COSC236 Lab 5

Introduction

In this assignment, you will work with legacy code, a common scenario for developers entering the workforce. Maintaining and improving existing codebases— often written by others—is a key part of software development. You'll gain practical experience refactoring and extending code without changing its core functionality, which is a valuable skill in the industry.

This lab focuses on enhancing a Library Management System by applying design patterns and solid software engineering principles. You will use patterns like Factory Method and Singleton to improve the system's design, making it more maintainable and scalable.

A critical part of this lab—and software development in general—is testing and test case maintenance. Writing tests ensures your code behaves as expected and helps catch bugs early. It also allows you to refactor code with confidence, knowing the functionality will be verified. As the codebase evolves, maintaining your test cases to keep them aligned with the system is crucial for ensuring long-term stability and quality. In this lab, you will gain hands-on experience writing, running, and maintaining tests—skills essential for modern software development and building robust, scalable systems.

The main objectives of this assignment are:

Refactoring the Library System: You will be working with a partially implemented library management system, where you'll refactor and extend the codebase by applying design patterns such as the Factory Method and Singleton patterns to manage book creation and borrowing operations. This will help you better understand the practical application of design patterns in real-world systems.

Applying Design Patterns: Throughout the project, you will apply core design patterns to solve real- world software engineering problems. The Factory

Method will help create different types of Book objects in a flexible way, and the Singleton pattern will centralize borrowing logic and ensure a single instance is used globally. These patterns will improve the system's modularity, flexibility, and maintainability.

Adhering to SOLID and GRASP Principles: You will also apply important design principles such as SOLID and GRASP to ensure the system's design is robust and follows best practices:

Testing and Debugging: You will ensure that the system works as expected by running provided tests and ensuring that they pass without errors. Additionally, you will write tests for the new functionality you implement to ensure robustness and reliability.

Collaborating with Your Team: This project emphasizes collaboration, and you will be working in teams. You'll set up a GitHub repository, manage version control, and collaborate with team members to complete the project.

By the end of this lab, you will have gained practical experience in applying design patterns, SOLID principles, and GRASP principles, all of which are essential skills in software development. You will also become familiar with refactoring legacy code, implementing testing practices, and collaborating effectively in a version-controlled environment to build scalable and maintainable systems.

Part 0. Setup

Download the Project Files:

- Download the `lab5.zip` file from the Moodle page.

Set Up the Project:

- Create a new package named `lab5` in Eclipse.
- Import the contents of `lab5.zip` into the `lab5` package in your Eclipse project.

Verify the Application:

- Ensure that the `lab5.LibraryApp` class can be executed without issues.

Verify the Test Suite:

- Confirm that the `lab5.tests.AllTests` suite runs successfully and without errors.
- You may need to add JUnit 5 libraries to the classpath. The easiest way to do this is to create a JUnit test from Eclipse. Right-click on any of the .java files under lab5, for example, "Book.java", then in the pop-up menu select New-> JUnit Test Case. At the prompt, select the radio button "New JUnit Jupiter case" and click "Finish." Then "Perform the following action:" "Add the JUnit 5 library to the build path" You can remove the automatically created JUnit file "BookTest.java". An alternative for Eclipse is to add the following string to the `.classpath` file

  ```
  <classpathentry kind="con" path="org.eclipse.jdt.junit.JUNIT_CONTAINER/5"/>
  ```

Set Up Version Control:

- Create a new GitHub repository for your project.
- Invite your team members to collaborate by adding them as contributors.

