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 розробки програмного забезпечення»

Паттерни програмування

Виконав:
студент групи ІА-34
Швець Роман Вадимович

Перевірив:
асистент кафедри ІСТ
Мягкий Михайло Юрійович

Київ - 2025

Тема: Паттерни програмування

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

4. **Графічний редактор** (proxy, prototype, decorator, bridge, flyweight, SOA)
Графічний редактор повинен вміти створювати / редагувати растрові (або векторні на розсуд студента) зображення в 2-3 основних популярних форматах (bmp, png, jpg), мати панель інструментів для створення графічних примітивів, вибору кольорів, нанесення тексту, додавання найпростіших візуальних ефектів (ч/б растр, інфрачервоний растр, 2-3 на вибір учня), роботи з шарами.

Теоретичні відомості

Шаблони проєктування (патерни) — це загальні рішення типових проблем, що виникають у процесі розроблення програмного забезпечення. Вони не є готовим кодом, а радше описують структуру взаємодії між класами та об’єктами, допомагаючи створювати гнучкі, масштабовані й легко підтримувані системи. Використання шаблонів дозволяє стандартизувати архітектурні рішення, полегшити командну розробку та уникнути повторення вже відомих помилок у проєктуванні.

Шаблон Adapter (Адаптер) використовується для узгодження несумісних інтерфейсів. Він дає змогу об’єктам із різними інтерфейсами взаємодіяти між собою, перетворюючи один інтерфейс на інший. Такий підхід часто застосовується для інтеграції сторонніх бібліотек або старих компонентів у нові системи. Адаптер виступає як проміжна ланка між клієнтом і адаптованим класом, забезпечуючи коректну передачу викликів методів.

Шаблон Builder (Будівельник) призначений для створення складних об’єктів із багатьма параметрами або етапами побудови. Його суть полягає в розділенні процесу створення об’єкта на окремі кроки, що дозволяє створювати різні варіації одного й того ж об’єкта, не змінюючи код його конструкторів. У типовій структурі

присутні директор, який керує процесом побудови, будівельник, що визначає етапи створення, та конкретний будівельник, який реалізує деталі побудови кінцевого продукту.

Шаблон Command (Команда) інкапсулює запит як окремий об’єкт, що дає змогу передавати команди як параметри, ставити їх у чергу, скасовувати або повторювати виконання. Це спрощує роботу з операціями, які потрібно відкладати або об’єднувати. Основними елементами цього шаблону є інтерфейс команди, конкретні класи команд, отримувач (об’єкт, що виконує дію) та ініціатор, який викликає команду. Такий підхід забезпечує незалежність між відправником і виконавцем дії.

Шаблон Chain of Responsibility (Ланцюг обов’язків) дозволяє передавати запит послідовно через ланцюг об’єктів, поки один із них не обробить його. Відправник запиту не знає, який саме об’єкт його виконає, що забезпечує слабке зв’язування між компонентами. Кожен обробник має посилання на наступного в ланцюгу, а клієнт лише ініціює процес. Це робить систему гнучкішою та зручною для розширення, адже можна легко додавати нові типи обробників.

Шаблон Prototype (Прототип) використовується для створення нових об’єктів шляхом копіювання вже існуючих екземплярів замість створення їх із нуля. Це особливо зручно, коли процес ініціалізації об’єкта є складним або ресурсоємним. Основна ідея полягає у створенні базового об’єкта-прототипу, який має метод clone() для створення власних копій. Таким чином можна швидко створювати нові об’єкти з однаковими властивостями, але незалежні один від одного.

Усі ці шаблони демонструють різні підходи до вирішення архітектурних завдань і дозволяють розробникам зосередитись на логіці застосунку, не повторюючи типові рішення, що вже довели свою ефективність на практиці.

Хід роботи

Для реалізації функціональності графічного редактора було обрано шаблон проєктування «Прототип» (Prototype), який дозволяє створювати копії вже існуючих об’єктів без необхідності повторної ініціалізації чи завантаження з бази

даних. У межах системи цей підхід використовується для швидкого дублювання зображень, шарів або цілих колажів, зберігаючи їхню структуру та властивості. Класи ImagePrototype, LayerPrototype та CollagePrototype реалізують інтерфейс Prototype і забезпечують механізм глибокого копіювання даних. Завдяки цьому користувач може створювати нові об’єкти на основі вже існуючих, не змінюючи оригінали. Такий підхід підвищує гнучкість системи, спрощує роботу з повторюваними елементами та зменшує витрати ресурсів на створення нових екземплярів складних об’єктів.

