



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
Тема: «Патерни проектування»

Виконала
студентка групи ІА-32:
Іванова Анастасія
Юріївна

Перевірив:
Мягкий Михайло
Юрійович

Зміст

.....	1
-------	---

Тема проекту.....	3
Теоретичні відомості.....	3
Хід роботи.....	4
Створення діаграми класів патерну.....	4
Висновки.....	7
Контрольні питання.....	8

Тема проекту

Варіант: 26

Опис теми: 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 браузері (firefox, opera, internet explorer, chrome).

Теоретичні відомості

Патерни проєктування (Design Patterns) — це типові, перевірені на практиці способи розв’язання поширених проблем, які виникають під час проєктування програмного забезпечення.

Вони описують загальні принципи побудови структури класів і взаємодії об’єктів, не прив’язуючись до конкретної мови програмування. Патерни не є готовим кодом — це шаблони, за якими можна побудувати гнучке, масштабоване й зрозуміле рішення.

Використання патернів дозволяє:

- підвищити гнучкість і розширюваність системи;
- зменшити дублювання коду й спростити супровід;
- покращити зрозумілість архітектури для інших розробників;
- забезпечити повторне використання перевірених рішень у різних проєктах.

Патерн **Command (Команда)** перетворює запит або дію у самостійний об’єкт. Це дозволяє відокремити об’єкт, який ініціює дію, від об’єкта, який цю дію виконує. Такий підхід дає можливість створювати гнучку систему команд, додавати функціональність скасування (undo), логування, відкладеного виконання або повторного виклику команд.

Ідея:

Замість безпосереднього виклику методу створюється клас-команда, який містить метод execute(). Цей об’єкт-команда зберігає всю необхідну інформацію для виконання операції — посилання на отримувача (receiver), параметри виклику тощо.

Типова структура патерну:

- **Command** — інтерфейс із методом execute().
- **ConcreteCommand** — конкретна реалізація команди, яка викликає методи отримувача.
- **Receiver** — об’єкт, який безпосередньо виконує дію.
- **Invoker** — зберігає посилання на команду та ініціює її виконання.
- **Client** — створює об’єкти команд і прив’язує їх до отримувачів.

Проблема, яку вирішує:

Патерн дозволяє уникнути дублювання коду при реалізації багатьох подібних дій (наприклад, обробників подій у графічному інтерфейсі), забезпечує можливість централізованого керування командами, а також спрощує тестування, оскільки логіка виконання відокремлена від користувацького інтерфейсу.

Переваги:

- Відокремлює виклик команди від її реалізації.
- Підтримує скасування (undo) і повторення (redo) дій.

- Дозволяє логувати та виконувати команди у потрібний час.
- Легко розширюється — додавання нових команд не потребує змін у наявному коді.

Недоліки:

– Може збільшити кількість класів у проєкті.

