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 проекту: «26. Download Manager»

Виконала:

студентка групи ІА-34

Мушта Анна

Перевірив:

Мягкий Михайло Юрійович

Дата здачі 25.10.2025

Захищено з балом

Київ 2025

Зміст

Тема 3

1.1. Теоретичні відомості 3

Шаблон «Adapter» 3

Шаблон «Builder» 3

Шаблон «Command» 3

Шаблон «Chain of Responsibility» 4

Шаблон «Prototype» 4

1.2. Хід роботи 5

1. Реалізувати частину функціоналу робочої програми у вигляді класів та їхньої взаємодії для досягнення конкретних функціональних можливостей. 5

2. Реалізувати шаблон Command з не менш ніж 3-ма класами 6

3. Продемонструвати результати 10

1.3. Висновки 11

1.4 Контрольні запитання 12

Тема: Download manager (iterator, command, observer, template method, composite, p2p) Інструмент для скачування файлів з інтернету по протоколах http або https з можливістю продовження завантаження в зупиненому місці, розподілу швидкостей активним завантаженням, ведення статистики завантажень, інтеграції в основні браузери (firefox, opera, internet explorer, chrome)

1.1. Теоретичні відомості

Шаблон «Adapter»

Патерн Adapter (Адаптер) використовується для узгодження інтерфейсів несумісних об’єктів. Його мета — надати єдиний уніфікований інтерфейс для роботи з різними реалізаціями. Наприклад, якщо існує кілька бібліотек для роботи з принтерами з різними інтерфейсами, можна створити адаптери, які “обгортають” ці бібліотеки та приводять їх до спільного інтерфейсу. Завдяки цьому клієнтський код не залежить від конкретної реалізації. Основна перевага — відокремлення логіки перетворення даних від бізнес-логіки, а недолік — збільшення кількості класів.

Шаблон «Builder»

Патерн Builder (Будівельник) відокремлює процес створення складного об’єкта від його представлення. Це зручно, коли об’єкт має багато етапів створення або кілька варіантів побудови. Наприклад, при формуванні відповіді web-сервера можна послідовно додавати заголовки, статус, вміст тощо. Будівельник забезпечує гнучкість і контроль над створенням, дозволяє легко змінювати або розширювати процес без впливу на решту системи. Основний недолік — залежність клієнта від конкретних реалізацій будівельників.

Шаблон «Command»

Патерн Command (Команда) перетворює дію в окремий об’єкт, що дозволяє відокремити ініціатора дії від її виконавця. Це зручно, коли потрібно створити гнучку систему команд, яку можна логувати, відміняти чи повторювати. Наприклад, у графічному додатку одна й та сама команда може виконуватись через меню, кнопку або контекстне меню без

дублювання коду. Команди мають спільний інтерфейс і зберігають інформацію, необхідну для виконання чи скасування дій. Основні переваги — підтримка undo/redo, логування та просте розширення системи; недоліки мінімальні, бо підхід робить код більш структурованим.

Шаблон «Chain of Responsibility»

Патерн Chain of Responsibility (Ланцюжок відповідальності) дозволяє передавати запит послідовно через ланцюжок обробників, поки один із них не обробить його. Наприклад, при формуванні контекстного меню у складному інтерфейсі кожен компонент може додати свої пункти і передати обробку “вгору” по ієрархії. Це зменшує залежності між компонентами, спрощує додавання нових обробників і підтримку коду. Недолік — запит може залишитись необробленим, якщо жоден елемент не підходить.

Шаблон «Prototype»

Патерн Prototype (Прототип) передбачає створення нових об’єктів шляхом копіювання вже існуючого “шаблонного” об’єкта. Це корисно, коли процес створення об’єкта складний або часто повторюється. Наприклад, у редакторі рівнів для гри кожен елемент можна створювати не з нуля, а клонуючи базовий зразок. Це підвищує продуктивність і гнучкість, дозволяючи змінювати копії незалежно одна від одної. Серед недоліків — складність реалізації глибокого клонування при наявності складних зв’язків між об’єктами.

