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

з дисципліни «Технології розроблення програмного забезпечення»

Тема: «Патерни проектування»

Тема роботи: 3. Текстовий редактор

Виконав
студент групи ІА–33
Супик Андрій Олександрович

Київ 2025

Тема: Патерни проектування

Мета: Вивчити структуру шаблонів «Adapter», «Builder», «Command», «Chain of responsibility», «Prototype» та навчитися застосовувати їх в реалізації програмної системи

Посилання на репозиторій: https://github.com/insxlll/trpzlab

Короткі теоретичні відомості

Адаптер (Adapter)

Шаблон проєктування "Адаптер" дозволяє об’єднати інтерфейси двох несумісних класів. Це корисно, коли потрібно використовувати наявний клас, але його інтерфейс не відповідає потребам клієнтського коду. Адаптер огортає клас-джерело і надає інтерфейс, що сумісний з інтерфейсом, який очікує клієнт. Таким чином, клієнтський код може працювати з адаптером, не змінюючи свій інтерфейс. У Java цей шаблон часто реалізується шляхом створення нового класу, який наслідує або реалізує необхідний інтерфейс і всередині використовує методи класу-джерела. Шаблон дозволяє покращити модульність і повторне використання коду, зберігаючи при цьому стабільний інтерфейс для клієнта.

Будівельник (Builder)

Шаблон "Будівник" забезпечує гнучкий спосіб створення складних об’єктів, дозволяючи будувати їх покроково. Це особливо корисно, коли об'єкт має багато опціональних параметрів або складну структуру. Замість того щоб створювати об'єкт через конструктор з численними параметрами, будівник дає можливість додавати значення по черзі і в кінці отримати готовий об’єкт. У Java це часто реалізується за допомогою вкладеного статичного класу "Builder" всередині основного класу, де кожен метод будівника повертає об'єкт будівника, дозволяючи ланцюжковий виклик

методів. Така реалізація робить код читабельнішим і гнучкішим для створення об'єктів з різними конфігураціями.

Команда (Command)

Шаблон "Команда" інкапсулює дію або запит як об’єкт, дозволяючи відкладене виконання операції, її скасування або збереження для подальшого використання. Кожна команда реалізує інтерфейс із методами для виконання дії (наприклад, execute). Це дозволяє відокремити клієнтський код від отримувача дії. Команди можна комбінувати, записувати в лог або ставити в чергу. У Java цей шаблон можна реалізувати за допомогою інтерфейсу Command і його конкретних реалізацій для різних операцій. Це полегшує обробку запитів в інтерфейсі, де користувач може обирати команди, а також зручно для реалізації таких функцій, як "Скасувати" або "Повторити" в додатках

Ланцюг відповідальностей (Chain of Responsibility)

Шаблон "Ланцюг відповідальностей" дозволяє передавати запит вздовж ланцюга обробників, де кожен обробник має шанс обробити запит або передати його далі. Це дозволяє клієнтському коду не знати, хто саме оброблятиме запит, і забезпечує гнучкість в додаванні чи видаленні обробників. Кожен обробник реалізує інтерфейс із методом для обробки запиту і зберігає посилання на наступного обробника в ланцюзі. У Java цей шаблон можна реалізувати шляхом створення абстрактного класу для обробників, де кожен конкретний обробник перевіряє, чи здатен він обробити запит, або передає його далі. Це зручно для систем обробки помилок або запитів з різними рівнями доступу.

Прототип (Prototype)

Шаблон "Прототип" дозволяє створювати нові об'єкти шляхом копіювання вже наявного об'єкта (прототипу) замість створення об’єктів "з нуля". Це ефективно, коли створення нового об'єкта є складним або ресурсозатратним, а також корисно для створення об'єктів зі схожими початковими станами. У Java шаблон може бути реалізований через інтерфейс Cloneable, що вимагає реалізації методу clone, який повертає копію об’єкта. Завдяки цьому шаблону можна створювати об'єкти зі схожою конфігурацією без необхідності повторювати весь процес ініціалізації, зберігаючи при цьому незалежність їхніх станів.

