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
Технології розроблення програмного забезпечення
ШАБЛОНИ «ADAPTER», «BUILDER»,
«COMMAND», «CHAIN OF RESPONSIBILITY», «PROTOTYPE»
CI-сервер

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

Перевірив:
Мягкий М. Ю

Київ 2024

Зміст

Тема: шаблони «ADAPTER», «BUILDER», «COMMAND», «CHAIN OF RESPONSIBILITY», «PROTOTYPE»
Мета: ознайомитись з теоретичними відомостями, розробити частину функціоналу системи з використанням одного із шаблонів проектування.

Теоретичні відомості

Шаблон Adapter використовується для узгодження інтерфейсів класів, які спочатку не були призначені для спільної роботи. Його головна мета полягає у створенні своєрідного мосту між класами без необхідності змінювати їхній код. Це дозволяє перетворювати інтерфейс одного класу в інший, очікуваний клієнтським кодом. Такий підхід особливо корисний, коли потрібно інтегрувати в систему сторонній код або бібліотеки з несумісними інтерфейсами.

Шаблон Builder допомагає спростити створення складних об'єктів, розділяючи цей процес на послідовні кроки. Його використання особливо виправдане, якщо об'єкт має багато конфігурацій. У цьому шаблоні відокремлюється логіка побудови об'єкта від його кінцевої структури, що сприяє гнучкості та розширюваності коду. Builder дозволяє легко створювати об'єкти з різними параметрами, зберігаючи при цьому їхню узгодженість.

Шаблон Command інкапсулює запити або операції як окремі об'єкти. Це дозволяє зберігати їх для подальшого виконання, передавати між об'єктами або навіть підтримувати операції скасування й повторення. Command використовується в системах, де необхідно організувати гнучке управління діями, наприклад, в системах з графічним інтерфейсом або у складних бізнес-логіках.

Шаблон Chain of Responsibility створює ланцюжок об'єктів, через який передається запит до тих пір, поки один із об'єктів не обробить його. Це дозволяє уникнути жорсткої прив'язки відправника запиту до конкретного одержувача. Кожен елемент ланцюжка відповідає за свою частину обробки, а якщо він не може виконати запит, то передає його далі. Такий підхід забезпечує гнучкість і полегшує розширення системи.

Шаблон Prototype дозволяє створювати нові об'єкти шляхом копіювання існуючих. Це особливо ефективно в ситуаціях, коли створення об'єкта з нуля є ресурсозатратним або складним. Прототипи використовуються для створення складних об'єктів, де важливо зберігати їхню початкову конфігурацію або стан.

Загалом, використання цих шаблонів у проєктуванні програмного забезпечення сприяє підвищенню якості архітектури, спрощує підтримку коду, робить систему більш гнучкою та зрозумілою для розробників. Їх вивчення та правильне застосування є невід’ємною частиною професійного зростання кожного програміста.

Хід роботи

В ході виконання лабораторної роботи було реалізовано клас Build із підтримкою патерну проектування Builder, який забезпечує зручний спосіб створення збірок.

У цій лабораторній роботі я використав патерн Builder, оскільки він дозволяє зручно і зрозуміло створювати складні об’єкти, такі як екземпляри класу Build, забезпечуючи при цьому гнучкість у виборі параметрів. Використання Builder дозволило уникнути великої кількості конструкторів з різною кількістю параметрів і зробило код більш зрозумілим.

Патерн Command має такі переваги:

- Інкапсуляція запитів або операцій у вигляді окремих об'єктів.
- Можливість передавати операції між різними частинами системи.
- Реалізація функціональності скасування або повторення операцій.
- Забезпечення гнучкості в управлінні діями в складних системах.

Однак я його не використав з наступних причин:

- Відсутність потреби у динамічному управлінні діями: У даній задачі функціональність системи не передбачає складних сценаріїв управління командами. Створення та конфігурація об'єктів відбувається послідовно та не потребує відкладеного виконання або скасування дій.

- Зайва складність для поточної реалізації: Використання Command вимагало б введення додаткових класів для представлення кожної операції (наприклад, створення збірки, додавання параметрів тощо). Це ускладнило б структуру проєкту без реальної потреби.

