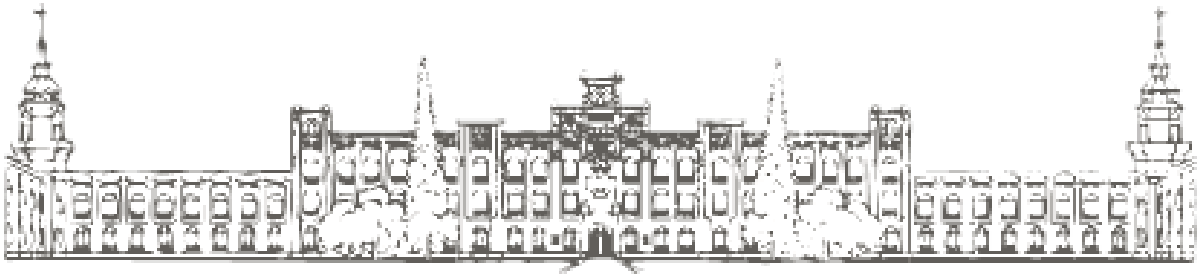


МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені ІГОРЯ СІКОРСЬКОГО»
КАФЕДРА ІНФОРМАЦІЙНИХ СИСТЕМ ТА ТЕХНОЛОГІЙ

Звіт лабораторної роботи №4

з курсу

«Технології розроблення програмного забезпечення»

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

«07» 11 2025 р.

Перевірив: **Мягкий М. Ю.**

Київ – 2025

4. ЛАБОРАТОРНА РОБОТА № 4

Тема: Вступ до паттернів проектування.

Мета: Вивчити структуру шаблонів «Singleton», «Iterator», «Proxy», «State», «Strategy» 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:

Поняття шаблону проєктування

Будь-який патерн проєктування, використовуваний при розробці інформаційних систем, являє собою формалізований опис, який часто зустрічається в завданнях проєктування, вдале рішення даної задачі, а також рекомендації по застосуванню цього рішення в різних ситуаціях [5]. Крім того, патерн проєктування обов’язково має загальновживане найменування. Правильно сформульований патерн проєктування дозволяє, відшукавши одного разу вдале рішення, користуватися ним знову і знову. Варто підкреслити, що важливим початковим етапом при роботі з патернами є адекватне моделювання розглянутої предметної області. Це є

необхідним як для отримання належним чином формалізованої постановки задачі, так і для вибору відповідних патернів проєктування.

Відповідне використання патернів проєктування дає розробнику ряд незаперечних переваг. Наведемо деякі з них. Модель системи, побудована в межах патернів проєктування, фактично є структурованим виокремленням тих елементів і зв’язків, які значимі при вирішенні поставленого завдання. Крім цього, модель, побудована з використанням патернів проєктування, більш проста і наочна у вивченні, ніж стандартна модель. Проте, не дивлячись на простоту і наочність, вона дозволяє глибоко і всебічно опрацювати архітектуру розроблюваної системи з використанням спеціальної мови.

Застосування патернів проєктування підвищує стійкість системи до зміни вимог та спрощує неминуче подальше доопрацювання системи. Крім того, важко переоцінити роль використання патернів при інтеграції інформаційних систем організації. Також слід зазначити, що сукупність патернів проєктування, по суті, являє собою єдиний словник проєктування, який, будучи уніфікованим засобом, незамінний для спілкування розробників один одним.

Таким чином шаблони представляють собою, підтверджені роками розробок в різних компаніях і на різних проєктах, «ескізи» архітектурних рішень, які зручно застосовувати у відповідних обставинах.

Хід роботи:

Реалізація частини функціоналу робочої програми у вигляді класів та їхньої взаємодії для досягнення конкретних функціональних можливостей.

```java
package com.example.graphiceditor.repository;

import com.example.graphiceditor.model.User;
import com.example.graphiceditor.persistence.JpaUtil;
import jakarta.persistence.EntityManager;
import java.util.Optional;

public class UserRepository extends JpaBaseRepository<User, Long> {  8 usages  new *
    public UserRepository() { super(User.class); }  3 usages  new *
    public Optional<User> findByUsername(String username) {  2 usages  new *
        EntityManager em = JpaUtil.emf().createEntityManager();
        try {
            return em.createQuery( s: "select u from User u where u.username=:name", User.class)
                    .setParameter( s: "name", username)
                    .getResultStream().findFirst();
        } finally { em.close(); }
    }
}
```

Рис.1 - **UserRepository**

Спеціалізований репозиторій для сутності User, що наслідує JpaBaseRepository<User, Long>. Додає метод findByUsername(String username), який за допомогою EntityManager виконує JPQL-запит select u from User u where u.username = :name та повертає результат як Optional<User>.

```java
package com.example.graphiceditor.repository;

import com.example.graphiceditor.model.Project;
import com.example.graphiceditor.persistence.JpaUtil;
import java.util.List;

public class ProjectRepository extends JpaBaseRepository<Project, Long> {  5 usages  new *
    public ProjectRepository() {  2 usages  new *
        super(Project.class);
    }

    public List<Project> findByUserId(Long userId) {  1 usage  new *
        var em = JpaUtil.emf().createEntityManager();
        try {
            return em.createQuery( s: """
                select p from Project p
                where p.user.id = :uid
                order by p.lastModified desc
            """, Project.class).setParameter( s: "uid", userId).getResultList();
        } finally {em.close();}
    }
}
```

