ASSIGNMENT # 2

DESIGN DEFECTS & RESTRUCTURING (BCS-8A)

HASSAN ALI
K20-1052

22nd FEB

Assignment #2

DDR, Spring 2024

TASK (A): Design Principle: Encapsulate that varies

Encapsulate What Varies, or 'Encapsulate What Changes' is the technique of reducing the impact of frequently changing code by encapsulating it. Encapsulating what varies is a technique that helps us handle frequently changing details. Code tends to get tangled <u>when it is continuously modified due to new features or requirements.</u> By isolating parts which are prone to change we limit surface area that will be affected by a shift in requirements.

Example 1

```
// ✖ This is hard to understand and
subject to change.
// We may need to check if a book is
reserved.
function checkoutBook(customer, book) {
  if (
    customer &&
    customer.fine <= 0.0 &&
    customer.card &&
    customer.card.expiration === null &&
    book &&
    !book.isCheckedOut
  ) {
    customer.books.push(book)
    book.isCheckedOut = true
  }
  return customer
}
```

```
// ✅ This is easy to read and won't change even if the
checkout requirements vary.
function checkoutBook(customer, book) {
  if (customer.canCheckout(book)) {
    customer.checkout(book)
  }
  return customer
}
```

Example 2

```
if (pet.type() == dog) {
  pet.bark();
} else if (pet.type() == cat) {
  pet.meow();
} else if (pet.type() == duck) {
  pet.quack()
}
```

or you can write code that looks like this:

```
pet.speak();
```

Now create two applications in **java** with and without "Encapsulate that varies" principle.

TASK (B): Create small sample applications in **java** demonstrating Abstract factory pattern.

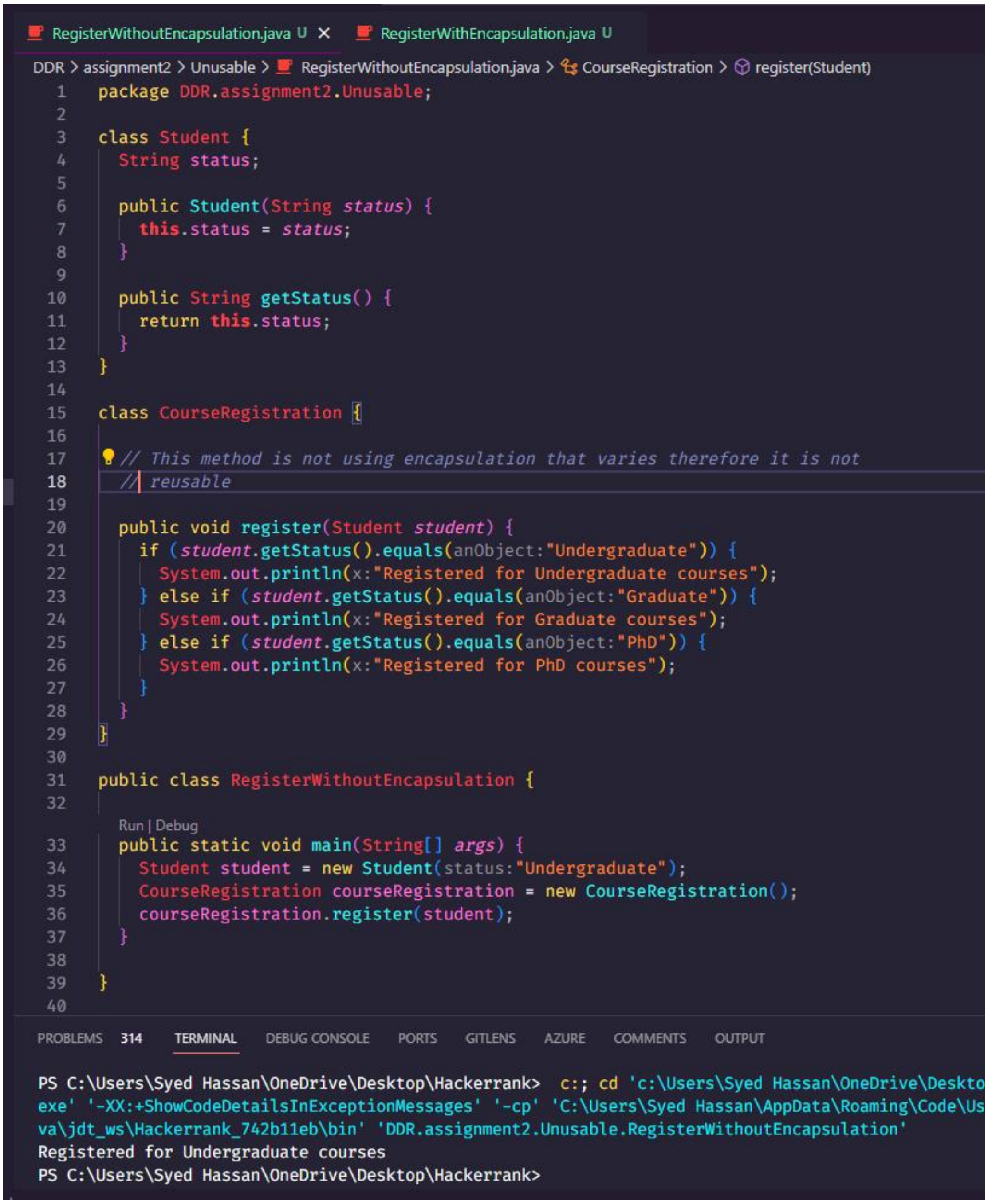
For both Task (a) and Task (b) given above, document your scenario in depth in a textual form and show screenshots or output and UML diagrams (class and interaction etc.). Do not copy paste examples from any public domain or internet? Create your own application.

