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 №6
Технології розроблення програмного забезпечення
ШАБЛОНИ «Abstract Factory»,
«Factory Method», «Memento»,
«Observer», «Decorator»
CI-server

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

Перевірив:
Мягкий М. Ю

Київ 2024

Зміст

Тема: шаблони «Abstract Factory», «Factory Method», «Memento»,«Observer», «Decorator»

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

Теоретичні відомості

Принципи SOLID є основоположними правилами об’єктно-орієнтованого програмування (ООП) і розробки програмного забезпечення. Вони були запропоновані Робертом Мартіном (Robert C. Martin) для покращення гнучкості, масштабованості та підтримуваності коду. Кожен принцип має свою специфіку і допомагає уникнути типових помилок у проектуванні.

1. Принцип єдиного обов'язку (Single Responsibility Principle - SRP)

Кожен клас повинен мати тільки одну причину для зміни.

Це означає, що клас повинен відповідати лише за одну конкретну функціональність або частину бізнес-логіки. Якщо клас виконує декілька завдань, він стає складним для підтримки і тестування, оскільки зміни в одній його частині можуть вплинути на інші.

2. Принцип відкритості/закритості (Open/Closed Principle - OCP)

Код повинен бути відкритим для розширення, але закритим для зміни.

Це означає, що нова функціональність має додаватися через розширення існуючого коду, а не шляхом його модифікації.

3. Принцип підстановки Лісков (Liskov Substitution Principle - LSP)

Об’єкти похідного класу повинні бути замінними на об’єкти базового класу без порушення коректності програми.

Це означає, що підкласи повинні поводитися так само, як і батьківські класи, і не змінювати їхню поведінку.

4. Принцип поділу інтерфейсу (Interface Segregation Principle - ISP)

Клієнти не повинні залежати від інтерфейсів, які вони не використовують.

Це означає, що великий інтерфейс слід розбивати на кілька дрібних, які містять лише необхідні методи для конкретного клієнта.

5. Принцип інверсії залежностей (Dependency Inversion Principle - DIP)

Модулі високого рівня не повинні залежати від модулів низького рівня. Обидва повинні залежати від абстракцій. Абстракції не повинні залежати від деталей. Деталі мають залежати від абстракцій. Цей принцип рекомендує використовувати інтерфейси або абстрактні класи для зв'язку між компонентами програми, щоб уникнути жорсткої залежності.

Шаблон «Abstract Factory» надає механізм для створення груп взаємопов'язаних об'єктів без зазначення їх конкретних класів. Він дозволяє розробникам створювати сімейства об'єктів, забезпечуючи узгодженість та спрощуючи заміну продуктів в майбутньому.

Шаблон «Factory Method» використовується для делегування створення об'єктів підкласам. Це забезпечує більшу гнучкість і розширюваність системи, оскільки підкласи можуть самостійно визначати, які об'єкти створювати.

Шаблон «Memento» орієнтований на збереження та відновлення стану об'єктів без порушення інкапсуляції. Його застосування корисне в програмах, де необхідно реалізувати функціональність скасування дій або історії змін.

Шаблон «Observer» організовує механізм спостереження, що дозволяє одному об'єкту (суб'єкту) повідомляти кілька інших об'єктів (спостерігачів) про зміни в його стані. Цей шаблон підтримує слабку залежність між об'єктами, що спрощує їхню взаємодію.

Шаблон «Decorator» надає можливість додавати нові функціональні можливості об'єктам динамічно. Його перевагою є можливість створювати ієрархію об'єктів із різними рівнями функціональності без необхідності створювати велику кількість підкласів.

Хід роботи

У цій лабораторній роботі я використав патерн Factory Method, оскільки він дозволяє легко додавати нові типи об’єктів (у нашому випадку — Docker-образи для різних мов програмування) без зміни основного коду. Реалізація інтерфейсу DockerImageFactory через окремі класи, такі як JavaDockerImageFactory і CSharpDockerImageFactory, забезпечила зручну модульність і простоту розширення функціональності.

