайте повернемося до інтерфейсу проекції Person та додамо новий метод:

```
public interface PersonView {
    // ...
    @Value("#{target.firstName + ' ' + target.lastName}")
    String getFullName();
}
```

Аргументом анотації @Value є покажчиком цілі, який вказує на об'єкт резервної сутності.

Тепер ми визначимо інший інтерфейс сховища:

```
public interface PersonRepository extends Repository<Person, Long> {
    PersonView findByLastName(String lastName);
}
```

Щоб спростити це, ми повернемо лише один об'єкт проекції замість колекції.

Цей тест підтверджує, що відкриті прогнози працюють належним чином:

```
@Autowired
private PersonRepository personRepository;
@Test
public void whenUsingOpenProjections_thenViewWithRequiredPropertiesIsReturned() {
    PersonView personView = personRepository.findByLastName("Doe");
    assertThat(personView.getFullName()).isEqualTo("John Doe");
}
```

Однак відкриті прогнози мають недолік. Spring Data не може оптимізувати виконання запиту, оскільки не знає заздалегідь, які властивості будуть використані. Таким чином, ми повинні використовувати відкриті проекції лише тоді, коли закриті проекції не здатні впоратися з нашими вимогами.

**Проекції на основі класів**

Замість використання проксі-серверів, які Spring Data створює з інтерфейсів проекції, ми можемо визначити власні класи проекції.

Наприклад, ось клас проекції для сутності Person:

```
public class PersonDto {
    private String firstName;
    private String lastName;
    public PersonDto(String firstName, String lastName) {
```

```
        this.firstName = firstName;
        this.lastName = lastName;
    }
    // getters, equals and hashCode
}
```

Щоб клас проекції працював у тандемі з інтерфейсом репозиторію, імена параметрів його конструктора повинні збігатися з властивостями кореневого класу сутності.

Ми також повинні визначити втілення equals і hashCode; вони дозволяють Spring Data обробляти об'єкти проекції в колекції.

Тепер давайте додамо метод до інтерфейсу Person:

```
public interface PersonRepository extends Repository<Person, Long> {
    // ...
    PersonDto findByFirstName(String firstName);
}
```

Цей тест перевіряє нашу проекцію на основі класу:

```
@Test
public void whenUsingClassBasedProjections_thenDtoWithRequiredPropertiesIsReturned() {
    PersonDto personDto = personRepository.findByFirstName("John");
    assertThat(personDto.getFirstName()).isEqualTo("John");
    assertThat(personDto.getLastName()).isEqualTo("Doe");
    }
```

Зауважте, що з підходом на основі класів ми не можемо використовувати вкладені проекції.

## *3.3 Лабораторна робота № 3*
## *Spring MVC*

### 3.3.1 Мета роботи

Здобуття практичних навичок використання Spring MVC. Реалізація шару представлення корпоративної програми. Оцінка продуктивності та тестування програми.

### 3.3.2 Теоретичні відомості

**Spring MVC**

Spring MVC – це модуль фреймворку Spring, який працює з шаблоном Model-View-Controller, скорочено MVC. Модель — це не що інше, як дані нашої програми, а представлення представленого будь-яким із різних механізмів шаблонів. Spring реалізує MVC із шаблоном переднього контролера за допомогою свого DispatcherServlet. DispatcherServlet діє як головний контролер для маршрутизації запитів до їхнього призначення.

### 3.3.3 Порядок виконання роботи

Щоб увімкнути підтримку Spring MVC через клас конфігурації Java, ми просто додаємо анотацію @EnableWebMvc:

```
@EnableWebMvc
@Configuration
public class WebConfig {
    /// ...
}
```

Це налаштує базову підтримку, необхідну для проєкту MVC, таку як реєстрація контролерів і відображень, конвертери типів, підтримка перевірки, конвертери повідомлень і обробка винятків.

Якщо ми хочемо налаштувати цю конфігурацію, нам потрібно реалізувати інтерфейс WebMvcConfigurer:

```
@EnableWebMvc
@Configuration
public class WebConfig implements WebMvcConfigurer {
    @Override
    public void addViewControllers(ViewControllerRegistry registry) {
        registry.addViewController("/").setViewName("index");
    }
    @Bean
    public ViewResolver viewResolver() {
    InternalResourceViewResolver bean = new InternalResourceViewResolver();
    bean.setViewClass(JstlView.class);
```

```
        bean.setPrefix("/WEB-INF/view/");
        bean.setSuffix(".jsp");
        return bean;
        }
}
```

У цьому прикладі ми зареєстрували компонент ViewResolver, який повертатиме перегляди .jsp із каталогу /WEB-INF/view.

