**Cinema System**

Group:se-2222

Team: Zhangazinov Timur, Yelemessova Aruzhan

Your project aims to create a simulated library system through an interactive console application. The core functionalities you've outlined involve managing books, users, lending, returning, providing recommendations, and notifying about user changes.

**Idea of the Project:** The idea is to replicate a library system where users can interact with a console-based application to perform various actions related to book management, user accounts, book lending, and returns. The application serves as a simplified model of a real library system, allowing users to handle book transactions and user accounts efficiently.

**Purpose of the Work**: The purpose is to implement a practical demonstration of software design patterns within a library management context. It provides an opportunity to apply different design patterns, fostering an understanding of their implementation and efficacy within a real-world scenario. Additionally, the project serves as a learning tool for users to grasp the nuances of managing a library system using an interactive console application.

**Objectives of the Work:**

* **Functionality Implementation**: Develop the core functions (add book, show available books, add user, lend book, return book) ensuring they work seamlessly within the console application.
* **Design Pattern Application**: Apply various software design patterns (such as Factory, Singleton, Observer, etc.) to different aspects of the application to enhance its scalability, maintainability, and flexibility.
* **User Notifications:** Implement a notification system to alert users about changes within the system, such as book availability, user actions, or system updates.
* **Recommendation Feature**: Develop a recommendation system to suggest books based on user preferences, borrowing history, or other relevant factors.
* **Documentation and Presentation**: Prepare comprehensive documentation explaining the design patterns used, the functionality of the application, and the rationale behind the design choices. Additionally, create a presentation to showcase the project's functionality, design patterns, and overall implementation.

By achieving these objectives, your project not only demonstrates a functional library system but also highlights the importance and effectiveness of employing design patterns in software development.

**Explanations:**

Implementing the **Decorator Pattern** to enhance the functionality of the Book class in the library management system allows for dynamic addition of features without altering the original structure of the Book class

public abstract class BookDecorator implements IBook {  
 protected IBook decoratedBook;  
  
 public BookDecorator(IBook decoratedBook) {  
 this.decoratedBook = decoratedBook;  
 }  
  
 @Override  
 public String getDescription() {  
 return decoratedBook.getDescription();  
 }  
  
 // other methods delegating to decoratedBook  
}

Applying the **Observer Pattern** to the User class in the library management system enables tracking user state changes and notifying various components interested in these changes. With observer pattern, any changes in the user's state (like borrowing or returning books) automatically notify all registered observers, allowing actions to be taken in response to these changes. This design enhances the User class, making it more flexible and easily integrable with various parts of the system interested in user activities.

User user = new User("id123", "password", "John Doe", "Member", 1);  
  
// Creating an observer  
 UserObserver notificationSystem = new NotificationSystem();  
  
// Registering the observer  
 user.addObserver(notificationSystem);  
  
// Performing actions that trigger notifications  
 user.borrowBook(new Book("The Great Gatsby", "123456789", "Fiction", "F. Scott Fitzgerald", 1925, 1, 1));  
 user.returnBook(new Book("1984", "987654321", "Dystopian", "George Orwell", 1949, 1, 1));

This implementation allows the Login class to delegate the creation of User objects to a factory, enabling the creation of different user types based on specific characteristics. The Factory Method Pattern enhances flexibility and maintainability, especially when new types of users are introduced into the system, by centralizing the creation logic within dedicated factory classes.

Implementing **the Singleton Pattern** for the DataBase class in your library management system is a great way to ensure a single, globally accessible instance for managing shared resources like database connections.

public class DataBase {  
 private static DataBase instance;  
 private String dataFilePath;  
  
 // Private constructor  
 private DataBase(String dataFilePath) {  
 this.dataFilePath = dataFilePath;  
 // Initialization code, like loading data from the file  
 }  
  
 // Static method to get the instance  
 public static DataBase getInstance(String dataFilePath) {  
 if (instance == null) {  
 instance = new DataBase(dataFilePath);  
 }  
 return instance;  
 }  
  
 // Other methods of the DataBase class  
}

By applying the Singleton Pattern to the DataBase class, you're ensuring consistent interaction with a shared database resource across your library management system. This approach maintains data integrity and prevents issues arising from multiple instances manipulating the same resource simultaneously

Implementing the **Adapter Pattern** will facilitate the integration of an external book recommendation service that has a different interface from your current system. Here's how you can apply this pattern

public class ExternalBookRecommendationService {  
 public List<String> getRecommendations(List<String> borrowedIsbns) {  
 // Logic to return book recommendations based on borrowed ISBNs  
 return new ArrayList<>(); // Returns a list of recommended book titles  
 }  
}

By implementing the Adapter Pattern, your Library system can seamlessly integrate with an external book recommendation service, even though the interfaces of both systems initially appeared incompatible. The adapter acts as a bridge, converting your system's data format into a compatible form for the external service. This approach allows your application to provide personalized book recommendations without requiring internal data structure modifications, showcasing the adaptability and versatility of the Adapter Pattern in integrating disparate systems.

Implementing the **Strategy Pattern** in the ManagerControls class will make the code more modular and maintainable.

* Encapsulates each menu option's functionality in its own class, adhering to the Strategy Pattern.
* Utilizes a map within ManagerControls to associate menu options with their corresponding strategy objects.
* When a menu option is chosen, the ManagerControls delegates the execution to the appropriate strategy object.

DataBase db = new DataBase();  
 UserFactory userFactory = new MemberUserFactory(); // or AdminUserFactory based on the context  
 Login login = new Login(db, userFactory);  
  
 User user = login.menu();

This implementation allows the Login class to delegate the creation of User objects to a factory, enabling the creation of different user types based on specific characteristics. **The Factory Method Pattern** enhances flexibility and maintainability, especially when new types of users are introduced into the system, by centralizing the creation logic within dedicated factory classes.

**Project Outcomes:**

1. Successful Implementation: Achieved a functional library management system with multiple design patterns integrated for improved structure and flexibility.
2. Pattern Understanding: Enhanced understanding of various design patterns and their practical application within a real-world project.

**Challenges Faced:**

1. Integration Complexity: Adapting different design patterns to work harmoniously within the system might have posed integration challenges.
2. User Interaction Design: Crafting a user-friendly console interface while ensuring all functionalities were easily accessible and understandable.

**Future Improvements:**

1. Enhanced Recommendation System: Refine the recommendation strategies by incorporating machine learning algorithms or collaborative filtering techniques for more accurate book suggestions.
2. User Authentication: Implement user authentication and authorization mechanisms to ensure secure access to the system.
3. Graphical User Interface (GUI): Develop a graphical interface for the library system, providing a more visually appealing and intuitive user experience.
4. Database Integration: Integrate a database to persistently store book and user information, enabling data retention between application sessions.

Your project has laid a solid foundation by implementing core functionalities and integrating various design patterns. Moving forward, refining these features, enhancing user experience, and incorporating advanced functionalities could make the system more robust and user-friendly, preparing it for potential real-world application or expansion.