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

Лабораторна робота №5
Технологія розроблення програмного забезпечення
«ADAPTER», «BUILDER», «COMMAND», «CHAIN OF RESPONSIBILITY», «PROTOTYPE»

Варіант 8

Виконав
студент групи ІА-13
Крутиус Владислав Віталійович

Перевірив:

Київ 2023р.

Мета: Дослідити шаблони ШАБЛОНИ «ADAPTER», «BUILDER», «COMMAND», «CHAIN OF RESPONSIBILITY», «PROTOTYPE»

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.

1. ШАБЛОН «ADAPTER»

Шаблон "ADAPTER" є одним з базових шаблонів проектування в програмній інженері. Його основна мета - дозволити існуючому інтерфейсу працювати з новим, несумісним інтерфейсом без модифікації вихідного коду.

Основні характеристики та особливості шаблону "ADAPTER":

- Цільовий інтерфейс (Target Interface): Визначає той інтерфейс, який відомий та очікується від клієнта.
- Адаптер (Adapter): Клас, який реалізує цільовий інтерфейс та містить посилання на адаптований об'єкт.
- Адаптований об'єкт (Adaptee): Об'єкт, чий інтерфейс несумісний з цільовим, і який потрібно адаптувати.
- Адаптація (Adaptation): Процес створення адаптера, який вирішує проблеми несумісності між інтерфейсами.
- Мост між інтерфейсами (Bridge between Interfaces): Адаптер служить мостом між цільовим інтерфейсом та адаптованим об'єктом.
- Конвертація запитів (Request Conversion): Адаптер перетворює запити від цільового інтерфейсу на запити, зрозумілі адаптованому об'єкту.

Зберігання адаптованого об'єкта (Holding the Adaptee): Адаптер має посилання на адаптований об'єкт для того, щоб можна було викликати його методи.

Шаблон "ADAPTER" дозволяє використовувати існуючий код та класи, які мають різний інтерфейс, в новому контексті без потреби їх змінювати. Він особливо корисний при роботі зі сторонніми бібліотеками, інтерфейси яких потрібно адаптувати під потреби програми.

2. ШАБЛОН «BUILDER»

Шаблон "BUILDER" є одним із базових шаблонів проектування, який використовується для конструювання складних об'єктів крок за кроком. Він дозволяє відокремити процес конструювання об'єкта від його представлення.

Основні характеристики та особливості шаблону "BUILDER":

- Директор (Director): Відповідає за взаємодію з будівельником для конструювання об'єкта. Він визначає послідовність кроків для створення об'єкта.
- Будівельник (Builder): Інтерфейс або абстрактний клас, що оголошує методи для конструювання окремих частин об'єкта.
- Конкретні будівельники (Concrete Builders): Реалізують інтерфейс будівельника для конструювання конкретних типів об'єктів.
- Продукт (Product): Результат конструювання, який представляє собою складний об'єкт.
- Додаткові деталі конструювання (Construction Details): Будівельник може мати методи для додавання додаткових деталей до об'єкта.
- Можливість отримати результат (Getting the Result): Після завершення конструювання, будівельник повертає готовий об'єкт.

Шаблон "BUILDER" дозволяє створювати різні варіанти одного об'єкта шляхом зміни будівельників. Він особливо корисний, коли існує багато можливих конфігурацій об'єкта, і важко чи незручно конструювати його без використання спеціального інтерфейсу.

3. ШАБЛОН «COMMAND»

Шаблон "COMMAND" є одним із базових шаблонів проектування, який дозволяє ізольовано обробляти запити чи команди як об'єкти. Він дозволяє реалізувати функціональність виклику методу в об'єкті без знання конкретної реалізації цього методу.

Основні характеристики та особливості шаблону "COMMAND":

- Команда (Command): Цей клас містить у собі інформацію про конкретну команду та об'єкт, на якому вона повинна виконатися. Він містить методи execute(), який викликається для виконання команди.
- Отримувач (Receiver): Це об'єкт, на якому виконується команда. Він містить фактичний код, який виконує дії, які вимагає команда.
- Викликач (Invoker): Цей клас викликає команди для виконання. Він має посилання на конкретну команду та може викликати її метод execute().
- Сполучник (Client): Цей клас створює об'єкти команд та прив'язує їх до відповідних отримувачів.
- Відміна операцій (Undo Operations): Шаблон "COMMAND" дозволяє легко реалізувати можливість відміни виконаних операцій.
- Композиція команд (Composite Commands): Команди можуть бути скомпоновані у складніше дерево, що дозволяє виконувати групу команд як єдину одиницю.

Шаблон "COMMAND" дозволяє відокремити ініціатора запиту від конкретної реалізації виконання команди. Це робить систему більш гнучкою та дозволяє легко розширювати набір команд. Він особливо корисний в ситуаціях, коли потрібно реалізувати відміну операцій або створити комплексні команди з різних простіших.