Тут дуже важливо те, що ми можемо зареєструвати контролери перегляду, які створюють пряме відображення між URL-адресою та назвою перегляду за допомогою ViewControllerRegistry. Таким чином, немає потреби в жодному контролері між ними.

Якщо ми також хочемо визначити та сканувати класи контролерів, ми можемо додати анотацію @ComponentScan до пакета, який містить контролери:

```
@EnableWebMvc
@Configuration
@ComponentScan(basePackages = { "com.baeldung.web.controller" })
public class WebConfig implements WebMvcConfigurer { // ... }
```

Для завантаження програми, яка завантажує цю конфігурацію, нам також потрібен клас ініціалізатора:

```
public class MainWebAppInitializer implements WebApplicationInitializer {
        @Override
        public void onStartup(final ServletContext sc) throws ServletException {
                AnnotationConfigWebApplicationContext root = new
                AnnotationConfigWebApplicationContext();
                root.scan("com.baeldung"); sc.addListener(new ContextLoaderListener(root));
                ServletRegistration.Dynamic appServlet = sc.addServlet("mvc", new DispatcherServlet(new
                GenericWebApplicationContext()));
                appServlet.setLoadOnStartup(1);
                appServlet.addMapping("/");
                }
}
```

Зверніть увагу, що для версій, раніших за Spring 5, ми повинні використовувати клас WebMvcConfigurerAdapter замість інтерфейсу.

**Контролер і представлення**

Давайте подивимося на приклад базового контролера:

```
        @Controller
        public class SampleController {
                @GetMapping("/sample")
                public String showForm() {
```

```
                return "sample";
        }
}
```

А відповідним ресурсом JSP є файл sample.jsp:

```
<html>
        <head></head>
        <body>
                <h1>This is the body of the sample view</h1>
        </body>
</html>
```

Файли перегляду на основі JSP розташовані в папці /WEB-INF проєкту, тому вони доступні лише для інфраструктури Spring, а не через прямий URL-доступ.

## *3.4 Лабораторна робота № 4*
## *Реалізація мікросервісної архітектури застосування*

### 3.4.1 Мета роботи

Здобуття практичних навичок використання Spring microservices.

### 3.4.2 Теоретичні відомості

**Spring Microservices**

Мікросервіси дозволяють створювати великі системи з ряду взаємодіючих компонентів. Неминуче існує ряд рухомих частин, які ви повинні встановити та налаштувати, щоб створити систему.

### 3.4.3 Порядок виконання роботи

Давайте поглянемо на клас Book:

```
1  public class Book {
2      private long id;
3      private String title;
4      private String author;
5  
6      // standard getters and setters
7  }
```

А також на клас Rating, що буде підтримувати другий API:

```
1  public class Rating {
2      private long id;
3      private Long bookId;
4      private int stars;
5  
6      // standard getters and setters
7  }
```

Тепер ми зробимо два простих API – /books і /ratings. Спочатку напишімо наш Books API:

```java
@RestController
@RequestMapping("/books")
public class BookController {
    @Autowired
    private BookService bookService;

    @GetMapping
    public List<Book> findAllBooks() {
        return bookService.findAllBooks();
    }

    @GetMapping("/{bookId}")
    public Book findBook(@PathVariable Long bookId) {
        return bookService.findBookById(bookId);
    }

    @PostMapping
    public Book createBook(@RequestBody Book book) {
        return bookService.createBook(book);
    }

    @DeleteMapping("/{bookId}")
    public void deleteBook(@PathVariable Long bookId) {
        bookService.deleteBook(bookId);
    }
    @PutMapping("/{bookId}")
    public Book updateBook(@RequestBody Book book, @PathVariable Long bookId) {
        return bookService.updateBook(book, bookId);
    }
    @PatchMapping("/{bookId}")
    public Book updateBook(
        @RequestBody Map<String, String> updates,
        @PathVariable Long bookId) {
         return bookService.updateBook(updates, bookId);
    }
}
```