Note:

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


RegisterWithoutEncapsulation.java U ×
RegisterWithEncapsulation.java U
DDR > assignment2 > Unusable > RegisterWithoutEncapsulation.java > CourseRegistration > register(Student)
Run | Debug
PROBLEMS 314
TERMINAL
DEBUG CONSOLE
PORTS
GITLENS
AZURE
COMMENTS
OUTPUT

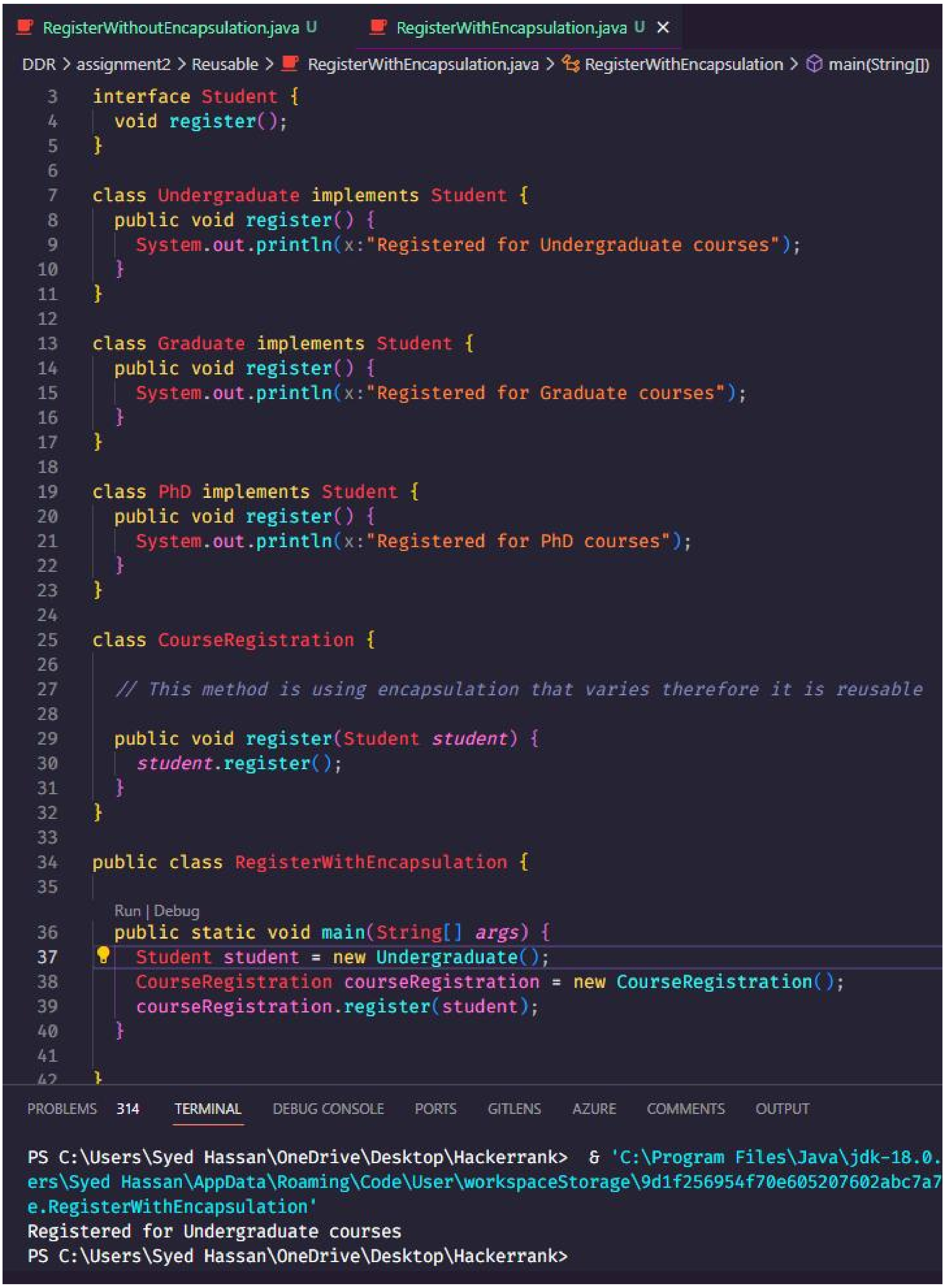
```java
package DDR.assignment2.Unusable;

class Student {
  String status;

  public Student(String status) {
    this.status = status;
  }

  public String getStatus() {
    return this.status;
  }
}

class CourseRegistration {

  // This method is not using encapsulation that varies therefore it is not
  // reusable

  public void register(Student student) {
    if (student.getStatus().equals(anObject:"Undergraduate")) {
      System.out.println(x:"Registered for Undergraduate courses");
    } else if (student.getStatus().equals(anObject:"Graduate")) {
      System.out.println(x:"Registered for Graduate courses");
    } else if (student.getStatus().equals(anObject:"PhD")) {
      System.out.println(x:"Registered for PhD courses");
    }
  }
}

public class RegisterWithoutEncapsulation {

  public static void main(String[] args) {
    Student student = new Student(status:"Undergraduate");
    CourseRegistration courseRegistration = new CourseRegistration();
    courseRegistration.register(student);
  }

}
```

```
PS C:\Users\Syed Hassan\OneDrive\Desktop\Hackerrank>  c:; cd 'c:\Users\Syed Hassan\OneDrive\Deskto
exe' '-XX:+ShowCodeDetailsInExceptionMessages' '-cp' 'C:\Users\Syed Hassan\AppData\Roaming\Code\Us
va\jdt_ws\Hackerrank_742b11eb\bin' 'DDR.assignment2.Unusable.RegisterWithoutEncapsulation'
Registered for Undergraduate courses
PS C:\Users\Syed Hassan\OneDrive\Desktop\Hackerrank>
```

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.


RegisterWithoutEncapsulation.java U
RegisterWithEncapsulation.java U ×
DDR > assignment2 > Reusable > RegisterWithEncapsulation.java > RegisterWithEncapsulation > main(String[])
Run | Debug
PROBLEMS 314
TERMINAL
DEBUG CONSOLE
PORTS
GITLENS
AZURE
COMMENTS
OUTPUT

