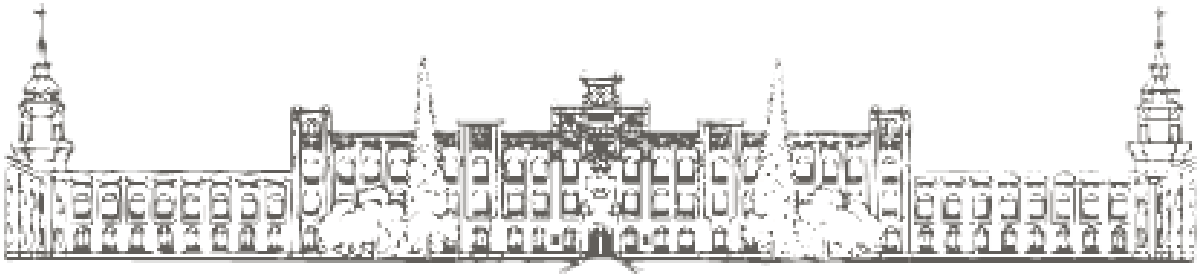


МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені ІГОРЯ СІКОРСЬКОГО»
КАФЕДРА ІНФОРМАЦІЙНИХ СИСТЕМ ТА ТЕХНОЛОГІЙ

Звіт лабораторної роботи №5

з курсу

# «Технології розроблення програмного забезпечення»

Виконавець:
Нікітченко Наталя Олегівна
студентка групи ІА-33
залікова книжка № ІА-3318

«14» 11 2025 р.

Перевірив: **Мягкий М. Ю.**

Київ – 2025

# 5. ЛАБОРАТОРНА РОБОТА № 5

**Тема:** Патерни проектування.

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

**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.

**Теоретичні відомості:**

**Шаблон «Prototype»**

Призначення патерну: Шаблон «Prototype» (Прототип) використовується для створення об'єктів за «шаблоном» (чи «кресленням», «ескізом») шляхом копіювання шаблонного об'єкту, який називається прототипом. Для цього визначається метод «клонувати» в об'єктах цього класу.

Цей шаблон зручно використати, коли заздалегідь відомо як виглядатиме кінцевий об'єкт (мінімізується кількість змін до об'єкту шляхом створення шаблону), а також для видалення необхідності створення об'єкту – створення відбувається за рахунок клонування, і самій програмі немає необхідності знати, як створювати об'єкт.

Також, це дозволяє маніпулювати об'єктами під час виконання програми шляхом настроювання відповідних прототипів; значно

зменшується ієрархія наслідування (оскільки в іншому випадку це були б не прототипи, а вкладені класи, що наслідують).

**Проблема:** Ви розробляєте редактор рівнів для 2D гри на основі спрайтів. В панелі інструментів ви маєте багато кнопок для різних елементів, які можна розташовувати на екрані, такі як сходи, стіни, підлога, оздоблення та інші. Ці елементи у вас об'єднані в ієрархію з базовим класом GameObject. Під кожен елемент можна зробити свій тип кнопки, але тоді ми отримаємо паралельну ієрархію кнопок і при додаванні нового типу ігрового об’єкту потрібно буде додавати і новий тип кнопки.

**Рішення:** Використовуючи патерн прототип, додаємо до базового об'єкта GameObject метод Clone(), а кнопки будуть зберігати посилання на об'єкт базового типу GameObject. При натисканні на кнопку об'єкт який потрібно добавити на ігровому полі отримуємо не створенням нового, а клонуванням прототипу, який прив'язаний до кнопки. Таким чином, коли ми будемо додавати 67 нові типи ігрових об'єктів, то логіка роботи з кнопками не буде змінюватися, тому що не має прив'язки до конкретних типів.

Також слід відзначити, що при копіюванні об'єкту, навіть приватні поля будуть скопійовані, тому що реалізація методу копіювання знаходиться в цьому класі, що копіюється і таким чином є доступ для копіювання і до відкритих і до закритих полів.

