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

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 «Singleton», «Iterator», «Proxy», «State»,

«Strategy» та навчитися застосовувати їх в реалізації програмної системи.

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 ситуаціях.

4.2.2. Шаблон «Singleton»

Призначення патерну: «Singleton» (Одинак) — це клас у термінах ООП, який може мати не більше одного екземпляра (звідси й назва «одинак») [6]. Насправді кількість екземплярів можна обмежити, тобто не дозволяти створення більше ніж **n** об’єктів даного класу. Зазвичай цей об’єкт зберігається як статичне поле всередині самого класу.

Проблема: Використання патерну «Одинак» доцільне у випадках:

- коли не може існувати більше **N** фізичних об’єктів, що відповідають певним класам;
- коли потрібно централізовано контролювати всі операції, що проходять через цей клас.

Шаблон «Одинак» вирішує обидві ці задачі одночасно, хоча при цьому порушується принцип єдиної відповідальності класу.

4.2.3. Шаблон «Iterator»

Призначення: «Iterator» (Ітератор) — це патерн, який забезпечує доступ до елементів колекції (агрегату) без розкриття її внутрішньої структури. Ітератор відокремлює логіку перебору елементів від самої колекції, забезпечуючи розподіл обов’язків: колекція відповідає за зберігання даних, а ітератор — за їх поетапне проходження [6].

Алгоритм роботи ітератора може змінюватися: для проходу у зворотному порядку застосовується інший ітератор, а також можливе створення ітератора, що спочатку проходить по парних елементах (2, 4, 6-й і т.д.), а потім по непарних. Таким чином, патерн «Ітератор» дозволяє реалізовувати різні способи обходу колекції незалежно від її внутрішньої структури та способу представлення даних.

4.2.4. Шаблон «Proxy»

Призначення: «Proxy» (Проксі) — це об’єкти, що виконують роль замінників або заглушок для реальних об’єктів певного типу. Головна ідея цього патерну полягає в тому, що замість безпосередньої взаємодії з дорогим або складним об’єктом використовується проміжний об’єкт, який контролює доступ, додає додаткову логіку або оптимізує взаємодію [5]. Проксі може виконувати різні функції: від контролю доступу та кешування до відкладеного створення об’єктів і ведення журналу дій.

Цей патерн особливо корисний, коли робота з об’єктом вимагає значних ресурсів або часу. Наприклад, у ранніх версіях інтернет-браузерів проксі використовувалися для відображення

зображень: поки реальне зображення завантажується з сервера, користувачеві показується тимчасова «заглушка». Таким чином, покращується взаємодія користувача з системою, підвищується швидкість роботи та економляться ресурси.

Використання проксі також дозволяє централізовано керувати доступом до об’єктів і легко змінювати або розширювати функціональність без модифікації основного класу. Це робить архітектуру системи більш гнучкою та підтримуваною. У сучасних інформаційних системах патерн «Proxy» застосовується у віддаленому доступі до ресурсів, захисті даних, логуванні та оптимізації взаємодії з великими об’єктами.

4.2.5. Шаблон «State»

Призначення: Шаблон «State» (Стан) дозволяє змінювати поведінку об’єктів залежно від їх внутрішнього стану [6]. Наприклад, відсоток нарахованих коштів на картковий рахунок може залежати від типу картки: Visa Electron, Classic, Platinum тощо. Або обсяг послуг, які надає хостинг-компанія, змінюється відповідно до обраного тарифного плану (стану членства — бронзовий, срібний або золотий клієнт).

Реалізація патерну полягає в наступному: всі поля, властивості, методи та дії, що залежать від стану, виділяються в окремий інтерфейс **State**. Кожен конкретний стан реалізується окремим класом (**ConcreteStateA**, **ConcreteStateB**), який наслідує цей інтерфейс [6, 8].

Об’єкт, що має стан (**Context**), при зміні стану просто присвоює новий об’єкт у поле **state**, що призводить до повної зміни його поведінки. Це забезпечує легке додавання нових станів у майбутньому, ізоляцію елементів об’єкта, залежних від стану, у окремі класи, та прозору заміну стану, що є корисним у багатьох практичних випадках.

