



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

з дисципліни "Технології розроблення програмного забезпечення"
Тема: «Веб-сервіс автоматизованої перевірки програм лабораторного
практикуму на мові програмування Java »

Виконав
студент групи IA–33:
Заранік М.Ю.

Перевірив:
Мягкий М.Ю.

Зміст

Вступ	3
Мета роботи	3
Завдання роботи	3
Теоретичні відомості	4
Хід роботи	7
Питання до лабораторної роботи	21
Висновки	23

Вступ

Мета роботи

Вивчити структуру шаблонів «Mediator», «Facade», «Bridge», «Template method» та навчитися застосовувати їх в реалізації програмної системи.

Завдання

1. Ознайомитись з короткими теоретичними відомостями.
2. Реалізувати частину функціоналу робочої програми у вигляді класів та їхньої взаємодії для досягнення конкретних функціональних можливостей.
3. Реалізувати один з розглянутих шаблонів за обраною темою.
4. Реалізувати не менше 3-х класів відповідно до обраної теми.
5. 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

Будь-який патерн проектування, що використовується при створенні інформаційних систем, являє собою формалізований опис часто повторюваних завдань проектування, оптимальне рішення цих завдань та рекомендації щодо його застосування у різних ситуаціях [5]. Кожен патерн має загальновживане найменування і дозволяє застосовувати знайдене ефективне рішення багаторазово. Важливим початковим етапом роботи з патернами є коректне моделювання предметної області, що необхідно для належної формалізації постановки задачі та правильного вибору відповідних патернів.

Використання патернів проектування забезпечує низку очевидних переваг. Модель системи, створена з урахуванням патернів, виділяє ключові елементи та зв'язки, необхідні для розв'язання конкретної задачі, і водночас робить її більш зрозумілою та наочною для вивчення порівняно зі стандартною моделлю. Незважаючи на простоту та наочність, така модель дозволяє всебічно опрацювати архітектуру системи, використовуючи спеціальну мову опису. Застосування патернів підвищує адаптивність системи до змін вимог та спрощує її подальше доопрацювання.

Особливу важливість патерни мають при інтеграції інформаційних систем організації. Сукупність патернів формує єдиний словник проектування, який є уніфікованим засобом для взаємодії розробників. Таким чином, патерни є перевіреними роками практики «ескізами» архітектурних рішень, що дозволяють ефективно застосовувати їх у відповідних ситуаціях.

7.2.1. Шаблон «Mediator»

Призначення: дозволяє організувати взаємодію об'єктів через посередника, щоб компоненти не знали один про одного [6].

Проблема: багато візуальних компонентів на формі взаємодіють між собою, зв'язки ростуть у тисячі, і важко підтримувати код.

Рішення: створюється клас-посередник, через який проходять всі взаємодії. Компоненти звертаються до медіатора, а не безпосередньо один до одного.

Переваги: зменшує зв'язність, спрощує розширення та тестування.

Недоліки: медіатор може стати надскладним («God Object»).

7.2.2. Шаблон «Facade»

Призначення: створює єдиний інтерфейс доступу до підсистеми, приховуючи внутрішню складність [6]. Проблема: складна структура класів для роботи з різними протоколами та endpoint. Рішення: один клас-фасад надає простий інтерфейс, всі внутрішні класи залишаються прихованими. Переваги: спрощує роботу клієнтів, приховує внутрішню реалізацію. Недоліки: зменшує гнучкість прямого налаштування підсистеми..

7.2.3. Шаблон «Bridge»

Призначення: патерн «Bridge» (Міст) розділяє абстракцію і реалізацію, дозволяючи змінювати їх незалежно [6]. Проблема: графічний редактор з фігурами та різними способами відображення (екран, принтер, bitmap) може привести до експоненційного росту кількості підкласів. Рішення: створюються дві ієархії — абстракції фігур (Shape) і реалізації відображення (DrawApi). Фігури делегують операції відображення об'єкту DrawApi.

Переваги: Незалежна модифікація абстракції та реалізації.

Гнучкість і простота супроводу.

Недоліки: Більш складна структура через додаткові проміжні рівні.

7.2.4 Шаблон «Template Method»

Призначення: патерн «Template Method» дозволяє винести загальний алгоритм в абстрактний клас, залишаючи специфіку реалізації підкласам [6].

Проблема: при додаванні нових форматів відео (MPEG-2, MPEG-1, H.262) в існуючий відеоредактор код стає заплутаним через багато умовних розгалужень.

Рішення: виділити загальні кроки алгоритму в базовому класі, а специфічні частини реалізувати у методах, що перевизначаються в підкласах.