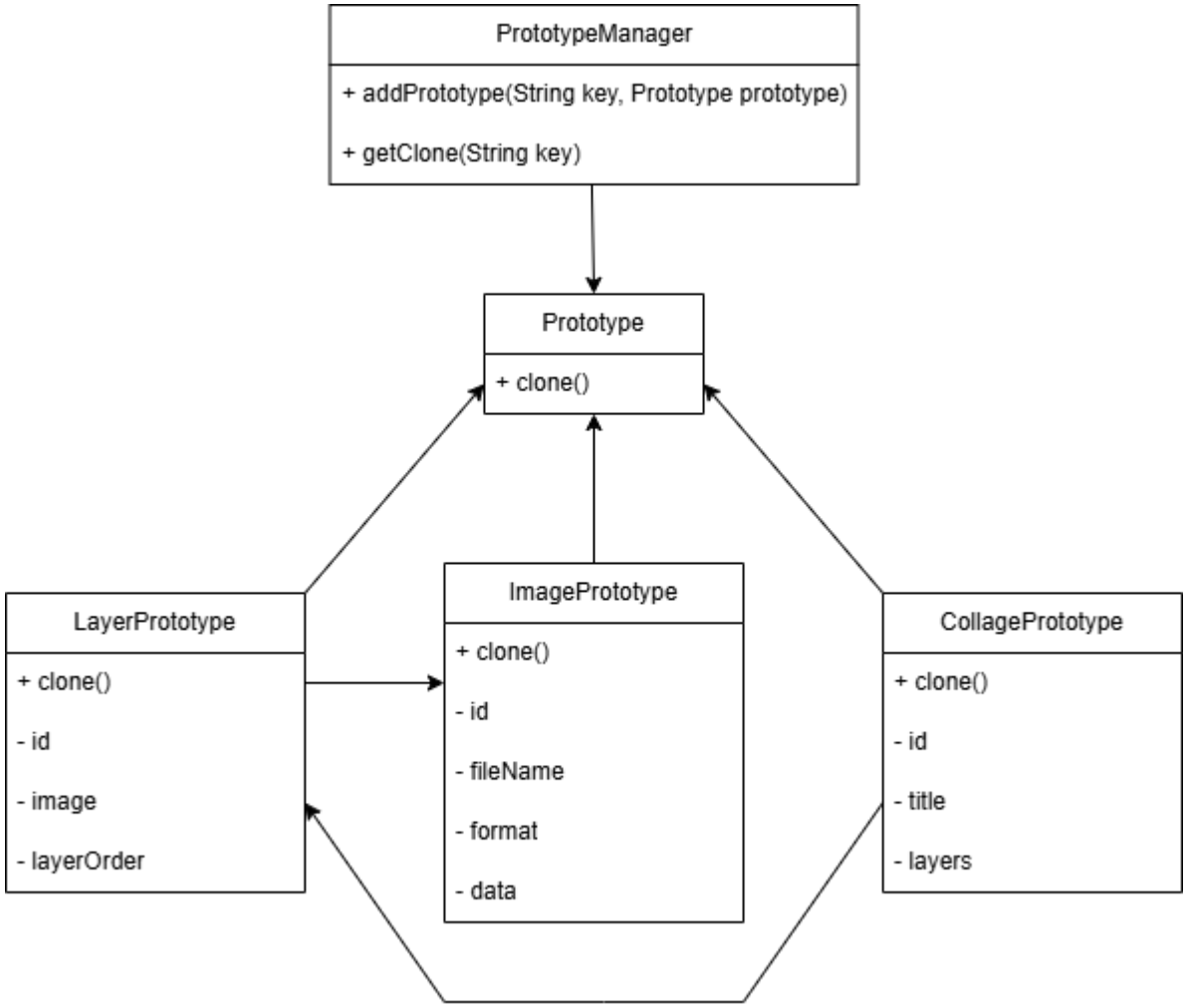


Рисунок 5.1 – Діаграма класів для Prototype

На поданій діаграмі класів зображено реалізацію шаблону «Прототип» у структурі графічного редактора. Інтерфейс Prototype визначає базову поведінку для клонування об’єктів, тоді як класи ImagePrototype, LayerPrototype і CollagePrototype реалізують цю поведінку для різних типів даних. Кожен клас створює власну копію

з повним збереженням стану – наприклад, CollagePrototype копіює всі свої шари, а ті, у свою чергу, копіюють зображення. Така архітектура дозволяє користувачеві швидко дублювати об’єкти та створювати нові композиції на основі вже існуючих, що підвищує ефективність роботи програми та полегшує подальше розширення її функціоналу.