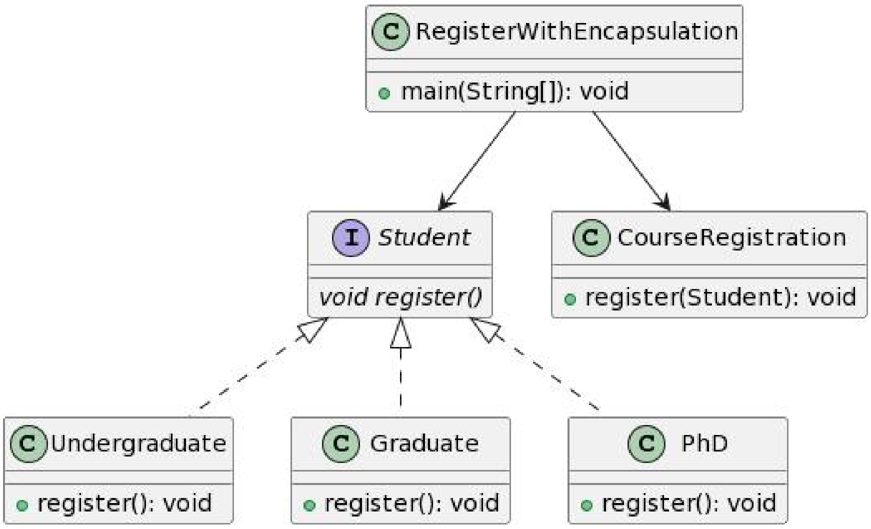
```java
interface Student {
  void register();
}

class Undergraduate implements Student {
  public void register() {
    System.out.println(x:"Registered for Undergraduate courses");
  }
}

class Graduate implements Student {
  public void register() {
    System.out.println(x:"Registered for Graduate courses");
  }
}

class PhD implements Student {
  public void register() {
    System.out.println(x:"Registered for PhD courses");
  }
}

class CourseRegistration {

  // This method is using encapsulation that varies therefore it is reusable

  public void register(Student student) {
    student.register();
  }
}

public class RegisterWithEncapsulation {

  public static void main(String[] args) {
    Student student = new Undergraduate();
    CourseRegistration courseRegistration = new CourseRegistration();
    courseRegistration.register(student);
  }

}
```

```
PS C:\Users\Syed Hassan\OneDrive\Desktop\Hackerrank>  & 'C:\Program Files\Java\jdk-18.0.
ers\Syed Hassan\AppData\Roaming\Code\User\workspaceStorage\9d1f256954f70e605207602abc7a7
e.RegisterWithEncapsulation'
Registered for Undergraduate courses
PS C:\Users\Syed Hassan\OneDrive\Desktop\Hackerrank>
```

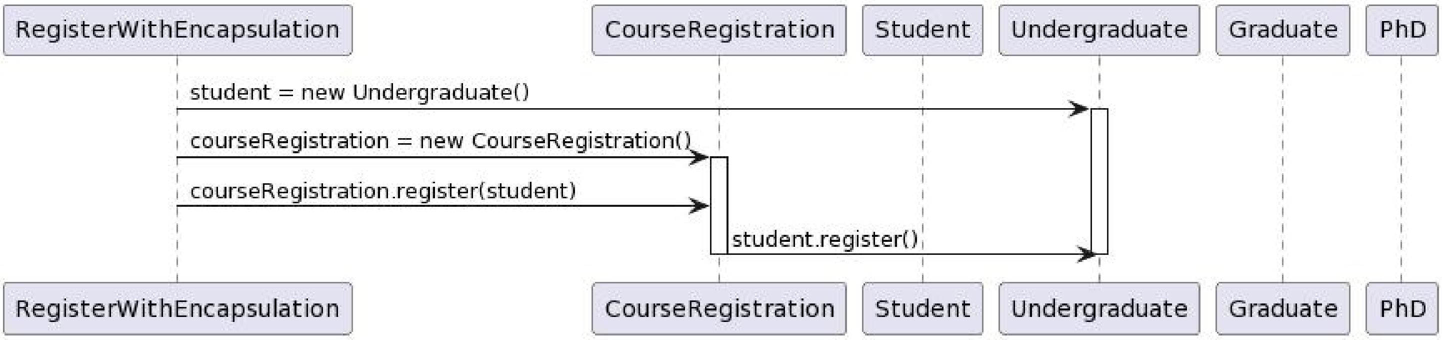
Each of these classes implements the **Student interface** and provides its own implementation of the register method. This means that the register method in the **CourseRegistration** class doesn't need to know about the different types of students or how they register. It simply calls the register 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