Рис.2 - **ProjectRepository**

Репозиторій для сутності Project, що наслідує JpaBaseRepository<Project, Long>. Реалізує метод findByUserId(Long userId), який виконує JPQL-запит на вибірку проєктів певного користувача (where p.user.id = :uid) з упорядкуванням за полем lastModified у спадаючому порядку і повертає список проєктів.

```java
package com.example.graphiceditor.repository;

import com.example.graphiceditor.persistence.JpaUtil;
import jakarta.persistence.EntityManager;
import java.util.List;

public abstract class JpaBaseRepository<T, ID> {  3 usages  3 inheritors  new *
    private final Class<T> type;  4 usages
    protected JpaBaseRepository(Class<T> type) { this.type = type; }  3 usages  new *

    public T save(T entity) {  6 usages  new *
        EntityManager em = JpaUtil.emf().createEntityManager();
        try {
            em.getTransaction().begin();
            T merged = em.merge(entity);
            em.getTransaction().commit();
            return merged;
        } finally { em.close(); }
    }

    public T find(ID id) {  no usages  new *
        EntityManager em = JpaUtil.emf().createEntityManager();
        try { return em.find(type, id); }
        finally { em.close(); }
    }

    public List<T> findAll() {  no usages  new *
        EntityManager em = JpaUtil.emf().createEntityManager();
        try {
            return em.createQuery( s: "select e from " + type.getSimpleName() + " e", type).getResultList();
        } finally { em.close(); }
    }
```

Рис.3 - **JpaBaseRepository**

Абстрактний узагальнений репозиторій, який інкапсулює базові операції доступу до даних через JPA. Містить методи save(T entity) для збереження/оновлення сутності в транзакції, find(ID id) для пошуку за первинним ключем та findAll() для вибірки всіх екземплярів певного типу. Тип сутності зберігається у полі type і передається через конструктор.

```java
package com.example.graphiceditor.service;

import com.example.graphiceditor.model.User;
import com.example.graphiceditor.repository.UserRepository;

public class AuthService {  3 usages  new *
    private final UserRepository users = new UserRepository();  2 usages

    public User loginOrCreate(String username) {  no usages  new *
        return users.findByUsername(username).orElseGet(() -> {
            var u = new User();
            u.setUsername(username);
            u.setAuthorized(true);
            return users.save(u);
        });
    }
}
```

Рис.4 - **AuthService**

Сервіс авторизації, який використовує UserRepository для роботи з користувачами. Метод loginOrCreate(String username) виконує пошук користувача за іменем; якщо користувача не знайдено, створює новий об’єкт User з заданим ім’ям, позначає його як авторизованого (setAuthorized(true)) та зберігає через репозиторій, повертаючи існуючого або щойно створеного користувача.

```java
package com.example.graphiceditor.ui;

import javafx.fxml.FXMLLoader;
import javafx.scene.Parent;
import javafx.scene.Scene;
import javafx.stage.Stage;

public class ViewRouter {  4 usages  new *

    private static Stage primaryStage;  7 usages

    public static void setPrimaryStage(Stage stage) {  1 usage  new *
        primaryStage = stage;
    }

    public static void show(String fxmlPath, String title) {  1 usage  new *
        try {
            FXMLLoader loader = new FXMLLoader(ViewRouter.class.getResource(fxmlPath));
            Parent root = loader.load();

            primaryStage.setTitle(title);
            primaryStage.setScene(new Scene(root));
            primaryStage.show();
        } catch (Exception ex) {
            ex.printStackTrace();
            throw new RuntimeException("Cannot load FXML: " + fxmlPath, ex);
        }
    }
```

Рис.5 - **ViewRouter**

```java
    public static <T> T showAndGetController(String fxmlPath, String title) {  no usages  new *
        try {
            FXMLLoader loader = new FXMLLoader(ViewRouter.class.getResource(fxmlPath));
            Parent root = loader.load();

            primaryStage.setTitle(title);
            primaryStage.setScene(new Scene(root));
            primaryStage.show();

            return loader.getController();
        } catch (Exception ex) {
            ex.printStackTrace();
            throw new RuntimeException("Cannot load FXML: " + fxmlPath, ex);
        }
    }
}
```

Рис.6 - **ViewRouter**

Клас ViewRouter(Рис.5-6), розташований у пакеті com.example.graphiceditor.ui, виконує роль централізованого навігатора між вікнами (формами) JavaFX у застосунку «Графічний редактор». Його основне призначення — інкапсулювати логіку завантаження FXML-ресурсів, створення сцен та відображення їх на головному вікні програми (primaryStage), забезпечуючи єдиний вхідний пункт для перемикання представлень (view).

