Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Факультет інформатики та обчислювальної техніки
Кафедра інформаційних систем та технологій

Лабораторна робота № 9

з дисципліни «Технології розроблення програмного забезпечення»

Тема: «Взаємодія компонентів системи»

Виконала:
студентка групи ІА-32
Глущенко Анастасія
Сергіївна

Перевірив:
Мягкий Михайло
Юрійович

Київ 2025

Зміст

Вступ

Метою роботи є вивчення видів взаємодії додатків (Client-Server, Peer-to-Peer, Serviceoriented Architecture), та реалізування в проєктованій системі архітектуру Client-Server.

Було обрано тему:
Installer generator (iterator, builder, factory method, bridge, interpreter, client-server)
Генератор інсталяційних пакетів повинен мати якийсь спосіб налаштування файлів, що входять в установку, установки вікон з інтерактивними можливостями (галочка – створити ярлик на робочому столі; ввести в текстове поле деякі дані, наприклад, ліцензійний ключ і т.д.). Генератор повинен вивести один файл .exe або .msi

Теоретичні відомості

1. «Client-Server» (Клієнт-Сервер)

Архітектурна модель розподілених обчислень, яка розподіляє завдання або робоче навантаження між постачальниками ресурсів або послуг (серверами) і замовниками послуг (клієнтами).Ключова ідея: Чіткий поділ обов'язків та централізація ресурсів. Сервер завжди очікує на запит, а клієнт завжди виступає ініціатором взаємодії.Застосування:

- Веб-додатки (браузер — клієнт, веб-сервер обробляє запити).
- Корпоративні бази даних (клієнт надсилає SQL-запити, сервер БД повертає дані).
- Електронна пошта, онлайн-ігри.
- Коли необхідна централізована безпека, керування даними та резервне копіювання.

Структура та реалізація

- Client (Клієнт): Активний компонент. Надсилає запити на сервер, очікує відповіді та відображає результат користувачеві. Зазвичай має обмежені обчислювальні ресурси («тонкий клієнт») або виконує частину обробки («товстий клієнт»).
- Server (Сервер): Пасивний компонент (слухає мережу). Приймає запити, обробляє їх (бізнес-логіка, доступ до БД) і надсилає відповідь. Має потужні обчислювальні ресурси.
- Communication Protocol (Протокол взаємодії): Набір правил, за якими клієнт і сервер обмінюються даними (наприклад, HTTP, TCP/IP, FTP).

2. «Peer-to-Peer» (P2P / Рівноправна архітектура)

Децентралізована мережева архітектура, в якій кожен учасник (вузол) є водночас і клієнтом, і сервером. Вузли об'єднують свої ресурси (обчислювальну потужність, пам'ять, пропускну здатність) для виконання спільного завдання.Ключова ідея: Рівноправність учасників та відсутність

єдиної точки відмови. Чим більше учасників у мережі, тим більша її загальна продуктивність і надійність.Застосування:

- Файлообмінні мережі (наприклад, BitTorrent).
- Блокчейн та криптовалюти (Bitcoin, Ethereum).
- Розподілені обчислення (наприклад, наукові проекти типу Folding@home).
- VoIP (Voice over IP) та месенджери без центрального сервера (на ранніх етапах Skype, Tox).

Структура та реалізація

- Peer (Вузол / Бенкет): Рівноправний учасник мережі. Виконує роль клієнта, коли завантажує дані, і роль сервера, коли роздає їх іншим.
- Overlay Network (Накладена мережа): Логічна мережа, яка будується поверх фізичної (Інтернет) для пошуку та з'єднання вузлів між собою.
- Discovery Mechanism (Механізм виявлення): Спосіб, яким вузли знаходять один одного (через центральні трекери або децентралізовані хеш-таблиці — DHT).

3. «Service-Oriented Architecture» (SOA / Сервіс-орієнтована архітектура)

Архітектурний підхід до розробки програмного забезпечення, де компоненти програми надають послуги (сервіси) іншим компонентам через комунікаційний протокол по мережі.Ключова ідея: Слабка зв'язність (Loose Coupling). Система будується з незалежних "чорних скриньок" (сервісів), які мають чітко визначені інтерфейси і можуть бути написані різними мовами та розміщені на різних платформах.Застосування:

- Інтеграція різнорідних корпоративних систем (Enterprise Application Integration), наприклад, зв'язок CRM, ERP та білінгу.
- Побудова складних бізнес-процесів, які охоплюють кілька департаментів або організацій.

