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
Технології розроблення програмного забезпечення
ШАБЛОНИ «MEDIATOR», «FACADE»,
«BRIDGE», «TEMPLATE METHOD»
CI-server

Виконав:
Студент групи ІА-22
Вдовиченко А.Х.

Перевірив:
Мягкий М. Ю

Київ 2024

Зміст

Тема: шаблони «MEDIATOR», «FACADE», «BRIDGE», «TEMPLATE METHOD»
Мета: ознайомитись з теоретичними відомостями, розробити частину функціоналу системи з використанням одного із шаблонів проектування.

Теоретичні відомості

Принципи проєктування відіграють ключову роль у створенні якісного програмного забезпечення. Принцип Don't Repeat Yourself (DRY) закликає уникати повторень у коді, оскільки це спрощує його читання, зменшує ймовірність появи помилок і полегшує внесення змін. Код без повторень є компактнішим і зрозумілішим, а помилки виправляються ефективніше, адже їх не потрібно шукати в різних місцях. Інший принцип – Keep it Simple, Stupid! (KISS) – наголошує на важливості простоти. Системи, побудовані з невеликих простих компонентів, працюють надійніше та легше піддаються обслуговуванню, ніж складні монолітні структури. Такий підхід сприяє створенню зрозумілого та функціонального коду, який легко сприймати і підтримувати.

Принцип You Only Load It Once! (YOLO) наголошує на тому, що ініціалізаційні змінні варто завантажувати один раз під час запуску програми. Це дозволяє уникнути зайвих операцій зчитування і прискорити роботу системи. Закон Парето, відомий як правило 80/20, акцентує на тому, що більшість результатів можна досягти, зосередившись на невеликій частині зусиль. Наприклад, 80% помилок у програмі можна виправити, усунувши лише 20% багів. Нарешті, принцип You Ain’t Gonna Need It (YAGNI) закликає відмовлятися від зайвої функціональності, яка може ніколи не знадобитися. Це зменшує складність системи, знижує витрати часу і ресурсів.

Шаблони проєктування допомагають організувати структуру програми. Шаблон "MEDIATOR" використовується для координації взаємодії між компонентами через окремий об’єкт, що дозволяє зменшити їх взаємозалежність. Це схоже на роботу диспетчера в аеропорту, який координує літаки, щоб уникнути хаосу. Водночас, надмірна кількість логіки в посереднику може ускладнити його підтримку.

Шаблон "FACADE" пропонує створення єдиного інтерфейсу для доступу до складної підсистеми. Він спрощує використання компонентів і приховує їхню складність. Наприклад, співробітник служби підтримки магазину виступає фасадом для клієнта, спрощуючи взаємодію з внутрішніми системами. Втім, фасад може стати надто громіздким, якщо в ньому зосереджується занадто багато функцій.

Шаблон "BRIDGE" дозволяє відокремлювати інтерфейс від його реалізації, що спрощує розширення системи. Наприклад, можна мати різні типи фігур і кольорів без створення безлічі підкласів для їх комбінацій. Проте додаткові класи, які вводяться в процесі реалізації, можуть ускладнити код.

Шаблон "TEMPLATE METHOD" дозволяє визначити загальний алгоритм у базовому класі, залишаючи реалізацію окремих кроків підкласам. Це схоже на будівництво типового будинку, де основні етапи стандартні, але є можливість додати унікальні елементи. Хоча шаблон полегшує повторне використання коду, він також може ускладнити підтримку, якщо алгоритм стає занадто деталізованим.

Хід роботи

Патерн Façade використовується для спрощення взаємодії зі складними підсистемами, надаючи єдиний інтерфейс для роботи з ними. Основною проблемою, яку вирішує цей патерн, є необхідність взаємодії з декількома класами та методами для виконання типових операцій. Без фасаду код стає перевантаженим залежностями та складним для розуміння, тестування й підтримки.

