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 №7

із дисципліни *«Технології розроблення програмного забезпечення»*

Тема: « ШАБЛОН «MEDIATOR», «FACADE», «BRIDGE», «TEMPLATE METHOD»»

Виконав:

Студент групи ІА-23 Хохол М.В.

Перевірив:

Мягкий М.Ю.

Київ 2024

Варіант №7

..7 Редактор зображень (state, prototype, memento, facade, composite, client-server)

Редактор зображень має такі функціональні можливості: відкриття/збереження зображень у найпопулярніших форматах (5 на вибір студента), застосування ефектів, наприклад поворот, розтягування, стиснення, кадрування зображення, можливість створення колажів шляхом «нашарування» зображень.

Завдання:

1. Ознайомитися з короткими теоретичними відомостями.

2. Реалізувати частину функціоналу робочої програми у вигляді класів та їх взаємодій для досягнення конкретних функціональних можливостей.

3. Застосування одного з розглянутих шаблонів при реалізації програми.

Зміст

Хід роботи:

Крок 1. Короткі теоретичні відомості.

1) DRY (Don't Repeat Yourself) – "Не повторюйся."
Цей принцип наголошує на уникненні дублювання коду. Замість повторення, спільну функціональність краще виділити в окремі методи, класи чи модулі, щоб полегшити підтримку та зменшити помилки.

2) KISS (Keep It Simple, Stupid) – "Роби просто, дурнику."
Принцип закликає створювати максимально простий та зрозумілий код. Уникайте надмірної складності, якщо завдання можна вирішити більш елегантним і прямолінійним способом.

3) YOLO (You Only Load It Once) – "Завантажуєш лише раз."
Суть у тому, щоб ініціалізувати та завантажувати змінні або дані тільки один раз під час запуску програми. Це важливо для оптимізації продуктивності, особливо в програмах, які працюють із великими обсягами даних.

4) Принцип Паретто (80/20)
Стверджує, що 80% результату досягається за рахунок 20% зусиль. У програмуванні це означає зосередження на найбільш важливих функціях, які дають найбільшу користь, а не на всіх можливих деталях.

5) YAGNI (You Aren't Gonna Need It) – "Тобі це не знадобиться."
Закликає уникати додавання функціональності, яка не потрібна прямо зараз. Вважається, що створення непотрібного коду ускладнює систему без очевидної користі.

6) Mediator – це шаблон проєктування, який використовується для спрощення комунікації між об'єктами. Він дозволяє централізувати обмін повідомленнями через спеціальний об'єкт-посередник, зменшуючи залежності між компонентами. Завдяки цьому об'єкти спілкуються не безпосередньо, а через посередника, що покращує модульність та полегшує зміну логіки взаємодії.

7) Facade – це структурний шаблон, який забезпечує спрощений інтерфейс для складної системи. Замість того щоб безпосередньо взаємодіяти з багатьма підсистемами, клієнт отримує єдиний вхідний пункт через фасад. Це знижує складність коду, приховує реалізацію деталей і дозволяє змінювати підсистеми, не впливаючи на клієнта.

8) Bridge – це структурний шаблон, який відділяє абстракцію від її реалізації, дозволяючи їм змінюватися незалежно одна від одної. Він використовується для зменшення кількості класів, що виникають через комбінацію абстракцій і їх реалізацій, завдяки введенню містка між ними. Це сприяє гнучкості та спрощує розширення системи.

9) Template Method – це поведінковий шаблон, який визначає основу алгоритму у вигляді абстрактного методу, залишаючи реалізацію деяких кроків у руках підкласів. Цей підхід дозволяє забезпечити загальну структуру для всіх реалізацій, уникаючи дублювання коду та забезпечуючи можливість змінювати окремі деталі алгоритму.