Хід роботи

Створення діаграми класів патерну

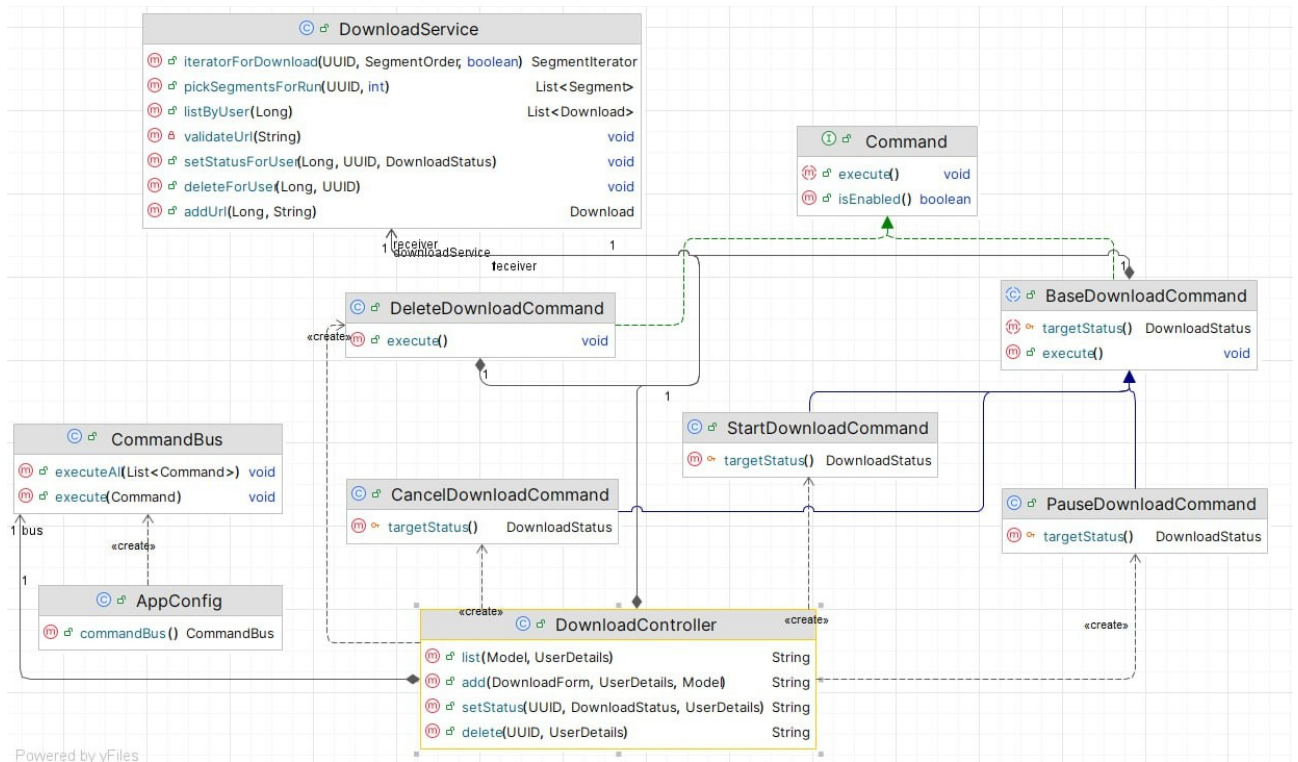


Рис. 1. Діаграма класів патерну Command

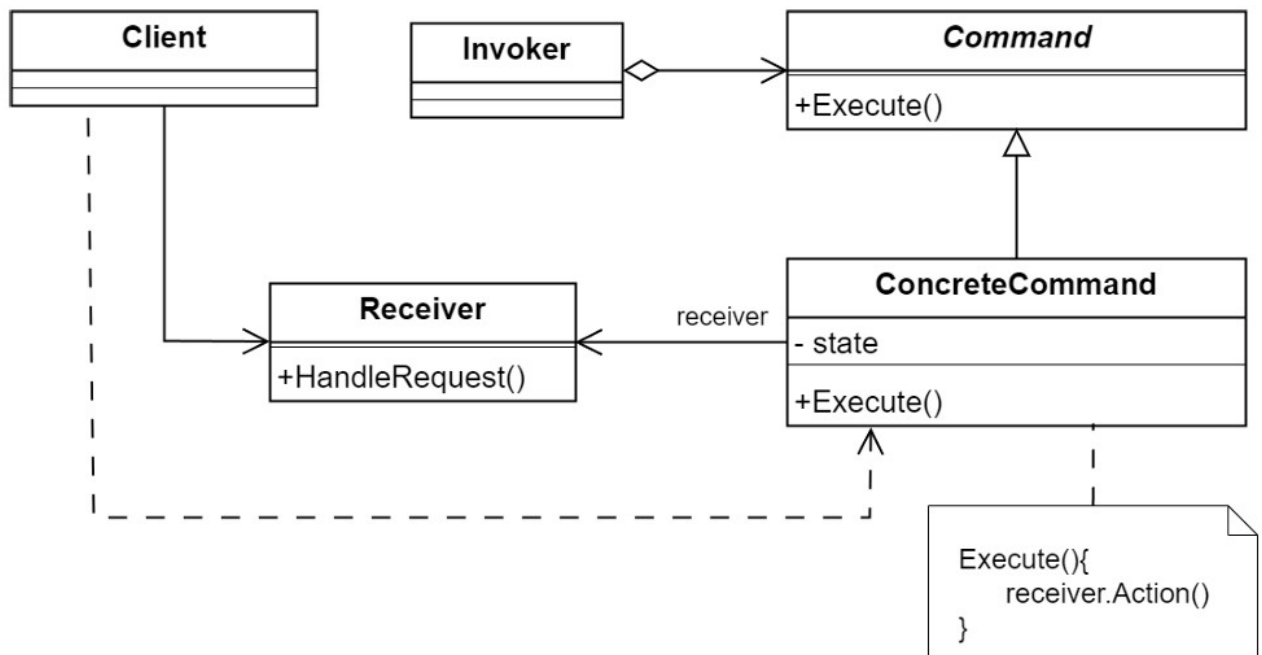


Рис. 2. Шаблон Command

1. Command → Command

Задає єдиний інтерфейс для всіх команд, які можна виконати.