Переваги: Загальна логіка зберігається в базовому класі. Легке додавання нових варіантів алгоритму.

Недоліки: Може зростати кількість класів при великій кількості специфічних реалізацій.

Хід роботи

У ході виконання лабораторної роботи було реалізовано патерн проектування **Facade**, який використовується для спрощення взаємодії з підсистемою контейнерного виконання через Docker. Метою використання цього патерну є надання єдиного інтерфейсу для створення, запуску, збору логів та видалення контейнерів, приховуючи внутрішню складність підсистеми від зовнішнього коду.

1. Ознайомлення з теорією

Патерн Facade дозволяє клієнтам працювати з підсистемою через спрощений інтерфейс, не знаючи деталей внутрішньої реалізації окремих класів. Це забезпечує легке керування та масштабування системи.

У нашій системі підсистема складається з трьох основних класів:

- ContainerExecutorFacade — відповідає за створення та запуск контейнерів у Docker;
- ContainerLogFacade — забезпечує збір логів із запущених контейнерів;
- DockerClientFacade — об'єднує виклики інших фасадів, надаючи єдиний метод runJob(), який створює контейнер, чекає завершення, збирає логи та видає контейнер.

Зовнішній код використовує лише клас DockerClientFacade, що дозволяє виконувати повний цикл роботи з контейнером через один виклик методу runJob(), без потреби взаємодіяти з внутрішніми класами.

Переваги:

- Інкапсуляція складної логіки підсистеми від клієнтського коду;
- Простий і зрозумілий інтерфейс для використання підсистеми;
- Можливість змінювати внутрішню реалізацію підсистеми без впливу на зовнішній код.

Недоліки:

- Зменшення гнучкості при тонкому налаштуванні окремих компонентів підсистеми;
- Може утворитися “God Facade”, якщо надто багато функцій збирати в один клас.

2. Реалізація класів

Для реалізації патерну Facade у системі створено кілька класів, які спрощують роботу з Docker. **ContainerExecutorFacade** відповідає за створення, запуск та видалення контейнерів, приховуючи складність взаємодії з Docker API.

ContainerLogFacade збирає логи контейнера і повертає їх як текст, щоб клієнтський код не працював напряму з API.

DockerClientFacade об'єднує роботу цих класів і надає єдиний інтерфейс для роботи з контейнерами. Завдяки цьому клієнтський код працює лише з DockerClientFacade, не турбуючись про внутрішню реалізацію, що робить систему більш зрозумілою, гнучкою і простою в супроводі.

Послідовність виконання

Послідовність виконання системи з патерном Facade виглядає наступним чином. Зовнішній код, тобто клієнт, взаємодіє виключно з фасадом, наприклад, з класом DockerClientFacade, і не турбується про внутрішні деталі роботи системи: створення контейнера, його запуск, збір логів чи видалення після завершення роботи. Клієнт виконує лише прості виклики методів фасаду, отримуючи необхідний результат, не вникаючи в складність внутрішньої логіки.

Фасад виконує роль координатора між різними внутрішніми компонентами системи. Спочатку ContainerExecutorFacade відповідає за створення контейнера, його налаштування та запуск. Після цього ContainerLogFacade здійснює збір логів контейнера, обробку інформації та підготовку її для клієнта. Коли контейнер завершує роботу, ContainerExecutorFacade виконує його коректне видалення та звільнення ресурсів.

Такий підхід дозволяє приховати всю внутрішню складність системи, роблячи код клієнта максимально простим, зрозумілим і легким для використання. Крім того, фасад забезпечує незалежність клієнтського коду від змін внутрішньої реалізації: можна змінювати логіку створення контейнерів, способи збору логів чи внутрішню структуру без будь-якого впливу на зовнішній код. Завдяки цьому патерн Facade сприяє підвищенню гнучкості системи, покращенню її підтримуваності та зменшенню ймовірності помилок у клієнтському коді, оскільки всі складні операції інкапсульовані всередині фасаду.