- Пріоритет статичної конфігурації: Параметри, що використовуються для створення об'єктів, визначаються заздалегідь і передаються через Builder. Тому немає необхідності інкапсулювати ці дії в окремі команди.

- Контекст завдання: Патерн Command доцільніше використовувати в системах, де потрібно зберігати історію дій, створювати чергу команд або інтегрувати з інтерфейсами, наприклад, графічними. У нашій роботі таких вимог немає.

Таким чином, патерн Command не відповідав вимогам даної лабораторної роботи, і його використання було б зайвим у контексті поставлених задач.

Шаблон Builder дозволяє гарно і лаконічно створювати екземпляри нашого класу. У класі BuildBuilder є методи, що відповідають за заповнення окремих полів об'єкта. В кінці створення викликається метод build(), який повертає екземпляр Build із заданими параметрами. (Рис. 1.1)

```
public static class BuildBuilder {  9 usages
    private Long id;  3 usages
    private Instant startTime;  3 usages
    private Instant endTime;  3 usages
    private String logFilePath;  3 usages
    private String branch;  3 usages
    private Project project;  3 usages
    private BuildStatus buildStatus;  3 usages

    BuildBuilder() {  1 usage
    }

    public BuildBuilder id(Long id) {...}

    public BuildBuilder startTime(Instant startTime) {...}

    public BuildBuilder endTime(Instant endTime) {...}

    public BuildBuilder logFilePath(String logFilePath) {...}

    public BuildBuilder branch(String branch) {...}

    public BuildBuilder project(Project project) {...}

    public BuildBuilder buildStatus(BuildStatus buildStatus) {...}

    public Build build() {
        return new Build(this.id, this.startTime, this.endTime, this.logFilePath, this.branch, this.project, this.buildStatus);
    }
```

Рисунок 1.1. клас BuildBuilder для створення екземплярів збірок

Використання шаблону Builder дозволяє зручно та швидко створювати об’єкти. При цьому нам не потрібна велика кількість різноманітних конструкторів з різною кількістю параметрів. Приклад використання (Рис 1.2)

```
@Transactional
public Build createBuild(Long projectId, String branch) {
    var project = findById(projectId).orElseThrow(IllegalArgumentException::new);
    var build = Build.builder()
            .buildStatus(BuildStatus.RUNNING)
            .branch(branch)
            .build();
    project.addBuild(build);
    buildRepository.persist(build);

    build.setLogFilePath(Path.of(BUILD_LOGS_DIR, LOG_FILENAME_PATTERN.formatted(build.getId())).toString());
    buildRepository.persist(build);

    return build;
}
```

Рисунок 1.2 Використання **шаблону** Builder для створення **збірки**

Висновки

Було детально вивчено шаблони проектування, які забезпечують ефективність у розробці складних систем. Зокрема, Builder спрощує створення об'єктів, Adapter дозволяє узгоджувати різні інтерфейси, а Command дає змогу реалізовувати зручний механізм виконання запитів. Chain of Responsibility спрощує передачу запитів між об'єктами, а Prototype забезпечує створення копій складних об'єктів. Робота з цими шаблонами навчила вирішувати типові завдання

розробки, роблячи систему більш модульною та гнучкою. Реалізовано частину функціоналу за допомогою патерну Builder.