У класі використовується статичне поле primaryStage, яке зберігає посилання на головний об’єкт Stage застосунку. Метод setPrimaryStage(Stage stage) призначений для початкової ініціалізації цього поля, зазвичай викликається з методу start(...) головного класу JavaFX. Таким чином, ViewRouter не створює нові вікна, а керує вже існуючим головним вікном, змінюючи його вміст (сцену) відповідно до запитів.

Метод show(String fxmlPath, String title) завантажує FXML-ресурс, розташований за шляхом fxmlPath, за допомогою FXMLLoader, створює кореневий вузол сцени (Parent root), після чого формує новий об’єкт Scene і встановлює його на primaryStage. Додатково оновлюється заголовок вікна (setTitle(title)), а саме вікно відображається на екрані через виклик show(). У разі виникнення помилок завантаження FXML (некоректний шлях, помилка у файлі розмітки тощо) генерується виняток типу RuntimeException з пояснювальним повідомленням, а стек викликів виводиться у консоль для спрощення діагностики.

Узагальнений метод showAndGetController(String fxmlPath, String title) реалізує подібний сценарій, але додатково повертає контролер завантаженої FXML-форми. Після завантаження ресурсу і встановлення нової сцени на primaryStage, метод викликає loader.getController() і повертає посилання на контролер, тип якого задається як

параметр-узагальнення <T>. Це дозволяє викликаючому коду одразу отримати доступ до відповідного контролера (наприклад, для передачі даних, ініціалізації стану або виклику спеціалізованих методів), не порушуючи інкапсуляції логіки відображення. Такий підхід сприяє централізації навігації між вікнами та підвищує узгодженість роботи графічного інтерфейсу.

Реалізація одного з розглянутих шаблонів, шаблону “State”.

Структура проекту з цим шаблоном:

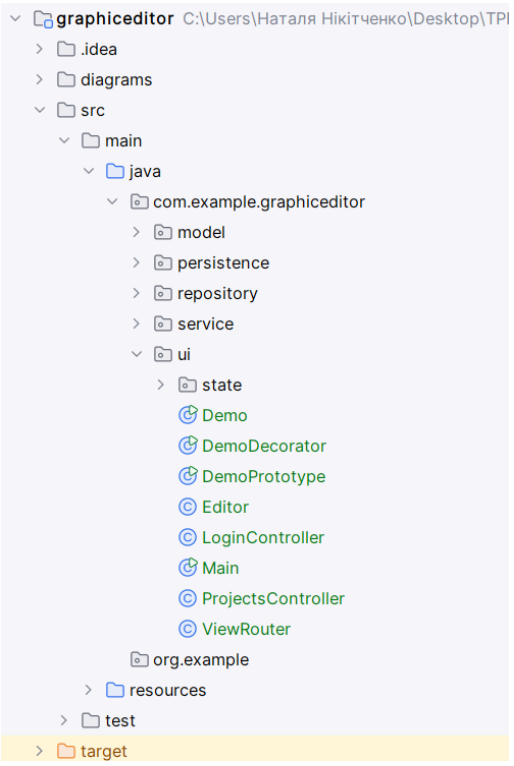


Рис.7 - Структура проекту

Реалізація програмна:

Контекст — Editor:

```
package com.example.graphiceditor.ui;

import com.example.graphiceditor.ui.state.ToolState;
import com.example.graphiceditor.ui.state.IdleState;

public class Editor {  22 usages  new *

    private ToolState state = new IdleState();  4 usages
    private int[][] selection;  6 usages
    private java.util.List<int[]> path = new java.util.ArrayList<>();  7 usages

@   public void setState(ToolState newState) {  5 usages  new *
        System.out.println("[Editor] Switch -> " + newState.getName());
        this.state = newState;
    }

    public void mouseDown(int x, int y) { state.onMouseDown(x, y,  ctx: this); }  2 usages  new *
    public void mouseMove(int x, int y) { state.onMouseMove(x, y,  ctx: this); }  3 usages  new *
    public void mouseUp(int x, int y)   { state.onMouseUp(x, y,  ctx: this); }  3 usages  new *


    public void beginPath(int x, int y) {  1 usage  new *
        path.clear();
        path.add(new int[]{x, y});
        System.out.println("[Canvas] begin path at (" + x + "," + y + ")");
    }
```