Part 1. Sequence diagrams

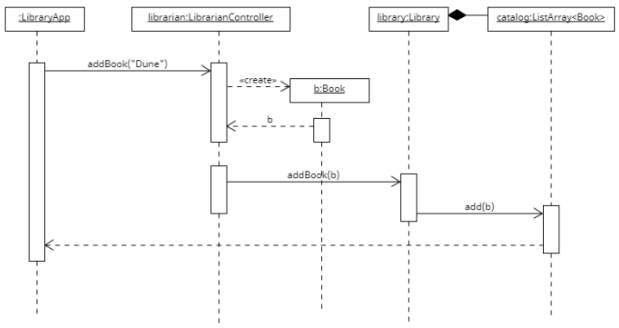
At the top of the `main()` function of `lab5.LibraryApp` class you will find following code:

```
addBook("Dune"); // Adding one book
// Create three sequence diagrams for each of these method calls
addMember("Alice");              // 1. Add a member
borrowBook("Dune", "Alice");     // 2. Borrow a book
returnBook("Dune", "Alice");     // 3. Return book
```

In this section of the assignment, you are required to create three sequence diagrams for the following function calls: `addMember("Alice")`, `borrowBook("Dune", "Alice")`, and `returnBook("Dune", "Alice")`. An example for `addBook("Dune")` is provided below. You may use any diagramming tool that supports sequence diagram fragments, especially to capture any conditional logic within these functions.

Hint: To trace the call stack of a method in Eclipse, press **Ctrl + Right-click** on the method's name.

A sequence diagram for `addBook("Dune")` call may look like this:



Submission: Please include the following three sequence diagrams in your submission:

- `addMember("Alice")` – Illustrating the steps involved when a new member, "Alice", is added to the system.
- `borrowBook("Dune", "Alice")` – Depicting the flow of operations when "Alice" attempts to borrow the book "Dune".
- `returnBook("Dune", "Alice")` – Showing the process when "Alice" returns the book "Dune" to the library.

Ensure that each diagram captures the interactions between relevant objects and incorporates any conditional logic where necessary.

Part 2. Rename the `Book` class to `PaperBook`

To practice refactoring and ensure that changes to the code are applied consistently across the project, we will begin by renaming the `Book` class to `PaperBook`. This also will minimize changes to the existing code as we continue working on this assignment.

Minimize Errors and Inconsistencies: When renaming a class, it's important to ensure that all references to that class—whether in other classes, methods, or test cases—are updated automatically. This helps prevent errors that could arise from missing or incorrect references.

Use IDE Refactoring Tools: Instead of manually searching for and updating all occurrences of the `Book` class, we'll use Eclipse's refactoring functionality to make these changes efficiently and safely.

Steps to Rename:

- Right-click on the `Book` class in the IDE.
- Choose the "Refactor" option from the context menu.
- Select "Rename" and change the name to PaperBook.
- Eclipse will automatically identify and update all references to Book throughout the codebase, including related classes and test cases.
- Verify correctness of this operation by running `LibraryApp.java` class and test suite `AllTests.java`.

Benefit of IDE Refactoring: This method allows you to rename the class and all its references in one step, reducing the risk of introducing bugs and making the codebase easier to maintain.

Part 3. Dependency Inversion

The `Member` and `PaperBook` classes are tightly coupled, meaning that the `Member` class directly depends on and manipulates `PaperBook` objects. In the current design, operations like borrowing and returning books are tightly bound to the `PaperBook` class, as they directly access and modify the `PaperBook` properties through getter and setter methods.

```
// In the Member class ...
public void borrowBook(PaperBook book) {
      if (book != null && book.getIsAvailable() == true) {
            borrowedBooks.add(book);
            book.setIsAvailable(false);
        }
    }
public void returnBook(PaperBook book) {
          if (book != null) {
            borrowedBooks.remove(book);
            book.setIsAvailable(true);
} }
```

This tight coupling creates a few issues:

Reduced Flexibility: The `Member` class can only work with `PaperBook` objects.

If we want to add other types of items (such as `EBook`, `AudioBook`, etc.), the `Member` class would need to be modified, violating the **Open/Closed Principle (OCP)**.

Difficulty in Maintenance: Changes to the `PaperBook` class (for example, adding new properties or changing the way a book is borrowed) would require changes to the `Member` class, leading to more maintenance overhead and potential for bugs.