1.2. Хід роботи.

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

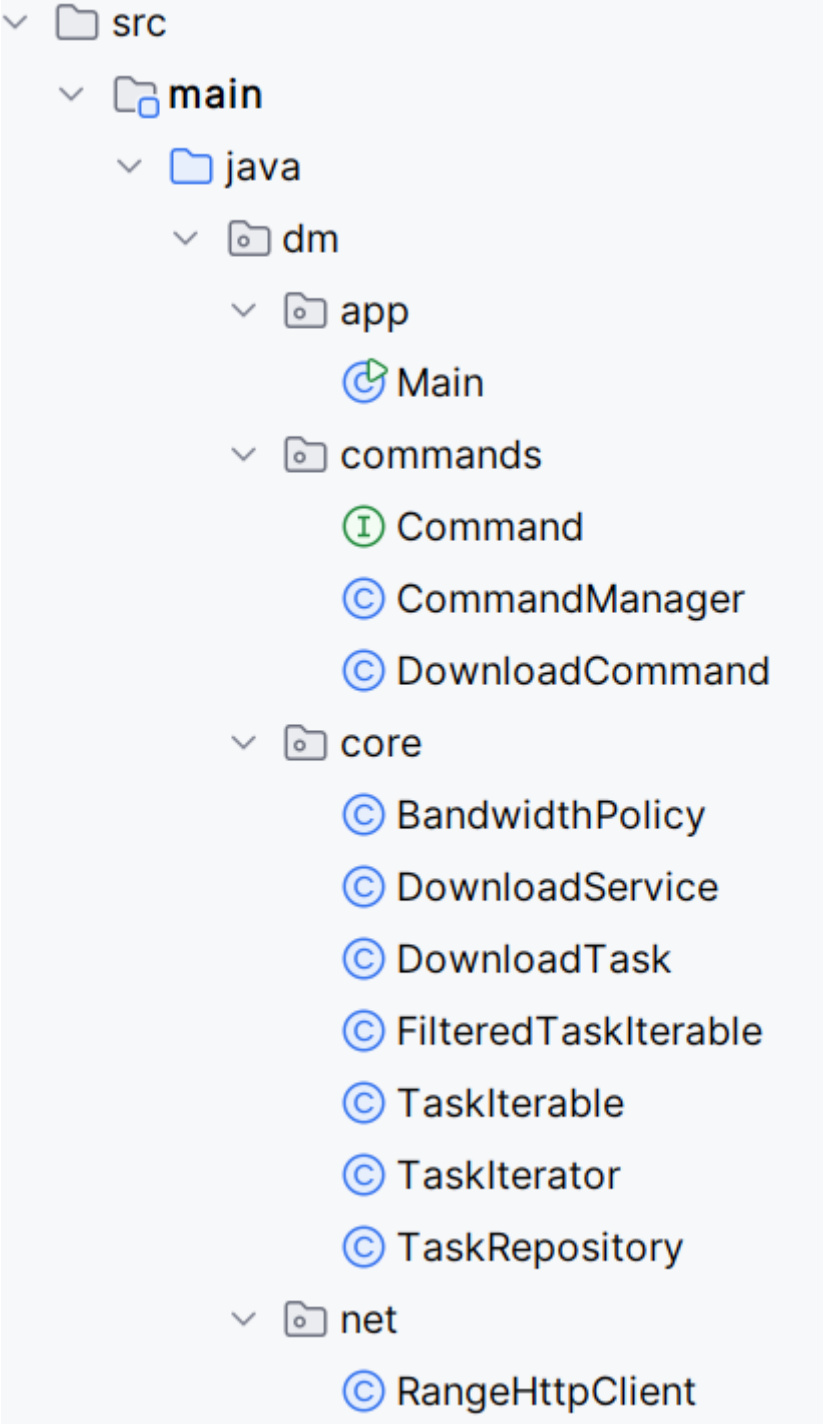


Рисунок 1 - Структура модулів і класів застосунку.