І схожим чином зробимо для класу RatingController:

```
1   @RestController
2   @RequestMapping("/ratings")
3   public class RatingController {
4
5       @Autowired
6       private RatingService ratingService;
7
8       @GetMapping
9       public List<Rating> findRatingsByBookId(
10        @RequestParam(required = false, defaultValue = "0") Long bookId) {
11          if (bookId.equals(0L)) {
12              return ratingService.findAllRatings();
13          }
14          return ratingService.findRatingsByBookId(bookId);
15      }
16
17      @PostMapping
18      public Rating createRating(@RequestBody Rating rating) {
19          return ratingService.createRating(rating);
20      }
21
22      @DeleteMapping("/{ratingId}")
23      public void deleteRating(@PathVariable Long ratingId) {
24          ratingService.deleteRating(ratingId);
25      }
26
27      @PutMapping("/{ratingId}")
28      public Rating updateRating(@RequestBody Rating rating, @PathVariable Long ratingId) {
29          return ratingService.updateRating(rating, ratingId);
30      }
31      @PatchMapping("/{ratingId}")
32      public Rating updateRating(
33        @RequestBody Map<String, String> updates,
34        @PathVariable Long ratingId) {
35          return ratingService.updateRating(updates, ratingId);
36      }
37  }
```

Кожен API має власний окремий завантажувальний додаток і розгортається повністю незалежно від будь-чого іншого. Після локального розгортання API будуть доступні за адресою:

```
1 http://localhost:8080/book-service/books
2 http://localhost:8080/rating-service/ratings
```

**Конфігурація безпеки**

Наступним кроком є захист двох API. Спочатку зробимо конфігурація безпеки програми для класу Book:

```
1  @EnableWebSecurity
2  @Configuration
3  public class SecurityConfig extends WebSecurityConfigurerAdapter {
4
5      @Autowired
6      public void configureGlobal1(AuthenticationManagerBuilder auth)
7        throws Exception {
8          auth.inMemoryAuthentication();
9      }
10
11     @Override
12     protected void configure(HttpSecurity http) throws Exception {
13         http.httpBasic()
14           .disable()
15         .authorizeRequests()
16           .antMatchers(HttpMethod.GET, "/books").permitAll()
17           .antMatchers(HttpMethod.GET, "/books/*").permitAll()
18           .antMatchers(HttpMethod.POST, "/books").hasRole("ADMIN")
19           .antMatchers(HttpMethod.PATCH, "/books/*").hasRole("ADMIN")
20           .antMatchers(HttpMethod.DELETE, "/books/*").hasRole("ADMIN")
21           .anyRequest().authenticated()
22           .and()
23         .csrf()
24           .disable();
25     }
26 }
```

А тепер для конфігурація класу Rating:

```
@EnableWebSecurity
@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Autowired
    public void configureGlobal1(AuthenticationManagerBuilder auth)
      throws Exception {
        auth.inMemoryAuthentication();
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.httpBasic()
          .disable()
        .authorizeRequests()
          .regexMatchers("^/ratings\\?bookId.*$").authenticated()
          .antMatchers(HttpMethod.POST,"/ratings").authenticated()
          .antMatchers(HttpMethod.PATCH,"/ratings/*").hasRole("ADMIN")
          .antMatchers(HttpMethod.DELETE,"/ratings/*").hasRole("ADMIN")
          .antMatchers(HttpMethod.GET,"/ratings").hasRole("ADMIN")
          .anyRequest().authenticated()
          .and()
        .csrf()
          .disable();
    }
}
```

Оскільки API прості, ми можемо використовувати глобальні відповідники прямо в конфігурації безпеки. Однак, оскільки вони стають дедалі складнішими, ми повинні перенести їх на реалізацію анотації на рівні методу. Наразі семантика безпеки дуже проста:

- Будь-хто може читати ресурси
- Лише адміністратори можуть змінювати ресурси

**Spring Cloud Configuration**

Spring Cloud надає розробникам інструменти для швидкого створення деяких загальних шаблонів у розподілених системах (керування конфігураціями, виявлення служб, автоматичні вимикачі, інтелектуальна маршрутизація, керуюча шина, одноразові токени, глобальні блокування, розподілені сесії). Координація розподілених систем призводить до шаблонів, і за допомогою Spring Cloud розробники можуть швидко створити служби та програми, які реалізують ці шаблони. Вони добре працюватимуть у будь-якому розподіленому середовищі, включаючи власний ноутбук розробника, центри обробки даних.

