Used Car DealerShip

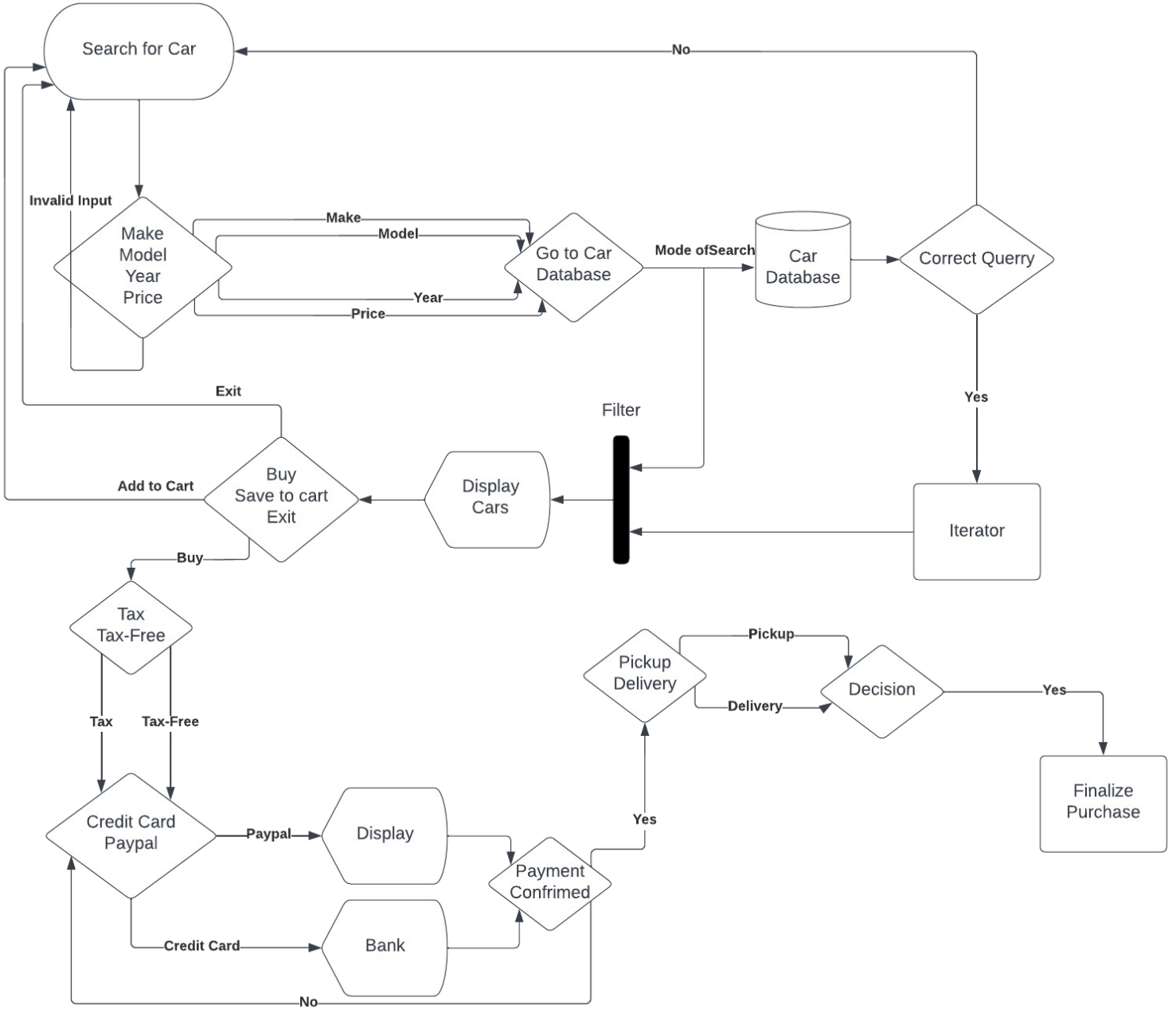
By Team 4

- Jarosław Rybak
- Mohammad Ariz Haider
- Satar Hassni
- Anthony Ferrara
- Mohammad U Uddin
- Fahim Ahmed

Project Description

A system capable to keeping, displaying, and editing an inventory of cars for the purpose of selling them.

