**-Các Design Pattern đã sử dụng trong dự án**

* **Singleton Pattern**
  + **Mục đích: Đảm bảo chỉ có một thể hiện duy nhất của một lớp và cung cấp điểm truy cập toàn cục cho nó.**
  + **Các lớp liên quan: LibraryManager, DateProvider.**
* **Observer Pattern**
  + **Mục đích: Định nghĩa mối quan hệ một-nhiều giữa các đối tượng để khi một đối tượng thay đổi trạng thái, tất cả các đối tượng phụ thuộc của nó sẽ được thông báo và cập nhật tự động.**
  + **Các lớp/giao diện liên quan: LibraryManager (Subject), Observer (Interface).**
* **Factory Method / Abstract Factory Pattern**
  + **Mục đích: Cung cấp một giao diện để tạo đối tượng trong một siêu lớp, nhưng cho phép các lớp con thay đổi loại đối tượng sẽ được tạo.**
  + **Các lớp/giao diện liên quan: BookFactory (Interface), AcademicBookFactory (Concrete Factory), EntertainmentBookFactory (Concrete Factory).**
* **Decorator Pattern**
  + **Mục đích: Gắn thêm các hành vi và chức năng mới vào một đối tượng hiện có một cách linh hoạt mà không làm thay đổi cấu trúc của nó.**
  + **Các lớp liên quan: BookDecorator (Abstract Decorator), FavoriteBookDecorator (Concrete Decorator).**
* **Strategy Pattern**
  + **Mục đích: Định nghĩa một họ các thuật toán, đóng gói mỗi thuật toán thành một lớp riêng biệt và làm cho chúng có thể hoán đổi cho nhau.**
  + **Các lớp/giao diện liên quan: FeeCalculator (Interface), DailyFeeCalculator (Concrete Strategy), QuantityFeeCalculator (Concrete Strategy), FeeContext (Context).**
* **Data Access Object (DAO) Pattern**
  + **Mục đích: Tách biệt logic truy cập dữ liệu ra khỏi logic nghiệp vụ của ứng dụng.**
  + **Các lớp liên quan: BookDAO, LoanDAO, MemberDAO.**

**Hiểu vấn đề cần giải quyết**

Vấn đề chính mà dự án của bạn giải quyết là **quản lý một thư viện**, bao gồm các khía cạnh:

1. **Quản lý Sách:** Thêm, xóa, cập nhật, tìm kiếm sách; phân loại sách (sách in, sách điện tử); và đánh dấu sách yêu thích.
2. **Quản lý Thành viên:** Thêm, xóa, cập nhật, tìm kiếm thành viên; quản lý thông tin đăng nhập.
3. **Quản lý Khoản mượn:** Ghi nhận việc mượn và trả sách; tính toán phí quá hạn dựa trên các quy tắc khác nhau.
4. **Tương tác dữ liệu:** Cách ứng dụng giao tiếp với cơ sở dữ liệu một cách an toàn và hiệu quả.
5. **Mở rộng và Bảo trì:** Làm cho hệ thống dễ dàng mở rộng khi có các yêu cầu mới (ví dụ: thêm loại sách mới, phương pháp tính phí mới) và dễ dàng bảo trì.
6. **Thông báo sự kiện:** Cách các thành phần khác nhau của hệ thống có thể phản ứng với các thay đổi xảy ra.

Bây giờ, chúng ta sẽ đi sâu vào việc các mẫu thiết kế đã giúp giải quyết những vấn đề này như thế nào.