4.2.6. Шаблон «Strategy»

Призначення: Шаблон «Strategy» (Стратегія) дозволяє замінювати один алгоритм поведінки об’єкта іншим, який досягає тієї ж мети іншим способом. Наприклад, алгоритми сортування можуть мати різні реалізації, кожна з яких визначена в окремому класі; при цьому алгоритми можна взаємозамінювати в об’єкті, який їх використовує [6].

Цей патерн особливо корисний, коли необхідно реалізувати різні «політики» обробки даних. По суті, «Strategy» схожий на патерн «State», але використовується для інших цілей — він дозволяє відобразити різні можливі поведінки об’єкта незалежно від його стану, при цьому досягаючи однієї або схожих цілей.

Хід роботи

У ході виконання лабораторної роботи було реалізовано патерн проєктування **Strategy** для обробки подань користувачів у системі. Метою використання цього патерну є послідовна обробка об’єктів SubmissionEntity різними етапами (стадіями) без прямої прив’язки клієнтського коду до конкретної реалізації обробника. Такий підхід забезпечує гнучкість, масштабованість та легку зміну логіки обробки без модифікації клієнтського коду.

1. **Ознайомлення з теорією**

Патерн проєктування Strategy дозволяє визначати сімейство алгоритмів або обробників і робити їх взаємозамінними, не змінюючи клієнтський код. Кожна конкретна стратегія реалізує спільний інтерфейс, а клієнт працює з ним, не прив’язуючись до конкретної реалізації. У нашій системі обробки подань користувачів (SubmissionEntity) патерн Strategy застосований для реалізації різних стадій обробки:

BuildJobStrategy — компіляція або підготовка подання до тестування.

TestJobStrategy — запуск тестів та перевірка результатів виконання. Кожна стратегія реалізує інтерфейс JobStrategy і може виконуватися незалежно від інших. Це дозволяє: легко додавати нові стадії обробки, змінювати логіку існуючих стадій, забезпечити послідовну обробку подань без зміни клієнтського коду, що робить систему гнучкою та масштабованою.

2. Реалізація класів

Для реалізації патерну **Strategy** у системі були створені наступні класи: JobStrategy — інтерфейс для всіх стратегій обробки подань, що містить метод execute(SubmissionEntity submission), який визначає логіку конкретної обробки. BuildJobStrategy — конкретна стратегія, яка відповідає за компіляцію або підготовку подання до тестування. TestJobStrategy — конкретна стратегія, яка відповідає за запуск тестів та перевірку правильності виконання. JobStrategyResolver — сервіс, який отримує зі Spring-контейнера конкретну реалізацію стратегії за типом роботи (JobType) і передає її клієнтському коду. JobExecutorContext — контекст, що зберігає посилання на обрану стратегію та виконує її метод execute для переданого подання. JobExecutor — сервіс, який отримує повідомлення про нове подання з RabbitMQ, визначає необхідні стратегії для виконання (Build та Test) та послідовно виконує їх через JobExecutorContext. Такий підхід дозволяє легко додавати нові стадії обробки, змінювати існуючі стратегії та забезпечує гнучку та масштабовану обробку подань без зміни клієнтського коду.

3. Послідовність виконання

Послідовність виконання у системі, що реалізує патерн Strategy, виглядає так: Подання надходить у систему і зберігається в базі даних із статусом SUBMITTED. JobExecutor отримує повідомлення з черги RabbitMQ і знаходить подання за ID. Для обробки визначаються необхідні стратегії (BuildJobStrategy → TestJobStrategy). JobExecutorContext встановлює першу стратегію і виконує її метод execute(), після чого встановлюється наступна стратегія і виконується її execute(). Після завершення всіх етапів JobExecutorContext скидає поточну стратегію, а JobExecutor логуватиме завершення обробки подання, забезпечуючи послідовну та керовану обробку без зміни клієнтського коду.

4. Використані класи та взаємодія

У реалізації лабораторної роботи за патерном Strategy було задіяно кілька основних класів та компонентів, які взаємодіють для забезпечення послідовної обробки подань користувачів.