Рис.8 - **Editor**

```
    public void addPoint(int x, int y) {  2 usages  new *
        path.add(new int[]{x, y});
        if (path.size() % 5 == 0)
            System.out.println("[Canvas] path len=" + path.size());
    }

    public void stroke() {  1 usage  new *
        System.out.println("[Canvas] stroke path, points=" + path.size());
        path.clear();
    }

    public void setSelection(int x1, int y1, int x2, int y2) {  1 usage  new *
        this.selection = new int[][]{{x1, y1}, {x2, y2}};
        System.out.println("[Canvas] selection: (" + x1 + "," + y1 + ") → (" + x2 + "," + y2 + ")");
    }

    public void moveSelection(int dx, int dy) {  1 usage  new *
        if (selection == null) {
            System.out.println("[Canvas] no selection to move");
            return;
        }
        selection[0][0] += dx;
        selection[1][0] += dx;
        selection[0][1] += dy;
        selection[1][1] += dy;

        System.out.println("[Canvas] moved selection dx=" + dx + ", dy=" + dy);
    }
}
```

Рис.9 - **Editor**

Клас Editor(Рис.8-9), розташований у пакеті com.example.graphiceditor.ui, моделює логіку полотна редактора та виступає контекстом для реалізації шаблону проєктування State (стан інструмента). Поле state типу ToolState з початковим значенням IdleState визначає поточний активний інструмент (стан), а метод setState(ToolState newState) забезпечує переключення між інструментами з діагностичним виводом у консоль.

Методи mouseDown(int x, int y), mouseMove(int x, int y) та mouseUp(int x, int y) делегують обробку подій миші поточному стану, передаючи координати та посилання на об’єкт Editor. Клас також містить допоміжні методи, що імітують низькорівневі операції над полотном: beginPath(int x, int y) та addPoint(int x, int y) формують послідовність точок контуру (path), а метод stroke() завершує побудову та «промальовує» шлях,

виводячи інформацію про кількість точок. Окремо реалізовано роботу з прямокутною області виділення: метод setSelection(int x1, int y1, int x2, int y2) задає координати виділення, а moveSelection(int dx, int dy) зсуває його на заданий вектор, контролюючи випадок відсутності активного виділення. Усі операції супроводжуються текстовими повідомленнями у консоль, що спрощує трасування поведінки під час тестування.

IdleState — неактивний інструмент:

```
package com.example.graphiceditor.ui.state;

import com.example.graphiceditor.ui.Editor;

public class IdleState implements ToolState {  4 usages  new *

    @Override  1 usage  new *
    public void onMouseDown(int x, int y, Editor ctx) {
        System.out.println("[Idle] Click ignored");
    }

    @Override  1 usage  new *
    public void onMouseMove(int x, int y, Editor ctx) {}

    @Override  1 usage  new *
    public void onMouseUp(int x, int y, Editor ctx) {}

    @Override  new *
    public String getName() {
        return "Idle";
    }
}
```

Рис.10 - **IdleState**

Клас IdleState реалізує інструмент «бездіяльності» і виступає станом за замовчуванням. У цьому стані натискання миші не призводять до зміни

полотна: метод onMouseDown(...) лише виводить повідомлення про ігнорування кліку, а методи onMouseMove(...) та onMouseUp(...) порожні. Метод getName() повертає назву стану "Idle".

PenState — інструмент “Перо”:

```
package com.example.graphiceditor.ui.state;

import com.example.graphiceditor.ui.Editor;

public class PenState implements ToolState {  1 usage  new *

    @Override  1 usage  new *
    public void onMouseDown(int x, int y, Editor ctx) {
        System.out.println("[Pen] Start drawing");
        ctx.beginPath(x, y);
    }

    @Override  1 usage  new *
    public void onMouseMove(int x, int y, Editor ctx) {
        ctx.addPoint(x, y);
    }

    @Override  1 usage  new *
    public void onMouseUp(int x, int y, Editor ctx) {
        ctx.addPoint(x, y);
        ctx.stroke();
        System.out.println("[Pen] Finish drawing");
        ctx.setState(new IdleState());
    }

    @Override  new *
    public String getName() {
        return "Pen";
    }
}
```

Рис.11 - **PenState**

Клас PenState реалізує інструмент «Перо» для малювання вільною рукою. У методі onMouseDown(...) стан ініціалізує новий шлях малювання через виклик ctx.beginPath(x, y) і виводить повідомлення про початок малювання. Під час переміщення миші (onMouseMove(...)) до поточного шляху додаються точки (ctx.addPoint(x, y)). При відпусканні кнопки миші (onMouseUp(...)) додається остання точка, шлях «промальовується» методом ctx.stroke(), після чого інструмент повертає контекст у стан IdleState (ctx.setState(new IdleState())). Назва стану повертається як "Pen".