Фрагмент коду:

```

public interface Command {
    void execute();
    default boolean isEnabled() { return true; }
}
  
```

Роль:

Визначає єдиний метод execute(), який реалізують усі конкретні команди.

Завдяки цьому всі дії (Start, Pause, Cancel, Delete) виконуються однаково через CommandBus.

2. ConcreteCommand → StartDownloadCommand, PauseDownloadCommand, CancelDownloadCommand, DeleteDownloadCommand

Конкретні команди, які інкапсулюють виклик певної дії (наприклад, змінити статус завантаження).

Це класи, що мають стан (-state) і викликають методи отримувача (receiver.Action()).

Фрагмент коду(для StartDownloadCommand):

```

public class StartDownloadCommand extends BaseDownloadCommand {
    public StartDownloadCommand(DownloadService s, Long userId, UUID id) {
        super(s, userId, id);
    }
    @Override protected DownloadStatus targetStatus() { return
DownloadStatus.RUNNING; }
}
  
```

Роль:

Кожна команда містить посилання на отримувача (DownloadService) і необхідні параметри (userId, downloadId).

Метод execute() виконує потрібну дію, наприклад:

```
receiver.setStatusForUser(userId, downloadId, DownloadStatus.RUNNING);
```

Таким чином, логіка виклику відокремлена від ініціатора дії (контролера).

3. Receiver → DownloadService

Виконує фактичні операції — зміну статусу, видалення, збереження тощо.

Фрагмент коду:

```
@Transactional
```

```
public void setStatusForUser(Long userId, UUID downloadId, DownloadStatus status) {
```

```
    var d = downloads.findByIdAndOwner_Id(downloadId, userId)
        .orElseThrow(() -> new IllegalArgumentException("Завантаження не знайдено"));
```

```
    d.setStatus(status);
```

```
    d.setUpdatedAt(Instant.now());
```

```
    downloads.save(d);
```

```
}
```

Роль:

DownloadService є отримувачем усіх команд і виконує реальні зміни у системі.

Команди лише передають сюди запит — вони не знають, як саме змінюються дані.

4. Invoker → CommandBus

Об'єкт, який отримує команди та виконує їх - саме той, хто викликає метод Execute() у команді.

Фрагмент коду:

```
public class CommandBus {
```

```
    public void execute(Command cmd) {
```

```
        if (cmd == null || !cmd.isEnabled()) return;
```

```
        cmd.execute();
```

```
    }
```

```
}
```

Роль:

Приймає будь-яку команду, що реалізує Command.

Викликає її метод execute() — без знання, яку саме дію вона виконує.

5. Client → DownloadController

Створює команди та відправляє їх до Invoker. Це об'єкт, який ініціює запит користувача.

Фрагмент коду:

```
@PostMapping("/{id}/status")
public String setStatus(@PathVariable UUID id,
                        @RequestParam("s") DownloadStatus status,
                        @AuthenticationPrincipal UserDetails auth) {
    var user = users.findByUsername(auth.getUsername()).orElseThrow();
    switch (status) {
        case RUNNING -> bus.execute(new
StartDownloadCommand(downloadService, user.getId(), id));
        case PAUSED -> bus.execute(new
PauseDownloadCommand(downloadService, user.getId(), id));
        case CANCELED-> bus.execute(new
CancelDownloadCommand(downloadService, user.getId(), id));
        default -> downloadService.setStatusForUser(user.getId(), id, status);
    }
    return "redirect:/downloads";
}
```

Роль:

Створює конкретні об'єкти команд залежно від дії користувача. Надсилає ці команди в CommandBus, який викликає execute().

