Lesson 04 Demo 04

Creating and Using Design Patterns Factory and State

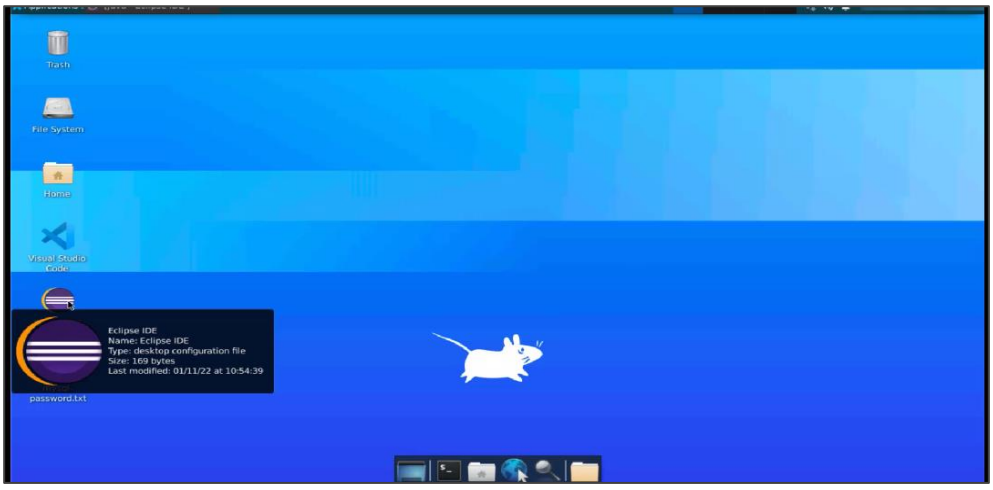
Objective: To implement the Factory and State design patterns **Tools required**: Eclipse IDE **Prerequisite**: None

Steps to be followed:

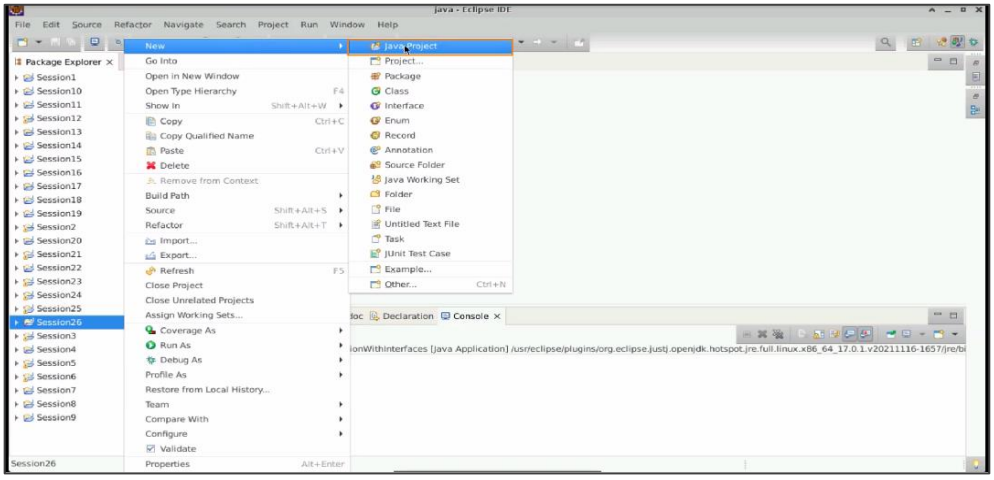
1. Implement and create Factory Design Pattern class and Plan interface
2. Create class Plan3G
3. Create class Plan4G and Plan5G
4. Create class Plan Factory
5. Implement the methods of the interface and classes
6. Create State Design Patterns

Step 1: Implement and create Factory Design Pattern class and Plan interface

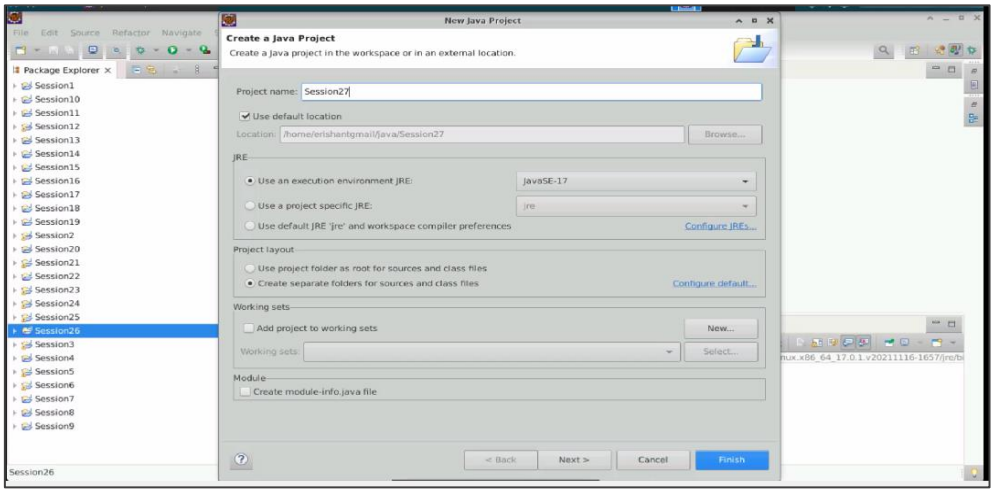
1.1 Open the **Eclipse IDE.**



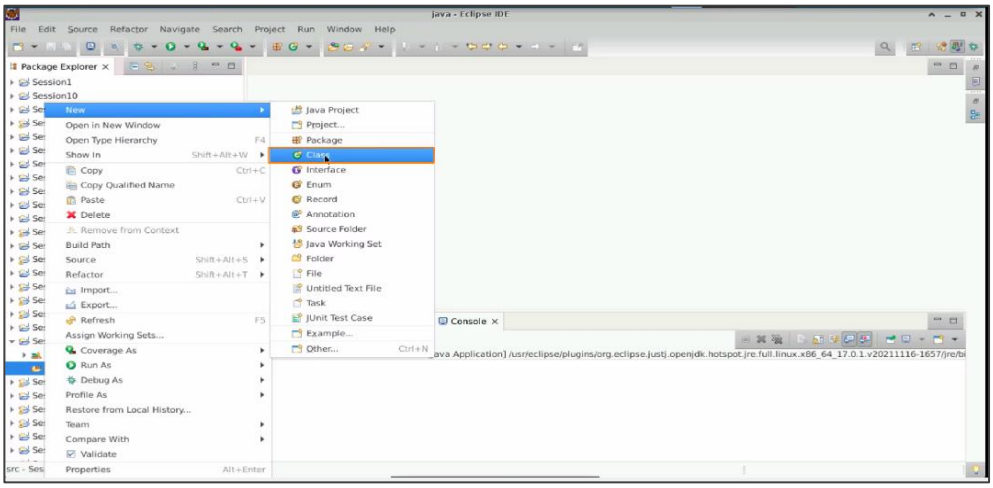
1.2 Select **File,** then **New**, and then **Java Project.**



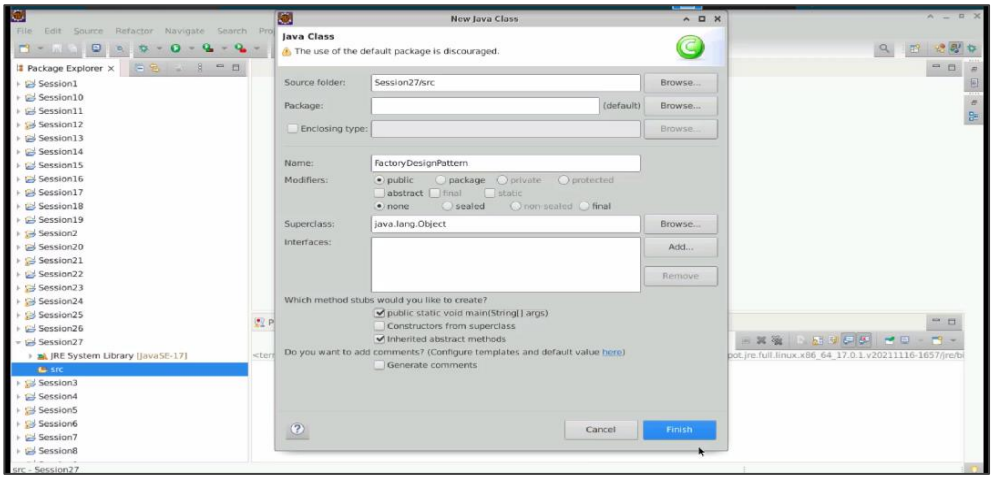
1.3 Name the project as “**Session27”** and select **Finish**.



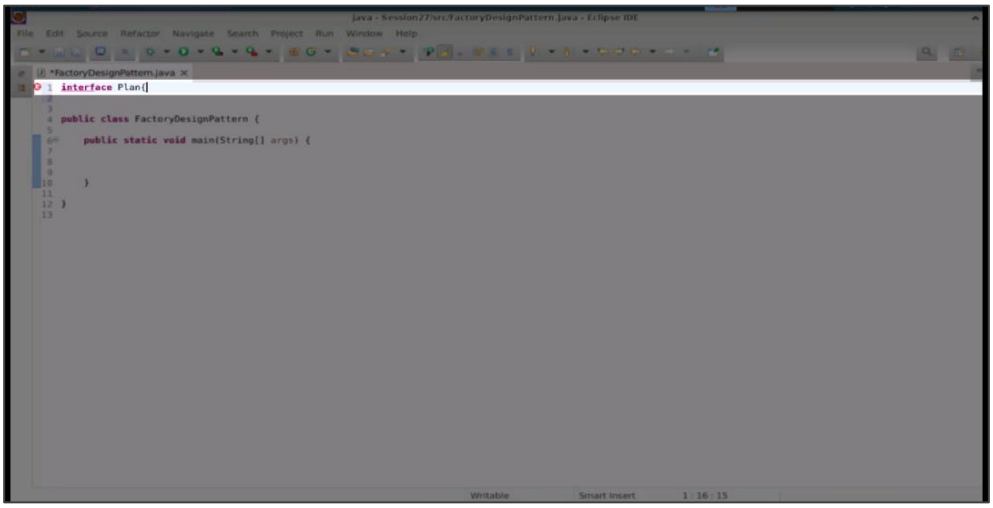
1.4 Right click on the **src** folder under the project name, select **New**, and then select **Class**.



1.5 Name the class as **FactoryDesignPattern** and select the option **public static void main(String[] args)**. Select **Finish**. This will create a java class with a **main()** method.

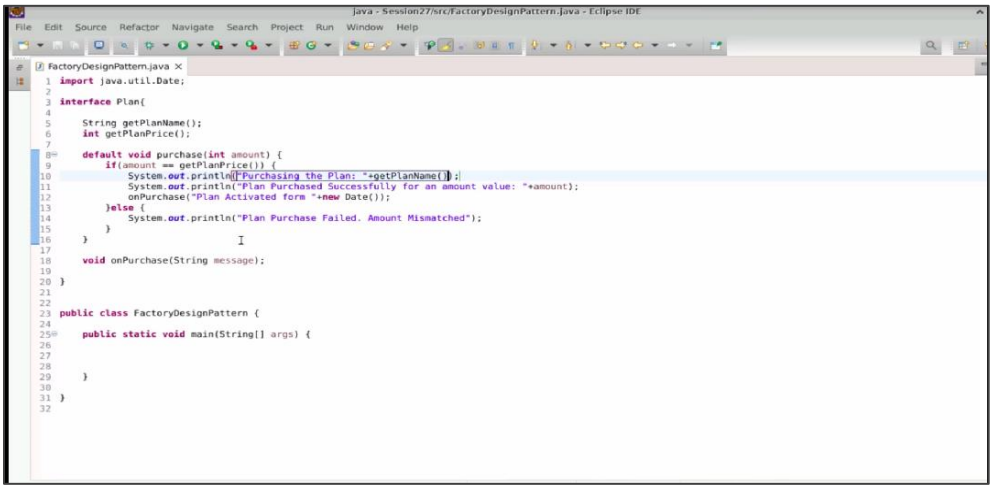


1.6 In the class **FactoryDesignPattern**, create an interface **Plan** by writing the **interface Plan.**



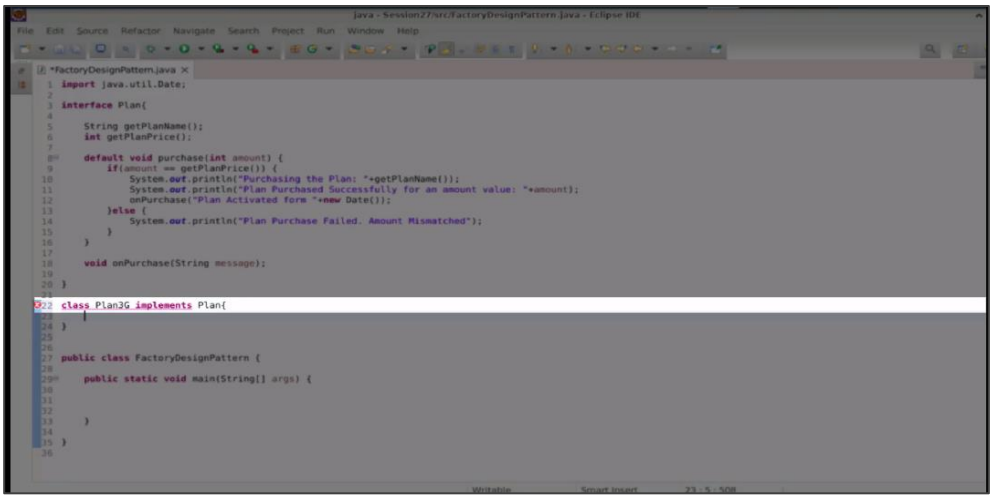
1.7 Create four methods inside the interface. The methods are:

- **String getPlanName() :** This method will return the plan name as a string.
- **int getPlanPrice() :** This method will return the price of the selected plan as an integer value.
- **default void purchase(int amount)** : In the purchasing technique, you will first determine whether the amount is equal to determine the plan price.
  If your amount equals the plan price only then will you complete a transaction, regardless of the plan price and print the **Plan purchased successfully for the amount** and mention the amount using the **system.out.println().** Else, print **Plan purchase failed. Amount Mismatched** in the **else** section.
- **void onPurchase(String message) :** This method will be executed in the **purchase()** method and will display a string message **Plan activated from** and the current date and time.