- Масштабовані веб-системи (сучасна еволюція SOA — це мікросервісна архітектура).

Структура та реалізація

- Service Provider (Постачальник сервісу): Розробляє сервіс, реалізує його функціональність та публікує його опис (контракт/інтерфейс) у реєстрі.
- Service Consumer (Споживач сервісу): Додаток або інший сервіс, який знаходить потрібний сервіс у реєстрі, зв'язується з ним і використовує його функції.
- Service Registry / Broker (Реєстр сервісів): Централізований каталог, де постачальники публікують свої сервіси, а споживачі їх знаходять (наприклад, UDDI).
- Enterprise Service Bus (ESB, Шина даних): Програмний шар, який забезпечує передачу повідомлень між сервісами, їх маршрутизацію та перетворення форматів даних.

Хід роботи

1. Архітектурний стиль «Client-Server» (Клієнт-Сервер)

Призначення: клієнт-серверна архітектура - це модель взаємодії в розподілених системах, яка поділяє завдання та робоче навантаження між постачальниками ресурсу або послуги, що називаються серверами, та замовниками послуг, що називаються клієнтами. У контексті веб-розробки це дозволяє відокремити логіку обробки та зберігання даних (Back-end) від логіки представлення та взаємодії з користувачем (Front-end/External Client), забезпечуючи обмін даними через мережу за допомогою стандартизованих протоколів (HTTP).

Проблема, яку вирішує архітектура: початкова версія проекту була реалізована як монолітний веб-додаток, де інтерфейс користувача (HTML-сторінки) був жорстко пов'язаний з логікою на сервері. Це створювало обмеження: взаємодія з системою була можлива виключно через веб-браузер людиною. Зовнішні системи, скрипти автоматизації або мобільні додатки не мали програмного доступу до функціоналу генерації інсталяторів. Була необхідність забезпечити можливість віддаленого виклику функцій генерації ("Machine-to-Machine" взаємодія) без прив'язки до візуального інтерфейсу.

Як архітектура вирішує проблему: для вирішення цієї проблеми було впроваджено REST API (Representational State Transfer). Це дозволило створити універсальний інтерфейс взаємодії, де:

- Сервер надає набір публічних точок доступу (Endpoints), які приймають та віддають дані у форматі JSON, ігноруючи особливості відображення.
- Клієнт (будь-яка зовнішня програма) формує запит із даними та надсилає його через мережу, отримуючи результат виконання операції.

- Це перетворило систему на розподілену, де клієнт і сервер можуть знаходитися на різних фізичних машинах і бути написаними на різних мовах програмування.

Процес взаємодії виглядає так:

- Клієнт формує об'єкт передачі даних (DTO), серіалізує його у формат JSON та відправляє HTTP POST запит на певну адресу сервера.
- Middleware (Загальна частина) забезпечує контракт даних - структура JSON відома обом сторонам.
- Сервер приймає запит, десеріалізує JSON у Java-об'єкт, виконує валідацію та бізнес-логіку (збереження в БД, запуск генерації).
- Сервер повертає HTTP-відповідь зі статусом (наприклад, 200 OK) та результатом операції (наприклад, ID створеного проекту).

Як реалізовано в проекті:

- Серверна частина: Реалізована на базі Spring Boot. Клас InstallerApiController виступає точкою входу REST API, приймаючи HTTP-запити та делегуючи обробку сервісному шару.
- Клієнтська частина: Реалізована у вигляді окремого Java-додатку SimpleClient (код створений у попередньому кроці), який емулює роботу зовнішньої системи, використовуючи HttpURLConnection для відправки запитів.
- Middleware (DTO): Клас ProjectRequestDto виконує роль об'єкта передачі даних, ізолюючи внутрішню структуру бази даних від зовнішнього API.
- Протокол: Використано протокол HTTP та формат даних JSON як сучасну альтернативу специфічним для .NET протоколам (WCF/TcpClient), що забезпечує кросплатформеність.

