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. class ImafeEffectFacade

```java
public class ImageEffectFacade {
    8 usages
    private ImageEffectState state;

    1 usage
    public void setEffect(String effectName, int... params) {
        switch (effectName.toLowerCase()) {
            case "grayscale" -> state = new GrayScaleEffect();
            case "sepia" -> state = new SepiaEffect();
            case "blur" -> state = new BlurEffect();
            case "pixelation" -> state = new PixelationEffect(params.length > 0 ? params[0] : 10);
            case "invert-colors" -> state = new InvertColorsEffect();
            case "threshold" -> state = new ThresholdEffect(params.length > 0 ? params[0] : 128);
            default -> throw new IllegalArgumentException("Unknown effect: " + effectName);
        }
    }

    public BufferedImage applyEffect(BufferedImage image) {
        if (state == null) {
            throw new IllegalStateException("No effect has been selected!");
        }
        return state.applyEffect(image);
    }
}
```

Рис 1. клас ImafeEffectFacade

Відповідає за управління ефектами для зображень, забезпечуючи єдиний інтерфейс для їх вибору та застосування. Він містить приватне поле state, яке зберігає поточний стан ефекту. Метод setEffect приймає назву ефекту та додаткові параметри (якщо потрібні) і налаштовує відповідний ефект. У цьому методі використовується конструкція switch, щоб створити об'єкт потрібного ефекту, наприклад, для "grayscale" встановлюється GrayScaleEffect, а для "pixelation" передається параметр розміру пікселя. Якщо назва ефекту не розпізнається, викидається виняток IllegalArgumentException.

Метод applyEffect застосовує поточний ефект до зображення, переданого як аргумент. Якщо ефект не був встановлений, викликається виняток IllegalStateException. Після цього метод викликає applyEffect поточного стану (ефекту), щоб модифікувати зображення відповідно до його логіки. Цей фасад дозволяє ізолювати логіку роботи з різними ефектами від основного коду, спрощуючи розширення системи новими ефектами.

2. method applyEffect

```
22      @PostMapping("/apply")
23      public ResponseEntity<String> applyEffect(@RequestParam String effect,
24                                                @RequestParam(required = false, defaultValue = "10") int pixelSize,
25                                                @RequestBody ImageRequest request) {
26          try {
27              Long imageId = request.getId();
28              ImageOriginator originator = originators.computeIfAbsent(imageId, id -> new ImageOriginator());
29              ImageCaretaker caretaker = caretakers.computeIfAbsent(imageId, id -> new ImageCaretaker());
30
31              if (caretaker.getInitialState(imageId) == null) {
32                  originator.setBase64(request.getBase64());
33                  caretaker.saveInitialState(imageId, originator.saveToMemento());
34              }
35
36              caretaker.saveState(imageId, originator.saveToMemento());
37
38              facade.setEffect(effect, pixelSize);
39              originator.applyEffect(facade::applyEffect);
40
41              caretaker.saveState(imageId, originator.saveToMemento());
42              return ResponseEntity.ok(originator.getBase64());
43          } catch (Exception e) {
44              return ResponseEntity.status(500).body("Error processing image: " + e.getMessage());
45          }
46      }
```

Рис 2. Метод applyEffect

У цьому коді фасад ImageEffectFacade використовується для спрощення роботи з ефектами. Він налаштовує потрібний ефект через метод setEffect, який отримує назву ефекту та додаткові параметри, такі як розмір пікселя. Ця логіка вибору ефекту та створення його об’єкта прихована у фасаді, що дозволяє контролеру не залежати від конкретних реалізацій ефектів. Після цього через метод applyEffect фасад застосовує обраний ефект до зображення, виконуючи всю необхідну обробку. Контролер передає цю дію через об’єкт originator, забезпечуючи збереження стану зображення до і після застосування ефекту. Завдяки фасаду контролер стає менш завантаженим логікою обробки, що підвищує його зрозумілість і дозволяє легко додавати нові ефекти без змін у самому контролері.

3. Class BlurEffect

```java
public class BlurEffect implements ImageEffectState {

    @Override
    public BufferedImage applyEffect(BufferedImage image) {
        System.out.println("Applying Strong Blur effect...");

        int width = image.getWidth();
        int height = image.getHeight();
        BufferedImage blurredImage = new BufferedImage(width, height, image.getType());

        int kernelSize = 15;
        int passes = 3;

        BufferedImage tempImage = image;

        for (int pass = 0; pass < passes; pass++) {
            BufferedImage currentImage = new BufferedImage(width, height, image.getType());

            for (int x = 0; x < width; x++) {
                for (int y = 0; y < height; y++) {
                    int[] rgb = getAverageColor(tempImage, x, y, kernelSize);
                    int blurredPixel = (rgb[0] << 16) | (rgb[1] << 8) | rgb[2];
                    currentImage.setRGB(x, y, blurredPixel);
                }
            }

            tempImage = currentImage;
        }
        return tempImage;
    }

    1 usage
    private int[] getAverageColor(BufferedImage image, int x, int y, int kernelSize) {
        int width = image.getWidth();
        int height = image.getHeight();
        int red = 0, green = 0, blue = 0, count = 0;

        int halfKernel = kernelSize / 2;

        for (int dx = -halfKernel; dx <= halfKernel; dx++) {
            for (int dy = -halfKernel; dy <= halfKernel; dy++) {
                int nx = x + dx;
                int ny = y + dy;

                if (nx >= 0 && nx < width && ny >= 0 && ny < height) {
                    int rgb = image.getRGB(nx, ny);
                    red += (rgb >> 16) & 0xFF;
                    green += (rgb >> 8) & 0xFF;
                    blue += rgb & 0xFF;
                    count++;
                }
            }
        }

        return new int[]{red / count, green / count, blue / count};
    }
}
```