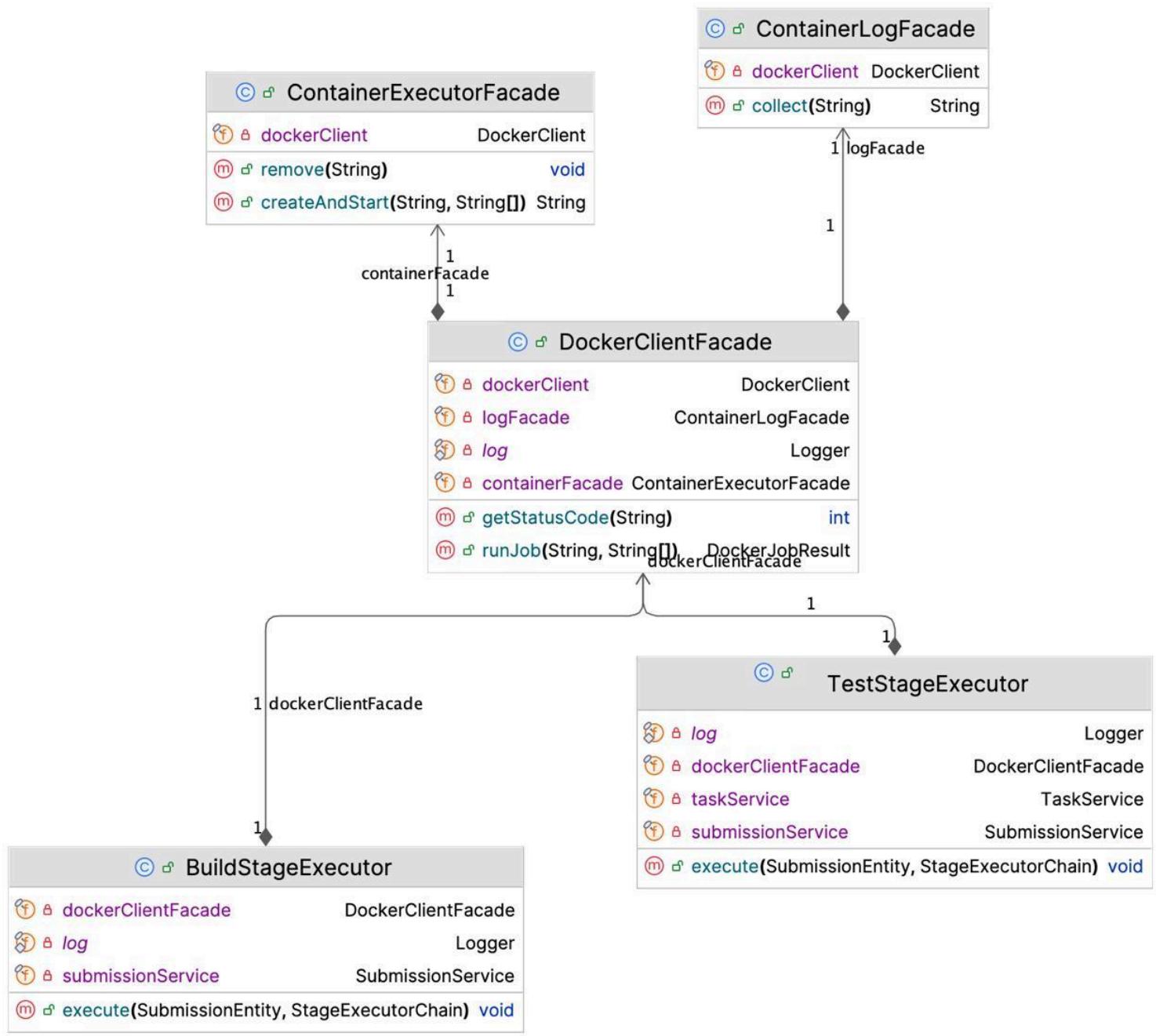


рис 1.1 Діаграмма класів системи

5. Код системи який реалізує паттерн Facade

ContainerExecutorFacade

```
package com.example.demo.service.executor.facade;

import com.github.dockerjava.api.DockerClient;
import com.github.dockerjava.api.model.HostConfig;
import lombok.RequiredArgsConstructor;
import org.springframework.stereotype.Component;

@Component
@RequiredArgsConstructor
public class ContainerExecutorFacade {
    private final DockerClient dockerClient;

    public String createAndStart(String name, String... args) {
        var container = dockerClient.createContainerCmd("java-maven-ci")
            .withCmd(args)
            .withHostConfig(HostConfig.newHostConfig().withNetworkMode("demo_default"))
            .withTty(true)
            .withName(name)
            .exec();

        dockerClient.startContainerCmd(container.getId()).exec();
        return container.getId();
    }

    public void remove(String id) {
        dockerClient.removeContainerCmd(id)
            .withRemoveVolumes(true)
            .withForce(true)
            .exec();
    }
}
```