Тепер, коли наші два API працюють незалежно, настав час подивитися на використання Spring Cloud і завантажити деякі дуже корисні компоненти в мікросервісу:

1. Configuration Server – надає, керує та централізує конфігурацію для зовнішньої конфігурації наших різних модулів.

2. Discovery Server — дозволяє програмам ефективно та гнучко знаходити один одного.

3. Gatway Server - діє як зворотний проксі та приховує складність нашої системи, надаючи всі наші API на одному порту.

4. Two REST APIs: Books API та Ratings API.

**Configuration Server**

Spring Cloud Config — це клієнт-серверний підхід Spring для зберігання та обслуговування розподілених конфігурацій у кількох програмах і середовищах.

Для цього нам знадобляться Cloud Config, Eureka та Security. Нам потрібно використовувати @EnableConfigServer, щоб зробити наш сервер конфігурації видимим через клієнт Eureka, як показано нижче:

```
1  @SpringBootApplication
2  @EnableConfigServer
3  @EnableEurekaClient
4  public class ConfigApplication {...}
```

application.properties повинен бути таким:

```
1   server.port=8081
2   spring.application.name=config
3   spring.cloud.config.server.git.uri=file:///${user.home}/application-config
4   eureka.client.region=default
5   eureka.client.registryFetchIntervalSeconds=5
6   eureka.client.serviceUrl.defaultZone=
7     http://discUser:discPassword@localhost:8082/eureka/
8   security.user.name=configUser
9   security.user.password=configPassword
10  security.user.role=SYSTEM
```

**Discovery Server**

Ми налаштуємо наш discovery server, спочатку додавши анотацію @EnableEurekaServer:

```
1  @SpringBootApplication
2  @EnableEurekaServer
3  public class DiscoveryApplication {...}
```

І переконайтеся, що нам також потрібно захистити наші кінцеві точки discovery server:

```
@Configuration
@EnableWebSecurity
@Order(1)
public class SecurityConfig extends WebSecurityConfigurerAdapter {
   @Autowired
    public void configureGlobal(AuthenticationManagerBuilder auth) {
        auth.inMemoryAuthentication()
            .withUser("discUser")
            .password("discPassword")
            .roles("SYSTEM");
   }

   @Override
   protected void configure(HttpSecurity http) {
       http
       .sessionManagement()
         .sessionCreationPolicy(SessionCreationPolicy.ALWAYS).and()
       .requestMatchers().antMatchers("/eureka/**").and()
       .authorizeRequests()
         .antMatchers("/eureka/**").hasRole("SYSTEM")
         .anyRequest().denyAll().and()
       .httpBasic().and()
       .csrf().disable();
   }
}
```

А також, щоб захистити інформаційну панель Eureka:

```
@Configuration
public static class AdminSecurityConfig extends WebSecurityConfigurerAdapter {
    @Override
    protected void configure(HttpSecurity http) {
       http
       .sessionManagement()
         .sessionCreationPolicy(SessionCreationPolicy.NEVER).and()
       .httpBasic().disable()
       .authorizeRequests()
         .antMatchers(HttpMethod.GET, "/").hasRole("ADMIN")
         .antMatchers("/info", "/health").authenticated()
         .anyRequest().denyAll().and()
       .csrf().disable();
    }
}
```

Тепер ми додамо bootstrap.properties у папку ресурсів нашого discovery server таким чином:

```
spring.cloud.config.name=discovery
spring.cloud.config.uri=http://localhost:8081
spring.cloud.config.username=configUser
spring.cloud.config.password=configPassword
```

Також ми додамо discovery.properties у наш Git-репозиторій конфігурації програми:

```
1  spring.application.name=discovery
2  server.port=8082
3  eureka.instance.hostname=localhost
4  eureka.client.serviceUrl.defaultZone=
5    http://discUser:discPassword@localhost:8082/eureka/
6  eureka.client.register-with-eureka=false
7  eureka.client.fetch-registry=false
8  spring.redis.host=localhost
9  spring.redis.port=6379
```

Зауважте, що:

- Ми використовуємо @Order(1), оскільки маємо захистити конфігурації для discovery server, один для кінцевих точок, а інший для інформаційної панелі.
- spring.cloud.config.name має збігатися з файлом властивостей discovery server в сховищі конфігурації.
- Ми повинні надати spring.cloud.config.uri у властивостях початкового завантаження discovery server, щоб мати можливість отримати повну конфігурацію з configuration server.

