Міністерство освіти і науки України
Національний технічний університет України
“Київський політехнічний інститут імені Ігоря Сікорського”
Факультет інформатики та обчислювальної техніки
Кафедра автоматики та управління в технічних системах

Лабораторна робота № 5

Технології розроблення програмного забезпечення

«Патерни проектування»

20. Mind-mapping software

Виконав:
студент групи ІА–33:
Хитрова НА

Перевірив:
Мягкий МЮ

Київ 2025

Тема: Патерни проектування.

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

5.1. Завдання

• Ознайомитись з короткими теоретичними відомостями.

• Реалізувати частину функціоналу робочої програми у вигляді класів та їхньої взаємодії для досягнення конкретних функціональних можливостей.

• Реалізувати один з розглянутих шаблонів за обраною темою.

• Реалізувати не менше 3-х класів відповідно до обраної теми.

• Підготувати звіт щодо виконання лабораторної роботи. Поданий звіт повинен містити: діаграму класів, яка представляє використання шаблону в реалізації системи, навести фрагменти коду по реалізації цього шаблону.

5.2. Теоретичні відомості

5.2.3. Шаблон «Command»

Призначення патерну: Шаблон "command" (команда) перетворить звичайний виклик методу в клас [6]. Таким чином дії в системі стають повноправними об'єктами. Це зручно в наступних випадках:

• Коли потрібна розвинена система команд – відомо, що команди будуть добавлятися;

• Коли потрібна гнучка система команд – коли з'являється необхідність додавати командам можливість відміни, логування і інш.;

• Коли потрібна можливість складання ланцюжків команд або виклику команд в певний час.

Об'єкт команда сама по собі не виконує ніяких фактичних дій окрім перенаправлення запиту одержувачеві (тобто команди все ж виконуються одержувачем), однак ці об'єкти можуть зберігати дані для підтримки додаткових функцій відміни, логування і інш. Наприклад, команда вставки символу може запам'ятовувати символ, і при виклику відміни викликати відповідну функцію витирання символу. Можна також визначити параметр «застосовності» команди (наприклад, на картинці писати не можна) – і використати цей аттрибут для засвічування піктограми в меню.

Такий підхід до команд дозволяє побудувати дуже гнучку систему команд, що настроюється. У більшості додатків це буде зайвим

(використовується спрощений варіант), проте життєво важливий в додатках з великою кількістю команд (редактори).

