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

з дисципліни "Технології розроблення програмного забезпечення"

Тема: «Веб-сервіс автоматизованої перевірки програм лабораторного практикуму на мові програмування Java »

Виконав

студент групи ІА–33:

Заранік М.Ю

Перевірив:

Мягкий М.Ю.

Київ 2025

Зміст

Вступ

Мета роботи

Вивчити структуру шаблонів «Abstract Factory», «Factory Method», «Memento», «Observer», «Decorator» 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
3. їхньої взаємодії для досягнення конкретних функціональних можливостей.
4. • Реалізувати один з розглянутих шаблонів за обраною темою.
5. • Реалізувати не менше 3-х класів відповідно до обраної теми.
6. • Підготувати звіт щодо виконання лабораторної роботи. Поданий звіт
7. повинен містити: діаграму класів, яка представляє використання шаблону в
8. реалізації системи, навести фрагменти коду по реалізації цього шаблону

Теоретичні відомості

Будь-який патерн проєктування, що використовується при створенні інформаційних систем, являє собою формалізований опис часто повторюваних завдань проєктування, оптимальне рішення цих завдань та рекомендації щодо його застосування у різних ситуаціях [5]. Крім того, патерн проєктування має обов’язкове загальновживане найменування. Правильно оформлений патерн дозволяє, знайшовши одного разу ефективне рішення, застосовувати його багаторазово. Важливим початковим етапом роботи з патернами є коректне моделювання предметної області, що необхідно як для належної формалізації постановки задачі, так і для вибору відповідних патернів.

Відповідне використання патернів проєктування забезпечує розробнику низку очевидних переваг. Модель системи, створена з урахуванням патернів, фактично виділяє ті елементи та зв’язки, які є ключовими для розв’язання конкретної задачі. До того ж така модель є більш зрозумілою та наочною для вивчення порівняно зі стандартною моделлю. Проте, попри простоту та наочність, вона дозволяє ґрунтовно та всебічно опрацювати архітектуру системи, використовуючи спеціальну мову опису. Застосування патернів підвищує адаптивність системи до змін вимог та полегшує її подальше доопрацювання. Окремо слід відзначити важливість патернів при інтеграції інформаційних систем організації. Крім того, сукупність патернів фактично утворює єдиний словник проєктування, який, будучи уніфікованим засобом, є незамінним для взаємодії розробників. Отже, патерни являють собою, перевірені роками практики в різних компаніях і проєктах, «ескізи» архітектурних рішень, які зручно застосовувати у відповідних ситуація

1. Шаблон «Abstract Factory»

Призначення шаблону: «Абстрактна фабрика» використовується для створення груп пов’язаних об’єктів без необхідності вказувати їх конкретні класи [6]. Для цього визначається спільний інтерфейс фабрики (AbstractFactory), а також реалізації цього інтерфейсу для різних сімейств продуктів. Прикладом застосування такого підходу є ADO.NET, де існує універсальний клас DbProviderFactory, який може створювати об’єкти типів DbConnection, DbDataReader, DbAdapter тощо. Реалізації цієї фабрики, як-от SqlProviderFactory, SqlConnection, SqlDataReader, SqlAdapter, відповідають конкретним типам баз даних. Таким чином, якщо програмі потрібно працювати з різними СУБД, достатньо використовувати базові класи (Db.) і під час ініціалізації підставити потрібну фабрику (Factory = new SqlProviderFactory()).

Даний шаблон допомагає організувати роботу з подібними об’єктами (сімействами), наприклад, класами доступу до баз даних, та забезпечує можливість легко замінювати одне сімейство іншим (робота з Oracle виглядає так само, як із SQL Server). Водночас недоліком є складність розширення: при додаванні нового методу потрібно змінювати всі фабрики й створювати відповідні класи для нових типів об’єктів.

2.Шаблон «Factory Method»

Призначення: Шаблон «Фабричний метод» задає інтерфейс для створення об’єктів певного базового типу [6]. Він корисний, коли необхідно мати можливість створювати не лише об’єкти базового класу, а й екземпляри його нащадків. У цьому випадку фабричний метод виступає своєрідною точкою розширення, що дозволяє визначати власний спосіб створення об’єктів. Основна ідея полягає в тому, щоб замінити створення об’єктів базового типу на створення їх підтипів, зберігаючи при цьому загальну функціональність. Решта поведінки, не пов’язаної з інтерфейсом (AnOperation), дозволяє працювати з ними як із базовими об’єктами. Через це шаблон також називають «віртуальним конструктором».