**Gateway Server**

Далі нам потрібно налаштувати наш gateway server наступним чином:

```
1  @SpringBootApplication
2  @EnableZuulProxy
3  @EnableEurekaClient
4  public class GatewayApplication {}
```

Додамо просту конфігурацію безпеки:

```
@EnableWebSecurity
@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    @Autowired
    public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {
        auth.inMemoryAuthentication()
            .withUser("user").password("password").roles("USER")
            .and()
            .withUser("admin").password("admin").roles("ADMIN");
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
        .authorizeRequests()
          .antMatchers("/book-service/books").permitAll()
          .antMatchers("/eureka/**").hasRole("ADMIN")
          .anyRequest().authenticated().and()
        .formLogin().and()
        .logout().permitAll().and()
        .csrf().disable();
    }
}
```

Нам також потрібно додати bootstrap.properties у папку ресурсів нашого gateway server, як показано нижче:

```
spring.cloud.config.name=gateway
spring.cloud.config.discovery.service-id=config
spring.cloud.config.discovery.enabled=true
spring.cloud.config.username=configUser
spring.cloud.config.password=configPassword
eureka.client.serviceUrl.defaultZone=
  http://discUser:discPassword@localhost:8082/eureka/
```

Нарешті, ми додамо gateway.properties у наш Git-репозиторій конфігурації програми:

```
spring.application.name=gateway
server.port=8080
eureka.client.region = default
eureka.client.registryFetchIntervalSeconds = 5
management.security.sessions=always

zuul.routes.book-service.path=/book-service/**
zuul.routes.book-service.sensitive-headers=Set-Cookie,Authorization
hystrix.command.book-service.execution.isolation.thread
  .timeoutInMilliseconds=600000
zuul.routes.rating-service.path=/rating-service/**
zuul.routes.rating-service.sensitive-headers=Set-Cookie,Authorization
hystrix.command.rating-service.execution.isolation.thread
  .timeoutInMilliseconds=600000
zuul.routes.discovery.path=/discovery/**
zuul.routes.discovery.sensitive-headers=Set-Cookie,Authorization
zuul.routes.discovery.url=http://localhost:8082
hystrix.command.discovery.execution.isolation.thread
  .timeoutInMilliseconds=600000

spring.redis.host=localhost
spring.redis.port=6379
```

Примітка: ми використовуємо zuul.routes.book-service.path для маршрутизації будь-якого запиту, який надходить на /book-service/**, до нашої програми Book Service, те ж саме стосується нашої Rating Service.

**REST APIs**

Ми також будемо використовувати @EnableEurekaClient для обох API, як показано нижче:

```
@SpringBootApplication
@EnableEurekaClient
public class ServiceApplication {...}
```

Додамо до Book Service конфігурація властивостей book-service.properties:

```
spring.application.name=book-service
server.port=8083
eureka.client.region=default
eureka.client.registryFetchIntervalSeconds=5
management.security.sessions=never
```

А також додамо до bootstrap.properties Book Service:

```
1  spring.cloud.config.name=book-service
2  spring.cloud.config.discovery.service-id=config
3  spring.cloud.config.discovery.enabled=true
4  spring.cloud.config.username=configUser
5  spring.cloud.config.password=configPassword
6  eureka.client.serviceUrl.defaultZone=
7    http://discUser:discPassword@localhost:8082/eureka/
```

Додамо до Rating Service rating-service.properties:

```
1  spring.application.name=rating-service
2  server.port=8084
3  eureka.client.region=default
4  eureka.client.registryFetchIntervalSeconds=5
5  management.security.sessions=never
```

Також треба додати до bootstrap.properties:

```
1  spring.cloud.config.name=rating-service
2  spring.cloud.config.discovery.service-id=config
3  spring.cloud.config.discovery.enabled=true
4  spring.cloud.config.username=configUser
5  spring.cloud.config.password=configPassword
6  eureka.client.serviceUrl.defaultZone=
7    http://discUser:discPassword@localhost:8082/eureka/
```

**Session Configuration**

Ми будемо ділитися сеансами між різними службами в нашій системі за допомогою Spring Session. Сеанси спільного доступу дозволяють реєструвати користувачів у нашій службі шлюзу та поширювати цю автентифікацію на інші служби.