ContainerLogFacade

```
package com.example.demo.service.executor.facade;

import com.github.dockerjava.api.DockerClient;
import com.github.dockerjava.api.async.ResultCallback;
import com.github.dockerjava.api.model.Frame;
import lombok.RequiredArgsConstructor;
import org.springframework.stereotype.Component;

import java.nio.charset.StandardCharsets;
import java.util.concurrent.TimeUnit;

@Component
@RequiredArgsConstructor
public class ContainerLogFacade {
```

```

private final DockerClient dockerClient;

public String collect(String containerId) throws InterruptedException {
    StringBuilder logs = new StringBuilder();
    dockerClient.logContainerCmd(containerId)
        .withStdOut(true)
        .withStdErr(true)
        .exec(new ResultCallback.Adapter<Frame>() {
            @Override
            public void onNext(Frame frame) {
                logs.append(new String(frame.getPayload(), StandardCharsets.UTF_8));
            }
        })
        .awaitCompletion(60, TimeUnit.SECONDS);
    return logs.toString();
}
}

```

DockerClientFacade

```

package com.example.demo.service.executor.facade;

import com.example.demo.model.submission.SubmissionEntity;
import com.github.dockerjava.api.DockerClient;
import com.github.dockerjava.api.command.WaitContainerResultCallback;
import lombok.RequiredArgsConstructor;
import lombok.extern.slf4j.Slf4j;
import org.springframework.stereotype.Component;

import java.util.concurrent.TimeUnit;

@Slf4j
@Component
@RequiredArgsConstructor
public class DockerClientFacade {

    private final DockerClient dockerClient;
    private final ContainerExecutorFacade containerFacade;
    private final ContainerLogFacade logFacade;

    public DockerJobResult runJob(String containerName, String... args) {
        String containerId = null;
        try {
            containerId = containerFacade.createAndStart(containerName, args);
            int status = getStatusCode(containerId);
            String logs = logFacade.collect(containerId);
            return new DockerJobResult(status, logs);
        } catch (Exception e) {
            log.error("Error running container: {}", e.getMessage(), e);
            return new DockerJobResult(-1, e.getMessage());
        } finally {
            if (containerId != null) containerFacade.remove(containerId);
        }
    }
}

```

```
}

public int getStatusCode(String containerId) {
    return dockerClient.waitContainerCmd(containerId)
        .exec(new WaitContainerResultCallback())
        .awaitStatusCode(60, TimeUnit.SECONDS);
}

public record DockerJobResult(Integer status, String logs) {}
```