Висновки

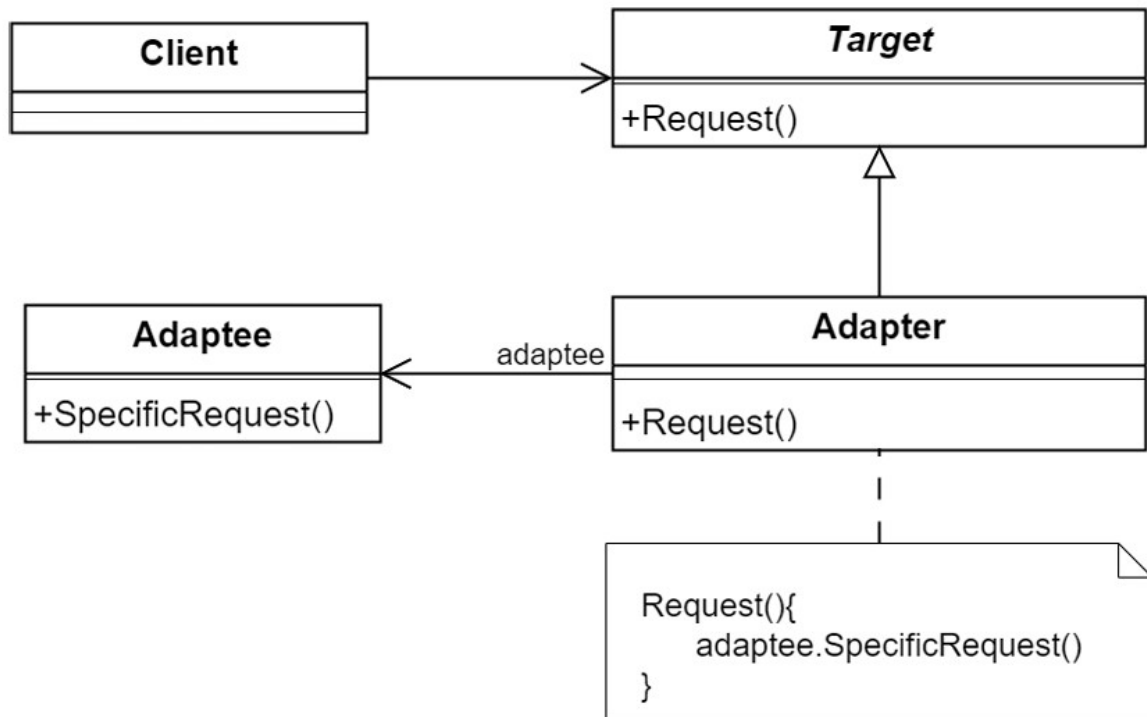
У ході виконання лабораторної роботи було розглянуто призначення та принцип роботи патерну Command (Команда). Цей шаблон дозволяє інкапсулювати дію у вигляді об'єкта, що дає можливість відокремити об'єкт, який ініціює запит, від об'єкта, який його виконує. У моєму застосунку Command застосовується для чіткого розділення логіки виконання дій над завантаженнями (старт, пауза, скасування, видалення) від ініціатора цих дій. Він дозволяє інкапсулювати кожну операцію в окремий клас-команду, що спрощує розширення функціоналу, уніфікує виконання різних дій через один механізм (CommandBus) та забезпечує чистішу архітектуру контролерів.

Контрольні питання

1. Яке призначення шаблону «Адаптер»?

Патерн Адаптер (Adapter) дозволяє узгодити інтерфейси двох несумісних класів. Він виступає проміжною ланкою між ними, забезпечуючи сумісність без зміни вихідного коду.

2. Нарисуйте структуру шаблону «Адаптер».



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

- **Target** — інтерфейс, який очікує клієнт.
- **Adapter** — клас-посередник, що реалізує інтерфейс **Target** і викликає методи іншого класу.
- **Adaptee** — клас із несумісним інтерфейсом, який потрібно адаптувати.
- **Client** — працює з об'єктами через інтерфейс **Target**.
Adapter перетворює виклики клієнта у формат, зрозумілий Adaptee.

