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 «*Технології розроблення програмного забезпечення*» **Тема:** «Різні види взаємодії додатків: client-server, peer-to-peer, service-oriented architecture»

Виконав:
Студент групи ІА-33
Панащук Р. А.

Перевірив:
Мягкий М. Ю.

Київ-2025

ЗМІСТ

Мета: дізнатися, вивчити та навчитися використовувати різні види взаємодії додатків client-server, peer-to-peer, service-oriented architecture.

Тема: Powershell terminal (strategy, command, abstract factory, bridge, interpreter, client-server).
Термінал для powershell повинен нагадувати типовий термінал з можливістю налаштування кольорів синтаксичних конструкцій, розміру вікна, фону вікна, а також виконання команд powershell і виконуваних файлів, а також працювати в декількох вікнах терміналу (у вкладках або одночасно шляхом розділення вікна).

Короткі теоретичні відомості

Client-Server: Модель взаємодії, де клієнт ініціює запити до сервера, а сервер обробляє ці запити та повертає результати. Сервер забезпечує централізоване зберігання та управління даними, тоді як клієнт виступає інтерфейсом для користувача. Така архітектура дозволяє розподілити обчислювальні ресурси між клієнтом і сервером. Основні компоненти включають клієнт (ініціатор запиту), сервер (обробник запиту) і мережевий протокол (наприклад, HTTP, TCP/IP).

Peer-to-Peer (P2P): Децентралізована модель, у якій усі вузли (peer) виступають рівноправними учасниками, виконуючи роль і клієнта, і сервера одночасно. Кожен вузол може надавати ресурси іншим і отримувати їх. P2P підходить для сценаріїв, де важлива автономність і відсутність центрального сервера. Основні елементи включають вузли (peer), які виконують одночасно функції запитувача і постачальника ресурсів, та мережу, що забезпечує зв'язок між ними.

Service-Oriented Architecture (SOA): Архітектурний стиль, у якому програми будуються як набір взаємодіючих сервісів. Кожен сервіс є автономною функціональною одиницею, яка має чітко визначений інтерфейс і взаємодіє з іншими сервісами через стандартизовані протоколи (наприклад, SOAP, REST). SOA забезпечує гнучкість, масштабованість і повторне використання компонентів. Основні компоненти включають сервіси (незалежні модулі),

сервісний реєстр (для пошуку сервісів), контракт (опис API сервісу) і транспорт (засоби передачі даних між сервісами).

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.

Хід роботи

Вид взааємодії client-server

Я обрав клієнт-серверну архітектуру для свого проєкту PowerShell терміналу, тому що вона дозволяє розподілити обробку даних між клієнтом (JavaFX) і сервером (Spring Boot). Це забезпечує такі переваги:

- Масштабованість: Сервер може обробляти численні запити від клієнтів, що полегшує розширення функціональності.
- Централізоване управління: Сервер зберігає дані (PostgreSQL) і виконує бізнес-логіку, тоді як клієнт відповідає за взаємодію з користувачем.
- Гнучкість: Використання REST API дозволяє легко інтегрувати нові компоненти або клієнтські додатки.
- Безпека: Вся логіка виконання команд і маніпуляцій з даними знаходиться на сервері, що мінімізує ризик несанкціонованого доступу.

```java
@RestController
@RequestMapping("/commands")
public class CommandController {

    private final TabService tabService;

    public CommandController(TabService tabService) {
        this.tabService = tabService;
    }

    @PostMapping("/execute")
    public ResponseEntity<Map<String, String>> executeCommand(@RequestBody Map<String, String> request) {
        String commandText = request.get("name");
        Long tabId = Long.valueOf(request.get("tabId").toString());

        Tab tab = tabService.findTabById(tabId);
        if (tab == null) {
            return ResponseEntity.status(HttpStatus.NOT_FOUND).body(Map.of( k1: "output",  v1: "Tab not found"));
        }

        ApplyThemeExpression themeExpression = new ApplyThemeExpression(commandText, tabId, tabService);
        CommandTypeExpression typeExpression = new CommandTypeExpression(new ContextStrategy(), tabService);

        String themeResult = themeExpression.interpret(commandText, tab);
        if (!themeResult.equals("Unknown command")) {
            return ResponseEntity.ok(Map.of( k1: "output", themeResult));
        }


        String result = typeExpression.interpret(commandText, tab);
        return ResponseEntity.ok(Map.of( k1: "output", result,  k2: "currentDirectory", tab.getCurrentDirectory()));
    }
}
```

