

### **Part 3**

Introducing the Book interface into the application increases the flexibility, maintainability, and scalability in a system as it allows the system to use objects without the need to instantiate them directly. This applies the Dependency Inversion Principle by allowing classes to refer to the interface, which then allows for modification of Book types without needing to modify unnecessary lines of code and potentially breaking the system. This further increases flexibility in a system by, as mentioned before, if the type of book needs to be changed for a specific book, you only need to modify the program in one space, and don't need to worry about it breaking the program as all of the Book types use the same interface and blueprints.

Furthermore, the Open/Closed principle is demonstrated in this system. This is shown by the use of the interface; in which the interface cannot be modified, but it is open for expansion by allowing for the addition of new book types like paperbook, e-book, hardcover, etc.. While working on this lab, I learned what made using interfaces and abstractions so helpful for developing a system, as they allow for much easier flexibility and maintainability over your systems in the long-run.

### **Part 4**

Applying the Dependency Inversion Principle to the book class is excellent because I did not have to modify much code to get the additional implementations of the Book interface up and running by ensuring they depended on the abstract intermediary Book.

To incorporate DVDs and renting rooms, I would make the book class inherit from a new top-level abstract class -> rentable, applying the same benefits to these new implementations as Book does for its own. If the DIP was not applied, it would be significantly more difficult as this new Rentable superclass would have to have more of its own code and implementation, having defaults that might interfere with other classes trying to access its subclasses. This would also mean that book's subclasses would have to only inherit (since no implementations anymore) from Book, applying the same downsides to them as well as possibly not being able to directly call or override methods from Rentable. Overall, this would make it much more difficult to add new book types and create implementations of Rentable, reducing maintainability and flexibility.

### **Part 5**

I took all borrowing aspects out of the member class as SRP requires us to have a class only focusing on one thing. Since the member class is focused on the information of the members the borrowing logic was moved. I chose to put the borrowing functionality into BorrowingService because it interacts with both the book and member classes. The borrowingService involves both members and books and so it is the best place to have this code. An issue that I ran into was that it broke existing behavior so I had to update some of the files (mainly the test ones) This is because too many files relied on the member class handling the borrowing function and created their own instances of members

### **Part 6**

1. These special cases do not need to be reflected to the members. If a book is damaged or stolen, it is simply unavailable, and we can just set its availability to false if so. Same with members we want to expunge. We just remove them from the list since removal due to death, cancellation and expulsion are functionally the same.

2.

A.

Returning the Wrong Book

- Caused by the user returning the improper book (A book that doesn't exist in the library)
- This could be handled simply by checking if the book the member is if the book existed in the list of books taken out by the member.
- It does impact the borrow class and means that since we have gone with a more coupled and less streamlined design, borrowing will have to add an extra catch case instead of this scenario being unable to happen.

B. Downtime

- The library needs to shut down temporarily to perform maintenance
- Just shut down the library system refuse to interact with and automatically calling any members who attempt to return a book's 'borrowbook' method and refusing to allow borrowing books. Extend any deadlines until past the maintenance, not allowing any interaction.
- It will affected all the classes including borrowingservice, but as borrowingservice will be unable to be called not much will change.

C. A book is stolen

- A book has been stolen from the library.
- Delete the book object.
- N/A.

## Part 7

The singleton pattern is used in the BorrowingService class to ensure that the system has one main class for handling all borrowing and returning operations.

Using a singleton provides several benefits in our lab 6, it proved maintainability, and if we want, we can change the rules of borrowing. Unnecessary creation of multiple objects, which can consume more memory and lead to duplicated logic

A drawback of the singleton pattern is that it makes changes globally, which can make tests more difficult

The alternative approach that can be used is to use the same Singleton pattern, but also use the enum. This makes the whole code much safer and it would prevent issues related to duplication, and will guarantee a single instance

## **Part 8**

The Factory Method Pattern is used to improve flexibility because of its inherent extensibility, to reduce dependency on concrete classes, and decouple the creation of classes from their use. This gives us the advantage of being able to make last-minute changes to the inputs of Books before creation, and this also means that in the future, we would only need to add a handler in the librariancontroller class, a new factory and a new concrete implementation rather than having to modify the whole code... But, this isn't necessarily the case in our example. By adding an extra step to the creation of book objects where we still need to add another method to the librariancontroller and library, meaning that this overcomplicates our code. Thus making it harder to navigate and maintain.