Refactoring Report for Kidzy LMS Project

Introduction

- In modern software development, the use of design patterns plays a vital role in producing highly maintainable, scalable, and extensible applications.
- The goal of this refactoring was to enhance the existing Kidzy LMS backend by introducing key design patterns to solve certain challenges, improve code readability, and follow best practices in object-oriented design.
- In this report, I will explain each refactoring step, the problem that existed, the solution applied using a design pattern, and finally the code before and after with detailed commentary.
- We applied the following Design Patterns:
  - Factory Pattern — for Achievement creation
  - Strategy Pattern — for Email Service flexibility
  - Composite Pattern — for Quiz & Questions hierarchy
  - Proxy Pattern — to control access to Notification Service

- Singleton Pattern — for centralized JWT Utility management

→ First Design Pattern: Factory Pattern for Achievements

- **Problem**
  - Originally, Achievement creation logic was scattered and rigid. Adding new achievement types would require modifying core classes, violating the Open-Closed Principle.
- **Solution**
  - We introduced a Factory Pattern to centralize the creation of different achievement types without modifying the core service logic.

Before Refactoring (Old Code)

```
Achievement achievement = new Achievement();
achievement.setName("Super Learner");
achievement.setDescription("Completed 5 quizzes");
achievement.setPoints(100);
achievementRepository.save(achievement);
```

- Hardcoded creation.
- No flexibility if we wanted to add different types (e.g., Badge, Trophy).

After Refactoring (New Code with Factory Pattern)

AchievementFactory.java

```
package ssf.project.kidzy.factory;

import ssf.project.kidzy.model.entity.Achievement;

public class AchievementFactory {

    public static Achievement createAchievement(String name, String descri
ption, int points) {
        Achievement achievement = new Achievement();
        achievement.setName(name);
        achievement.setDescription(description);
```

```
        achievement.setPoints(points);
        return achievement;
    }
}
```

Updated Usage in AchievementServiceImpl.java

```
Achievement newAchievement = AchievementFactory.createAchievement(
    "Super Learner", "Completed 5 quizzes", 100
);
achievementRepository.save(newAchievement);
```

→ Second Design Pattern: Strategy Pattern for EmailService

- **Problem**
  - Previously, the EmailService was hardcoded to one method (e.g., SMTP), making it difficult to switch to other methods like SendGrid or MockEmail during tests.
- **Solution**
  - We used the Strategy Pattern to allow injecting different email sending strategies dynamically.

Before Refactoring (Old Code)

```
public void sendSimpleMessage(String to, String subject, String text) {
    SimpleMailMessage message = new SimpleMailMessage();
    message.setTo(to);
    message.setSubject(subject);
    message.setText(text);
    emailSender.send(message);
}
```

After Refactoring (New Code with Strategy Pattern)

EmailStrategy.java (interface)

```
package ssf.project.kidzy.strategy;

public interface EmailStrategy {
    void send(String to, String subject, String body);
}
```

SmtpEmailStrategy.java (Concrete strategy)

```
package ssf.project.kidzy.strategy;

import org.springframework.mail.SimpleMailMessage;
import org.springframework.mail.javamail.JavaMailSender;

public class SmtpEmailStrategy implements EmailStrategy {

    private final JavaMailSender emailSender;

    public SmtpEmailStrategy(JavaMailSender emailSender) {
        this.emailSender = emailSender;
    }

    @Override
    public void send(String to, String subject, String body) {
        SimpleMailMessage message = new SimpleMailMessage();
        message.setTo(to);
        message.setSubject(subject);
        message.setText(body);
        emailSender.send(message);
    }
}
```

Updated EmailServiceImpl.java

```
private EmailStrategy emailStrategy;

@Autowired
public EmailServiceImpl(EmailStrategy emailStrategy) {
    this.emailStrategy = emailStrategy;
```

```
}

@Override
public void sendSimpleMessage(String to, String subject, String body) {
    emailStrategy.send(to, subject, body);
}
```

→ Third Design Pattern: Composite Pattern for Quiz-Question Structure

- **Problem**
  - The relationship between Quiz and Questions was treated inconsistently. It was difficult to perform operations over the entire quiz as a whole (e.g., calculating total points).
- **Solution**
  - We used the Composite Pattern to treat both Quiz and Questions uniformly as AssessmentComponent.