Рис. 1 – Клас CommandController

Цей контролер відповідає за обробку команд, що надходять від клієнта, та їх виконання на сервері. Метод executeCommand приймає команду та ідентифікатор вкладки, використовує патерн Interpreter для визначення типу команди та виконання, повертає результат виконання команди або повідомлення про помилку.

```java
@RestController
@RequestMapping("/settings")
public class SettingsController {

    private final TabService tabService;
    private final SettingsService settingsService;
    private final CommandInvoker commandInvoker;

    public SettingsController(TabService tabService, SettingsService settingsService) {
        this.tabService = tabService;
        this.settingsService = settingsService;
        this.commandInvoker = new CommandInvoker();

    }

    @PostMapping("/change-text-color")
    public ResponseEntity<String> changeTextColor(@RequestBody Map<String, Object> request) {
        Long tabId = Long.valueOf(request.get("tabId").toString());
        String newColor = request.get("newColor").toString();
        Command changeColorCommand = new ChangeColorTextCommand(settingsService, tabId, newColor);
        commandInvoker.setCommand(changeColorCommand);
        commandInvoker.executeCommand();
        return ResponseEntity.ok( body: "Text color changed to: " + newColor);
    }

    @PostMapping("/change-background-color")
    public ResponseEntity<String> changeBackgroundColor(@RequestBody Map<String, Object> request) {
        Long tabId = Long.valueOf(request.get("tabId").toString());
        String newBackgroundColor = request.get("newBackgroundColor").toString();
        Command changeBackgroundColorCommand = new ChangeColorBackgroundCommand(settingsService, tabId, newBackgroundColor);
        commandInvoker.setCommand(changeBackgroundColorCommand);
        commandInvoker.executeCommand();
        return ResponseEntity.ok( body: "Background color changed to: " + newBackgroundColor);
    }

    @GetMapping("/get-styles")
    public ResponseEntity<Map<String, String>> getStyles(@RequestParam Long tabId) {
        Tab tab = tabService.findTabById(tabId);
        if (tab == null) {
            return ResponseEntity.notFound().build();
        }
        return ResponseEntity.ok(Map.of(
                k1: "background", tab.getWindowBackground(),
                k2: "textColor", tab.getSyntaxColor()
        ));
    }
}
```

Рис. 2 – Клас SettingsController

Цей контролер керує налаштуваннями вкладок, включаючи зміну стилів. Метод changeTextColor змінює колір тексту в обраній вкладці, використовуючи патерн Command. Метод changeBackgroundColor аналогічно змінює колір фону вкладки. Метод getStyles повертає поточні стилі вкладки, включаючи кольори тексту та фону.

```java
@RestController
@RequestMapping("/tabs")
public class TabController {

    private final TabService tabService;
    private final CommandInvoker commandInvoker;

    public TabController(TabService tabService) {
        this.tabService = tabService;
        this.commandInvoker = new CommandInvoker();

    }
    @PostMapping("/create")
    public ResponseEntity<Tab> createTab() {
        Command createTabCommand = new CreateTabCommand(tabService);
        commandInvoker.setCommand(createTabCommand);
        Tab createdTab = commandInvoker.executeAndReturn();
        return ResponseEntity.ok(createdTab);
    }

    @PostMapping("/close")
    public ResponseEntity<String> closeTab(@RequestBody Map<String, Long> request) {
        Long tabId = request.get("tabId");
        Command closeTabCommand = new CloseTabCommand(tabService, tabId);
        commandInvoker.setCommand(closeTabCommand);
        commandInvoker.executeCommand();
        return ResponseEntity.ok( body: "Tab closed: " + tabId);
    }

    @PostMapping("/close-all")
    public ResponseEntity<String> closeAllTabs() {
        tabService.closeAllTabs();
        return ResponseEntity.ok( body: "All tabs closed.");
    }

    @GetMapping("/get-directory")
    public ResponseEntity<String> getCurrentDirectory(@RequestParam Long tabId) {
        Tab tab = tabService.findTabById(tabId);
        return ResponseEntity.ok(tab.getCurrentDirectory());
    }
}
```

Рис. 3 – Клас TabController

Цей контролер забезпечує створення, закриття та управління вкладками. Метод createTab використовує команду для створення нової вкладки, яка зберігається на сервері. Метод closeTab закриває вказану вкладку. Метод closeAllTabs закриває всі відкриті вкладки. Метод getCurrentDirectory повертає поточну робочу директорію для зазначеної вкладки.