Вихідні коди класів системи

Лістинг 1 – клас Image

```
package com.mycompany.code;

import jakarta.persistence.*;

@Entity
@Table(name = "images")
public class Image {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    @Column(name = "file_name", nullable = false)
    private String fileName;

    @Column(nullable = false)
    private String format;

    @Lob
    @Column(nullable = false)
    private byte[] data;

    public Image() {}

    public int getId() { return id; }
    public void setId(int id) { this.id = id; }

    public String getFileName() { return fileName; }
    public void setFileName(String fileName) { this.fileName = fileName; }

    public String getFormat() { return format; }
    public void setFormat(String format) { this.format = format; }

    public byte[] getData() { return data; }
    public void setData(byte[] data) { this.data = data; }
}
```

Лістинг 2 – Інтерфейс IImage

```
package com.mycompany.code;

public interface IImage {
    void display();
    byte[] getData();
    String getFileName();
}
```

Лістинг 3 – клас GrayscaleEffect

```
public class GrayscaleEffect implements EffectStrategy {

    @Override
    public Image apply(Image img) {
        System.out.println("Застосовано ефект: чорно-білий (Grayscale)");
        // Реалізація перетворення кольорів у градації сірого
        return img;
    }
}
```

Лістинг 4 – клас SepiaEffect

```
public class SepiaEffect implements EffectStrategy {

    @Override
    public Image apply(Image img) {
        System.out.println("Застосовано ефект: сепія (Sepia)");
        // Реалізація перетворення зображення в сепію
        return img;
    }
}
```

Лістинг 5 – клас InvertEffect

```
public class InvertEffect implements EffectStrategy {

    @Override
    public Image apply(Image img) {
        System.out.println("Застосовано ефект: негатив (Invert)");
        // Реалізація інверсії кольорів
        return img;
    }
}
```

Лістинг 6 – клас EffectService

```
public class EffectService {

    private EffectStrategy strategy;

    public void setStrategy(EffectStrategy strategy) {
        this.strategy = strategy;
    }

    public Image applyEffect(Image img) {
        if (strategy == null) {
            throw new IllegalStateException("Ефект не вибрано!");
        }
        return strategy.apply(img);
    }
}
```

Лістинг 7 – клас Layer

```
import jakarta.persistence.*;

@Entity
@Table(name = "layers")
public class Layer {

    @Id
```

```
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    @ManyToOne
    @JoinColumn(name = "image_id", nullable = false)
    private Image image;

    @Column(name = "layer_order")
    private int layerOrder;

    public Layer() {}

    public int getId() { return id; }
    public void setId(int id) { this.id = id; }

    public Image getImage() { return image; }
    public void setImage(Image image) { this.image = image; }

    public int getLayerOrder() { return layerOrder; }
    public void setLayerOrder(int layerOrder) { this.layerOrder = layerOrder; }
}
```

Лістинг 8 – клас Collage

```
import jakarta.persistence.*;
import java.util.ArrayList;
import java.util.List;

@Entity
@Table(name = "collages")
public class Collage {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    @Column(nullable = false)
    private String title;

    @OneToMany(cascade = CascadeType.ALL, orphanRemoval = true)
    @JoinColumn(name = "collage_id")
    private List<Layer> layers = new ArrayList<>();

    public Collage() {}

    public int getId() { return id; }
    public void setId(int id) { this.id = id; }

    public String getTitle() { return title; }
    public void setTitle(String title) { this.title = title; }

    public List<Layer> getLayers() { return layers; }
    public void setLayers(List<Layer> layers) { this.layers = layers; }

    public void addLayer(Layer layer) {
        layers.add(layer);
    }

    public void removeLayer(Layer layer) {
        layers.remove(layer);
    }
}
```