Таким чином, архітектура Client-Server забезпечує:

- Інтероперабельність: Система стала доступною для інтеграції з будь-якими клієнтами, що підтримують HTTP (мобільні додатки, CI/CD скрипти, Postman).
- Масштабованість: Серверна логіка відокремлена від клієнта; навантаження на відображення інтерфейсу перенесене на бік клієнта.
- Незалежність розвитку: Можна змінювати внутрішню реалізацію сервера або структуру бази даних, не ламаючи клієнтів, доки зберігається контракт API (структура DTO).

2. Діаграма класів, яка представляє спроєктовану архітектуру

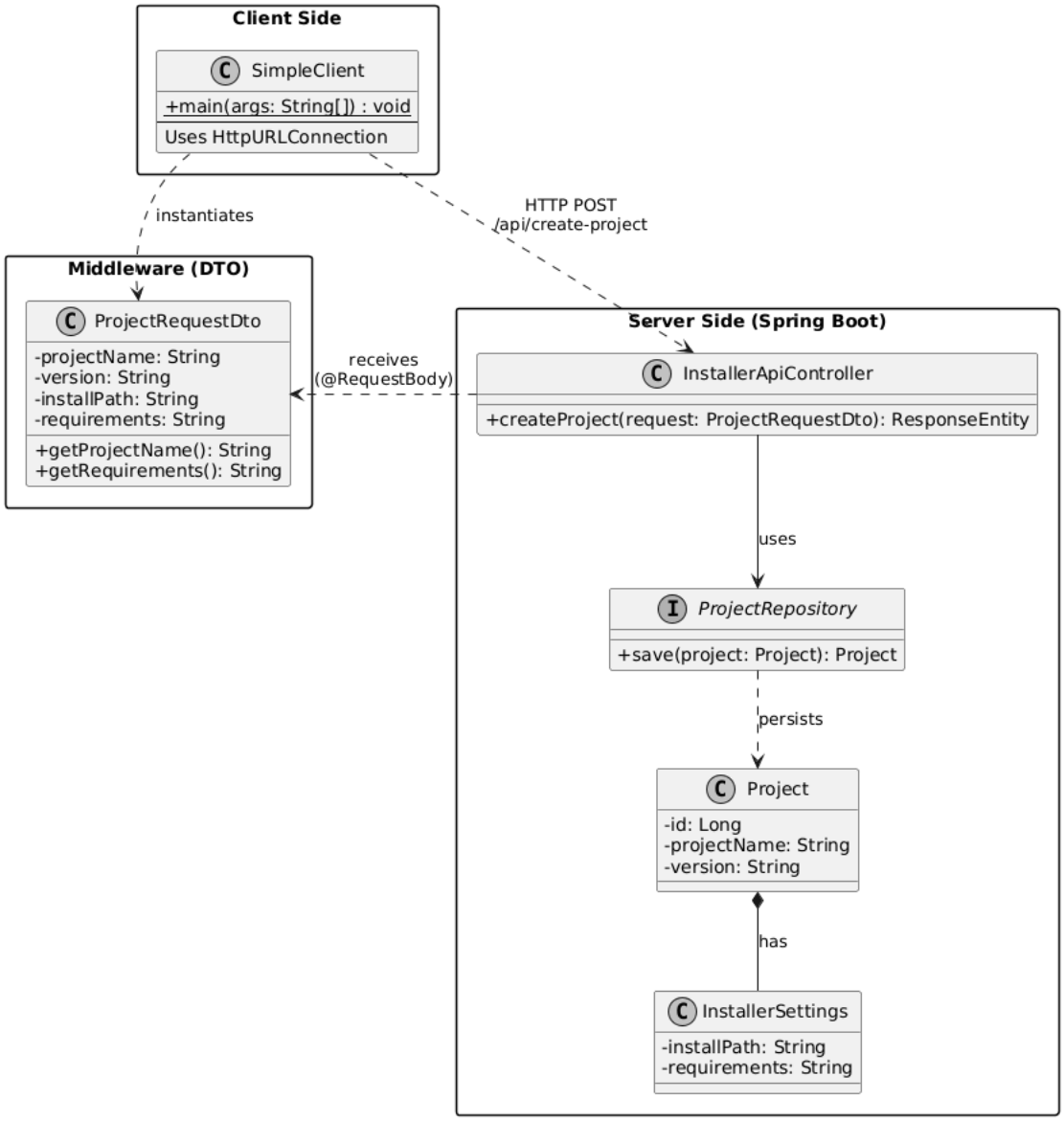

Рисунок 1 - Діаграма класів

Опис діаграми класів реалізації архітектури Client-Server

На діаграмі зображено структуру класів та їх взаємодію в межах реалізованої клієнт-серверної архітектури, побудованої на основі REST API. Система розділена на три логічні рівні: клієнтська частина (Client Side), серверна частина (Server Side) та шар обміну даними (Middleware/DTO).