Рис 3. клас BlurEffect

Клас BlurEffect реалізує інтерфейс ImageEffectState і відповідає за застосування ефекту розмиття до зображення. Основний метод applyEffect приймає об'єкт типу BufferedImage, що представляє зображення, до якого потрібно застосувати розмиття. Розмиття виконується за допомогою кількох проходів (параметр passes), які імітують більш глибокий ефект. У кожному проході створюється новий об'єкт BufferedImage, і кожен піксель у новому зображенні розраховується на основі середнього значення кольорів сусідніх пікселів, які визначаються ядром (kernelSize). Метод getAverageColor обчислює середнє значення кольорів для певного пікселя і його оточення. Розмір оточення визначається розміром ядра. У циклах метод проходить через сусідні пікселі, обчислюючи суму червоного, зеленого і синього компонентів для кожного пікселя, після чого ділить ці суми на кількість пікселів у ядрі, щоб отримати середній колір. Після кожного проходу розраховане зображення стає новим базовим для наступного проходу. У результаті розмиття поступово посилюється. Після завершення всіх проходів метод повертає розмите зображення. Цей клас імітує сильне розмиття шляхом багаторазового застосування усереднення кольорів на локальному рівні.

4. Class GrayScaleEffect

```
1 usage
public class GrayScaleEffect implements ImageEffectState {
    @Override
    public BufferedImage applyEffect(BufferedImage image) {
        System.out.println("Applying GrayScale effect...");
        BufferedImage grayImage = new BufferedImage(
                image.getWidth(),
                image.getHeight(),
                BufferedImage.TYPE_BYTE_GRAY
        );
        Graphics g = grayImage.getGraphics();
        g.drawImage(image,  x: 0,  y: 0,  observer: null);
        g.dispose();
        return grayImage;
    }
}
```

Рис 4. клас GrayScaleEffect

Клас GrayScaleEffect реалізує інтерфейс ImageEffectState і відповідає за перетворення кольорового зображення у відтінки сірого. Основний метод applyEffect приймає на вхід об'єкт BufferedImage, що представляє вихідне зображення.

У цьому методі створюється новий об'єкт BufferedImage з тими самими розмірами, що й вихідне зображення, але тип даних зображення встановлений як TYPE_BYTE_GRAY, що означає, що зображення зберігатиметься у відтінках сірого.
Далі за допомогою об'єкта Graphics метод малює вихідне зображення у новому зображенні, конвертуючи його в процесі у відтінки сірого. Після завершення роботи графічного об'єкта його ресурси звільняються за допомогою методу dispose. У результаті повертається нове зображення, яке вже оброблене та переведене у відтінки сірого.

5. Class InvertColorsEffect

```
7       @Override
8       public BufferedImage applyEffect(BufferedImage image) {
9           System.out.println("Applying Invert Colors effect...");
10
11          int width = image.getWidth();
12          int height = image.getHeight();
13          BufferedImage invertedImage = new BufferedImage(width, height, image.getType());
14
15          for (int x = 0; x < width; x++) {
16              for (int y = 0; y < height; y++) {
17                  int rgb = image.getRGB(x, y);
18                  int red = (rgb >> 16) & 0xFF;
19                  int green = (rgb >> 8) & 0xFF;
20                  int blue = rgb & 0xFF;
21                  int invertedRed = 255 - red;
22                  int invertedGreen = 255 - green;
23                  int invertedBlue = 255 - blue;
24                  int invertedRgb = (invertedRed << 16) | (invertedGreen << 8) | invertedBlue;
25                  invertedImage.setRGB(x, y, invertedRgb);
26              }
27          }
28
29          return invertedImage;
30      }
31  }
```

Рис 5. клас InvertColorsEffect

Клас реалізує ефект інверсії кольорів зображення. Основний метод applyEffect приймає об'єкт BufferedImage, що представляє вихідне зображення, і створює нове зображення з інвертованими кольорами. Спочатку визначаються розміри зображення (ширина та висота), після чого створюється новий об'єкт BufferedImage, який буде містити оброблене зображення.йде подвійний цикл, що проходить по кожному пікселю зображення. Для кожного пікселя зчитується його значення кольору (RGB) через метод getRGB. Це значення розділяється на три складові: червону, зелену і синю (red, green, blue).