Лістинг 9 – клас models.sql

```
CREATE TABLE Images (
    Id INTEGER PRIMARY KEY,
    FileName VARCHAR(255) NOT NULL,
    Format VARCHAR(50) NOT NULL,
    Data BLOB,
    EffectApplied VARCHAR(50)
);

CREATE TABLE Collages (
    Id INTEGER PRIMARY KEY,
    Title VARCHAR(255) NOT NULL
);

CREATE TABLE CollageImages (
    CollageId INTEGER NOT NULL,
    ImageId INTEGER NOT NULL,
    LayerOrder INTEGER,
    PRIMARY KEY (CollageId, ImageId),
    FOREIGN KEY (CollageId) REFERENCES Collages(Id),
    FOREIGN KEY (ImageId) REFERENCES Images(Id)
);
```

Лістинг 10 – Клас ImageRepository

```
package com.mycompany.code;

import java.sql.*;

public class ImageRepository {

    private final Connection connection;

    public ImageRepository(Connection connection) {
        this.connection = connection;
    }

    public Image findByFileName(String fileName) throws SQLException {
        String query = "SELECT * FROM Images WHERE FileName = ?";
        PreparedStatement stmt = connection.prepareStatement(query);
        stmt.setString(1, fileName);
        ResultSet rs = stmt.executeQuery();

        if (rs.next()) {
            Image img = new Image();
            img.setId(rs.getInt("Id"));
            img.setFileName(rs.getString("FileName"));
            img.setFormat(rs.getString("Format"));
            img.setData(rs.getBytes("Data"));
            return img;
        }
        return null;
    }

    public void update(Image img) throws SQLException {
        String query = "UPDATE Images SET Data = ? WHERE Id = ?";
        PreparedStatement stmt = connection.prepareStatement(query);
        stmt.setBytes(1, img.getData());
        stmt.setInt(2, img.getId());
        stmt.executeUpdate();
    }
}
```

Лістинг 11 – Клас ImageController

```
package com.mycompany.code;

public class ImageController {

    private final ImageRepository imageRepo;
    private final EffectService effectService;

    public ImageController(ImageRepository repo, EffectService service) {
        this.imageRepo = repo;
        this.effectService = service;
    }

    public void displayImage(String fileName) {
        IImage proxy = new ProxyImage(fileName, imageRepo);
        proxy.display();
    }

    public Image applyEffectToImage(Image img, String effectName) {
        switch (effectName.toLowerCase()) {
            case "grayscale" -> effectService.setStrategy(new GrayscaleEffect());
            case "sepia" -> effectService.setStrategy(new SepiaEffect());
            case "invert" -> effectService.setStrategy(new InvertEffect());
            default -> throw new IllegalArgumentException("Невідомий ефект: " +
effectName);
        }
        return effectService.applyEffect(img);
    }
}
```

Лістинг 12 – Імітація форми

```
public class EditorUI {

    private final ImageController controller;

    public EditorUI(ImageController controller) {
        this.controller = controller;
    }

    public void onApplyEffectButtonClick(int imageId, String effectName) {
        try {
            Image result = controller.applyEffectToImage(imageId, effectName);
            System.out.println("Ефект " + effectName + " успішно застосовано!");
        } catch (Exception e) {
            System.err.println("Помилка під час застосування ефекту: " +
e.getMessage());
        }
    }
}
```

Лістинг 13 – клас ProxyImage

```
package com.mycompany.code;

import java.sql.SQLException;

public class ProxyImage implements IImage {

    private String fileName;
    private ImageRepository imageRepo;
    private RealImage realImage;
```

```
    public ProxyImage(String fileName, ImageRepository imageRepo) {
        this.fileName = fileName;
        this.imageRepo = imageRepo;
    }

    @Override
    public void display() {
        try {
            if (realImage == null) {
                System.out.println("Проксі: ліниве завантаження зображення " +
fileName);
                Image dbImage = imageRepo.findByFileName(fileName);
                if (dbImage != null) {
                    realImage = new RealImage(dbImage.getFileName(),
dbImage.getData());
                } else {
                    System.out.println("Зображення не знайдено у базі!");
                    return;
                }
            }
            realImage.display();
        } catch (SQLException e) {
            System.err.println("Помилка доступу до БД: " + e.getMessage());
        }
    }

    @Override
    public byte[] getData() {
        return realImage != null ? realImage.getData() : null;
    }

    @Override
    public String getFileName() {
        return fileName;
    }
}
```