Хід роботи

3. **Текстовий редактор** (strategy, command, observer, template method, flyweight, SOA)

Текстовий редактор повинен вміти розпізнавати текстові файли в будь-якій кодуванні, мати розширені функції редагування: макроси, сніппети, підказки, закладки, перехід на рядок / сторінку, підсвічування синтаксису (для однієї мови програмування або розмітки на розсуд студента).

Текстовий редактор має такі функціональні можливості, як відкриття/збереження документа, редагування тексту (вставка, видалення), а найголовніше — можливість скасування та повторення останніх дій (Undo/Redo).

У цій роботі я використовую патерн "Команда" (Command). Цей патерн було обрано як архітектурне рішення для інкапсуляції дій користувача (вставка тексту, видалення) як об'єктів.

У процесі роботи над проєктом стало зрозуміло, що для реалізації надійної функції скасування (Undo) необхідно відокремити логіку виконання дії від інтерфейсу, який її ініціює.

Без використання патерну Command: реалізація Undo/Redo була б складною, вимагаючи дублювання логіки відміни безпосередньо в класі документа або контролера.

Патерн Command вирішує цю проблему елегантно. Він дозволяє:

- Інкапсулювати кожну дію (наприклад, InsertTextCommand або DeleteTextCommand) як окремий об'єкт.
- Забезпечити для кожного об'єкта два ключові методи: execute() (виконати дію) та undo() (скасувати дію).
- Зберігати послідовність виконаних команд у спеціальному менеджері історії (CommandHistory), що дозволяє легко керувати стеками для відміни та повтору.

У моєму проєкті цей підхід реалізований через інтерфейс Command, який використовує клас Document (Отримувач) для виконання фактичних змін тексту.

Command (Інтерфейс): ICommand. * Concrete Commands (Конкретні Команди): Класи, як-от InsertTextCommand, DeleteTextCommand, ReplaceTextCommand, SetTextCommand, ClearDocumentCommand та FindAndReplaceCommand. * Receiver (Отримувач): Клас Document, який містить дані (текст) та методи для їх фактичної зміни. * Invoker (Ініціатор/Менеджер): Клас DocumentController викликає конкретні команди. * History Manager (Історія Команд): Клас CommandHistory відповідає за управління стеками undo та redo.

Реалізація шаблону "Command"

Інтерфейс Command (Command.java)

Інтерфейс оголошує методи, необхідні для виконання та скасування операції.

```
public interface ICommand {
    void execute();
    void undo();
    void redo();
    public String getName();
}
```

Рисунок 1. - реалізація інтерфейсу ICommand

Конкретна Команда (TextChangeCommand.java)

Команда "Зміна Тексту" (Text Change) інкапсулює повну заміну вмісту документа.

```java
public class TextChangeCommand implements ICommand {
    private final Document document;
    private final String oldText;
    private final String newText;

    public TextChangeCommand(Document document, String oldText, String newText) {
        this.document = document;
        this.oldText = oldText;
        this.newText = newText;
    }

    @Override
    public void execute() {
        DocumentTextHelper.setText(document, newText);
    }

    @Override
    public void undo() {
        DocumentTextHelper.setText(document, oldText);
    }

    @Override
    public void redo() {
        DocumentTextHelper.setText(document, newText);
    }

    @Override
    public String getName() {
        return "Text Change";
    }
}
```

Рисунок 2. - реалізація класу TextChangeCommand

Конкретна Команда (InsertTextCommand.java)

Команда "Вставити текст" зберігає інформацію, необхідну для виконання та скасування.

```java
public class InsertTextCommand implements ICommand {
    private final Document document;
    private final int position;
    private final String text;
    private boolean executed;

    public InsertTextCommand(Document document, int position, String text) {
        this.document = document;
        this.position = position;
        this.text = text;
        this.executed = false;
    }

    @Override
    public void execute() {
        String currentText = DocumentTextHelper.getText(document);

        if (position < 0 || position > currentText.length()) {
            throw new IndexOutOfBoundsException("Invalid position: " + position);
        }

        String newText = currentText.substring(0, position) + text + currentText.substring(position);
        DocumentTextHelper.setText(document, newText);
        executed = true;
    }

    @Override
    public void undo() {
        if (executed) {
            String currentText = DocumentTextHelper.getText(document);
            String newText = currentText.substring(0, position) + currentText.substring(position + text.length());
            DocumentTextHelper.setText(document, newText);
        }
    }

    @Override
    public void redo() {
        execute();
    }

    @Override
    public String getName() {
        return "Insert Text";
    }
}
```