Нам потрібно додати конфігурацію сеансу до нашого discovery server та REST API:

```
1  @EnableRedisHttpSession
2  public class SessionConfig
3    extends AbstractHttpSessionApplicationInitializer {
4  }
```

Для gateway server треба буде написати дещо іншимчином, як показано нижче:

```
@Configuration
@EnableRedisHttpSession(redisFlushMode = RedisFlushMode.IMMEDIATE)
public class SessionConfig extends AbstractHttpSessionApplicationInitializer {
}
```

Ми також додаємо простий фільтр до нашого сервера шлюзу для перенаправлення сесії, щоб автентифікація поширювалася на іншу службу після входу:

```
@Component
public class SessionSavingZuulPreFilter
  extends ZuulFilter {

    @Autowired
    private SessionRepository repository;

    @Override
    public boolean shouldFilter() {
        return true;
    }

    @Override
    public Object run() {
        RequestContext context = RequestContext.getCurrentContext();
        HttpSession httpSession = context.getRequest().getSession();
        Session session = repository.getSession(httpSession.getId());

        context.addZuulRequestHeader(
          "Cookie", "SESSION=" + httpSession.getId());
        return null;
    }

    @Override
    public String filterType() {
        return "pre";
    }

    @Override
    public int filterOrder() {
        return 0;
    }
}
```

**Test REST API**

Зараз ми протестуємо наш REST API. Для цього зробимо просте налаштування:

```
private final String ROOT_URI = "http://localhost:8080";
private FormAuthConfig formConfig
  = new FormAuthConfig("/login", "username", "password");

@Before
public void setup() {
    RestAssured.config = config().redirect(
      RedirectConfig.redirectConfig().followRedirects(false));
}
```

Візьмемо всі книги:

```
@Test
public void whenGetAllBooks_thenSuccess() {
    Response response = RestAssured.get(ROOT_URI + "/book-service/books");

    Assert.assertEquals(HttpStatus.OK.value(), response.getStatusCode());
    Assert.assertNotNull(response.getBody());
}
```

Спробуймо отримати доступ до захищеного ресурсу без авторизації:

```
@Test
public void whenAccessProtectedResourceWithoutLogin_thenRedirectToLogin() {
    Response response = RestAssured.get(ROOT_URI + "/book-service/books/1");

    Assert.assertEquals(HttpStatus.FOUND.value(), response.getStatusCode());
    Assert.assertEquals("http://localhost:8080/login",
      response.getHeader("Location"));
}
```

Увійдіть і створіть нову книгу:

```
1  @Test
2  public void whenAddNewBook_thenSuccess() {
3      Book book = new Book();
4      book.setTitle("How to spring cloud");
5      book.setAuthor("Baeldung");
6
7      Response bookResponse = RestAssured.given()
8        .auth()
9        .form("admin", "admin", formConfig)
10       .and()
11       .contentType(ContentType.JSON)
12       .body(book)
13       .post(ROOT_URI + "/book-service/books");
14
15     Book result = bookResponse.as(Book.class);
16
17     Assert.assertEquals(HttpStatus.OK.value(), bookResponse.getStatusCode());
18     Assert.assertEquals(book.getAuthor(), result.getAuthor());
19     Assert.assertEquals(book.getTitle(), result.getTitle());
20 }
```

Отримати доступ до захищеного ресурсу після входу:

```
1  @Test
2  public void whenAccessProtectedResourceAfterLogin_thenSuccess() {
3      Response response = RestAssured.given().auth()
4        .form("user", "password", formConfig)
5        .get(ROOT_URI + "/book-service/books/1");
6
7      Assert.assertEquals(HttpStatus.OK.value(), response.getStatusCode());
8      Assert.assertNotNull(response.getBody());
9  }
```