Single Responsibility Principle Violation: The `Member` class is handling responsibilities related to both maintaining information about the member and performing operations related to books (borrowing/returning). This violates the **Single Responsibility Principle (SRP)**, which suggests that a class should only have one reason to change.

This design also violates the **Dependency Inversion Principle (DIP)**, which states that high-level modules (like the `Member` class) should not depend on low-level modules (like the `PaperBook` class). Instead, both should depend on abstractions.

In this part of the assignment you need to refactor the code and break this dependency by introducing an `abstraction`. This will also demonstrate the power of the *GRAPS* principle of **Indirection** . The principle of **Indirection** suggests introducing a mediator or intermediary between two classes to reduce direct dependencies, thereby enhancing flexibility. By adding an abstraction, we allow the `Member` class to remain decoupled from the low-level `PaperBook` class while still enabling the system to function smoothly.

Below are the steps to introduce an abstraction, then make the class `Member` depend on this abstraction, and the class `PaperBook` implement it.

Tasks:

1. Create a Java interface `Book` to model the borrow-return behavior of the book that can be rented. Include public methods `toString()`, `setIsAvailable(boolean isAvailable)`, `getIsAvailable()`, `getTitle()`.
2. Make the class `PaperBook` implement this interface.
3. Make `Library` and `Member` methods depend on the `Book` interface rather than on the concrete class
4. Validate your changes to make sure they didn't break the application.
5. Modify the tests in the test suite `AllTests.java` to validate correctness of the code.
6. Create new tests as needed and include them into the test suite (add their class names to the `AllTests.java`)

Submission. Answer the following questions:

1. Why did you introduce the `Book` interface, and how does this relate to the Dependency Inversion Principle?
2. How does this design improve the flexibility of the system?
3. Can you explain how your changes support the Open/Closed Principle?
4. What did you learn about the benefits of using abstractions and interfaces in this example?

Part 4. Other book types

In this section, we will extend the functionality of the current `PaperBook` class by introducing new types of books. To do this efficiently and maintainably, we will use a common interface `Book` created in the previous section.

Your Task:

Create New Book Types: You will implement new classes such as `Ebook`, `AudioBook`, or `RareBook` that all conform to the `Book` interface. Each of these classes will have its own unique properties and methods, but they will all adhere to the structure defined by the `Book` interface.

Refactor Existing Code: As you introduce these new book types, you will need to update the existing code and test cases to accommodate the new classes. The goal is to ensure that all types of books can be used interchangeably, thanks to their shared interface.

Update Test Cases: After implementing the new types of books, don't forget to update or add test cases that verify each new book type works as expected.

Tasks:

1. Implement EBook Class: Create a class `EBook` that implements the `Book` interface and represents an eBook available at the library. An `EBook` can be rented and returned by any object of the `Member` class.
2. Implement AudioBook Class: Similarly, create a class `AudioBook` that implements the `Book` interface and models an audio book available at the library. This class should also allow renting and returning by `Member` objects.
3. Update `LibrarianController` Methods: Rename the existing `addBook()` method in the `LibrarianController` class to `addPaperBook()` and add two new methods: `addEBook()` and `addAudioBook()`, to handle the addition of `EBook` and `AudioBook` objects, respectively.
4. Update `LibraryApp` Method: Modify the method `addBook(String title)` in the `LibraryApp` class so it can handle adding different types of books (i.e., paper books, eBooks, and audio books). This will allow you to test the addition of various types of books:

```
// TODO add to LibraryApp.java ...
private static void addBook(String title) {
    librarian.addAudioBook(title); // To test additions of AudioBook
}
private static void addBook(String title) {
    librarian.addEBook(title); // To test additions of eBook
}
```

5. Test Cases for Rental and Return: You need to write new or modify existing test cases to verify the rental and return of `EBook` and `AudioBook` objects. Add these test cases to the test suite (`AllTests.java`) and ensure that all tests pass before moving on to the next part of the assignment.
6. Test Addition and Removal of Books: Test the addition and removal of books from the library system. Modify the existing code as needed to ensure that books can be added and removed properly from the system.

