Lab 5 – Refactoring Library Management System

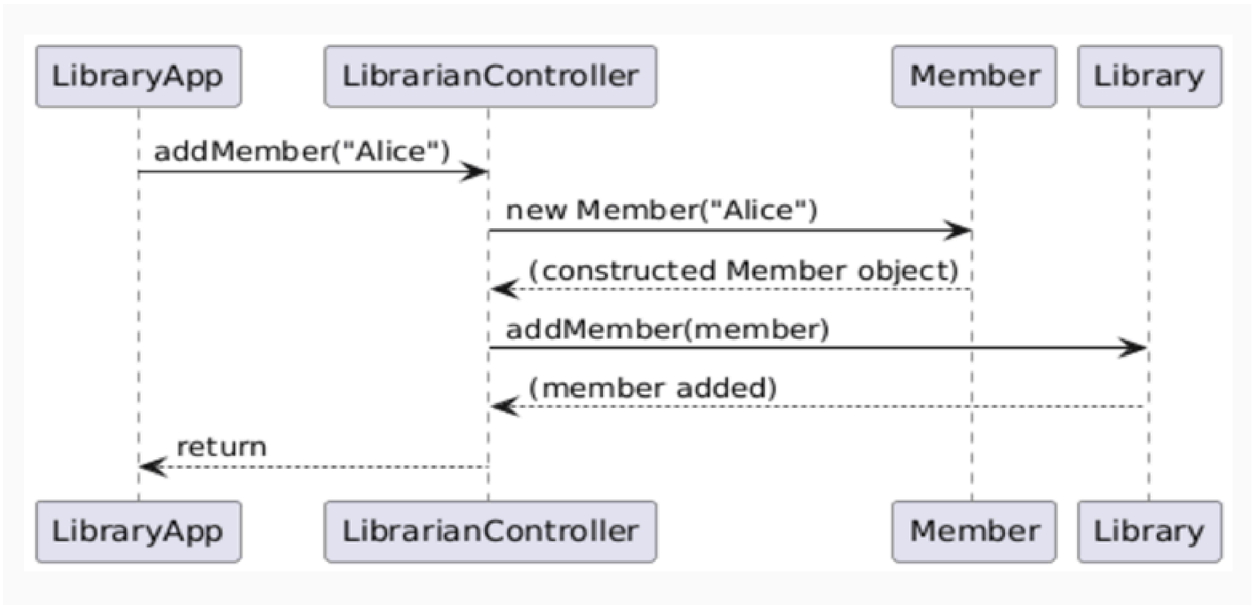
Shondel Hayles

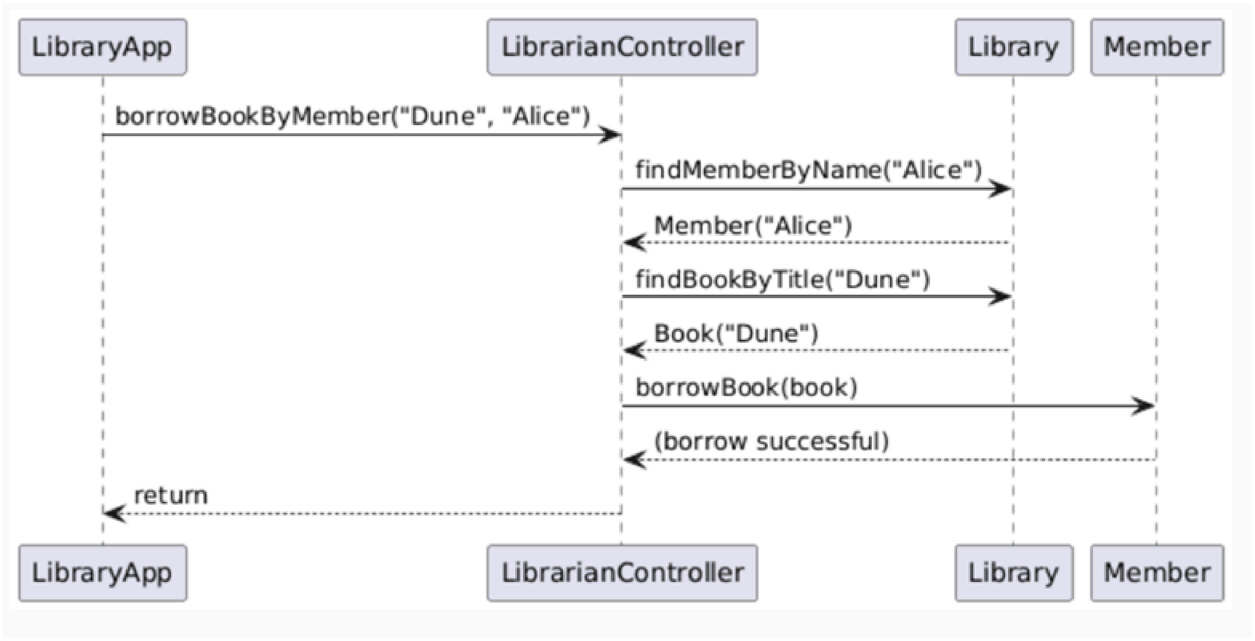
Daniel Cuevas

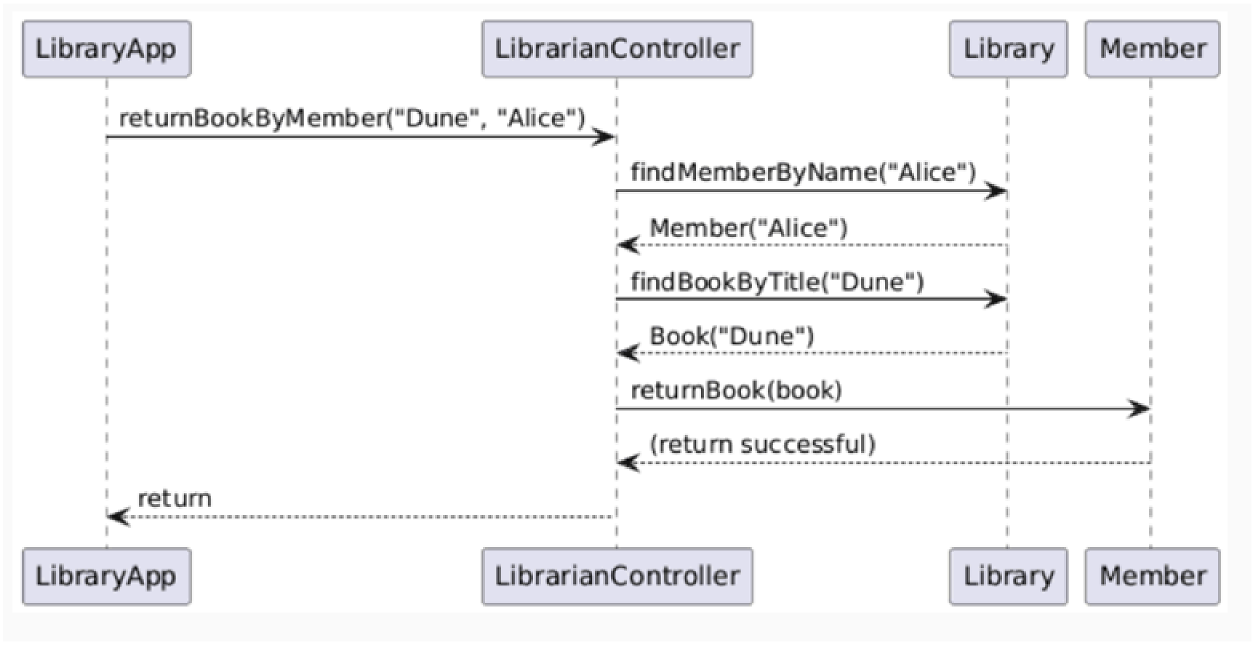
Simon Velez

https://github.com/Shondiiee/Lab5-RefactoringLibraryManagementSystem/commit/591cbec795069123c34708d88e57f7a2bffb76df

Part 1: The Sequence Diagrams







Part 3: Dependency Inversion

Q1: Why did you introduce the Book interface, and how does this relate to the Dependency Inversion Principle (DIP)?