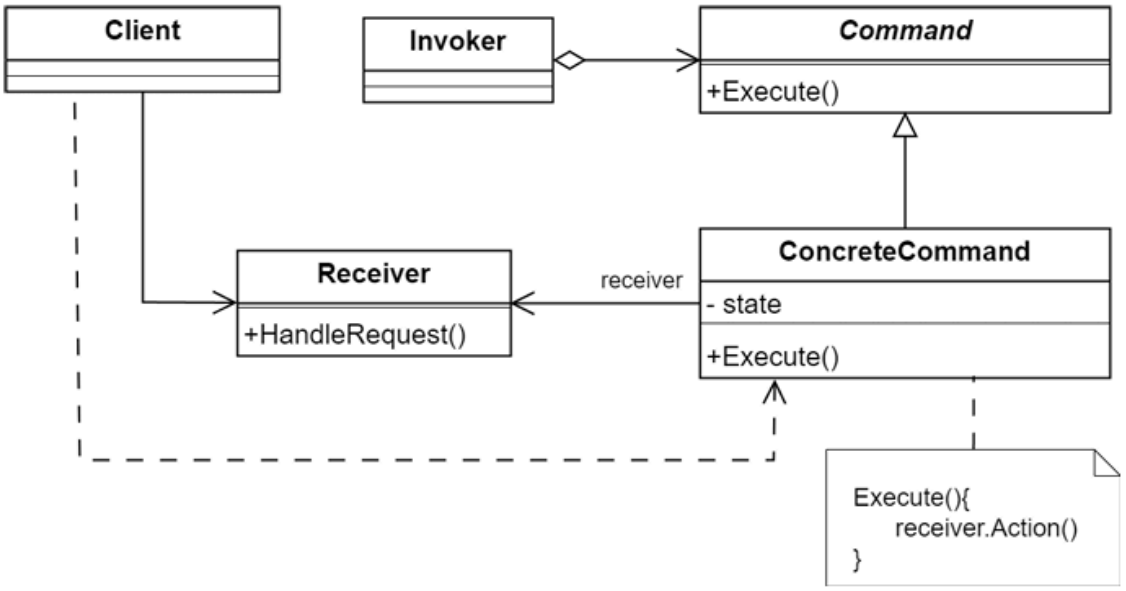


Рисунок 5.3. Структура патерну Команда

Проблема: Ви реалізуєте товстий клієнт який має багатий візуальний інтерфейс: має меню, кнопки і контекстне меню. Кожна дія, яку можна виконати, має три варіанти виконання – через меню, натисканням кнопки та через контекстне меню. Реалізацію кожної дії можна розмістити в обробнику візуального елементу, але тоді потрібно буде продублювати цей функціонал для меню, кнопки та контекстного меню. Таким чином ми будемо мати дублювання коду і при зміні функціоналу змінювати його в трьох місцях.

Додатковим викликом буде реалізувати автоматизоване тестування такої системи, тому що нам необхідно емулювати натискання кнопок користувачем.

Рішення: Виділення функціоналу який виконується по натисканню на кнопку в окремий клас дозволяє відвязати візуальну частину від логіки обробки. Таким чином ми будемо мати шар з UI елементами і шар з логікою обробки дій виконаних на UI. Це дозволяє один і той самий об'єкт з шару логіки обробки дій використати для реакцію на натискання кнопки і пункту меню і контекстного меню. Кнопки тепер не знають про конкретні класи реалізації команд, а взаємодіють з об'єктами команд через загальний інтерфейс. Ще одна перевага, що тепер ми можемо достатньо просто реалізувати механізм enable-disable для кнопок та контекстного меню через

виклик у об'єкта команди метода IsEnabled() або через прив'язку (binding) на поле IsEnabled у об'єкта команди.

При такій реалізації також набагато простіше організувати тестування застосунку, тому що ми тепер не потрібно емулювати дії користувача, а достатньо протестувати шар логіки обробки використовуючи модульні тести (unit tests)

Переваги та недоліки:

+ Ініціатор виконання команди не знає деталей реалізації виконавця команди.

+ Підтримує операції скасування та повторення команд.

+ Послідовність команд можна логувати і при необхідності виконати цю послідовність ще раз.

+ Простота розширення за рахунок додавання нових команд без необхідності внесення змін в уже існуючий код (принцип відкритості закритості).

Хід роботи

У нашій темі Mind-mapping software ми реалізуємо повноцінний графічний редактор карт пам’яті. Користувач вже може додавати вузли, переміщувати їх, видаляти, редагувати назви, запускати експорт, працювати з кількома картами тощо, тобто це класичний “товстий клієнт з багатим інтерфейсом”, для якого в методичці якраз і наводиться приклад задачі для шаблону Command.

Якщо реалізовувати дії “в лоб”, то кожна дія користувача (додати вузол, видалити, перемістити, обвести область, експортувати карту) реалізується безпосередньо в обробниках UI, тобто у кнопці “Add node” – один обробник, у пункті меню “Add node” – другий, у контекстному меню по правому кліку – третій, плюс окремо обробка гарячих клавіш. У результаті ми б отримали один і той самий функціонал, який дублюється в різних місцях. При зміні логіки (наприклад, ми додали категорії, зберігання в БД, додаткову валідацію) потрібно міняти код у кількох обробниках одразу. Складно організувати автоматизоване тестування – доводиться емулювати кліки по UI, замість того щоб тестувати чисту бізнес-логіку.

Для редактора mind-map це критично, тому що у нас багато команд (додавання / видалення / переміщення вузлів, додавання картинок, зміна категорій, експортування карти і т.д.), кожну з цих команд логічно викликати з різних місць інтерфейсу (меню, тулбар, контекстне меню,

гарячі клавіші). Тобто ми потрапляємо рівно в ту ситуацію, яка описана як “Проблема” для шаблону Command.

Як шаблон Command вирішує цю проблему в нашому проєкті? Ми виділяємо кожну дію над картою в окремий клас-команду, яка реалізує спільний інтерфейс Command з методами execute() і undo():

AddNodeCommand – додає новий вузол на карту;

MoveNodeCommand – змінює координати вузла;

DeleteNodeCommand – видаляє вузол (разом з піддеревом);

CommandManager – зберігає історію виконаних команд і підтримує undo/redo.