Submission:

1. Document any changes made to the code to accommodate for this updated functionality.
2. Reflect on the Dependency Inversion Principle (DIP). Write a brief summary explaining the benefits of applying the **DIP** to the extension of the book borrowing functionality, particularly in terms of expanding the system to handle other types of rentals, such as rooms, laptops, and video DVDs. In partricular, answer the following questions:
    - What changes would you need to make in the current design to support renting items such as rooms, laptops, and video DVDs?

- Provide an explanation of how you would modify the test cases to test these new rental types.
- How would you implement this extended functionality if the **DIP** was not applied? Discuss the impact this would have on maintainability and flexibility.

Part 5. BorrowingServices

Applying the **Dependency Inversion Principle (DIP)** has reduced the coupling between the `Member` class and the specific classes representing books. The `Member` class is now only dependent on the `Book` interface, which provides more flexibility.

However, there is still room for improvement. While the `Member` class adheres to the **Knowledge Expert** principle by managing information about library patrons, it also handles the responsibility of borrowing books. This violates the **Single Responsibility Principle (SRP)**. As a result, the `Member` class now has two reasons to change: one for updates to the attributes of library patrons, and another if the `Rentable` interface changes.

To fully comply with **SRP**, these responsibilities should be separated. The `Member` class will continue to focus on its role as a **Knowledge Expert**, while the borrowing functionality will be delegated to a new abstraction that will be created in this part of the assignment.

Tasks:

1. Create a class called `BorrowingService` that will manage borrowing and returning operations in the library system. This class will implement the `BorrowingServiceAPI` interface, which defines methods for borrowing and returning books.

```
public interface BorrowingServiceAPI {
    // Two methods to manage Member's books public boolean borrowBook(Member
   member, Book book); public boolean returnBook(Member member, Book book);
}
public class BorrowingService implements BorrowingServiceAPI {
    // TODO: Implement methods for borrowing
    // and returning Books
}
```

Note: It is a good practice for the `BorrowingService` to manage the relationship between the `Member` and the `Book` items that are borrowed. This is because the responsibility of managing borrowed items logically fits within the `BorrowingService`, as it handles the borrowing and returning operations. This keeps the `Member` class focused on encapsulating the information about the member, while the `BorrowingService` focuses on managing borrowing operations.

```
public class BorrowingService implements BorrowingServiceAPI {
    @Override
    public boolean borrowBook(Member member, Book book) {
        // Here you can implement logic to check if the book is available to
        // borrow and if the member can borrow it
        //(e.g., item limit, member status).
        System.out.println("Borrowing book: " + book);
        return true; // Return true for success
    }
    @Override
    public boolean returnBook(Member member, Book book) {
        // Implement logic to handle returning a book
        System.out.println("Returning book: " + book);
        return true; // Return true for success
    }
}
```

2. Integrate the `BorrowingService` into the `Member` class. Refactor the `Member` class so that it instantiates a `BorrowingService` object when borrowing or returning a `Book`, and delegate these responsibilities to the `BorrowingService` instance instead of directly managing `Book` objects. This will decouple the borrowing logic and adhere to better design principles.

```
public class Member {
   private String name; private ArrayList<Book> borrowedBooks;
   // Constructors, getters/setters etc.
   ...
   public void borrowBook(Book book) {
       BorrowingService borrowingService = new BorrowingService();
       boolean success = borrowingService.borrowBook(this, book); if(success)
       {
         // print something
       } else {
         // print something else
       }
   }
   public void returnBook(Book book) {
       BorrowingService borrowingService = new BorrowingService();
       boolean success = borrowingService.returnBook(this, book); if(success)
       {
         // print something
       } else {
         // print something else
       }
   }
   // other things ....
}
```

This solution follows the **Single Responsibility Principle (SRP)**: The `Member` class should not be responsible for managing the borrowed items because it's focusing on the member's data. The service should handle the mechanics of borrowing and returning, ensuring that each class has a single responsibility.