The Book interface was introduced to break the tight coupling between the high-level classes (Member and Library) and the low-level implementation class (PaperBook).

This relates directly to the Dependency Inversion Principle (DIP):

- Before Refactoring: The high-level module (Member) depended on the low-level concrete module (Book/PaperBook).
- After refactoring, both the high-level modules (Member, Library) and the low-level module (PaperBook) now depend on the abstraction (Book interface). This reverses the typical dependency relationship, fulfilling the DIP.

Q2: How does this design improve the flexibility of the system?

The refactored design dramatically improves flexibility by adhering to the Open/Closed Principle (OCP).

- The Member class now operates on the generic Book interface, meaning it doesn't care if the object is a PaperBook, an EBook, or an AudioBook.

To add a new item, we only need to create a new class that implements the Book interface. No existing high-level code in Member or Library needs to change. This is an improvement over the old design, which required modifying the Member class for every new book type.

Q3: Can you explain how your changes support the Open/Closed Principle (OCP)?

The Open/Closed Principle (OCP) states that software entities (classes, modules, functions, etc.) should be open for extension but closed for modification.

Our changes support OCP as follows:

- Open for Extension: The system is open to new types of books (e.g., e-book, AudioBook). To add a new type, we simply extend the system by creating a new class that implements the existing Book interface.
- Closed for Modification: The high-level core logic in the Member and Library classes is now closed for modification. These classes depend only on the stable Book interface. We can introduce new book types without ever touching and potentially breaking the code in the Member or Library classes.

Q4:What did you learn about the benefits of using abstractions and interfaces in this example?

We learned that using abstractions and interfaces provides critical benefits:

- Decoupling: Interfaces break direct dependencies, transforming a tightly coupled system into a loosely coupled one. This drastically reduces maintenance overhead and potential for bugs.
- Adherence to Principles: They are the key mechanism for satisfying the Dependency Inversion Principle (DIP) and the Open/Closed Principle (OCP), which are foundational to robust, scalable object-oriented design.

<u>**Part 4: Other book types**</u>

Q1: Document any changes made to the code to accommodate this updated functionality.

The following changes were made to support the new eBook and AudioBook types, enabled by the existing Book interface:

New Classes:

- ○ **EBook.java**: Implemented the Book interface, added a unique property (fileSizeMB), and updated toString().
- ○ **AudioBook.java**: Implemented the Book interface, added a unique property (durationMinutes), and updated toString().

PaperBook.java: Verified it implements the Book interface (renamed from Book.java in the previous step).
LibrarianController.java:
- Renamed addBook(String title) to addPaperBook(String title).
- Added two new methods: addEBook(String title) and addAudioBook(String title), each responsible for instantiating its respective concrete class and calling the library.addBook(Book book).
- All core methods (borrowBookByMember, returnBookByMember, etc.) remain **unchanged**, confirming that the Member and Library classes were successfully decoupled from the concrete type.

LibraryApp.java: Modified the static helper methods (addPaperBook, addEBook, addAudioBook) and the main loop to demonstrate adding and interacting with all three different book types.
Test Suite:
- **TestMultiBookTypes.java**: A new test class was created to explicitly verify that Member correctly handles borrowing and returning all three polymorphic types (PaperBook, EBook, AudioBook).
- **AllTests.java**: Updated to include the TestMultiBookTypes class.
- **Existing Tests**: All existing test classes that instantiated the old Book class (e.g., TestBorrowBooks, TestAddRemoveBooks) were updated to instantiate the concrete PaperBook class, using the Book interface as the variable type where possible.

Q2: Reflect on the Dependency Inversion Principle (DIP).

The application of the Dependency Inversion Principle (DIP) in this refactoring is clearly visible. The high-level modules (Member and Library) depend only on the high-level abstraction (Book interface), not the low-level details of PaperBook, EBook, or AudioBook.

What changes would you need to make in the current design to support renting items such as rooms, laptops, and video DVDs?

To support renting entirely new types of items like rooms, laptops, and video DVDs, the core abstraction would need to be generalized.