Рисунок 3. - реалізація класу InsertTextCommand

Конкретна Команда (ReplaceTextCommand.java)

Команда "Замінити текст" (Replace Text) зберігає інформацію, необхідну для виконання та скасування.

```java
public class ReplaceTextCommand implements ICommand {
    private final Document document;
    private final int start;
    private final int end;
    private final String newText;
    private String oldText;

    public ReplaceTextCommand(Document document, int start, int end, String newText) {
        this.document = document;
        this.start = start;
        this.end = end;
        this.newText = newText;
    }

    @Override
    public void execute() {
        String currentText = DocumentTextHelper.getText(document);

        if (start < 0 || end > currentText.length() || start > end) {
            throw new IndexOutOfBoundsException("Invalid range: " + start + " to " + end);
        }

        oldText = currentText.substring(start, end);
        String resultText = currentText.substring(0, start) + newText + currentText.substring(end);
        DocumentTextHelper.setText(document, resultText);
    }

    @Override
    public void undo() {
        if (oldText != null) {
            String currentText = DocumentTextHelper.getText(document);
            String resultText = currentText.substring(0, start) + oldText + currentText.substring(start + newText.length());
            DocumentTextHelper.setText(document, resultText);
        }
    }

    @Override
    public void redo() {
        String currentText = DocumentTextHelper.getText(document);
        String resultText = currentText.substring(0, start) + newText + currentText.substring(start + oldText.length());
        DocumentTextHelper.setText(document, resultText);
    }

    @Override
    public String getName() {
        return "Replace Text";
    }
}
```

Рисунок 4. - реалізація класу ReplaceTextCommand

Конкретна Команда (SetTextCommand.java)

Команда "Задати текст" (Set Text) зберігає інформацію, необхідну для виконання та скасування.

```java
public class SetTextCommand implements ICommand {
    private final Document document;
    private final String newText;
    private String oldText;

    public SetTextCommand(Document document, String newText) {
        this.document = document;
        this.newText = newText;
    }

    @Override
    public void execute() {
        oldText = DocumentTextHelper.getText(document);
        DocumentTextHelper.setText(document, newText);
    }

    @Override
    public void undo() {
        if (oldText != null) {
            DocumentTextHelper.setText(document, oldText);
        }
    }

    @Override
    public void redo() {
        DocumentTextHelper.setText(document, newText);
    }

    @Override
    public String getName() {
        return "Set Text";
    }
}
```

Рисунок 5. - реалізація класу SetTextCommand

Клас Історії Команд (CommandHistory)

Клас "Історія Команд" (Command History) зберігає послідовність виконаних команд, необхідну для забезпечення функціоналу скасування (undo) та повторення (redo).

```
public class CommandHistory {
    private final Stack<ICommand> undoStack;
    private final Stack<ICommand> redoStack;
    private final int maxHistorySize;

    public CommandHistory() {
        this(maxHistorySize:100);
    }

    public CommandHistory(int maxHistorySize) {
        this.undoStack = new Stack<>();
        this.redoStack = new Stack<>();
        this.maxHistorySize = maxHistorySize;
    }

    public void push(ICommand command) {
        redoStack.clear();

        if (undoStack.size() >= maxHistorySize) {
            undoStack.remove(0);
        }

        undoStack.push(command);
    }

    public boolean undo() {
        if (undoStack.isEmpty()) {
            return false;
        }

        ICommand command = undoStack.pop();
        command.undo();
        redoStack.push(command);
        return true;
    }

    public boolean redo() {
        if (redoStack.isEmpty()) {
            return false;
        }

        ICommand command = redoStack.pop();
        command.redo();
        undoStack.push(command);
        return true;
    }

    public boolean canUndo() {
        return !undoStack.isEmpty();
    }

    public boolean canRedo() {
        return !redoStack.isEmpty();
    }

    public void clear() {
        undoStack.clear();
        redoStack.clear();
    }

    public int getUndoSize() {
        return undoStack.size();
    }

    public int getRedoSize() {
        return redoStack.size();
    }

    public String getLastUndoCommandName() {
        if (undoStack.isEmpty()) {
            return null;
        }
        return undoStack.peek().getName();
    }

    public String getLastRedoCommandName() {
        if (redoStack.isEmpty()) {
            return null;
        }
        return redoStack.peek().getName();
    }
}
```