1. SubmissionEntity

Клас представляє модель подання користувача, містить унікальний ідентифікатор, дані завдання, результати виконання та статус (Status). Статус визначає, чи можна обробляти подання. Об’єкт передається між усіма стратегічними обробниками, а кожна стратегія (JobStrategy) змінює його стан або додає результати виконання.

2. JobStrategy

Інтерфейс визначає метод execute(SubmissionEntity submission) для реалізації конкретних стратегій обробки. Це дозволяє клієнтському коду працювати зі стратегіями без прив’язки до конкретних реалізацій, підвищуючи гнучкість та масштабованість системи.

3.BuildJobStrategy / TestJobStrategy

Конкретні стратегії, які реалізують обробку подання на різних етапах: компіляція або підготовка (Build) та запуск тестів з перевіркою результатів (Test). Кожна стратегія інкапсулює власну логіку виконання.

4.JobStrategyResolver

Сервіс, який отримує зі Spring-контейнера конкретну реалізацію стратегії за типом роботи (JobType) і передає її клієнтському коду. Забезпечує динамічний вибір стратегії під час виконання.

5.JobExecutorContext

Контекст, що зберігає посилання на обрану стратегію та виконує її метод execute() для переданого подання. Дозволяє легко змінювати або чергувати стратегії без модифікації клієнтського коду.

6.JobExecutor

Сервіс інтегрує стратегії з системою надходження подань через RabbitMQ. Отримавши submissionId, знаходить подання через SubmissionService, визначає необхідні стратегії (Build → Test) і послідовно виконує їх через JobExecutorContext. Відповідає за зовнішню інтеграцію та координує обробку, делегуючи логіку конкретним стратегіям.