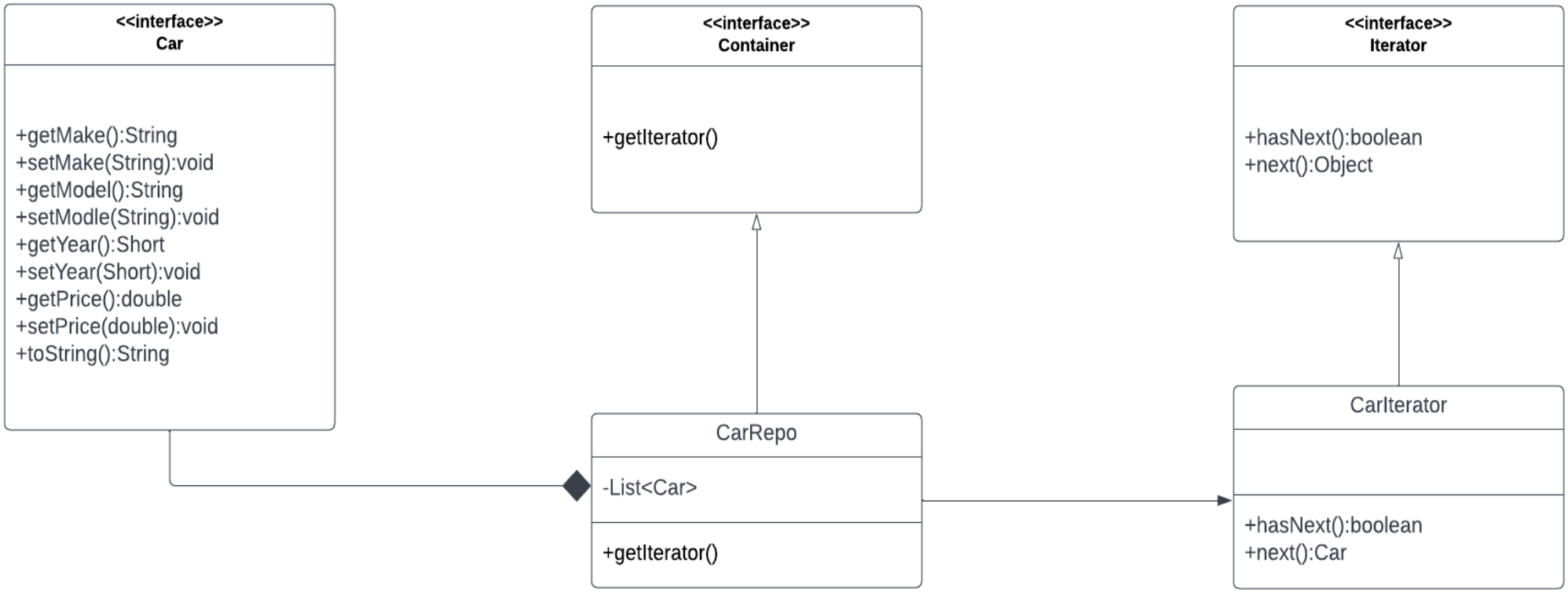
Flow Chart


Search for Car
No
Invalid Input
Make
Model
Year
Price
Make
Model
Year
Price
Go to Car Database
Mode ofSearch
Car Database
Correct Querry
Yes
Exit
Filter
Add to Cart
Buy
Save to cart
Exit
Display Cars
Iterator
Buy
Tax
Tax-Free
Tax
Tax-Free
Pickup
Delivery
Pickup
Delivery
Decision
Yes
Finalize Purchase
Credit Card
Paypal
Paypal
Display
Yes
Payment Confrimed
Credit Card
Bank
No