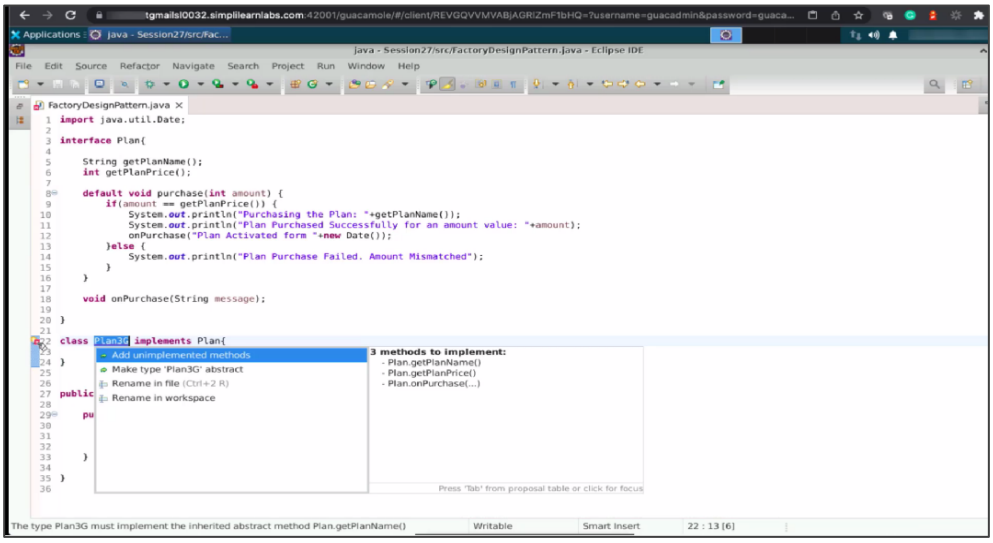


Step 2: Create class Plan3G

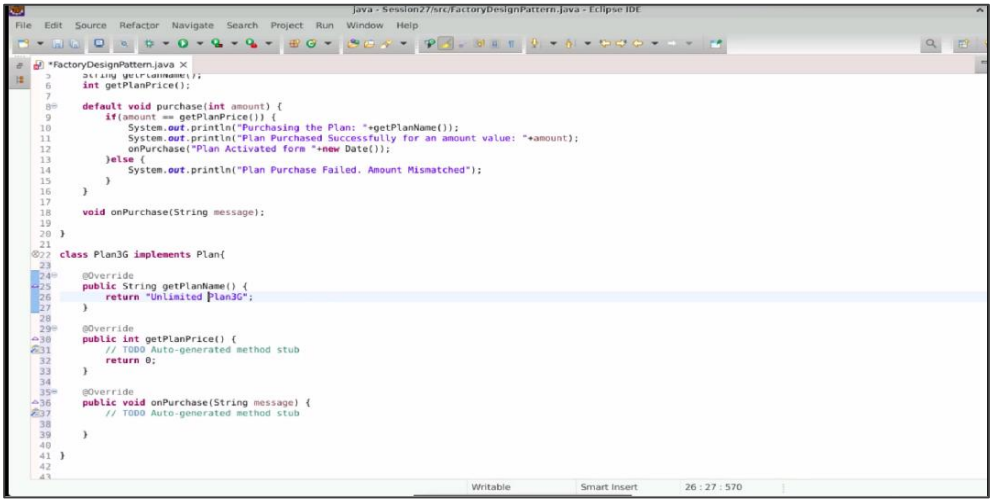
2.1 Create a new class **Plan3G** which implements **Plan** interface as **class Plan3G implements Plan.**



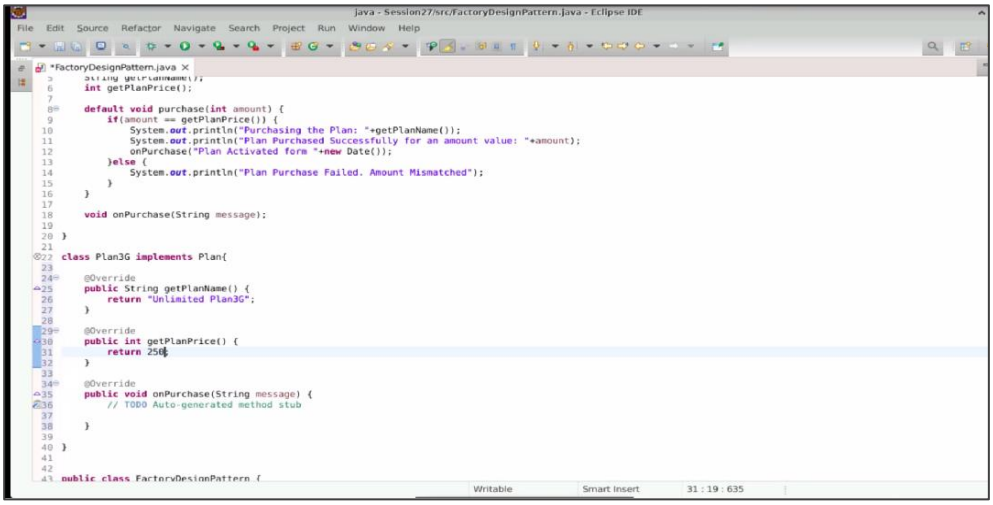
2.2 Click on the error bulb of **Plan3G** and select the option **Add unimplemented methods** to implement the methods of the plan interface.



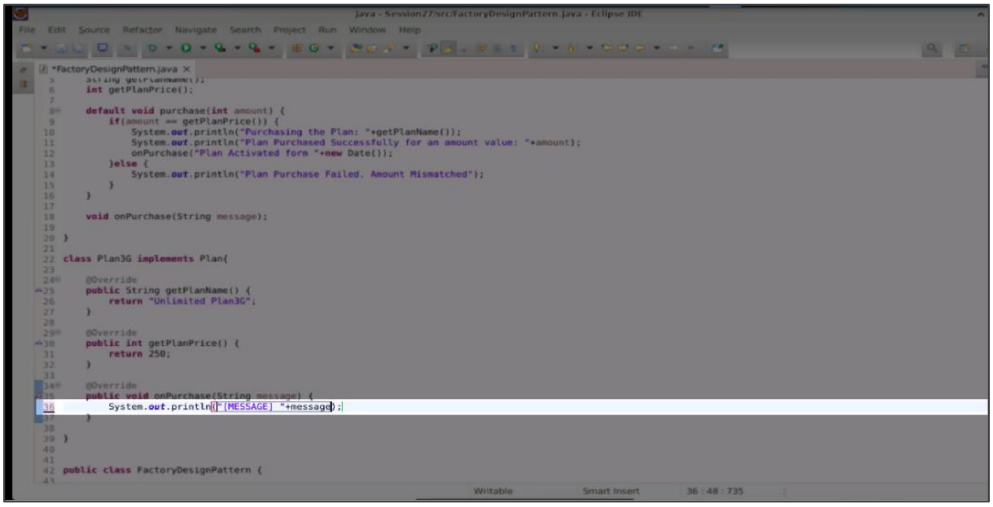
2.3 In **getPlanName()** method, write **return “Unlimited plan3G”.**



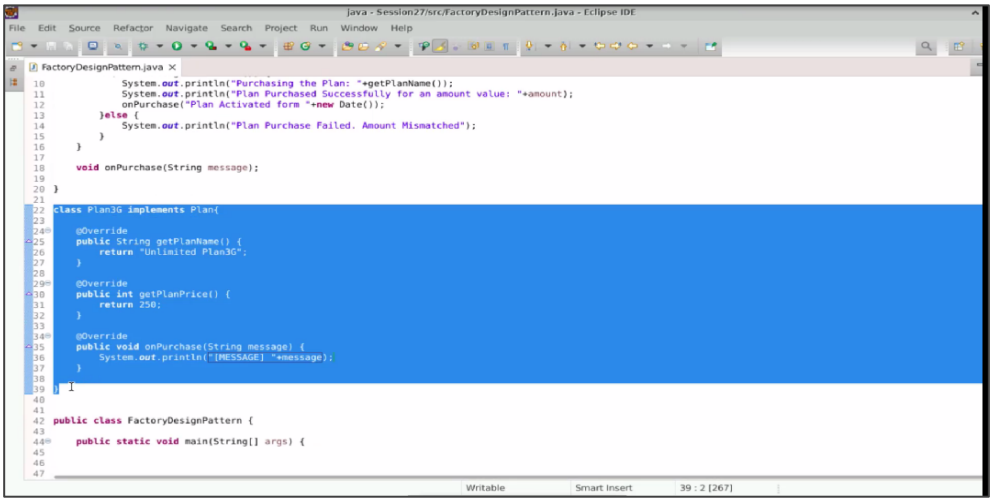
2.4 In the **getPlanPrice()** method, write **return 250**.



2.5 In the **onPurchase()** method, write **system.out.println("[MESSAGE] "+message)**

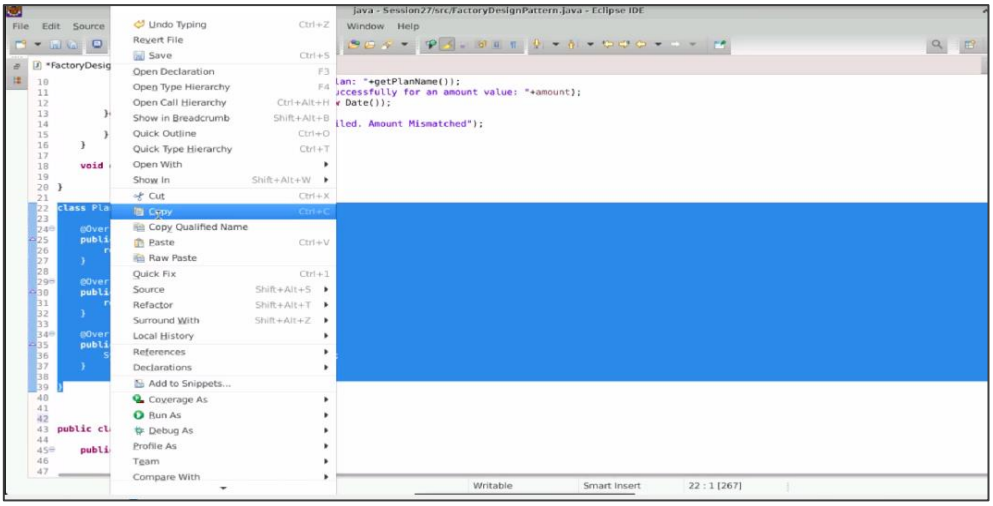


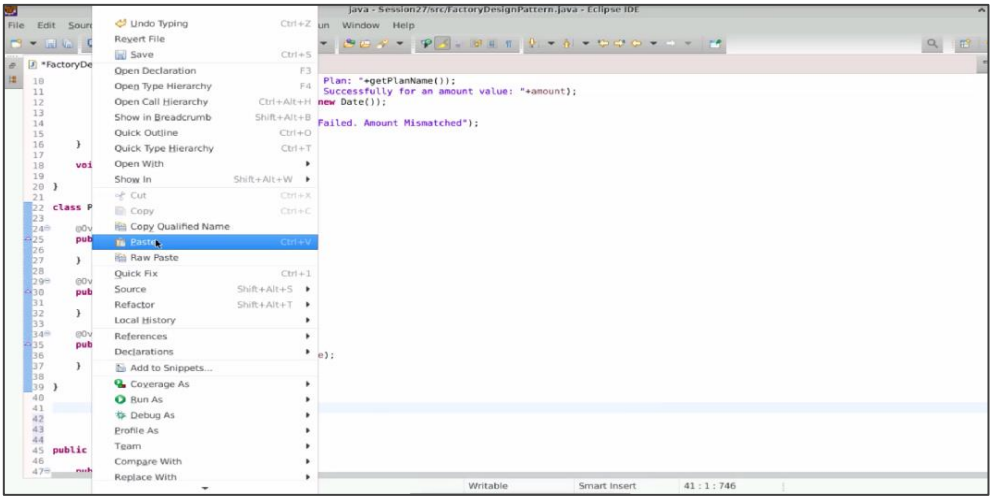
2.6 Here, we have **Plan3G** implementing one of the plans of the interface.