4. ШАБЛОН «CHAIN OF RESPONSIBILITY»

Шаблон "CHAIN OF RESPONSIBILITY" є одним з поведінкових шаблонів проектування, який дозволяє передавати запити через ланцюг обробників. Якщо один обробник не може обробити запит, він передає його на обробку наступному обробнику в ланцюжку.

Основні характеристики та особливості шаблону "CHAIN OF RESPONSIBILITY":

- Обробник (Handler): Абстрактний клас або інтерфейс, що описує метод обробки запиту та має посилання на наступний обробник в ланцюжку.
- Конкретні обробники (Concrete Handlers): Конкретні реалізації обробників, які спробують обробити запит. Якщо вони не можуть обробити його, вони передають його наступному обробнику в ланцюжку.
- Ланцюг обробників (Chain of Handlers): Обробники формують ланцюг, в якому кожен обробник може намагатися обробити запит, або передати його далі.
- Передача запиту (Request Propagation): Якщо обробник не може обробити запит, він передає його наступному обробнику в ланцюжку.
- Зупинка ланцюжка (Stopping the Chain): Обробник може вирішити не передавати запит далі, зупиняючи ланцюг.
- Динамічне додавання та видалення обробників (Dynamic Addition and Removal of Handlers): Обробники можуть динамічно додаватися та видалятися з ланцюжка під час виконання програми.

Шаблон "CHAIN OF RESPONSIBILITY" дозволяє побудувати логічний ланцюг для обробки запитів, при цьому кожен обробник вирішує, чи може він обробити запит, чи повинен передати його далі. Це робить систему більш гнучкою та дозволяє легко додавати чи змінювати обробників.

5. ШАБЛОН «PROTOTYPE»

Шаблон "PROTOTYPE" є одним з паттернів проектування, який дозволяє створювати нові об'єкти на основі вже існуючих прототипів. Він використовує копіювання об'єктів для генерації нових екземплярів.

Основні характеристики та особливості шаблону "PROTOTYPE":

- Прототип (Prototype): Це абстрактний клас або інтерфейс, який визначає метод для копіювання самого себе.
- Конкретні прототипи (Concrete Prototypes): Конкретні реалізації прототипів, які реалізують метод копіювання.
- Клієнт (Client): Використовує прототип для створення нових об'єктів шляхом копіювання існуючого.
- Копіювання (Cloning): Прототип дозволяє копіювати себе, створюючи новий екземпляр.
- Глибоке та поверхневе копіювання (Deep and Shallow Copy): Прототип може здійснювати глибоке або поверхневе копіювання, залежно від потреб програми.
- Ефективне створення об'єктів (Efficient Object Creation): Шаблон "PROTOTYPE" дозволяє ефективно створювати нові об'єкти, особливо коли є необхідність у багатьох подібних екземплярах.
- Динамічна зміна класу об'єкта (Dynamic Class Change): Прототип дозволяє динамічно змінювати клас об'єкта, створюючи новий екземпляр на основі прототипу.

Шаблон "PROTOTYPE" дозволяє зекономити час та ресурси, оскільки нові об'єкти можна створювати на основі вже існуючих. Він особливо корисний, коли об'єкти мають складну ініціалізацію або коли потрібно генерувати велику кількість подібних екземплярів.