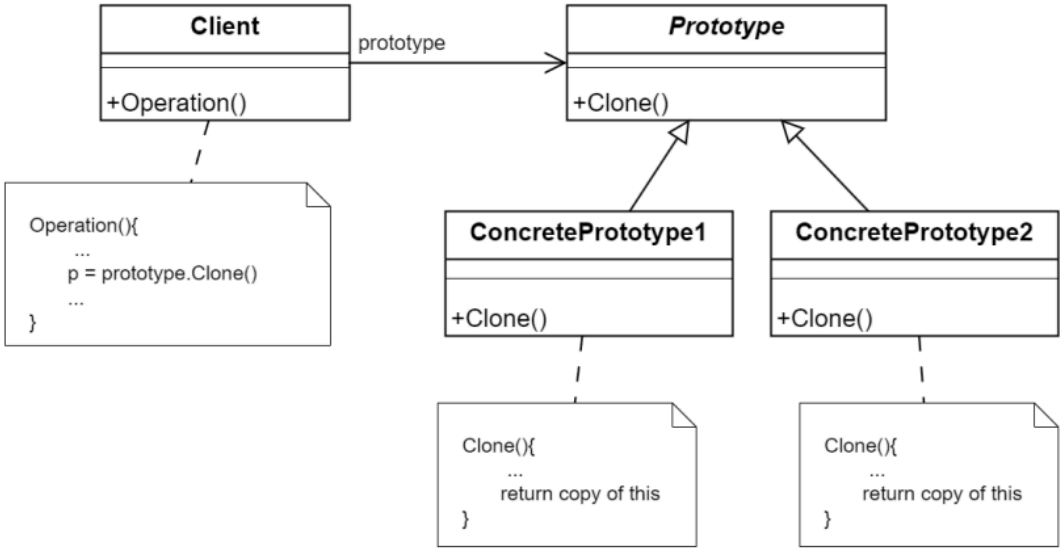



Рис.1 - Структура патерну «Прототип»

## Хід роботи:

**Тема:** “Графічний редактор”

Реалізувати один з розглянутих шаблонів за обраною темою, а саме патерн «Prototype».

Розглянемо структуру цього прототипу(Рис.2):

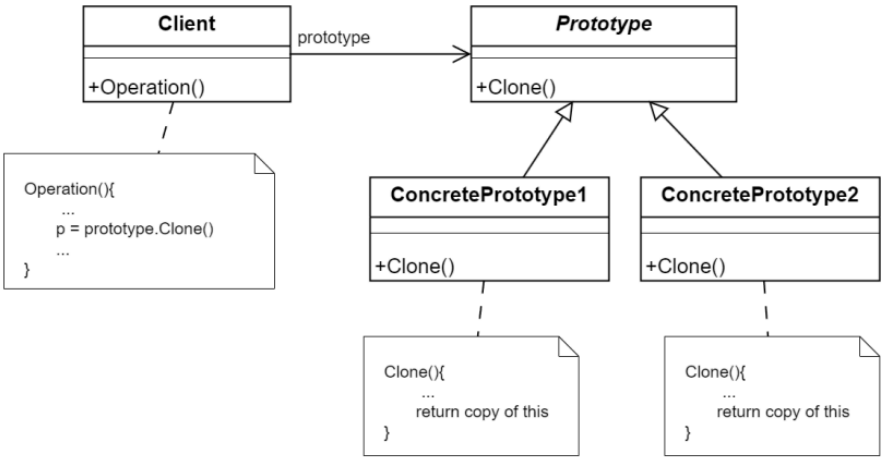



Рис.2 - Структура патерну «Прототип»

Реалізація цього протипу відповідно даної теми(Рис.3):

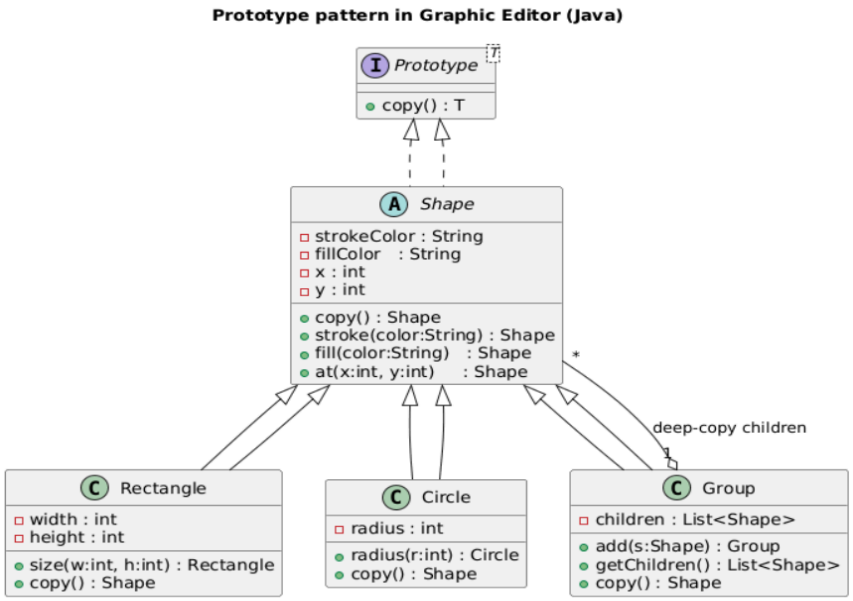



Рис.3- діаграму класів, яка представляє використання шаблону в реалізації системи

