

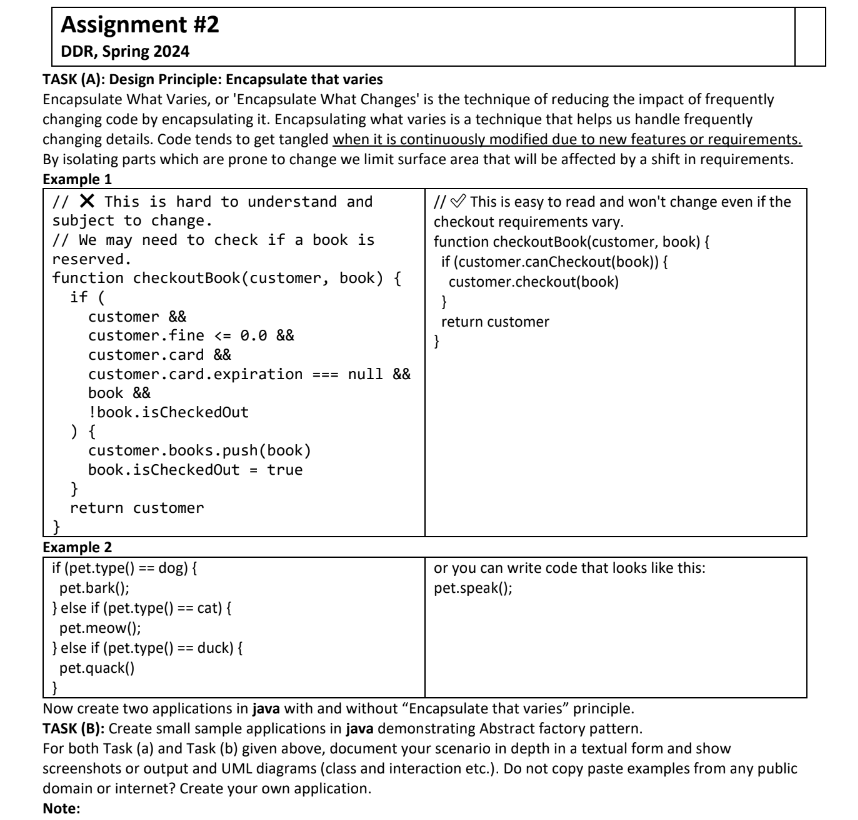
**ASSIGNMENT # 2**

**DESIGN DEFECTS & RESTRUCTURING (BCS-8A)**

**HASSAN ALI**

**K20-1052**

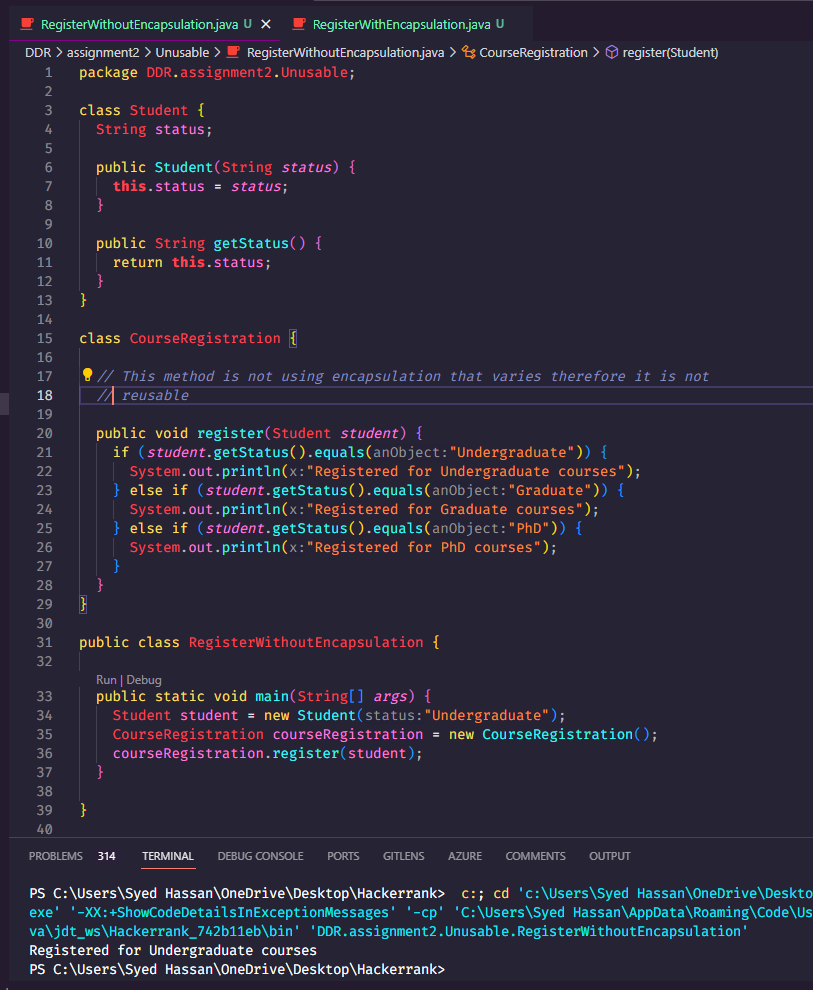
**22nd FEB**



**Task A: Design Principle - Encapsulation that varies**

The Encapsulation That Varies principle suggests that you should encapsulate the parts of the system that are expected to change, protecting the rest of the system from having to know about those changes.

**Example 1**



In this code, the **register** method in the **CourseRegistration** class is not following this principle. The method is directly checking the **status** of the **Student** object and deciding what to do based on that status.

If a new status is added (for example, **"Postgraduate"**), or if the name of an existing status changes, you would have to modify the **register** method. This means that the **register** method is not protected from changes in the **Student** class, violating the Encapsulation That Varies principle.

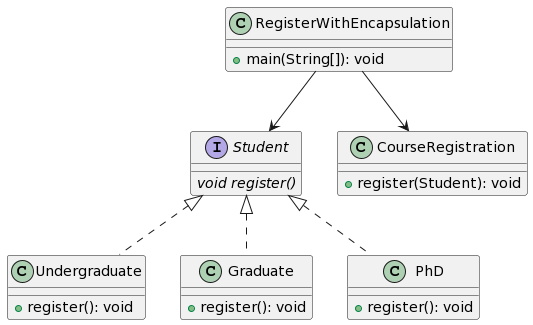