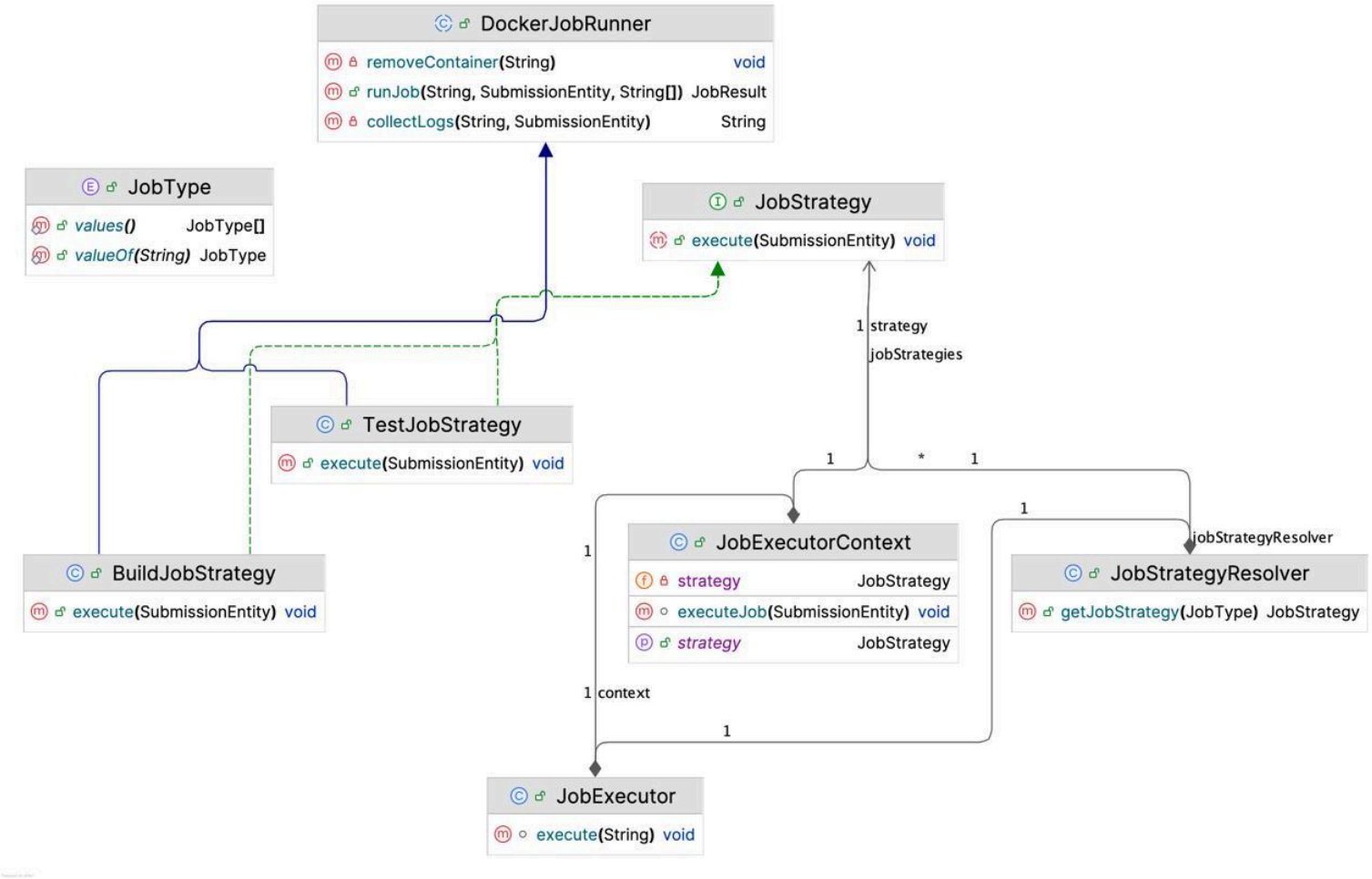

рис 1.1 Діаграмма класів системи

5. Код системи який реалізує паттерн Strategy

BuildJobStrategy

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

import com.example.demo.model.submission.SubmissionEntity;
import com.example.demo.service.executor.JobStrategyResolver;
import com.example.demo.service.submission.SubmissionService;
import com.github.dockerjava.api.DockerClient;
import lombok.extern.slf4j.Slf4j;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Slf4j
@Component("BUILD")
public class BuildJobStrategy extends DockerJobRunner implements JobStrategy {
```

```
    private final SubmissionService submissionService;

    @Autowired
    public BuildJobStrategy(DockerClient dockerClient, SubmissionService submissionService)
{
        super(dockerClient);
        this.submissionService = submissionService;
    }


    @Override
    public void execute(SubmissionEntity submission) {

        log.info("Build stage for submission {}.", submission.getId());
        String downloadPath = "http://host.docker.internal:8080/files/download/%s";
        String solutionUri = String.format(downloadPath, submission.getSourceCodeFileId());

        String cmd = String.format(
                "wget -O solution.zip %s && unzip solution.zip" +
                        " && cd solution" +
                        " && mvn clean compile -q",
                solutionUri
        );

        JobResult jobResult = runJob(
                "build_container",
                submission,
                "/bin/bash", "-c", cmd
        );

        Integer statusCode = jobResult.statusCode();
        String logs = jobResult.logs();

        log.info("Status code is {}", statusCode);
        if (statusCode == 0) {
            submission.setStatus(SubmissionEntity.Status.COMPILATION_SUCCESS);
        } else {
            submission.setStatus(SubmissionEntity.Status.COMPILATION_ERROR);
        }

        submission.setLogs(logs);
        submissionService.save(submission);
    }
}
```

TestJobStrategy

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

import com.example.demo.model.submission.SubmissionEntity;
import com.example.demo.model.task.TaskEntity;
import com.example.demo.service.submission.SubmissionService;
import com.example.demo.service.task.TaskService;
import com.github.dockerjava.api.DockerClient;
import lombok.extern.slf4j.Slf4j;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;


@Slf4j
@Component("TEST")
public class TestJobStrategy extends DockerJobRunner implements JobStrategy {

    private final TaskService taskService;
    private final SubmissionService submissionService;


    @Autowired
    public TestJobStrategy(DockerClient dockerClient,
                           SubmissionService submissionService,
                           TaskService taskService
    ) {
        super(dockerClient);
        this.taskService = taskService;
        this.submissionService = submissionService;
    }

    @Override
    public void execute(SubmissionEntity submission) {

        log.info("Test stage for submission {}.", submission.getId());
        TaskEntity task = taskService.findTaskById(submission.getTaskId());
        String testsFileId = task.getTestsFileId();

        String downloadPath = "http://host.docker.internal:8080/files/download/%s";
        String solutionUri = String.format(downloadPath, submission.getSourceCodeFileId());
        String testUri = String.format(downloadPath, testsFileId);

        String cmd = String.format(
                "wget -O solution.zip %s && unzip solution.zip" +
                        " && wget -O test.zip %s && unzip test.zip" +
                        " && mv test solution/src/test" +
                        " && cd solution" +
                        " && mvn clean test -q",
                solutionUri, testUri
        );

        JobResult jobResult = runJob(
                "test_job",
                submission,
                "/bin/bash", "-c", cmd
        );
```

```
        Integer statusCode = jobResult.statusCode();
        String logs = jobResult.logs();

        log.info("Status code is {}", statusCode);
        if (statusCode == 0) {
            submission.setStatus(SubmissionEntity.Status.ACCEPTED);
        } else {
            submission.setStatus(SubmissionEntity.Status.WRONG_ANSWER);
        }

        submission.setLogs(logs);
        submissionService.save(submission);
    }

}
```