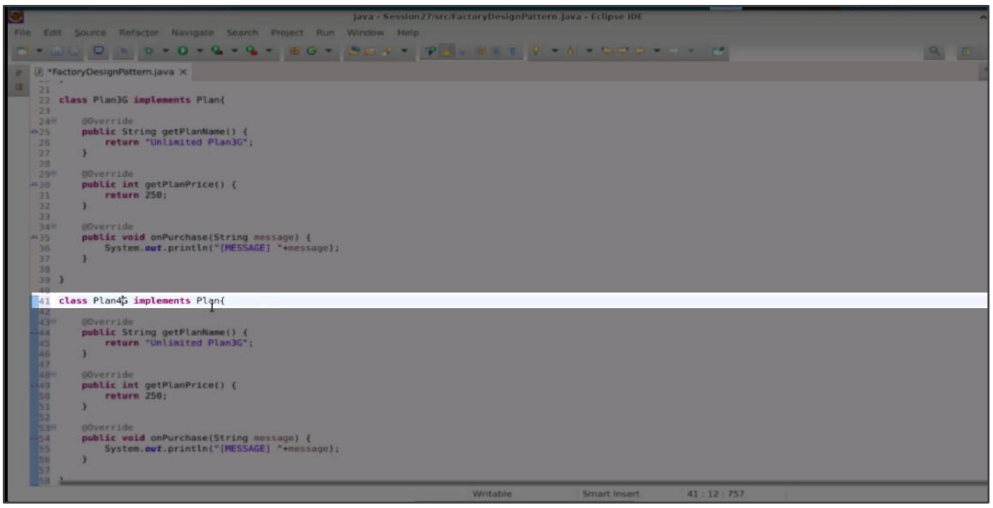


Step 3: Create class Plan4G and Plan5G

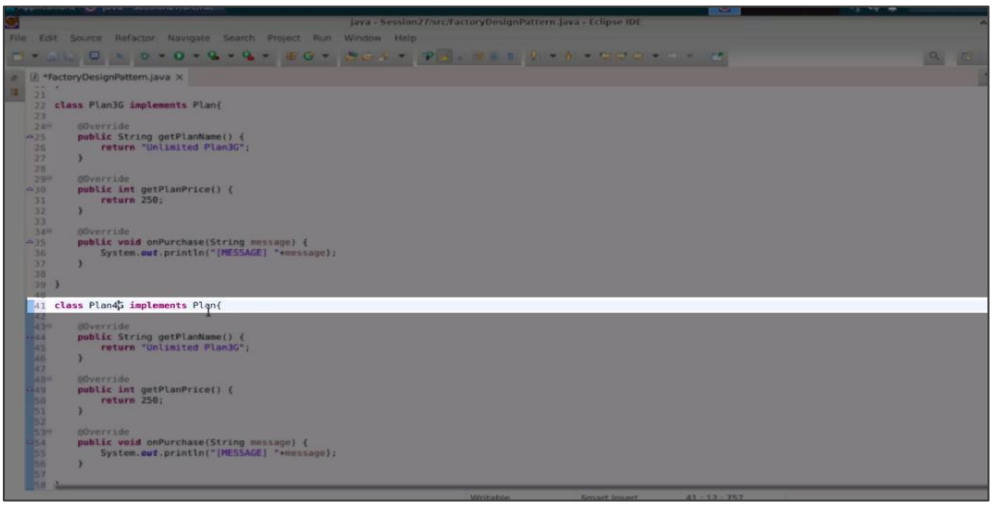
3.1 Create class **Plan4G** by copying and pasting class **Plan3G.**



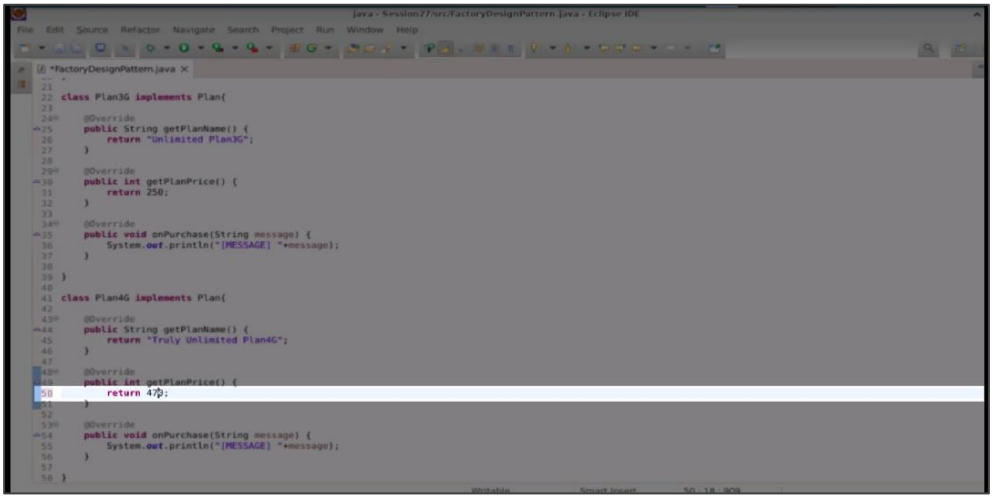




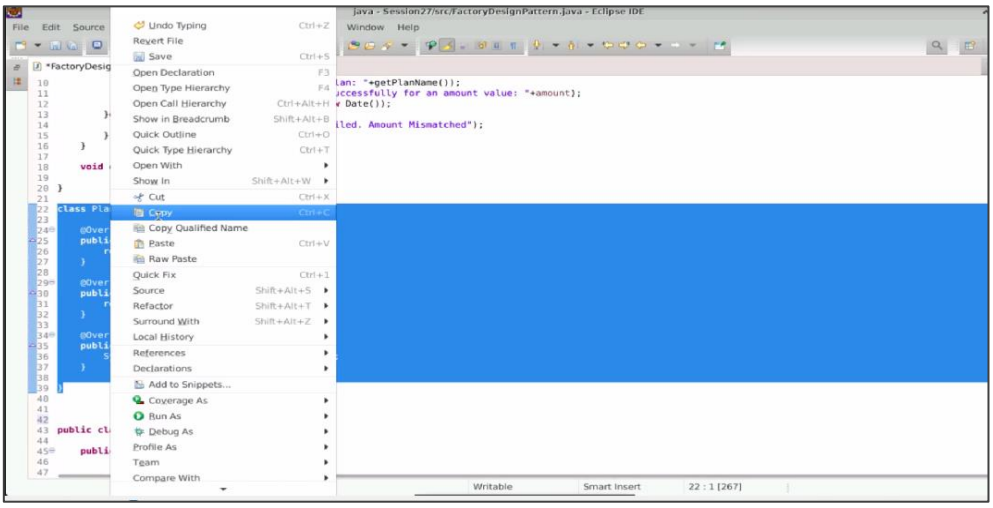
3.2 In **getPlanName()** method, write **return "Truly Unlimited plan4G".**

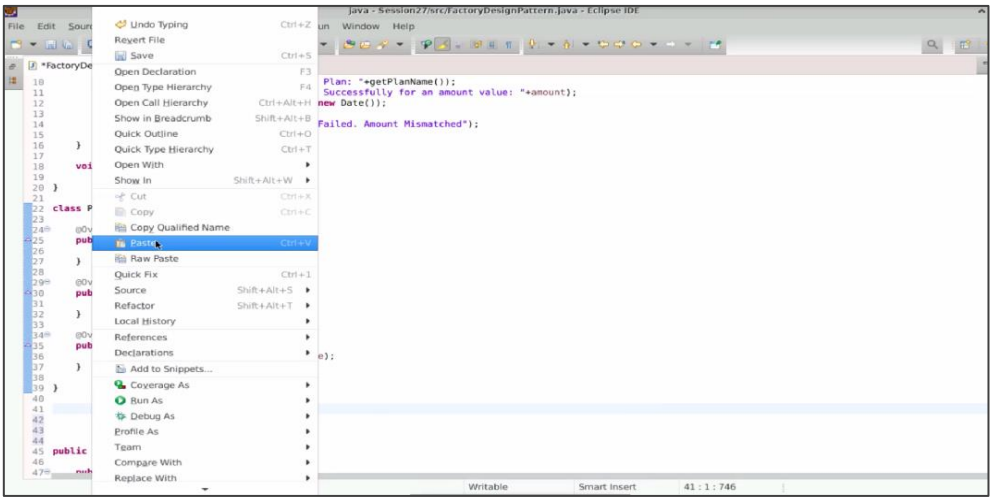


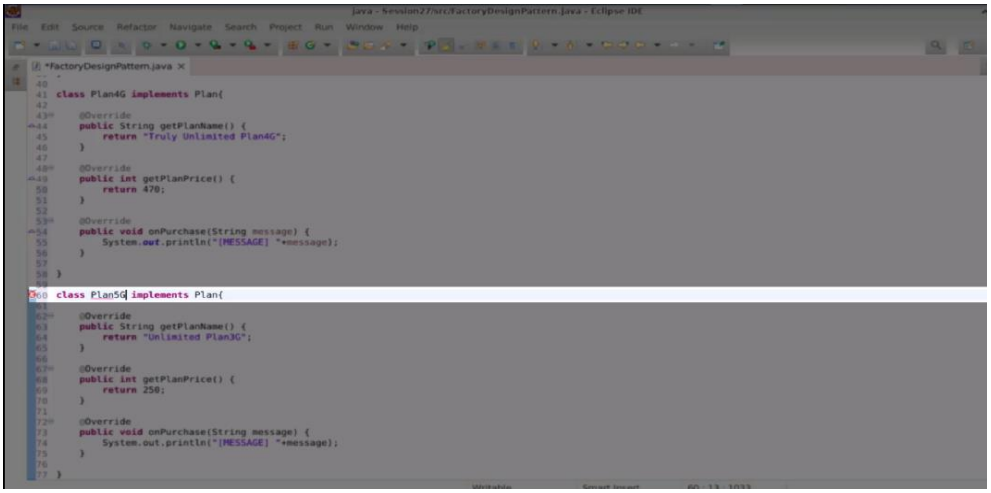
3.3 In the **getPlanPrice()** method, write **return 470**.



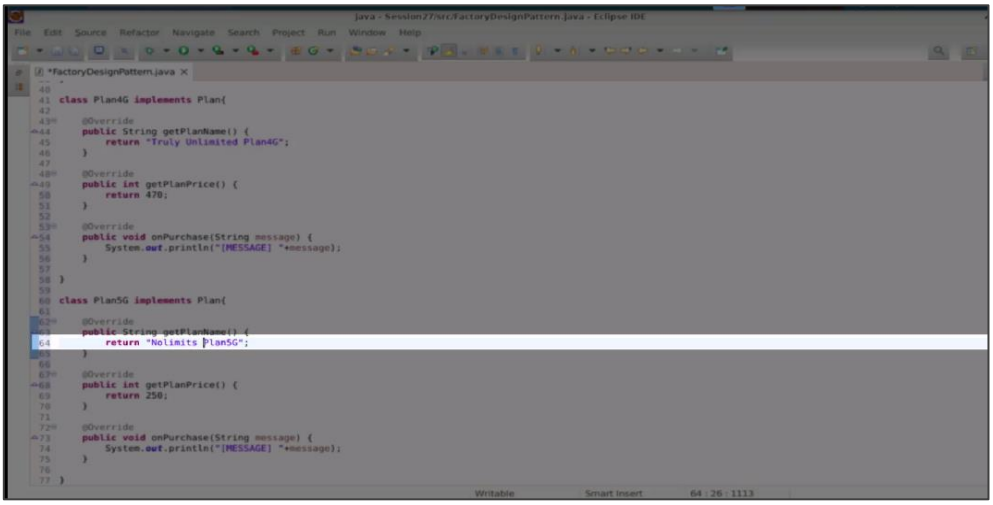
3.4 Create class **Plan5G** by copying and pasting class **Plan3G.**