Above code is tangled as each time in the course registration it has to check the status of student, whether it is undergraduate, graduate or Phd. Therefore, this code is not extendable.



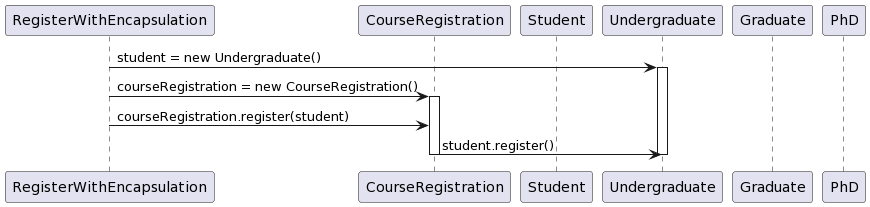
Each of these classes implements the **Student interface** and provides its own implementation of the register method. This means that the register method in the **CourseRegistration** class doesn't need to know about the different types of students or how they register. It simply calls the register method on the **Student object** it receives, and the correct registration process is carried out based on the type of the Student object.

This design makes the code more flexible and easier to maintain. If a **new type of student** is added in the future, you only need to create a new class that implements the Student interface and provides its own register method. You don't need to modify the CourseRegistration class or any of the existing Student classes. This is the essence of the Encapsulation That Varies principle.