1. Клієнтська частина (Client Side): Представлена класом **SimpleClient**, який виконує роль зовнішнього споживача послуг API. Цей клас є автономним Java-додатком, що містить метод main. Його основна відповідальність - ініціювати мережеву взаємодію: він формує JSON-структуру запиту та використовує стандартний клас HttpURLConnection для відправки HTTP POST-запиту на сервер. Зв'язок між SimpleClient та сервером є слабким, оскільки взаємодія відбувається через мережевий протокол, а не прямий виклик методів.

2. Шар обміну даними (Middleware): Представлений класом **ProjectRequestDto** (Data Transfer Object). Цей клас виступає контрактом (інтерфейсом даних) між клієнтом і сервером. Він містить лише ті поля, які необхідні для створення проекту (назва, версія, шлях інсталяції, вимоги), і не містить бізнес-логіки. Використання DTO дозволяє відокремити внутрішню модель даних сервера від формату зовнішніх запитів.

3. Серверна частина (Server Side):

- **InstallerApiController**: Це REST-контролер, який слугує точкою входу (Entry Point) на стороні сервера. Він приймає HTTP-запити, автоматично десеріалізує вхідний JSON у об'єкт ProjectRequestDto (завдяки анотації @RequestBody) та ініціює обробку даних.
- **ProjectRepository**: Інтерфейс, що забезпечує абстракцію доступу до бази даних. Контролер використовує його для збереження новостворених сутностей.
- **Project та InstallerSettings**: Класи сутностей, що відображають структуру бази даних. Між ними встановлено зв'язок композиції, що

означає сильний зв'язок життєвого циклу: об'єкт InstallerSettings не може існувати окремо від об'єкта Project.

Типи зв'язків: На діаграмі відображено потік даних зверху вниз:

- Клієнт залежить (dependency) від ProjectRequestDto, оскільки створює його екземпляри для формування запиту.
- Клієнт надсилає запит до InstallerApiController через протокол HTTP.
- Контролер використовує (association) ProjectRepository для персистентності даних.
- Репозиторій оперує класами сутностей (Project).

3. Фрагменти програмного коду, які відображають реалізовану архітектуру

Шар Middleware (Об'єкт передачі даних)

Клас **ProjectRequestDto** виступає контрактом даних між клієнтом і сервером. Він визначає структуру JSON-повідомлення, абстрагуючи клієнта від внутрішньої моделі бази даних.

```
package com.example.installer.dto;

public class ProjectRequestDto {
    private String projectName;
    private String version;

    private String installPath;
    private boolean createDesktopShortcut;
    private String language;
    private String mainClass;
    private String requirements;

    public ProjectRequestDto() {
    }

    public String getProjectName() { return projectName; }
    public void setProjectName(String projectName) { this.projectName =
projectName; }

    public String getVersion() { return version; }
    public void setVersion(String version) { this.version = version; }

    public String getInstallPath() { return installPath; }
    public void setInstallPath(String installPath) { this.installPath =
installPath; }

    public boolean isCreateDesktopShortcut() { return
createDesktopShortcut; }
```

```
    public void setCreateDesktopShortcut(boolean createDesktopShortcut) {
this.createDesktopShortcut = createDesktopShortcut; }

    public String getLanguage() { return language; }
    public void setLanguage(String language) { this.language = language; }

    public String getMainClass() { return mainClass; }
    public void setMainClass(String mainClass) { this.mainClass =
mainClass; }

    public String getRequirements() { return requirements; }
    public void setRequirements(String requirements) { this.requirements =
requirements; }
}
```

Серверна частина (REST Controller)

