Міністерство освіти і науки України
Національний технічний університет України
“Київський політехнічний інститут імені Ігоря Сікорського”
Факультет інформатики та обчислювальної техніки
Кафедра інформаційні систем та технологій

Лабораторна робота №9

із дисципліни «*ТРПЗ*»

Тема: РІЗНІ ВИДИ ВЗАЄМОДІЇ ДОДАТКІВ: CLIENT-SERVER, PEER-TO-PEER, SERVICE-ORIENTED ARCHITECTURE

Виконав:
Студент групи ІА-24
Бакалець А.І.

Перевірив:
Мягкий М.Ю.

Київ-2024

Мета: Навчитися використовувати різні види взаємодії додатків: client-server, peer-to-peer, service-oriented-architecture

Зміст

Завдання.

1. Ознайомитися з короткими теоретичними відомостями.
2. Реалізувати частину функціоналу робочої програми у вигляді класів і їх взаємодій для досягнення конкретних функціональних можливостей.
3. Реалізувати взаємодію програми в одній з архітектур відповідно до обраної теми.

Тема:

..1 Музичний програвач (iterator, command, memento, facade, visitor, client-server)

Музичний програвач становить собою програму для програвання музичних файлів або відтворення потокової музики з можливістю створення, запам'ятовування і редагування списків програвання, перемішування/повторення (shuffle/repeat), розпізнавання різних аудіо-форматів, еквалайзер.

Github: https://github.com/Clasher3000/TRPZ_labs

Теоретичні відомості

Client-Server

Шаблон "Client-Server" визначає взаємодію між двома типами компонентів: клієнтом, який робить запити, і сервером, який їх обробляє та повертає результати. Використовується для побудови розподілених систем, де клієнт взаємодіє з сервером через мережу. У Java цей шаблон реалізується за допомогою серверного додатка (наприклад, з використанням Java Servlet API чи Spring Boot) і клієнтської програми

(через HTTP-клієнти, такі як Apache HttpClient чи RESTTemplate). Сервер обробляє запити клієнта, забезпечуючи централізоване управління даними, а клієнт отримує доступ до функціоналу через чітко визначений API.

Peer-to-Peer (P2P)

Шаблон "Peer-to-Peer" забезпечує децентралізовану модель, де кожен вузол може діяти як клієнт і як сервер. Використовується для створення мереж, де учасники обмінюються ресурсами без центрального вузла. У Java цей шаблон реалізується через мережеве програмування (наприклад, з використанням java.net для сокетів або бібліотек для P2P-мереж, таких як JXTA). Кожен вузол має механізми для з'єднання з іншими вузлами, надсилання запитів і відповіді на них. Завдяки цьому система стає стійкою до відмови окремих вузлів і легко масштабується.

Service-Oriented Architecture (SOA)

Шаблон "Service-Oriented Architecture" визначає побудову системи з незалежних сервісів, які взаємодіють через стандартизовані протоколи. Використовується для проектування модульних систем, де кожен сервіс відповідає за певну бізнес-логіку. У Java цей шаблон реалізується через технології, такі як SOAP (з використанням JAX-WS) або RESTful сервіси (з використанням Spring REST). Сервіси спілкуються через API, що дозволяє інтегрувати їх незалежно від платформи або мови програмування. Такий підхід забезпечує повторне використання компонентів і спрощує їх модифікацію чи заміну.

.

Хід роботи

У цій лабораторній роботі я реалізував модель **клієнт-сервер (Client-Server),** яка дозволила організувати взаємодію між двома компонентами програми: клієнтом і сервером. Сервер виконує роль центрального елемента, який зберігає дані, обробляє запити клієнта і повертає відповіді, тоді як клієнт відповідає за ініціалізацію запитів і обробку отриманих результатів.

Взаємодія між ними реалізована через сокети, де сервер використовує ServerSocket для очікування підключень, а клієнт підключається за допомогою Socket. Сервер відповідає за зберігання та управління треками і плейлистами, надаючи клієнту доступ до цих даних через мережу. Клієнт надсилає текстові команди, такі як запити на отримання списку треків або плейлистів, а сервер обробляє ці запити та повертає відповідь у вигляді даних.

Наприклад, клієнт може запросити список треків за допомогою команди "find_all", і сервер поверне відповідний список. Аналогічно, клієнт може запитати список Реалізація забезпечила централізоване управління даними, що уникнуло їх дублювання та забезпечило цілісність, а також дозволила легко додавати нові функції, наприклад, команди для створення нового плейлиста чи додавання треку. Така архітектура зробила програму гнучкою та масштабованою, дозволивши запускати клієнт і сервер на різних пристроях, що забезпечило розподілену роботу та можливість легко змінювати клієнтську частину без змін у сервері.