Рисунок 6-7. - реалізація класу CommandHistory

Конкретна Команда (ClearDocumentCommand.java)

Команда "Очистити Документ" (Clear Document) зберігає інформацію, необхідну для виконання та скасування.

```java
public class ClearDocumentCommand implements ICommand {
    private final Document document;
    private String previousText;

    public ClearDocumentCommand(Document document) {
        this.document = document;
    }

    @Override
    public void execute() {
        previousText = DocumentTextHelper.getText(document);
        DocumentTextHelper.setText(document, text:"");
    }

    @Override
    public void undo() {
        if (previousText != null) {
            DocumentTextHelper.setText(document, previousText);
        }
    }

    @Override
    public void redo() {
        DocumentTextHelper.setText(document, text:"");
    }

    @Override
    public String getName() {
        return "Clear Document";
    }
}
```

Рисунок 8. - реалізація класу ClearDocumentCommand

Конкретна Команда (FindAndReplaceCommand.java)

Команда "Знайти та замінити" (Find and Replace) зберігає інформацію, необхідну для виконання та скасування.

```java
public class FindAndReplaceCommand implements ICommand {
    private final Document document;
    private final String findText;
    private final String replaceText;
    private String oldText;

    public FindAndReplaceCommand(Document document, String findText, String replaceText) {
        this.document = document;
        this.findText = findText;
        this.replaceText = replaceText;
    }

    @Override
    public void execute() {
        oldText = DocumentTextHelper.getText(document);
        String newText = oldText.replace(findText, replaceText);
        DocumentTextHelper.setText(document, newText);
    }

    @Override
    public void undo() {
        if (oldText != null) {
            DocumentTextHelper.setText(document, oldText);
        }
    }

    @Override
    public void redo() {
        if (oldText != null) {
            String newText = oldText.replace(findText, replaceText);
            DocumentTextHelper.setText(document, newText);
        }
    }

    @Override
    public String getName() {
        return "Find and Replace";
    }
}
```

Рисунок 9. - реалізація класу FindAndReplaceCommand

Конкретна Команда (DeleteTextCommand.java)

Команда "Видалити текст" (Delete Text) зберігає інформацію, необхідну для виконання та скасування.

```java
public class DeleteTextCommand implements ICommand {
    private final Document document;
    private final int start;
    private final int end;
    private String deletedText;

    public DeleteTextCommand(Document document, int start, int end) {
        this.document = document;
        this.start = start;
        this.end = end;
    }

    @Override
    public void execute() {
        String currentText = DocumentTextHelper.getText(document);

        if (start < 0 || end > currentText.length() || start > end) {
            throw new IndexOutOfBoundsException("Invalid range: " + start + " to " + end);
        }

        deletedText = currentText.substring(start, end);
        String newText = currentText.substring(0, start) + currentText.substring(end);
        DocumentTextHelper.setText(document, newText);
    }

    @Override
    public void undo() {
        if (deletedText != null) {
            String currentText = DocumentTextHelper.getText(document);
            String newText = currentText.substring(0, start) + deletedText + currentText.substring(start);
            DocumentTextHelper.setText(document, newText);
        }
    }

    @Override
    public void redo() {
        if (deletedText != null) {
            String currentText = DocumentTextHelper.getText(document);
            String newText = currentText.substring(0, start) + currentText.substring(start + deletedText.length());
            DocumentTextHelper.setText(document, newText);
        }
    }

    @Override
    public String getName() {
        return "Delete Text";
    }
}
```

Рисунок 10. - реалізація класу DeleteTextCommand

Конкретна Команда (TextChangeCommand.java)