Лістинг 14 – Інтерфейс EffectStrategy

```
package com.mycompany.code;

public interface EffectStrategy {
    Image apply(Image img);
}
```

Лістинг 15 – Клас RealImage

```
package com.mycompany.code;

public class RealImage implements IImage {

    private String fileName;
    private byte[] data;

    public RealImage(String fileName, byte[] data) {
        this.fileName = fileName;
        this.data = data;
        System.out.println("Завантажено реальне зображення: " + fileName);
    }

    @Override
    public void display() {
        System.out.println("Відображення зображення: " + fileName);
    }
```

```
    @Override
    public byte[] getData() {
        return data;
    }

    @Override
    public String getFileName() {
        return fileName;
    }
}
```

Лістинг 16 – Інтерфейс Prototype

```
package com.mycompany.code;

public interface Prototype<T> {
    T clone();
}
```

Лістинг 17 – Клас ImagePrototype

```
package com.mycompany.code;

public class ImagePrototype extends Image implements Prototype<ImagePrototype> {

    public ImagePrototype() {
        super();
    }

    @Override
    public ImagePrototype clone() {
        ImagePrototype copy = new ImagePrototype();
        copy.setId(this.getId()); // можна опустити, якщо потрібно створювати
новий запис у БД
        copy.setFileName(this.getFileName() + "_copy");
        copy.setFormat(this.getFormat());
        copy.setData(this.getData().clone()); // глибоке копіювання байтового
масиву
        return copy;
    }
}
```

Лістинг 18 – Клас LayerPrototype

```
package com.mycompany.code;

public class LayerPrototype extends Layer implements Prototype<LayerPrototype> {

    @Override
    public LayerPrototype clone() {
        LayerPrototype copy = new LayerPrototype();
        copy.setLayerOrder(this.getLayerOrder());
        copy.setImage(this.getImage()); // або new ImagePrototype().clone() — для
копії зображення
        return copy;
    }
}
```

Лістинг 19 – Клас CollagePrototype

```
package com.mycompany.code;

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

public class CollagePrototype extends Collage implements
Prototype<CollagePrototype> {

    @Override
    public CollagePrototype clone() {
        CollagePrototype copy = new CollagePrototype();
        copy.setTitle(this.getTitle() + "_copy");

        List<Layer> clonedLayers = new ArrayList<>();
        for (Layer layer : this.getLayers()) {
            LayerPrototype clonedLayer = new LayerPrototype();
            clonedLayer.setLayerOrder(layer.getLayerOrder());
            clonedLayer.setImage(layer.getImage());
            clonedLayers.add(clonedLayer);
        }

        copy.setLayers(clonedLayers);
        return copy;
    }
}
```

Лістинг 20 - Клас PrototypeController

```
package com.mycompany.code;

public class PrototypeController {

    public ImagePrototype duplicateImage(ImagePrototype original) {
        System.out.println("Створення копії зображення: " +
original.getFileName());
        return original.clone();
    }

    public CollagePrototype duplicateCollage(CollagePrototype original) {
        System.out.println("Створення копії колажу: " + original.getTitle());
        return original.clone();
    }
}
```

У поданих лістингах реалізовано приклад використання шаблону проектування Prototype (Прототип) у системі графічного редактора. Створено інтерфейс Prototype, який визначає метод clone() для створення копій об’єктів. Класи ImagePrototype, LayerPrototype і CollagePrototype реалізують цей інтерфейс, забезпечуючи можливість дублювання відповідних елементів – зображень, шарів і колажів. Кожен клас створює незалежну копію свого стану, включно з вкладеними об’єктами, що дозволяє уникати повторного створення складних структур із нуля.

Клас PrototypeManager виступає централізованим сховищем базових зразків, з яких можна швидко отримати копії. Такий підхід підвищує ефективність і гнучкість роботи системи, зменшуючи навантаження на пам’ять і спрощуючи створення нових елементів на основі вже існуючих.