Розглянемо приклад: застосунок працює з мережевими драйверами й використовує клас Packet для зберігання даних, що передаються мережею. Залежно від протоколу існують два варіанти — TcpPacket і UdpPacket, а також відповідні класи створення — TcpCreator і UdpCreator, які реалізують фабричний метод для створення цих об’єктів. При цьому спільна функціональність (передача, прийом і заповнення пакету даними) розміщується у базовому класі PacketCreator. Таким чином, поведінка системи залишається незмінною, але з’являється можливість гнучко підмінювати типи об’єктів під час створення та роботи з ними

3.Шаблон «Memento»

Призначення: Шаблон «Мементо» використовується для збереження та відновлення стану об’єктів без порушення принципу інкапсуляції [6]. Об’єкт типу «Memento» слугує сховищем для фіксації змін, зроблених над початковим об’єктом (Originator). Лише сам Originator має доступ до збереження та відновлення власного стану через об’єкт «Memento», який для інших учасників системи залишається «закритим». Об’єкт «Caretaker» відповідає за зберігання й передачу мементо між компонентами системи.

Цей підхід забезпечує кілька важливих переваг:

• збереження стану відокремлюється від логіки початкового об’єкта, що спрощує його реалізацію;

• управління передачею об’єктів «Memento» покладається на Caretaker, що робить роботу зі станами більш гнучкою і зменшує складність класів початкових об’єктів;

• операції збереження та відновлення стану реалізовані у вигляді двох простих методів, доступних лише самому Originator, що дозволяє зберегти інкапсуляцію.

Шаблон «Memento» особливо зручний у поєднанні з шаблоном «Команда», коли потрібно реалізувати можливість «скасування» дій — у цьому випадку дані про виконану дію зберігаються у мементо, а команда може відновити попередній стан об’єкта.

4.2.2. Шаблон «Observer»

Призначення: Шаблон «Спостерігач» визначає залежність типу «один-до-багатьох», коли зміна стану одного об’єкта автоматично повідомляє всі пов’язані з ним об’єкти, дозволяючи їм оновити власний стан [6].

Розглянемо приклад: у банківській системі кілька користувачів переглядають баланс рахунку пана І. Якщо він поповнює рахунок, баланс змінюється, і всі користувачі, які спостерігали за рахунком, отримують повідомлення про оновлення. Для них це може проявлятися як автоматичне оновлення суми або як сповіщення про зміну балансу. У результаті можуть стати доступними нові можливості, наприклад, перехід до іншої категорії клієнтів.

Шаблон «Observer» широко застосовується у патерні MVVM та в механізмах «прив’язок» (bindings) у WPF і частково у WinForms. Його ще називають шаблоном «підписка/розсилка». Спостерігачі самостійно підписуються на оновлення конкретних об’єктів, а ті, у свою чергу, зобов’язані сповіщати всіх своїх підписників про зміни. У мовах .NET на даний момент відсутні вбудовані механізми автоматичного сповіщення про зміну стану, тому реалізація цього процесу покладається на розробника.

4.2.3. Шаблон «Decorator»

Призначення: Шаблон «Декоратор» використовується для динамічного розширення функціональності об’єкта під час виконання програми [6]. Декоратор «обгортає» початковий об’єкт за допомогою агрегації, зберігаючи його основну поведінку, але додаючи нові можливості. Такий підхід є більш гнучким, ніж спадкування, оскільки дозволяє змінювати або розширювати поведінку лише конкретних об’єктів, не впливаючи на інші. Початкова функціональність при цьому повністю зберігається.

Прикладом може бути додавання смуги прокрутки до будь-якого візуального елемента. Об’єкт, який підтримує прокручування, «обгортається» у спеціальний декоратор, що додає можливість прокрутки. Якщо це необхідно, з’являється смуга прокрутки, але при цьому початкові функції елемента (наприклад, відображення статусного рядка) залишаються незмінними.