Структура Client-Server

```
public class Server {

    14 usages
    private static MusicPlayer musicPlayer;

    Clasher3000
    public static void main(String[] args) throws IOException {
        ServerSocket ss = new ServerSocket( port: 4999);
        System.out.println("Server is waiting for connection...");

        Socket s = ss.accept();
        System.out.println("Client connected.");


        BufferedReader in = new BufferedReader(new InputStreamReader(s.getInputStream()));
        PrintWriter out = new PrintWriter(s.getOutputStream(), autoFlush: true);

        PlaylistService playlistService = new PlayListServiceImpl(out);
        TrackService trackService = new TrackServiceImpl(out);

        musicPlayer = new MusicPlayer(out,trackService,playlistService);

        String clientMessage;
        while ((clientMessage = in.readLine()) != null) {
            System.out.println("Received from client: " + clientMessage);
```

Рис.1 – Сервер та алгоритм з’єднання з клієнтом

Цей код створює серверний сокет на порту 4999, який очікує підключення клієнта. Після встановлення з'єднання клієнтський сокет обробляється через потоки вводу/виводу. Ініціалізуються сервіси для роботи з плейлистами та треками, а також створюється об'єкт MusicPlayer, який використовує ці сервіси для взаємодії з клієнтом.

```
String clientMessage;
while ((clientMessage = in.readLine()) != null) {
    System.out.println("Received from client: " + clientMessage);

    String[] parts = clientMessage.split( regex: " ",  limit: 2); // Розділити команду та аргументи
    String command = parts[0].toLowerCase();
    String myArgs = parts.length > 1 ? parts[1] : "";

    // Розділяємо аргументи за комами
    String[] arguments = myArgs.split( regex: ","); // args[0] - назва треку, args[1] - шлях до файлу

    try {
        switch (command) {
            case "help":
                Command helpCommand = new HelpCommand(out);
                helpCommand.execute();
                break;
            case "play":
                Command playCommand = new PlayCommand(musicPlayer, arguments[0], out);
                playCommand.execute();
                break;
            case "pause":
                Command pauseCommand = new PauseCommand(musicPlayer, out);
                pauseCommand.execute();
                break;
```

Рис.2 – Логіка роботи з повідомленнями отриманих з клієнту

Цей код обробляє повідомлення, отримані від клієнта, і виконує відповідні команди. Повідомлення розбивається на команду та її аргументи, які обробляються за допомогою конструкції switch. Залежно від команди, створюється відповідний об'єкт команди (наприклад, HelpCommand, PlayCommand, PauseCommand, ResumeCommand), який викликає метод execute() для виконання дії. Таким чином, сервер реагує на запити клієнта, дозволяючи керувати музичним плеєром.

```
                }
                catch (ResourceNotFoundException | ResourceAlreadyExistsException e) {
                    out.println(e.getMessage());
                }
                catch (Exception e) {
                    out.println("Something went wrong: " +e.getClass() + e.getMessage());
                }
            }
        }
    }
```

Рис.3 – Логіка перехоплення винятків на сервері

Цей код обробляє виключення, які можуть виникнути під час виконання команд. Якщо виникає ResourceNotFoundException або ResourceAlreadyExistsException, клієнту надсилається повідомлення з текстом виключення. У випадку інших виключень клієнту надсилається повідомлення про помилку із зазначенням типу виключення та його опису. Це забезпечує зворотний зв'язок для клієнта у разі помилок.

```
public class Client {

    Clasher3000
    public static void main(String[] args) throws IOException, InterruptedException {
        // Підключення до сервера
        Socket socket = new Socket( host: "localhost",  port: 4999);

        // Створюємо потоки для надсилання та отримання даних
        PrintWriter out = new PrintWriter(socket.getOutputStream(),  autoFlush: true);  // Для надс
        BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));

        // Запускаємо окремий потік для отримання повідомлень від сервера
        Thread receiveThread = new Thread(() -> {
            try {
                String serverMessage;
                while ((serverMessage = in.readLine()) != null) {  // Читання відповіді від сер
                    System.out.println("Server: " + serverMessage);
                }
            } catch (IOException e) {
                System.out.println("Connection closed.");
            }
        });
        receiveThread.start();
```

Рис.4 – Клієнт та логіка зчитування повідомлень з сервера

Цей код реалізує клієнтську програму, яка підключається до сервера через сокет на порту 4999. Для взаємодії із сервером створюються два потоки: один для відправлення даних (PrintWriter) і один для отримання даних (BufferedReader). Щоб обробляти відповіді від сервера асинхронно, запускається окремий потік. У цьому потоці клієнт безперервно читає повідомлення, що надходять від сервера, та виводить їх у консоль. Якщо зв'язок із сервером переривається, у консоль виводиться повідомлення про закриття з'єднання.

```
        Scanner scanner = new Scanner(System.in);

        // Цикл для відправки команд на сервер
        while (true) {
            System.out.print("Enter command: ");
            String command = scanner.nextLine();  // Читання ко

            // Відправка команди на сервер
            out.println(command);

            if(command.equals("find_all") ) {
                // Чекаємо на відповідь від сервера перед тим,
                String serverMessage = in.readLine();  // Читає
                System.out.println("Server: " + serverMessage);
            }
            // Перевірка на команду виходу
            if (command.equalsIgnoreCase( anotherString: "exit")) {
                break;  // Завершуємо програму
            }

            Thread.sleep( millis: 500);  // Додаємо невелику затри
        }
        // Закриття ресурсів
        out.close();
        in.close();
        socket.close();
    }
```