**Giải quyết vấn đề bằng các Design Pattern**

**1. Mẫu Singleton (Singleton Pattern)**

* **Vấn đề cần giải quyết:** Đảm bảo rằng chỉ có **một thể hiện duy nhất** của một lớp trong toàn bộ ứng dụng và cung cấp một điểm truy cập toàn cục cho thể hiện đó. Trong dự án của bạn, điều này áp dụng cho:
  + **LibraryManager**: Đây là lớp quản lý trung tâm, việc có nhiều thể hiện có thể dẫn đến trạng thái không nhất quán của dữ liệu sách, thành viên, và khoản mượn trong bộ nhớ.
  + **DateProvider**: Để đảm bảo rằng tất cả các phần của ứng dụng (đặc biệt là trong môi trường kiểm thử hoặc môi trường có nhu cầu về ngày/giờ đồng bộ) đều sử dụng cùng một "ngày hiện tại", việc có một thể hiện duy nhất là cần thiết.
* **Vì sao lại giải quyết bằng pattern này:**
  + **Kiểm soát trạng thái:** Ngăn chặn việc tạo ra các đối tượng quản lý trùng lặp có thể gây ra xung đột dữ liệu hoặc logic.
  + **Tiết kiệm tài nguyên:** Đối với các tài nguyên nặng hoặc cần khởi tạo phức tạp, Singleton giúp tránh việc khởi tạo nhiều lần.
  + **Điểm truy cập toàn cục:** Dễ dàng truy cập đến thể hiện duy nhất từ bất kỳ đâu trong ứng dụng mà không cần truyền tham số.
* **Giải quyết vấn đề như thế nào:**
  + **Constructor riêng tư (- LibraryManager(), - DateProvider()):** Ngăn không cho các lớp khác tạo ra các thể hiện mới trực tiếp bằng từ khóa new.
  + **Biến static duy nhất (- static instance):** Giữ một tham chiếu đến thể hiện duy nhất của lớp.
  + **Phương thức static công khai (+ static getInstance()):** Đây là điểm truy cập toàn cục. Nó kiểm tra nếu thể hiện đã tồn tại, nếu chưa thì tạo mới (chỉ một lần duy nhất), sau đó trả về thể hiện đó.

**2. Mẫu Observer (Observer Pattern)**

* **Vấn đề cần giải quyết:** LibraryManager cần thông báo cho các thành phần khác của hệ thống (ví dụ: giao diện người dùng, các module ghi log) về những thay đổi quan trọng (ví dụ: sách được thêm, thành viên được thêm, khoản mượn được tạo) mà không cần biết chi tiết về các thành phần đó. Vấn đề là **giảm sự phụ thuộc chặt chẽ (tight coupling)** giữa các đối tượng.
* **Vì sao lại giải quyết bằng pattern này:**
  + **Tách biệt (Decoupling):** LibraryManager (Subject) không cần biết cụ thể ai đang "lắng nghe" nó. Nó chỉ cần biết về giao diện Observer. Điều này giúp hệ thống linh hoạt và dễ mở rộng.
  + **Phản ứng động:** Các thành phần khác có thể đăng ký (addObserver) hoặc hủy đăng ký (không có trong sơ đồ nhưng là một phần của pattern) để nhận thông báo mà không cần thay đổi code của LibraryManager.
* **Giải quyết vấn đề như thế nào:**
  + **Observer interface:** Định nghĩa phương thức + update(message: String): void, đây là hợp đồng mà tất cả các đối tượng muốn nhận thông báo phải tuân thủ.
  + **LibraryManager (Subject):**
    - Duy trì một danh sách các observers (- observers: List<Observer>).
    - Có phương thức + addObserver(observer: Observer): void để các đối tượng Observer đăng ký.
    - Có phương thức + notifyObservers(message: String): void để lặp qua danh sách observers và gọi update(message) trên từng observer, thông báo về thay đổi.

**3. Mẫu Factory Method / Abstract Factory (Factory Method / Abstract Factory Pattern)**

* **Vấn đề cần giải quyết:** Cần tạo các đối tượng Book thuộc các loại khác nhau (PrintedBook, EBook) mà không muốn logic khởi tạo cụ thể bị phân tán khắp nơi trong code. Vấn đề này trở nên phức tạp hơn khi bạn muốn dễ dàng thêm các loại sách mới mà không sửa đổi code hiện có.
* **Vì sao lại giải quyết bằng pattern này:**
  + **Tách biệt logic khởi tạo:** Logic tạo đối tượng được tập trung vào các lớp Factory, giúp code sạch sẽ hơn.
  + **Tuân thủ Open/Closed Principle:** Bạn có thể dễ dàng thêm các loại sách mới (ví dụ: AudioBook) bằng cách tạo một AudioBookFactory mới mà không cần thay đổi BookFactory hoặc các code sử dụng Factory.
  + **Giảm phụ thuộc:** Các lớp như BookDAO không cần biết chi tiết về PrintedBook hay EBook; chúng chỉ cần biết về BookFactory.
* **Giải quyết vấn đề như thế nào:**
  + **abstract class BookFactory:** Định nghĩa phương thức trừu tượng + createBook(id: int, title: String, author: String): Book. Đây là giao diện để tạo sách.
  + **Các lớp Factory cụ thể (AcademicBookFactory, EntertainmentBookFactory):** Kế thừa từ BookFactory và triển khai phương thức createBook() để trả về một loại Book cụ thể (ví dụ: AcademicBookFactory tạo PrintedBook, EntertainmentBookFactory tạo EBook).
  + **Sử dụng:** Các lớp như BookDAO sẽ sử dụng BookFactory để tạo đối tượng Book mà không cần quan tâm đến chi tiết khởi tạo của PrintedBook hay EBook.