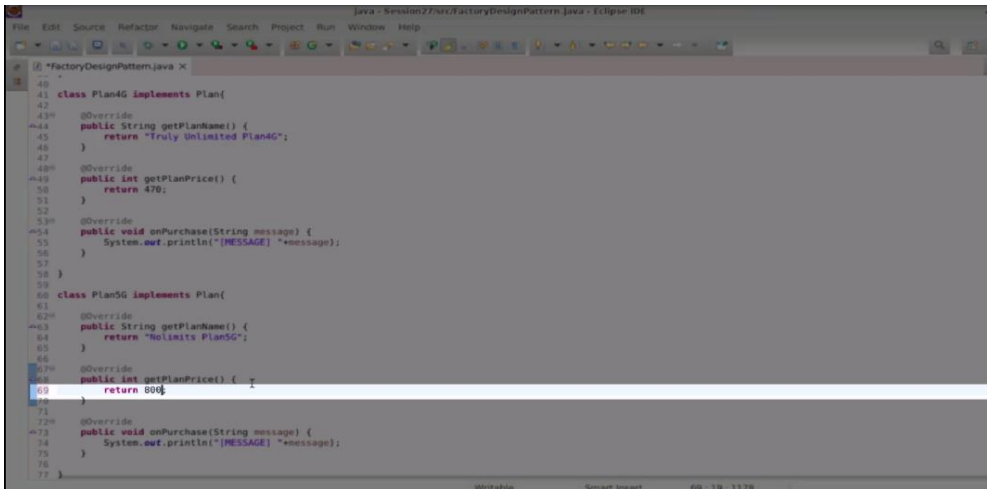




3.5 In **getPlanName()** method, write **return Nolimits plan5G.**

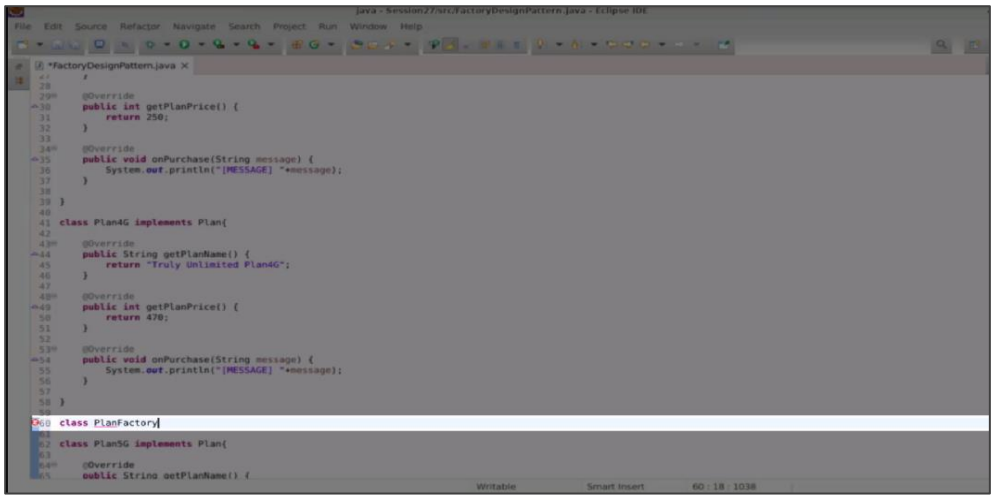


3.6 In the **getPlanPrice()** method, write **return 800**.

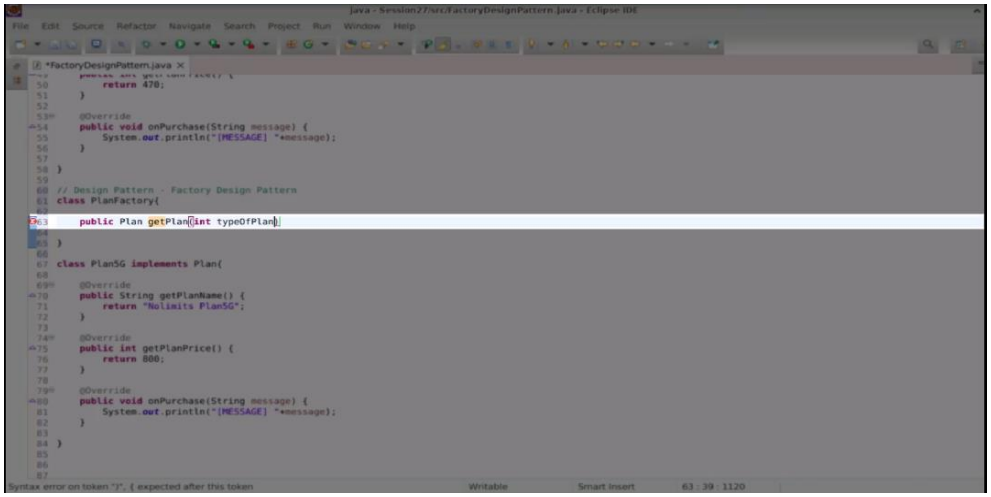


Step 4: Create class PlanFactory

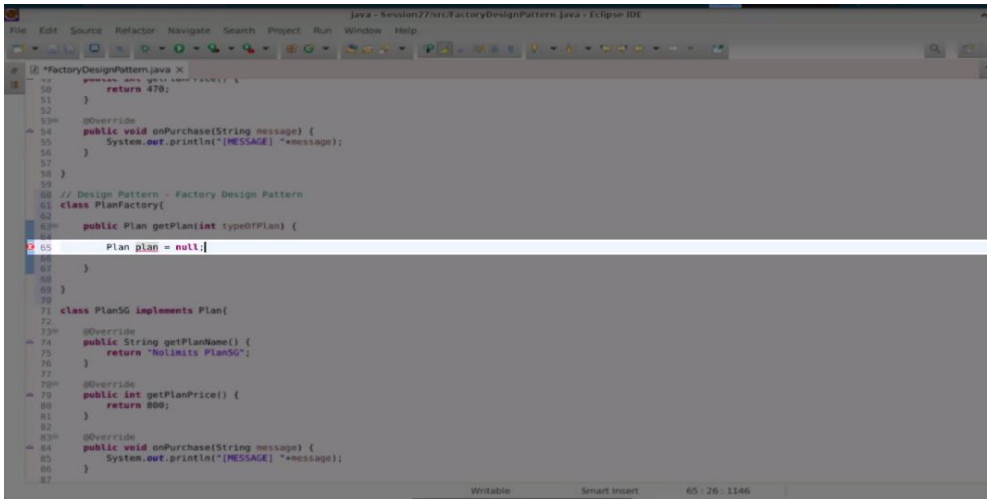
4.1 Create another class **PlanFactory** by writing the **class PlanFactory** to implement the factory design pattern.



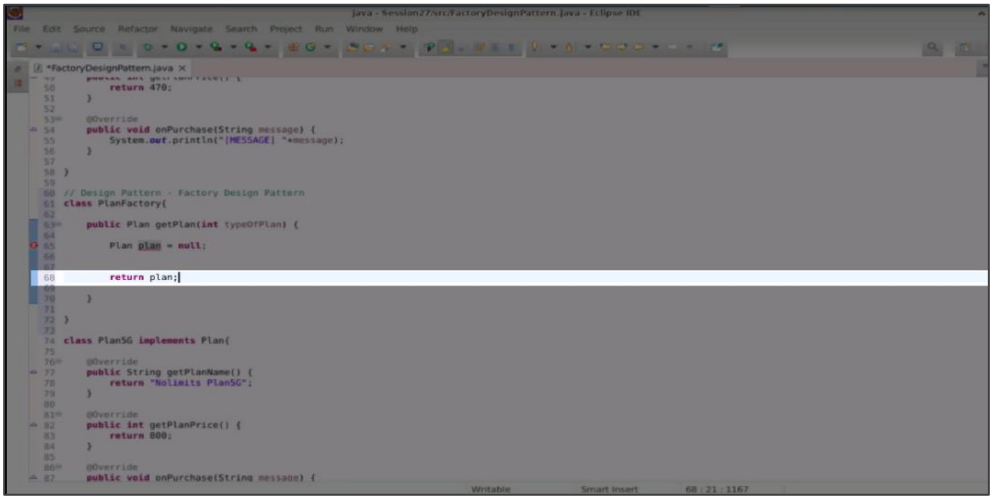
4.2 Create a public method which returns a **plan**. Code it as **public Plan getPlan(int typeOfPlan)** which takes typeOfPlan as input.



4.3 Create a reference variable of plan which is by default **null.**



4.4 Return the reference variable plan.



4.5 If the type of plan is equivalent to 3, you are attempting to make reference to a 3G plan. And, "4" refers to the 4G plan. The type of plan is equivalent to five then denotes a 5G plan, and in any other situation, you can inform the user that their plan is invalid.

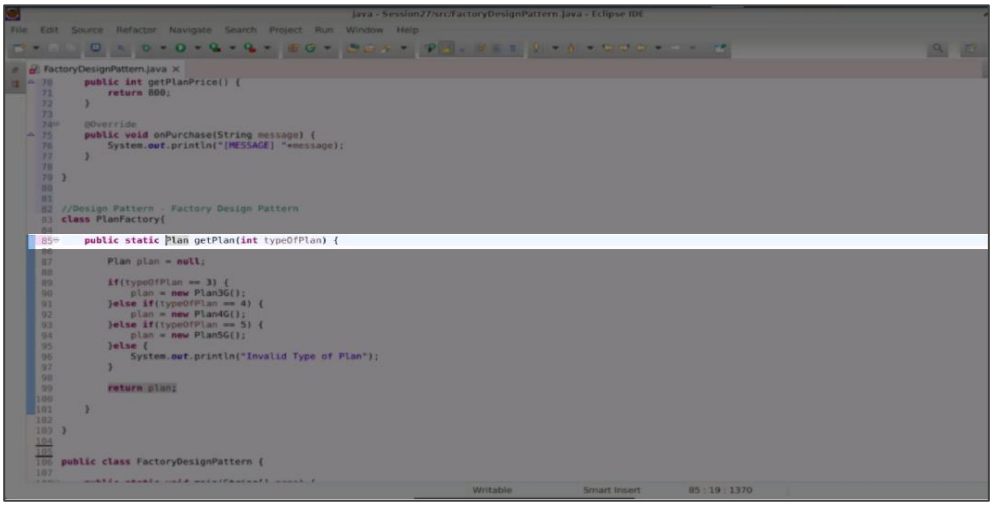
```
// Design Pattern - Factory Design Pattern
class PlanFactory{

    public Plan getPlan(int typeOfPlan) {

        Plan plan = null;

        if(typeOfPlan == 3) {

        }else if(typeOfPlan == 4) {

        }else if(typeOfPlan == 5) {

        }else {
            System.out.println("Invalid Type of Plan");
        }

        return plan;

    }

}
```

4.6 According to the selected plan, create the reference objects of respective plans. This is called runtime polymorphism; the same reference variable plan can refer to any plan object. This is the polymorphic behavior with the interfaces.

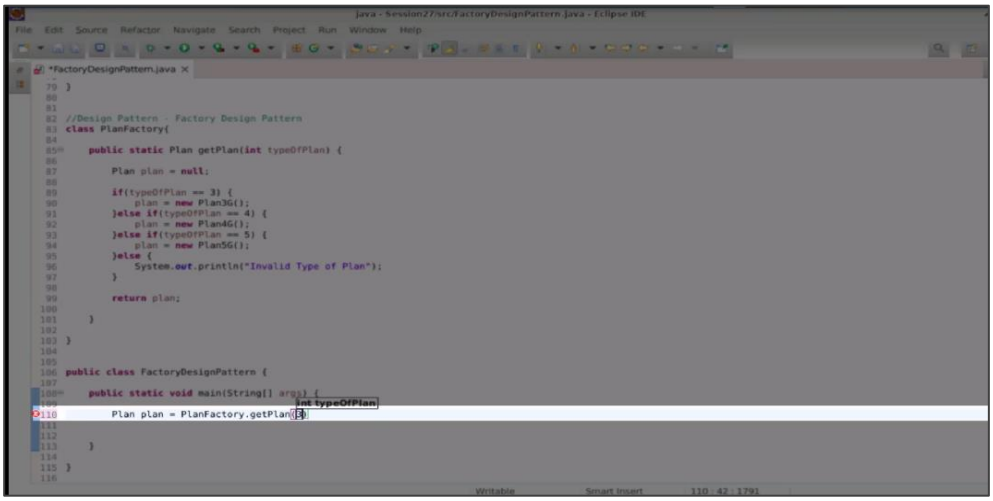
```
// Design Pattern - Factory Design Pattern
class PlanFactory{

    public Plan getPlan(int typeOfPlan) {

        Plan plan = null;

        if(typeOfPlan == 3) {
            plan = new Plan3G();
        }else if(typeOfPlan == 4) {
            plan = new Plan4G();
        }else if(typeOfPlan == 5) {
            plan = new Plan5G();
        }else {
            System.out.println("Invalid Type of Plan");
        }

        return plan;

    }

}
```

Step 5: Implement the methods of the interface and classes

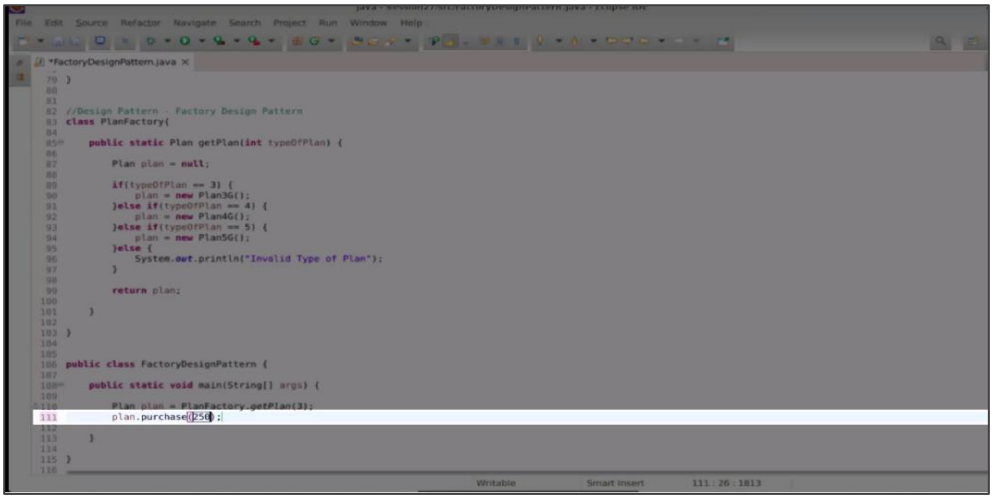
5.1 Make the **getPlan** method of class **PlanFactory** as **static** so that we need not create the object to implement this method.



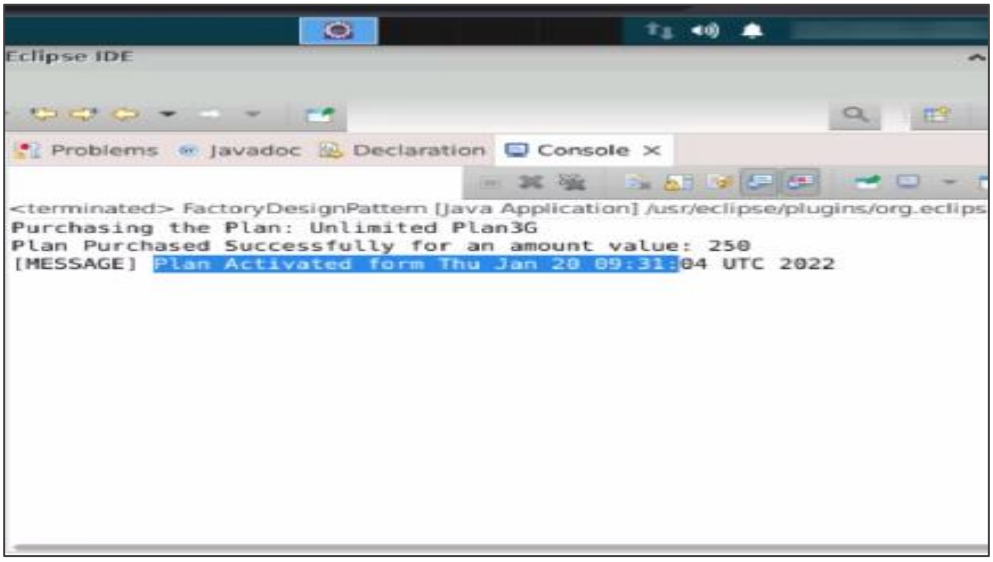
5.2 Now under the main of the class **FactoryDesignPattern**, call the getPlan method as **Plan plan = PlanFactory.getPlan().** Pass the value 3 inside the function call.



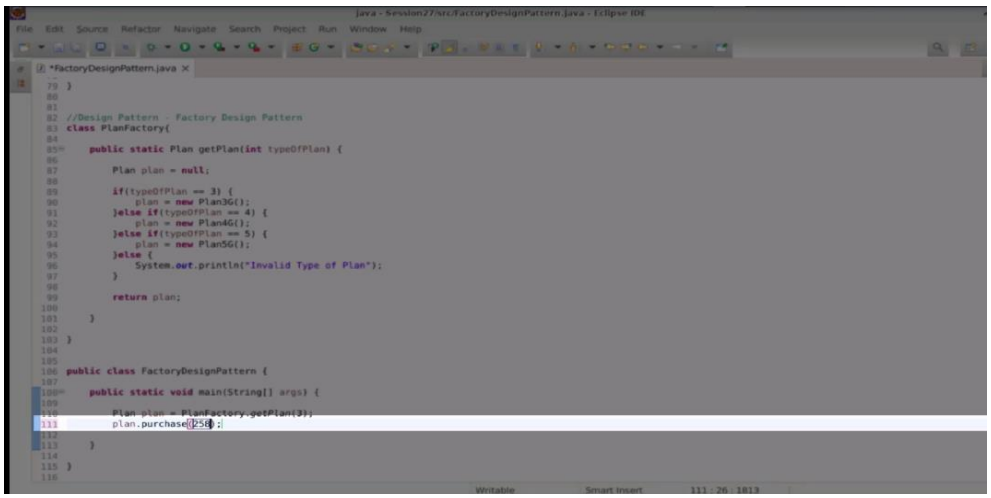
5.3 Call the **purchase()** method as **plan.purchase()** and then enter the amount. This amount should match the type of plan amount.