Діаграма показує, що:

- Усі фігури — це прототипи, які можуть клонувати самі себе.
- Прості фігури (Rectangle, Circle) копіюються легко — просто переносять власні поля.
- Складена фігура (Group) робить глибоке копіювання, копіюючи кожну дитину.
- Така модель дозволяє:
    - клонувати будь-яку фігуру чи групу,
    - створювати копії композицій,
    - дублювати елементи сцени як у Figma/Canva.

Фрагменти коду по реалізації цього шаблону:

```java
package com.example.graphiceditor.model.prototype;

public class Circle extends Shape {  6 usages  new *
    private int radius;  4 usages

    public Circle() {}  1 usage  new *

    public Circle(Circle other) {  1 usage  new *
        super(other);
        this.radius = other.radius;
    }

    public Circle radius(int r) {  no usages  new *
        this.radius = r;
        return this; }

    @Override  2 usages  new *
    public Shape copy() {
        return new Circle( other: this); }

    @Override  new *
    public String toString() {
        return super.toString() + " r=" + radius + " @" + id();
    }
}
```

Рис.4 - Circle

Клас Circle розширює Shape і додає поле radius. Має конструктор без параметрів та конструктор-копіювальник Circle(Circle other), у якому

викликається конструктор базового класу й копіюється радіус. Метод radius(int r) задає радіус і повертає поточний об’єкт для побудови ланцюга викликів. Метод copy() повертає новий об’єкт Circle, ініціалізований на основі поточного екземпляра (new Circle(this)), що реалізує прототипне копіювання. toString() доповнює базове текстове представлення інформацією про радіус та ідентифікатор.

```java
package com.example.graphiceditor.model.prototype;

import java.util.ArrayList;
import java.util.List;

public class Group extends Shape {  9 usages  new *
    private final List<Shape> children = new ArrayList<>();  5 usages

    public Group() {}  1 usage  new *

    public Group(Group other) {  1 usage  new *
        super(other);
        for (Shape child : other.children) {
            this.children.add(child.copy());
        }
    }

    public Group add(Shape s) { children.add(s); return this; }  new *
    public List<Shape> getChildren() { return children; }  2 usages  new *

    @Override  2 usages  new *
    public Shape copy() { return new Group( other: this); }

    @Override  new *
    public String toString() {
        return super.toString() + " children=" + children.size() + " @" + id();
    }
    @Override  4 usages  new *
    public Group at(int x, int y) {
        this.x = x;
        this.y = y;
        return this;
```

Рис.5 - Group

Клас Group також є підкласом Shape і представляє складену фігуру, що містить колекцію дочірніх фігур List<Shape> children. Конструктор за замовчуванням

створює порожню групу, а конструктор-копіювальник Group(Group other) викликає конструктор Shape(other) і формує **глибоку копію**: для кожної дочірньої фігури викликається child.copy(), і результат додається до нової колекції children. Методи add(Shape s) та getChildren() дозволяють додавати елементи та отримувати список дочірніх фігур. Метод copy() повертає новий екземпляр Group, створений на основі поточного (new Group(this)), що забезпечує глибоке копіювання всієї композиції. Метод toString() доповнює базове представлення інформацією про кількість дочірніх елементів.

```
package com.example.graphiceditor.model.prototype;

public interface Prototype<T> {  1 usage  4 implementations  new *
    T copy();  2 usages  4 implementations  new *
}
```

Рис.6 - Prototype<T>

Інтерфейс Prototype<T> задає єдиний метод T copy(), який визначає контракт на створення копії об’єкта. У ролі типового параметра використовується конкретний клас, що реалізує прототип.

```java
package com.example.graphiceditor.model.prototype;

public class Rectangle extends Shape {  5 usages  new *
    private int width;  4 usages
    private int height;  4 usages

    public Rectangle() {}  1 usage  new *

    public Rectangle(Rectangle other) {  1 usage  new *
        super(other);
        this.width = other.width;
        this.height = other.height;
    }

    public Rectangle size(int w, int h) {  new *
        this.width = w;
        this.height = h;
        return this;
    }

    @Override  2 usages  new *
    public Shape copy() {
        return new Rectangle( other: this);
    }

    @Override  new *
    public String toString() {
        return super.toString() + " w=" + width + " h=" + height + " @" + id();
    }
}
```

Рис.7 - Rectangle

Пояснення коду Rectangle(Рис.7):

- Rectangle — нащадок Shape, додає ширину + висоту.
- Конструктор Rectangle(other) — копіює інші властивості (позиція, кольори) + копіює власні поля.
- size(w, h) — змінює розмір; повертає *this* для chain-викликів.
- copy() — створює повний (deep) клон нового Rectangle.
- toString() — для зручного виводу.