3. Add checks to prevent borrowing a book if the `Member` has already borrowed it, and to prevent returning a book if it is not in the Member's list of borrowed books.

4. Create test cases to validate the functionality of the `BorrowingService` and add them to the test suite (`AllTests.java`).

Submission:

Write a short reflection (1-2 paragraphs) on the design decisions you made during the refactoring process. Specifically, discuss:

- How you ensured that the `Member` class adheres to the **Single Responsibility Principle** after the changes.

- Why would you chose the particular abstraction or class for handling borrowing functionality? Can you suggest other solutions?
- Any challenges or trade-offs you encountered while applying the **Dependency Inversion Principle** and **Single Responsibility Principle.**

Part 6. Refactoring the BorrowingServiceAPI

In a real library management system, there are several scenarios in which borrowing and returning a book can fail. For the purpose of this assessment, we will consider the following failure conditions:

- **Borrowing Failures:**
- The book is already borrowed by the same member.
- The book is currently borrowed by another member and is unavailable.
- The member has exceeded their borrowing limit, see below.
- **Returning Failures:**
- The member has not borrowed the book.
- The book has already been returned.

The borrowing limit will be managed by the `BorrowingService`, which will ensure that no member can borrow more than **3 books** at any given time.

To provide more detailed information about borrowing and returning operations, we will refactor the `BorrowingServiceAPI` so that the methods return a `BorrowingBookResult` object.

This object will allow us to provide more descriptive feedback, including a message that explains the success or failure of a borrowing or returning operation by encapsulating both the status (isSuccess) and a message (borrowingMessage) that describes the outcome of the operation.

Tasks:

1. Create the `BorrowingBookResult` class: This class will hold two attributes `isSuccess` and `borrowingMessage` and will have appropriate getters and setters for these fields.

2. Update the `BorrowingServiceAPI` interface: The `borrowBook` and `returnBook` methods will now return a `BorrowingBookResult` instead of a `boolean`.

3. Update the `BorrowingService` class: Implement the `borrowBook` and `returnBook` methods to return a `BorrowingBookResult` with both a success flag and a message indicating the result.

```
class BorrowingBookResult {
    private boolean isSuccess; private String borrowingMessage;
    // Constructor
    public BorrowingBookResult(boolean isSuccess,
                String borrowingMessage) { this.isSuccess = isSuccess;
        this.borrowingMessage = borrowingMessage;
    }
    // Getters and setters
    // TODO implement getters and setters
}
```

4. Replace the return type of methods defined in `BorrowingServiceAPI` from `boolean` to `BorrowingBookResult` type and make necessary modifications in the `Member` class. For instance, the `borrowBook(book: Book): void` method of the `Member` class reduces to the following method call:

```
...
 // Borrow the book using the service
BorrowingBookResult borrowingResult =
        borrowingService.borrowBook(this, book);
System.out.println("Success: " + borrowingResult.isSuccess +
        ": " + borrowingResult.getMessage());
...
```

5. Create detailed messages for all possible outcomes of the borrowing and returning processes. Update corresponding test cases for `BorrowingService` in the test suite to verify these scenarios.

Submission:

1. Reflect on different situations that could arise in a library management system when borrowing or returning books. Consider additional failure conditions or edge cases that might not have been covered in the initial specification. For example, what happens if the system is temporarily unavailable, or if a book is damaged or lost? Are there special cases related to library membership status or book availability?
2. Identify at least 3 new possible outcomes or failure conditions that could occur during the borrowing or returning process. These can involve failures, exceptions, or even success scenarios that were not covered initially.
3. For each new outcome, briefly explain:
    - The scenario and its cause (e.g., "The book is marked as lost or damaged").
    - How this condition could be handled or prevented in the system.
    - Whether it impacts the `BorrowingService`, `Member` class, `Book` classes, or requires a new class/abstraction.