Проєкт розбитий на пакети dm.app, dm.core, dm.net. У dm.core реалізовано шаблон Iterator для обходу задач завантажень.

- DownloadTask - сутність задачі (id, url, target, status, прогрес).
- TaskRepository - доступ до SQLite; метод listRange(offset, limit) віддає записи батчами. Aggregate (джерело даних для ітератора).
- TaskIterable - колекція, що вміє створювати ітератор (iterator()); зберігає repo і розмір батча. Iterable.
- TaskIterator - послідовно підвантажує задачі через listRange, методи hasNext()/next(). Iterator.
- FilteredTaskIterable - обгортка над TaskIterable (наприклад, лише COMPLETED або PAUSED). Варіант ітератора з фільтрацією.
- DownloadService - керує паузою/відновленням, лімітом швидкості та надає доступ до TaskRepository.
- BandwidthPolicy — ліміт швидкості.
- RangeHttpClient — HTTP клієнт із підтримкою Range/відновлення.
- dm.app.Main — консольний інтерфейс; команда list використовує TaskIterable/FilteredTaskIterable для демонстрації шаблону.

Також шаблон Iterator застосовано для ефективного перегляду великої кількості задач з БД без завантаження всіх записів у пам’ять: TaskIterable/TaskIterator підвантажують їх батчами через TaskRepository.listRange(), а FilteredTaskIterable дозволяє додавати фільтри (наприклад, лише COMPLETED). Клієнтський код у Main працює з цією колекцією через звичний цикл for-each, не знаючи про внутрішню структуру сховища та спосіб вибірки.

2. Реалізувати шаблон Command з не менш ніж 3-ма класами

Я обрала Command Pattern для реалізації download-manager, тому що він найкраще відповідає структурі та вимогам мого проєкту. Нижче детальніше поясню, чому саме цей патерн є оптимальним вибором.

1. **Консольний інтерфейс з командами**
   У моєму download-manager користувач взаємодіє із системою через

консоль, вводячи команди типу pause, resume, add тощо. Це вже передбачає чітку структуру дій, які потрібно виконувати. Завдяки цьому я можу легко додавати нові команди або змінювати вже існуючі, не порушуючи роботу решти програми.

2. **Можливість реалізувати Undo/Redo функціонал**
   Однією з сильних сторін Command Pattern є можливість реалізувати скасування або повторення виконаних дій. У моєму випадку це дуже зручно: користувач може випадково поставити завантаження на паузу, і система повинна дозволяти йому легко повернутися до попереднього стану.
   Завдяки тому, що кожна дія зберігається як окремий об'єкт-команда, я можу легко "відкотити" її або повторити.
3. **Історія операцій та аудит**
   Ще одна важлива перевага цього патерну полягає у можливості вести історію всіх виконаних команд. У контексті download-manager це означає, що я можу зберігати список усіх дій користувача – які файли були додані, коли вони були призупинені чи відновлені.
   Це може бути корисним не лише для користувача, який хоче переглянути свої дії, а й для відновлення системи після збою або для діагностики помилок.

Тож перейдемо до реалізації

1. **Command.java** - інтерфейс з методами execute(), undo(), getDescription()

```
package dm.commands;

public interface Command {  9 usages  1 implementation  new *

    void execute() throws Exception;  2 usages  1 implementation  new *

    void undo() throws Exception;  1 usage  1 implementation  new *

    String getDescription();  3 usages  1 implementation  new *
}
```

Рисунок 2 – Інтерфейс Command з базовими методами execute(), undo() та getDescription()