Клас Rectangle розширює Shape і містить поля width та height. Конструктор-копіювальник Rectangle(Rectangle other) переносить ширину і висоту з вихідного прямокутника. Метод size(int w, int h) задає розміри та повертає поточний об’єкт. Метод copy() створює новий прямокутник на основі поточного (new Rectangle(this)), а toString() додає інформацію про ширину, висоту й ідентифікатор.

```java
package com.example.graphiceditor.model.prototype;

public abstract class Shape implements Prototype<Shape> {  18 usages  3 inheritors  new *
    protected String strokeColor;  4 usages
    protected String fillColor;  4 usages
    protected int x;
    protected int y;

    protected Shape() {}  3 usages  new *

    protected Shape(Shape other) {  3 usages  new *
        this.strokeColor = other.strokeColor;
        this.fillColor = other.fillColor;
        this.x = other.x;
        this.y = other.y;
    }

    public abstract Shape copy();  2 usages  3 implementations  new *

    public Shape stroke(String color) {  2 usages  new *
        this.strokeColor = color;
        return this; }
    public Shape fill(String color) {  2 usages  new *
        this.fillColor = color;
        return this; }
    public Shape at(int x, int y) {  4 usages  1 override  new *
        this.x = x; this.y = y;
        return this; }
```

Рис.8 - Shape

```java
    @Override  3 overrides  new *
    public String toString() {
        return getClass().getSimpleName()
                + "{x=" + x + ", y=" + y
                + ", stroke=" + strokeColor
                + ", fill=" + fillColor + "}";
    }

    public String id() { return Integer.toHexString(System.identityHashCode( x: this)); }  new *
}
```

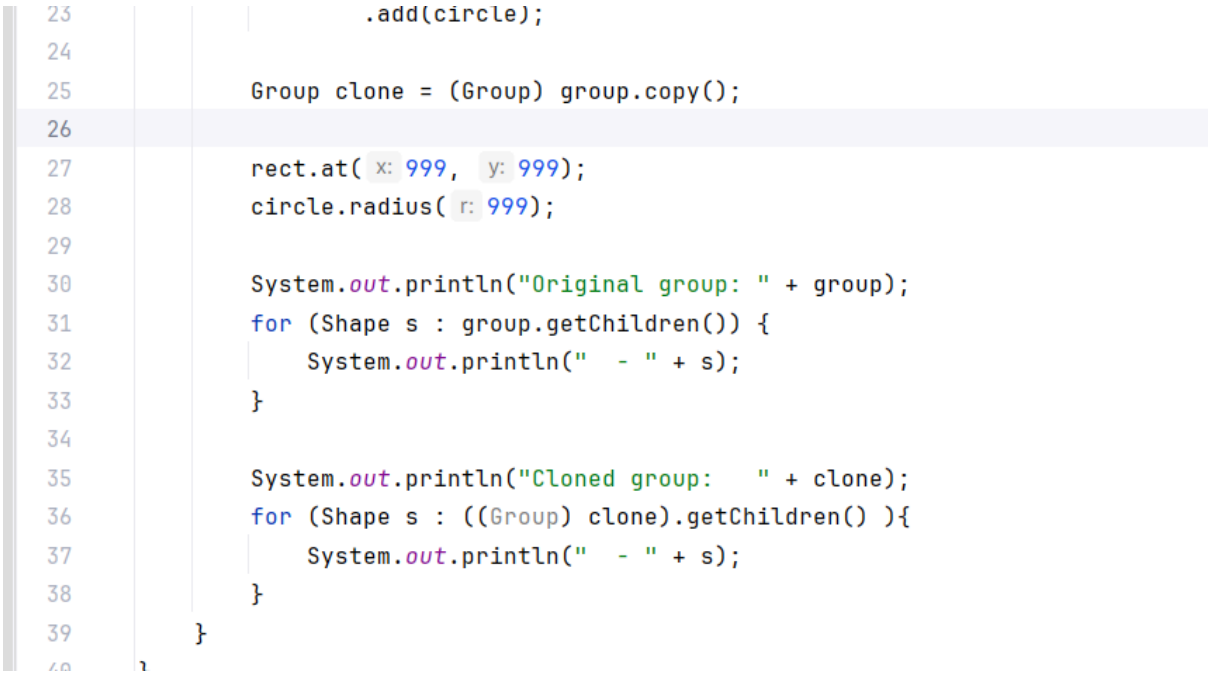
Рис.9 - Shape

Абстрактний клас Shape (пакет com.example.graphiceditor.model.prototype) реалізує інтерфейс Prototype<Shape> і є базовою фігурою для всіх