SelectState — інструмент “Виділення”:

```
package com.example.graphiceditor.ui.state;

import com.example.graphiceditor.ui.Editor;

public class SelectState implements ToolState {  1 usage  new *

    private int anchorX, anchorY;  3 usages
    private boolean dragging = false;  3 usages

    @Override  1 usage  new *
    public void onMouseDown(int x, int y, Editor ctx) {
        anchorX = x;
        anchorY = y;
        dragging = true;
        System.out.println("[Select] Anchor at (" + x + "," + y + ")");
    }

    @Override  1 usage  new *
    public void onMouseMove(int x, int y, Editor ctx) {
        if (dragging) {
            int x1 = Math.min(anchorX, x);
            int y1 = Math.min(anchorY, y);
            int x2 = Math.max(anchorX, x);
            int y2 = Math.max(anchorY, y);
            ctx.setSelection(x1, y1, x2, y2);
        }
    }
```

Рис.12 - **SelectState**

```
    @Override  1 usage  new *
    public void onMouseUp(int x, int y, Editor ctx) {
        dragging = false;
        System.out.println("[Select] Selection fixed");
        ctx.setState(new MoveState()); // автоматичний перехід
    }

    @Override  new *
    public String getName() {
        return "Select";
    }
}
```

Рис.13 - **SelectState**

Клас SelectState реалізує інструмент «Виділення». Він зберігає координати початкової точки (anchorX, anchorY) та прапорець dragging, який позначає процес перетягування. У методі onMouseDown(...) фіксується початкова точка виділення і вмикається режим перетягування. Під час руху миші (onMouseMove(...)), якщо dragging == true, розраховуються координати протилежного кута прямокутника (мінімальні й максимальні значення по осях), після чого в контексті встановлюється область виділення (ctx.setSelection(x1, y1, x2, y2)). У методі onMouseUp(...) перетягування завершується, виводиться повідомлення про фіксацію виділення, а контекст автоматично переводиться в стан MoveState для подальшого переміщення вибраної області. Метод getName() повертає "Select".

MoveState — інструмент переміщення:

```java
package com.example.graphiceditor.ui.state;

import com.example.graphiceditor.ui.Editor;

public class MoveState implements ToolState {  1 usage  new *
    @Override public void onMouseDown(int x, int y, Editor ctx) {  1 usage  new *
        System.out.println("[Move] Drag to move selection");
    }
    @Override public void onMouseMove(int x, int y, Editor ctx) {  1 usage  new *
        ctx.moveSelection( dx: 1,  dy: 0); // демо-рух
    }
    @Override public void onMouseUp(int x, int y, Editor ctx) {  1 usage  new *
        System.out.println("[Move] Move done");
        ctx.setState(new IdleState());
    }
    @Override public String getName() { return "Move"; }  new *
}
```

Рис.14 - **MoveState**

Клас MoveState реалізує інструмент «Переміщення». У методі onMouseDown(...) виводиться службове повідомлення про початок перетягування. У методі onMouseMove(...) демонстраційно викликається ctx.moveSelection(dx, dy) із фіксованими зсувами (наприклад, dx = 1, dy = 0), що моделює рух виділеної області по полотну. Після завершення дії (onMouseUp(...)) виводиться повідомлення про завершення переміщення, і контекст повертається до стану IdleState. Назва стану повертається як "Move".

Демо (main):

```
package com.example.graphiceditor.ui;

import com.example.graphiceditor.ui.state.*;

public class Demo {  new *
    public static void main(String[] args) {  new *

        Editor ed = new Editor();

        // Pen tool
        ed.setState(new PenState());
        ed.mouseDown( x: 10,  y: 10);
        for (int x = 11; x <= 30; x++)
            ed.mouseMove(x,  y: 10 + (x % 3));
        ed.mouseUp( x: 30,  y: 12);

        // Select tool
        ed.setState(new SelectState());
        ed.mouseDown( x: 5,  y: 5);
        ed.mouseMove( x: 20,  y: 15);
        ed.mouseUp( x: 20,  y: 15);

        // Move tool automatically activated
        for (int i = 0; i < 5; i++)
            ed.mouseMove( x: 0,  y: 0);
        ed.mouseUp( x: 0,  y: 0);
    }
}
```

Рис.15 - **Демо (main)**

Клас Demo містить метод public static void main(String[] args) і слугує демонстраційним сценарієм роботи шаблону State. У ньому створюється екземпляр Editor, після чого послідовно активуються різні інструменти (PenState, SelectState, автоматично — MoveState) через setState(...). Далі викликаються методи mouseDown, mouseMove та mouseUp з різними координатами, що імітують дії користувача мишею. Вивід у консоль дозволяє прослідкувати зміну станів і реакцію редактора на події.

Діаграма класів, яка представляє використання шаблону в реалізації системи(Рис.16):