Used Car Dealership
Iterator Pattern

By Jarosław Rybak

Iterator Pattern UML Diargram



Iterator Pattern Features

- No matter what kind of car it is, as long that it is a car, it will be output from the next() method.
- In this example it is using any class that implements the List interface.


```
2
3  public interface Iterator
4  {
5      public boolean hasNext();
6      public Object next();
7  }
```

```
2
3  public interface Container
4  {
5      public Iterator getIerator();
6  }
7
```

```
public interface Car
{
    public String getMake();
    public void setMake(String make);
    public String getModel();
    public void setModel(String model);
    public short getYear();
    public void setYear(short year);
    public double getPrice();
    public void setPrice(double price);
    public String toString();
}
```

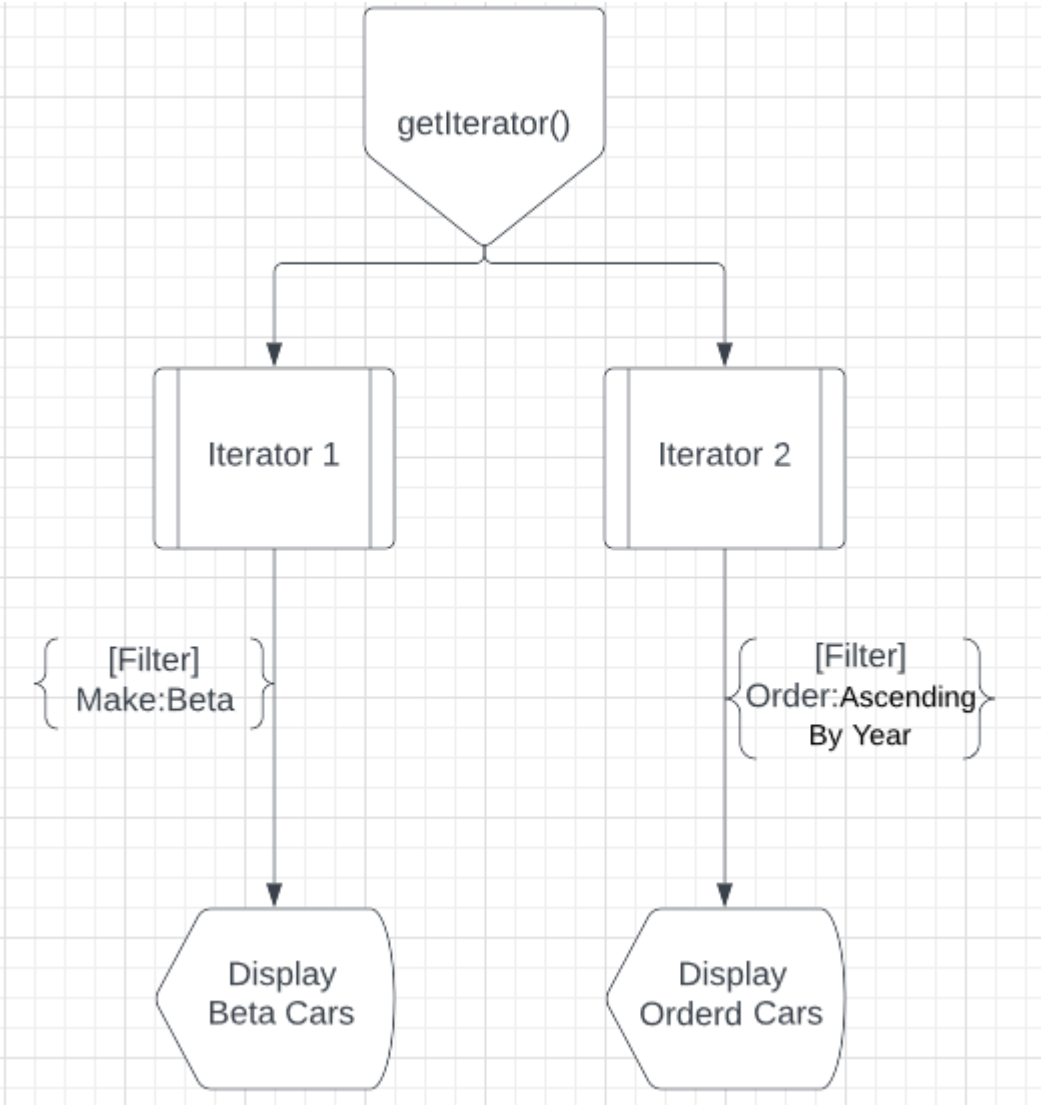
All the is relavent to the user

- Only 3 parts are relavent to the user.
- 1: getIterator() to make an instance.
- 2: hasNext() to check for the end of the list.
- 3: next() to output the current car be ready for the next car.

```
@Override
public Iterator getIerator()
{
    return new CarIterator();
}

private class CarIterator implements Iterator
{
    int index;

    @Override
    public boolean hasNext()
    {
        if(index < Cars.size())
        {
            return true;
        }
        return false;
    }

    @Override
    public Car next()
    {
        if(this.hasNext()){
            return Cars.get(index++);
         }
        return null;
    }
}
```

Component Testing

Make 2 different instances of the Iterator. Use them separately for different purposes without one interfering with the other. Iterator 1 will filter Cars from “Make”: “Beta” and return them. Iterator 2 will return Cars in ascending order of year. Neither will know of the other nor have an effect on the others output.

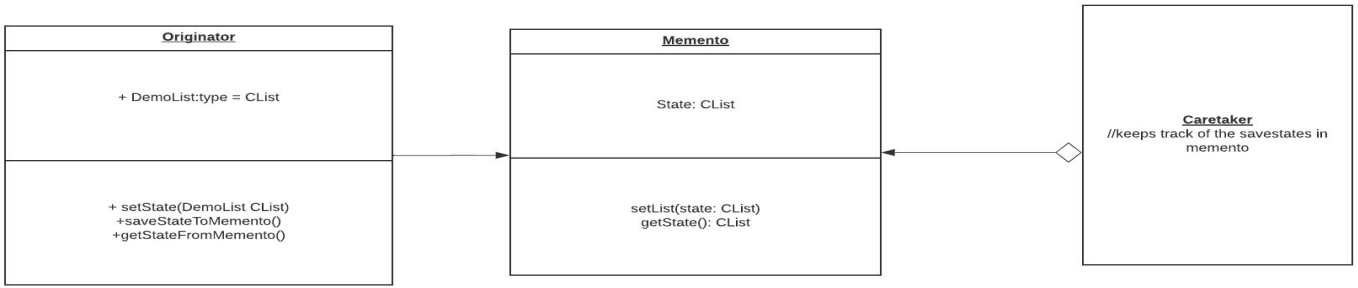


USED CAR DEALERSHIP