графічних примітивів. Клас зберігає спільні атрибути: координати x, y, колір контуру strokeColor та колір заливки fillColor. Конструктор-копіювальник Shape(Shape other) переносить ці значення з вихідного об’єкта в новий екземпляр. Метод copy() оголошений як абстрактний і реалізується в підкласах. Додатково реалізовано «ланцюгові» методи налаштування stroke(String), fill(String) та at(int x, int y), що повертають this, а також допоміжні методи toString() (текстове представлення фігури) та id() (ідентифікатор об’єкта на основі identityHashCode).

```
1   package com.example.graphiceditor.ui;
2
3   import com.example.graphiceditor.model.prototype.*;
4
5   public class DemoPrototype {  new *
6       public static void main(String[] args) {  new *
7
8           Rectangle rect = new Rectangle();
9           rect.at( x: 10,  y: 20)
10                  .stroke( color: "#000")
11                  .fill( color: "#EEE");
12          rect.size( w: 100,  h: 60);
13
14          Circle circle = (Circle) new Circle();
15          circle.at( x: 80,  y: 40)
16                  .stroke( color: "#F00")
17                  .fill( color: "#FCC");
18          circle.radius( r: 30);
19
20          Group group = new Group()
21                  .at( x: 0,  y: 0)
22                  .add(rect)
23                  .add(circle);
24
25          Group clone = (Group) group.copy();
26
27          rect.at( x: 999,  y: 999);
28          circle.radius( r: 999);
```

Рис.10 - DemoPrototype



```
23                .add(circle);
24
25            Group clone = (Group) group.copy();
26
27            rect.at( x: 999,  y: 999);
28            circle.radius( r: 999);
29
30            System.out.println("Original group: " + group);
31            for (Shape s : group.getChildren()) {
32                System.out.println("  - " + s);
33            }
34
35            System.out.println("Cloned group:   " + clone);
36            for (Shape s : ((Group) clone).getChildren() ){
37                System.out.println("  - " + s);
38            }
39        }
```

Рис.11 - DemoPrototype

Пояснення коду DemoPrototype(Рис.10-11):

**Створюються два об’єкти-фігури** — прямокутник і коло. Їм задаються координати, кольори та розміри.

**Створюється група (Group)**, яка працює як складена фігура. До неї додаються обидві створені фігури.

**Виконується копіювання групи** через метод copy().

Група використовує глибоке копіювання: усередині вона створює нові копії кожної фігури, а не посилається на старі.

**Після копіювання змінюються дані в оригінальних фігурах**, щоб продемонструвати незалежність копії.

**Програма виводить у консоль стан оригіналу та копії**.

Оригінал містить змінені фігури, а копія зберігає старі значення — це підтверджує, що копіювання було глибоким і об’єкти не пов’язані.

Клас DemoPrototype містить метод main(String[] args) і демонструє роботу шаблону **Prototype**. У програмі створюються та налаштовуються примітиви

Rectangle і Circle за допомогою ланцюгових викликів (at(...), stroke(...), fill(...), size(...), radius(...)), після чого вони додаються до групи Group. Далі формується клон групи через виклик group.copy(). Після створення клону вихідні фігури модифікуються (змінюються координати прямокутника та радіус кола), а потім у консоль виводяться дані про «оригінальну» та «клоновану» групи разом із їхніми дочірніми елементами. Це демонструє, що зміни в оригінальних об’єктах не впливають на клон, тобто реалізовано **глибоке копіювання** згідно з шаблоном Prototype.

**Висновки:**

У реалізації прототипу для графічного редактора було побудовано цілісну і узгоджену ієрархію класів, що повністю відповідає діаграмі класів патерну *Prototype*. Інтерфейс Prototype<T> задає єдиний контракт copy(), а абстрактний клас Shape реалізує цей контракт на рівні моделі фігури: інкапсулює спільні атрибути (x, y, strokeColor, fillColor) і надає конструктор-копіювальник Shape(Shape other). Таким чином логіка копіювання базових властивостей винесена в один центр, що спрощує розширення ієрархії новими видами фігур.