2. **DownloadCommand.java** - конкретна команда для pause/resume з підтримкою undo

```
public class DownloadCommand implements Command {  4 usages  new *
    private final DownloadService service;  6 usages
    private final int taskId;  11 usages
    private final Action action;  5 usages
    private DownloadTask.Status previousStatus;  3 usages

    public enum Action { PAUSE, RESUME }  7 usages  new *

    public DownloadCommand(DownloadService service, int taskId, Action action) {  2 usages  new *
        this.service = service;
        this.taskId = taskId;
        this.action = action;
    }

    @Override  2 usages  new *
    public void execute() throws Exception {
        DownloadTask task = service.getRepository().findById(taskId);
        if (task != null) {
            previousStatus = task.status;
        }

        if (action == Action.PAUSE) {
            service.pause(taskId);
            System.out.println("Paused task #" + taskId);
        } else {
            service.resume(taskId);
            System.out.println("Resumed task #" + taskId);
        }
    }

    @Override  1 usage  new *
    public void undo() throws Exception {
        if (action == Action.PAUSE && previousStatus == DownloadTask.Status.RUNNING) {
            service.resume(taskId);
            System.out.println("Undo: Resumed task #" + taskId);
        } else if (action == Action.RESUME && previousStatus == DownloadTask.Status.PAUSED) {
            service.pause(taskId);
            System.out.println("Undo: Paused task #" + taskId);
        }
    }

    @Override  3 usages  new *
    public String getDescription() {
        return action + " task #" + taskId;
    }
}
```

Рисунок 3 – Клас DownloadCommand, що реалізує команди pause/resume з підтримкою скасування операцій

3. **CommandManager.java** - менеджер команд з історією та undo/redo

```java
public class CommandManager {  2 usages  new *
    private final Stack<Command> history = new Stack<>();  7 usages
    private final Stack<Command> redoStack = new Stack<>();  5 usages

    public void execute(Command command) throws Exception {  2 usages  new *
        command.execute();
        history.push(command);
        redoStack.clear();
    }

    public void undo() throws Exception {  1 usage  new *
        if (history.isEmpty()) {
            System.out.println("Nothing to undo");
            return;
        }

        Command cmd = history.pop();
        cmd.undo();
        redoStack.push(cmd);
        System.out.println("Undone: " + cmd.getDescription());
    }

    public void redo() throws Exception {  1 usage  new *
        if (redoStack.isEmpty()) {
            System.out.println("Nothing to redo");
            return;
        }

        Command cmd = redoStack.pop();
        cmd.execute();
        history.push(cmd);
        System.out.println("Redone: " + cmd.getDescription());
    }

    public void showHistory() {  1 usage  new *
        if (history.isEmpty()) {
            System.out.println("History is empty");
            return;
        }

        System.out.println("Command history:");
        int i = 1;
        for (Command cmd : history) {
            System.out.println(i++ + ". " + cmd.getDescription());
        }
    }
```

```
    public boolean canUndo() {  no usages  new *
        return !history.isEmpty();
    }

    public boolean canRedo() {  no usages  new *
        return !redoStack.isEmpty();
    }
}
```

Рисунок 4 - Клас CommandManager для управління історією команд та реалізації функціоналу undo/redo

3. Продемонструвати результати

```
<=Task created: #8% EXECUTING [4m 26s]          <=========----> 75% EXECUTING [4m 28s]
```

Рисунок 5 - Додавання нового завантаження через команду add

```
Paused task #8
```

Рисунок 6 - Призупинення завантаження за допомогою команди pause

```
> :app:run

#8 [PAUSED] https://speed.hetzner.de/100MB.bin (0/-1) -> test.bin
```

Рисунок 7 - Перегляд списку завантажень командою list

```
<=Undone: PAUSE task #8 EXECUTING [5m 45s]
```

Рисунок 8 - Скасування останньої операції командою undo

```
Redone: PAUSE task #8
```

Рисунок 9 - Повторення скасованої операції командою redo

```
Command history:
1. PAUSE task #8
2. RESUME task #8
```

Рисунок 10 - Перегляд історії виконаних команд

1.3. Висновки.