Створимо новий рейтинг:

```java
@Test
public void whenAddNewRating_thenSuccess() {
    Rating rating = new Rating();
    rating.setBookId(1L);
    rating.setStars(4);

    Response ratingResponse = RestAssured.given()
      .auth()
      .form("admin", "admin", formConfig)
      .and()
      .contentType(ContentType.JSON)
      .body(rating)
      .post(ROOT_URI + "/rating-service/ratings");

    Rating result = ratingResponse.as(Rating.class);

    Assert.assertEquals(HttpStatus.OK.value(), ratingResponse.getStatusCode());
    Assert.assertEquals(rating.getBookId(), result.getBookId());
    Assert.assertEquals(rating.getStars(), result.getStars());
}
```

Спробуйте отримати доступ до захищеного адміністратором рейтингового ресурсу:

```java
@Test
public void whenAccessAdminProtectedResource_thenForbidden() {
    Response response = RestAssured.given().auth()
      .form("user", "password", formConfig)
      .get(ROOT_URI + "/rating-service/ratings");

    Assert.assertEquals(HttpStatus.FORBIDDEN.value(), response.getStatusCode());
}
```

Отримайте доступ до захищеного рейтингу, увійшовши за допомогою адміністратора:

```java
@Test
public void whenAdminAccessProtectedResource_thenSuccess() {
    Response response = RestAssured.given().auth()
      .form("admin", "admin", formConfig)
      .get(ROOT_URI + "/rating-service/ratings");

    Assert.assertEquals(HttpStatus.OK.value(), response.getStatusCode());
    Assert.assertNotNull(response.getBody());
}
```

Увійдіть до ресурсу discovery як адміністратор:

```
1  @Test
2  public void whenAdminAccessDiscoveryResource_thenSuccess() {
3      Response response = RestAssured.given().auth()
4        .form("admin", "admin", formConfig)
5        .get(ROOT_URI + "/discovery");
6
7      Assert.assertEquals(HttpStatus.OK.value(), response.getStatusCode());
8  }
```

**Рекомендована література**


1. Martin Fowler. Patterns of Enterprise Application Architecture. – 2002. – 560 p.
2. Фаулер, М. Архітектура корпоративних програмних систем: пров. з англ. - М.: Видавничий дім "Вільямс", 2006. - 544 с.
3. Robert C. Martin. Clean Architecture: A Craftsman's Guide to Software Structure and Design. – Prentice Hall, 2017. – 432 p.
4. The Jakarta EE Tutorial Release Version 9.1. – December 2021. – [Електронний ресурс]. – Режим доступу: https://eclipse-ee4j.github.io/jakartaee-tutorial.
5. David Heffelfinger. Demystifying Microservices for Jakarta EE Developers / Payara User Guide. - 34 p.
6. Craig Walls. Spring in Action. Fifth Edition, 2019.
7. Craig Walls. Spring in Action, Sixth Edition. – [Електронний ресурс]. – Режим доступу: https://www.manning.com/books/spring-in-action-sixth-edition.
8. Bauer C., King G. Java Persistence with Hibernate. – Manning, 2016. – 610 p.
9. Baeldung. – [Електронний ресурс]. – https://www.baeldung.com/.
10. Spring | Home. – [Електронний ресурс]. – https://spring.io/.


**Інформаційні ресурси**


1. Jakarta Enterprise Edition Your First Cup: An Introduction to Jakarta EE, Release 8. – [Електронний ресурс]. – Режим доступу: https://eclipse-ee4j.github.io/jakartaee-firstcup/toc.html.
2. Jakarta EE Platform Specification Version 9.1 Final Release. –April 27, 2021. – [Електронний ресурс]. – Режим доступу: https://jakarta.ee/specifications/platform/9.1/jakarta-platform-spec-9.1.html.
3. Github: Jakartaee Tutorial Examples. – [Електронний ресурс]. – Режим доступу: https://github.com/eclipse-ee4j/jakartaee-tutorial-examples.
4. MicroProfile | projects.eclipse.org. – [Електронний ресурс]. – Режим доступу:https://projects.eclipse.org/projects/technology.microprofile
5. Home MicroProfile. – [Електронний ресурс]. – Режим доступу: https://microprofile.io/.