Висновок:

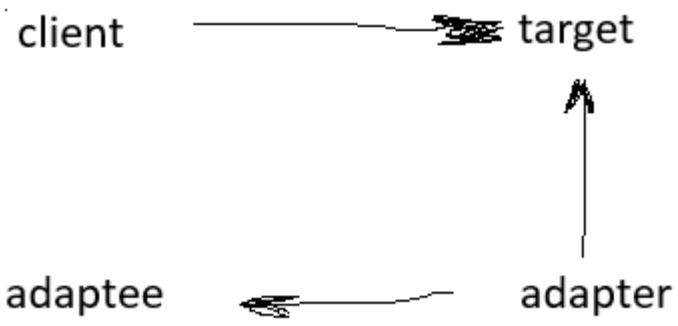
У ході виконання лабораторної роботи було реалізовано шаблон проектування Prototype (Прототип) у контексті системи графічного редактора. Цей шаблон дав змогу створювати копії складних об’єктів без необхідності повторного їх побудування, що значно спрощує роботу з багаторівневими структурами, такими як колажі, шари та зображення. Реалізація продемонструвала можливість глибокого копіювання об’єктів і незалежного редагування їхніх копій без зміни оригіналів. Створені класи ImagePrototype, LayerPrototype і CollagePrototype забезпечили ефективне клонування елементів системи, а PrototypeManager організував централізоване керування зразками. Використання патерну Prototype підвищило гнучкість, продуктивність і масштабованість програми, спростивши створення нових елементів на основі вже існуючих.

Відповіді на контрольні запитання:

1. Яке призначення шаблону «Адаптер»?

Шаблон «Адаптер» (Adapter) призначений для узгодження несумісних інтерфейсів між класами. Його основна мета полягає в тому, щоб дозволити взаємодію між об’єктами, які інакше не могли б працювати разом через різницю в інтерфейсах. Адаптер створює проміжний рівень між клієнтом і цільовим класом, «перекладаючи» виклики з одного інтерфейсу на інший.

2. Нарисуйте структуру шаблону «Адаптер».



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

Target (Цільовий інтерфейс) – визначає стандартний інтерфейс, який очікує клієнт.

Adapter (Адаптер) – реалізує інтерфейс Target і переводить виклики клієнта у виклики до адаптованого об’єкта.

Adaptee (Адаптований клас) – має власний інтерфейс, несумісний із Target.

Client (Клієнт) – працює лише через Target, не знаючи про внутрішні перетворення.

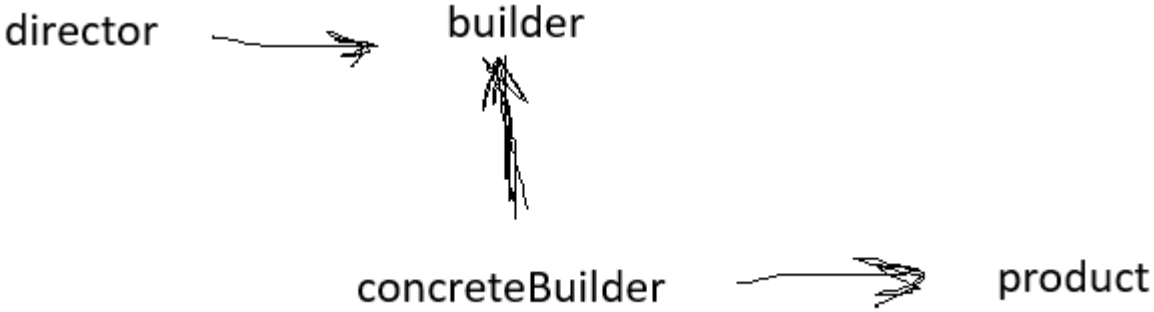
4. Яка різниця між реалізацією «Адаптера» на рівні об’єктів та на рівні класів?

На рівні об’єктів адаптер містить посилання на об’єкт Adaptee (композиція). Це більш гнучкий підхід. На рівні класів адаптер наслідує інтерфейс і Target, і Adaptee (множинне успадкування, можливе не у всіх мовах).

5. Яке призначення шаблону «Будівельник»?

Шаблон «Будівельник» (Builder) використовується для покрокового створення складних об’єктів. Він відділяє процес побудови від його представлення, дозволяючи створювати різні варіанти одного й того ж об’єкта.

6. Нарисуйте структуру шаблону «Будівельник».



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

Builder (Будівельник) – інтерфейс, що визначає етапи створення об’єкта.

ConcreteBuilder (Конкретний будівельник) – реалізує створення конкретних частин.

Director (Директор) – керує процесом побудови.

Product (Продукт) – кінцевий об’єкт, що створюється.

