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 №5

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 «Adapter», «Builder», «Command», «Chain of responsibility», «Prototype» 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.

5.2.2 Шаблон «Singleton»

Призначення патерну: Шаблон **«Builder» (Будівельник)** застосовується для **розділення процесу створення об'єкта та його подання**. Це зручно у ситуаціях, коли створення об'єкта є складним (наприклад, веб-сторінка як частина відповіді веб-сервера) або коли об'єкт може мати **декілька варіантів побудови** (наприклад, під час конвертації тексту з одного формату в інший).

5.2.3. Шаблон «Command»

Призначення патерну: Шаблон **«Command» (Команда)** перетворює звичайний виклик методу на окремий клас. Завдяки цьому дії в системі стають **повноцінними об'єктами**. Це корисно у таких випадках:

- Коли потрібна розширювана система команд – нові команди можуть легко додаватися;
- Коли потрібна гнучка система команд – можна додавати функції відміни, логування тощо;
- Коли потрібна можливість формувати ланцюжки команд або викликати їх у певний час.

Об'єкт-команда сам по собі не виконує фактичні дії, а лише перенаправляє запит **одержувачу**, проте може зберігати дані для підтримки відміни, логування та інших функцій. Наприклад, команда вставки символу зберігає символ і при відміні викликає видалення символу. Можна також визначити параметр «застосовності» команди (наприклад, на картинці писати не можна) і використовувати його для активації чи деактивації елементів меню. Такий підхід дозволяє будувати **гнучкі та настроювані системи команд**, що особливо важливо для додатків з великою кількістю команд, наприклад редакторів.

5.2.4. Шаблон «Chain of Responsibility»

Призначення патерну: Шаблон «Chain of Responsibility» (Ланцюжок відповідальності) передбачає передавання запиту по ланцюгу об'єктів, доки його не обробить відповідний отримувач. Аналогічно, як у житті, коли документ підписується спочатку співробітником, потім менеджером, начальником і нарешті головним керівником, який ставить остаточний підпис.

5.2.5. Шаблон «Prototype»

Призначення патерну: Шаблон **«Prototype» (Прототип)** застосовується для створення об’єктів за **«шаблоном»** шляхом копіювання існуючого прототипного об’єкта. Для цього в класі визначається метод **«клонувати»**.

Цей патерн зручний, коли заздалегідь відомий вигляд кінцевого об’єкта, що дозволяє мінімізувати зміни після створення, а також **усуває потребу знати, як створювати об’єкт** — він просто клонуються. Крім того, патерн дозволяє маніпулювати об’єктами під час виконання програми через налаштування прототипів і значно зменшує ієрархію наслідування, уникаючи створення багатьох вкладених класів.

Хід роботи

У ході виконання лабораторної роботи було реалізовано патерн проектування **Chain of Responsibility** для обробки подань користувачів у системі. Метою використання даного патерну є послідовна обробка об’єктів (SubmissionEntity) різними етапами (стадіями) без прив’язки виклику конкретного обробника на рівні клієнта.

1. **Ознайомлення з теорією**

Chain of Responsibility дозволяє передавати запит вздовж ланцюжка об’єктів, поки він не буде оброблений. Кожен об’єкт-обробник визначає, чи може він обробити запит, чи передати його наступному об’єкту у ланцюжку. У нашій системі було визначено два основні етапи обробки подань: BuildStageExecutor — компіляція або підготовка подання до тестування. TestStageExecutor — запуск тестів та перевірка правильності виконання. Ці етапи реалізовані як окремі класи, що реалізують інтерфейс StageExecutor

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

import com.example.demo.model.submission.SubmissionEntity;

public interface StageExecutor {

  void execute(SubmissionEntity submission, StageExecutorChain chain);