**4. Mẫu Decorator (Decorator Pattern)**

* **Vấn đề cần giải quyết:** Cần thêm các chức năng hoặc thuộc tính mới vào một đối tượng Book một cách linh hoạt, tại thời gian chạy, mà không làm thay đổi cấu trúc của lớp Book gốc hoặc tạo ra một lượng lớn các lớp con phức tạp để kết hợp các chức năng khác nhau. Ví dụ: thêm khả năng "đánh dấu yêu thích" cho một cuốn sách.
* **Vì sao lại giải quyết bằng pattern này:**
  + **Linh hoạt động:** Chức năng có thể được thêm hoặc bớt vào một đối tượng một cách linh hoạt trong thời gian chạy.
  + **Tránh "bùng nổ" lớp con:** Thay vì tạo FavoritePrintedBook, FavoriteEBook, ReadableFavoritePrintedBook, v.v., bạn chỉ cần có FavoriteBookDecorator và áp dụng nó lên bất kỳ Book nào.
  + **Tuân thủ Single Responsibility Principle:** Mỗi Decorator có một trách nhiệm duy nhất (ví dụ: FavoriteBookDecorator chỉ lo việc đánh dấu yêu thích).
* **Giải quyết vấn đề như thế nào:**
  + **abstract class BookDecorator:** Kế thừa từ Book và chứa một tham chiếu đến đối tượng Book mà nó sẽ "trang trí" (# decoratedBook: Book). Nó ủy quyền các phương thức của Book cho decoratedBook.
  + **FavoriteBookDecorator:** Kế thừa từ BookDecorator. Nó ghi đè phương thức getTitle() để thêm hậu tố "[Yêu thích]" và thêm các phương thức mới như markAsFavorite(), unmarkAsFavorite() để tương tác với BookDAO và cập nhật trạng thái yêu thích trong cơ sở dữ liệu.

**5. Mẫu Strategy (Strategy Pattern)**

* **Vấn đề cần giải quyết:** Có nhiều cách khác nhau để tính phí quá hạn cho một khoản mượn (Loan), và bạn muốn có thể dễ dàng chuyển đổi giữa các phương pháp tính toán này (ví dụ: tính theo ngày, tính theo số lượng) mà không làm thay đổi code của lớp Loan.
* **Vì sao lại giải quyết bằng pattern này:**
  + **Khả năng hoán đổi (Interchangeability):** Các thuật toán (chiến lược tính phí) được đóng gói vào các lớp riêng biệt và có thể thay thế cho nhau.
  + **Tách biệt logic:** Lớp Loan không cần biết chi tiết về cách phí được tính toán; nó chỉ cần biết rằng có một FeeCalculator để thực hiện công việc đó.
  + **Tuân thủ Open/Closed Principle:** Dễ dàng thêm các chiến lược tính phí mới mà không cần sửa đổi Loan hoặc FeeContext.
* **Giải quyết vấn đề như thế nào:**
  + **interface FeeCalculator:** Định nghĩa phương thức + calculateFee(loan: Loan): double. Đây là giao diện chung cho tất cả các chiến lược tính phí.
  + **Các lớp chiến lược cụ thể (DailyFeeCalculator, QuantityFeeCalculator):** Triển khai giao diện FeeCalculator với logic tính phí riêng của chúng.
  + **class FeeContext:** Chứa một tham chiếu đến một đối tượng FeeCalculator (- feeCalculator: FeeCalculator). Nó có phương thức + setFeeCalculator(calculator: FeeCalculator): void để thay đổi chiến lược tại thời gian chạy và + calculateFee(loan: Loan): double để ủy quyền việc tính toán cho chiến lược hiện tại.
  + **class Loan:** "Có một" (\*--) FeeContext và sử dụng nó để tính phí quá hạn thông qua phương thức calculateOverdueFee(). Nó cũng lưu feeStrategy dưới dạng String để đồng bộ với DB và có phương thức setFeeStrategy để thay đổi chiến lược.

Các mẫu thiết kế này kết hợp lại giúp xây dựng một hệ thống thư viện **mạnh mẽ, linh hoạt, dễ mở rộng và dễ bảo trì**, phù hợp với một thiết kế MVP ban đầu và có thể phát triển trong tương lai.