У такій структурі у нас UI-шар (MindMapEditorForm) взагалі не знає деталей реалізації , він працює з абстракцією Command:

кнопка “Add node” просто викликає commandManager.executeCommand(addNodeCommand);

пункт меню “Add node” робить те саме з тією ж командою;

гаряча клавіша – теж;

логіка дій над картою (додавання / переміщення / видалення вузлів) повністю винесена в окремі класи в шарі сервісів.

Таким чином ми розв’язуємо UI і бізнес-логіку, візуальні елементи не знають, “як саме” виконується дія, вони просто делегують її об’єкту-команді.

Одна з ключових переваг Command – це те, що команда зберігає достатньо даних для відміни дії. У нашому випадку MoveNodeCommand запам’ятовує старі координати вузла (oldX, oldY), щоб у undo() повернути його на місце, AddNodeCommand у undo() видаляє створений вузол, DeleteNodeCommand перед видаленням зберігає піддерево вузла, щоб при відміні повернути його назад. Менеджер команд CommandManager тримає два стеки. Один для виконаних команд (undo-історія), інший – для відмінених (redo). Це дає нам повноцінну функціональність Undo/Redo, яка є стандартом для будь-якого серйозного редактора та можливість у майбутньому логувати історію дій.

Реалізація в нашому додатку:

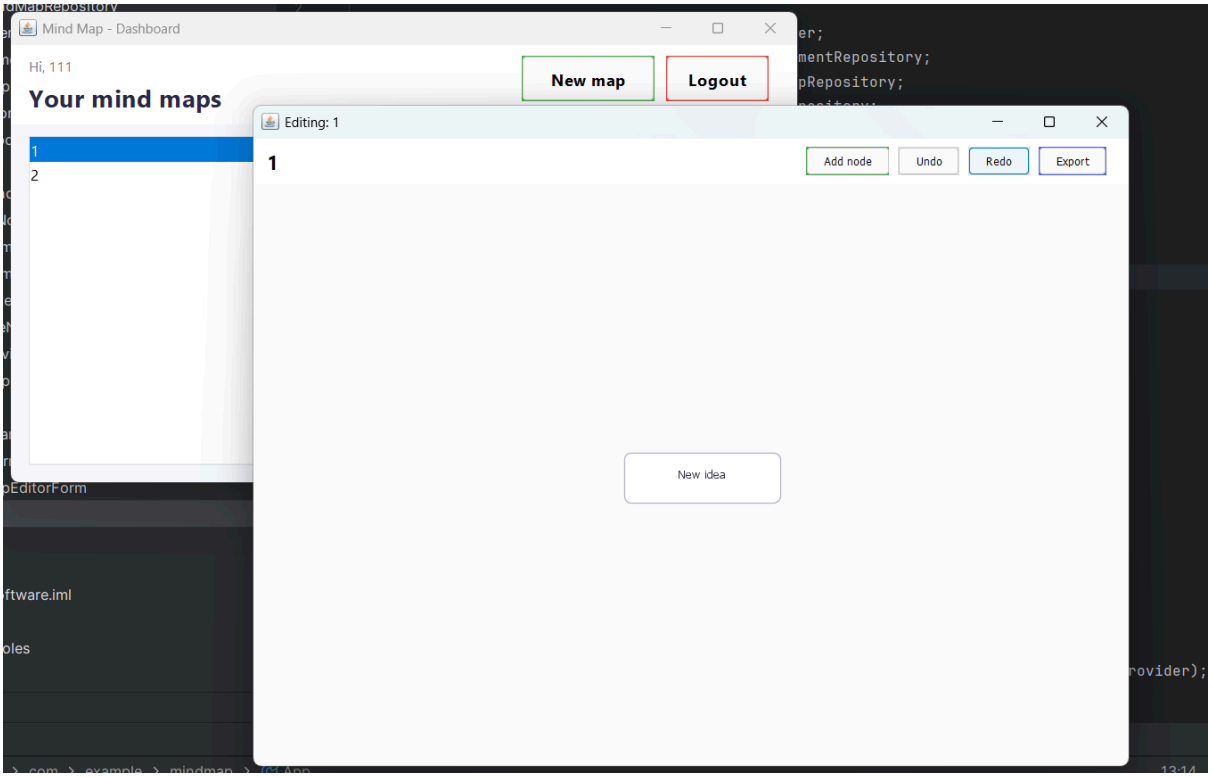


Рис. 1 Реалізація “Undo” “Redo” у нашому додатку

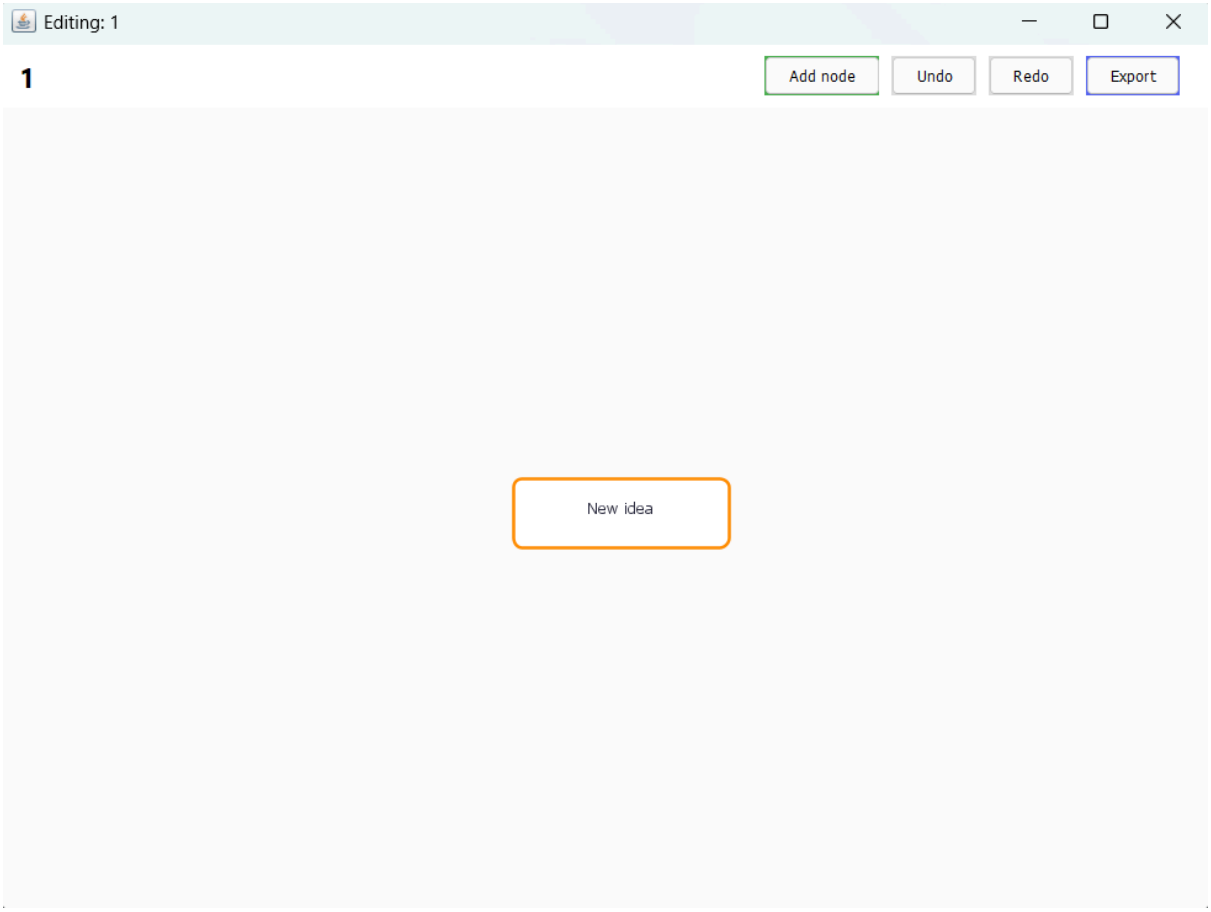
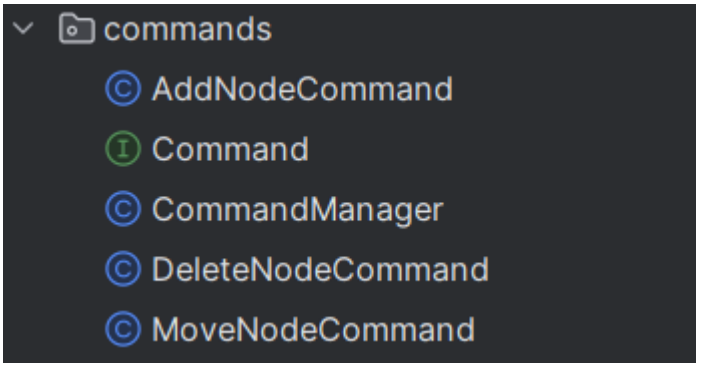