MEMENTO PATTERN

By
MOHAMMAD ARIZ HAIDER

MEMENTO PATTERN: UML



Features of Memento

01 Able to save the state of an object in one or in as many states (checkpoints) as required

02 Able to restore the object back to a previous state, if any mistakes were made or i original state was required

3 Key things the Client needs to know

```java
package Memento;
public class Originator //Creates the save time
{
    DemoList CList; //CList = Car List

    public DemoList getState()
    {
        return CList;
    }

    public void setState(DemoList  CList) // saves a sate of the current list
    {
        this.CList = CList;
    }

    public Memento saveStateToMemento()
    {
        return new Memento(CList);
    }

    public void getStateFromMemento(Memento memento)// loads  or revert back saved state
    {
        CList = memento.getState();
    }
}
```

```java
package Memento;
public class Memento //save state
{
    private DemoList CList;

    public Memento (DemoList CList)
    {
        super();
        this.CList = CList;
    }
    public DemoList getState()
    {
        return CList;
    }
    public void setList(Memento memento)
    {
        CList = memento.getState();
    }

    public String toString()
    {
        return "Add the Cars to the List: " + CList;
    }
}
```

```java
package Memento;
import java.util.ArrayList;
import java.util.List;

public class CareTaker //adds more save states as list is changed
{
    private List<Memento> CarList = new ArrayList<Memento>();
    private List<Memento> CopyCarList = (List)(((ArrayList)CarList).clone());

    public void add(Memento mem)
    {
        CarList.add(mem); //
        System.out.println("List of Cars:" + CarList + "\n");
    }

    public Memento get(int index)
    {
        return CarList.get(index);
    }
}
```

```java
@Test
public void saveSate() //FIrst Test : saves a state
{
    Org.setState(CList); //adds a default state
}
```