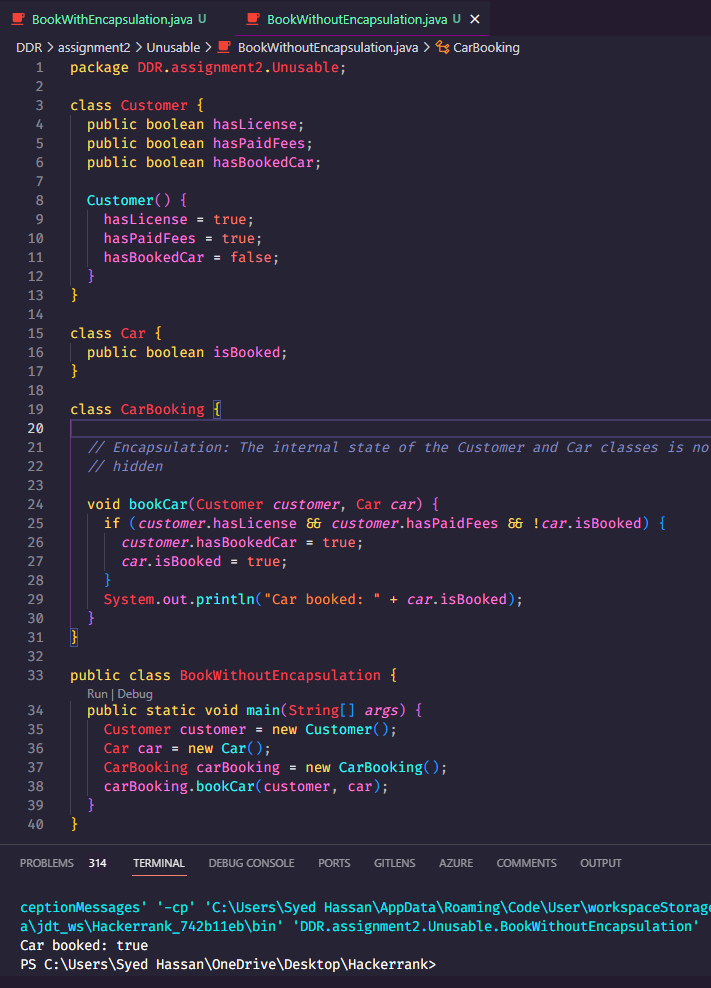
**Class Diagram**

****

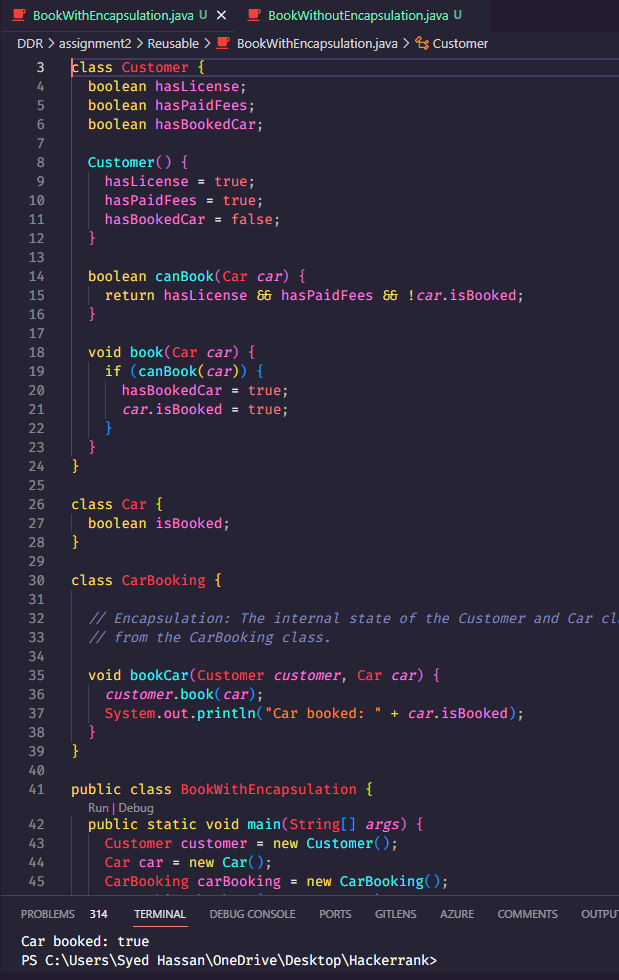
**Interaction Diagram**

****

**Example 2**



Above code deals with the business logic of car booking in the bookcar method, making it very repeatable. Instead, this logic should have been catered in a different method.