JobStrategy

```
package com.example.demo.service.executor.stage;
import com.example.demo.model.submission.SubmissionEntity;

public interface JobStrategy {
    void execute(SubmissionEntity submission);
}
```

JobExecutor

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

import com.example.demo.config.RabbitConfig;
import com.example.demo.model.submission.SubmissionEntity;
import com.example.demo.service.executor.stage.BuildJobStrategy;
import com.example.demo.service.executor.stage.JobStrategy;
import com.example.demo.service.submission.SubmissionService;
import lombok.RequiredArgsConstructor;
import org.springframework.amqp.rabbit.annotation.RabbitListener;
import org.springframework.stereotype.Service;

import java.util.Map;

@Service
@RequiredArgsConstructor
public class JobExecutor {

    private final SubmissionService submissionService;
    private final JobExecutorContext context;
    private final JobStrategyResolver jobStrategyResolver;

    @RabbitListener(queues = RabbitConfig.SUBMISSION_QUEUE)
    void execute(String submissionId) {
        SubmissionEntity submission = submissionService.findSubmissionById(submissionId);
```

```
        //build
        context.setStrategy(jobStrategyResolver.getJobStrategy(JobType.BUILD));
        context.executeJob(submission);

        //test
        context.setStrategy(jobStrategyResolver.getJobStrategy(JobType.TEST));
        context.executeJob(submission);

    }
}
```

JobExecutorContext

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


import com.example.demo.model.submission.SubmissionEntity;
import com.example.demo.service.executor.stage.JobStrategy;
import lombok.Setter;
import org.springframework.stereotype.Service;

@Setter
@Service
public class JobExecutorContext {

    private JobStrategy strategy;

    void executeJob(SubmissionEntity submission) {
        strategy.execute(submission);
    }

}
```

JobStrategyResolver

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

import com.example.demo.service.executor.stage.JobStrategy;
import lombok.RequiredArgsConstructor;
import org.springframework.stereotype.Service;

import java.util.Map;

@Service
@RequiredArgsConstructor
public class JobStrategyResolver {

    private final Map<String, JobStrategy> jobStrategies; // test -> TestJobStrategy...

    public JobStrategy getJobStrategy(JobType jobType) {
        return jobStrategies.get(jobType.name());
```

```
    }

}
```

JobType

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

public enum JobType {
    BUILD,
    TEST
}
```

6. Питання до лабораторної роботи

1. **Що таке шаблон проєктування?**
   Шаблон проєктування — це перевірене рішення типових проблем програмного забезпечення, яке описує перевірену структуру класів або об’єктів та їхню взаємодію для досягнення певної функціональності. Його мета — стандартизувати підхід до розробки та зробити систему більш зрозумілою та підтримуваною.

2. **Навіщо використовувати шаблони проєктування?**
   Використання шаблонів проєктування дозволяє створювати більш гнучкі та масштабовані системи, зменшувати дублікацію коду, підвищувати його повторне використання та забезпечувати єдиний, зрозумілий підхід до вирішення повторюваних задач проектування.

3. **Призначення шаблону «Стратегія»**
   Шаблон «Стратегія» дозволяє динамічно змінювати алгоритм поведінки об’єкта, відокремлюючи конкретну реалізацію алгоритму від класу, який його використовує. Це забезпечує можливість підміняти алгоритми під час виконання без зміни коду контекстного класу.