1. Originator saves the default state of the object, in this case, CList (Car List) to Memento

2. This is where the states of the object are kept,

```java
@Test
public void test2() //Checks if it can go back to last saved state
{
    CareT.add(Org.saveStateToMemento());
}
```

3. CareTaker keeps track of how many states are saved in Memento. It also used for reloading a previously saved state, which was saved by the Originator

Component Testing

A list of cars exist for the user to buy (gets removed from the list) and sell, (gets added to the car list). The List is first a default state (untouched) which exists in the Originator, it then is saved as a Memento (Save File), and the CareTaker keeps track of how many different save files are added or removed (keeps track of states). If a car is bought by accident and the list must be return to previous state, the CareTaker takes the saved state in the Memento which has the index no. of state before purchase and restores the list to that state.

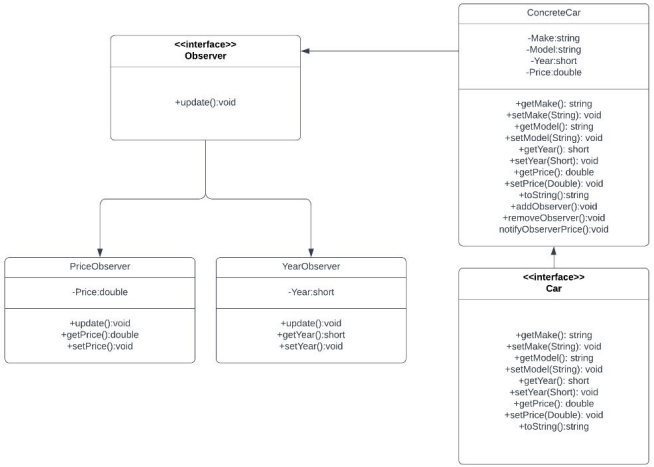
Observer Pattern

Anthony Ferrara
CSCI 370

What is the observer pattern?

- In software design and engineering, the observer pattern is a software design pattern in which an object, named the subject, maintains a list of its dependents, called observers, and notifies them automatically of any state changes, usually by calling one of their methods.
- In my code, the “Car” class notifies two observers about changes in the price of the car, and changes in the cars year. It does this by calling the update() method, which is defined inside of an interface.
- The observer pattern has many advantages. It supports the principle of loose coupling between objects that interact with each other. It allows sending data to other objects effectively without any change in the Subject or Observer classes. Observers can be added/removed at any point in time.

UML Class Diagram



Unit Tests

- Unit testing ensures that all code meets quality standards before being deployed.
- The most popular testing framework for Java is JUnit5, which is what is used in this project.
- The first test creates a new Car class, and adds a priceObserver to it. Then, we call the notifyObserverPrice() method, and pass in a new price, which updates the original car class, and the observers.
- The second test creates a new Car class, and adds a yearObserver to it. Then, we call the notifyObserverYear() method, and pass in a new year, which updates the original car class, and the observers.

```
@Test
public void doesReturnPrice(){
    ConcreteCar subject = new ConcreteCar( make: "Ford", model: "Focus", (short) 2020, price: 25000);
    PriceObserver observer = new PriceObserver();
    subject.addObserver(observer);
    subject.notifyObserversPrice(23000);
    Assertions.assertEquals( expected: 23000, observer.getPrice());
}
@Test
public void doesReturnYear(){
    ConcreteCar subject = new ConcreteCar( make: "Ford", model: "Focus", (short) 2020, price: 25000);
    YearObserver observer = new YearObserver();
    subject.addObserver(observer);
    subject.notifyObserversYear((short) 2021);
    Assertions.assertEquals( expected: 2021, observer.getYear());

}
```