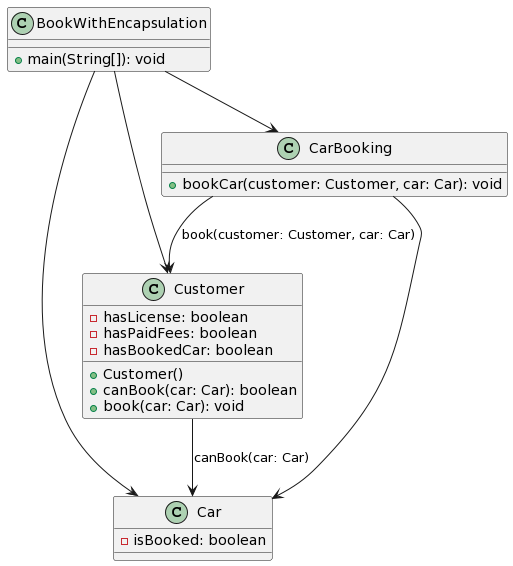


In the Customer class, the fields hasLicense, hasPaidFees, and hasBookedCar are private and can only be accessed or modified through the methods provided within the class, such as canBook() and book(). This ensures that the state of a Customer object can only be changed in controlled ways.

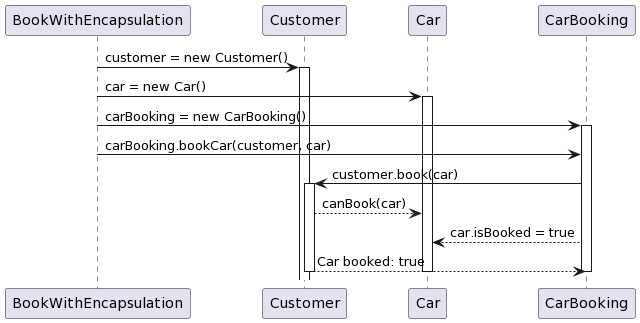
Similarly, in the Car class, the isBooked field is private and can only be accessed or modified through the methods in the Customer class.

The CarBooking class interacts with Customer and Car objects through their public methods, without needing to know the details of their internal state. This is the essence of encapsulation: hiding the internal details of an object and providing a public interface for interacting with that object.

