



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 №5

із дисципліни *«Технології розроблення програмного забезпечення»*

Тема: «ШАБЛОНИ «ADAPTER», «BUILDER», «COMMAND», «CHAIN OF RESPONSIBILITY», «PROTOTYPE»»

Виконав:
Студент групи ІА-24
Котлярчук М. С.

Перевірів:
Мягкий М. Ю.

Київ-2024

ЗМІСТ

Короткі теоретичні відомості.....	3
Завдання.....	4
Хід роботи	4
Шаблон Command	4
Висновки... ..	10

Мета: дізнатися, вивчити та навчитися використовувати шаблони проектування adapter, builder, command, chain of responsibility, prototype.

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

Adapter — це структурний шаблон, який дозволяє об'єднати несумісні інтерфейси, перетворюючи інтерфейс одного класу в інтерфейс, який очікує клієнт. Він часто використовується для інтеграції старого коду або сторонніх бібліотек у нові системи. Основними елементами є клієнт, адаптер та адаптований об'єкт. Адаптер забезпечує перетворення запитів клієнта у виклики методів адаптованого об'єкта.

Builder — це шаблон проектування, який дозволяє створювати складні об'єкти крок за кроком, розділяючи процес побудови на незалежні етапи. Він застосовується, коли об'єкт має багато параметрів або варіантів конфігурації. Основними елементами є будівельник, що відповідає за поетапне створення, і директор, який керує цим процесом. Шаблон спрощує створення об'єктів зі складною структурою без перевантаження конструктора.

Command — це поведінковий шаблон, який інкапсулює запити у вигляді об'єктів, дозволяючи зберігати, передавати і виконувати їх незалежно від клієнта. Він часто використовується для організації черги запитів, відкладеного виконання або функціональності "Undo". Основними елементами є відправник, отримувач, і об'єкт-команда, який містить всю необхідну інформацію для виконання дії.

Chain of Responsibility — це поведінковий шаблон, який передає запит через ланцюг об'єктів, поки один із них не обробить його. Цей шаблон використовується для зменшення зв'язності між об'єктами та забезпечення гнучкого додавання нових обробників у ланцюг. Основними елементами є клієнт, що ініціює запит, і об'єкти-обробники, які послідовно перевіряють можливість виконання запиту.

Prototype — це шаблон проектування, який дозволяє створювати нові об'єкти через копіювання вже існуючих. Це корисно, коли створення об'єкта є складним або ресурсозатратним. Основними елементами є інтерфейс або базовий клас з методом clone(), який реалізують конкретні класи. Шаблон

зручний для створення великої кількості схожих об'єктів із можливістю модифікації їх властивостей після копіювання.

Завдання

1. Ознайомитися з короткими теоретичними відомостями.
2. Реалізувати частину функціоналу робочої програми у вигляді класів та їхньої взаємодії для досягнення конкретних функціональних можливостей.
3. Застосування одного з розглянутих шаблонів при реалізації програми.

Хід роботи

Шаблон Command

Для цієї лабораторної роботи я обрав реалізувати шаблон Command. Реалізація цього шаблону дозволить інкапсулювати різні дії, у вигляді окремих об'єктів-команд. Це спрощує додавання нових операцій, знижує залежність між компонентами та полегшує впровадження додаткових функціональностей. У межах лабораторної роботи я реалізував наступні команди: створення вкладки, закриття вкладки, зміна розміру тексту, зміна кольору тексту та зміна фону. Нижче наведено ключові класи, що використовуються у реалізації шаблону Command:

```
package com.example.terminal_powershell.command;  
  
public interface Command { 5 implementations  
    void execute(); 5 implementations  
}
```

Рис. 1 – Інтерфейс Command

Інтерфейс Command є основою шаблону, що визначає метод execute(). Цей метод реалізується всіма командами для виконання операцій.

```

public class CreateTabCommand implements Command { 1 usage
    private TabService tabService; 2 usages
    private String tabName; 2 usages

    public CreateTabCommand(TabService tabService, String tabName) {
        this.tabService = tabService;
        this.tabName = tabName;
    }

    public void execute() {
        tabService.createTab(tabName);
    }
}