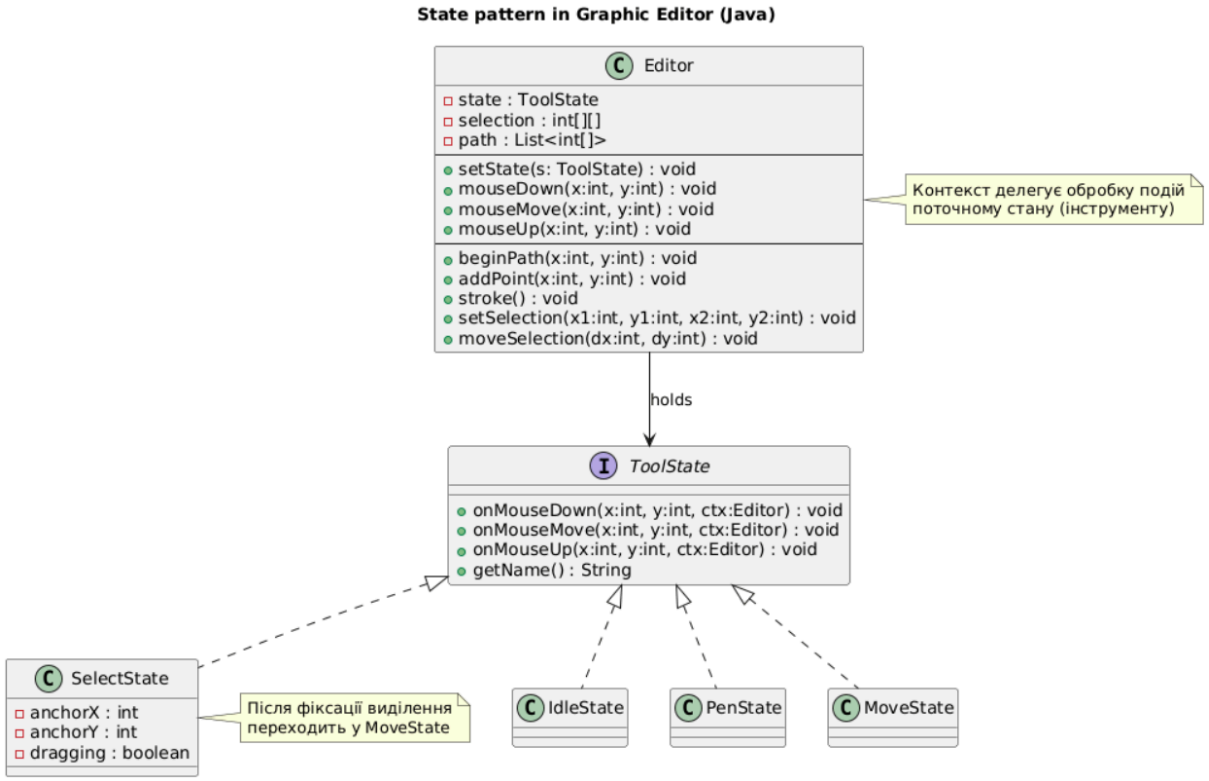


Рис.16 - Діаграма класів, яка представляє використання даного шаблону

ToolState інтерфейс

```
package com.example.graphiceditor.ui.state;

import com.example.graphiceditor.ui.Editor;

public interface ToolState {  7 usages  4 implementations  new *

    void onMouseDown(int x, int y, Editor ctx);  1 usage  4 implementations  new *

    void onMouseMove(int x, int y, Editor ctx);  1 usage  4 implementations  new *

    void onMouseUp(int x, int y, Editor ctx);  1 usage  4 implementations  new *

    String getName();  4 implementations  new *
}
```

Рис.17 - **ToolState інтерфейс**

Інтерфейс ToolState визначає контракт для всіх станів (інструментів) редактора. Він задає три методи обробки подій миші — onMouseDown(int x, int y, Editor ctx), onMouseMove(int x, int y, Editor ctx) і onMouseUp(int x, int y, Editor ctx) — а також метод getName(), що повертає назву інструмента. Параметр Editor ctx є контекстом, через який стани взаємодіють із полотном та іншою логікою редактора.

Висновки:

У ході роботи було реалізовано фрагмент програмної системи «Графічний редактор» із використанням шаблону проектування **State**. Даний патерн дозволив організувати змінювану поведінку інтерфейсу залежно від активного інструмента (стану), що є характерним для графічних редакторів.

Було створено інтерфейс **ToolState** та чотири конкретні стани: **IdleState**, **PenState**, **SelectState** і **MoveState**, а також клас **Editor**, який виконує роль контексту та делегує обробку подій відповідному стану. Завдяки цьому вдалося розподілити поведінку між окремими класами, уникнути надмірних умовних конструкцій та забезпечити гнучкість розширення системи новими інструментами.

Діаграма класів підтверджує коректну реалізацію шаблону State: кожен інструмент інкапсулює власну логіку обробки подій, а переходи між станами відбуваються динамічно в процесі роботи. Таким чином, поставлену задачу виконано, а використання патерну дозволило підвищити структурованість, зрозумілість і масштабованість застосунку.