Рис.5 – Логіка відправки команд

Користувач вводить команду через консоль, яка надсилається серверу через out.println(command). Якщо команда "find_all", клієнт чекає на відповідь від сервера, зчитує її через in.readLine() та виводить у консоль. У випадку команди "exit" програма завершується, виходячи з циклу за допомогою break. Для плавності між відправленнями команд додається затримка у 500 мс за допомогою Thread.sleep(500). Після завершення циклу закриваються всі ресурси: потоки вводу/виводу та сокет.

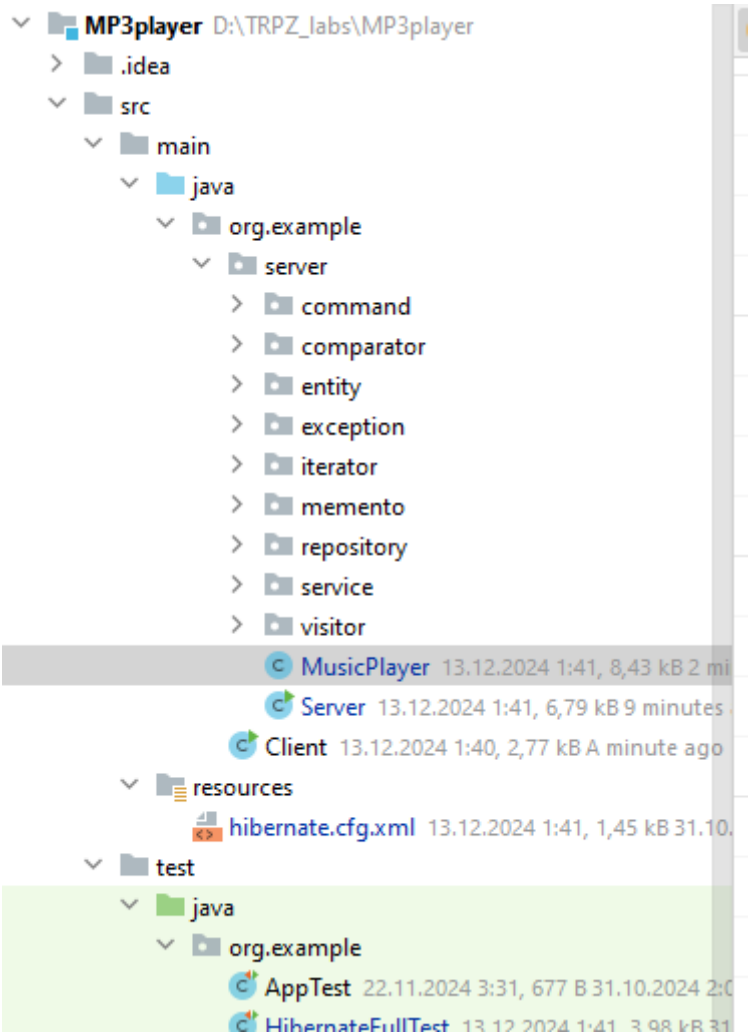

Рис.6 – Структура компонентів, при побудові архітектури клієнт-сервер

Демонстрація виконання

```
Enter command: play Lady
Enter command: Server: Playing: Lady
stop
Server: Track stopped
Enter command: find_all
Server: Get Lucky - Track
Server: Lady - Track
Server: Corona - Track
```

Рис.7 – Демонстрація з’єднання між сервером і клієнтом

На зображенні показано консольний вивід клієнтської програми, яка взаємодіє з сервером. Клієнт вводить різні команди, такі як "play Lady", "stop" і "find_all". Кожен введений запит відображається на екрані разом з відповідями від сервера. Сервер підтверджує виконання команд, наприклад, для команди "play Lady" сервер виводить

"Playing: Lady", а для команди "stop" — "Track stopped". Команда "find_all" повертає список доступних треків та списків відтворення.

Висновок:

У процесі виконання лабораторної роботи я ознайомився з принципами роботи сервер-клієнт взаємодії за допомогою сокетів. Я реалізував сервер, який приймає підключення від клієнта, обробляє команди, як-от запуск і зупинка треків, а також здійснює пошук всіх доступних треків. Клієнтська програма надсилає команди серверу та отримує відповідь, забезпечуючи двосторонню взаємодію через мережу.

Особливу увагу я приділив правильній організації обміну даними між сервером і клієнтом. Завдяки потокам для читання і запису, я забезпечив асинхронну обробку повідомлень, що підвищує ефективність і зручність використання програми. Цей підхід дозволив реалізувати зручну та масштабовану систему для керування треками, що значно полегшує додавання нових функцій у майбутньому.