Крок 2. Реалізація шаблону Facade:

1. ImageProcessingFacade

```
4 usages  1 implementation
public interface ImageProcessingFacade {
    1 usage  1 implementation
    Image applySobelOperator(Image image);
    1 usage  1 implementation
    Image applyGaussianBlur(Image image, double sigma);
    1 usage  1 implementation
    Image createCollage(Image[] images, int width, int height);
    1 usage  1 implementation
    Image applyGrayscaleEffect(Image image);
    1 usage  1 implementation
    Image adjustBrightness(Image image, double percentage);
    1 usage  1 implementation
    Image applyPixelation(Image image, int pixelSize);
}
```

Рис 1. Реалізація інтерфейсу ImageProcessingFacade

інтерфейс ImageProcessingFacade виступає фасадом для спрощення взаємодії з підсистемами обробки зображень. Він декларує набір методів для виконання різних операцій з зображеннями. Основна мета цього інтерфейсу — забезпечити єдиний, спрощений API для обробки зображень у проєкті. applySobelOperator(Image image): застосовує оператор Собеля для виділення країв на зображенні. applyGaussianBlur(Image image, double sigma): застосовує гаусівське розмиття із заданим значенням sigma. createCollage(Image[] images, int width, int height): створює колаж із кількох зображень із вказаними розмірами. applyGrayscaleEffect(Image image): переводить зображення у відтінки сірого. adjustBrightness(Image image, double percentage): змінює яскравість зображення на певний відсоток. applyPixelation(Image image, int pixelSize): додає пікселізацію із заданим розміром пікселя.

2. **ImageProcessingFacadeImpl**

```java
package com.proImg.image_editor.facade;

import com.proImg.image_editor.entities.Image;
import com.proImg.image_editor.service.ImageProcessingService;
import org.springframework.stereotype.Component;

@Component
public class ImageProcessingFacadeImpl implements ImageProcessingFacade {

    7 usages
    private final ImageProcessingService imageProcessingService;

    public ImageProcessingFacadeImpl(ImageProcessingService imageProcessingService) {
        this.imageProcessingService = imageProcessingService;
    }

    1 usage
    @Override
    public Image applySobelOperator(Image image) {
        return imageProcessingService.applySobelOperator(image);
    }

    1 usage
    @Override
    public Image applyGaussianBlur(Image image, double sigma) {
        return imageProcessingService.applyGaussianBlur(image, sigma);
    }

    @Override
    public Image createCollage(Image[] images, int width, int height) {
        return imageProcessingService.createCollage(images, width, height);
    }

    1 usage
    @Override
    public Image applyGrayscaleEffect(Image image) {
        return imageProcessingService.applyGrayscaleEffect(image);
    }

    1 usage
    @Override
    public Image adjustBrightness(Image image, double percentage) {
        return imageProcessingService.adjustBrightness(image, percentage);
    }

    1 usage
    @Override
    public Image applyPixelation(Image image, int pixelSize) {
        return imageProcessingService.applyPixelation(image, pixelSize);
    }
```

Рис 2. Реалізація класу ImageProcessingFacadeImpl

Цей клас містить реалізацію інтерфейсу ImageProcessingFacade. Основна функція цього класу — забезпечити єдиний доступ до різних методів обробки зображень через делегування викликів сервісу ImageProcessingService. Клас помічений як Spring-компонент за допомогою анотації @Component, що дозволяє автоматично інтегрувати його в контекст додатка. imageProcessingService: залежність від сервісу ImageProcessingService, яка забезпечує фактичну логіку обробки зображень. Передається через конструктор.

3. ImageProcessingService

```
@Service
public class ImageProcessingService {

    1 usage
    public Image applySobelOperator(Image image) {
        // Логіка застосування оператора Собеля
        System.out.println("Applying Sobel operator");
        return image;
    }

    1 usage
    public Image applyGaussianBlur(Image image, double sigma) {
        // Логіка гауссового розмиття
        System.out.println("Applying Gaussian blur with sigma: " + sigma);
        return image;
    }

    1 usage
    public Image createCollage(Image[] images, int width, int height) {
        // Логіка створення колажу
        System.out.println("Creating collage with dimensions: " + width + "x" + height);
        return new Image(); // Повертає нове зображення-колаж
    }
```

```
1 usage
public Image applyGrayscaleEffect(Image image) {
    // Логіка чорно-білого ефекту
    System.out.println("Applying grayscale effect");
    return image;
}

1 usage
public Image adjustBrightness(Image image, double percentage) {
    // Логіка корекції яскравості
    System.out.println("Adjusting brightness by: " + percentage + "%");
    return image;
}

1 usage
public Image applyPixelation(Image image, int pixelSize) {
    // Логіка пікселізації
    System.out.println("Applying pixelation with pixel size: " + pixelSize);
    return image;
}
```

Рис 3. Реалізація сервісу ImageProcessingService

Відповідає за основну логіку обробки зображень. Клас позначений як Spring-сервіс за допомогою анотації @Service, що робить його доступним для використання в інших компонентах програми через ін'єкцію залежностей.

