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 №4
Технології розроблення програмного забезпечення
ШАБЛОНИ «SINGLETON», «ITERATOR»,
«PROXY», «STATE», «STRATEGY»
CI-сервер

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

Перевірив:
Мягкий М. Ю

Київ 2024

Зміст

Теоретичні відомості 3

Хід роботи 4

Висновки 6

Вихідний код 6

Тема: шаблони «SINGLETON», «ITERATOR», «PROXY», «STATE», «STRATEGY»

Мета: ознайомитись з теоретичними відомостями, розробити частину функціоналу системи з використанням одного із шаблонів проектування.

Теоретичні відомості

Шаблони проєктування є важливою частиною розробки програмного забезпечення, адже вони надають можливість стандартизовано вирішувати типові проблеми, що виникають під час побудови складних систем. Основна ідея шаблонів полягає в тому, щоб надати розробникам універсальні рішення, які вже довели свою ефективність у багатьох схожих ситуаціях. Шаблони містять формалізований опис задачі, що вирішується, деталі її вирішення та рекомендації з реалізації. Вони не є готовим кодом, а скоріше описують архітектурний підхід, який може бути адаптований до конкретної системи.

Шаблон **Singleton** створений для ситуацій, коли необхідно забезпечити наявність тільки одного екземпляра певного класу. Це корисно в тих випадках, коли ресурс, який представляє клас, є спільним для всієї системи, наприклад, конфігурація програми або підключення до бази даних. Singleton гарантує, що екземпляр класу буде створений лише один раз, а доступ до нього можна отримати через спеціальний метод. Такий підхід дозволяє уникнути дублювання ресурсів і забезпечує централізований доступ.

Ітератор є шаблоном, який забезпечує зручний спосіб послідовного доступу до елементів колекції без розкриття її внутрішньої структури. Його основною метою є розділення алгоритмів обробки даних та структури зберігання, що дозволяє значно спростити роботу з різноманітними типами даних. Використання ітератора дозволяє абстрагуватися від деталей реалізації колекції, зосередившись на логіці обробки її елементів.

Proxy, або заступник, є потужним інструментом для створення об'єктів, які діють як замінники реальних об'єктів. Цей шаблон особливо корисний у випадках, коли необхідно додати рівень контролю до взаємодії із реальним об'єктом. Наприклад, Proxy може використовуватися для реалізації доступу до об'єкта через мережу, для кешування результатів обчислень або для перевірки дозволів. Завдяки Proxy можна знизити навантаження на систему або реалізувати додаткові функції без змін у коді самого об'єкта.

Шаблон **State** дозволяє змінювати поведінку об'єкта залежно від його внутрішнього стану. Він надає об'єкту можливість динамічно адаптувати свою логіку, забезпечуючи більшу гнучкість і модульність системи. Використання State особливо доречне, коли об'єкт має кілька станів і поведінка, пов'язана з цими станами, змінюється в ході виконання програми. Це зменшує кількість умовних операторів у коді та сприяє поліпшенню читабельності.

Шаблон **Strategy** спрямований на інкапсуляцію різних алгоритмів у межах окремих класів із можливістю взаємозаміни їх у системі. Використання Strategy дозволяє вибрати найбільш підходящий алгоритм залежно від умов, які змінюються під час виконання програми. Це зручний підхід для задач, які

потребують високої гнучкості у виборі способу виконання певної логіки, наприклад, сортування або шифрування даних.

Загалом, ці шаблони підвищують якість програмного забезпечення за рахунок його більшої структурованості, гнучкості та повторного використання. Їх вивчення та правильне застосування є невід'ємною частиною професійного зростання розробника.

Хід роботи

Мною було прийнято рішення використати шаблон проектування Singleton для ініціалізація єдиного екземпляру класу, що відповідатиме за обробку конфігурації збірки – YamlWorkflowProcessor.