Хід роботи

У ході виконання лабораторної роботи було реалізовано патерн проєктування **Decorator**, який використовується для розширення функціональності системи обробки подань користувачів без зміни базового коду. Метою використання цього патерну є динамічне додавання додаткових дій під час виконання етапів обробки об’єктів SubmissionEntity, зберігаючи при цьому основну логіку роботи. Такий підхід забезпечує гнучкість, масштабованість і розширюваність системи без втручання у вже реалізовані класи..

1.Ознайомлення з теорією

Патерн Decorator дозволяє динамічно розширювати поведінку об’єктів, обгортаючи їх у додаткові класи-декоратори, які реалізують той самий інтерфейс, що й базовий об’єкт. Це дає змогу додавати нову логіку, не змінюючи вихідний код початкових класів. У нашій системі цей патерн застосовується для поетапної обробки подань (SubmissionEntity) через механізм StageExecutor, де кожен декоратор додає певну функціональність:

PersistResultDecorator — зберігає поточний стан подання у базі даних перед переходом до наступного етапу;

LogResultDecorator — логує інформацію про подання та його статус після виконання обробки.

Кожен декоратор розширює базовий клас StageResultDecorator, який, у свою чергу, делегує виклик обгорнутому об’єкту. Це дозволяє будувати ланцюг обробників, де кожен може виконувати власні додаткові дії до або після виклику основного методу execute. Такий підхід забезпечує можливість легко додавати нові поведінкові елементи (наприклад, логування, перевірку, оновлення статусу тощо) без зміни основного коду обробки.

2.Реалізація класів

Для реалізації патерну Decorator у системі створено такі класи:

StageExecutor — базовий інтерфейс, який містить метод execute(SubmissionEntity submission, StageExecutorChain chain) для обробки подань.

StageResultDecorator — абстрактний клас, який реалізує інтерфейс StageExecutor та зберігає посилання на обгорнутий об’єкт, передаючи йому управління після виконання власної логіки.

PersistResultDecorator — декоратор, який відповідає за збереження поточного стану подання через SubmissionService перед продовженням виконання наступного етапу.

LogResultDecorator — декоратор, який фіксує у логах інформацію про ID подання та його поточний статус. Таке розділення дозволяє додавати нові декоратори (наприклад, для валідації або надсилання сповіщень) без зміни вже існуючої структури, що робить систему розширюваною та простою в супроводі.

Послідовність виконання

Послідовність роботи системи з використанням патерну Decorator виглядає так:

Подання надходить у систему та зберігається з початковим статусом SUBMITTED. Під час виконання кожного етапу обробки викликається ланцюг декораторів — наприклад, спочатку PersistResultDecorator зберігає подання, потім LogResultDecorator записує інформацію до логів, після чого управління передається базовому StageExecutor для виконання основної бізнес-логіки.

Використані класи та взаємодія

У реалізації лабораторної роботи, що поєднує патерни Strategy та Decorator, було задіяно кілька основних класів і компонентів, які взаємодіють для забезпечення гнучкої, розширюваної та послідовної обробки подань користувачів.

1. SubmissionEntity

Клас відповідає за організацію та послідовне виконання етапів обробки подання (SubmissionEntity) у системі. Він формує ланцюг виконавців (StageExecutor) і обгортає їх у необхідні декоратори для додавання додаткової логіки, такої як збереження стану або логування.

2. StageExecutor

Базовий інтерфейс, який визначає метод execute(SubmissionEntity submission) для виконання певної стадії обробки. Забезпечує єдиний контракт для всіх виконавців та декораторів, дозволяючи гнучко підключати нову поведінку.

3.StageResultDecorator

Абстрактний клас-декоратор, який реалізує інтерфейс StageExecutor і містить посилання на об’єкт StageExecutor, що обгортається. Він делегує виклик методу execute() обгорнутому об’єкту, дозволяючи підкласам додавати власну логіку до або після основного виконання. Цей клас слугує базою для всіх конкретних декораторі..

4. PersistResultDecorator

Конкретний декоратор, який відповідає за збереження стану подання в базі даних. Використовує сервіс SubmissionService для збереження SubmissionEntity перед передачею керування наступному виконавцю у ланцюгу. Це дозволяє гарантувати, що результати обробки на даному етапі не будуть втрачені.