4. **Структура шаблону «Стратегія»**
   Контекст (Context) взаємодіє з інтерфейсом Strategy, який реалізують конкретні алгоритми (ConcreteStrategy). Контекст делегує виконання алгоритму об’єкту Strategy, не знаючи конкретної реалізації.

5. **Класи та взаємодія («Стратегія»)**

- **Strategy** — інтерфейс, що описує метод виконання алгоритму.
- **ConcreteStrategy** — реалізація конкретного алгоритму.
- **Context** — клас, який містить посилання на Strategy і делегує йому виконання алгоритму, дозволяючи змінювати поведінку без зміни Context.

6. **Призначення шаблону «Стан»**
   Шаблон «Стан» дозволяє об’єкту змінювати свою поведінку залежно від внутрішнього стану, не змінюючи клас. Клас Context делегує виконання дій об’єкту стану, що забезпечує гнучкість і відокремлення станів від логіки класу.

7. **Структура шаблону «Стан»**
   Контекст (Context) взаємодіє з інтерфейсом State, який реалізують конкретні стани (ConcreteState). Кожен ConcreteState реалізує специфічну поведінку для об’єкта Context, дозволяючи змінювати поведінку під час виконання.

8. **Класи та взаємодія («Стан»)**

- **State** — інтерфейс стану, що визначає набір дій.
- **ConcreteState** — реалізація конкретного стану з відповідною поведінкою.
- **Context** — містить посилання на State і делегує йому виконання дій. Під час зміни стану Context використовує інший ConcreteState, змінюючи поведінку динамічно.

9. **Призначення шаблону «Ітератор»**
   Шаблон «Ітератор» дозволяє послідовно обходити елементи колекції без розкриття її внутрішньої структури. Це забезпечує єдиний інтерфейс перебору для різних типів колекцій і підвищує абстракцію доступу до даних.

10. **Структура шаблону «Ітератор»**
    Колекція (Aggregate) надає метод для створення Iterator. Iterator визначає інтерфейс перебору, який реалізує ConcreteIterator для конкретної колекції (ConcreteAggregate). Контролюючи перебір через Iterator, клієнт не залежить від внутрішньої структури колекції.

\

11. **Класи та взаємодія («Ітератор»)**

- **Iterator** — інтерфейс, що описує методи для перебору елементів.
- **ConcreteIterator** — реалізація конкретного перебору.
- **Aggregate** — інтерфейс колекції.
- **ConcreteAggregate** — реалізація колекції, яка створює ConcreteIterator для обходу елементів. Клієнт отримує доступ до елементів через Iterator, не знаючи структури колекції.

12. **Ідея шаблону «Одинак» (Singleton)**
    Шаблон «Одинак» забезпечує наявність лише одного екземпляра класу в системі та надає глобальну точку доступу до нього. Це гарантує, що ресурси, стан або налаштування будуть спільними для всіх частин програми.

13. **Чому «Одинак» вважають «анти-шаблоном»**
    Singleton вважають «анти-шаблоном», оскільки він створює глобальний стан, ускладнює тестування, порушує принципи інверсії залежностей і може призводити до жорсткої зв’язності компонентів.

14. **Призначення шаблону «Проксі»**

Шаблон «Проксі» надає об’єкт-замінник для контролю доступу до реального об’єкта, додаткової логіки або віддалених викликів. Це дозволяє захистити або оптимізувати роботу з ресурсом.

15. **Структура шаблону «Проксі»**

Клієнт (Client) звертається до Proxy, який реалізує той же інтерфейс, що RealSubject. Proxy контролює доступ до RealSubject, делегуючи виклики та додаючи власну логіку при необхідності.

16. **Класи та взаємодія («Проксі»)**

- **Subject** — спільний інтерфейс для Proxy та RealSubject.
- **RealSubject** — реальний об’єкт із необхідною функціональністю.
- **Proxy** — контролює доступ до RealSubject, делегує виклики через Subject, додає логування, кешування або контроль доступу за потреби.

Висновки

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-4-backend

Frontend:

https://github.com/makszaranik/trpz-4-frontend

Gateway:

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