Astana IT University

Students: Alibek Kamiluly , Dias Karataev

Project Documentation

Course name: Software Design Patterns

Project Overview

The "Membership Management System" project aims to create a comprehensive software application for managing user subscriptions. The application will provide functionality for selling and managing memberships, including user authentication, subscription types, shopping cart management, and notification services. The primary goal is to implement various design patterns to ensure a well-structured and scalable software solution for a membership-based platform.

Performance Goal

The performance goal of the "Membership Management System" project is to develop a software application that efficiently utilizes design patterns to achieve the following objectives.

- **Scalability: The application should accommodate the addition of new subscription types and features without extensive code modifications.**
- **Maintainability: The codebase should be well-organized and easy to maintain, allowing for future updates and enhancements.**
- **Flexibility: The system should easily adapt to changing business requirements, enabling seamless integration of new features and functionalities.**
- **Reusability: Design patterns should be implemented to promote code reuse and minimize redundancy.**
- **User-Friendly Interface: The application should provide an intuitive and user-friendly interface for managing memberships and subscriptions.**

Performance Objectives

To achieve the performance goal, the project outlines specific performance objectives

- **Singleton Pattern: Create a single instance of the user management system to ensure data consistency and prevent multiple instances of user-related operations.**

- **Strategy Pattern: Implement strategies for managing the shopping cart and types of payment to provide flexibility and adaptability in handling user transactions.**
- **Decorator Pattern: Apply the Decorator pattern to enhance subscription features, allowing users to upgrade or improve their subscriptions dynamically.**
- **Adapter Pattern: Use the Adapter pattern to manage different types of messages (e.g., registration notifications, purchase confirmations) and adapt them to a unified messaging system.**
- **Observer Pattern: Apply the Observer pattern to notify subscribers about updates and events, such as new subscription offerings or changes in their account status.**
- **Factory Method Pattern: Implement factory methods for creating different types of subscriptions, catering to various membership levels and features.**
- **User-Friendly Interface: Develop an intuitive and user-friendly interface, whether through a web-based GUI or a mobile app, to allow users to manage their subscriptions seamlessly.**

Main Part

Singleton Pattern: The Singleton pattern is applied to the User Management System, ensuring a single and consistent instance for handling user-related operations. This prevents the creation of multiple instances and ensures data integrity.

```
public
class UserManagement {
    3 usages
    private static UserManagement instance;
    6 usages
    private List<User> users;

    1 usage  Dias
    public List<User> getUsers() { return users; }

    1 usage  Dias
    private UserManagement() { users = new ArrayList<>(); }

    1 usage  Dias
    public static UserManagement getInstance() {
        if (instance == null) {
            instance = new UserManagement();
        }
        return instance;
    }

    2 usages  Dias
    public void addUser(User user) { users.add(user); }

    1 usage  Dias
    public String getUserPasswordByEmail(String email) {
        for(User user : users){
            if(user.getEmail().equals(email)){
                return user.getPassword();
```

Strategy Pattern: The Strategy pattern is implemented in the shopping cart and payment modules. It allows for the dynamic selection of strategies, providing flexibility in managing shopping transactions and accommodating various payment methods.

```java
public class ShoppingCart {
    3 usages
    private int amount;
    2 usages
    private PaymentStrategy paymentStrategy;
    9 usages
    private List<Membership> productList = new ArrayList<>();

    1 usage  Dias
    public void updateProduct(Membership product){
        Optional<Membership> existingProduct = productList.stream()
                .filter(p -> p.getClass().equals(product.getClass()))
                .findFirst();

        if (existingProduct.isPresent()) {
            // Если абонемент уже существует, заменяем его в списке
            int index = productList.indexOf(existingProduct.get());
            productList.set(index, product);
        } else {
            // Если абонемент не существует, добавляем его в список
            productList.add(product);
        }
    }

    2 usages  Alibek
    public void setPaymentStrategy(PaymentStrategy paymentStrategy) { this.paymentStrategy = paymentStrategy; }
```

Decorator Pattern: The Decorator pattern is utilized to enhance subscription features. Users can dynamically upgrade their subscriptions by adding or modifying features without altering the base subscription code.

```java
5 usages  Dias
public class PersonalMembership implements MembershipDecorator {
    1 usage
    private Membership membership;
    2 usages  Dias
    public PersonalMembership(Membership decoratedMembership) { this.membership = decoratedMembership; }
    1 usage  Dias
    @Override
    public String getDescription() {
        return "+ персональный треннер";
    }
    1 usage  Dias
    @Override
    public int getPrice() { return 40000; }
}
```