Патерн Декоратор має такі переваги:

- Дозволяє динамічно додавати нові функціональні можливості об'єктам.
- Сприяє зменшенню кількості підкласів за рахунок композиції замість наслідування.
- Підтримує принцип відкритості/закритості (OCP), оскільки дозволяє додавати функціональність без модифікації існуючого коду.

Однак я його не використав з наступних причин:

- Відсутність потреби в динамічному додаванні функціональності: У задачі, що розв'язувалася, логіка створення Docker-образів залежить від статичних параметрів (мова програмування, версія тощо). Потреби динамічно змінювати або додавати функціонал під час виконання програми не було.
- Пріоритетність модульності через фабричний метод: Використання патерну Factory Method дозволило ефективно організувати створення об’єктів без ускладнення архітектури. У нашому випадку це було більш природним рішенням, оскільки акцент був на відокремленні логіки створення об’єктів для різних мов у відповідні фабрики.
- Непотрібність динамічної ієрархії об'єктів: Патерн Декоратор доцільний, коли об’єкт потребує багаторазового нашарування функціональності. У нашому випадку поведінка Docker-образів визначається конкретним класом фабрики і не потребує подальших модифікацій під час виконання програми.
- Спрощення коду: Використання Декоратора у цьому контексті могло б ускладнити реалізацію через необхідність створення додаткових класів-

декораторів для різних випадків використання, що є надмірним для даної задачі.

Таким чином, використання патерну Декоратор не відповідало вимогам даної лабораторної роботи, оскільки задача не потребувала динамічного додавання функціональності або створення складної ієрархії об’єктів.

У контексті розробки CI-серверу ключовою проблемою є створення об'єктів, які залежать від багатьох факторів, наприклад, від мови програмування, версії середовища, або інших специфічних параметрів. Створення таких об'єктів вручну може призвести до дублювання коду, складнощів у підтримці та ризику помилок, якщо будуть додані нові мови чи специфікації.

Наприклад, для побудови контейнерів різних мов (Java, C# тощо) необхідно враховувати

- Шлях до відповідного Dockerfile.
- Специфічні аргументи для збірки образу.
- Логіку для вибору версії за замовчуванням.

Без структурованого підходу логіка для кожної мови могла б стати громіздкою, розкиданою по коду і важко розширюваною. Це знижує масштабованість і ускладнює впровадження нових мов.

Шаблон «Фабричний метод» пропонує інкапсулювати логіку створення специфічних об'єктів у окремих класах, які реалізують спільний інтерфейс. Це дозволяє розділити відповідальність, полегшити підтримку, забезпечити розширюваність та зменшити дублювання коду. (Рис 1.1)

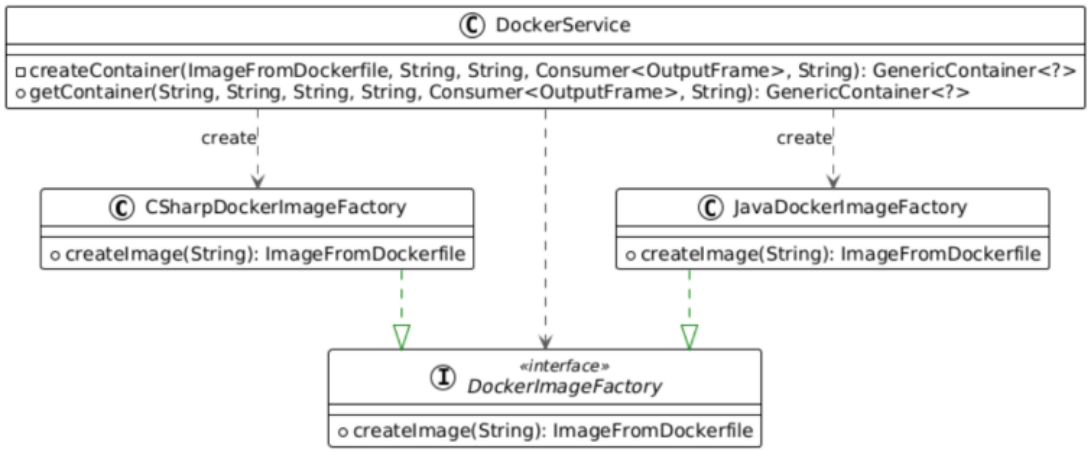

Рис. 1.1 - Використання фабричного методу для створення оточення

У наведеному прикладі інтерфейс DockerImageFactory визначає контракт для створення Docker-образів. Класи, такі як JavaDockerImageFactory і CSharpDockerImageFactory, реалізують цей інтерфейс, інкапсулюючи специфіку створення образів для відповідних мов (Рис 1.2). А у класі DockerService