4. **ImageController**

```java
@RestController
@RequestMapping("/api/images")
public class ImageController {

    7 usages
    private final ImageProcessingFacade imageProcessingFacade;
    6 usages
    private final ImageService imageService;

    @Autowired
    public ImageController(ImageProcessingFacade imageProcessingFacade, ImageService imageService) {
        this.imageProcessingFacade = imageProcessingFacade;
        this.imageService = imageService;
    }

    @PostMapping("/{id}/sobel")
    public ResponseEntity<Image> applySobelOperator(@PathVariable Long id) {
        Image image = imageService.findById(id);
        if (image == null) {
            return ResponseEntity.notFound().build();
        }
        Image processedImage = imageProcessingFacade.applySobelOperator(image);
        return ResponseEntity.ok(processedImage);
    }

    @PostMapping("/{id}/blur")
    public ResponseEntity<Image> applyGaussianBlur(@PathVariable Long id, @RequestParam double sigma) {
        Image image = imageService.findById(id);
        if (image == null) {
            return ResponseEntity.notFound().build();
        }
        Image processedImage = imageProcessingFacade.applyGaussianBlur(image, sigma);
        return ResponseEntity.ok(processedImage);
    }

    @PostMapping("/collage")
    public ResponseEntity<Image> createCollage(@RequestBody Image[] images, @RequestParam int width, @RequestParam int height) {
        Image collage = imageProcessingFacade.createCollage(images, width, height);
        return ResponseEntity.ok(collage);
    }

    @PostMapping("/{id}/grayscale")
    public ResponseEntity<Image> applyGrayscaleEffect(@PathVariable Long id) {
        Image image = imageService.findById(id);
        if (image == null) {
            return ResponseEntity.notFound().build();
        }
        Image processedImage = imageProcessingFacade.applyGrayscaleEffect(image);
        return ResponseEntity.ok(processedImage);
    }

    @PostMapping("/{id}/pixelate")
    public ResponseEntity<Image> applyPixelation(@PathVariable Long id, @RequestParam int pixelSize) {
        Image image = imageService.findById(id);
        if (image == null) {
            return ResponseEntity.notFound().build();
        }
        Image processedImage = imageProcessingFacade.applyPixelation(image, pixelSize);
        return ResponseEntity.ok(processedImage);
    }
```

Рис 4. Реалізація контроллеру ImageController

REST-контролер ImageController, відповідає за обробку HTTP-запитів, пов'язаних із зображеннями. Контролер взаємодіє з фасадом ImageProcessingFacade для виконання операцій обробки зображень і з сервісом ImageService для отримання даних про зображення. imageProcessingFacade: Використовується для виклику операцій обробки зображень через фасад. imageService: Забезпечує доступ до даних зображень через сервіс. Конструктор виконує ін'єкцію залежностей через анотацію @Autowired

/api/images/{id}/sobel: Застосовує оператор Собеля до зображення з вказаним id.

/api/images/{id}/blur: Застосовує гаусівське розмиття із заданим параметром sigma.

/api/images/collage: Створює колаж із кількох зображень із вказаними розмірами.

/api/images/{id}/grayscale: Перетворює зображення з вказаним id у відтінки сірого.

/api/images/{id}/brightness: Регулює яскравість зображення на певний відсоток.

/api/images/{id}/pixelate: Додає пікселізацію до зображення з заданим розміром

Детальніше код можна переглянути в репозиторії проекту:

https://github.com/Maxim-Khokhol/image-editor

Висновок: В цій лабораторній роботі я познайомився з паттернами «MEDIATOR», «FACADE», «BRIDGE», «TEMPLATE METHOD». Було програмно реалізовано паттерн **FACADE**. Реалізація патерну Facade дозволила значно спростити взаємодію клієнтського коду із системою, надавши єдиний уніфікований інтерфейс для роботи зі складними підсистемами. Завдяки цьому вдалося приховати деталі реалізації та знизити зв'язаність компонентів, що покращує модульність і гнучкість системи