

***Code quality and Unit Test***

**Training Assignments**

|  |  |
| --- | --- |
| Document Code | 25e-BM/HR/HDCV/FSOFT |
| Version | 1.1 |
| Effective Date | 20/11/2022 |

Contents

[Objective: 4](#_Toc13606408)

[Working Environments 4](#_Toc13606409)

[Product Architecture 4](#_Toc13606410)

[Technologies 4](#_Toc13606411)

[Assignment Description 4](#_Toc13606412)

[Exercise 1 4](#_Toc13606413)

|  |  |
| --- | --- |
|  | **CODE: CNU.M.A101**  **TYPE: LONG**  **LOC:**  **DURATION: 180 MINUTES** |

# Technologies

The software product is developed based on:

* Java Core (OOP)
* String, int …
* S.O.L.I.D principles

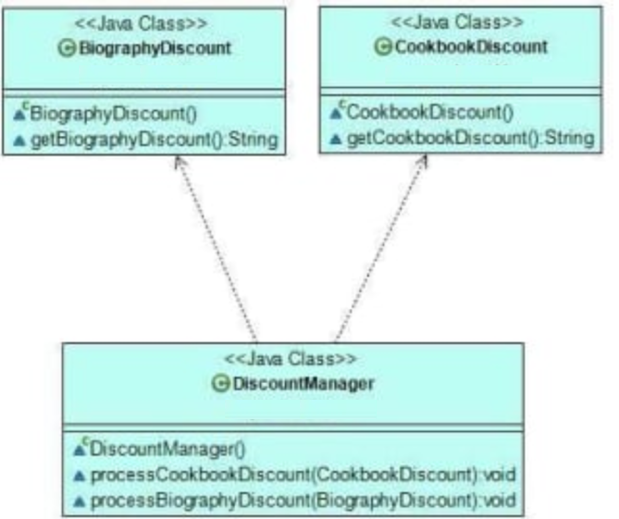
# Assignment Descriptions

Write a Java application for a book store.

1. Each book contain **title**, **author**. Application also provide a method allow user **search** the book in the inventory. To apply the Single Responsibility Principle, what should you do in this in this situation?

(Write your code in package named **edu.fa.solid.srp)**

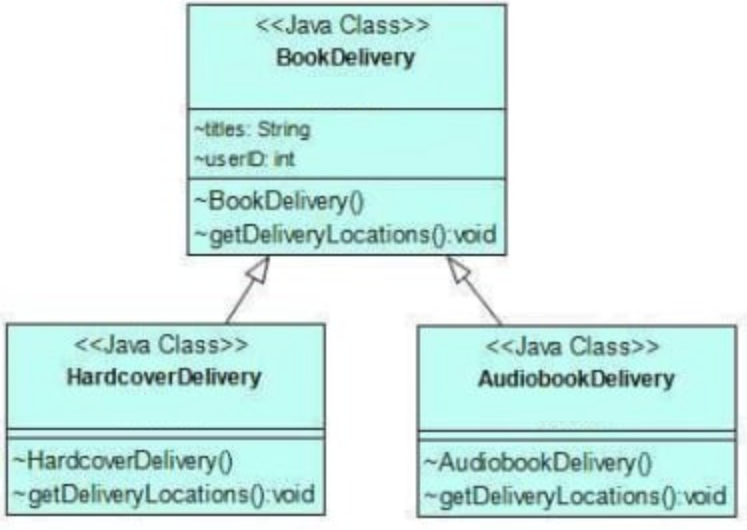
1. Now, book store wants to hand out cookbooks at a discount price before Christmas. Create two separate classes: CookbookDiscount to hold the details of the discount and DiscountManager to apply the discount to the price. After few months, the store wants to hand out every biography with a 50% discount on the subject’s birthday. To add the new feature, create a new BiographyDiscount class and add the new functionality to the DiscountManager class. See UML graph below:



You should refactor code follows the Open/Closed principle, means that in the future, the store can extend our app with other discount types without modifying the existing code base.  
  