Конкретні класи Rectangle і Circle демонструють класичне використання патерну: кожен має власний конструктор-копіювальник (Rectangle(Rectangle other), Circle(Circle other)), у якому спочатку викликається super(other) для копіювання спільних полів, а потім дублюються специфічні атрибути (width, height, radius). Метод copy() у цих класах просто створює новий екземпляр через відповідний конструктор – return new Rectangle(this) / new Circle(this). Це дозволяє створювати повністю незалежні копії об’єктів без дублювання коду, а fluent-методи (at(...), stroke(...), fill(...), size(...), radius(...)) забезпечують зручне налаштування екземплярів у стилі графічних редакторів.

Особливу роль відіграє клас Group, який представляє складену фігуру (композицію). Він містить список дочірніх об’єктів List<Shape> children і в конструкторі-копіювальнику проходить по other.children, створюючи копії кожної дитини через child.copy(). Таким чином реалізовано глибоке копіювання: клон групи містить не ті самі посилання, а нові екземпляри фігур. Це принципово важливо для графічного редактора: зміни, внесені в оригінальну групу (переміщення, зміна розмірів чи кольорів), не впливають на вже створені копії, що підтверджується поведінкою демо-класу DemoPrototype.

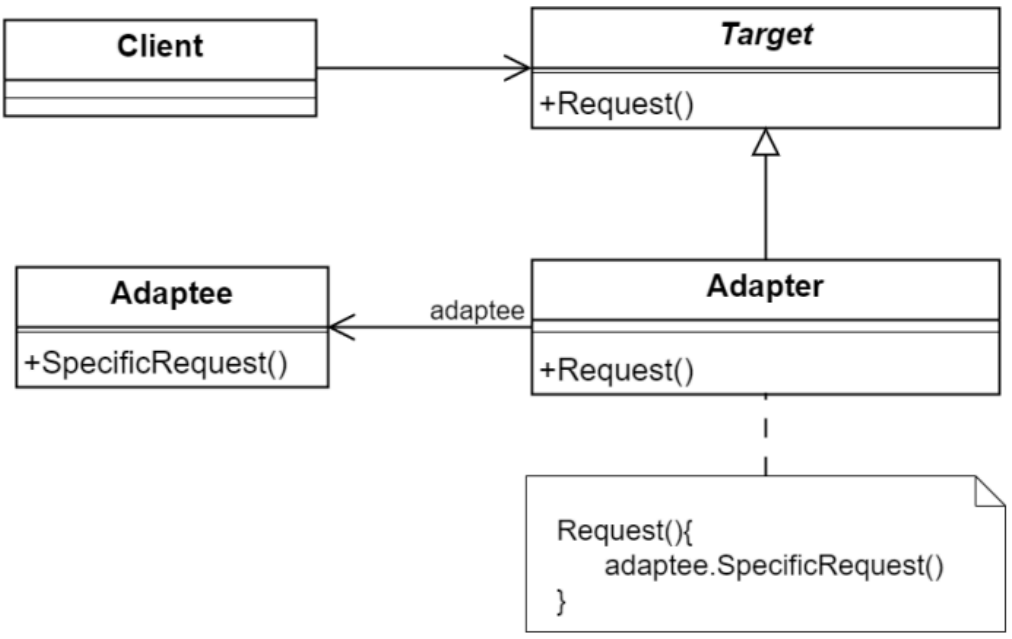
**Контрольні питання:**

**1.Яке призначення шаблону «Адаптер»?**

Адаптер перетворює інтерфейс одного класу в інший, очікуваний клієнтом.

Дає змогу використати готовий клас, який «майже підходить», але має інший інтерфейс.

**2. Нарисуйте структуру шаблону «Адаптер».**




Умовна UML-структура (object adapter):

- Client → працює з інтерфейсом Target
- Target → очікуваний клієнтом інтерфейс
- Adapter implements Target → всередині має посилання на Adaptee
- Adaptee → існуючий клас з «незручним» інтерфейсом

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

**Client** – викликає методи Target.

**Target** – описує інтерфейс, який потрібен клієнту.

**Adaptee** – реальний/старий/чужий клас з іншим інтерфейсом.

**Adapter** – реалізує Target і всередині викликає методи Adaptee, перетворюючи параметри/результати.

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

**На рівні об'єктів (object adapter):**

Adapter тримає посилання на екземпляр Adaptee (композиція).

Гнучкий, можна підміняти Adaptee під час роботи.

**На рівні класів (class adapter):**

Adapter *наслідує* Adaptee і *реалізує* Target одночасно.