Рис. 2 Виділення вузла при натисканні, щоб можна було видалити кнопкою “delete” або “backspace”

Лістинг вихідного код системи, який було додано в цій лабораторній роботі:

https://github.com/ch1fir/trpz-lab-5



Command.java:

```
package com.example.mindmap.services.commands;

public interface Command {
    void execute();
    void undo();
```

```
}
```

CommandManager.java:

```
package com.example.mindmap.services.commands;

import java.util.Stack;

public class CommandManager {

    private final Stack<Command> undoStack = new Stack<>();
    private final Stack<Command> redoStack = new Stack<>();

    public void executeCommand(Command command) {
        command.execute();
        undoStack.push(command);
        redoStack.clear(); // після нової дії redo вже не актуальні
    }

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

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

    public void undo() {
        if (!canUndo()) return;

        Command command = undoStack.pop();
        command.undo();
        redoStack.push(command);
    }

    public void redo() {
        if (!canRedo()) return;

        Command command = redoStack.pop();
        command.execute();
        undoStack.push(command);
    }
}
```

AddNodeCommand.java:

```
package com.example.mindmap.services.commands;

import com.example.mindmap.entities.MindMap;
import com.example.mindmap.entities.MapElement;
import com.example.mindmap.services.MindMapService;

import java.util.List;

public class AddNodeCommand implements Command {

    private final MindMap mindMap;
```

```
    private final MindMapService mindMapService;
    private final List<MapElement> elements; // список CanvasPanel
    private final float x, y;
    private final String text;

    private MapElement createdElement;

    public AddNodeCommand(
            MindMap mindMap,
            MindMapService mindMapService,
            List<MapElement> elements,
            float x,
            float y,
            String text
    ) {
        this.mindMap = mindMap;
        this.mindMapService = mindMapService;
        this.elements = elements;
        this.x = x;
        this.y = y;
        this.text = text;
    }

    @Override
    public void execute() {
        createdElement = mindMapService.addTextNode(mindMap, x, y, text);
        elements.add(createdElement);
    }

    @Override
    public void undo() {
        elements.remove(createdElement);
        mindMapService.deleteElement(createdElement);
    }
}
```

MoveNodeCommand.java:

```
package com.example.mindmap.services.commands;

import com.example.mindmap.entities.MapElement;
import com.example.mindmap.services.MindMapService;

public class MoveNodeCommand implements Command {

    private final MapElement element;
    private final float oldX, oldY;
    private final float newX, newY;
    private final MindMapService mindMapService;

    public MoveNodeCommand(
            MapElement element,
            float oldX,
            float oldY,
```

```
            float newX,
            float newY,
            MindMapService mindMapService
    ) {
        this.element = element;
        this.oldX = oldX;
        this.oldY = oldY;
        this.newX = newX;
        this.newY = newY;
        this.mindMapService = mindMapService;
    }

    @Override
    public void execute() {
        element.setX(newX);
        element.setY(newY);
        mindMapService.updateElement(element);
    }

    @Override
    public void undo() {
        element.setX(oldX);
        element.setY(oldY);
        mindMapService.updateElement(element);
    }
}
```

DeleteNodeCommand.java:

```
package com.example.mindmap.services.commands;

import com.example.mindmap.entities.MapElement;
import com.example.mindmap.entities.MindMap;
import com.example.mindmap.services.MindMapService;

import java.util.List;

public class DeleteNodeCommand implements Command {

    private final MindMap mindMap;
    private final MindMapService mindMapService;
    private final List<MapElement> elements;

    private final MapElement target;
    private float x, y;
    private String text;

    public DeleteNodeCommand(
            MindMap mindMap,
            MindMapService mindMapService,
            List<MapElement> elements,
            MapElement target
    ) {
        this.mindMap = mindMap;
        this.mindMapService = mindMapService;
        this.elements = elements;
```

```
        this.target = target;
    }

    @Override
    public void execute() {
        // Збережемо дані для undo
        x = target.getX();
        y = target.getY();
        text = target.getTextForDisplay();

        elements.remove(target);
        mindMapService.deleteElement(target);
    }

    @Override
    public void undo() {
        MapElement restored = mindMapService.addTextNode(mindMap, x, y,
text);
        elements.add(restored);
    }
}
```