Команда "Зміна Тексту" (Text Change) зберігає інформацію, необхідну для виконання та скасування.

```java
public class TextChangeCommand implements ICommand {
    private final Document document;
    private final String oldText;
    private final String newText;

    public TextChangeCommand(Document document, String oldText, String newText) {
        this.document = document;
        this.oldText = oldText;
        this.newText = newText;
    }

    @Override
    public void execute() {
        DocumentTextHelper.setText(document, newText);
    }

    @Override
    public void undo() {
        DocumentTextHelper.setText(document, oldText);
    }

    @Override
    public void redo() {
        DocumentTextHelper.setText(document, newText);
    }

    @Override
    public String getName() {
        return "Text Change";
    }
}
```

Рисунок 11. - реалізація класу TextChangeCommand

Допоміжний Клас (DocumentTextHelper)

Клас "Помічник Тексту Документа" (Document Text Helper) зберігає функціональність для доступу та зміни основного тексту документа.

```java
public class DocumentTextHelper {

    public static String getText(Document document) {
        String[] content = document.getContent();
        if (content == null || content.length == 0) {
            return "";
        }
        return String.join("\n", content);
    }

    public static void setText(Document document, String text) {
        if (text == null || text.isEmpty()) {
            document.setContent(new String[0]);
        } else {
            document.setContent(text.split("\n", -1));
        }
    }

    public static int getTextLength(Document document) {
        return getText(document).length();
    }

    public static boolean isEmpty(Document document) {
        String[] content = document.getContent();
        return content == null || content.length == 0 || getText(document).isEmpty();
    }
}
```

Рисунок 12. - реалізація класу DocumentTextHelper

Діаграма класів:

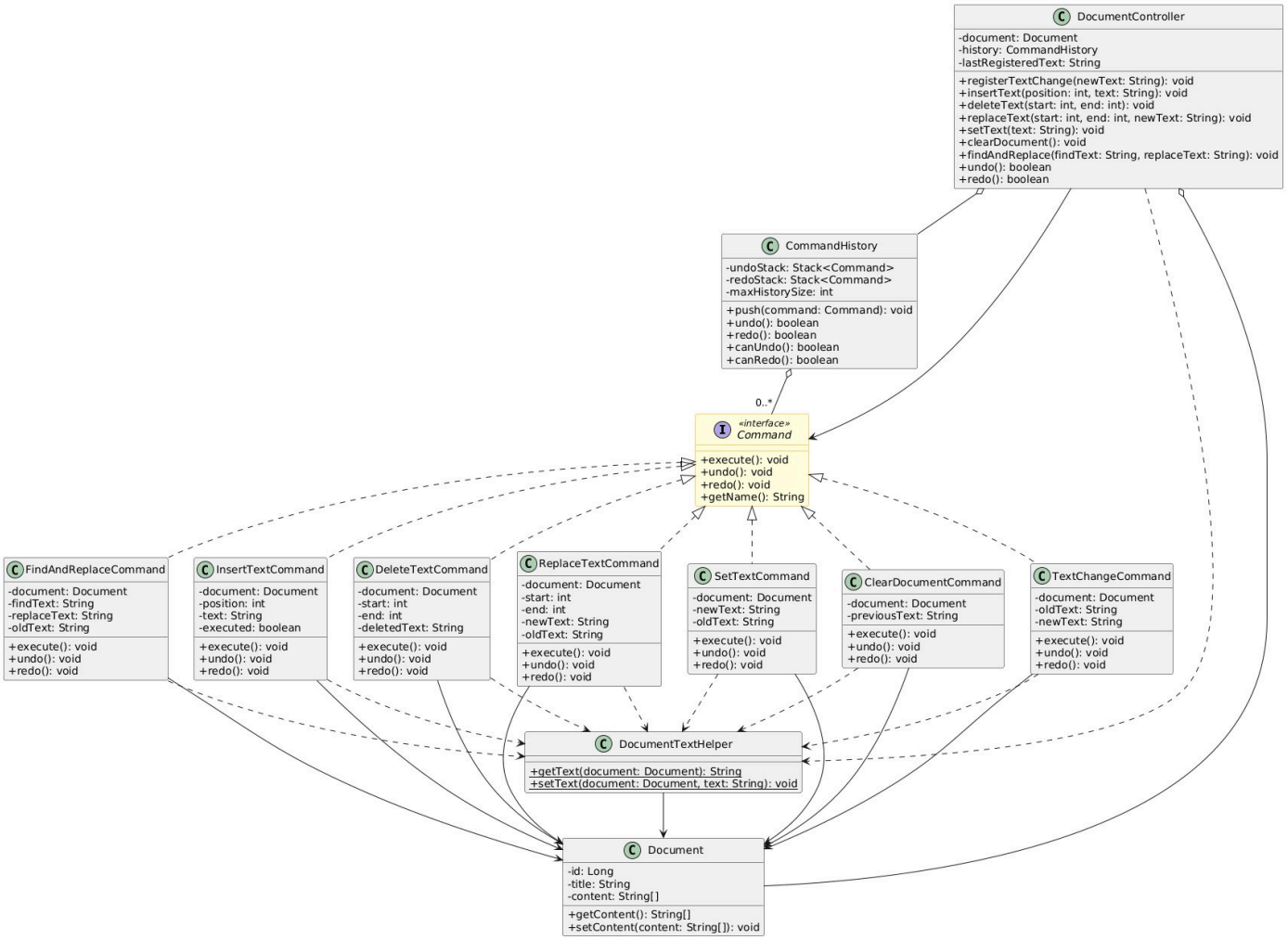


Рисунок 13. - Діаграма класів використання шаблону Command

Висновок: Виконуючи цю лабораторну роботу, я ознайомився з такими патернами, як Adapter, Builder, Command, Chain of Responsibility, Prototype.

Особливу увагу я приділив реалізації шаблону Command, детально описавши його основну логіку та функціональність у контексті мого проєкту текстового редактора.

Цей патерн було обрано як архітектурне рішення для спрощення реалізації функціоналу "Скасувати" та "Повторити". Я зрозумів, наскільки цей патерн спрощує процес управління складними діями, дозволяючи відокремити ініціатора (інтерфейс) від виконавця (документа).

Використання шаблону Command забезпечує:

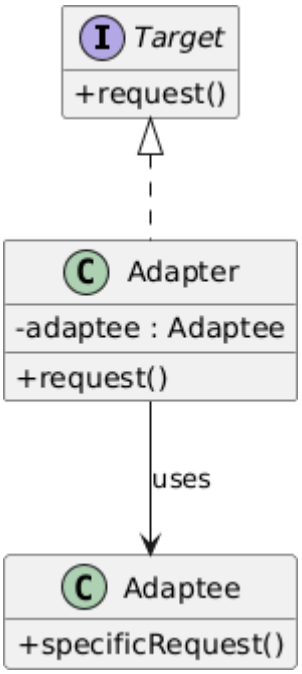
- Гнучкість та розширюваність.
- Можливість легко логіювати та ставити в чергу дії.
- Чітку та структуровану архітектуру програми, особливо для реалізації механізму Undo/Redo.

Контрольні запитання

1. Яке призначення шаблону «Адаптер»?

Призначення шаблону «Адаптер» - забезпечити спільну роботу класів з несумісними інтерфейсами. Він діє як "перехідник" або "обгортка" навколо існуючого класу (Adaptee), перетворюючи його інтерфейс на інший, очікуваний клієнтським кодом (Client)

2. Нарисуйте структуру шаблону «Адаптер».



3. Які класи входять в шаблон «Адаптер», та яка між ними взаємодія?

Client (Клієнт): Клас, який хоче використовувати функціонал, але очікує певний інтерфейс (Target). Target (Цільовий інтерфейс): Інтерфейс, який використовує Client. Adapter (Адаптер): Клас, який реалізує інтерфейс Target і містить посилання на об'єкт Adaptee. Він перенаправляє виклики від Client до Adaptee, виконуючи необхідні перетворення. Adaptee (Об'єкт, що адаптується): Існуючий клас з несумісним інтерфейсом.

4. Яка різниця між реалізацією «Адаптера» на рівні об’єктів та на рівні класів?