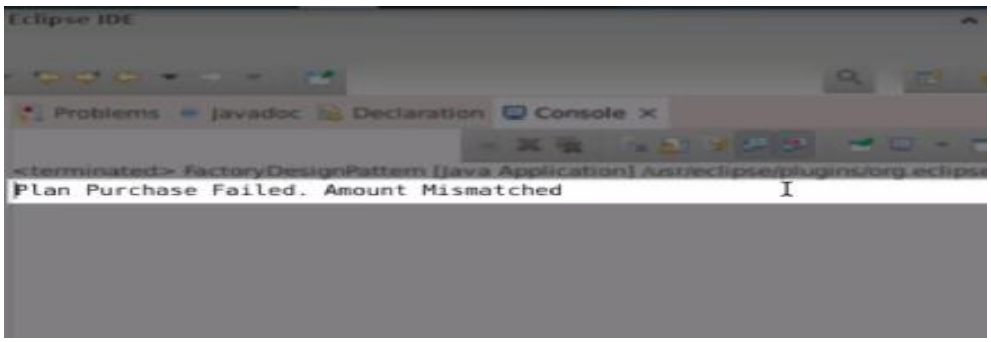


5.4 Run the code and the following output will be obtained.

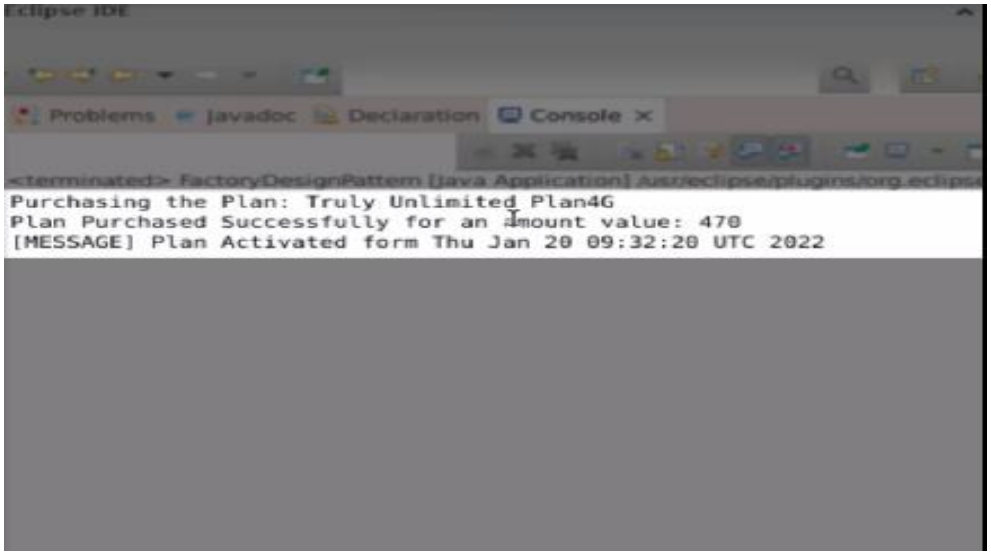


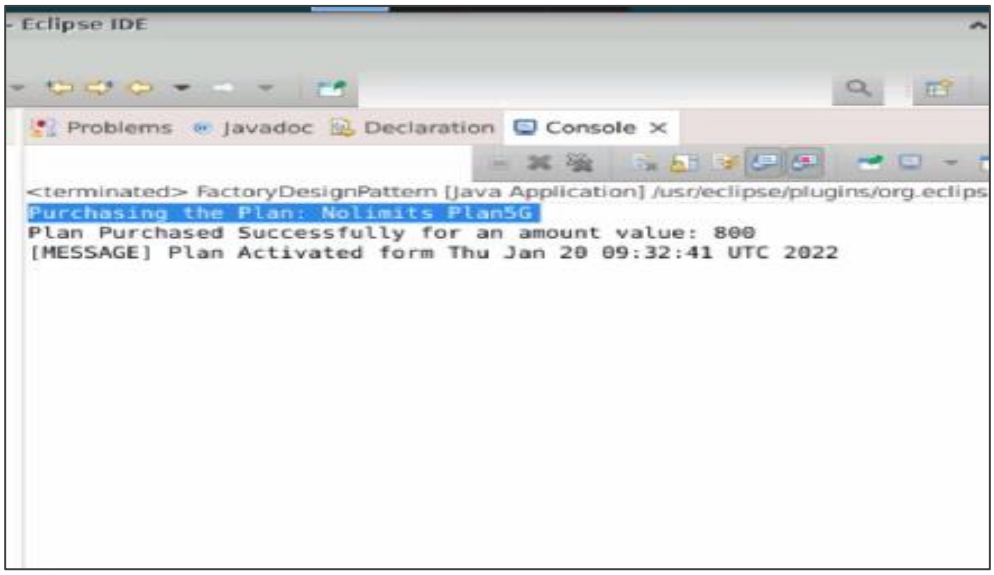
5.5 Enter the wrong amount and you will get the message of the amount mismatched.



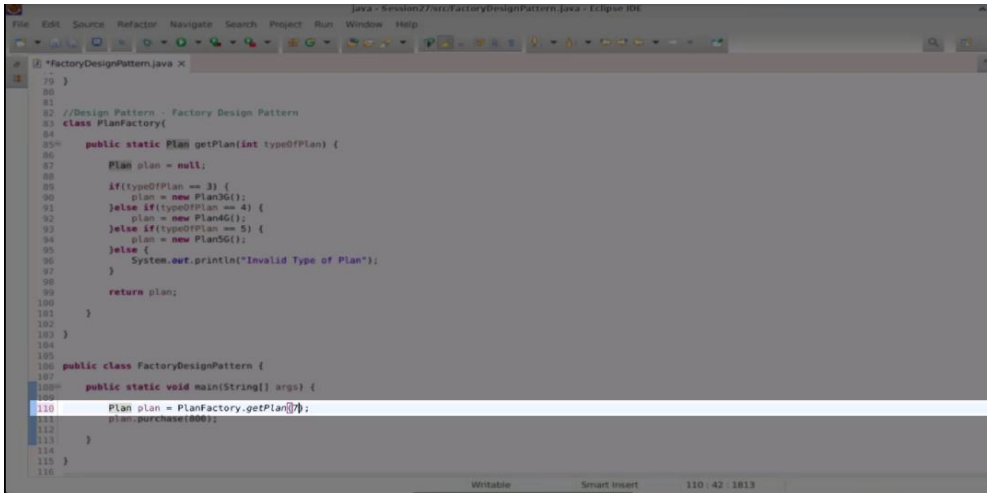


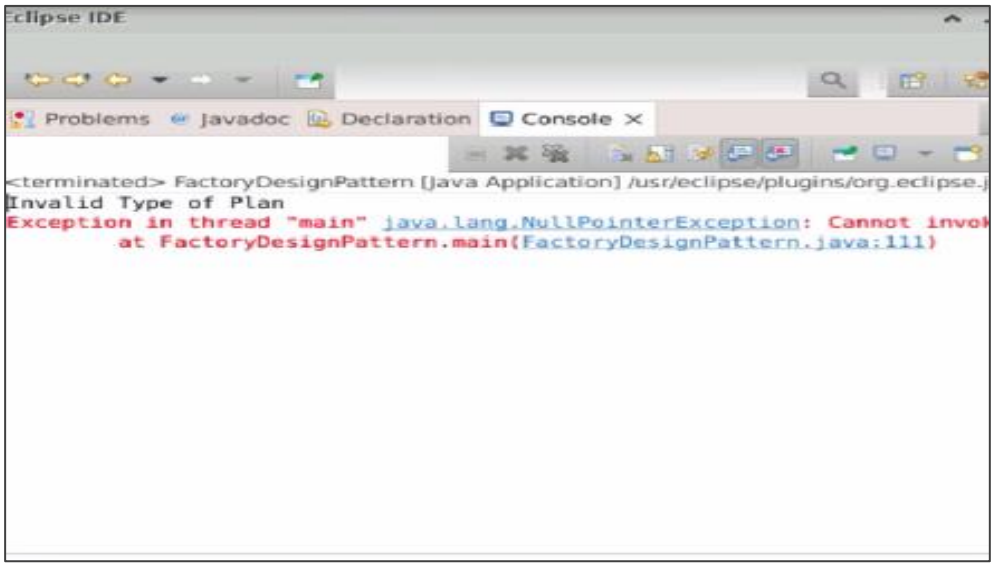
5.6 Run the code for 4G and 5G plans also by changing the amount and **typeOfplan** value. The following outputs will be obtained.



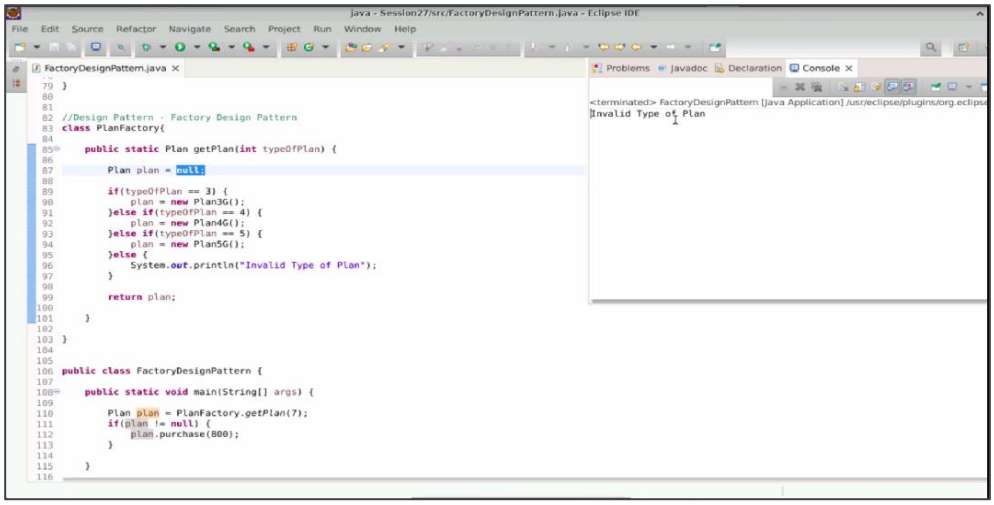


5.7 Pass the value as **7** in the getPlan method and run the code. The code will crash.



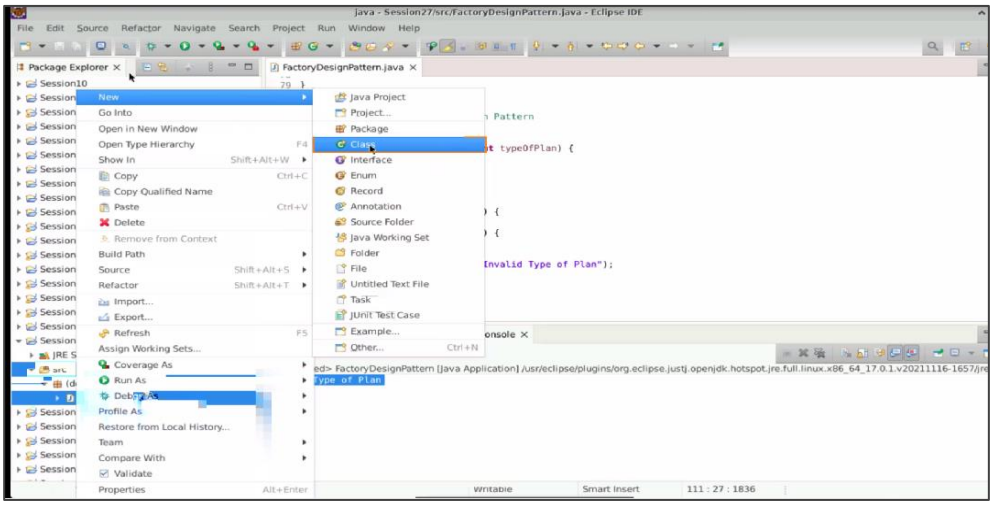


5.8 This is where you need to perform a check. The check involves verifying if your plan is not null. Only if the plan is not null, should you attempt to execute the method called plan.purchase. Initially, the plan was null. With this check in place, you won't encounter a crash and instead, you will receive a message indicating an invalid type of plan.

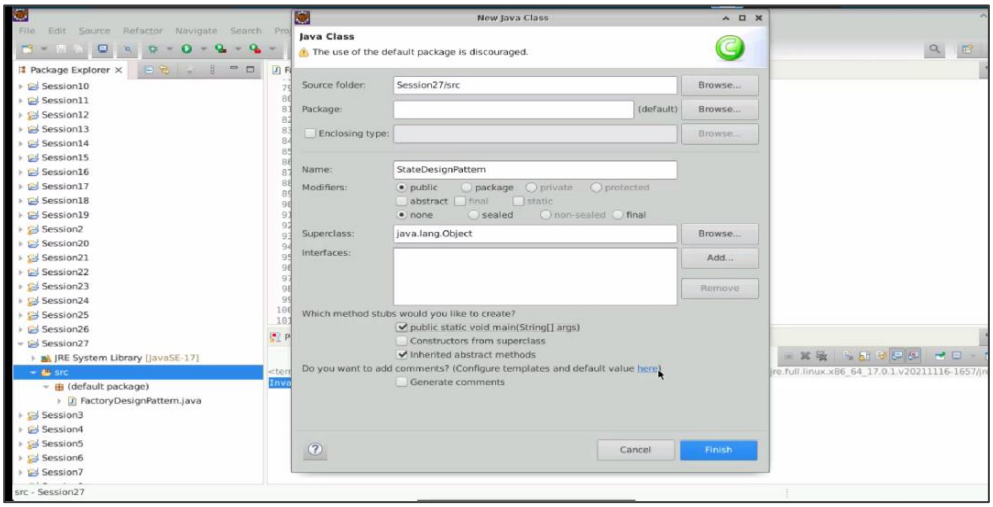


Step 6: Create State Design Patterns

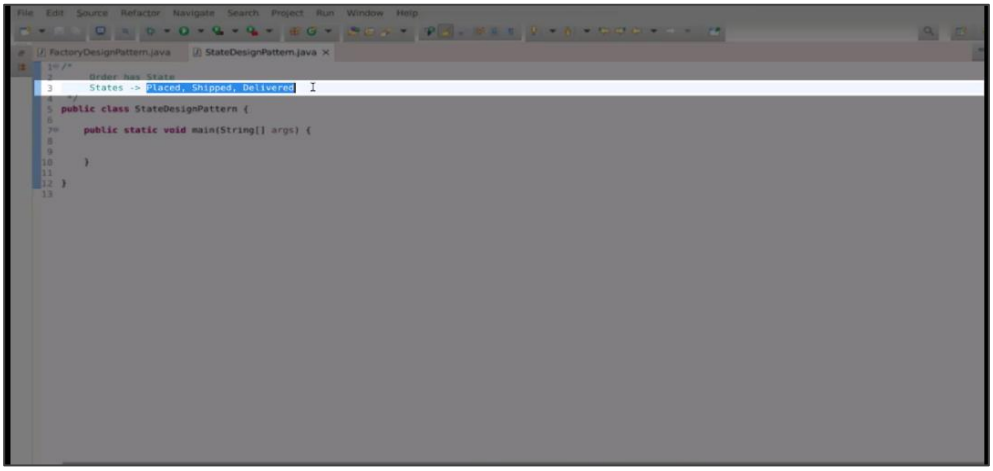
6.1 Right-click on the **src** folder under the project name, select **New**, and then select **Class**.