Висновок: У ході цієї лабораторної роботи було вивчено і практично реалізовано шаблон проектування Command. Основною метою впровадження цього шаблону у наш проект було відокремлення логіки виконання команд від користувацького інтерфейсу, а також створення гнучкого механізму для історії дій. Ми визначили, що у програмі "Mind-mapping software" користувач постійно взаємодіє з елементами карти – додає вузли, переміщує їх, видаляє. Традиційний підхід, коли кожен UI-елемент самостійно виконує відповідну дію, призводив би до дублювання коду, складності підтримки та неможливості впровадити undo/redo. Використання шаблону Command дозволило інкапсулювати кожну операцію у вигляді окремого класу, а інтерфейс Command став спільною точкою взаємодії.

5.3. Питання до лабораторної роботи:

1. Яке призначення шаблону «Адаптер»?

Адаптер дозволяє узгодити інтерфейси двох несумісних класів, щоб вони могли працювати разом, не змінюючи їх коду.

2. Нарисуйте структуру шаблону «Адаптер».

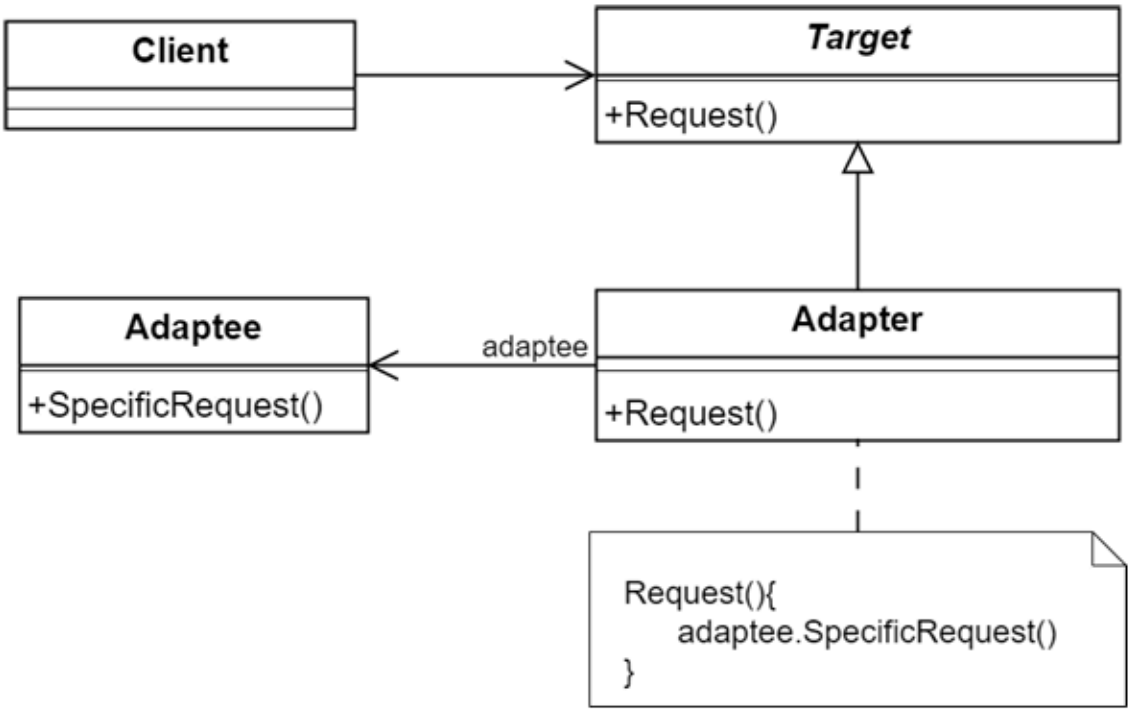


Рисунок 5.1. Структура патерну Адаптер на рівні об’єктів

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

До шаблону входять:

Цільовий інтерфейс – той, який очікує клієнт.

Адаптуємий клас – об’єкт із несумісним інтерфейсом.

Адаптер – об’єкт-перехідник, який переводить виклики клієнта у формат, зрозумілий адаптуємому класу.

Клієнт викликає методи через адаптер, а той перенаправляє їх до адаптуємого класу.