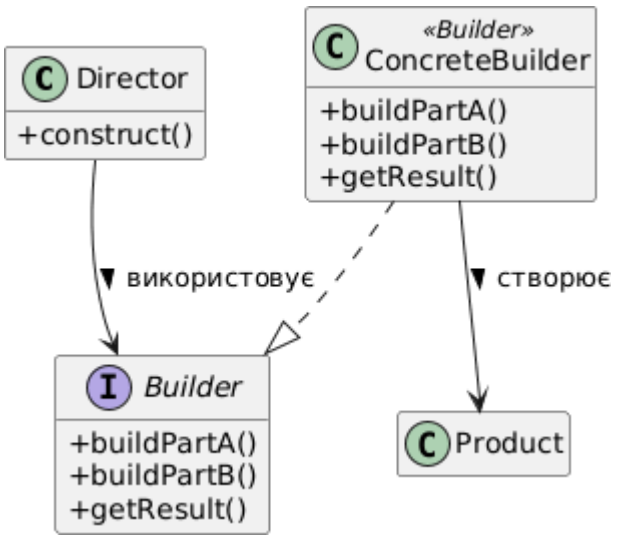
Адаптер об'єктів: Використовує композицію. Клас Adapter містить екземпляр класу Adaptee. Цей підхід є більш гнучким, оскільки дозволяє адаптувати будь-який підклас Adaptee.

Адаптер класів: Використовує множинне успадкування (в Java реалізується через успадкування класу та реалізацію інтерфейсу). Клас Adapter одночасно успадковує Adaptee та реалізує інтерфейс Target. Цей підхід менш гнучкий.

5. Яке призначення шаблону «Будівельник»?

Шаблон «Будівельник» (Builder) використовується для покрокового створення складних об'єктів. Він дозволяє відокремити процес конструювання об'єкта від його представлення, завдяки чому один і той самий процес конструювання може створювати різні представлення

6. Нарисуйте структуру шаблону «Будівельник».



7. Які класи входять в шаблон «Будівельник», та яка між ними взаємодія?

Product (Продукт): Складний об'єкт, який створюється. Builder (Будівельник): Інтерфейс для створення частин об'єкта Product. ConcreteBuilder (Конкретний будівельник): Реалізує інтерфейс Builder і конструює конкретне представлення продукту. Director (Директор): Клас, який керує процесом побудови, використовуючи об'єкт Builder.

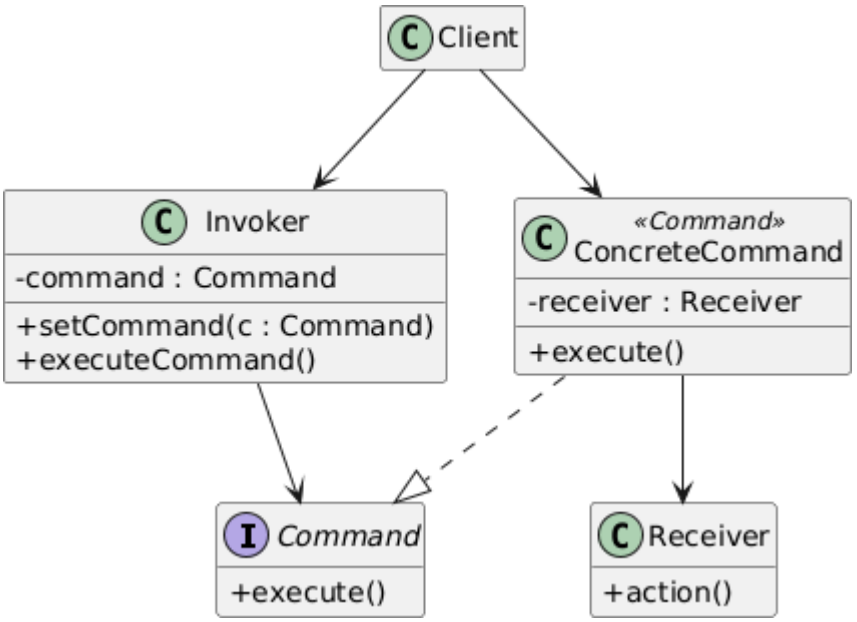
8. У яких випадках варто застосовувати шаблон «Будівельник»"?