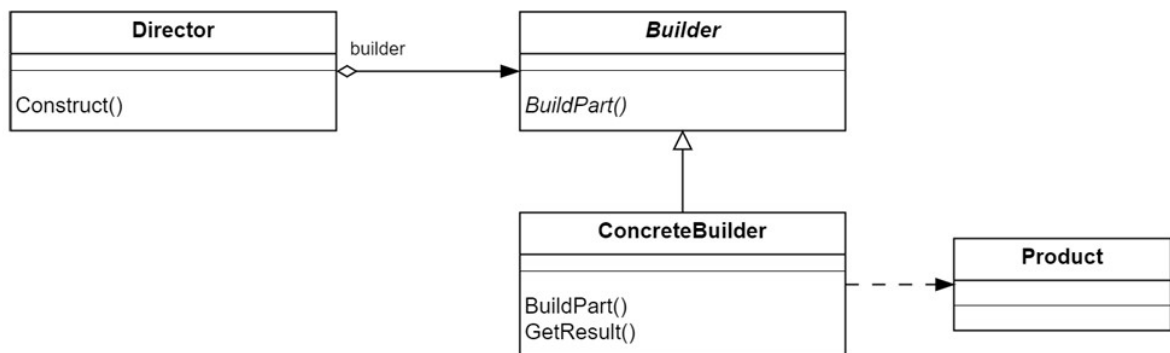
4. Яка різниця між реалізацією «Адаптера» на рівні об'єктів та на рівні класів?

- На рівні об'єктів: адаптер містить посилання на об'єкт **Adaptee** і делегує йому виклики (композиція).
- На рівні класів: адаптер наслідує як інтерфейс **Target**, так і клас **Adaptee** (множинне наслідування).

5. Яке призначення шаблону «Будівельник»?

Патерн Будівельник (Builder) відокремлює конструювання складного об'єкта від його представлення, що дозволяє створювати різні варіації об'єкта, не змінюючи його структуру.

6. Нарисуйте структуру шаблону «Будівельник».



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

- **Builder** — інтерфейс, який визначає кроки побудови.
- **ConcreteBuilder** — реалізує кроки побудови конкретного продукту.
- **Director** — керує процесом побудови, викликаючи методи **Builder** у певній послідовності.
- **Product** — кінцевий об'єкт, який створюється.

Director координує роботу **Builder** для створення об'єкта **Product**.

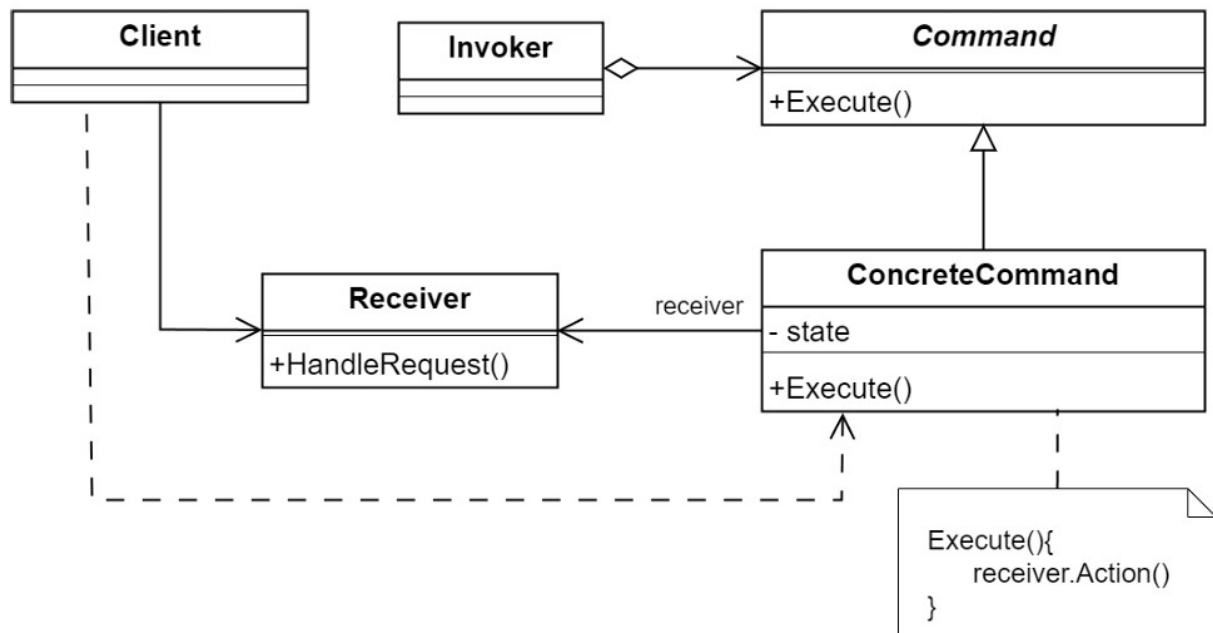
8. У яких випадках варто застосовувати шаблон «Будівельник»?

Коли необхідно створювати складні об'єкти покроково або мати різні представлення одного й того ж об'єкта, не змінюючи процес побудови.