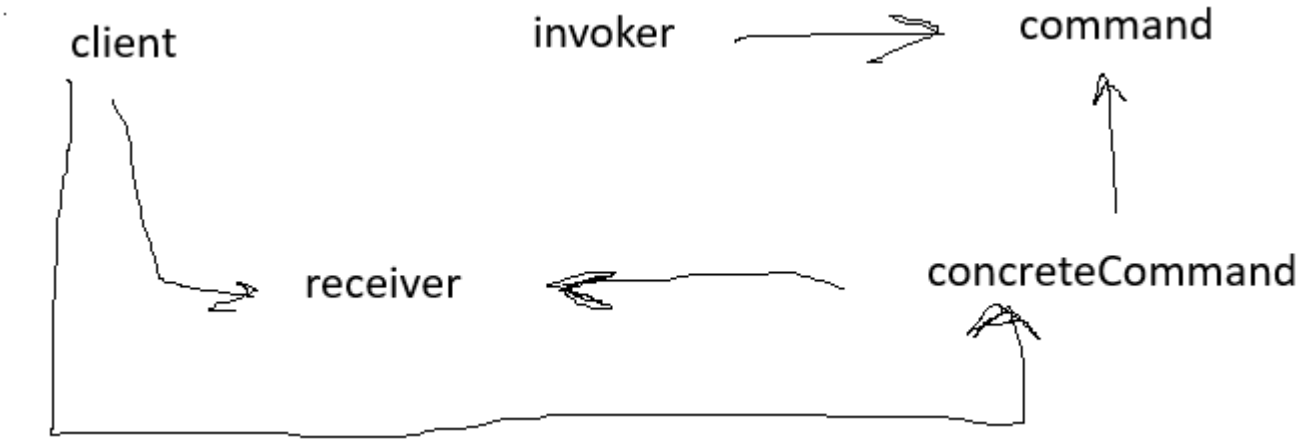
8. У яких випадках варто застосовувати шаблон «Будівельник»"?

Шаблон Builder варто застосовувати, коли об’єкт має багато варіантів конфігурації або складну структуру, і потрібно уникнути конструктора з великою кількістю параметрів.

9. Яке призначення шаблону «Команда»?

Шаблон Builder варто застосовувати, коли об’єкт має багато варіантів конфігурації або складну структуру, і потрібно уникнути конструктора з великою кількістю параметрів.

10. Нарисуйте структуру шаблону «Команда».



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

Command (Команда) – інтерфейс, що визначає метод виконання.

ConcreteCommand (Конкретна команда) – реалізує виклик операції у виконавця.

Receiver (Виконавець) – виконує фактичну дію.

Invoker (Ініціатор) – зберігає команду та запускає її.

Client (Клієнт) – створює команду та передає її ініціатору.

12. Розкажіть як працює шаблон «Команда».

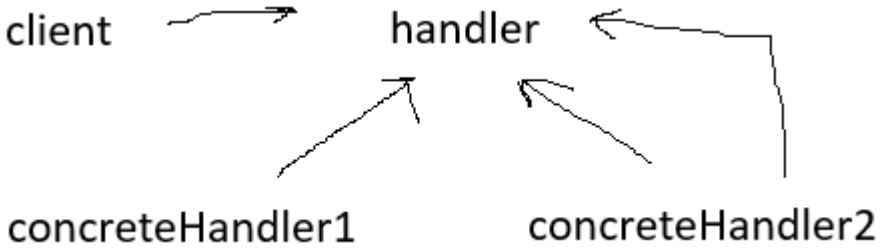
Шаблон «Команда» працює за принципом інкапсуляції дії: команда зберігає всю інформацію для її виконання. Invoker викликає метод execute(), а Receiver виконує реальну роботу.

13. Яке призначення шаблону «Прототип»?

Шаблон «Прототип» (Prototype) дозволяє створювати нові об’єкти шляхом копіювання вже існуючих, а не через конструктор. Це корисно, коли створення

об’єкта є дорогим або складним.

14. Нарисуйте структуру шаблону «Прототип».



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

Prototype (Прототип) – інтерфейс із методом clone().

ConcretePrototype (Конкретний прототип) – реалізує метод клонування.

Client (Клієнт) – викликає clone() для створення нових об’єктів.

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

обробка запитів у ланцюгу фільтрів або middleware (наприклад, у веб-серверах);

система обробки подій у GUI (натискання кнопок, перехоплення подій);

логування, коли запит передається кільком обробникам (консоль, файл, база даних).