Коли процес створення об'єкта є складним, багатоетапним, або коли конструктор має занадто багато параметрів (особливо опціональних). Також, коли потрібно створювати різні представлення одного й того ж об'єкта.

9. Яке призначення шаблону «Команда»?

Шаблон «Команда» (Command) інкапсулює запит на виконання дії як об'єкт. Це дозволяє параметризувати клієнтські об'єкти різними запитами, ставити запити в чергу, логувати їх, а також підтримувати операції скасування (undo).

10. Нарисуйте структуру шаблону «Команда».



11. Які класи входять в шаблон «Команда», та яка між ними взаємодія?

Command: Інтерфейс, що оголошує метод для виконання операції (execute). ConcreteCommand: Реалізує інтерфейс Command, містить посилання на Receiver і викликає його методи. Client: Створює об'єкт ConcreteCommand і встановлює його одержувача. Invoker: Просить команду виконати запит. Receiver: "Одержувач", який знає, як виконати операцію.

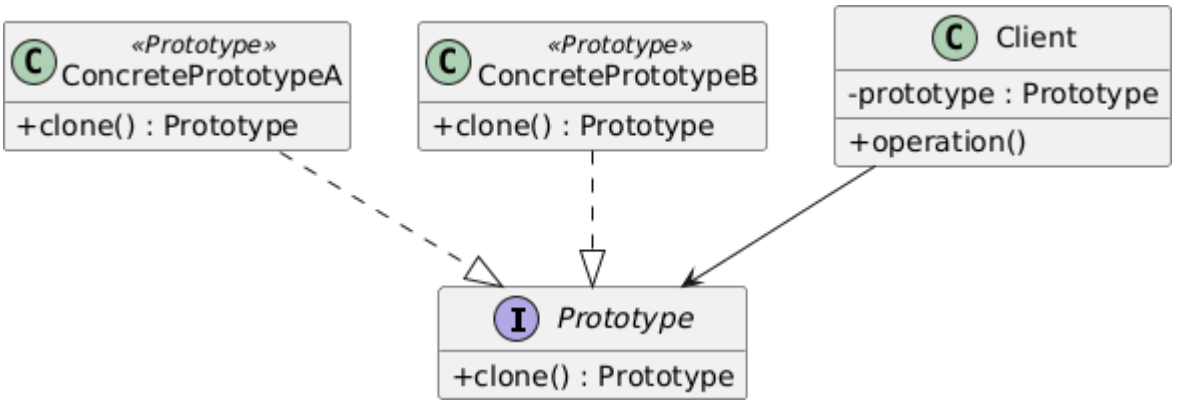
12. Розкажіть як працює шаблон «Команда».

Клієнт створює об'єкт-команду, пов'язуючи її з конкретним одержувачем. Потім цей об'єкт-команда передається ініціатору (Invoker), наприклад, кнопці в меню. Коли користувач натискає кнопку, ініціатор викликає метод execute() у команди, а команда, в свою чергу, викликає потрібний метод у свого одержувача.

13. Яке призначення шаблону «Прототип»?

Шаблон «Прототип» (Prototype) дозволяє створювати нові об'єкти шляхом копіювання існуючого об'єкта (прототипу). Це дозволяє уникнути прив'язки до класів об'єктів, що створюються.

14. Нарисуйте структуру шаблону «Прототип».



15. Які класи входять в шаблон «Прототип», та яка між ними взаємодія? Prototype: Інтерфейс, що оголошує метод клонування (clone).

ConcretePrototype: Реалізує інтерфейс Prototype та метод clone. Client: Створює новий об'єкт, викликаючи метод clone у прототипу.

16. Які можна привести приклади використання шаблону «Ланцюжок відповідальності»?

Обробка подій в GUI: Коли користувач клікає на кнопку, подія спочатку обробляється кнопкою, потім може бути передана батьківській панелі, потім вікну, і так далі, доки не буде оброблена. Системи логування: Повідомлення може проходити через ланцюжок обробників: один записує в консоль, інший - у файл, третій - відправляє по email, залежно від рівня важливості.

Системи авторизації та валідації: Запит користувача може проходити через ланцюжок перевірок: перевірка автентифікації, перевірка прав доступу, перевірка валідності даних.