}
```

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

import com.example.demo.model.submission.SubmissionEntity;
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.data.mongodb.core.aggregation.ArrayOperators;
import org.springframework.stereotype.Component;

import java.util.ArrayList;
import java.util.List;

@Slf4j
@Component("build")
public class BuildStageExecutor extends DockerJobRunner implements StageExecutor {

  private final SubmissionService submissionService;

  @Autowired
  public BuildStageExecutor(DockerClient dockerClient, SubmissionService submissionService) {
```

```
        super(dockerClient);
        this.submissionService = submissionService;
    }


    @Override
    public void execute(SubmissionEntity submission, StageExecutorChain chain) {

        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);
            submission.setLogs(logs);
            submissionService.save(submission);
            chain.doNext(submission, chain);
        } else {
            submission.setStatus(SubmissionEntity.Status.COMPILATION_ERROR);
            submission.setLogs(logs);
            submissionService.save(submission);
        }
    }
}


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.RequiredArgsConstructor;
import lombok.extern.slf4j.Slf4j;
```

```java
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;


@Slf4j
@Component("test")
public class TestStageExecutor extends DockerJobRunner implements StageExecutor {

  private final TaskService taskService;
  private final SubmissionService submissionService;


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

  @Override
  public void execute(SubmissionEntity submission, StageExecutorChain chain) {

    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);
            submission.setLogs(logs);
            submissionService.save(submission);
            chain.doNext(submission, chain);
        } else {
            submission.setStatus(SubmissionEntity.Status.WRONG_ANSWER);
            submission.setLogs(logs);
            submissionService.save(submission);
        }
    }

}
```

2. Реалізація класів

Для реалізації патерну були створені наступні класи: StageExecutor — інтерфейс для всіх обробників, що містить метод execute(SubmissionEntity submission, StageExecutorChain chain);

StageExecutorChain — клас, який управляє ланцюжком обробки і викликає наступний етап через метод doNext(). Він ініціалізує список обробників у методі @PostConstruct initExecutorChain(), що дозволяє Spring автоматично зв'язати залежності;
PipelineExecutor — сервіс, який отримує повідомлення з RabbitMQ про нове подання і запускає ланцюжок через StageExecutorChain

3. Послідовність виконання

Подання надходить у систему і зберігається в базі даних із статусом SUBMITTED. PipelineExecutor отримує повідомлення з черги RabbitMQ і знаходить подання за ID. Якщо подання має статус SUBMITTED, запускається метод startChain() у StageExecutorChain. StageExecutorChain послідовно викликає методи execute() обробників (BuildStageExecutor → TestStageExecutor). Після завершення всіх етапів ланцюжок скидає індекс поточного етапу і логуватиме завершення обробки (рис. 1.1)

```
{
    _id: ObjectId('68e8d7700ab0c9c39cf67061'),
    taskId: '68e375c3b23773694e7c2385',
    userId: 'userId',
    sourceCodeFileId: '68e3782bb23773694e7c238a',
    logs: '-\n-2025-10-10 09:52:54--  http://app:8080/files
    status: 'WRONG_ANSWER',
    createdAt: ISODate('2025-10-10T09:52:48.828Z'),
    _class: 'com.example.demo.model.submission.SubmissionEn
}
```

рис 1.1 - статус submission після надсилання та тестування

4. Використані класи та взаємодія

У реалізації лабораторної роботи за патерном Chain of Responsibility було задіяно кілька основних класів та компонентів, які взаємодіють між собою для забезпечення послідовної обробки подань користувачів.

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 стадіями ланцюжка, а кожен обробник (StageExecutor) змінює його стан або додає результати.

StageExecutor

Інтерфейс визначає метод execute(SubmissionEntity submission, StageExecutorChain chain) для реалізації обробників. Це дозволяє передавати подання наступному етапу через chain.doNext() без знання деталей наступних стадій, що підвищує гнучкість і масштабованість системи.

StageExecutorChain

Клас керує черговістю обробників та викликом наступного етапу через doNext(). Список обробників ініціалізується у @PostConstruct, а після завершення ланцюжка індекс поточного етапу скидається. Chain виступає посередником між обробниками, забезпечуючи централізоване управління потоком обробки.

PipelineExecutor

Сервіс інтегрує ланцюжок з системою надходження подань через RabbitMQ. Отримавши submissionId, він знаходить подання через SubmissionService і, якщо статус SUBMITTED, запускає ланцюжок через StageExecutorChain.startChain(). Відповідає за зовнішню інтеграцію, делегуючи логіку обробки ланцюжку.