9. Яке призначення шаблону «Команда»?

Патерн Команда (Command) інкапсулює запит у вигляді об'єкта, дозволяючи зберігати, відкладати, відмінити або повторно виконувати дії незалежно від об'єкта, що їх ініціює.

10. Нарисуйте структуру шаблону «Команда».



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

- Command — інтерфейс із методом execute().
- ConcreteCommand — реалізує команду, викликаючи методи отримувача (Receiver).
- Receiver — виконує реальну дію.
- Invoker — ініціює виконання команди.
- Client — створює команду та задає отримувача.
Invoker викликає команду, а ConcreteCommand делегує її виконання Receiver.

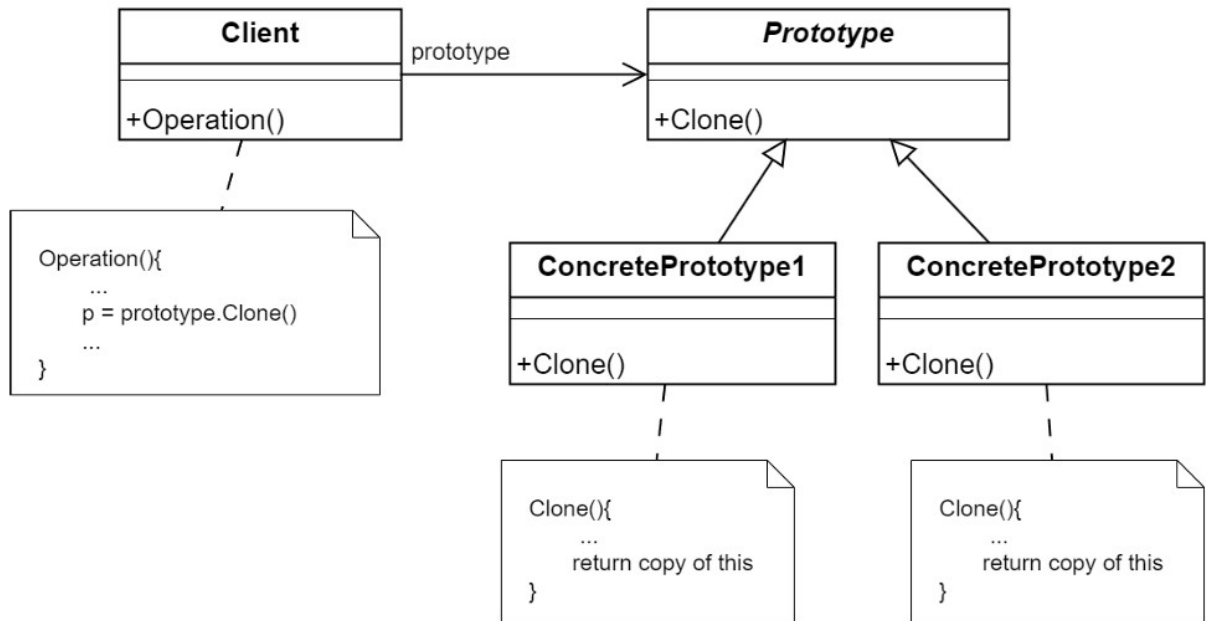
12. Розкажіть, як працює шаблон «Команда».

Клієнт створює об'єкт-команду і зв'язує його з отримувачем. Команда передається виконавцю (Invoker), який у потрібний момент викликає її метод execute(). Команда виконує дію через об'єкт Receiver. Такий підхід дозволяє легко додавати нові команди та реалізовувати функції скасування.

13. Яке призначення шаблону «Прототип»?

Патерн Прототип (Prototype) дозволяє створювати нові об'єкти шляхом копіювання вже існуючих замість створення через конструктор.

14. Нарисуйте структуру шаблону «Прототип».



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

- Prototype — інтерфейс із методом clone().
- ConcretePrototype — реалізує метод копіювання.
- Client — створює нові об'єкти, копіюючи існуючі екземпляри.
Клієнт отримує новий об'єкт, викликавши метод clone() у Prototype.

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

Приклади:

- обробка запитів у системах доступу (перевірка прав користувача, аутентифікація, логування);
- обробка подій у графічних інтерфейсах (передача події по ланцюгу елементів);
- системи підтримки (запит передається вгору, доки не знайдеться відповідальний обробник).