**Class Diagram**

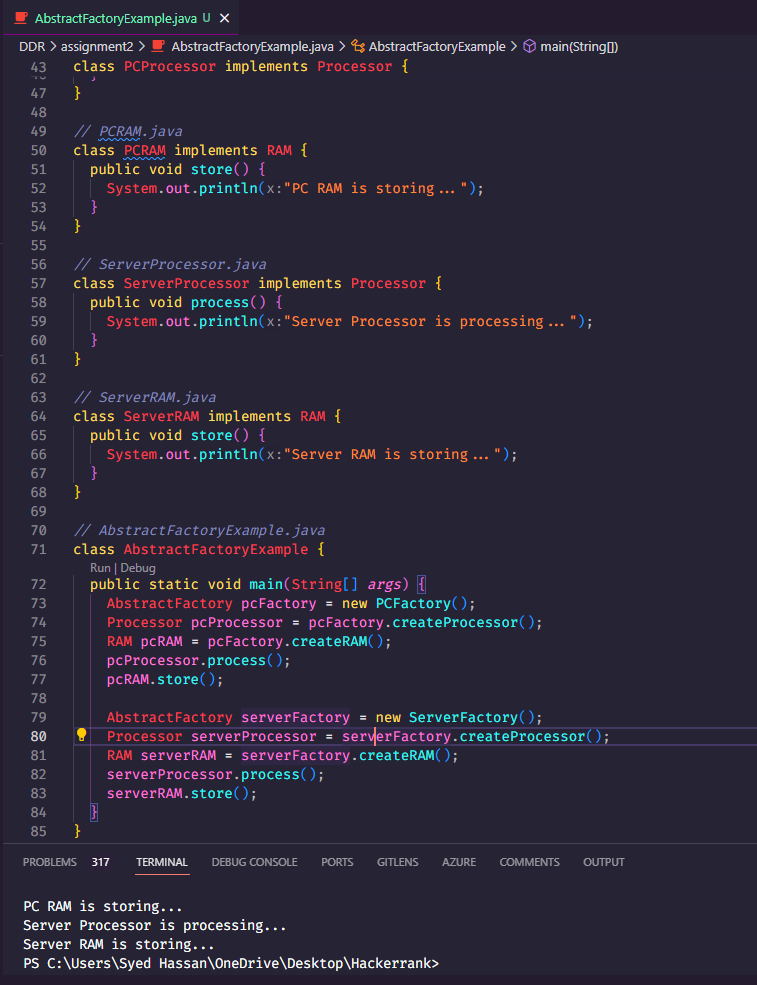
****

**Interaction Diagram**

****

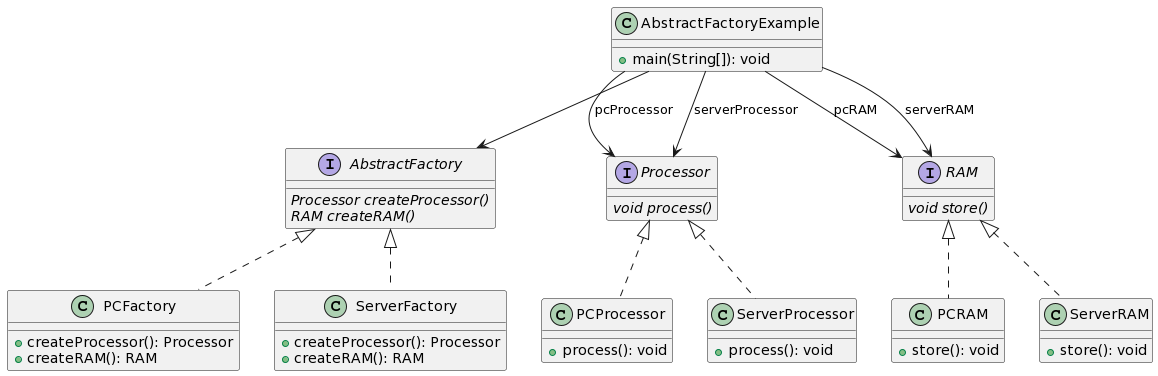
**Task B**



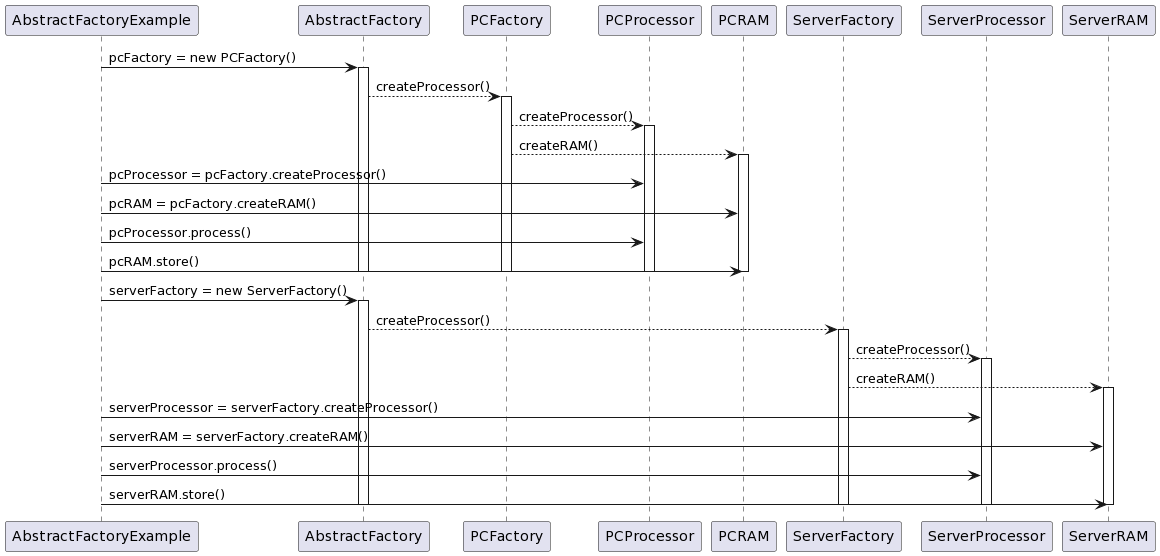


AbstractFactory is the abstract factory, PCFactory and ServerFactory are the concrete factories, Processor and RAM are the abstract products, and PCProcessor, PCRAM, ServerProcessor, ServerRAM are the concrete products. The AbstractFactoryExample class demonstrates how to use the abstract factory to create different types of products.

**Class Diagram**

****

**Interaction Diagram**

****