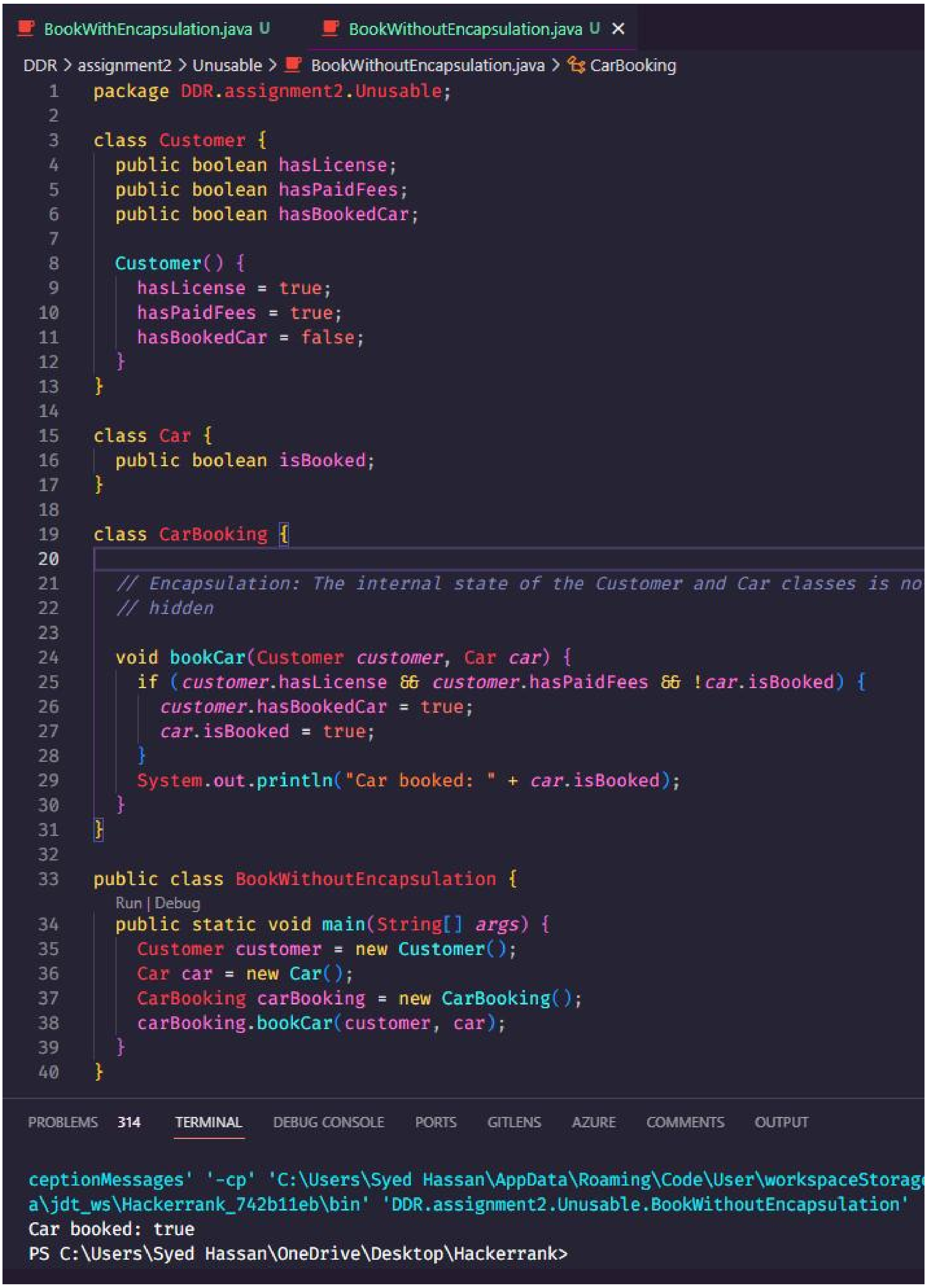


Interaction Diagram



RegisterWithEncapsulation
CourseRegistration
Student
Undergraduate
Graduate
PhD
student = new Undergraduate()
courseRegistration = new CourseRegistration()
courseRegistration.register(student)
student.register()
RegisterWithEncapsulation
CourseRegistration
Student
Undergraduate
Graduate
PhD