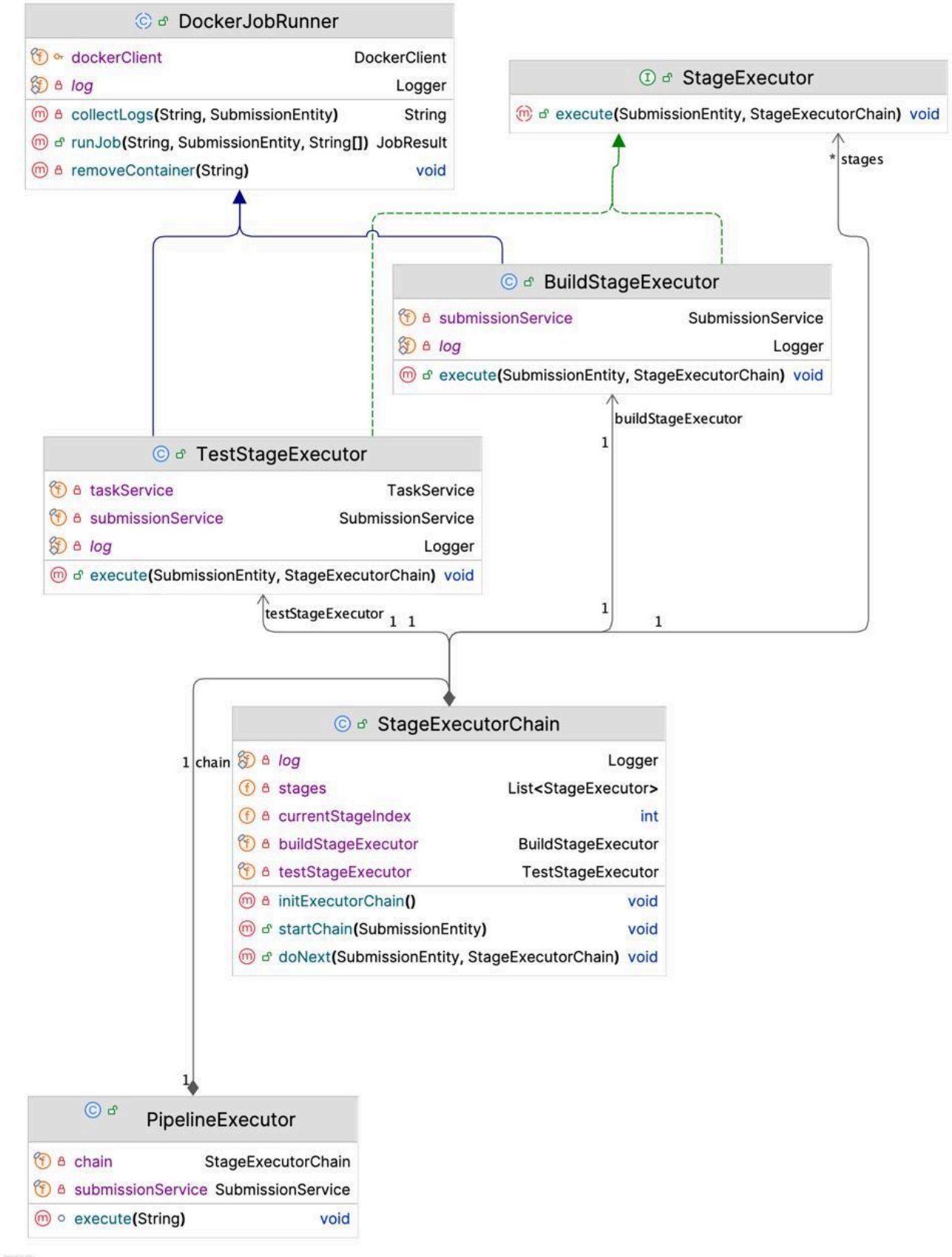

рис 1.2 діаграма класів

На рисунку 1.2 представлено діаграму класів, на ній знаходяться такі елементи StageExecutor (інтерфейс) Визначає метод execute(SubmissionEntity, StageExecutorChain) для виконання стадії пайплайну та передачі контролю наступній стадії. Конкретні екзекутори:

BuildStageExecutor — відповідає за збірку, залежить від SubmissionService та Logger. TestStageExecutor — відповідає за тестування, залежить від TaskService, SubmissionService, Logger і використовує DockerJobRunner для запуску контейнерів. StageExecutorChain Керує послідовністю виконання стадій через список stages і методи startChain() та doNext().

Має посилання на конкретні екзекутори та індекс поточної стадії. PipelineExecutor Точка входу пайплайну, отримує подання через SubmissionService і запускає ланцюжок StageExecutorChain. Взаємозв'язки BuildStageExecutor і TestStageExecutor реалізують StageExecutor. StageExecutorChain агрегує екзекутори; TestStageExecutor залежить від DockerJobRunner. PipelineExecutor асоційований з ланцюжком та SubmissionService.