Інверсія кольорів виконується для кожного компонента окремо: з 255 (максимальне значення кольору) віднімається значення відповідного компонента. У результаті колір кожного пікселя змінюється на протилежний (наприклад, чорний стає білим, червоний – бірюзовим тощо). Після цього інвертовані компоненти об’єднуються в одне значення RGB, яке записується у відповідний піксель нового зображення за допомогою методу setRGB.
Метод повертає нове зображення з інверсією кольорів, зберігаючи вихідне зображення без змін. Таким чином, цей клас забезпечує інверсію кольорової палітри кожного пікселя.

6. Class SephiaEffect

```
5   public class SepiaEffect implements ImageEffectState {
6       @Override
7       public BufferedImage applyEffect(BufferedImage image) {
8           System.out.println("Applying Sepia effect...");
9           BufferedImage sepiaImage = new BufferedImage(image.getWidth(), image.getHeight(), image.getType());
10
11          for (int x = 0; x < image.getWidth(); x++) {
12              for (int y = 0; y < image.getHeight(); y++) {
13                  int pixel = image.getRGB(x, y);
14                  int alpha = (pixel >> 24) & 0xff;
15                  int red = (pixel >> 16) & 0xff;
16                  int green = (pixel >> 8) & 0xff;
17                  int blue = pixel & 0xff;
18
19                  int tr = Math.min((int)(0.393 * red + 0.769 * green + 0.189 * blue), 255);
20                  int tg = Math.min((int)(0.349 * red + 0.686 * green + 0.168 * blue), 255);
21                  int tb = Math.min((int)(0.272 * red + 0.534 * green + 0.131 * blue), 255);
22
23                  sepiaImage.setRGB(x, y, rgb: (alpha << 24) | (tr << 16) | (tg << 8) | tb);
24              }
25          }
26          return sepiaImage;
27      }
28  }
```

Рис 6. клас SephiaEffect

Клас SepiaEffect реалізує інтерфейс ImageEffectState і відповідає за застосування ефекту сепії до зображення. Метод applyEffect приймає об'єкт BufferedImage, що представляє вихідне зображення, і створює нове зображення з ефектом сепії. Спочатку створюється новий об'єкт BufferedImage, який матиме ті ж розміри та тип, що й вихідне зображення. Потім код проходить по кожному пікселю зображення за допомогою вкладеного циклу. Для кожного пікселя зчитується його значення (ARGB), яке розділяється на компоненти: альфа (прозорість), червоний (red), зелений (green) і синій (blue). На основі цих

компонентів обчислюються нові значення для червоного, зеленого та синього кольорів, використовуючи формули для сепії. Формули використовують зважені суми компонентів RGB для створення теплого ефекту сепії. Значення обмежуються 255 за допомогою Math.min, щоб уникнути переповнення. Нові компоненти збираються назад в одне значення ARGB і встановлюються в новий піксель за допомогою методу setRGB. Після обробки всіх пікселів метод повертає нове зображення з ефектом сепії. Цей клас забезпечує теплий і м'який вигляд зображення, характерний для фотографій у стилі ретро.

7. Class ThresholdEffect

```
5   public class ThresholdEffect implements ImageEffectState {
6
        2 usages
7       private final int threshold;
8
        1 usage
9       public ThresholdEffect(int threshold) {
10          this.threshold = threshold;
11      }
12      @Override
13      public BufferedImage applyEffect(BufferedImage image) {
14          System.out.println("Applying Threshold effect...");
15          int width = image.getWidth();
16          int height = image.getHeight();
17          BufferedImage thresholdImage = new BufferedImage(width, height, image.getType());
18
19          for (int x = 0; x < width; x++) {
20              for (int y = 0; y < height; y++) {
21                  int rgb = image.getRGB(x, y);
22                  int red = (rgb >> 16) & 0xFF;
23                  int green = (rgb >> 8) & 0xFF;
24                  int blue = rgb & 0xFF;
25                  int brightness = (red + green + blue) / 3;
26                  int newColor = brightness > threshold ? 0xFFFFFF : 0x000000;
27
28                  thresholdImage.setRGB(x, y, newColor);
29              }
30          }
```

Рис 7. клас ThresholdEffect

Клас ThresholdEffect реалізує інтерфейс ImageEffectState і відповідає за застосування порогового ефекту до зображення. Цей ефект перетворює зображення на чорно-біле

залежно від рівня яскравості кожного пікселя. У методі applyEffect спочатку зчитуються розміри зображення, після чого створюється новий об'єкт BufferedImage, який зберігатиме результати обробки. Далі за допомогою вкладеного циклу код проходить через усі пікселі зображення. Для кожного пікселя витягується його значення кольору (RGB), яке розбивається на компоненти: червоний (red), зелений (green) і синій (blue).Обчислюється яскравість пікселя як середнє значення трьох компонентів. Якщо яскравість перевищує заданий поріг (threshold), піксель встановлюється в білий (0xFFFFFF); інакше — в чорний (0x000000). Отримане значення кольору записується в нове зображення. Метод повертає оброблене зображення, яке тепер має лише два кольори — білий і чорний. Цей ефект дозволяє виділити контури або створити двоколірні версії зображень, ґрунтуючись на заданому рівні порога.

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