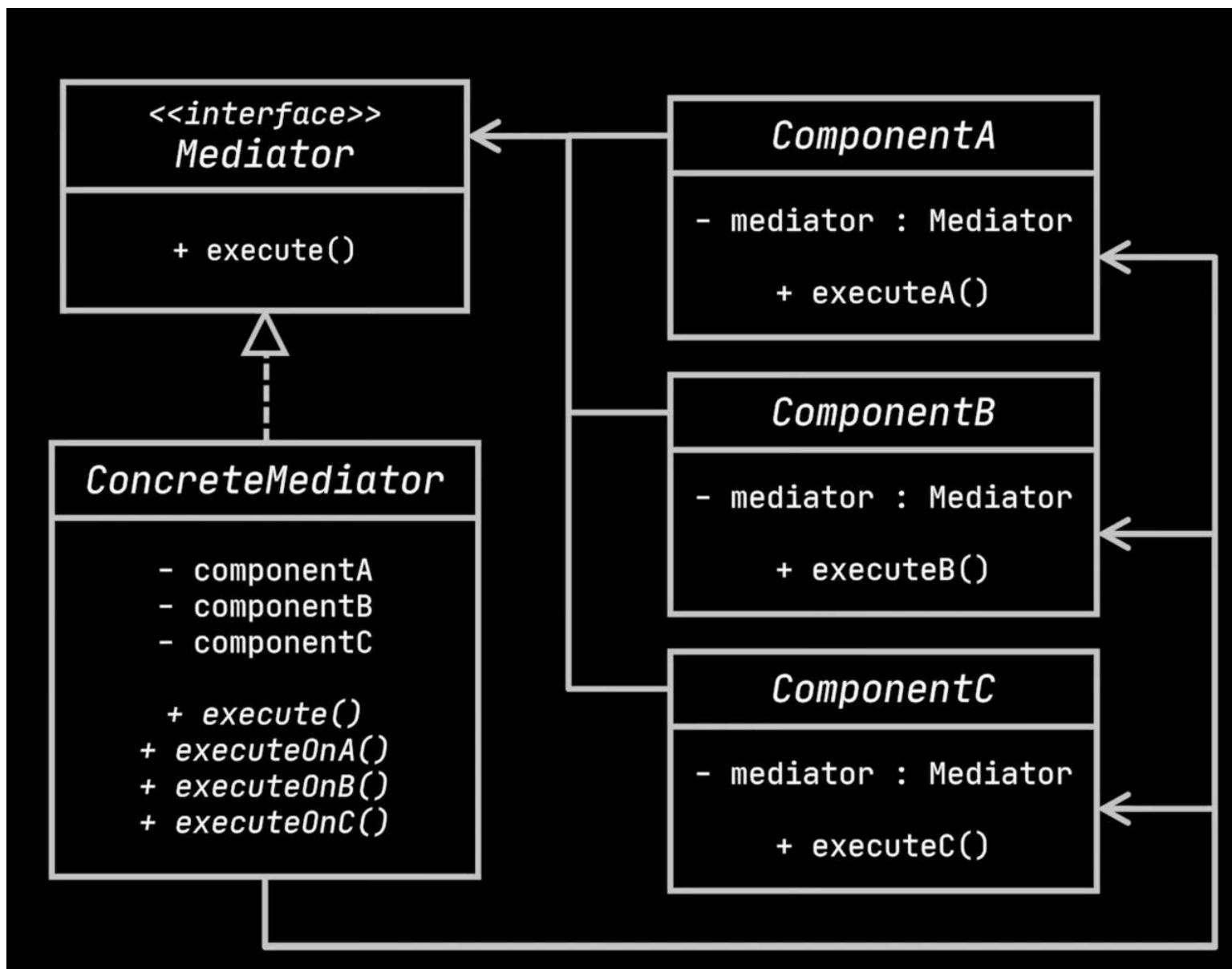
```
}
```

6. Питання до лабораторної роботи

1. Яке призначення шаблону «Посередник»?

Шаблон «Посередник» призначений для організації взаємодії між багатьма об'єктами системи через один центральний об'єкт — посередник. Його основна мета — зменшення прямої зв'язності між компонентами. Кожен об'єкт знає лише посередника, через якого відбувається обмін інформацією з іншими об'єктами.

2. Нарисуйте структуру шаблону «Посередник».



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

Mediator (Інтерфейс Посередника): Визначає єдиний інтерфейс для взаємодії об'єктів-Компонентів. На діаграмі це `<<interface>> Mediator`.

ConcreteMediator (Конкретний Посередник): Реалізує інтерфейс **Mediator**, містить посилання на всі Компоненти (`componentA`, `componentB`, `componentC`) і здійснює логіку їхньої координації.

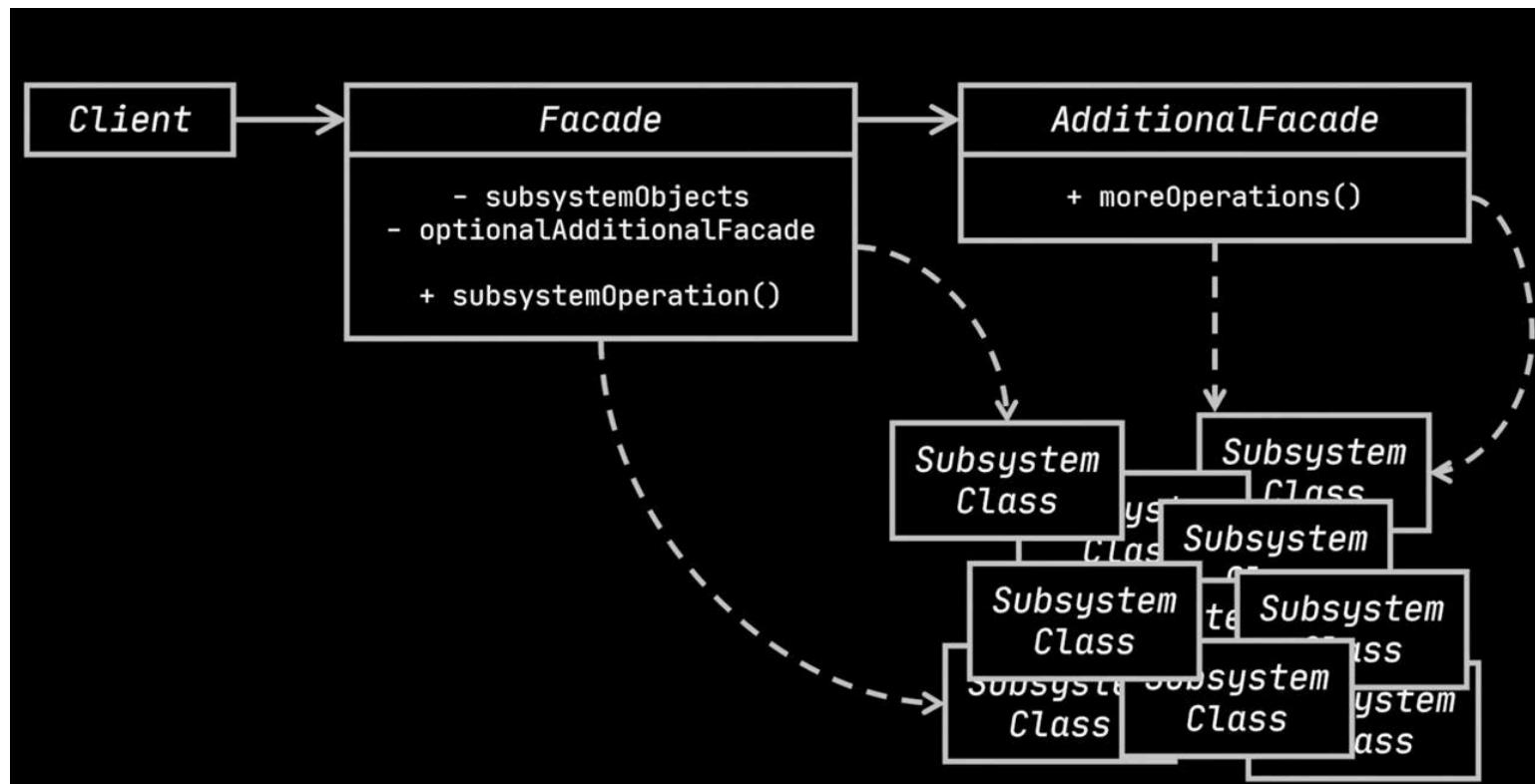
Компоненти: Це класи, які взаємодіють між собою, але не напряму. На діаграмі це

