**E-Library Management System - Design Document**

**1. Overview**

The E-Library Management System is a console-based application that allows users to manage books, user profiles, and loans. It incorporates creational, structural, and behavioral design patterns to ensure scalability, flexibility, and adherence to SOLID principles. This document details the application design, including key components, applied design patterns, and their purpose.

**2. System Requirements**

**Functional Requirements**

1. **Book Management**
   * Add, update, delete, and view books with attributes like title, author, ISBN, and category.
2. **User Management**
   * Register, update, delete user profiles with roles: Member and Librarian.
3. **Loan Processing**
   * Borrow and return books, with tracking of loan status and due dates.
4. **Notifications**
   * Notify users of due dates and overdue books via console messages.

**Technical Requirements**

* In-memory data storage.
* Error handling and logging.
* Console-based interface.
* Open for extension but closed for modification to allow for future growth.

**3. Design Patterns and Their Application**

**3.1 Creational Patterns**

**Factory Method**

* **Purpose**: Create instances of different user roles (Member and Librarian) in a flexible and scalable way.
* **Implementation**:
  + A UserFactory class will encapsulate the creation logic. It will have a createUser(role: String) method that returns either a Member or Librarian instance based on the input role.
* **Benefit**: Allows easy extension for adding more user roles without modifying the core logic.

**Singleton**

* **Purpose**: Ensure that only one instance of the library catalogue and user registry exists.
* **Implementation**:
  + LibraryCatalogue and UserRegistry will both be singleton classes.
  + The singleton instance is accessed through a getInstance() method.
* **Benefit**: Prevents duplicate catalogues and registries, ensuring consistency across operations.

**3.2 Structural Patterns**

**Adapter**

* **Purpose**: Integrate an external notification service that doesn’t match the internal interface.
* **Implementation**:
  + Create an INotificationService interface that defines methods for notifications.
  + Implement an NotificationAdapter class that adapts the external notification service to the system's interface.
* **Benefit**: Allows flexibility in changing notification services in the future without impacting the core application.

**Decorator**

* **Purpose**: Add additional responsibilities to books (e.g., reserving a book) without modifying the core Book class.
* **Implementation**:
  + A ReserveDecorator class will be created that extends a BookComponent interface.
  + This decorator can add reservation functionality to a book.
* **Benefit**: Keeps the base Book class simple and open to additional features without direct modifications.

**Facade**

* **Purpose**: Simplify the interface for accessing various subsystems such as book management, loan processing, and notifications.
* **Implementation**:
  + A LibraryFacade class will provide simplified methods like addBook, borrowBook, returnBook, and notifyUser.
* **Benefit**: Offers a straightforward interface for managing common operations, improving code readability and ease of use.

**3.3 Behavioral Patterns**

**Command**

* **Purpose**: Encapsulate book management requests (add, update, delete) as objects, allowing for easy management of operations.
* **Implementation**:
  + Command classes (AddBookCommand, UpdateBookCommand, DeleteBookCommand) encapsulate each operation.
  + A CommandInvoker class will be responsible for executing commands.
* **Benefit**: Decouples the request execution from the main application, allowing undo/redo operations if needed in the future.

**Observer**

* **Purpose**: Notify users about due dates and overdue books.
* **Implementation**:
  + LibraryCatalogue or LoanProcessor can act as subjects that manage subscribers (users).
  + Implement attach and notify methods to alert users of due dates or overdue notifications.
* **Benefit**: Notifies users automatically when due dates approach, enhancing usability and user experience.

**Iterator**

* **Purpose**: Access elements of book and user lists sequentially.
* **Implementation**:
  + BookIterator and UserIterator classes will implement custom iterator functionality for these lists.
  + A LibraryIterator interface will define hasNext() and next() methods for navigation.
* **Benefit**: Provides a flexible way to traverse through lists without exposing their underlying representations.

**4. Class Diagram**

**Key Classes and Their Relationships**

1. **User Management**:
   * User: Abstract class with attributes like name, role, etc.
   * Member and Librarian: Subclasses of User.
   * UserFactory: Uses Factory Method to create users.
   * UserRegistry: Singleton class to manage registered users.
2. **Book Management**:
   * Book: Class with attributes title, author, ISBN, etc.
   * LibraryCatalogue: Singleton class that maintains the list of books.
   * ReserveDecorator: Adds reservation functionality to Book.
   * BookIterator: Provides sequential access to books.
3. **Loan Processing**:
   * LoanProcessor: Handles loan requests and returns.
   * LoanCommand: Base class for loan commands.
   * BorrowBookCommand and ReturnBookCommand: Specific commands for borrowing and returning books.
4. **Notifications**:
   * INotificationService: Interface for notification.
   * NotificationAdapter: Adapts external notification services.
   * Observer: Users are registered as observers for due date notifications.
5. **Facade**:
   * LibraryFacade: Provides simplified methods for core library functions.

**5. Key Interactions**

1. **Adding a New Book**
   * The client calls LibraryFacade.addBook().
   * LibraryFacade forwards this to the LibraryCatalogue, which then uses the CommandInvoker to execute an AddBookCommand.
2. **User Registration**
   * The client calls LibraryFacade.registerUser().
   * UserFactory creates the appropriate user instance (Member or Librarian) based on the role provided.
3. **Borrowing a Book**
   * The client calls LibraryFacade.borrowBook().
   * LibraryFacade invokes LoanProcessor to create and execute a BorrowBookCommand.
   * LoanProcessor attaches the user to the list of observers for notification purposes.
4. **Notification for Due Dates**
   * When the loan period approaches, LoanProcessor notifies all observers (users) using the Observer pattern.

**6. Assumptions**

1. Only console-based notifications are required; no actual email or SMS integration is needed.
2. Static, in-memory data is used, with no persistence mechanism.
3. The system is expected to scale in terms of new user roles, book properties, and additional notification methods.
4. Notification format and frequency are simplified for this implementation.