У цій лабораторній роботі я використав патерн Singleton, тому що він забезпечує наявність лише одного екземпляра класу YamlWorkflowProcessor, який відповідає за обробку конфігурації збірки. Це було доцільно, оскільки клас не має власного стану, використовується як допоміжний і забезпечує централізований доступ до ресурсів, зокрема до файлу конфігурації збірки. Singleton дозволив уникнути дублювання ресурсів, спростив управління ними та забезпечив єдину точку доступу до функціональності.

Патерн State має такі переваги:

- Дозволяє змінювати поведінку об’єкта залежно від його внутрішнього стану.

- Полегшує читабельність коду, оскільки зменшує кількість умовних операторів (наприклад, if або switch).

- Підвищує гнучкість і модульність, розділяючи логіку поведінки за станами на окремі класи.

Однак я не використав патерн State з кількох причин:

- Відсутність множинних станів: Клас YamlWorkflowProcessor виконує статичні функції, які не залежать від стану об’єкта. Його завданням є лише обробка конфігураційного файлу, і жодна поведінка класу не змінюється залежно від його стану.

- Недоцільність у контексті задачі: У випадку обробки файлу workflow.yml не потрібно змінювати логіку залежно від будь-яких умов. Навпаки, використання патерну State могло б ускладнити архітектуру, розбивши просту функціональність на зайві класи, що було б надлишковим для цієї задачі.

- Фокус на стабільності функцій: Логіка обробки конфігурації має бути стабільною, передбачуваною і незалежною від станів чи контекстів виконання. Тому використання шаблону, який орієнтований на зміну поведінки, суперечить природі даної задачі.

- Пріоритет оптимізації: Використання Singleton забезпечило легкий доступ до єдиного екземпляра без необхідності вводити додаткові шари абстракції. Це позитивно вплинуло на продуктивність та зрозумілість коду.

Таким чином, хоча патерн State має свої переваги, його застосування у даній роботі не відповідало потребам задачі і було б зайвим.

Було реалізовано клас YamlWorkflowProcessor з приватним конструктором, щоб уникнути створення об'єктів ззовні. Клас має метод getInstance який вертає єдиний екземпляр класу, що був створений до цього. (Рис. 1.1)

```
public class YamlWorkflowProcessor {  6 usages
    private static YamlWorkflowProcessor instance;  4 usages

    private YamlWorkflowProcessor() {  1 usage
    }

    public static YamlWorkflowProcessor getInstance() {  1 usage
        if (instance == null) {
            synchronized (YamlWorkflowProcessor.class) {
                if (instance == null) {
                    instance = new YamlWorkflowProcessor();
                }
            }
        }
        return instance;
    }
```

Рисунок 1.1. Реалізація шаблону Singleton для класу YamlWorkflowProcessor

Цей клас містить в собі бізнес-логіку, що відповідає за пошук та обробку файлу workflow.yml у проекті, для якого ми запускаємо збірку. Цей клас не має свого стану, а використовується виключно як допоміжний клас, тому дуже зручно мати єдиний екземпляр на весь проект.

Приклад використання цього класу в коді. (Рис 1.2)

```
public class BuildService {  6 usages

    private final OpenConnections connections;  2 usages
    private final BuildRepository buildRepository;  4 usages
    private final GitService gitService;  2 usages
    private final DockerService dockerService;  3 usages
    private final YamlWorkflowProcessor yamlWorkflowProcessor = YamlWorkflowProcessor.getInstance();  3 usages
```

Рисунок 1.2 Використання YamlWorkflowProcessor в BuildService

Таким чином було реалізовано функціонал обробки конфігурацій збірки за допомогою шаблону Сінглтон.

Висновки

У цій лабораторній роботі було реалізовано функціонал обробки конфігурацій збірки за допомогою шаблону Сінглтон.