ComponentA, ComponentB, ComponentC.

4. Яке призначення шаблону «Фасад»?

Патерн «Фасад» створює єдиний спрощений інтерфейс для роботи зі складною підсистемою. Він дозволяє клієнту взаємодіяти з великою кількістю класів через один об'єкт, приховуючи внутрішню реалізацію. Це скорочує складність використання підсистеми, робить код більш зрозумілим і легким для підтримки.

5. Нарисуйте структуру шаблону «Фасад».



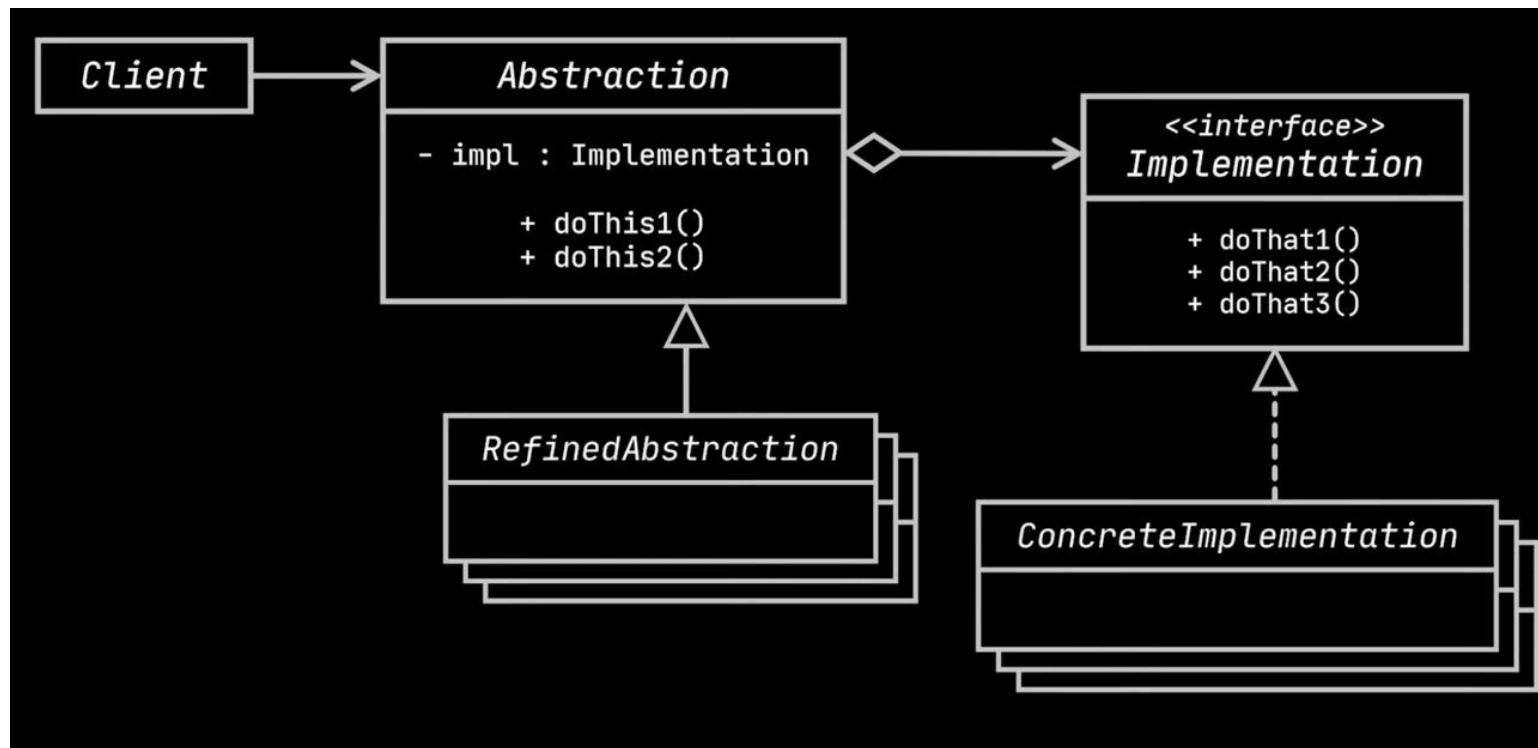
6. Які класи входять в шаблон «Фасад», та яка між ними взаємодія?