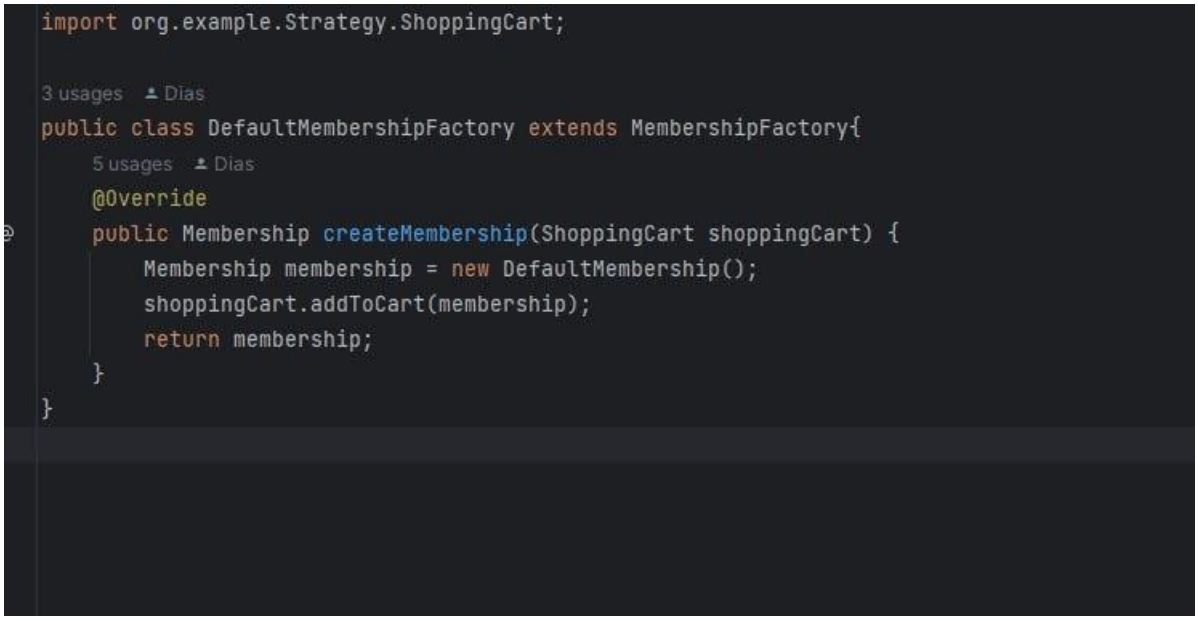
Adapter Pattern: The Adapter pattern is employed to manage different types of messages related to user interactions, such as registration and subscription notifications. It adapts these messages to a common messaging format, ensuring consistent communication.

```java
3 usages  Dias
public class JoinMembershipNotificationEmailAdapter implements EmailService {
    2 usages
    private final EmailServiceImpl legacyEmailService;

    1 usage  Dias
    public JoinMembershipNotificationEmailAdapter(EmailServiceImpl legacyEmailService) {
        this.legacyEmailService = legacyEmailService;
    }

    2 usages  Dias
    @Override
    public void sendEmail(String to) {
        String message = "Вы успешно приобрели абонемент";
        legacyEmailService.send(to,  topic: "Приобретение абонемента",message);
    }
}
```

Observer Pattern: The Observer pattern is applied to notify subscribers about important events and updates. Subscribers receive notifications about changes in subscription offerings, account status, and other relevant information.

```java
import java.util.List;

3 usages  Dias
public class MembershipEmailNotification implements Observable{
    1 usage
    private EmailServiceImpl emailService = new EmailServiceImpl();
    3 usages
    List<User> observers = new ArrayList<>();

    1 usage  Dias
    @Override
    public void addMailingSubscriber(User user) { observers.add(user); }

    no usages  Dias
    @Override
    public void removeMailingSubscriber(User user) { observers.remove(user); }

    1 usage  Dias
    @Override
    public void notifyMembers(String topic, String message) {
        for(User observer : observers){
            emailService.send(observer.getEmail(), topic,message);
            System.out.println("Send email to " + observer.getEmail());
        }
    }
```

Factory Method Pattern: Factory methods are implemented for creating different types of subscriptions. This includes the creation of basic and premium memberships, each with its set of features and privileges.



```
import org.example.Strategy.ShoppingCart;

3 usages  ▲ Dias
public class DefaultMembershipFactory extends MembershipFactory{
    5 usages  ▲ Dias
    @Override
    public Membership createMembership(ShoppingCart shoppingCart) {
        Membership membership = new DefaultMembership();
        shoppingCart.addToCart(membership);
        return membership;
    }
}
```

UML Diagram

Conclusion

The "Membership Management System" project successfully applies various design patterns to create a robust and scalable software solution for handling user subscriptions. Throughout the development process, the following key accomplishments were achieved.

Challenges Faced

- **Design Pattern Selection: Choosing appropriate design patterns for different aspects of the project required careful consideration. Deciding which patterns to apply and where to apply them was a significant challenge.**
- **Implementation Complexity: Implementing multiple design patterns within a single project introduced complexity. Ensuring that these**

patterns interacted seamlessly and did not lead to code bloat was a challenge.

- **User Interface Design: Creating a user-friendly interface was challenging. Balancing functionality with ease of use and aesthetics required careful design decisions.**
- **Testing and Validation:Verifying the correctness and robustness of the code, especially with multiple design patterns at play, was an ongoing challenge. Ensuring that the software behaves as expected and handles various scenarios was critical.**

Future Improvements

To enhance the "Membership Management System" consider the following improvements:

- **Enhanced Subscription Features:Expand the range of subscription features to offer more customization options for users.**
- **Advanced User Interface: Develop a more sophisticated and user-friendly graphical user interface (GUI) with enhanced interactive features.**
- **Database Integration: Implement a database system for persisting inventory and transaction data, facilitating historical sales data analysis, inventory management, and scalability.**
- **Integration with External Systems:Explore integration with external systems, such as payment gateways, for more streamlined payment processing.**
- **User Analytics:Implement user analytics to gather insights into user behavior and preferences, facilitating targeted improvements.**
- **User Authentication:Enhance security by implementing user authentication and authorization features for store management.**

References

- **Java SE Development Kit (JDK): The project was developed in Java using the JDK for Java 20.**
- **Integrated Development Environment (IDE): IntelliJ IDEA Ultimate.**
- **UML Diagram Tools: dbdiagram.io**