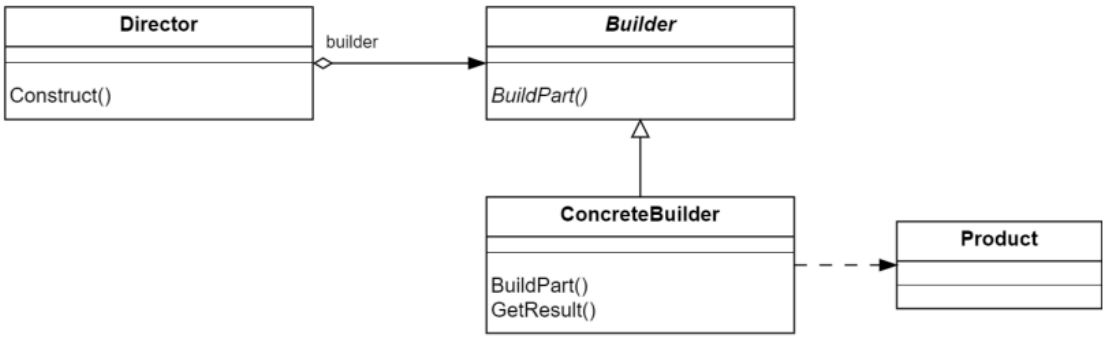
Потрібне множинне наслідування (є не у всіх мовах). Менш гнучкий, але іноді простіший.

**5. Яке призначення шаблону «Будівельник»?**

Розділити процес конструювання складного об'єкта від його представлення.

Дозволяє крок за кроком будувати об'єкт з різними конфігураціями, не створюючи гігантських конструкторів.

**6. Нарисуйте структуру шаблону «Будівельник».**




**Типова UML-структура:**

- Director
- Builder (інтерфейс)
- ConcreteBuilder
- Product

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

Product – складний об’єкт, який створюємо.

Builder – інтерфейс з методами buildPartX(), buildPartY(), getResult().

ConcreteBuilder – реалізує кроки побудови і зберігає проміжний стан Product.

Director – знає порядок виклику кроків (алгоритм складання), але не знає деталей реалізації.

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

Коли об’єкт має багато полів / частин, і конструктори стають незручними.

Коли потрібно підтримувати різні представлення одного продукту (різні варіанти комплектації).

Коли треба мати зрозумілий покроковий процес створення (наприклад, генерація складного документа, UI-вікна, конфігуратора).

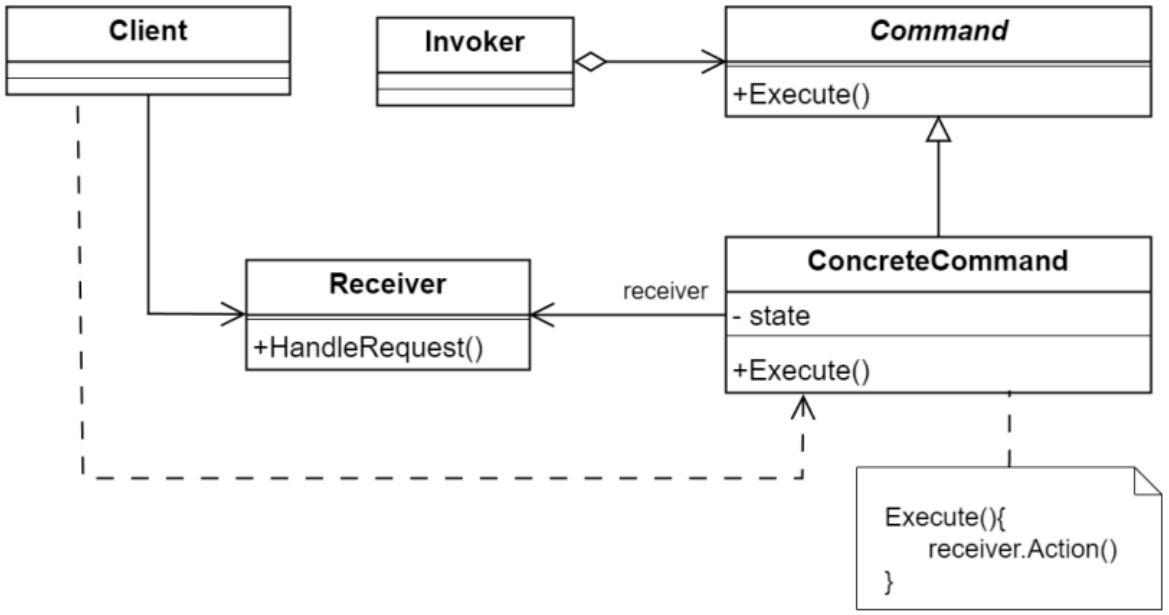
**9. Яке призначення шаблону «Команда»?**

Інкапсулювати запит (дію) як об’єкт.