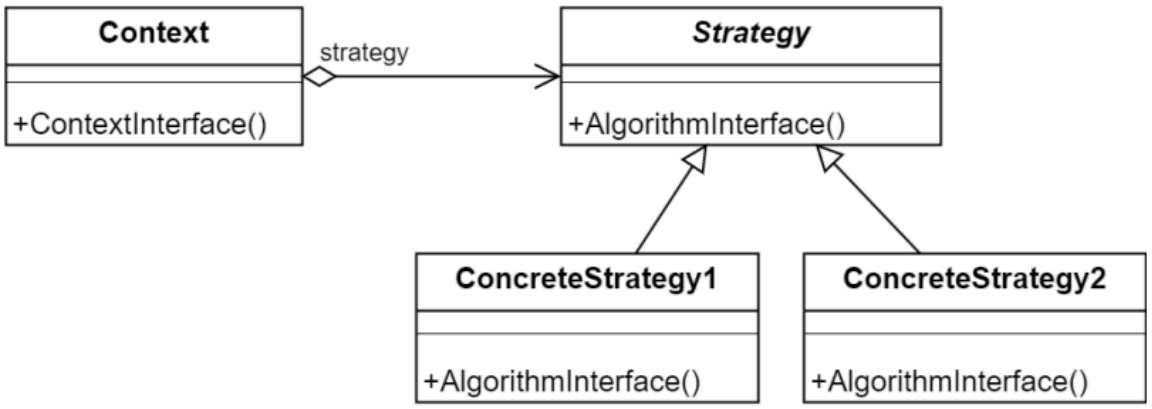
Контрольні питання:

1.Шаблон проєктування — це перевірене повторюване рішення типових проблем проєктування ПЗ на рівні взаємодії класів/об’єктів (а не готовий код).

2.Використовують шаблони, щоб підвищити зрозумілість і підтримуваність коду, зменшити зв’язність, покращити гнучкість/розширюваність і повторно застосовувати найкращі практики.

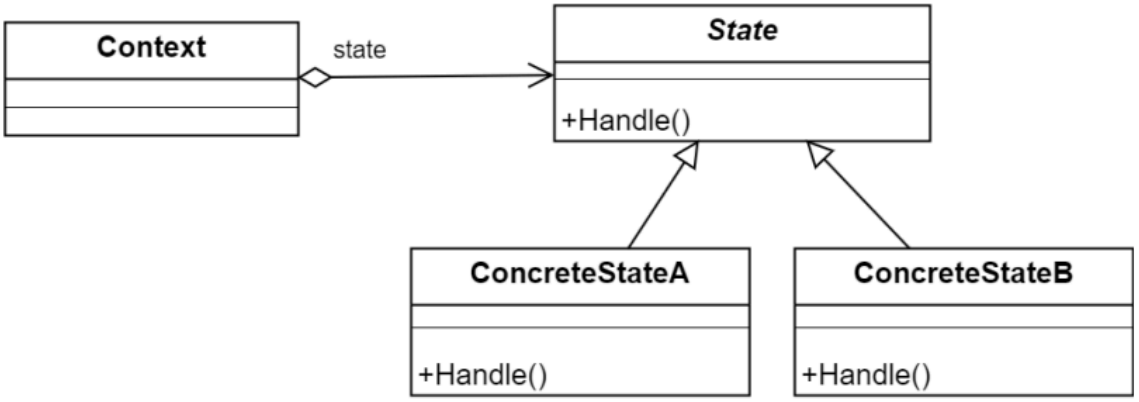
3.«Стратегія» (Strategy) інкапсулює взаємозамінні алгоритми за спільним інтерфейсом і дозволяє перемикати їх під час виконання без зміни коду клієнта.

4.Структура «Стратегії» (спрощено, ASCII-UML):



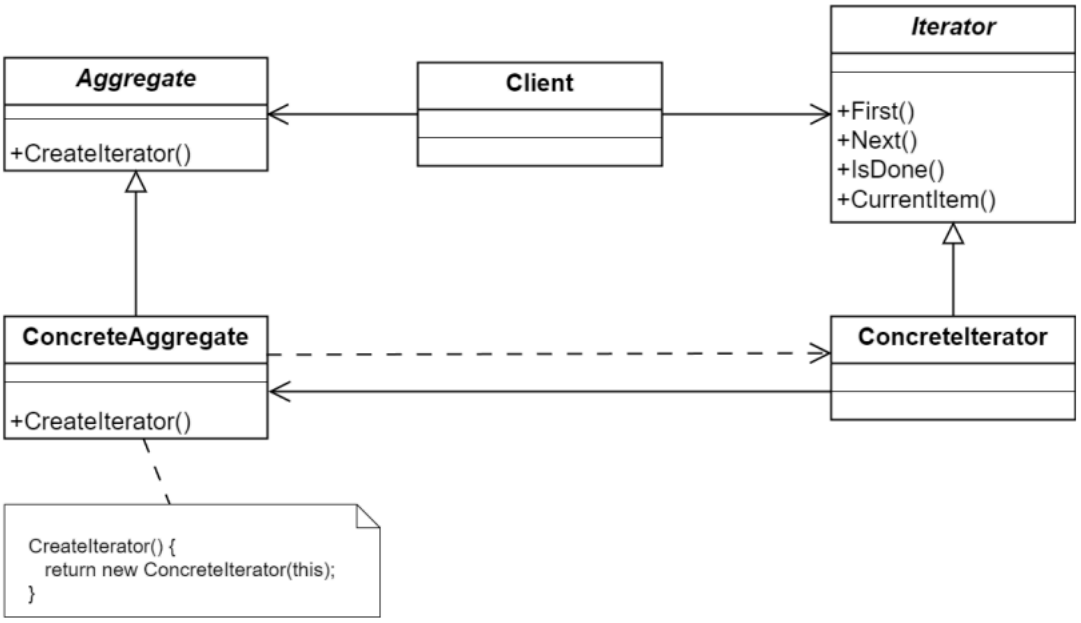
5. Класи та взаємодія «Стратегії»: Context містить посилання на Strategy; Strategy — інтерфейс алгоритму; ConcreteStrategyA/B — реалізації. Context делегує виконання алгоритму вибраній стратегії.
6. «Стан» (State) дозволяє об’єкту змінювати свою поведінку при зміні внутрішнього стану: замість розгалужень if/switch — делегування на об’єкти станів.