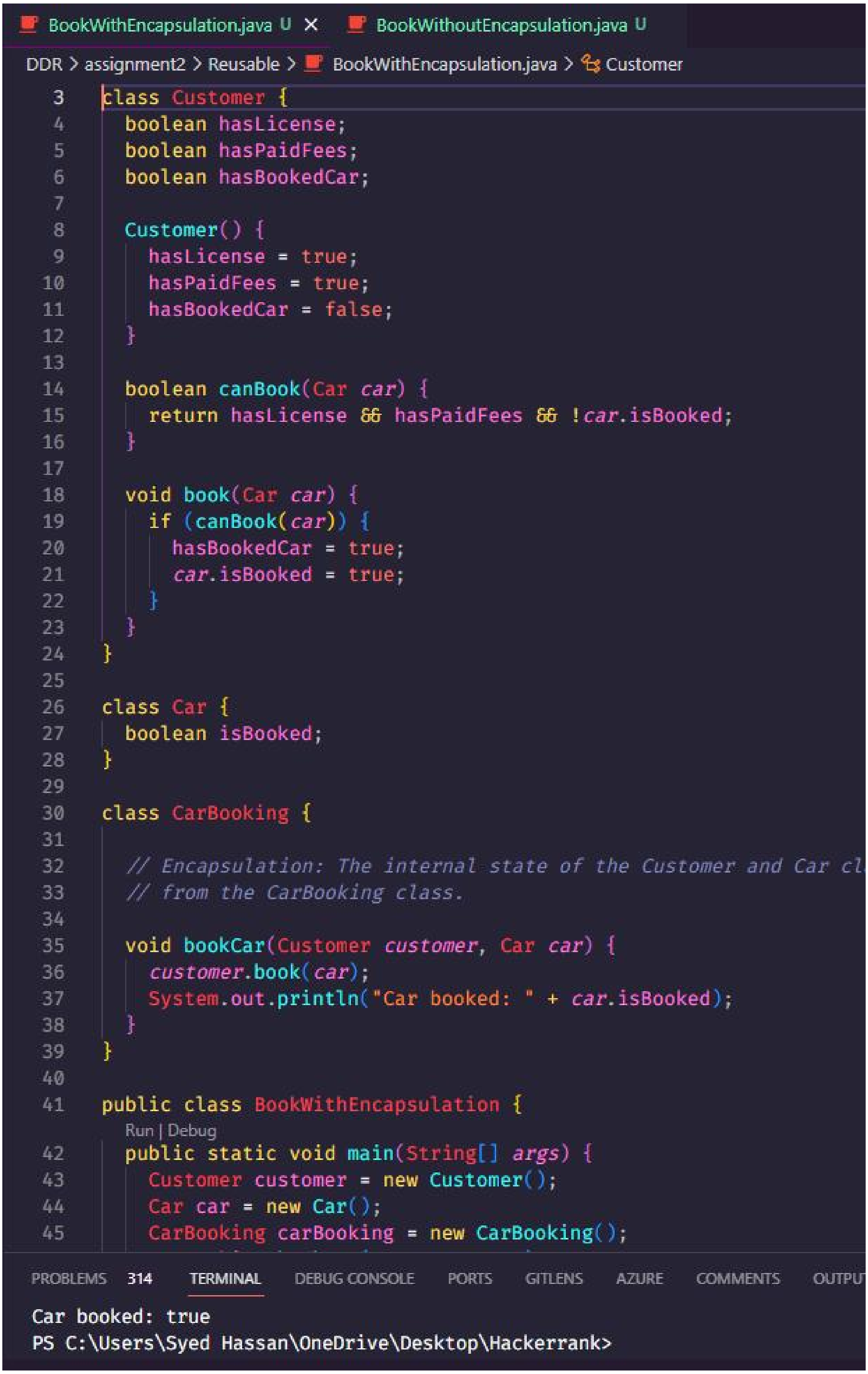
Example 2


BookWithEncapsulation.java U
BookWithoutEncapsulation.java U ×
DDR > assignment2 > Unusable > BookWithoutEncapsulation.java > CarBooking
Run | Debug
PROBLEMS 314
TERMINAL
DEBUG CONSOLE
PORTS
GITLENS
AZURE
COMMENTS
OUTPUT

```java
package DDR.assignment2.Unusable;

class Customer {
  public boolean hasLicense;
  public boolean hasPaidFees;
  public boolean hasBookedCar;

  Customer() {
    hasLicense = true;
    hasPaidFees = true;
    hasBookedCar = false;
  }
}

class Car {
  public boolean isBooked;
}

class CarBooking {

  // Encapsulation: The internal state of the Customer and Car classes is no
  // hidden

  void bookCar(Customer customer, Car car) {
    if (customer.hasLicense && customer.hasPaidFees && !car.isBooked) {
      customer.hasBookedCar = true;
      car.isBooked = true;
    }
    System.out.println("Car booked: " + car.isBooked);
  }
}

public class BookWithoutEncapsulation {
  public static void main(String[] args) {
    Customer customer = new Customer();
    Car car = new Car();
    CarBooking carBooking = new CarBooking();
    carBooking.bookCar(customer, car);
  }
}
```

```
ceptionMessages' '-cp' 'C:\Users\Syed Hassan\AppData\Roaming\Code\User\workspaceStorag
a\jdt_ws\Hackerrank_742b11eb\bin' 'DDR.assignment2.Unusable.BookWithoutEncapsulation'
Car booked: true
PS C:\Users\Syed Hassan\OneDrive\Desktop\Hackerrank>
```

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.


BookWithEncapsulation.java U ×
BookWithoutEncapsulation.java U
DDR > assignment2 > Reusable > BookWithEncapsulation.java > Customer
Run | Debug
PROBLEMS 314
TERMINAL
DEBUG CONSOLE
PORTS
GITLENS
AZURE
COMMENTS
OUTPU

```java
class Customer {
  boolean hasLicense;
  boolean hasPaidFees;
  boolean hasBookedCar;

  Customer() {
    hasLicense = true;
    hasPaidFees = true;
    hasBookedCar = false;
  }

  boolean canBook(Car car) {
    return hasLicense && hasPaidFees && !car.isBooked;
  }

  void book(Car car) {
    if (canBook(car)) {
      hasBookedCar = true;
      car.isBooked = true;
    }
  }
}

class Car {
  boolean isBooked;
}

class CarBooking {

  // Encapsulation: The internal state of the Customer and Car cl
  // from the CarBooking class.

  void bookCar(Customer customer, Car car) {
    customer.book(car);
    System.out.println("Car booked: " + car.isBooked);
  }
}

public class BookWithEncapsulation {
  public static void main(String[] args) {
    Customer customer = new Customer();
    Car car = new Car();
    CarBooking carBooking = new CarBooking();
```

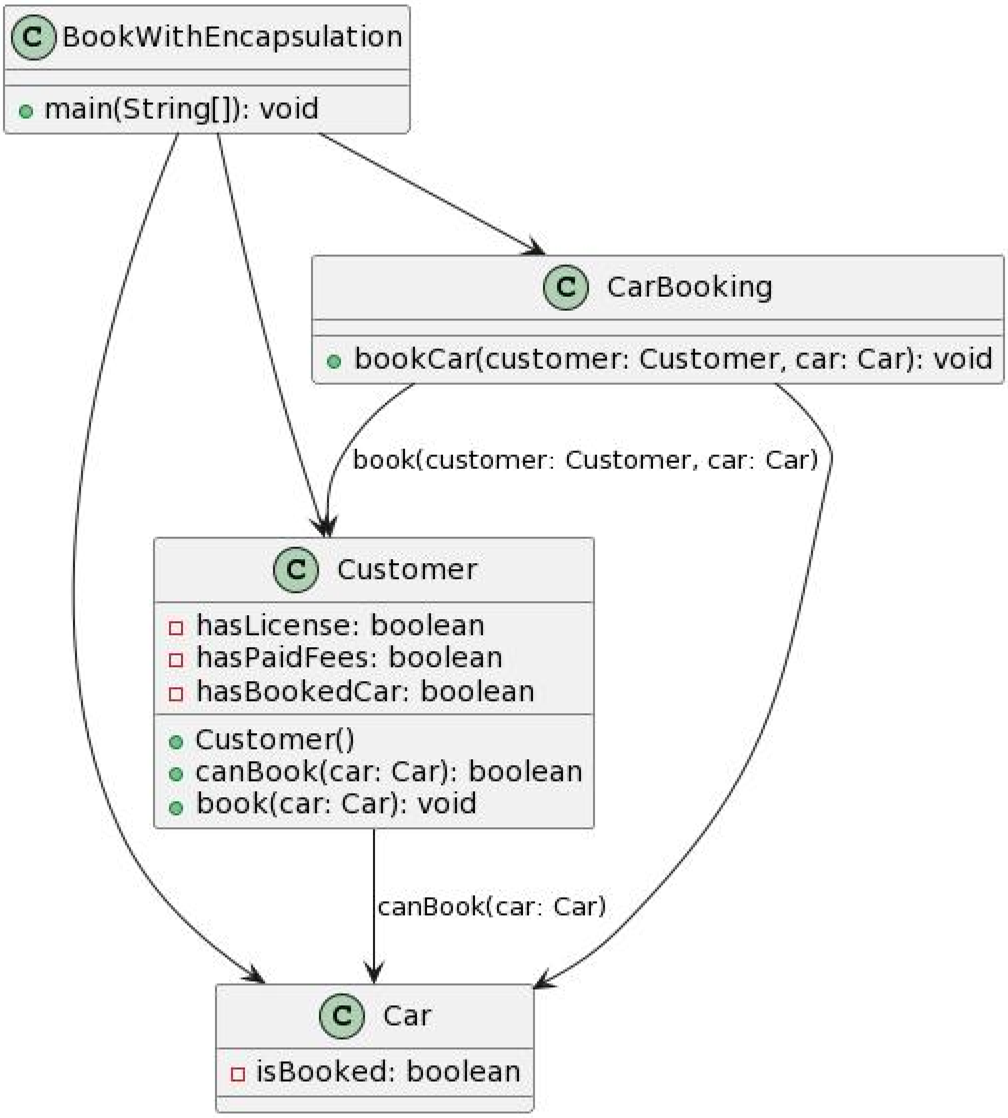
```
Car booked: true
PS C:\Users\Syed Hassan\OneDrive\Desktop\Hackerrank>
```

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.

