Task- 1 & 2

1. In Cart.java:

- Leaky Encapsulation: The updateBookPrice() method directly modifies the Book object's price, violating encapsulation

Solution:

---> *delegate the responsibility to the Book class.*

```
public void updateBookPrice(double newPrice) {
   if (book != null) {
       book.setPrice(newPrice);    }
}
```

- Missing Encapsulation: No validation for quantity in addMore() and removeSome() methods

Solution :

```
public void addMore(int more) {
   if (more <= 0) {
       throw new IllegalArgumentException("Quantity to add must be positive");
   }
   quantity += more;}
```

2. In Book.java:

- Missing Encapsulation: All setters modify fields directly without any business rules or validation

Solution :

```
public void setBarcode(String barcode) {
    if (barcode == null || barcode.trim().isEmpty()) {
        throw new IllegalArgumentException("Barcode cannot be empty");
    }
    this.barcode = barcode;
}



public void setPrice(double price) {
    if (price <= 0) {
        throw new IllegalArgumentException("Price must be positive");
    }
    this.price = price;
}

public void setQuantity(int quantity) {
    if (quantity < 0) {
        throw new IllegalArgumentException("Quantity cannot be negative");
    }
    this.quantity = quantity;}
```

3. In User.java:

- Missing Encapsulation: Direct field access through getters/setters without validation

Soultion:

```
public void setEmailId(String emailId) {
    if (!emailId.matches("[.....]+@[......]")) {
        throw new IllegalArgumentException("Invalid email format");
    }
    this
}
```

- Unutilized Abstraction: Commented out retrieveFromHttpServletRequest method shows dead code

Solution :

Remove it

4. In BookService.java:

- Multifaceted Abstraction: The interface mixes book management with email notification (sendEmailNotification)

Solution :

Remove the sendEmailNotificaeion method

- Broken Modularization: Email notification doesn't belong in a book service interface

SOlution :

--> Create another interface

```
public interface NotificationService {
    void sendEmailNotification(User user, String message);
}
```

Task-3

The Solution code is in “ModifiedBookServiceImpl.java” file

Task-4

Eager Initialization:

The eager initialization singleton is implemented in DBUtil.java where the instance is created when the class loads.

The instance is created immediately when the class is loaded, whether it's needed or not. This means memory is taken up even if the instance is never used. It ensures the instance is always available but can be wasteful if it’s rarely or never needed.

Lazy Initialization:

The lazy initialization singleton is implemented in DBConnection.java where the instance is created only when getConnection() is first called.

The instance is created only when it's actually needed. If the instance is never requested, it never takes up memory. This makes it a better choice when memory efficiency matters.

So, **Which One is More Memory-Intensive :**

Eager initialization always consumes memory, even when unused. So , That is more memory intensive .
Lazy initialization waits until it's necessary ; so it is more efficient

Task-5

Used **HashMap** to solve the issue.

The Solution code is in “ModifiedCart.java” file