5.**LogResultDecorator**

Декоратор, що додає логування. Під час виконання методу execute() записує у лог інформацію про ID подання та його поточний статус, після чого передає керування обгорнутому виконавцю. Це забезпечує прозорість обробки та можливість відслідковування стану подань на різних етапах.

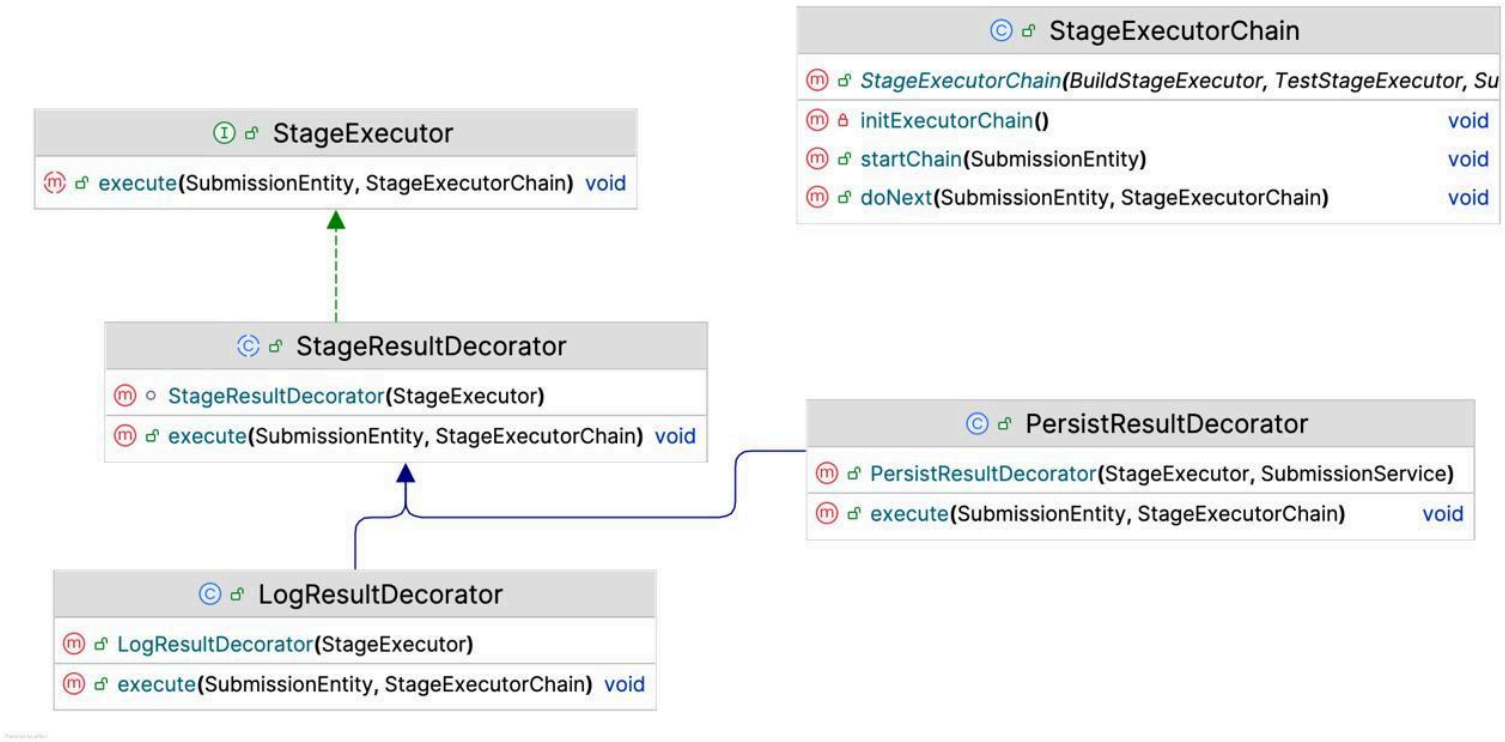

рис 1.1 Діаграмма класів системи

5. Код системи який реалізує паттерн Decorator

LogResultDecorator

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

import com.example.demo.model.submission.SubmissionEntity;
import com.example.demo.service.executor.stage.StageExecutor;
import com.example.demo.service.executor.stage.StageExecutorChain;
import lombok.extern.slf4j.Slf4j;