7. Структура «Стану»:



8. Класи та взаємодія «Стану»: Context тримає поточний State; State — інтерфейс дій; ConcreteStateA/B реалізують дії й можуть переводити Context у інший стан (виклик setState).
9. «Ітератор» (Iterator) надає стандартизований спосіб покрокового доступу до елементів колекції без розкриття її внутрішньої структури.

10. Структура «Ітератора»:



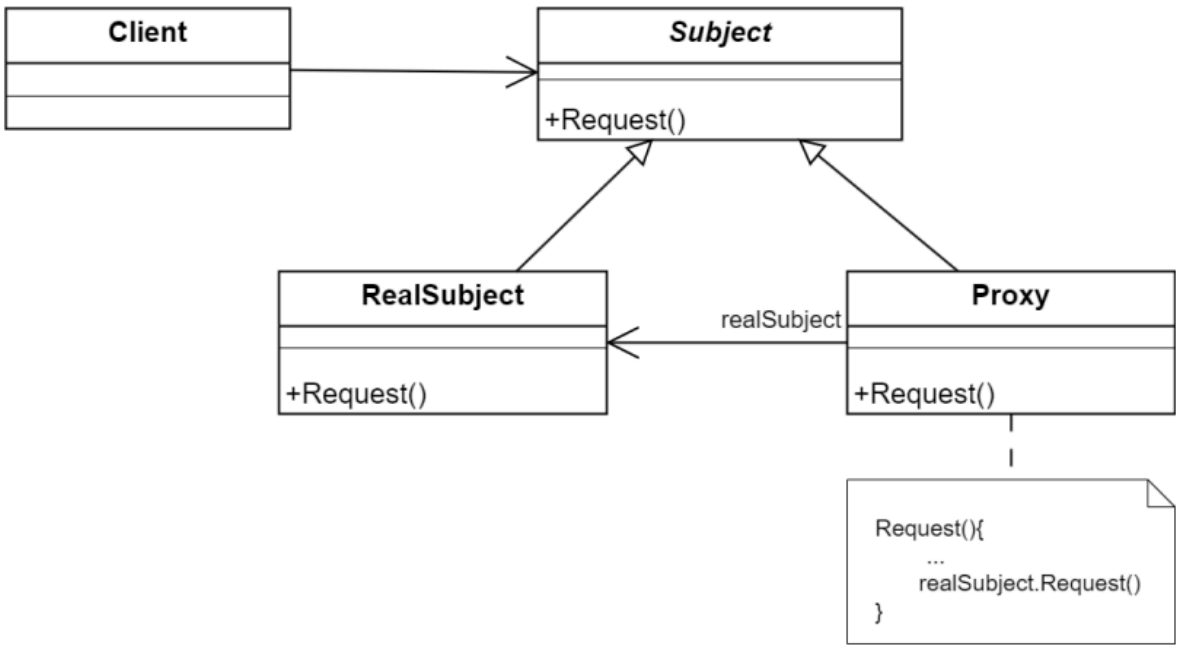
11. Класи та взаємодія «Ітератора»: Aggregate/ConcreteAggregate — колекція; Iterator/ConcreteIterator — інтерфейс і реалізація перебору (hasNext(), next()); клієнт отримує ітератор у Aggregate і рухається елементами через Iterator.

12. Ідея «Одинака» (Singleton): гарантувати, що клас має рівно один екземпляр і надати глобальну точку доступу до нього.

13. Чому «Одинак» часто вважають анти-шаблоном: він ускладнює тестування (глобальний стан), приховано підвищує зв’язність, порушує принцип єдиного обов’язку та часто замінюється DI/контейнерами, конфігурацією або фабриками.

14. «Проксі» (Proxy) надає сурогат/замінник об’єкта для контролю доступу до нього: ліниве створення, кешування, безпека, віддалений доступ, логування тощо.

15.Структура «Проксі»:



16.Класи та взаємодія «Проксі»: Subject — інтерфейс сервісу; RealSubject — реальна реалізація; Proxy реалізує Subject, тримає посилання на RealSubject і додає додаткову поведінку (перевірки, кеш, мережа), перш ніж делегувати виклик реальному об’єкту.