Паттерн Command

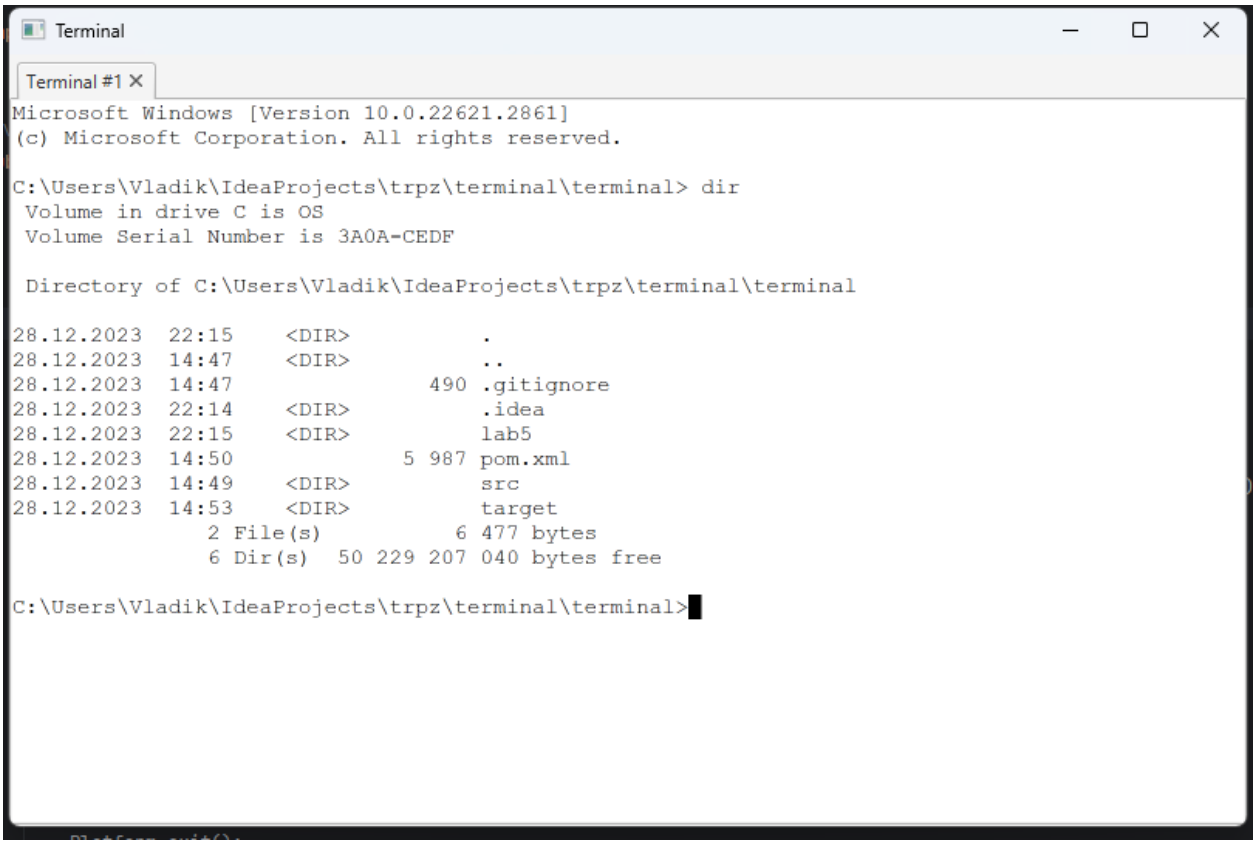
Розглянемо клас Terminal

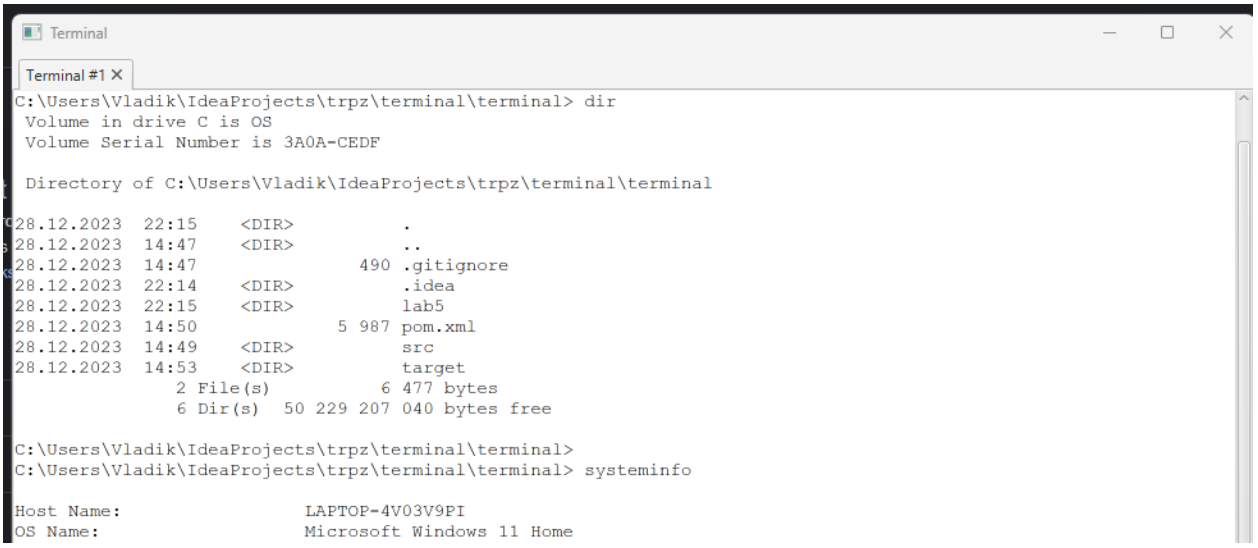
```
public class Terminal extends TerminalView {

    private PtyProcess process;
    private final ObjectProperty<Writer> outputWriterProperty;
    private final Path terminalPath;
    private String[] termCommand;
    private final LinkedBlockingQueue<String> commandQueue;

    public Terminal() {
        this(null, null);
    }

    public Terminal(TerminalConfig terminalConfig, Path
terminalPath) {

        setTerminalConfig(terminalConfig);
        this.terminalPath = terminalPath;
        outputWriterProperty = new SimpleObjectProperty<>();
        commandQueue = new LinkedBlockingQueue<>();
    }

    @WebkitCall
    public void command(String command) {
        try {
            commandQueue.put(command);
        } catch (final InterruptedException e) {
            throw new RuntimeException(e);
        }
        ThreadHelper.start(() -> {
            try {
                final String commandToExecute =
commandQueue.poll();
                getOutputWriter().write(commandToExecute);
                getOutputWriter().flush();
            } catch (final IOException e) {
                throw new RuntimeException(e);
            }
        });
    }

    @Override
    public void onTerminalReady() {
        ThreadHelper.start(() -> {
            try {
                initializeProcess();
```

```
            } catch (final Exception e) {
                throw new RuntimeException(e);
            }
        });
    }

    private void initializeProcess() throws Exception {
        final Path dataDir = getDataDir();
        if (SystemUtils.IS_OS_WINDOWS) {
            this.termCommand =
getTerminalConfig().getWindowsTerminalStarter().split("\\s+");
        } else {
            this.termCommand =
getTerminalConfig().getUnixTerminalStarter().split("\\s+");
        }

        final Map<String, String> envs = new
HashMap<>(System.getenv());
        envs.put("TERM", "xterm");

        System.setProperty("PTY_LIB_FOLDER",
dataDir.resolve("libpty").toString());

        if (Objects.nonNull(terminalPath) &&
Files.exists(terminalPath)) {
            this.process = PtyProcess.exec(termCommand, envs,
terminalPath.toString());
        } else {
            this.process = PtyProcess.exec(termCommand, envs);
        }

        columnsProperty().addListener(evt -> updateWinSize());
        rowsProperty().addListener(evt -> updateWinSize());
        updateWinSize();
        String
defaultCharEncoding=System.getProperty("file.encoding");
        setInputReader(new BufferedReader(new