1. **Refactor/Rename Book Interface:** The current Book interface is too specific. It should be renamed to a more generic abstraction like **RentalItem** or **Rentable**.
2. **Update Interface Methods:** The methods like getTitle() and getIsAvailable() are fine, but could be broadened (e.g., getIdentifier()). The signature for borrowing and returning

(which happens in the Member class) would remain the same: member.borrow(Rentable item).

3. **Create New Concrete Classes:**
    - Implement **Room** class: Implements Rentable (with properties like capacity, location).
    - Implement **Laptop** class: Implements Rentable (with properties like serialNumber, OS).
    - Implement **VideoDVD** class: Implements Rentable (with properties like runningTime, rating).
4. **Update Controllers:** The LibrarianController would require new instantiation methods (e.g., addRoom(), addLaptop()) to create these new concrete types and pass them to Library.addBook(Rentable item).

Provide an explanation of how you would modify the test cases to test these new rental types.

Test cases would be modified similarly to how TestMultiBookTypes.java was created:

1. **New Test Class:** Create a new test class, e.g., TestMultiRentalItems.java.
2. **Instantiation:** In the setup or test methods, instantiate all concrete types: Laptop laptop = new Laptop(...), Room room = new Room(...), etc.
3. **Core Interaction Testing:** Write tests that pass these different concrete objects to the Member's borrowing and returning methods:
    - member.borrow(laptop);
    - assertTrue(member.getBorrowedItems().contains(laptop));
    - member.returnItem(room);
4. **Unique Property Testing:** Write specific assertion tests to ensure the unique properties of the new classes are handled correctly (e.g., assertEquals("Windows", laptop.getOS())).
5. **Polymorphic Collection Testing:** Assert that the Library's catalog and the Member's borrowed list can hold a mix of all item types.

How would you implement this extended functionality if the DIP were not applied? Discuss the impact this would have on maintainability and flexibility.

If the DIP was **not** applied, the Member class would have remained directly dependent on the initial concrete class.

Maintainability – Extremely Low -Every new item type (DVD, Camera) requires modifying the *existing* Member and Library classes. This is high maintenance overhead and a breeding ground for bugs.

Flexibility – Zero - The system is rigid. If the business model changes (e.g., PaperBook is split into Hardcover and Softcover), the Member class breaks.

In summary, implementing without DIP would violate the Open/Closed Principle, leading to a highly fragile system that is almost impossible to maintain or extend.

Part 5: Borrowing Services

The refactoring process in Part 5 was primarily driven by the Single Responsibility Principle (SRP), aiming to resolve the violation where the Member class handled both its identity/state (Knowledge Expert) and the complex business rules of borrowing/returning (Process Expert). To ensure the Member adheres to SRP, all transaction-related logic, including checking book availability, verifying the member hasn't already borrowed the book, and updating the state of both the book and the member's borrowed list, was delegated to the newly created BorrowingService via the BorrowingServiceAPI interface. This functional separation ensures that changes to borrowing rules require modifying the service, leaving the Member class stable and closed for modification.

We chose the BorrowingService class because borrowing is inherently a cross-entity business process that manages the interaction between Member and Book, making a dedicated Service/Interactor layer the most appropriate architectural abstraction. While the current implementation tightly couples Member to the concrete BorrowingService via direct instantiation (new BorrowingService()), achieving stricter Dependency Inversion Principle (DIP) compliance would require using Dependency Injection (passing the BorrowingServiceAPI instance through the Member's constructor). This trade-off prioritized demonstrating clear functional separation for SRP within the provided code constraints, while acknowledging that injection is necessary for true decoupling and testability in a production environment.

Part 6:Refactoring the BorrowingServiceAPI

The introduction of the BorrowingBookResult class significantly enhances the system's ability to communicate the outcome of complex transactions, improving the user experience and enabling more detailed business logic checks within the BorrowingService.