Клас **InstallerApiController** демонструє реалізацію точки доступу (Endpoint). Анотація @RestController вказує Spring Boot, що цей клас обробляє REST-запити, а @RequestBody автоматично перетворює вхідний JSON у об'єкт DTO.

```
package com.example.installer.controller;

import com.example.installer.InstallerSettings;
import com.example.installer.Project;
import com.example.installer.dto.ProjectRequestDto;
import com.example.installer.repository.ProjectRepository;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

@RestController
@RequestMapping("/api")
public class InstallerApiController {

    private final ProjectRepository projectRepository;

    public InstallerApiController(ProjectRepository projectRepository) {
        this.projectRepository = projectRepository;
    }

    @PostMapping("/create-project")
    public ResponseEntity<?> createProject(@RequestBody ProjectRequestDto
request) {
        try {
            Project project = new Project();
            project.setProjectName(request.getProjectName());
            project.setVersion(request.getVersion());

            InstallerSettings settings = new InstallerSettings();
            settings.setInstallPath(request.getInstallPath());

settings.setCreateDesktopShortcut(request.isCreateDesktopShortcut());
            settings.setLanguage(request.getLanguage());
            settings.setMainClass(request.getMainClass());
            settings.setRequirements(request.getRequirements());

            settings.setProject(project);
```

```
            project.setInstallerSettings(settings);

            Project savedProject = projectRepository.save(project);

            return ResponseEntity.ok().body("Project created successfully
with ID: " + savedProject.getId());

        } catch (Exception e) {
            return ResponseEntity.badRequest().body("Error creating
project: " + e.getMessage());
        }
    }
}
```

Клієнтська частина (External Client)