Клієнт викликає методи фасаду, який у свою чергу делегує виконання методів відповідним класам підсистеми. Підсистема виконує конкретну роботу, повертає результат фасаду, а клієнт отримує потрібний функціонал без прямого доступу до складних компонентів. Це дозволяє централізовано керувати роботою підсистеми, приховувати складну логіку та полегшувати інтеграцію з іншими модулями.

7. Яке призначення шаблону «Міст»?

Міст дозволяє розділити абстракцію та її реалізацію, щоб їх можна було розвивати незалежно. Патерн використовується тоді, коли існують різні абстракції та способи їх реалізації, і потрібно уникнути експоненціального зростання кількості підкласів. Міст забезпечує делегування роботи абстракції конкретній реалізації, що підвищує гнучкість і спрощує підтримку.

8. Нарисуйте структуру шаблону «Міст».



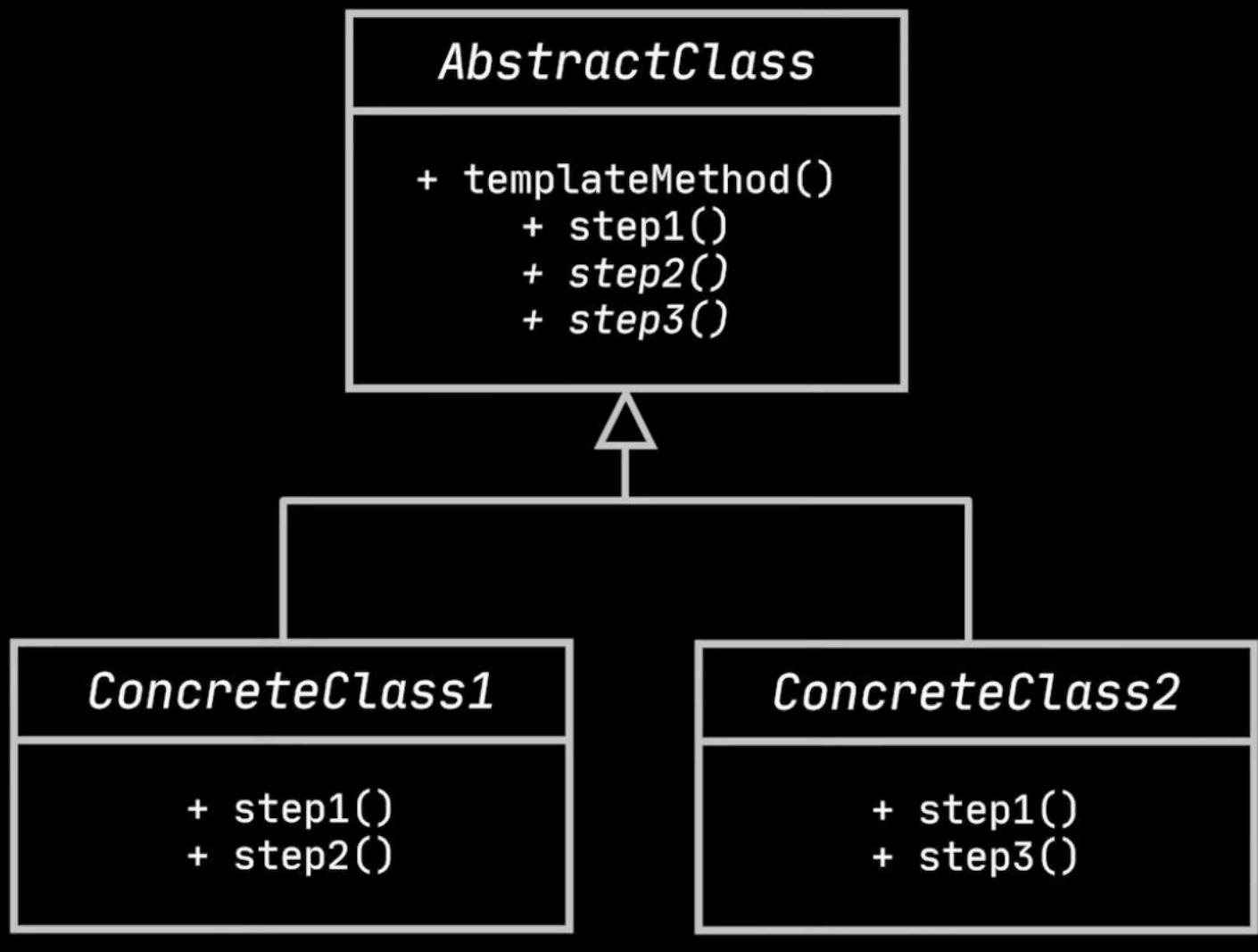
9. Які класи входять в шаблон «Міст», та яка між ними взаємодія?