Призначення:

- передавати дії як об’єкти;
- ставити команди в чергу;
- логувати виконання;
- легко реалізувати **Undo / Redo** (скасування та повтор);
- відділити **клієнта**, який ініціює дію, від **виконавця**, який її виконує.

**10. Нарисуйте структуру шаблону «Команда».**




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

**Command (інтерфейс):** містить метод execute(),іноді undo().

**ConcreteCommand:** реалізує execute(), зберігає посилання на Receiver, викликає метод Receiver.

**Receiver:** реальний виконавець дії (наприклад: файл, документ, світильник, персонаж у грі).

**Invoker:** запускає команду (button.click() → command.execute()).

**Client:** створює конкретні команди та задає, хто їх виконує.

**12. Розкажіть як працює шаблон «Команда».**

Client створює об’єкт-команду, наприклад:

CopyCommand

PasteCommand

DeleteCommand

Кожна команда містить посилання на Receiver, який уміє виконати дію.

Invoker (кнопка, меню, гаряча клавіша) викликає command.execute().

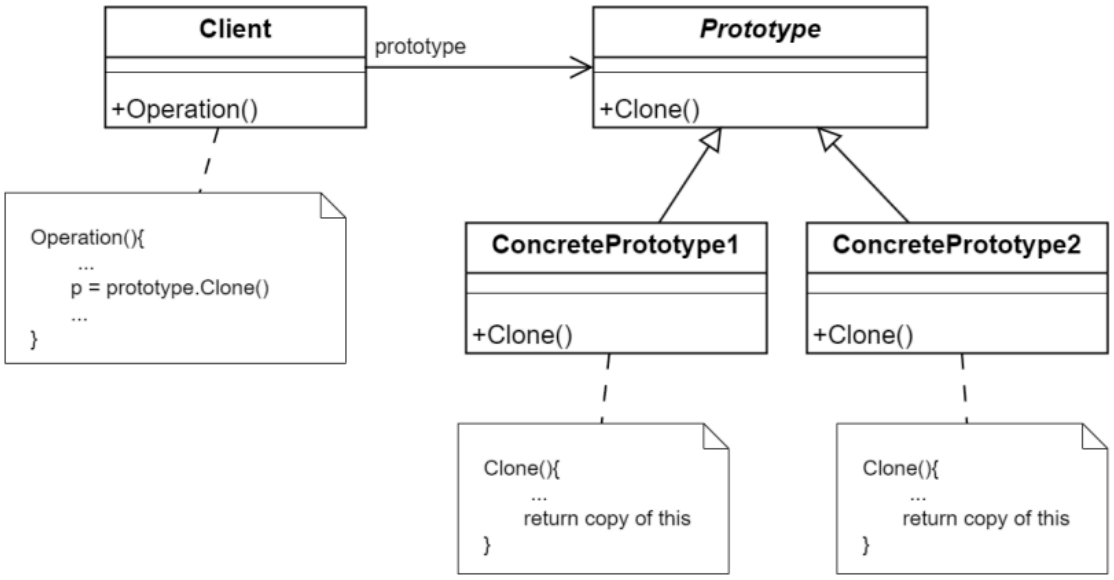
Команда делегує виконання Receiver.

Якщо потрібно підтримати Undo, команда зберігає попередній стан.

**13. Яке призначення шаблону «Прототип»?**

Шаблон «Prototype» (Прототип) використовується для створення об'єктів за «шаблоном» (чи «кресленням», «ескізом») шляхом копіювання шаблонного об'єкту, який називається прототипом.

**14. Нарисуйте структуру шаблону «Прототип».**




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

Класи входять в шаблон «Прототип»:

- Prototype;
- ConcretePrototype1 / ConcretePrototype2;
- Client;

Взаємодія між ними:

**Client** отримує посилання на об’єкт типу **Prototype**.

Коли потрібно створити новий об’єкт, Client викликає: prototype.Clone() **.**

Викликається метод Clone() конкретного класу (**ConcretePrototype1** або **ConcretePrototype2**).

Конкретний прототип повертає нову копію самого себе.

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

Перевірка даних (валідація)

Ланцюг перевірок:

- поле не пусте
- правильний формат
- довжина
- унікальність у БД