Example Outcomes (for inspiration):

- Scenario 1: The book is reserved for another member (but the system still allows the current member to borrow it).
- Scenario 2: A member with overdue books tries to borrow a new book.
- Scenario 3: A member returns a book with missing pages or significant damage.
- Scenario 4: The library system experiences downtime, preventing borrowing or returning actions from being processed.

Part 7. Implementing the Singleton pattern

In this section, you will refactor the current `Member` design to use a `BorrowingService` class as a Singleton. By applying the Singleton pattern, the system will ensure that there is only one instance of the `BorrowingService` used throughout the application, promoting a centralized management of borrowing operations.

This will also help reduce redundancy and prevent the creation of multiple instances of the `BorrowingService`, which should handle borrowing and returning logic globally.

Tasks:

1. Write a Singleton implementation: You can use any thread-safe version of the Singleton pattern. Here is the partial implementation.

```
public class BorrowingService implements BorrowingServiceAPI {
    private static BorrowingService instance; // private member
    private int borrowingLimit; // to restrict the count of borrowed books
    private BorrowingService() { // private constructor
            borrowingLimit = 3;
    }
    public static BorrowingService getInstance( ) {
        // TODO
        // Implement Singleton pattern. return instance;
    }
    // The rest of code ...
    @Override
    public BorrowingBookResult borrowBook(Member member, Book book) { ... }
    @Override
    public BorrowingBookResult returnBook(Member member, Book book) { ... }
}
```

2. Write a test case to validate the Singleton functionality: Create two instances of the Singleton and check if they refer to the same object.

```
class TestSingleton {

    @Test void TestSingleton() {

      BorrowingService service1 = BorrowingService.getInstance();
      BorrowingService service2 = BorrowingService.getInstance();
      assertEquals(service1, service2,"Two Singleton instances detected");

    }

}
```

3. Apply the Dependency Injection Principle to inject the `BorrowingService` into the `Member` class, ensuring that the class no longer directly creates instances of the `BorrowingService` but instead relies on the Singleton instance. You can do this by passing the `BorrowingService` instance into the Member class constructor (constructor injection) or via a setter method (setter injection).

Note: Since the `Member` objects are created by the `LibrarainController`, make the `BorrowingService` a part of the `LibrarainController` by using composition. Then, pass the instance of `BorrowingService` to each `Member` through the constructor.

```
// LibrarianController class
public class LibrarainController {
    private Library library; // Library dependency
    private BorrowingService borrowingService; // Singleton
    public LibrarainController() {
        // The LibraranController holds
        // the single instance of BorrowingService
        this.borrowingService = BorrowingService.getInstance();
   }
```

```
    ...

    // Method to add a Member
    public void addMember(String name) {
         // Inject the Singleton instance into the Member
         library.addMember(new Member(name, borrowingService)); }
        // Other things ...
    }
    ...
    // Member class
    public class Member {
        private String name;
        private List<Rentable> borrowedItems;
        private BorrowingService borrowingService; // Injected via constructor
        // Constructor now takes the Singleton BorrowingService instance
         public Member(String name, BorrowingService service) {
          this.name = name;
          this.borrowedItems = new ArrayList<>();
          this.borrowingService = service; // Store the Singleton instance
        }
     // TODO Change Member::borrowBook(Book book) and
     // Member::borrowBook(Book book) to use the borrowingService member ...
}
```

Rather than creating a new instance of `BorrowingService` each time a `Member` borrows or returns an Item, the Singleton instance passed from the `LibrarianController` to the `Member` ensures that all `Member` instances share the same `BorrowingService`. This maintains the benefits of the Singleton pattern while also leveraging composition for managing dependencies.