Before Refactoring (Old Code)

```
int totalPoints = 0;
for (Question question : quiz.getQuestions()) {
    totalPoints += question.getTotalPoints();
}
```

After Refactoring (New Code with Composite Pattern)

AssessmentComponent.java (abstract class)

```
package ssf.project.kidzy.model.composite;

public abstract class AssessmentComponent {
    public abstract int getPoints();
}
```

QuestionLeaf.java (leaf)

```
package ssf.project.kidzy.model.composite;

public class QuestionLeaf extends AssessmentComponent {

    private int points;

    public QuestionLeaf(int points) {
        this.points = points;
    }

    @Override
    public int getPoints() {
        return points;
    }
}
```

QuizComposite.java (composite)

```
package ssf.project.kidzy.model.composite;

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

public class QuizComposite extends AssessmentComponent {

    private List<AssessmentComponent> children = new ArrayList<>();

    public void add(AssessmentComponent component) {
        children.add(component);
    }

    @Override
    public int getPoints() {
        return children.stream().mapToInt(AssessmentComponent::getPoints).
sum();
    }
}
```

Usage in Quiz.java

```
public AssessmentComponent toComposite() {
    QuizComposite composite = new QuizComposite();
    for (Question question : this.getQuestions()) {
        composite.add(new QuestionLeaf(question.getTotalPoints()));
    }
    return composite;
}
```

→ Fourth Design Pattern: Proxy Pattern for NotificationService

- **Problem**
  - NotificationService had no control over who can send notifications.
- **Solution**
  - We introduced a Proxy Pattern to control access and add preconditions before delegating to the real NotificationService.

After Refactoring (Proxy Code)

NotificationService.java

```
package ssf.project.kidzy.service.interfaces;

public interface NotificationService {
    void sendNotification(String message, String recipient);
}
```

RealNotificationService.java

```
package ssf.project.kidzy.service.impl;

import org.springframework.stereotype.Service;
import ssf.project.kidzy.service.interfaces.NotificationService;

@Service
```

```
public class RealNotificationService implements NotificationService {

    @Override
    public void sendNotification(String message, String recipient) {
        System.out.println("Sending notification to " + recipient + ": " + messa
ge);
    }
}
```

NotificationProxyService.java

```
package ssf.project.kidzy.service.impl;

import ssf.project.kidzy.service.interfaces.NotificationService;

public class NotificationProxyService implements NotificationService {

    private final RealNotificationService realService;

    public NotificationProxyService(RealNotificationService realService) {
        this.realService = realService;
    }

    @Override
    public void sendNotification(String message, String recipient) {
        if (recipient != null && !recipient.isEmpty()) {
            realService.sendNotification(message, recipient);
        } else {
            System.out.println("Recipient not valid. Notification not sent.");
        }
    }
}
```

→ Fifth Design Pattern: Singleton Pattern for JwtUtil

- **Problem**
  - JWT Token Utility was instantiated multiple times across the project.

- **Solution**
  - We applied the Singleton Pattern to ensure only one instance exists.

After Refactoring (Singleton Code)

JwtUtil.java

```
package ssf.project.kidzy.config;

import io.jsonwebtoken.*;
import org.springframework.stereotype.Component;
import java.util.Date;

@Component
public class JwtUtil {

    private static JwtUtil instance;

    private JwtUtil() {
    }

    public static JwtUtil getInstance() {
        if (instance == null) {
            instance = new JwtUtil();
        }
        return instance;
    }

    // JWT generation and validation methods...
}
```

→ Conclusion

- By applying these five design patterns:
  - We improved code flexibility.
  - We increased scalability and reusability.
  - We followed SOLID principles.

    - We prepared the project for future growth and easier maintenance.
- This refactoring was critical to making the Kidzy LMS system a truly professional and enterprise-grade backend.