використовується умовний оператор, щоб вибрати потрібну фабрику залежно від вхідного параметра language. Це дозволяє динамічно створювати образи для різних мов без дублювання логіки.

```
public class JavaDockerImageFactory implements DockerImageFactory {

    @Override  1 usage
    public ImageFromDockerfile createImage(String version) {
        if (version == null || version.isEmpty()) {
            version = "21";
        }
        return new ImageFromDockerfile()
                .withDockerfile(Paths.get(JAVA_DOCKERFILE_PATH))
                .withBuildArg(JAVA_VERSION_ARG, version);
    }

}
```

Рис. 1.2 - Одна з Реалізацій інтерфейсу DockerImageFactory для Java

Висновки

У цій лабораторній роботі було реалізовано функціонал вибору правильного ізольованого оточення збірки за допомогою шаблону проектування Factory Method.

Вихідний код

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

import org.testcontainers.images.builder.ImageFromDockerfile;

public interface DockerImageFactory {
    ImageFromDockerfile createImage(String version);
}
```

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

import org.testcontainers.images.builder.ImageFromDockerfile;

import java.nio.file.Paths;

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

public class CSharpDockerImageFactory implements DockerImageFactory {

    @Override
    public ImageFromDockerfile createImage(String version) {
```

```
        if (version == null || version.isEmpty()) {
            version = "7.0";
        }
        return new ImageFromDockerfile()
                .withDockerfile(Paths.get(C_SHARP_DOCKERFILE_PATH))
                .withBuildArg(C_SHARP_VERSION_ARG, version);
    }

}
```

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

import org.testcontainers.images.builder.ImageFromDockerfile;
import java.nio.file.Paths;

import static org.example.service.utility.ServiceConstants.JAVA_DOCKERFILE_PATH;
import static org.example.service.utility.ServiceConstants.JAVA_VERSION_ARG;

public class JavaDockerImageFactory implements DockerImageFactory {

    @Override
    public ImageFromDockerfile createImage(String version) {
        if (version == null || version.isEmpty()) {
            version = "21";
        }
        return new ImageFromDockerfile()
                .withDockerfile(Paths.get(JAVA_DOCKERFILE_PATH))
                .withBuildArg(JAVA_VERSION_ARG, version);
    }

}
```

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

import jakarta.enterprise.context.ApplicationScoped;
import org.example.service.wait.ExecutionWaitStrategy;
import org.testcontainers.containers.GenericContainer;
import org.testcontainers.containers.output.OutputFrame;
import org.testcontainers.images.builder.ImageFromDockerfile;
import org.testcontainers.utility.MountableFile;

import java.util.function.Consumer;

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

@ApplicationScoped
public class DockerService {


    public GenericContainer<?> getContainer(String language, String version,
String clonedProjectPath,
                                            String workingDir,
Consumer<OutputFrame> consumer, String command) {
        DockerImageFactory factory = switch (language.toUpperCase()) {
            case JAVA -> new JavaDockerImageFactory();
            case C_SHARP -> new CSharpDockerImageFactory();
            default -> throw new IllegalStateException("Unsupported language: "
+ language);
        };
        ImageFromDockerfile image = factory.createImage(version);
        return createContainer(image, clonedProjectPath, workingDir, consumer,
command);
    }
```

```
    private GenericContainer<?> createContainer(ImageFromDockerfile image,
String clonedProjectPath, String workingDir,
                                                Consumer<OutputFrame> consumer,
String command) {
        return new GenericContainer<>(image)

.withCopyFileToContainer(MountableFile.forHostPath(clonedProjectPath),
workingDir)
                .withWorkingDirectory(workingDir)
                .withLogConsumer(consumer)
                .withCommand(command)
                .waitingFor(new ExecutionWaitStrategy().withCommand(command));
    }
}
```