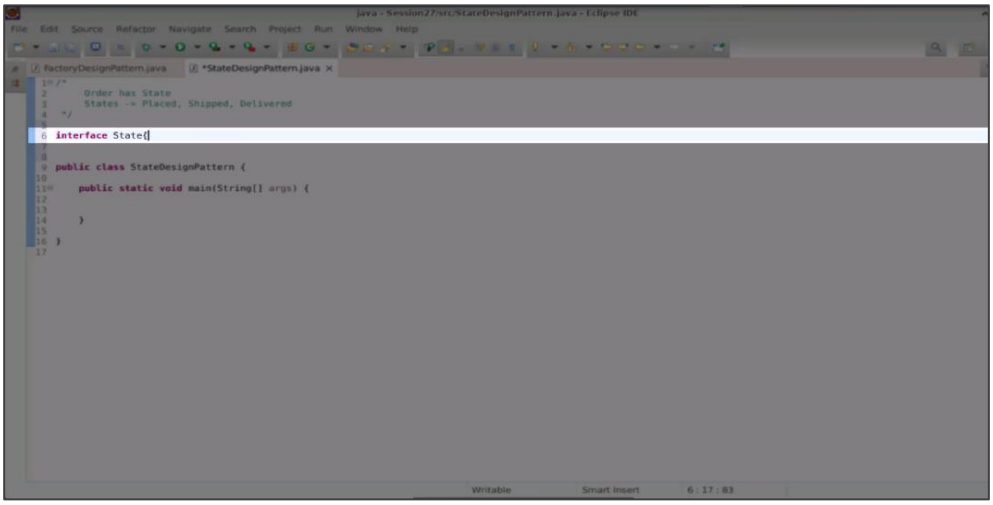
6.2 Name the class as **StateDesignPattern** and select the option **public static void main(String[] args)**. Select **Finish**. This will create a java class with a **main()** method.



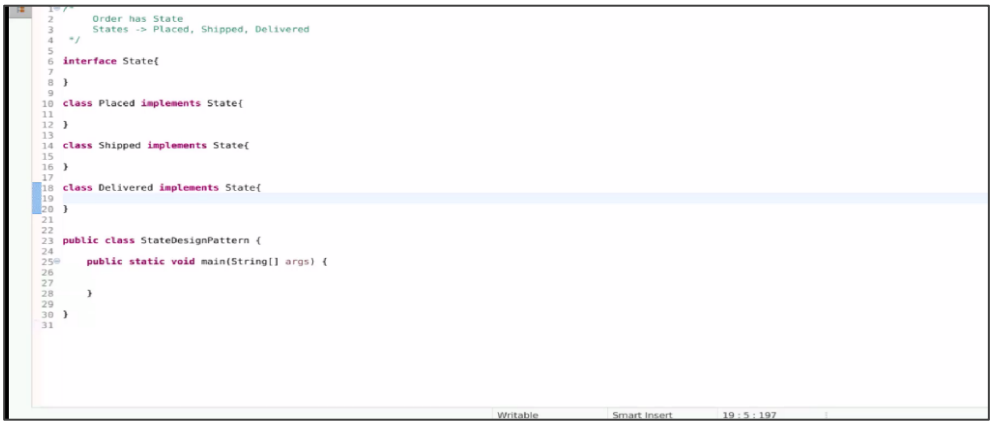
6.3 An order has various states. These states could be **placed, shipped, and delivered.**



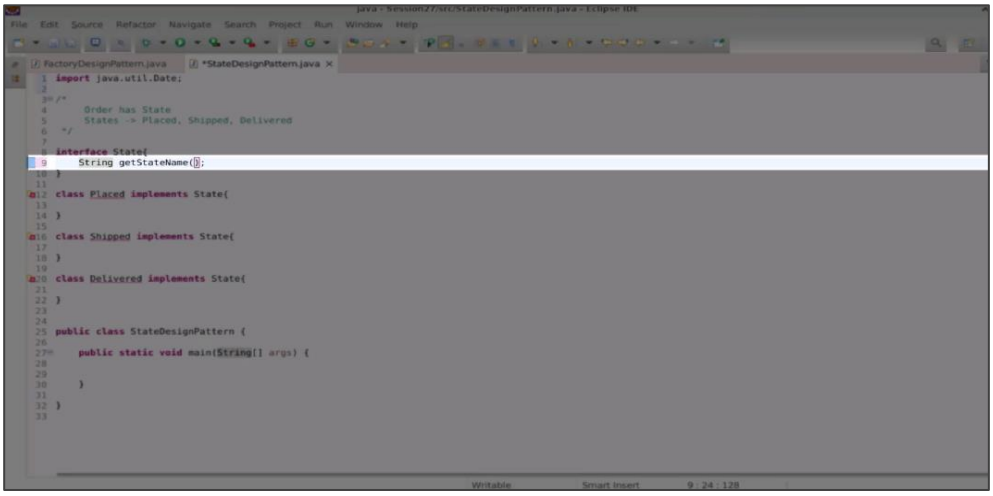
6.4 In the class **StateDesignPattern**, create an interface **State** by writing **interface State.**



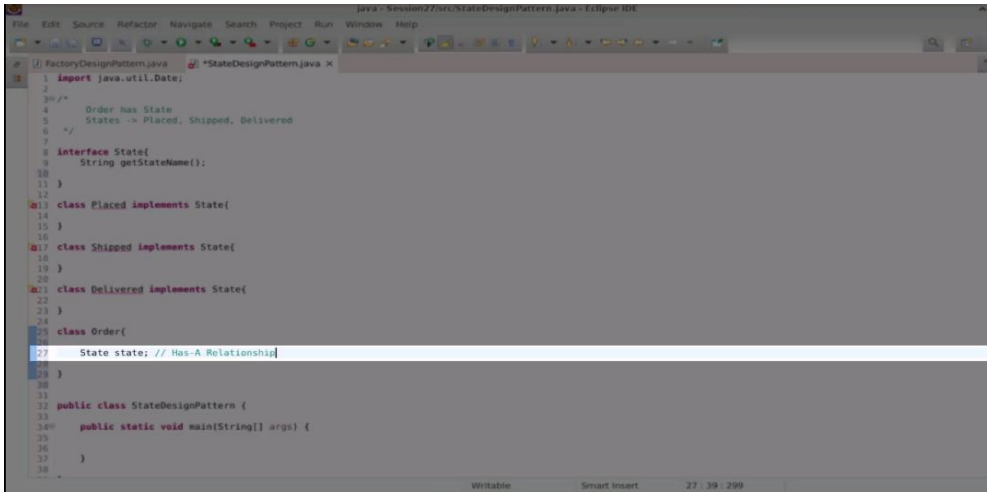
6.5 Create three classes, **class Placed, class Shipped** and **class Delivered**. All these classes must implement the interface State.



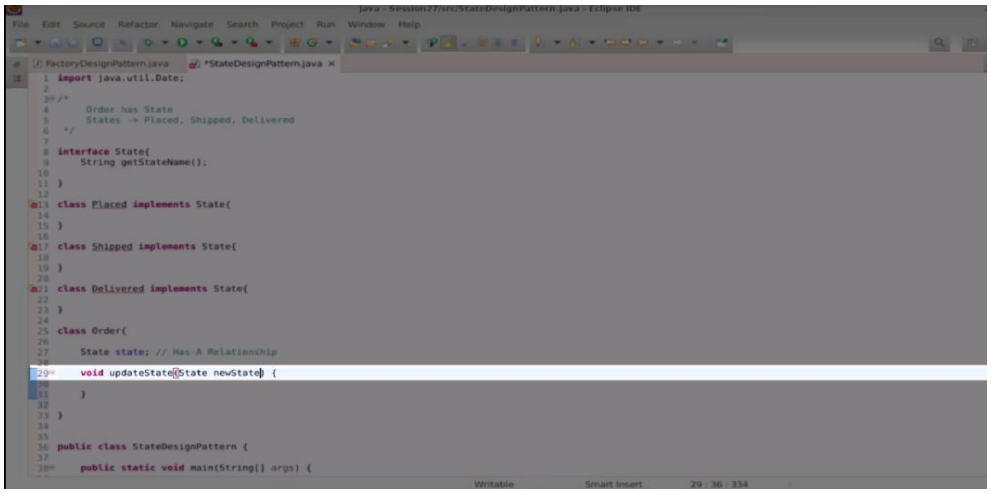
6.6 Create a method **String getStateName()** inside the interface **State.**

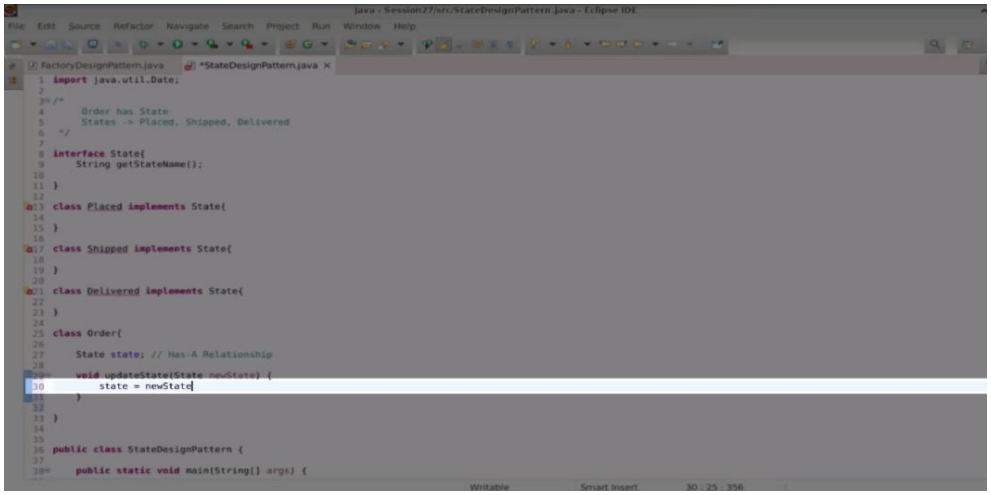


6.7 Create a new class **Order** which would have a reference to the state. This is called a **has-a-relationship**.

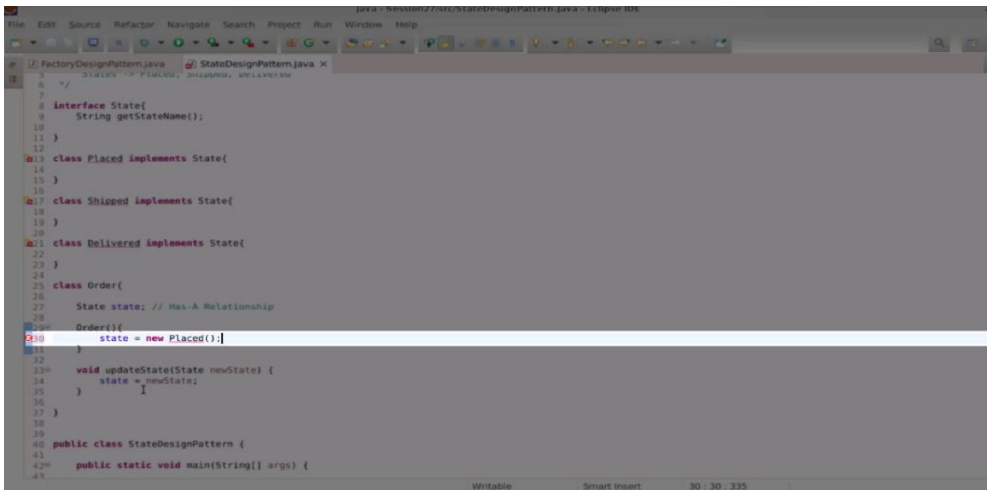


6.8 There will be a change in the state. Therefore, create a method **void updateState(State newState)** and pass newState as an input. Make the state as newState.