Під час виконання лабораторної роботи було детально проаналізовано п'ять ключових шаблонів проєктування: Adapter, Builder, Command, Chain of Responsibility та Prototype. Кожен із них виконує власну функцію, допомагаючи створювати більш гнучку, зрозумілу та підтримувану архітектуру програмного забезпечення. Зокрема:

- Adapter забезпечує взаємодію між класами з несумісними інтерфейсами, дозволяючи інтегрувати зовнішні бібліотеки без зміни їхнього коду.
- Builder спрощує процес створення складних об'єктів, розділяючи етапи побудови й кінцевий результат, що особливо корисно для об'єктів з багатьма параметрами.
- Command інкапсулює запити у вигляді об'єктів, що дає змогу гнучко керувати виконанням операцій, підтримувати історію команд та реалізовувати механізми скасування/повторення дій.
- Chain of Responsibility передає запит уздовж ланцюжка обробників, що дозволяє легко змінювати або розширювати логіку обробки без зміни клієнтського коду.
- Prototype дає можливість створювати нові об'єкти шляхом копіювання вже існуючих, зберігаючи їхній стан та уникаючи складної ініціалізації.

Особливий акцент у роботі зроблено на шаблоні Command, який був реалізований у межах проєкту «Download Manager». Його використання дозволило інкапсулювати операції керування завантаженнями (pause/resume) як окремі об'єкти-команди. Завдяки цьому вдалося реалізувати функціонал відміни та повторення операцій (undo/redo), створити історію виконаних команд та відокремити бізнес-логіку від користувацького інтерфейсу.

Реалізація складається з трьох ключових компонентів: інтерфейсу Command, що визначає базові операції execute() та undo(); класу DownloadCommand, який інкапсулює конкретні дії над завантаженнями та зберігає попередній стан для можливості скасування; та класу CommandManager, що виступає invoker'ом, управляє історією команд та забезпечує механізм undo/redo через використання стеків.

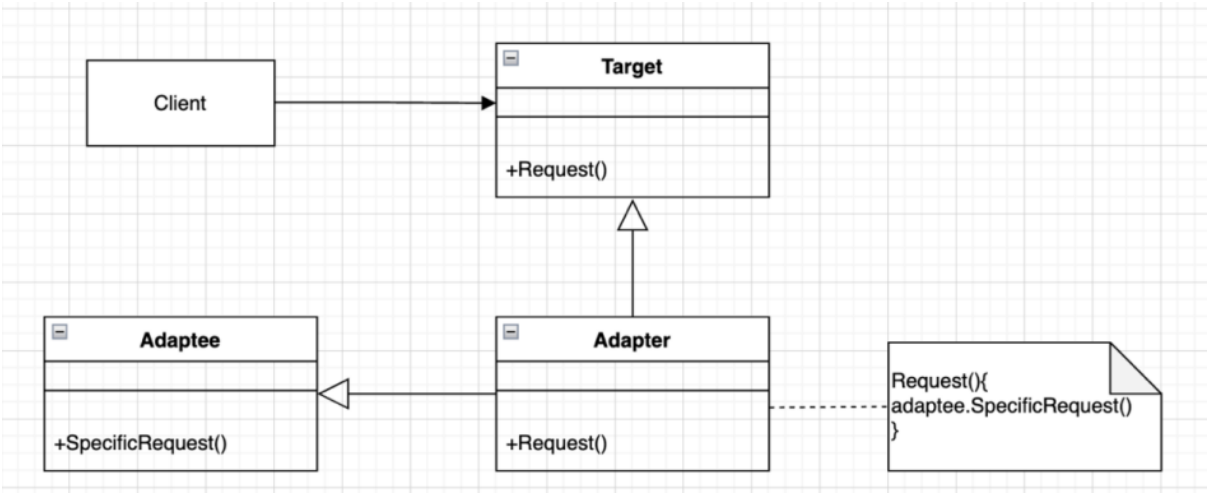
Застосування шаблонів проєктування продемонструвало їхню ефективність у розв'язанні типових інженерних задач і побудові масштабованої архітектури. Практична реалізація шаблону Command у проєкті підтвердила його значення для створення гнучких систем управління операціями, підвищення модульності коду та спрощення тестування окремих компонентів. Такий підхід полегшує подальший розвиток і підтримку програмного продукту, дозволяючи легко додавати нові команди (наприклад, SetLimitCommand, DeleteDownloadCommand) без зміни існуючого коду, що робить систему більш стійкою до змін і розширень у майбутньому.