```

Рис. 2 – Реалізація команди створення нової вкладки

Ця команда відповідає за створення нової вкладки у терміналі. Вона містить всю необхідну логіку для додавання вкладки з вказаним іменем. Використання такої команди дозволяє легко інтегрувати функціонал створення вкладок у загальну архітектуру програми, а також робить систему більш масштабованою та зручною для тестування.

```

public class CloseTabCommand implements Command { 1 usage
    private TabService tabService; 2 usages
    private Long tabId; 2 usages

    public CloseTabCommand(TabService tabService, Long tabId) {
        this.tabService = tabService;
        this.tabId = tabId;
    }

    public void execute() {
        tabService.closeTab(tabId);
    }
}

```

Рис. 3 – Реалізація команди закриття існуючої вкладки

Команда для закриття вкладки у терміналі. Її реалізація включає видалення вкладки за унікальним ідентифікатором, забезпечуючи контроль над операціями закриття.

```

public class ChangeSizeTextCommand implements Command { 1 usage
    private TabService tabService; 2 usages
    private Long tabId; 2 usages
    private int newSize; 2 usages

    public ChangeSizeTextCommand(TabService tabService, Long tabId, int newSize) {
        this.tabService = tabService;
        this.tabId = tabId;
        this.newSize = newSize;
    }

    public void execute() {
        tabService.changeTextSize(tabId, newSize);
    }
}

```

Рис. 4 – Реалізація команди зміни розміру шрифту

Ця команда забезпечує зміну розміру тексту в межах певної вкладки. Вона дозволяє задати новий розмір, який потім застосовується до вибраного елемента. Такий підхід сприяє централізації управління властивостями синтаксичних конструкцій і дозволяє уніфікувати механізми їх зміни для різних вкладок.

```

public class ChangeColorTextCommand implements Command { 1 usage
    private TabService tabService; 2 usages
    private Long tabId; 2 usages
    private String newColor; 2 usages

    public ChangeColorTextCommand(TabService tabService, Long tabId, String newColor) {
        this.tabService = tabService;
        this.tabId = tabId;
        this.newColor = newColor;
    }

    public void execute() {
        tabService.changeTextColor(tabId, newColor);
    }
}

```

Рис. 5 – Реалізація команди зміни кольору синтаксичних конструкцій

Команда, яка реалізує зміну кольору тексту у вибраній вкладці. Вона дозволяє задати новий колір, що застосовується до текстового контенту. Використання цієї команди робить систему більш гнучкою і дозволяє легко адаптувати функціонал.

```

public class ChangeColorBackgroundCommand implements Command { 1 usage
    private TabService tabService; 2 usages
    private Long tabId; 2 usages
    private String newBackgroundColor; 2 usages

    public ChangeColorBackgroundCommand(TabService tabService, Long tabId, String newBackgroundColor) {
        this.tabService = tabService;
        this.tabId = tabId;
        this.newBackgroundColor = newBackgroundColor;
    }

    public void execute() {
        tabService.changeBackgroundColor(tabId, newBackgroundColor);
    }
}

```

Рис. 6 – Реалізація команди зміни фону вкладки

Ця команда дозволяє змінювати фон вкладки, задаючи новий колір. Такий підхід забезпечує модульність і спрощує розширення функціональності.

```

@Service 18 usages
public class TabService {

    private final TabRepository tabRepository; 9 usages

    public TabService(TabRepository tabRepository) { this.tabRepository = tabRepository; }

    public void createTab(String tabName) { 1 usage
        Tab newTab = new Tab(tabName);
        newTab.setTitle(tabName);
        tabRepository.save(newTab);
        System.out.println("Tab created: " + tabName);
    }

    public void closeTab(Long tabId) { 1 usage
        tabRepository.deleteById(tabId);
        System.out.println("Tab closed: " + tabId);
    }

    public void changeTextSize(Long tabId, int newSize) { 1 usage
        Tab tab = tabRepository.findById(tabId).orElseThrow(() -> new IllegalArgumentException("Tab not found"));
        tab.setTextSize(newSize);
        tabRepository.save(tab);
        System.out.println("Text size for tab " + tabId + " changed to " + newSize);
    }
}