InputStreamReader(process.getInputStream(),
defaultCharEncoding)));
        setErrorReader(new BufferedReader(new
InputStreamReader(process.getErrorStream(),
defaultCharEncoding)));
        setOutputWriter(new BufferedWriter(new
OutputStreamWriter(process.getOutputStream(),
defaultCharEncoding)));
        focusCursor();

        countDownLatch.countDown();
```

```
        process.waitFor();
    }

    private Path getDataDir() {
        final String userHome = System.getProperty("user.home");
        final Path dataDir =
Paths.get(userHome).resolve(".terminalfx");
        return dataDir;
    }

    public Path getTerminalPath() {
        return terminalPath;
    }

    private void updateWinSize() {
        try {
            process.setWinSize(new WinSize(getColumns(),
getRows()));
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }

    public ObjectProperty<Writer> outputWriterProperty() {
        return outputWriterProperty;
    }

    public Writer getOutputWriter() {
        return outputWriterProperty.get();
    }

    public void setOutputWriter(Writer writer) {
        outputWriterProperty.set(writer);
    }

    public PtyProcess getProcess() {
        return process;
    }

}
```

Кожна команда представляє собою окремий об’єкт надісланий на обробку
Приклади:





Висновок:

Лабораторна робота була спрямована на дослідження та реалізацію п'яти важливих шаблонів проектування: "ADAPTER", "BUILDER", "COMMAND", "CHAIN OF RESPONSIBILITY" та "PROTOTYPE". Кожен із них має свою унікальну функціональність та може бути використаний для вирішення різноманітних завдань у програмній розробці.

- Шаблон проектування "ADAPTER": Цей шаблон дозволяє об'єктам з різними інтерфейсами працювати разом. Він надає проміжний інтерфейс, який перетворює один інтерфейс у інший, щоб вони могли взаємодіяти без проблем.
- Шаблон проектування "BUILDER": Цей шаблон дозволяє побудувати складні об'єкти крок за кроком. Він використовує декілька підприємств для конструювання об'єкта, щоб дати можливість створювати різні варіанти одного об'єкта.
- Шаблон проектування "COMMAND": Цей шаблон дозволяє ізольовано обробляти запити або команди як об'єкти. Він дозволяє реалізувати функціональність виклику методу в об'єкті без знання конкретної реалізації цього методу.
- Шаблон проектування "CHAIN OF RESPONSIBILITY": Цей шаблон дозволяє передавати запити через ланцюжок обробників. Якщо один обробник не може обробити запит, він передає його на обробку наступному обробнику в ланцюжку.
- Шаблон проектування "PROTOTYPE": Цей шаблон дозволяє створювати нові об'єкти на основі вже існуючих прототипів. Він використовує копіювання об'єктів для генерації нових екземплярів.