5. Код системи який реалізує паттерн Chain of responsebility

PipelineExecutor

```
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.StageExecutorChain;
import com.example.demo.service.submission.SubmissionService;
import lombok.RequiredArgsConstructor;
import org.springframework.amqp.rabbit.annotation.RabbitListener;
import org.springframework.scheduling.annotation.Scheduled;
import org.springframework.stereotype.Service;

@Service
@RequiredArgsConstructor
public class PipelineExecutor {

  private final SubmissionService submissionService;
  private final StageExecutorChain chain;

  @RabbitListener(queues = RabbitConfig.SUBMISSION_QUEUE)
  void execute(String submissionId) {
     SubmissionEntity submission = submissionService.findSubmissionById(submissionId);
     if(submission.getStatus() == SubmissionEntity.Status.SUBMITTED){
        chain.startChain(submission);
     }
  }
}
```

StageExecutor

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

import com.example.demo.model.submission.SubmissionEntity;

public interface StageExecutor {

  void execute(SubmissionEntity submission, StageExecutorChain chain);

}
```

DockerJobRunner

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

import com.example.demo.model.submission.SubmissionEntity;
import com.example.demo.service.submission.SubmissionService;
import com.github.dockerjava.api.DockerClient;
import com.github.dockerjava.api.async.ResultCallback;
import com.github.dockerjava.api.command.CreateContainerResponse;
import com.github.dockerjava.api.command.WaitContainerResultCallback;
import com.github.dockerjava.api.model.Frame;
import com.github.dockerjava.api.model.HostConfig;
import lombok.RequiredArgsConstructor;
import lombok.SneakyThrows;
import lombok.extern.slf4j.Slf4j;

import java.nio.charset.StandardCharsets;
import java.util.concurrent.TimeUnit;


@Slf4j
@RequiredArgsConstructor
public abstract class DockerJobRunner {

  protected final DockerClient dockerClient;

  public JobResult runJob(String containerName, SubmissionEntity submission, String... args) {
    Integer statusCode;
    String containerLogs = "";
    String containerId = "";

    try {
      CreateContainerResponse container = dockerClient.createContainerCmd("java-maven-ci")
          .withCmd(args)
          .withHostConfig(HostConfig.newHostConfig().withNetworkMode("demo_default"))
          .withAttachStdin(true)
          .withAttachStdout(true)
          .withTty(true)
          .withName(containerName)
```

```
                .exec();

        containerId = container.getId();
        dockerClient.startContainerCmd(containerId).exec();

        log.info("container with id {} running.", containerId);

        statusCode = dockerClient
                .waitContainerCmd(containerId)
                .exec(new WaitContainerResultCallback())
                .awaitStatusCode(60, TimeUnit.SECONDS); //await build for 60 sec

        log.info("container with id {} finished.", containerId);

    } catch (Exception e) {
        statusCode = -1;
        log.error(e.getMessage(), e);

    } finally {
        containerLogs = collectLogs(containerId, submission);
        removeContainer(containerId);
    }
    return new JobResult(statusCode, containerLogs);
}

@SneakyThrows
private String collectLogs(String containerId, SubmissionEntity submission) {
    StringBuilder logs = new StringBuilder();
    dockerClient.logContainerCmd(containerId)
            .withStdOut(true)
            .withStdErr(true)
            .withFollowStream(false)
            .exec(new ResultCallback.Adapter<Frame>() {
                @Override
                public void onNext(Frame frame) {
                    String line = new String(frame.getPayload(), StandardCharsets.UTF_8).trim();
                    logs.append(line).append(System.lineSeparator());
                }
            })
            .awaitCompletion(60, TimeUnit.SECONDS); //await logs for 60 sec

    log.info("container {} logs", logs);
    return logs.toString();
}

private void removeContainer(String containerId) {
    dockerClient.removeContainerCmd(containerId)
            .withRemoveVolumes(true)
            .withForce(true)
            .exec();
}
```

```
  public record JobResult(Integer statusCode, String logs){}

}
```

BuildStageExecutor

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

import com.example.demo.model.submission.SubmissionEntity;
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.data.mongodb.core.aggregation.ArrayOperators;
import org.springframework.stereotype.Component;

import java.util.ArrayList;
import java.util.List;

@Slf4j
@Component("build")
public class BuildStageExecutor extends DockerJobRunner implements StageExecutor {

  private final SubmissionService submissionService;

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


  @Override
  public void execute(SubmissionEntity submission, StageExecutorChain chain) {