1.4 Контрольні запитання

1. Яке призначення шаблону «Адаптер»?

Забезпечує сумісність між класами з різними інтерфейсами, «перетворюючи» інтерфейс одного класу до формату, який очікує інший.

2. Нарисуйте структуру шаблону «Адаптер».



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

• Target - інтерфейс, який очікує клієнт.

• Client - клас, що використовує Target.

• Adaptee - клас із несумісним інтерфейсом.

• Adapter - проміжний клас, який реалізує Target і викликає методи Adaptee.

Взаємодія:

Клієнт звертається до адаптера через інтерфейс Target. Адаптер, у свою чергу, викликає відповідні методи класу Adaptee.

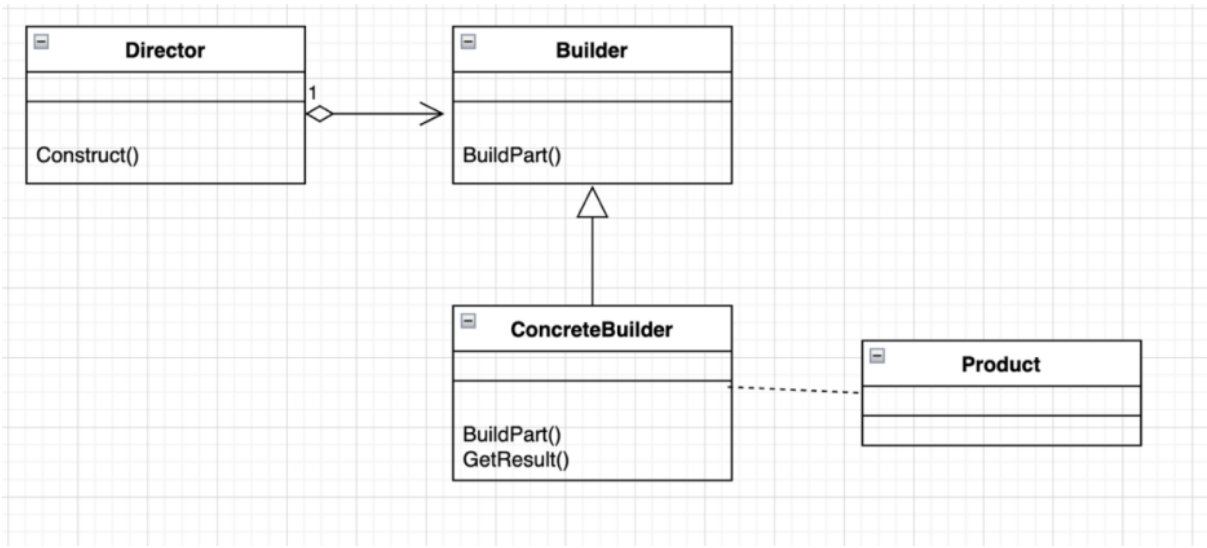
4. Яка різниця між реалізацією «Адаптера» на рівні об’єктів та на рівні класів?

- Об’єктний адаптер - використовує композицію (посилання на Adaptee).
- Класовий адаптер - використовує наслідування від Adaptee.

5. Яке призначення шаблону «Будівельник»?

Відокремлює процес створення складного об’єкта від його представлення.

6. Нарисуйте структуру шаблону «Будівельник».



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

• Builder - абстрактний інтерфейс для створення частин продукту.

• ConcreteBuilder - конкретна реалізація, що створює об’єкт.

• Director - керує процесом побудови, використовуючи Builder.

• Product - кінцевий об’єкт, який створюється.

Взаємодія:

Director викликає методи Builder у певній послідовності: ConcreteBuilder формує

Product - результат повертається клієнту.