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

Адаптер на рівні об’єктів використовує композицію: він містить всередині екземпляр адаптуємого класу. Адаптер на рівні класів використовує наслідування: він одночасно наслідує і цільовий інтерфейс, і адаптуємий клас.

5. Яке призначення шаблону «Будівельник»?

Будівельник дозволяє створювати складні об’єкти крок за кроком, відокремлюючи процес побудови від кінцевої структури об’єкта.

6. Нарисуйте структуру шаблону «Будівельник».

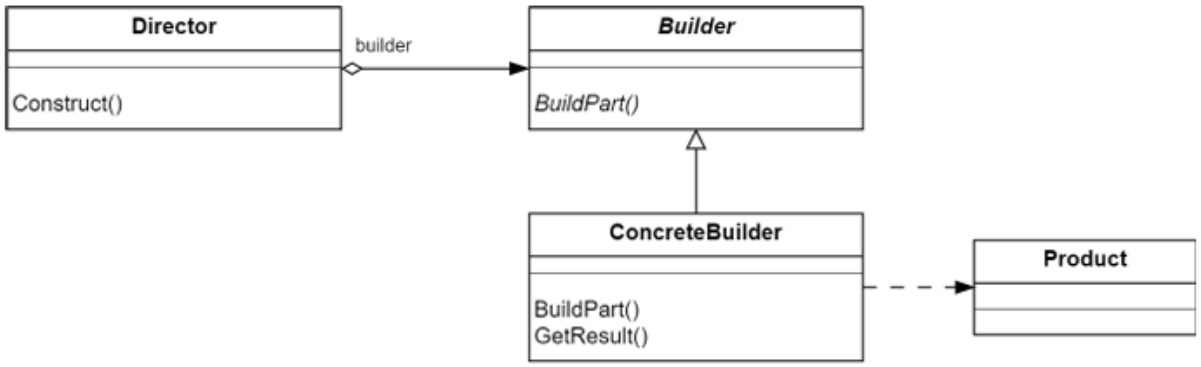


Рисунок 5.2. Структура патерну Builder

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

У шаблон входять:

Builder (інтерфейс) – описує кроки побудови об’єкта.

ConcreteBuilder – реалізує кроки та створює конкретний варіант продукту.

Director – керує процесом побудови, викликаючи методи билдера у потрібному порядку.

Product – кінцевий об’єкт, який будується.

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

Його використовують, коли об’єкт має багато кроків створення, багато параметрів або різні способи конфігурації, і коли важливо ізолювати логіку побудови.

9. Яке призначення шаблону «Команда»?

Команда перетворює запит у окремий об’єкт, що дозволяє передавати, ставити в чергу, скасовувати та логувати операції.

10. Нарисуйте структуру шаблону «Команда».

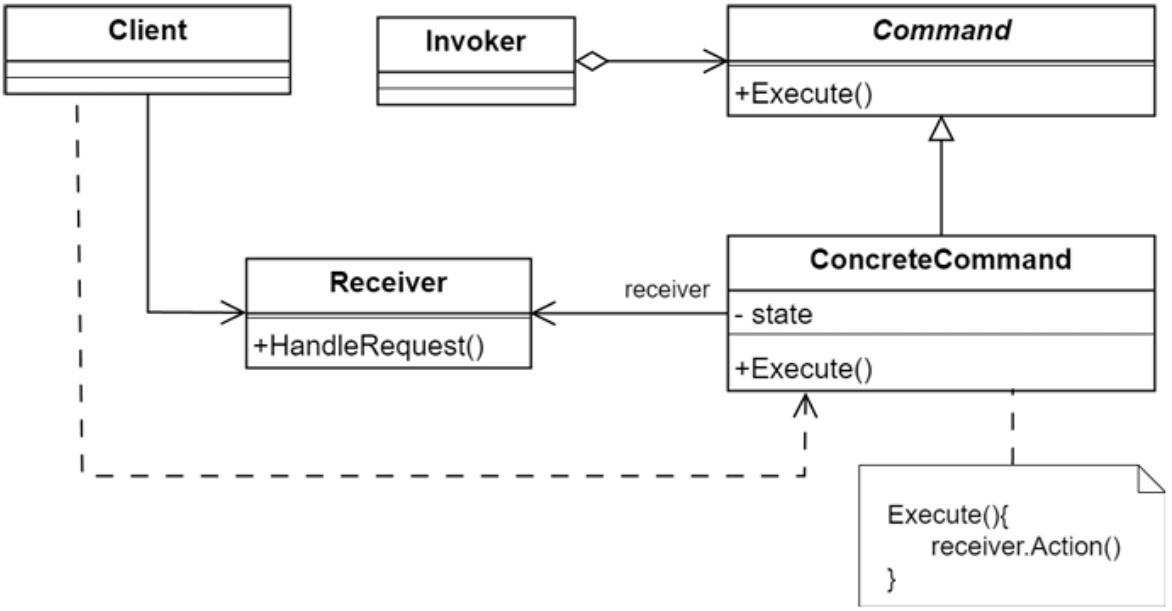


Рисунок 5.3. Структура патерну Команда

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

Компоненти:

Command (інтерфейс) – визначає метод виконання.

ConcreteCommand – містить посилання на отримувача та викликає його дії.

Receiver – об’єкт, який виконує конкретну роботу.

Invoker – об’єкт, який ініціює виконання команди.

Client – створює команди та передає їх Invoker-у.

12. Розкажіть як працює шаблон «Команда».

Клієнт створює об’єкт-команду і передає Invoker-у. Коли Invoker виконує команду, вона переносить запит до Receiver-а. Так запит стає об’єктом, який можна зберігати, скасовувати чи повторювати.

13. Яке призначення шаблону «Прототип»?

Прототип дозволяє створювати нові об’єкти шляхом копіювання існуючих, замість створення «з нуля», що зручно для складних об’єктів.

14. Нарисуйте структуру шаблону «Прототип».

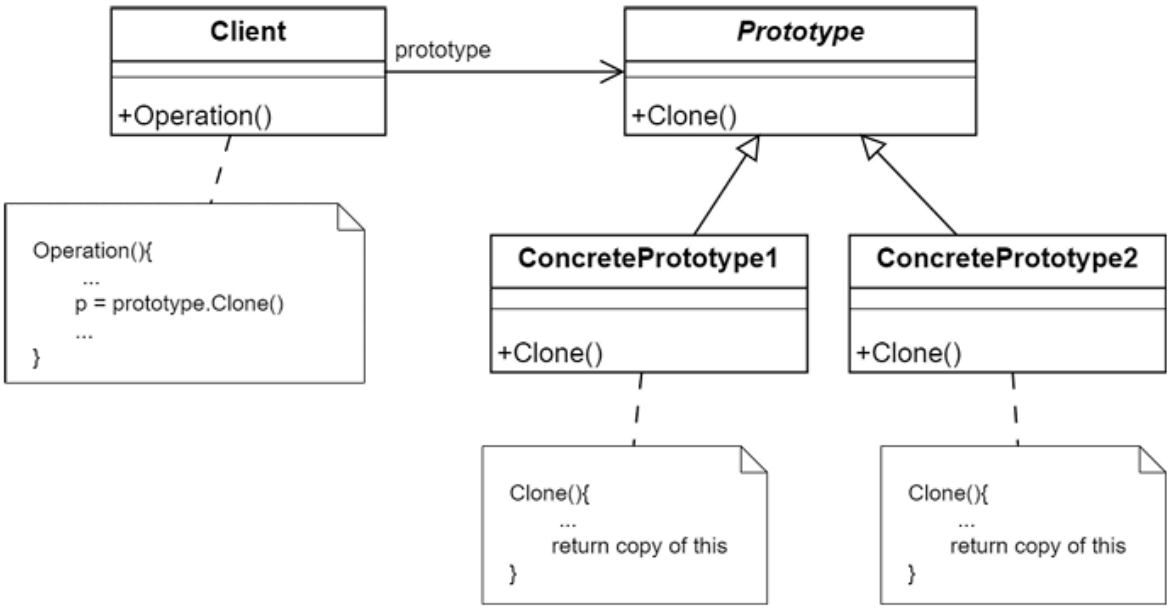


Рисунок 5.5. Структура патерну «Прототип»

15. Які класи входять в шаблон «Прототип», та яка між ними взаємодія?

Є два елементи:

Prototype (інтерфейс) – визначає метод clone().

ConcretePrototype – реалізує копіювання свого стану.

Клієнт викликає clone(), щоб отримати новий об’єкт, заснований на існуючому.

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

Обробка HTTP-запитів у middleware, система логування з різними рівнями detail, обробка подій у GUI (натискання миші проходить через ієрархію компонентів), перевірка бізнес-правил, де кожен обробник відповідає за свою умову.