@Slf4j
public class LogResultDecorator extends StageResultDecorator {

    public LogResultDecorator(StageExecutor wrapped) {
        super(wrapped);
    }

    @Override
    public void execute(SubmissionEntity submission, StageExecutorChain chain) {
        log.info("submission with id: {}, saved with status: {}", submission.getId(),
submission.getStatus());
        super.execute(submission, chain);
    }

}
```

PersistResultDecorator

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


import com.example.demo.model.submission.SubmissionEntity;
import com.example.demo.service.executor.stage.StageExecutor;
import com.example.demo.service.executor.stage.StageExecutorChain;
import com.example.demo.service.submission.SubmissionService;

public class PersistResultDecorator extends StageResultDecorator {

   private final SubmissionService submissionService;

   public PersistResultDecorator(StageExecutor wrapped, SubmissionService
submissionService) {
        super(wrapped);
        this.submissionService = submissionService;
   }

   @Override
   public void execute(SubmissionEntity submission, StageExecutorChain chain) {
        submissionService.save(submission);
        super.execute(submission, chain);
   }
}
```

StageResultDecorator

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


import com.example.demo.model.submission.SubmissionEntity;
import com.example.demo.service.executor.stage.StageExecutor;
import com.example.demo.service.executor.stage.StageExecutorChain;
import com.example.demo.service.submission.SubmissionService;

public class PersistResultDecorator extends StageResultDecorator {

   private final SubmissionService submissionService;

   public PersistResultDecorator(StageExecutor wrapped, SubmissionService
submissionService) {
        super(wrapped);
        this.submissionService = submissionService;
   }
```

```
    @Override
    public void execute(SubmissionEntity submission, StageExecutorChain chain) {
        submissionService.save(submission);
        super.execute(submission, chain);
    }
}
```

StageExecutorChain

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

import com.example.demo.model.submission.SubmissionEntity;
import com.example.demo.service.executor.decorators.LogResultDecorator;
import com.example.demo.service.executor.decorators.PersistResultDecorator;
import com.example.demo.service.submission.SubmissionService;
import jakarta.annotation.PostConstruct;
import lombok.RequiredArgsConstructor;
import lombok.extern.slf4j.Slf4j;
import org.springframework.stereotype.Component;

import java.util.List;

@Slf4j
@Component("chain")
@RequiredArgsConstructor
public class StageExecutorChain {

    private final BuildStageExecutor buildStageExecutor;
    private final TestStageExecutor testStageExecutor;
    private final SubmissionService submissionService;

    private List<StageExecutor> stages;
    private int currentStageIndex = 0;

    @PostConstruct
    private void initExecutorChain() {
        stages = List.of(
                new LogResultDecorator(new PersistResultDecorator(buildStageExecutor,
submissionService)),
                new PersistResultDecorator(testStageExecutor, submissionService)
        );
    }

    public void doNext(SubmissionEntity submission, StageExecutorChain chain) {
        if (currentStageIndex < stages.size()) {
            StageExecutor currentStage = stages.get(currentStageIndex++);
            currentStage.execute(submission, chain);
        } else {
            log.info("All stages completed.");
            currentStageIndex = 0;
        }
    }
```

```
    public void startChain(SubmissionEntity submission) {
        this.currentStageIndex = 0;
        this.doNext(submission, this);
    }

}
```

6. Питання до лабораторної роботи

1. Призначення шаблону «Абстрактна фабрика»

Шаблон «Абстрактна фабрика» надає інтерфейс для створення сімейств взаємопов'язаних об'єктів без прив'язки до конкретних класів реалізації.