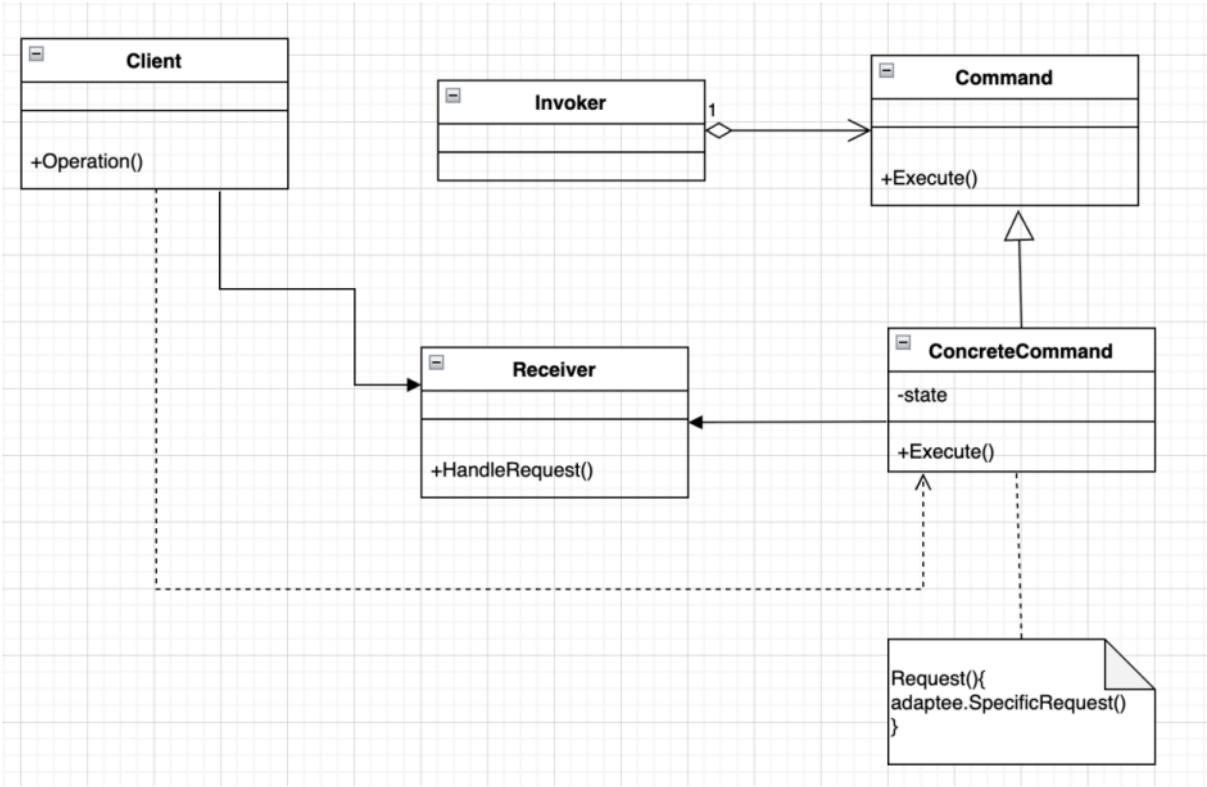
8. У яких випадках варто застосовувати шаблон «Будівельник»?

Коли об’єкт створюється поетапно або має багато варіантів конфігурації.

9. Яке призначення шаблону «Команда»?

Інкапсулює дію в окремий об’єкт, відокремлюючи ініціатора дії від її виконавця.

10. Нарисуйте структуру шаблону «Команда».



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

- Command - інтерфейс із методом execute().

• ConcreteCommand - конкретна команда, що виконує дію через Receiver.

• Receiver - виконує реальну роботу.

• Invoker - ініціює виконання команди.

• Client - створює команди та передає їх виконавцю.