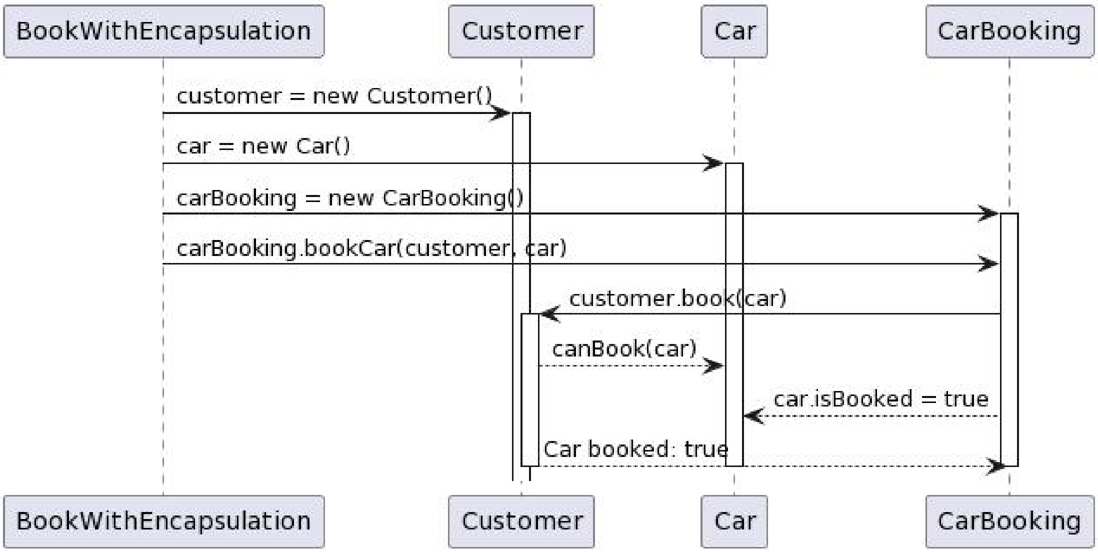
Similarly, in the Car class, the isBooked field is private and can only be accessed or modified through the methods in the Customer class.

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.