2. Структура шаблону «Абстрактна фабрика»

[AbstractFactory] → [ConcreteFactoryA], [ConcreteFactoryB]
[AbstractProductA], [AbstractProductB] ← [ConcreteProductA1], [ConcreteProductA2], [ConcreteProductB1], [ConcreteProductB2]

3. Класи та взаємодія у шаблоні «Абстрактна фабрика»

* AbstractFactory – інтерфейс фабрики
* ConcreteFactory – конкретна фабрика, створює конкретні продукти
* AbstractProduct – інтерфейс продукту
* ConcreteProduct – конкретні продукти

Взаємодія: клієнт використовує фабрику для створення продуктів → фабрика повертає конкретні об'єкти продуктів

4. Призначення шаблону «Фабричний метод»

Шаблон «Фабричний метод» визначає інтерфейс для створення об'єкта, але дозволяє підкласам вирішувати, який клас інстанціювати.

5. Структура шаблону «Фабричний метод»

[Creator] → factoryMethod()
[ConcreteCreator] → factoryMethod()
[Product]
[ConcreteProduct]

6. Класи та взаємодія у шаблоні «Фабричний метод»

* `Creator` – базовий клас із фабричним методом
* `ConcreteCreator` – підклас, який реалізує фабричний метод
* `Product` – інтерфейс продукту
* `ConcreteProduct` – конкретний продукт

Взаємодія: клієнт викликає фабричний метод → підклас створює конкретний продукт

7. Відмінність «Абстрактна фабрика» від «Фабричного методу»

Абстрактна фабрика створює сімейства взаємопов'язаних об'єктів, тоді як фабричний метод створює один конкретний об'єкт.

8. Призначення шаблону «Знімок»

Шаблон «Знімок» дозволяє зберегти внутрішній стан об'єкта і відновити його пізніше без порушення інкапсуляції.

9. Структура шаблону «Знімок»

[Originator] → createMemento(), restore(Memento)
[Memento]
[CareTaker] → keepMemento()

10. Класи та взаємодія у шаблоні «Знімок»

* `Originator` – об’єкт, стан якого зберігається
* `Memento` – об’єкт-знімок стану
* `CareTaker` – зберігає знімки, не змінює їх

Взаємодія: Originator створює Memento → CareTaker зберігає Memento → Originator може відновити стан з Memento

11. Призначення шаблону «Декоратор»

Шаблон «Декоратор» динамічно додає об’єктам нову функціональність, обгортаючи їх без зміни їх класу.

12. Структура шаблону «Декоратор»

[Component] ← [ConcreteComponent]
[Decorator] ← [ConcreteDecorator]

13. Класи та взаємодія у шаблоні «Декоратор»

* `Component` – базовий інтерфейс або клас
* `ConcreteComponent` – конкретний об’єкт
* `Decorator` – абстрактний декоратор, містить посилання на Component
* `ConcreteDecorator` – конкретний декоратор, додає поведінку

Взаємодія: клієнт взаємодіє з Component → Decorator обгортає компонент і додає функціональність

14. Обмеження використання шаблону «Декоратор»

* Може ускладнювати структуру через багато шарів обгортання
* Важко відстежувати поведінку об’єкта при багатьох декораторах
* Не завжди підходить для об’єктів із великою кількістю взаємозалежностей

Висновок

У ході виконання лабораторної роботи було досягнуто поставленої мети — вивчено структуру та принципи роботи шаблонів проєктування «Singleton», «Iterator», «Proxy», «State» та «Strategy», а також реалізовано шаблон проєктування «Strategy» у програмній системі. Було ознайомлено з теоретичними відомостями про кожен патерн, їх призначення та взаємодію класів.

Практична частина роботи включала реалізацію обраного шаблону за допомогою не менше ніж трьох класів, що дозволило наочно продемонструвати взаємодію об’єктів та делегування відповідальності між класами. Виконані фрагменти коду та діаграма класів показують правильне використання патернів у реалізації функціоналу системи.

Завдяки виконаній роботі було підтверджено, що застосування шаблонів проєктування підвищує гнучкість, повторне використання коду, зрозумілість архітектури та полегшує подальшу підтримку і розширення системи. Робота дозволила закріпити практичні навички побудови структурованого і масштабованого програмного забезпечення з урахуванням кращих практик об’єктно-орієнтованого проєктування.

Backend:

https://github.com/makszaranik/trpz-6-backend

Frontend:

https://github.com/makszaranik/trpz-frontend

Gateway:

https://github.com/makszaranik/trpz-gateway