Three new outcomes that should be considered for future system expansion are:

Failure: Overdue Account Block:

- Scenario and Cause: A member attempts to borrow a new book despite having one or more items past their due date. The cause is a library policy that suspends borrowing privileges until overdue items are returned or fines are paid.
- Handling: The Member class would need an additional state (e.g., a boolean isBlocked or a method hasOverdueItems()). The BorrowingService.borrowBook() method would add a new failure check immediately after the borrowing limit

check: if (member.hasOverdueItems()) return new BorrowingBookResult(false, "Borrowing failed: Account blocked due to overdue items.");
  - Impact: This primarily impacts the BorrowingService and the Member class.

Failure/Special Success: Item Damaged on Return:
  - Scenario and Cause: A member returns a physical item that library staff note has significant damage. The cause is an accident or negligence by the patron.
  - Handling: The BorrowingService.returnBook() method would need to be augmented so that, if true, it prevents the book from being made available again (book.setIsAvailable(false)) and instead calls a new BillingService to assess and apply a damage fine to the member's account. The BorrowingBookResult message would confirm the return but notify the fine: return new BorrowingBookResult(true, "Returned successfully, but damage fee of $X.00 applied.");
  - Impact: This requires a new BillingService abstraction, new status fields/methods in the Book interface (to mark as damaged/out-of-circulation), and modifications to the BorrowingService to handle the conditional flow.

Special Success: Reserved Item Immediate Hold:
  - Scenario and Cause: A member returns a book that has been reserved by another patron. Upon successful return, the system must immediately place the item on hold for Patron B and notify them.
  - Handling: After the BorrowingService.returnBook() successfully updates the book's availability and the member's list, it would call a new ReservationService abstraction to check for any active reservations against that specific book title. If a reservation is found, the book is immediately marked as "On Hold," and the success message informs staff of the mandatory action: return new BorrowingBookResult(true, "Returned successfully. Item immediately placed on hold for Patron B.");
  - Impact: This requires a new ReservationService abstraction to manage the hold queue, and the BorrowingService must integrate this check into its successful return path.

PART 7: Implementing the Singleton Pattern

The Singleton pattern was chosen for the BorrowingService class because borrowing and returning rules (like the 3-book limit) represent a global, centralized business policy that must be consistent across the entire application. Ensuring only one instance exists prevents conflicting borrowing policies, reduces memory overhead by avoiding redundant object creation every time a member borrows a book, and provides a single access point for any future services (e.g., LibrarianController) that need to access the core borrowing logic. This centralization simplifies the management of the global state relevant to transactions.

Two key benefits of using the Singleton here are controlled global access and memory efficiency. However, a potential drawback is reduced testability and flexibility. When using the Singleton, unit testing code that relies on the BorrowingService becomes harder because the test cannot easily substitute a mock service or reset the service's internal state between test cases without affecting other tests. While we mitigated tight coupling by injecting the Singleton instance into the Member's constructor, this pattern could make it difficult to refactor the system later if we ever needed multiple, independent borrowing domains. An alternative approach would be to manage the BorrowingService instance using the Factory Pattern or a Dependency Injection Container, which would still ensure one instance is used but allow for greater flexibility in configuration and testing.

Part 8: Implementing Factory Method Pattern

The Factory Method pattern was used to decouple the object creation logic from the LibrarianController. Before, the controller had explicit knowledge of the concrete classes (new PaperBook(...), new EBook(...)), which meant the controller had to be modified every time a new book type was introduced. The Factory Method pattern fixes this by moving the new operator into dedicated, concrete factories (PaperBookFactory, etc.), adhering to the Open/Closed Principle (OCP). This is clearly shown by the new generic addBook(BookFactory factory, String title) method.

The primary advantage of using this pattern over direct instantiation is improved flexibility and maintainability. When the library introduces a new item type, such as VideoDVD, the only modifications needed are adding VideoDVDFactory and updating the LibrarianController to hold and use that new factory, without changing the logic of how existing books are created or managed. This reduces the risk of introducing bugs into the existing, stable code. However, a potential drawback or limitation of the Factory Method pattern is the proliferation of classes: for every new book type, we now require two new classes, the concrete product (VideoDVD) and the concrete creator (VideoDVDFactory), which increases the overall complexity and class count of the system.