4. Check the correctness of the `Member` constructor. Add this test cases to the test suite.

```
class TestMembersBorrowingService {
    BorrowingService service = BorrowingService.getInstance();
    @Test
    void TestMemberServices() {
        Member member1 = new Member("Member 1", service); Member member2 = new
        Member("Member 2", service);
        assertEquals(member1.getBorrowingService(),
        member2.getBorrowingService(),
                        "Members have two different borrowing services");
                        }
}
```

5. Fix any errors in the remaining test cases and ensure that the entire test suite passes successfully. Every test case that uses `Member` objects needs to have an instance of `BorrowingService` ready to be injected into the `Member` class via its constructor.

For example,

```
class TestAddRemoveBooks {
    private Library library;
    private BorrowingService service = BorrowingService.getInstance();
    ...
    Member member1 = new Member("Alice",service);
    Member member2 = new Member("Bob",service); ...
}
```

Submission:

Provide a short reflection (1-2 paragraphs) discussing the use of the Singleton pattern in the `BorrowingService` class.

Explain the Role of the Singleton Pattern:

- Describe why the Singleton pattern was chosen for the BorrowingService class in this context.
- Discuss how using a Singleton helps with the centralization of borrowing and returning operations, and why it makes sense to ensure that only one instance of the BorrowingService is used throughout the system.

Evaluate the Benefits and Drawbacks:

- In your own words, outline at least two benefits of using the Singleton pattern in this scenario. For example, consider how it affects memory usage, maintainability, and the management of global state.
- Discuss at least one potential drawback or limitation of using the Singleton pattern. For instance, think about issues related to testing, global state, or flexibility in scaling the system.

Alternative Design Choices:

- Briefly mention any alternative design patterns (or approaches) that could be used to manage the borrowing logic, other than the Singleton. What would be the impact of using a different approach on the overall design of the system?

Part 8. Implementing the Factory Method pattern

In this part, you will apply the **Factory Method** pattern to create `Book` objects in a library system. The **Factory Method** pattern is a creational pattern that provides an interface for creating objects in a superclass but allows subclasses to alter the type of objects that will be created. By using this pattern, you will decouple the process of creating `Book` objects from the actual implementation of the `Book` class, and you will allow the flexibility to create different types of `Book` objects.

Tasks:

1. Define an abstract class `BookFactory` that declares a factory method `createBook(String title)`. This method will be responsible for creating `Book` objects. The method should return a `Book` instance.

2. Create concrete classes that extend `BookFactory`. Each class should implement the `createBook(String name)` method to instantiate the appropriate `LibraryBook` object (e.g., `PaperBook`, `EBook`, `AudioBook`).

3. Update the `LibrarianController` class to include three members, one for each type of `BookFactory`. Initialize these members in the constructor.

4. Replace the direct creation of `Book` objects using the **new** operator in methods `addPaperBook()`, `addEBook()` and `addAudioBook()` with the use of the appropriate `BookFactory` instances.

5. Add the following method to the `LibrarianController`

```
public void addBook(BookFactory factory, String title) {
      library.addBook(factory.createBook(title)); // Book type depends on
                                                  // the factory passed in
}
```

6. Write test cases to verify that the Factory Method correctly creates the appropriate type of `Book` objects. Ensure the tests cover the following scenarios: * The factory correctly creates instances of different types of books. * The created objects are of the expected type (i.e., the right subclass of Book). * Include tests for edge cases, such as trying to create unsupported book types.

Submission:

Write a short reflection (1-2 paragraphs) on the use of the **Factory Method** pattern in this scenario:

- Why was the Factory Method pattern used to create `Book` objects, and how does it improve flexibility in the system?
- What are the advantages of using this pattern over directly instantiating `Book` objects in client code?
- Can you think of any potential drawbacks or limitations of using the Factory Method pattern in this case?

Group Submission

Gather all the required results from the Submission sections of this assignment, compile them into the Lab Submission Document, include the URL to your GitHub repository, convert the document to PDF format, and upload it to Moodle before the deadline.