```

Рис. 7 – Реалізація сервісу, що виконує основну бізнес-логіку

```

public void changeTextColor(Long tabId, String newColor) { 1 usage
    Tab tab = tabRepository.findById(tabId).orElseThrow(() -> new IllegalArgumentException("Tab not found"));
    tab.setSyntaxColor(newColor);
    tabRepository.save(tab);
    System.out.println("Text color for tab " + tabId + " changed to " + newColor);
}

public void changeBackgroundColor(Long tabId, String newBackgroundColor) { 1 usage
    Tab tab = tabRepository.findById(tabId).orElseThrow(() -> new IllegalArgumentException("Tab not found"));
    tab.setWindowBackground(newBackgroundColor);
    tabRepository.save(tab);
    System.out.println("Background color for tab " + tabId + " changed to " + newBackgroundColor);
}
}

```

Рис. 8 – Продовження Рис. 7

TabService є "отримувачем" (receiver) у шаблоні. Він виконує основну бізнес-логіку, наприклад, створення, закриття вкладок або зміну їх властивостей.

```

public class CommandInvoker { 2 usages
    private Command command; 3 usages

    public void setCommand(Command command) {
        this.command = command;
    }

    public void executeCommand() { 5 usages
        if (command != null) {
            command.execute();
        } else {
            throw new IllegalStateException("No command set");
        }
    }
}

```

Рис. 9 – Реалізація ініціатора виконання команд

Клас виконує роль ініціатора виконання команд. Він відповідає за збереження посилання на команду та її виконання через метод executeCommand(). Цей механізм дозволяє інкапсулювати виклик команд у одному місці, забезпечуючи гнучкість і простоту інтеграції команд у систему.


```

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

    private final TabService tabService; 6 usages
    private final CommandInvoker commandInvoker; 11 usages

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

    @PostMapping("/create")
    public ResponseEntity<String> createTab(@RequestBody Map<String, String> request) {
        String tabName = request.get("tabName");
        Command createTabCommand = new CreateTabCommand(tabService, tabName);
        commandInvoker.setCommand(createTabCommand);
        commandInvoker.executeCommand();
        return ResponseEntity.ok( body: "Tab created: " + tabName);
    }

    @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("/change-text-size")
    public ResponseEntity<String> changeTextSize(@RequestBody Map<String, Object> request) {
        Long tabId = Long.valueOf(request.get("tabId").toString());
        int newSize = Integer.valueOf(request.get("newSize").toString());
        Command changeSizeCommand = new ChangeSizeTextCommand(tabService, tabId, newSize);
        commandInvoker.setCommand(changeSizeCommand);
        commandInvoker.executeCommand();
        return ResponseEntity.ok( body: "Text size changed to: " + newSize);
    }

    @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(tabService, tabId, newBackgroundColor);
        commandInvoker.setCommand(changeBackgroundColorCommand);
        commandInvoker.executeCommand();
        return ResponseEntity.ok( body: "Background color changed to: " + newBackgroundColor);
    }
}

```

Рис. 10 – Реалізація контролера (клієнта у шаблоні)

Цей клас виступає посередником між зовнішніми запитами та командним механізмом. Він отримує запити, ініціалізує відповідні команди, передає їх `CommandInvoker` і виконує. Такий підхід забезпечує зрозумілий та упорядкований процес управління діями, що знижує складність інтеграції з користувацьким інтерфейсом.

Висновки Під час виконання даної лабораторної роботи я дізнався та вивчив шаблони «ADAPTER», «BUILDER», «COMMAND», «CHAIN OF RESPONSIBILITY», «PROTOTYPE». У процесі виконання цієї лабораторної роботи було реалізовано шаблон проектування `Command`, який дозволяє ефективно організувати виконання різних операцій в терміналі через інтерфейс команд. За допомогою цього шаблону вдалося створити систему, де кожна операція, така як створення, закриття вкладок, зміна розміру шрифтів та кольорів, представлена окремою командою класом.

Використання шаблону `Command` в цьому проекті дозволило не лише організувати виконання команд, але й зробити програму більш гнучкою, тестованою та легкою для розширення в майбутньому, забезпечуючи підтримку нових функціональностей без значних змін у вже існуючому коді.