У нашому випадку фасад TestContainersFacade спрощує роботу з бібліотекою TestContainers для створення, запуску, зупинки контейнерів, копіювання файлів і управління їхнім станом. Він інкапсулює логіку взаємодії з сервісом DockerService і приховує низькорівневі деталі налаштування контейнерів, зокрема створення образів, передачу файлів та налаштування команд. (Рис 1.1)

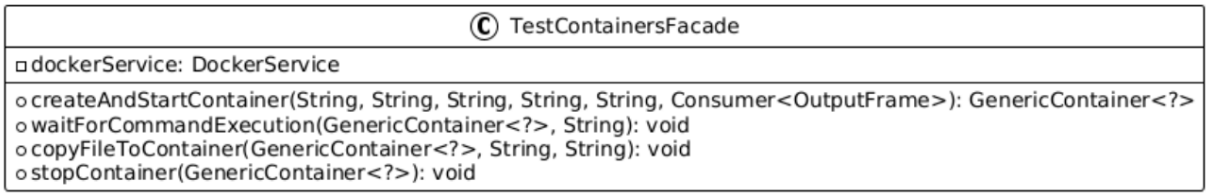

Рис. 1.1. – Загальна структура класу TestContainersFacade

Повний код класу буде наданий у додатку.

Це дозволяє іншим компонентам системи використовувати можливості TestContainers через єдиний інтерфейс, не турбуючись про внутрішні механізми.

Результатом роботи стала реалізація класу TestContainersFacade, який надає методи для базових операцій з контейнерами: створення та запуск, зупинка, копіювання файлів і очікування виконання команд. Такий підхід підвищує зручність роботи з контейнерами, зменшує залежності між компонентами системи та покращує масштабованість і тестованість коду.

Висновки

У цій лабораторній роботі було реалізовано загальний інтерфейс, що спрошує взаємодію з бібліотекою testcontainers за допомогою патерну Facade.

Вихідний код

```
package org.example.service.docker;

import jakarta.enterprise.context.ApplicationScoped;
import jakarta.inject.Inject;
import org.example.service.docker.DockerService;
import org.example.service.wait.ExecutionWaitStrategy;
import org.testcontainers.containers.GenericContainer;
import org.testcontainers.images.builder.ImageFromDockerfile;
import org.testcontainers.utility.MountableFile;

import java.util.function.Consumer;
import org.testcontainers.containers.output.OutputFrame;

@ApplicationScoped
public class TestContainersFacade {

    private final DockerService dockerService;

    @Inject
    public TestContainersFacade(DockerService dockerService) {
        this.dockerService = dockerService;
    }

    /**
     * Створення контейнера для виконання команди.
     *
     * @param language     - мова програмування
     * @param version      - версія мови
     * @param command      - команда для виконання
     * @param projectPath  - шлях до проекту
     * @param workingDir   - робоча директорія в контейнері
     * @param consumer     - логування виводу
     * @return Контейнер для подальшого управління
     */
    public GenericContainer<?> createAndStartContainer(String language, String
version, String command, String projectPath,
                                                        String workingDir,
Consumer<OutputFrame> consumer) {
        GenericContainer<?> container = dockerService.getContainer(language,
version, projectPath, workingDir, consumer, command);

        container.start();
        return container;
    }

    /**
     * Зупинити контейнер.
     *
     * @param container Контейнер для зупинки
     */
    public void stopContainer(GenericContainer<?> container) {
        if (container != null && container.isRunning()) {
            container.stop();
        }
    }

    /**
     * Додати файл до контейнера.
     *
     * @param container     Контейнер
```

```
     * @param sourcePath    Шлях на хості
     * @param destination   Шлях у контейнері
     */
    public void copyFileToContainer(GenericContainer<?> container, String
sourcePath, String destination) {
        container.copyFileToContainer(MountableFile.forHostPath(sourcePath),
destination);
    }

    /**
     * Очікувати виконання команди в контейнері.
     *
     * @param container Контейнер
     * @param command   Команда для виконання
     */
    public void waitForCommandExecution(GenericContainer<?> container, String
command) {
        container.waitingFor(new ExecutionWaitStrategy().withCommand(command));
    }
}
```