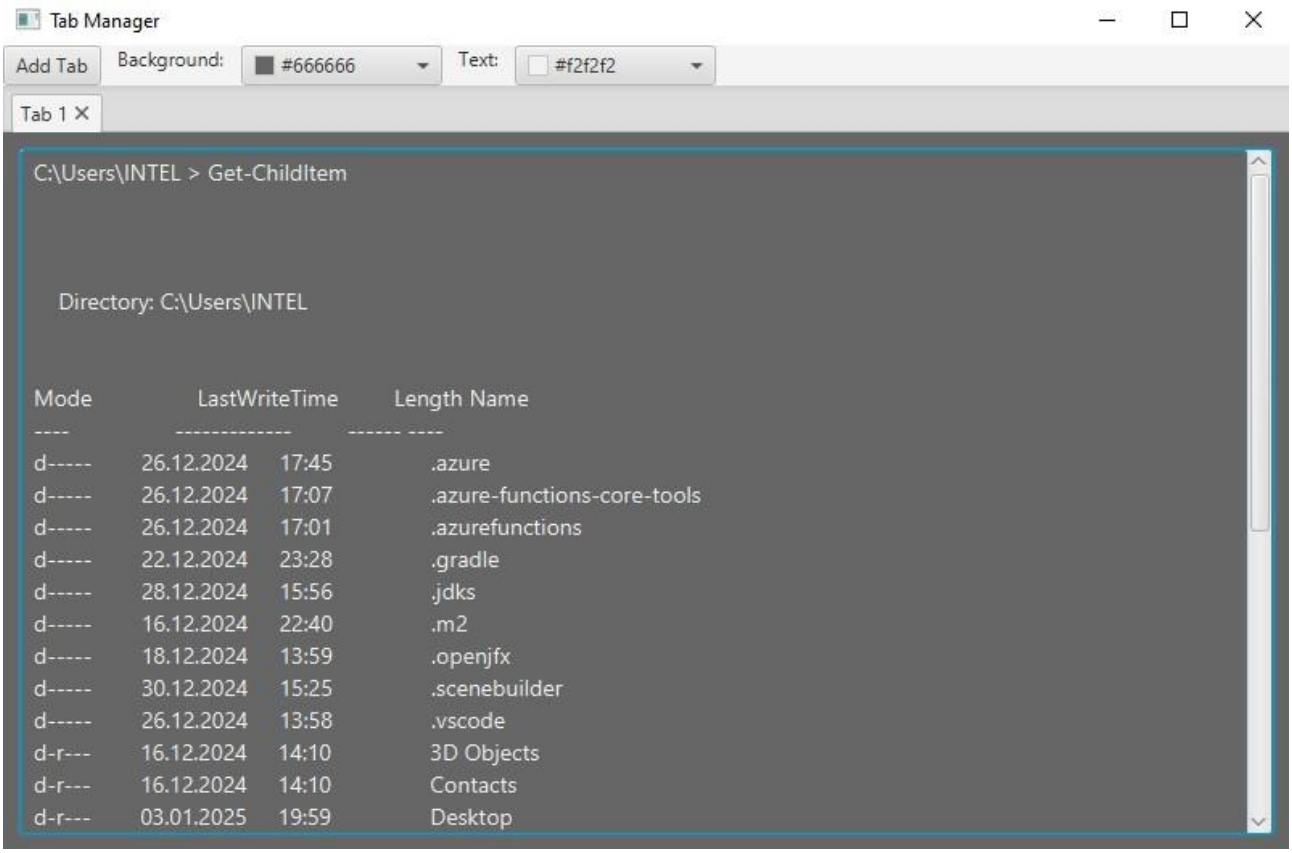

Рис. 4 – Клієнт терміналу

На рисунку 4 показна клієнт терміналу, який відправляє команди для виконання на сервер. Створюється client-server архітектура, де клієнт створений на JavaFX, а сервер на Spring Boot.

Висновки: Під час виконання даної лабораторної роботи я дізнався, вивчив та навчився використовувати різні види взаємодії додатків: client-server, peer-topeer, service-oriented architecture. Для реалізації свого проєкту я обрав клієнтсерверну архітектуру, яка дозволяє централізувати управління даними на сервері та забезпечує зручну взаємодію між клієнтом і сервером через REST API. Я реалізував серверну частину додатку, використовуючи Spring Boot, і створив функціональність для роботи з вкладками, зміни стилів та виконання команд. Обрана архітектура продемонструвала високу модульність, простоту розширення функціональності та ефективність у роботі з даними, розподіленими між клієнтом і сервером.