    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);
            submission.setLogs(logs);
            submissionService.save(submission);
            chain.doNext(submission, chain);
        } else {
            submission.setStatus(SubmissionEntity.Status.COMPILATION_ERROR);
            submission.setLogs(logs);
            submissionService.save(submission);
        }
    }
}
```

TestStageExecutor

```
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.RequiredArgsConstructor;
import lombok.extern.slf4j.Slf4j;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;


@Slf4j
@Component("test")
public class TestStageExecutor extends DockerJobRunner implements StageExecutor {

    private final TaskService taskService;
    private final SubmissionService submissionService;


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

```
    @Override
    public void execute(SubmissionEntity submission, StageExecutorChain chain) {

        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);
            submission.setLogs(logs);
            submissionService.save(submission);
            chain.doNext(submission, chain);
        } else {
            submission.setStatus(SubmissionEntity.Status.WRONG_ANSWER);
            submission.setLogs(logs);
            submissionService.save(submission);
        }
    }

}
```

StageExecutorChain

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

```java
import com.example.demo.model.submission.SubmissionEntity;
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 List<StageExecutor> stages;
  private int currentStageIndex = 0;

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

  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);
  }

}
```

5.3 Питання до лабораторної роботи

1. Яке призначення шаблону «Адаптер»?

Дозволяє об’єднати несумісні інтерфейси, щоб клієнт міг працювати з класом через потрібний йому інтерфейс, не змінюючи сам клас.

2. Нарисуйте структуру шаблону «Адаптер».

Client → Adapter → Adaptee

3. Які класи входять в шаблон «Адаптер», та яка між ними взаємодія?

Client – використовує інтерфейс Target; Target – потрібний інтерфейс; Adaptee – клас з іншим інтерфейсом; Adapter – реалізує Target і викликає методи Adaptee.

4. Яка різниця між реалізацією «Адаптера» на рівні об’єктів та на рівні класів?

Об’єктний адаптер використовує композицію (Adapter містить Adaptee), класовий – спадкування (Adapter наслідує Adaptee і Target).

5. Яке призначення шаблону «Будівельник»?

Розділяє конструювання складного об’єкта від його представлення, щоб один процес створював різні представлення.

6. Нарисуйте структуру шаблону «Будівельник».

Director → Builder → ConcreteBuilder → Product

7. Які класи входять в шаблон «Будівельник», та яка між ними взаємодія?

Director – керує процесом; Builder – абстрактний інтерфейс для створення частин продукту; ConcreteBuilder – реалізація Builder, створює продукт; Product – кінцевий об’єкт.

8. У яких випадках варто застосовувати шаблон «Будівельник»?

Коли об’єкт складний, має багато компонентів, потрібні різні представлення одного об’єкта або потрібно ізолювати конструювання від використання.

9. Яке призначення шаблону «Команда»?

Інкапсулює запит як об’єкт, відокремлюючи того, хто виконує дію, від того, хто її ініціює.

10. Нарисуйте структуру шаблону «Команда».

Client → Command → ConcreteCommand → Receiver → Invoker

11. Які класи входять в шаблон «Команда», та яка між ними взаємодія?

Client – створює команду; Command – абстрактний інтерфейс; ConcreteCommand – реалізація команди, викликає Receiver; Receiver – виконує операцію; Invoker – ініціює виконання команди.

12. Розкажіть як працює шаблон «Команда».

Client створює ConcreteCommand з посиланням на Receiver, передає її Invoker, який викликає execute(), що активує дію в Receiver.

13. Яке призначення шаблону «Прототип»?

Створює нові об’єкти копіюванням існуючого (прототипу) замість створення з нуля.

14. Нарисуйте структуру шаблону «Прототип».

Client → Prototype → ConcretePrototype

15. Які класи входять в шаблон «Прототип», та яка між ними взаємодія?

Client – запитує клон; Prototype – інтерфейс з методом clone(); ConcretePrototype – реалізує клонування.

16. Які можна привести приклади використання шаблону «Ланцюжок відповідальності»?

Обробка запитів у GUI (кнопки, панелі), логування різних рівнів, перевірка прав доступу, валідація форм, обробка подій у системах без центрального контролю.

Висновки

У ході виконання лабораторної роботи було досягнуто поставленої мети — вивчено структуру та принципи роботи шаблонів проєктування «Adapter», «Builder», «Command», «Chain of responsibility», «Prototype», а також набуті практичні навички їх застосування 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-5-backend

Frontend:

https://github.com/makszaranik/trpz-5-frontend

Gateway:

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