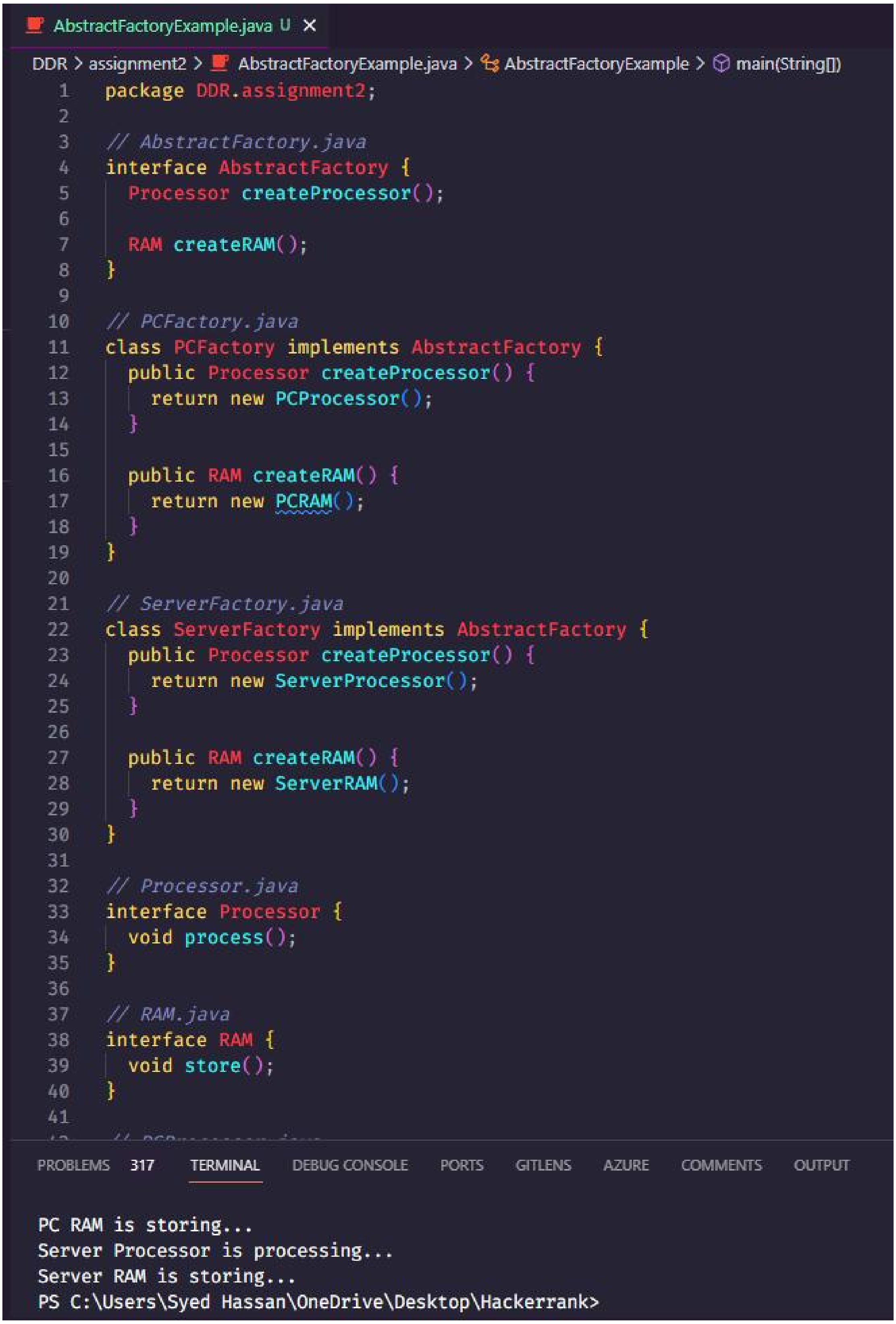
Class Diagram



Interaction Diagram

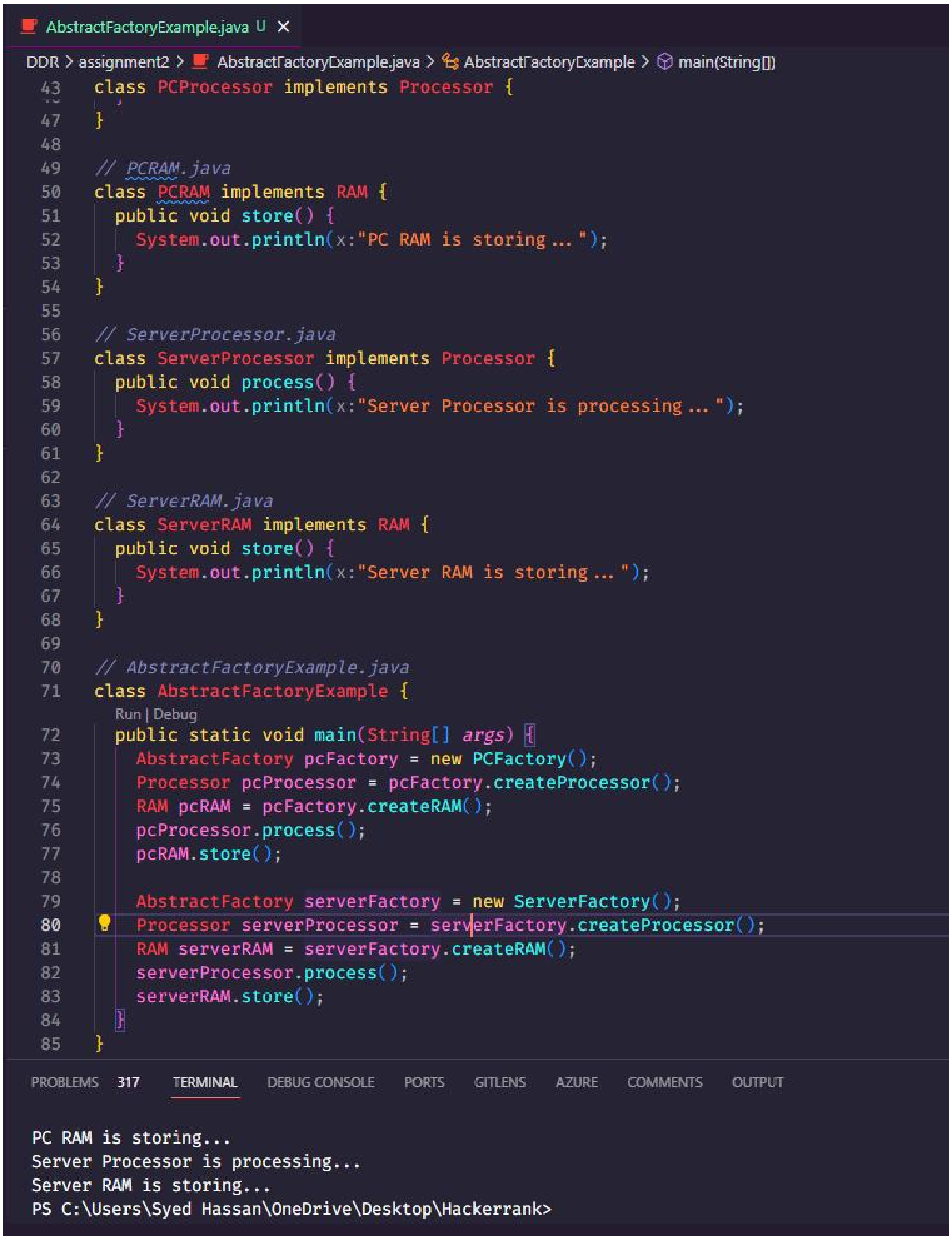


Task B


AbstractFactoryExample.java U ✕
DDR > assignment2 > AbstractFactoryExample.java > AbstractFactoryExample > main(String[])
PROBLEMS 317
TERMINAL
DEBUG CONSOLE
PORTS
GITLENS
AZURE
COMMENTS
OUTPUT

```java
package DDR.assignment2;

// AbstractFactory.java
interface AbstractFactory {
  Processor createProcessor();

  RAM createRAM();
}

// PCFactory.java
class PCFactory implements AbstractFactory {
  public Processor createProcessor() {
    return new PCProcessor();
  }

  public RAM createRAM() {
    return new PCRAM();
  }
}

// ServerFactory.java
class ServerFactory implements AbstractFactory {
  public Processor createProcessor() {
    return new ServerProcessor();
  }

  public RAM createRAM() {
    return new ServerRAM();
  }
}

// Processor.java
interface Processor {
  void process();
}

// RAM.java
interface RAM {
  void store();
}

```

```
PC RAM is storing...
Server Processor is processing...
Server RAM is storing...
PS C:\Users\Syed Hassan\OneDrive\Desktop\Hackerrank>
```