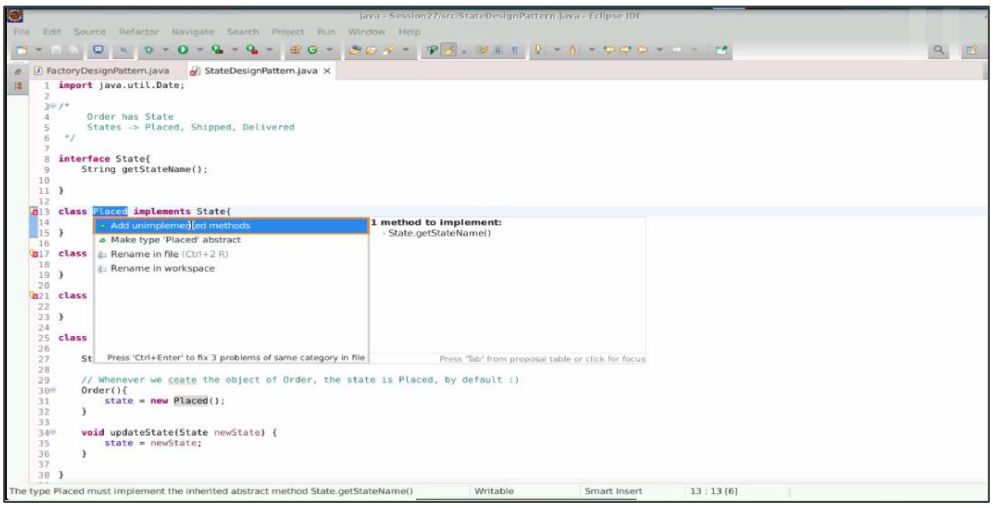


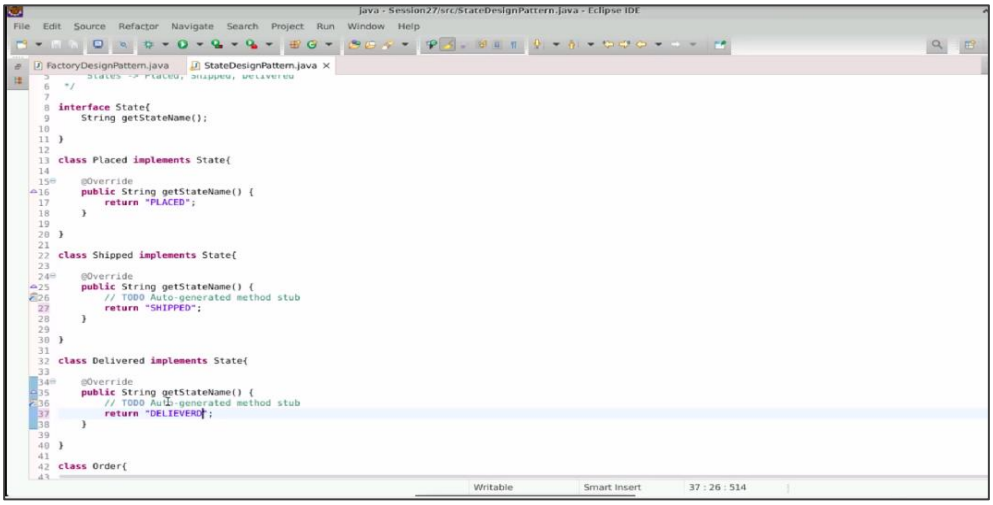


6.9 Create the constructor of class **Order** and set the state as placed so that whenever an object of order is created, the state is set to placed by default.

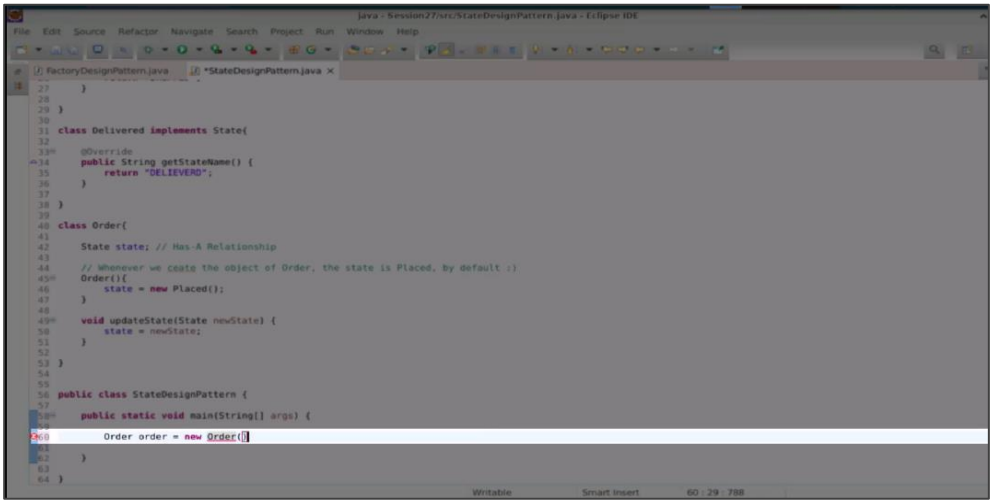


6.10 Complete the unimplemented methods of classes **Placed, Shipped and Delivered.** Return the strings **PLACED**, **SHIPPED**, and **DELIVERED** in the **getStateName** method of the three classes.

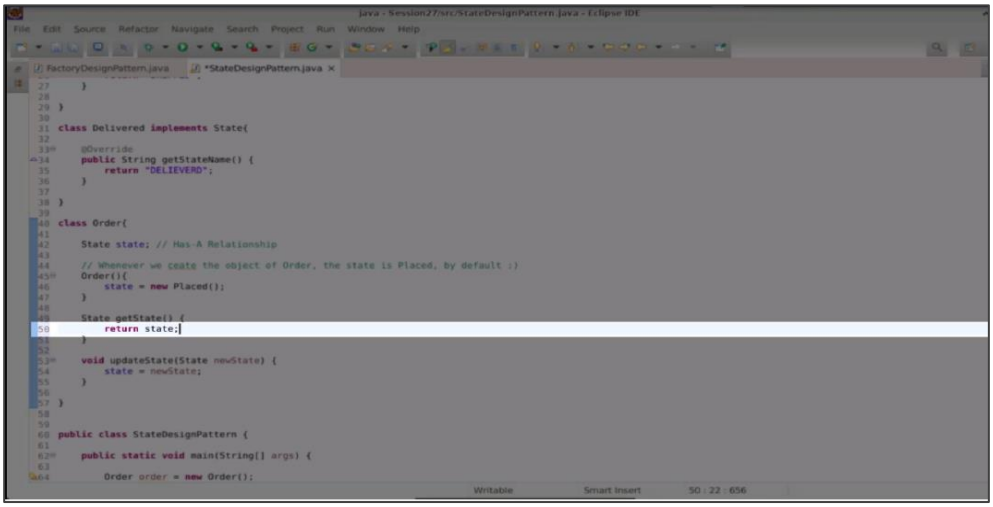




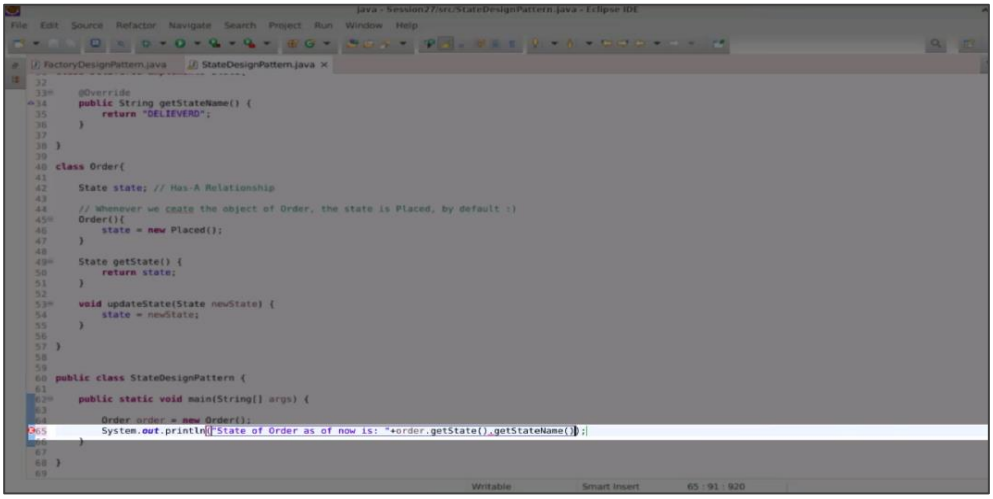
6.11 Inside the class **StateDesignPattern** create an object of class **Order**.



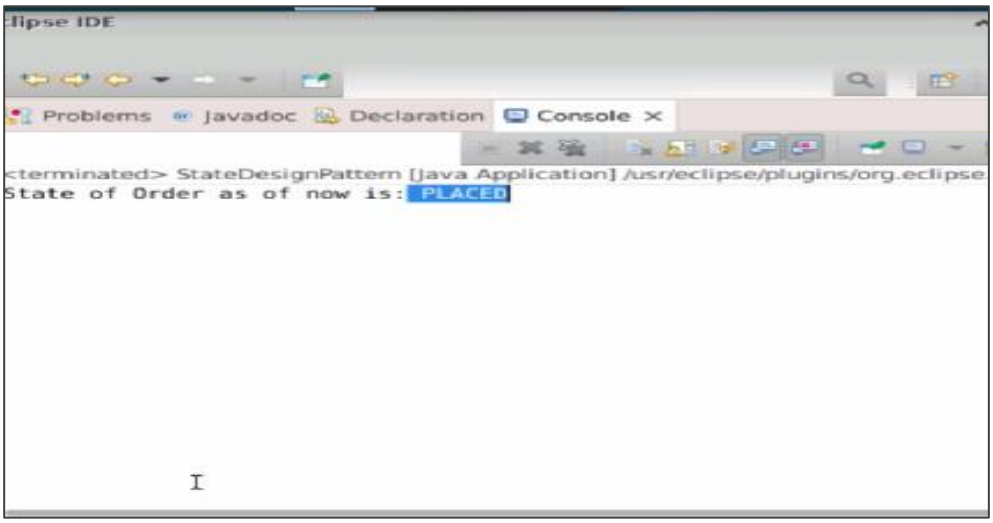
6.12 In the class Order, create a method **getState()** which returns the state of an order.



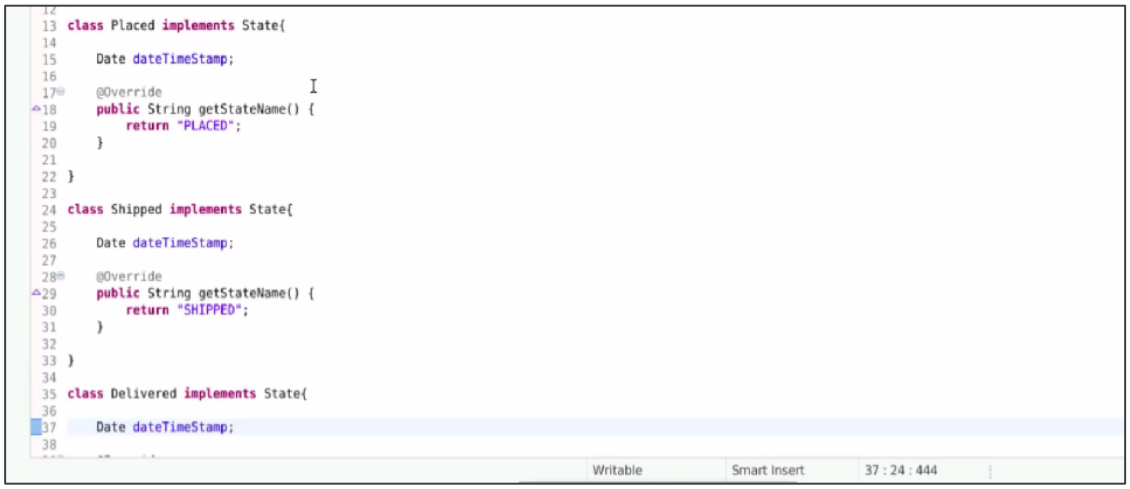
6.13 Print the state of the order as **order.getState().getStateName().**



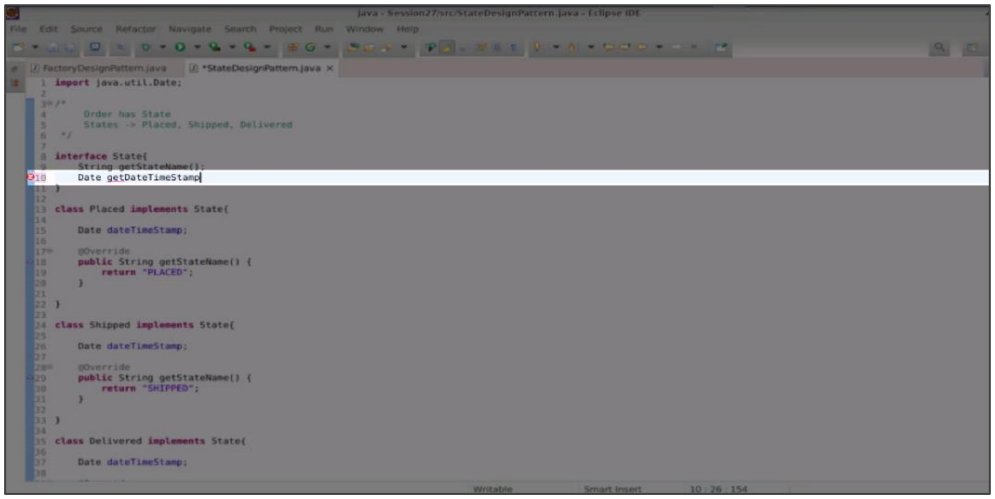
6.14 Run the code and implement the basic code of **StateDesignPattern** and obtain the following output.

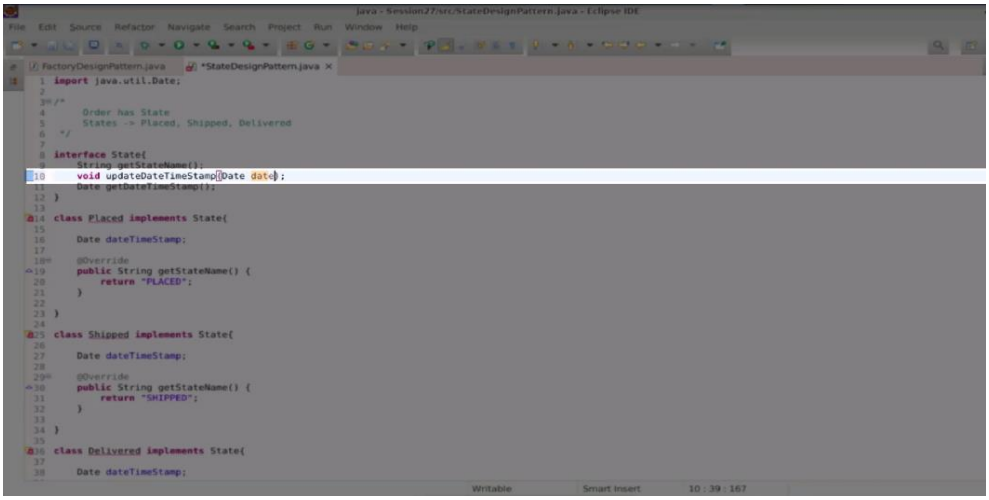


6.15 Write the statement **Date dateTimeStamp** in the three classes to record the date timestamp associated with these classes.