Вихідний код

```
package org.example.service;

import org.yaml.snakeyaml.Yaml;

import java.io.FileInputStream;
import java.io.IOException;
import java.nio.file.Path;
import java.util.Map;

import static org.example.service.utility.ServiceConstants.WORKFLOW_FILENAME;
```

```
public class YamlWorkflowProcessor {
    private static YamlWorkflowProcessor instance;

    private YamlWorkflowProcessor() {
    }

    public static YamlWorkflowProcessor getInstance() {
        if (instance == null) {
            synchronized (YamlWorkflowProcessor.class) {
                if (instance == null) {
                    instance = new YamlWorkflowProcessor();
                }
            }
        }
        return instance;
    }

    public Map<String, Object> extractWorkflowData(String projectDir) {
        try (var yamlFile = new FileInputStream(Path.of(projectDir,
WORKFLOW_FILENAME).toString())) {
            Yaml yaml = new Yaml();
            return yaml.load(yamlFile);
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }

    public String getCommandFromWorkflowData(Map<String, Object> workflowData) {
        return workflowData.get("run").toString();
    }

    public Map<String, String> getWithVariablesFromWorkflowData(Map<String,
Object> workflowData) {
        return (Map<String, String>) workflowData.get("with");
    }
}
```

```
package org.example.service;

import io.quarkus.websockets.next.OpenConnections;
import jakarta.enterprise.context.ApplicationScoped;
import jakarta.inject.Inject;
import jakarta.transaction.Transactional;
import org.example.domain.Build;
import org.example.domain.BuildStatus;
import org.example.domain.Project;
import org.example.repository.BuildRepository;
import org.example.service.cosumers.LogFileConsumer;
import org.example.service.cosumers.WebSocketConsumer;
import org.testcontainers.containers.ContainerLaunchException;
import org.testcontainers.images.builder.ImageFromDockerfile;

import java.time.Instant;
import java.util.Map;
import java.util.Optional;
import java.util.UUID;

import static
org.example.service.utility.ServiceConstants.WORKSPACE_DIR_PATTERN;

@ApplicationScoped
public class BuildService {
```

```
    private final OpenConnections connections;
    private final BuildRepository buildRepository;
    private final GitService gitService;
    private final DockerService dockerService;
    private final YamlWorkflowProcessor yamlWorkflowProcessor =
YamlWorkflowProcessor.getInstance();

    @Inject
    public BuildService(BuildRepository buildRepository, OpenConnections
connections,
                        GitService gitService, DockerService dockerService) {
        this.buildRepository = buildRepository;
        this.connections = connections;
        this.gitService = gitService;
        this.dockerService = dockerService;
    }

    public Optional<Build> findById(Long id) {
        return buildRepository.findByIdOptional(id);
    }

    @Transactional
    public void runProjectBuild(String id) {
        Build build =
findById(Long.parseLong(id)).orElseThrow(IllegalArgumentException::new);
        Project project = build.getProject();
        var consumer = new WebSocketConsumer(id, connections).andThen(new
LogFileConsumer(id, build.getLogFilePath()));
        var tmpProjectDirName = "%s-%s".formatted(project.getName(),
UUID.randomUUID());
        var clonedProjectDir = gitService.cloneRepository(tmpProjectDirName,
project.getRepositoryUrl(), build.getBranch());

        Map<String, Object> workflowData =
yamlWorkflowProcessor.extractWorkflowData(clonedProjectDir);
        String command =
yamlWorkflowProcessor.getCommandFromWorkflowData(workflowData);
        Map<String, String> withVariables =
yamlWorkflowProcessor.getWithVariablesFromWorkflowData(workflowData);

        ImageFromDockerfile image =
dockerService.configureImage(withVariables.get("language"),
withVariables.get("version"));
        var workingDir = WORKSPACE_DIR_PATTERN.formatted(tmpProjectDirName);
        try (var container = dockerService.configureContainer(image,
clonedProjectDir, workingDir, consumer, command)) {
            container.start();
            build.setBuildStatus(BuildStatus.SUCCESS);
            build.setEndTime(Instant.now());
            buildRepository.persist(build);;
        } catch (ContainerLaunchException launchException) {
            build.setBuildStatus(BuildStatus.FAILURE);
            build.setEndTime(Instant.now());
            buildRepository.persist(build);;
        };
    }
}
```