Вихідний код

```
package org.example.domain;

import jakarta.persistence.*;

import java.time.Instant;

@Entity
@Table(name = "BUILDS")
public class Build {
    @Id
    @GeneratedValue
    private Long id;

    private Instant startTime = Instant.now();

    private Instant endTime;

    private String logFilePath;

    private String branch;

    @ManyToOne(optional = false)
    @JoinColumn(name = "project id", nullable = false)
    private Project project;

    @Enumerated(EnumType.STRING)
    @Column(name = "build_status")
    private BuildStatus buildStatus;

    public Build(Long id, Instant startTime, Instant endTime, String
logFilePath, String branch, Project project, BuildStatus buildStatus) {
        this.id = id;
        this.startTime = startTime;
        this.endTime = endTime;
        this.logFilePath = logFilePath;
        this.branch = branch;
        this.project = project;
        this.buildStatus = buildStatus;
    }

    public Build() {
    }

    public static BuildBuilder builder() {
        return new BuildBuilder();
    }

    public BuildStatus getBuildStatus() {
        return buildStatus;
    }

    public void setBuildStatus(BuildStatus buildStatus) {
        this.buildStatus = buildStatus;
    }

    public Long getId() {
        return id;
    }
```

```
public void setId(Long id) {
    this.id = id;
}

public Instant getStartTime() {
    return startTime;
}

public void setStartTime(Instant startTime) {
    this.startTime = startTime;
}

public Instant getEndTime() {
    return endTime;
}

public void setEndTime(Instant endTime) {
    this.endTime = endTime;
}

public String getLogFilePath() {
    return logFilePath;
}

public void setLogFilePath(String logFilePath) {
    this.logFilePath = logFilePath;
}

public Project getProject() {
    return project;
}

public void setProject(Project project) {
    this.project = project;
}

public String getBranch() {
    return branch;
}

public void setBranch(String branch) {
    this.branch = branch;
}

public static class BuildBuilder {
    private Long id;
    private Instant startTime;
    private Instant endTime;
    private String logFilePath;
    private String branch;
    private Project project;
    private BuildStatus buildStatus;

    BuildBuilder() {
    }

    public BuildBuilder id(Long id) {
        this.id = id;
        return this;
    }

    public BuildBuilder startTime(Instant startTime) {
        this.startTime = startTime;
        return this;
```

```
        }

        public BuildBuilder endTime(Instant endTime) {
            this.endTime = endTime;
            return this;
        }

        public BuildBuilder logFilePath(String logFilePath) {
            this.logFilePath = logFilePath;
            return this;
        }

        public BuildBuilder branch(String branch) {
            this.branch = branch;
            return this;
        }

        public BuildBuilder project(Project project) {
            this.project = project;
            return this;
        }

        public BuildBuilder buildStatus(BuildStatus buildStatus) {
            this.buildStatus = buildStatus;
            return this;
        }

        public Build build() {
            return new Build(this.id, this.startTime, this.endTime,
this.logFilePath, this.branch, this.project, this.buildStatus);
        }

        public String toString() {
            return "Build.BuildBuilder(id=" + this.id + ", startTime=" +
this.startTime + ", endTime=" + this.endTime + ", logFilePath=" +
this.logFilePath + ", branch=" + this.branch + ", project=" + this.project + ",
buildStatus=" + this.buildStatus + ")";
        }
    }
}

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.repository.ProjectRepository;l.List;
import java.nio.file.Path;
import java.util.List;
import java.util.Optional;

import static org.example.service.utility.LoggingUtility.BUILD_LOGS_DIR;
import static org.example.service.utility.LoggingUtility.LOG_FILENAME_PATTERN;

@ApplicationScoped
public class ProjectService {

    private final OpenConnections connections;
    private final ProjectRepository projectRepository;
    private final BuildRepository buildRepository;
```

```
    @Inject
    public ProjectService(ProjectRepository projectRepository, BuildRepository
buildRepository, OpenConnections connections) {
        this.projectRepository = projectRepository;
        this.buildRepository = buildRepository;
        this.connections = connections;
    }

    public Optional<Project> findById(Long id) {
        return projectRepository.findByIdOptional(id);
    }

    public List<Project> findAll() {
        return projectRepository.findAll().list();
    }

    @Transactional
    public Optional<List<Build>> getAllBuilds(Long id) {
        return findById(id).map(Project::getBuilds).map(List::copyOf);
    }

    @Transactional
    public Project save(Project project) {
        projectRepository.persist(project);
        return project;
    }

    @Transactional
    public Build createBuild(Long projectId, String branch) {
        var project =
findById(projectId).orElseThrow(IllegalArgumentException::new);
        var build = Build.builder()
                .buildStatus(BuildStatus.RUNNING)
                .branch(branch)
                .build();
        project.addBuild(build);
        buildRepository.persist(build);

        build.setLogFilePath(Path.of(BUILD_LOGS_DIR,
LOG_FILENAME_PATTERN.formatted(build.getId())).toString());
        buildRepository.persist(build);

        return build;
    }
}
```