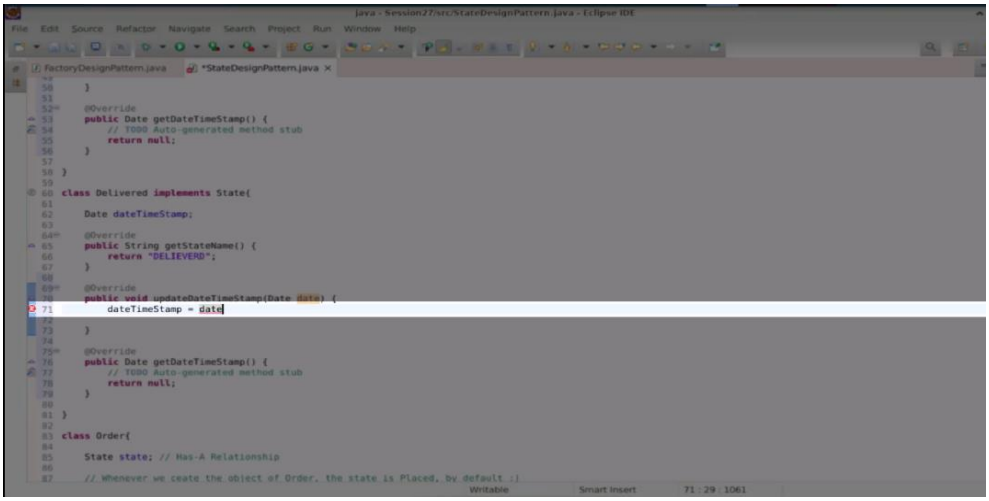


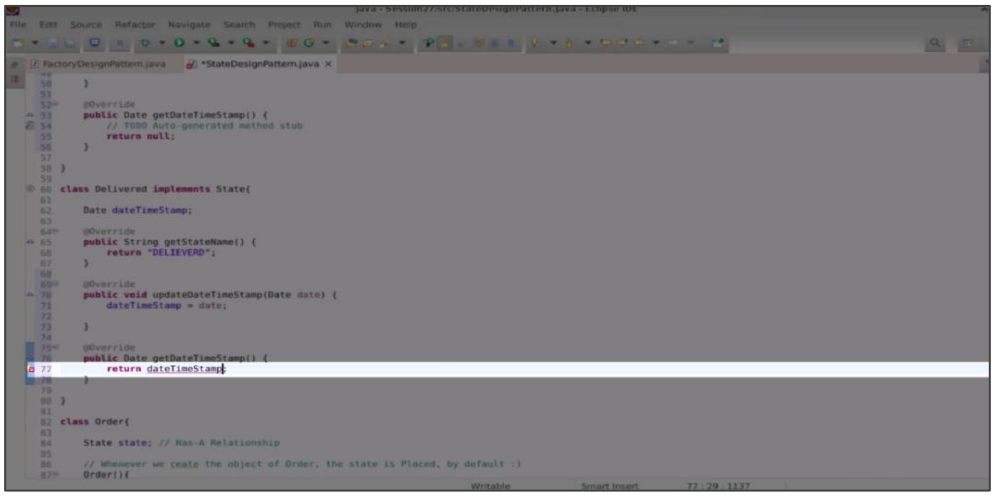
6.16 Create two more methods in the state interface. The methods are: **void updateDateTimeStamp() and Date getDateTimeStamp().**



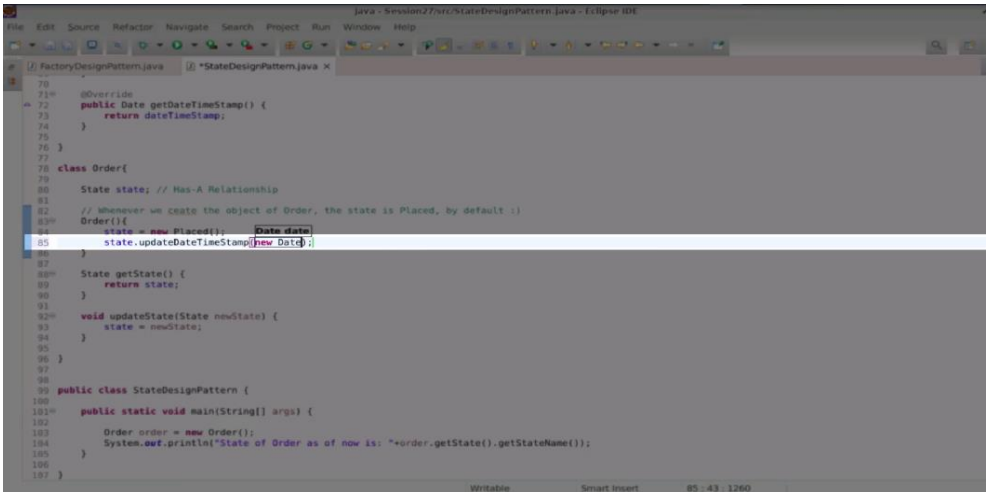


6.17 Implement these two methods inside the three classes. In the updateDateTimeStamp method, copy the date to dateTimeStamp. And in the getDateTimeStamp method, simply return the dateTimeStamp attribute.

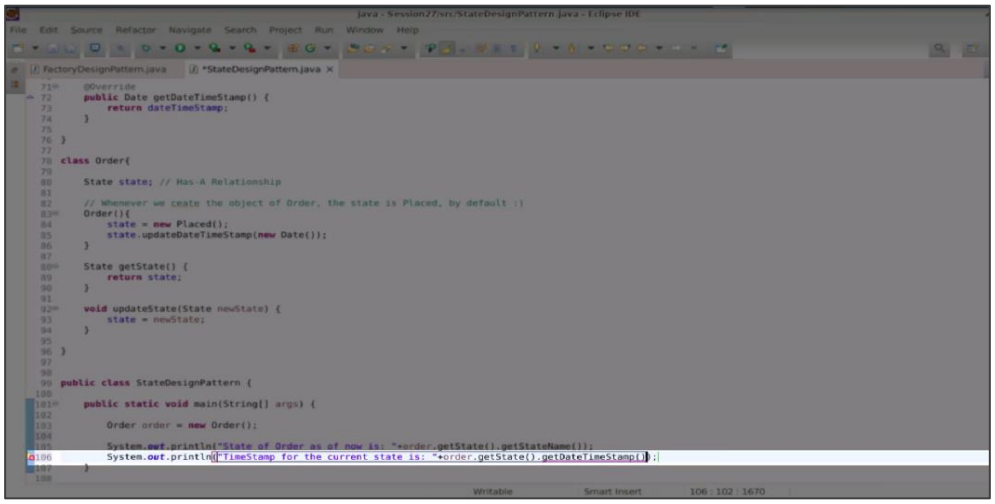




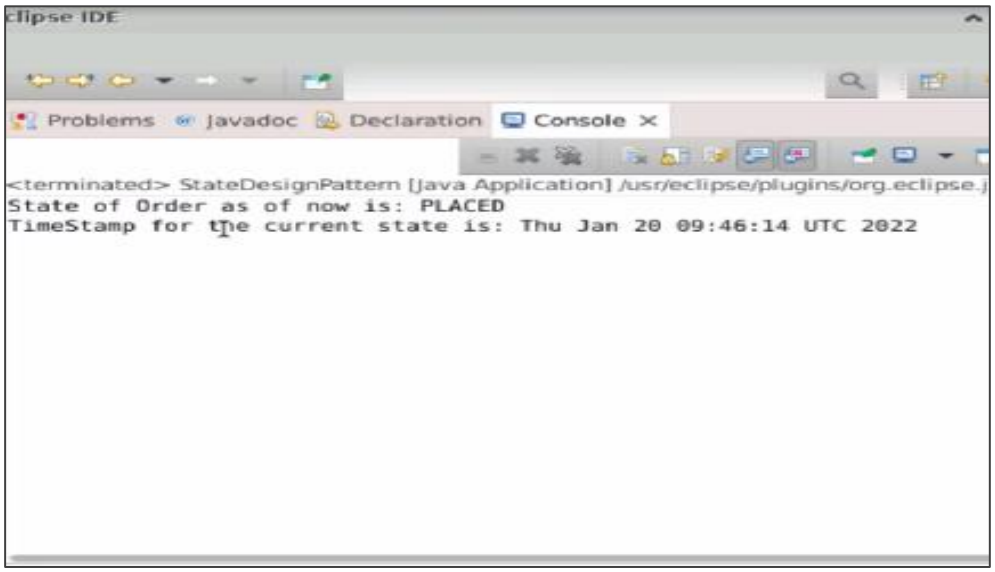
6.18 Create an object of date to display the current date when you place an order by writing **state.updateDateTimeStamp(new Date)** in the constructor **Order.**



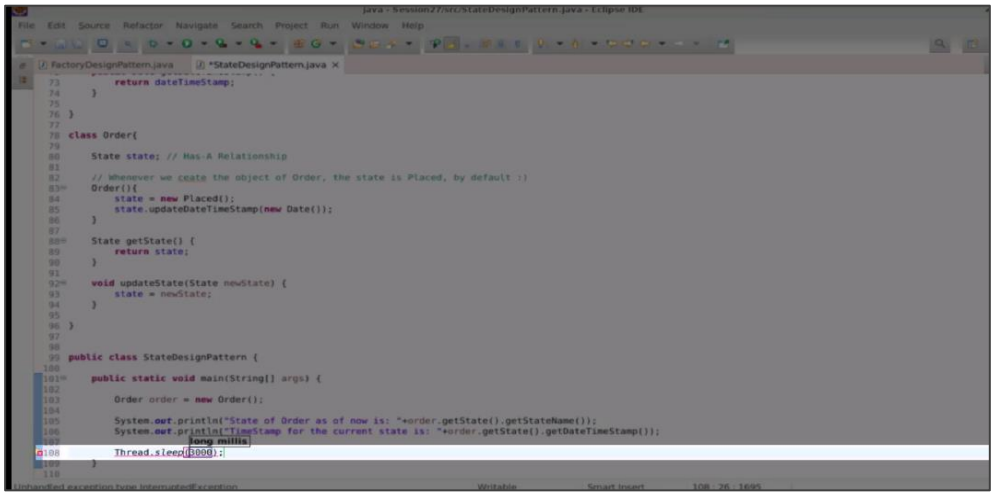
6.19 Check the state of your order and date timestamp by calling **order.getState().getDateTimeStamp().**



6.20 Run the code and the following output will be obtained.

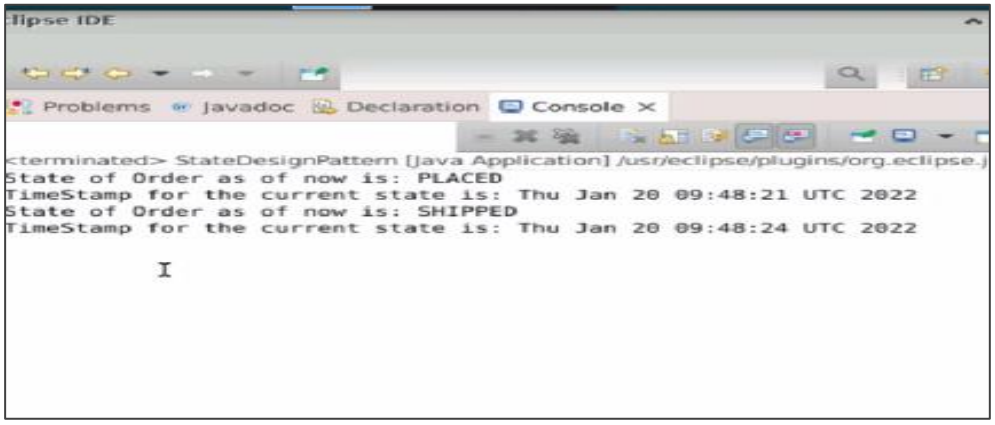


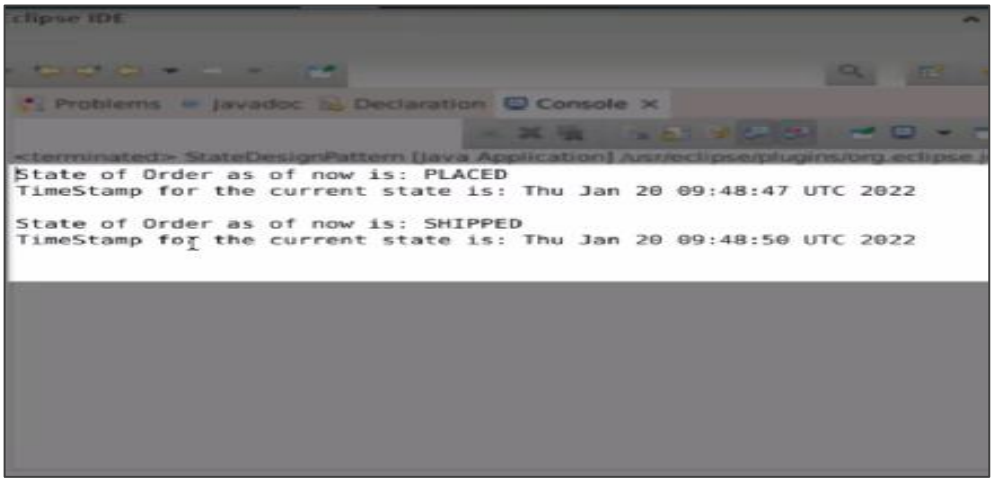
6.21 Introduce a delay of 3 seconds by writing Thread.sleep(3000). Surround this by try catch block by clicking on the error bulb.



6.22 Create a new state called shipped state and update the date timestamp and give once again a new object of date. Then update the state to this new state by writing order.updateState(state). Copy paste the two print statements above.

6.23 Run the code.





By following the above steps, you have successfully implemented the Factory and State design patterns. These patterns enhance flexibility, reusability, and scalability in your software, creating a robust and efficient solution.