Клас **SimpleClient** демонструє програмну взаємодію з сервером через протокол HTTP. Він не використовує браузер, а напряму формує запит, що є ознакою machine-to-machine взаємодії.

```
package com.example.installer.client;

import java.io.OutputStream;
import java.net.HttpURLConnection;
import java.net.URL;
import java.nio.charset.StandardCharsets;

public class SimpleClient {
    public static void main(String[] args) {
        try {
            URL url = new URL("http://localhost:8080/api/create-project");
            HttpURLConnection conn = (HttpURLConnection)
url.openConnection();

            conn.setRequestMethod("POST");
            conn.setRequestProperty("Content-Type", "application/json; utf-
8");
            conn.setDoOutput(true);

            String jsonInputString = """
                {
                    "projectName": "DistributedApp",
                    "version": "1.0.0",
                    "installPath": "C:/Temp/Dist",
                    "createDesktopShortcut": true,
                    "language": "English",
                    "mainClass": "com.test.Main",
                    "requirements": "os windows AND java 17"
                }
                """;

            try (OutputStream os = conn.getOutputStream()) {
                byte[] input =
jsonInputString.getBytes(StandardCharsets.UTF_8);
                os.write(input, 0, input.length);
            }

            int responseCode = conn.getResponseCode();
            System.out.println("Response Code: " + responseCode);
```

```
            if (responseCode == 200) {
                System.out.println("Success! Project created via REST
API.");
            } else {
                System.out.println("Something went wrong.");
            }

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Клас SimpleClient розроблено з метою демонстрації програмної взаємодії (Machine-to-Machine) та перевірки універсальності розробленого REST API. Його наявність підтверджує, що серверна частина не залежить від конкретного типу клієнта і здатна обробляти запити від будь-яких зовнішніх систем (мобільних додатків, сторонніх сервісів, скриптів автоматизації), що є ключовою характеристикою розподіленої клієнт-серверної архітектури.

Висновки

В ході виконання лабораторної роботи було успішно модернізовано проект «Installer Generator», трансформувавши його з локального монолітного додатку в розподілену систему, побудовану на основі архітектури Client-Server.

Основні результати роботи:

1. **Реалізація архітектури Client-Server:** Замість використання специфічних для платформи .NET технологій (WCF, .NET Remoting), було обрано сучасний архітектурний стиль REST API, який є стандартом для Java-екосистеми. Це дозволило реалізувати серверну частину на базі Spring Boot, створивши точку доступу InstallerApiController, яка обробляє HTTP-запити та повертає відповіді у форматі JSON, забезпечуючи слабку зв'язність (Loose Coupling) між компонентами системи.

2. **Організація взаємодії через Middleware:** Для передачі даних між клієнтом і сервером було впроваджено шар Middleware у вигляді об'єктів DTO (Data Transfer Object). Клас ProjectRequestDto визначив чіткий контракт інтерфейсу, ізолювавши внутрішню структуру бази даних (Entity) від зовнішніх споживачів API. Це підвищило безпеку та гнучкість системи, дозволяючи змінювати внутрішню логіку сервера без порушення роботи клієнтів.

3. **Демонстрація Machine-to-Machine взаємодії:** Розроблено окремий клієнтський додаток SimpleClient, який емулює роботу зовнішньої автоматизованої системи. Його успішна взаємодія з сервером через протокол HTTP підтвердила, що система здатна працювати в розподіленому середовищі без участі користувача-людини, що є критично важливою вимогою для сучасних Enterprise-рішень.

Контрольні запитання

1. Що таке клієнт-серверна архітектура?

Це модель взаємодії в розподілених системах, де навантаження розділене між постачальниками послуг (серверами) та замовниками послуг (клієнтами). Клієнт ініціює запит, а сервер обробляє його, зберігає дані та надсилає відповідь.

2. Розкажіть про сервіс-орієнтовану архітектуру (SOA).

SOA - це архітектурний стиль, у якому програма будується з окремих, незалежних компонентів (сервісів), орієнтованих на бізнес-процеси. Ці сервіси взаємодіють через мережу, часто використовуючи корпоративну сервісну шину (ESB) для координації.

3. Якими принципами керується SOA?

Основні принципи:

- Повторне використання: Сервіси можна використовувати в різних додатках.
- Слабка зв'язність (Loose Coupling): Зміни в одному сервісі не повинні ламати інші.
- Стандартизований контракт: Чіткий опис інтерфейсу взаємодії.
- Абстракція: Клієнт знає лише інтерфейс, а не деталі реалізації.

4. Як між собою взаємодіють сервіси в SOA?

Вони обмінюються повідомленнями через мережу. Найчастіше використовується Enterprise Service Bus (ESB) - проміжне ПЗ, яке маршрутизує повідомлення, трансформує формати даних та забезпечує взаємодію між різнорідними сервісами.

5. Як розробники взнають про існуючі сервіси і як робити до них запити?

Інформація про сервіси зберігається в Реєстрі сервісів (Service Registry). Щоб зробити запит, розробник використовує Контракт сервісу (наприклад,

WSDL файл або Swagger/OpenAPI документацію), де описано методи, параметри та формати даних.

6. У чому полягають переваги та недоліки клієнт-серверної моделі?

- Переваги: Централізоване управління даними, простіше забезпечення безпеки, легше оновлювати логіку (тільки на сервері).
- Недоліки: Сервер є єдиною точкою відмови (якщо впав - не працює ніхто), ризик перевантаження сервера, залежність від мережі.

7. У чому полягають переваги та недоліки одноранговової моделі (P2P) взаємодії?

- Переваги: Висока відмовостійкість (немає центрального сервера), легка масштабованість (кожен вузол додає ресурси).
- Недоліки: Складність адміністрування та контролю, проблеми з безпекою, не гарантована швидкість пошуку даних.

8. Що таке мікросервісна архітектура?

Це підхід, при якому додаток розбивається на набір невеликих, повністю автономних сервісів, кожен з яких виконує одну бізнес-функцію, має власну базу даних і може бути розгорнутий незалежно від інших.

9. Які протоколи використовуються для обміну даними в мікросервісній архітектурі?

Використовуються легковажні протоколи:

- Синхронні: HTTP/REST, gRPC.
- Асинхронні: Протоколи черг повідомлень (AMQP) через брокери типу RabbitMQ або Kafka.

10. Чи можна назвати підхід сервіс-орієнтованою архітектурою, коли ми в проєкті між шаром веб-контролерів та шаром доступу до даних реалізуємо шар бізнес-логіки у вигляді сервісів?

Ні. Це називається Service Layer Pattern (шар сервісів) у рамках Монолітної архітектури. У SOA сервіси є фізично розділеними програмами, що

спілкуються через мережу, а у вашому випадку це просто Java-класи всередині одного процесу (JVM).