Абстракція містить посилання на Implementor. При виклику методів абстракції вона делегує роботу конкретній реалізації. Таким чином, можна створювати нові абстракції та нові реалізації незалежно один від одного, без множення підкласів для кожної комбінації.

10. Яке призначення шаблону «Шаблонний метод»?

Шаблонний метод дозволяє визначити базовий алгоритм виконання операцій у абстрактному класі, залишаючи конкретну реалізацію окремих кроків підкласам. Це дозволяє уникнути дублювання коду, централізувати логіку та спростити додавання нових реалізацій алгоритмів. Шаблон використовується, коли більшість кроків алгоритму є спільними, а лише частина змінюється.

11. Нарисуйте структуру шаблону «Шаблонний метод».



12. Які класи входять в шаблон «Шаблонний метод», та яка між ними взаємодія?

Клієнт створює об'єкт *ConcreteClass* і викликає метод шаблону. Метод *Template Method* виконує загальні кроки алгоритму та делегує специфічні дії *ConcreteClass*. Це дозволяє змінювати конкретні кроки алгоритму без модифікації загальної структури.

13. Чим відрізняється шаблон «Шаблонний метод» від «Фабричного методу»?

Фабричний метод створює об'єкти, відокремлюючи логіку їх створення від клієнта. Шаблонний метод визначає послідовність виконання операцій, залишаючи реалізацію окремих кроків підкласам. Головна різниця в тому, що *Template Method* концентрується на алгоритмі, а *Factory Method* — на створенні об'єктів.

14. Яку функціональність додає шаблон «Міст»?

Міст дозволяє змінювати абстракцію та реалізацію незалежно одна від одної, зменшує кількість підкласів, дозволяє комбінувати різні абстракції з різними реалізаціями та підвищує гнучкість і розширюваність системи.

Висновок

У ході виконання лабораторної роботи було досягнуто поставленої мети — вивчено структуру та принципи роботи шаблонів проєктування «Mediator», «Facade», «Bridge» та «Template Method», а також реалізовано шаблон «Facade» у програмній системі. Було ознайомлено з теоретичними відомостями про кожен патерн, їх призначення, взаємодію класів та можливості спрощення архітектури.

Практична частина роботи включала реалізацію обраного шаблону за допомогою не менше ніж трьох класів, що дозволило наочно продемонструвати приховування складності внутрішньої системи, централізацію управління взаємодією між об'єктами та спрощення інтерфейсу для клієнтського коду. Виконані фрагменти коду та діаграма класів показують правильне застосування патернів у реалізації функціоналу системи, забезпечуючи чітку структуру та логіку роботи компонентів.

Завдяки виконаній роботі було підтверджено, що використання шаблонів проєктування підвищує гнучкість системи, повторне використання коду, зрозумілість архітектури та спрощує подальшу підтримку і розширення програмного забезпечення. Робота дозволила закріпити практичні навички побудови структурованого та масштабованого ПЗ, а також зрозуміти, як правильний вибір патернів сприяє оптимізації взаємодії об'єктів і спрощує керування складними системами.

Backend: <https://github.com/makszaranik/trpz-7-backend>

Frontend: <https://github.com/makszaranik/trpz-frontend>

Gateway: <https://github.com/makszaranik/trpz-gateway>