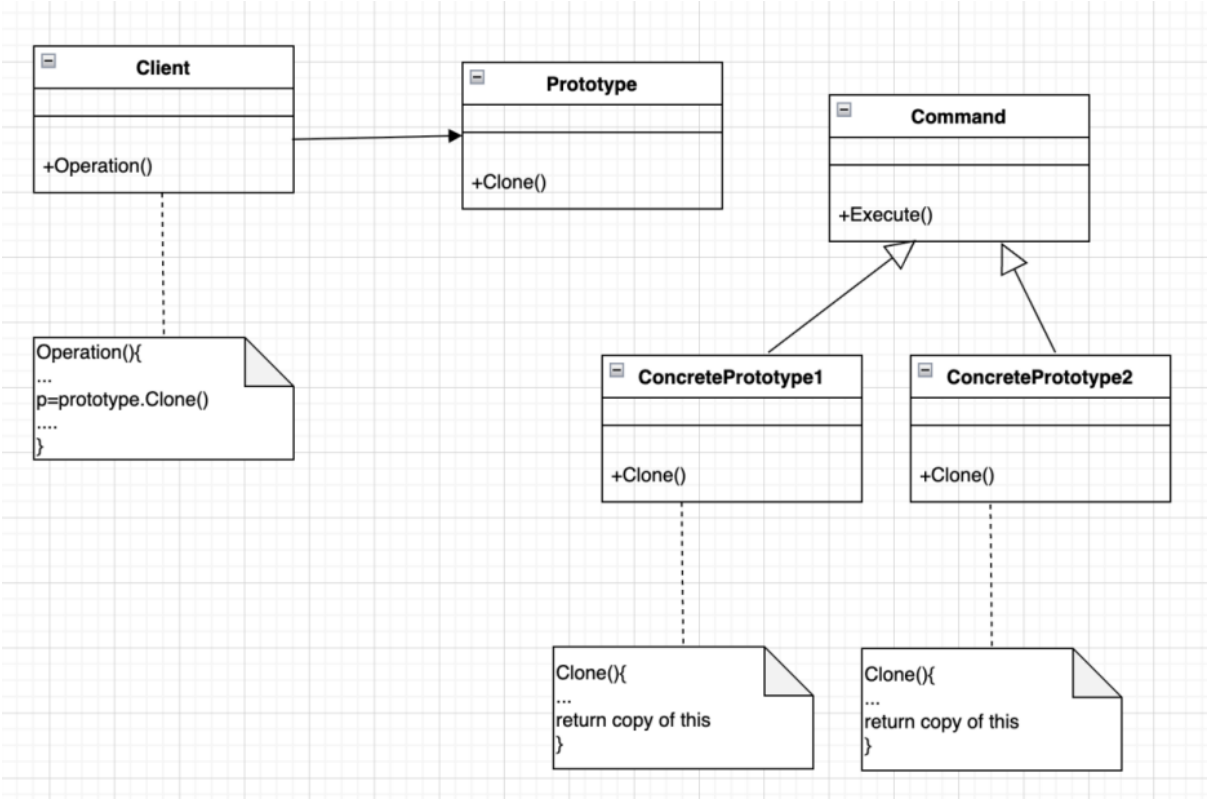
12. Розкажіть як працює шаблон «Команда».

Invoker викликає метод команди, команда передає запит об’єкту Receiver, який виконує потрібну дію.

13. Яке призначення шаблону «Прототип»?

Дозволяє створювати нові об’єкти шляхом копіювання існуючих зразків.

14. Нарисуйте структуру шаблону «Прототип».



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

• Prototype - оголошує метод clone().

• ConcretePrototype - реалізує клонування об’єкта.

• Client - створює нові об’єкти через копіювання прототипів.

Взаємодія:

Client викликає метод clone() у Prototype, отримуючи новий екземпляр об’єкта з таким самим станом.

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

- Обробка подій у графічному інтерфейсі (GUI).
- Система логування з різними рівнями (info, warning, error).
- Перевірка прав доступу (запит проходить кілька рівнів перевірки).
- Фільтрація запитів у веб-додатках.