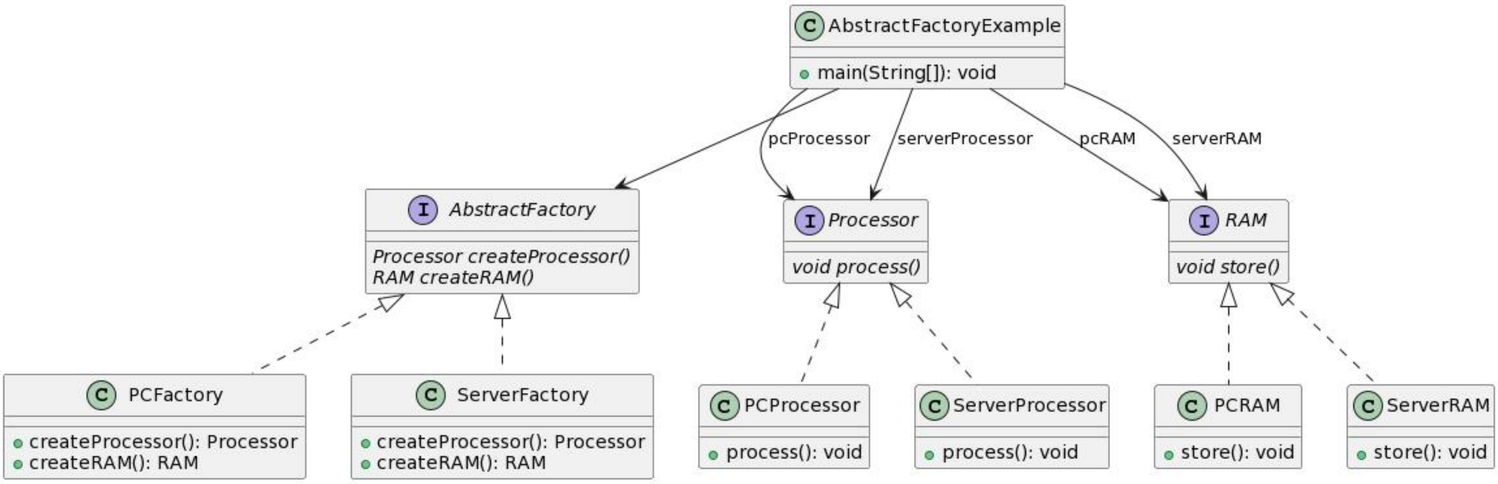

AbstractFactoryExample.java U ×
DDR > assignment2 > AbstractFactoryExample.java > AbstractFactoryExample > main(String[])
Run | Debug
PROBLEMS 317
TERMINAL
DEBUG CONSOLE
PORTS
GITLENS
AZURE
COMMENTS
OUTPUT

```java
class PCProcessor implements Processor {
}

// PCRAM.java
class PCRAM implements RAM {
    public void store() {
        System.out.println(x:"PC RAM is storing...");
    }
}

// ServerProcessor.java
class ServerProcessor implements Processor {
    public void process() {
        System.out.println(x:"Server Processor is processing...");
    }
}

// ServerRAM.java
class ServerRAM implements RAM {
    public void store() {
        System.out.println(x:"Server RAM is storing...");
    }
}

// AbstractFactoryExample.java
class AbstractFactoryExample {
    public static void main(String[] args) {
        AbstractFactory pcFactory = new PCFactory();
        Processor pcProcessor = pcFactory.createProcessor();
        RAM pcRAM = pcFactory.createRAM();
        pcProcessor.process();
        pcRAM.store();

        AbstractFactory serverFactory = new ServerFactory();
        Processor serverProcessor = serverFactory.createProcessor();
        RAM serverRAM = serverFactory.createRAM();
        serverProcessor.process();
        serverRAM.store();
    }
}
```

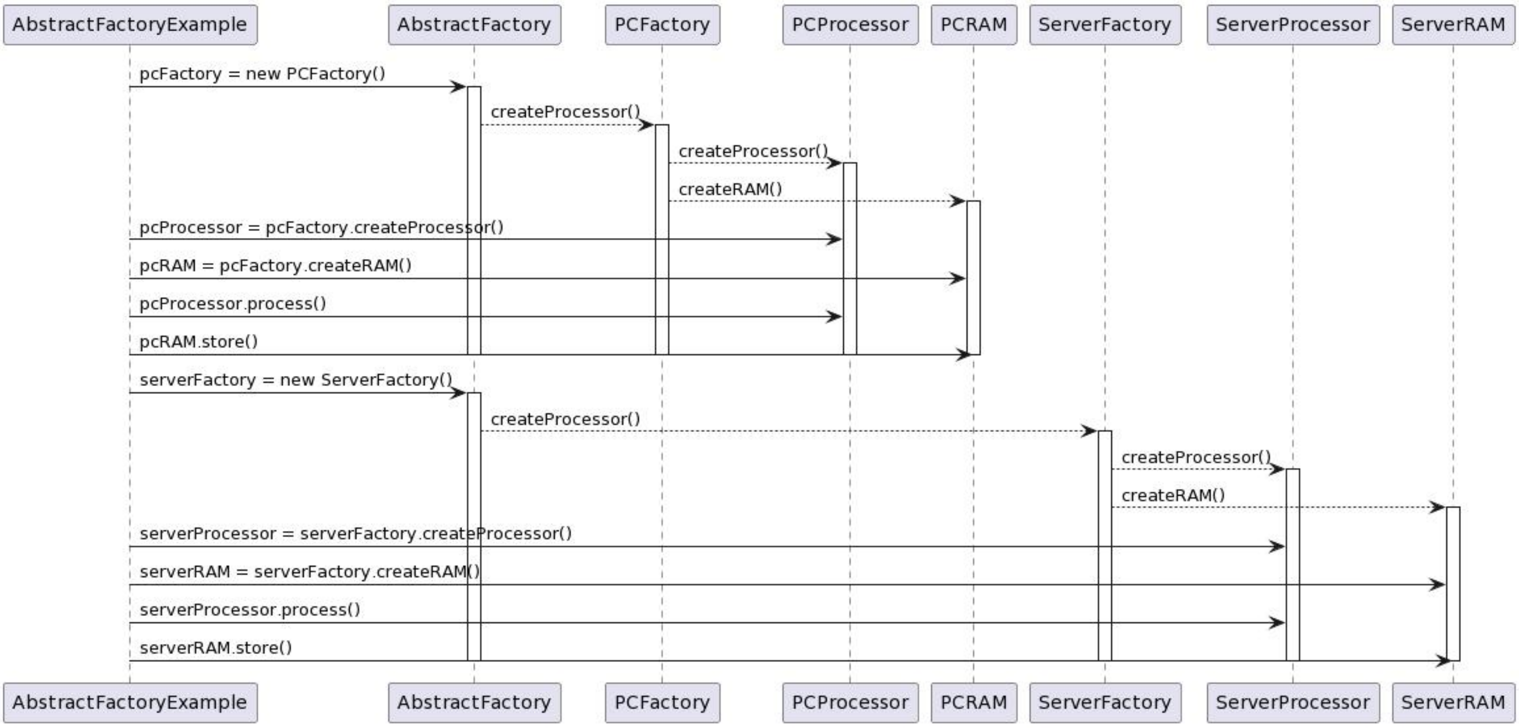
```
PC RAM is storing...
Server Processor is processing...
Server RAM is storing...
PS C:\Users\Syed Hassan\OneDrive\Desktop\Hackerrank>
```

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


AbstractFactoryExample
AbstractFactory
PCFactory
PCProcessor
PCRAM
ServerFactory
ServerProcessor
ServerRAM
pcFactory = new PCFactory()
createProcessor()
createProcessor()
createRAM()
pcProcessor = pcFactory.createProcessor()
pcRAM = pcFactory.createRAM()
pcProcessor.process()
pcRAM.store()
serverFactory = new ServerFactory()
createProcessor()
createProcessor()
createRAM()
serverProcessor = serverFactory.createProcessor()
serverRAM = serverFactory.createRAM()
serverProcessor.process()
serverRAM.store()
AbstractFactoryExample
AbstractFactory
PCFactory
PCProcessor
PCRAM
ServerFactory
ServerProcessor
ServerRAM