khi có thêm discount của loại sách mới thì lại phải thay đổi discountManager

(Write your code in package named **edu.fa.solid.ocp)**

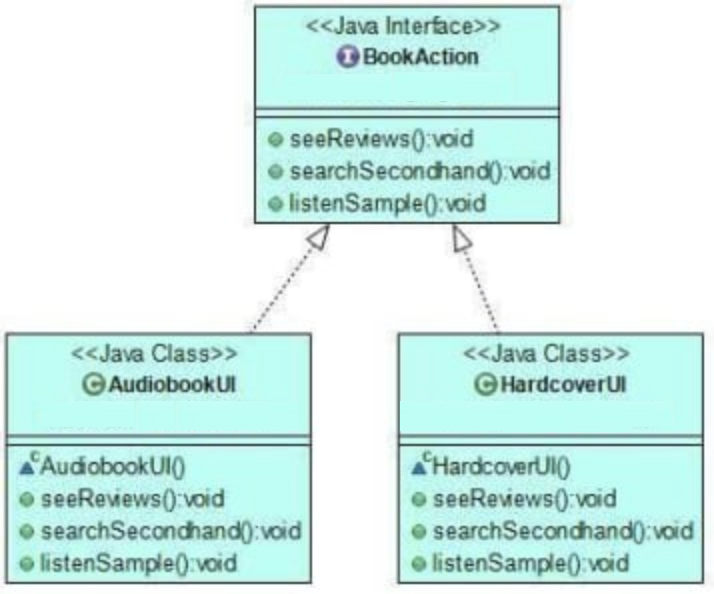
1. Now, the book store wants to add a new delivery functionality to the application. Create a **BookDelivery** class contain **title** and **userId** that informs customers about the number of locations where they can collect their order. However, the store also sells fancy **hardcovers** they only want to deliver to their high street shops. So, we create a new **HardcoverDelivery** subclass that extends **BookDelivery** and overrides the **getDeliveryLocations**() method with its own functionality. Later, the store wants to create delivery functionalities for audiobooks, too. Now, we extend the existing **BookDelivery** class with an **AudiobookDelivery** subclass. But, when we want to override the **getDeliveryLocations**() method, we realize that audiobooks can’t be delivered to physical locations. See UML graph below:



You should refactor code follows the Liskov Substitution Principle, means that you could use any subclass in place of its superclass without breaking the application.

(Write your code in package named **edu.fa.solid.lsp)**

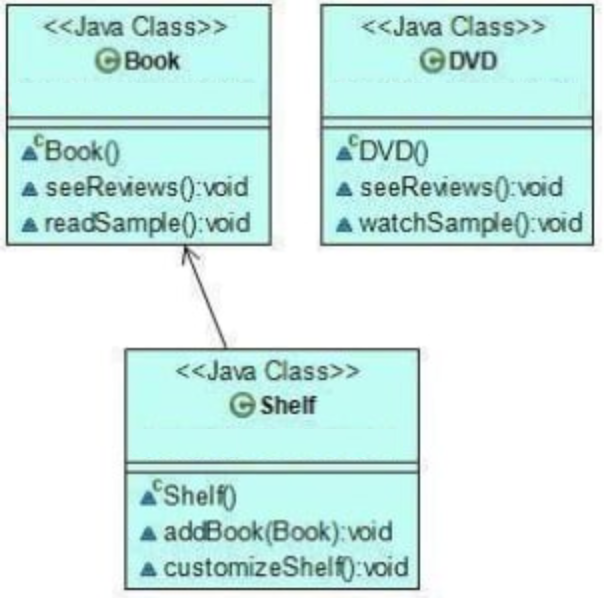
1. Let’s add some user actions to our online bookstore so that customers can interact with the content before making a purchase. To do so, we create an interface called **BookAction** with three methods: **seeReviews**(), **searchSecondHand**(), and **listenSample**(). Then, we create two classes: **HardcoverUI** and an **AudiobookUI** that implement the **BookAction** interface with their own functionalities. Both classes depend on methods they don’t use. Hardcover books can’t be listened to, so the **HardcoverUI** class doesn’t need the **listenSample**() method. Similarly, audiobooks don’t have second-hand copies, so the **AudiobookUI** class doesn’t need it, either. See UML graph below:



You should refactor code follows the Interface Segregation Principle, as neither classes depend on methods they don’t use.

(Write your code in package named **edu.fa.solid.isp)**

1. Now, the book store asked us to build a new feature that enables customers to put their favorite books on a shelf. To implement the new functionality, we create a lower-level **Book** class and a higher-level **Shelf** class. The **Book** class will allow users to see reviews and read a sample of each book they store on their shelves. The **Shelf** class will let them add a book to their shelf and customize the shelf. After few months, the store asks us to enable customers to add DVDs to their shelves, too. To fulfill the demand, we create a new DVD class. See UML graph below:



Now, we should modify the **Shelf** class so that it can accept DVDs, too. However, this would clearly break the Open/Closed Principle.

You should refactor code flow the Dependency Inversion Principle, as in the refactored code, high-level classes don’t depend on low-level classes, either.

(Write your code in package named **edu.fa.solid.dip)**

**-- THE END --**