Component Tests

- The primary objective of component testing is to validate the behavior of the individual component, as specified in the requirements document.
- For this project, I have written the following component test:

Create two observer classes, one that observes the car's year of production, and one that observes the car's price. When these values change, they are passed to the observers via the ConcreteCar class. Add an instance of either observer to the array list located within the ConcreteCar class. Then, use the getter methods located within each observer class, (priceObserver and yearObserver) to retrieve the information.

Strategy Pattern

Presented by
Fahim Ahmed
CSCI 370 46 [28015]

Strategy Pattern – What is it?

Type

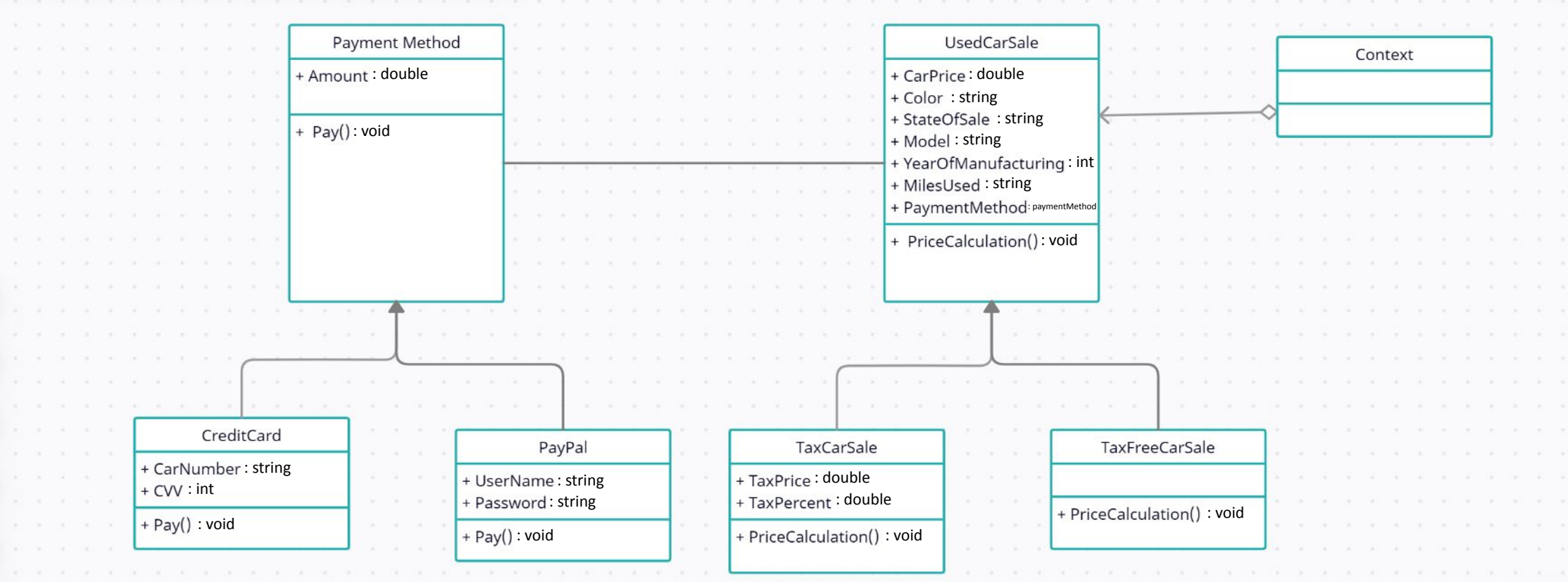
Behavioral Pattern

Objective

To change class behavior or algorithm at runtime.

Approach

Objects are created using different strategies. A context object is basically created whose behavior can be changed as per its concrete strategy object. The concrete object decides behavior of the context object.



Concrete Classes

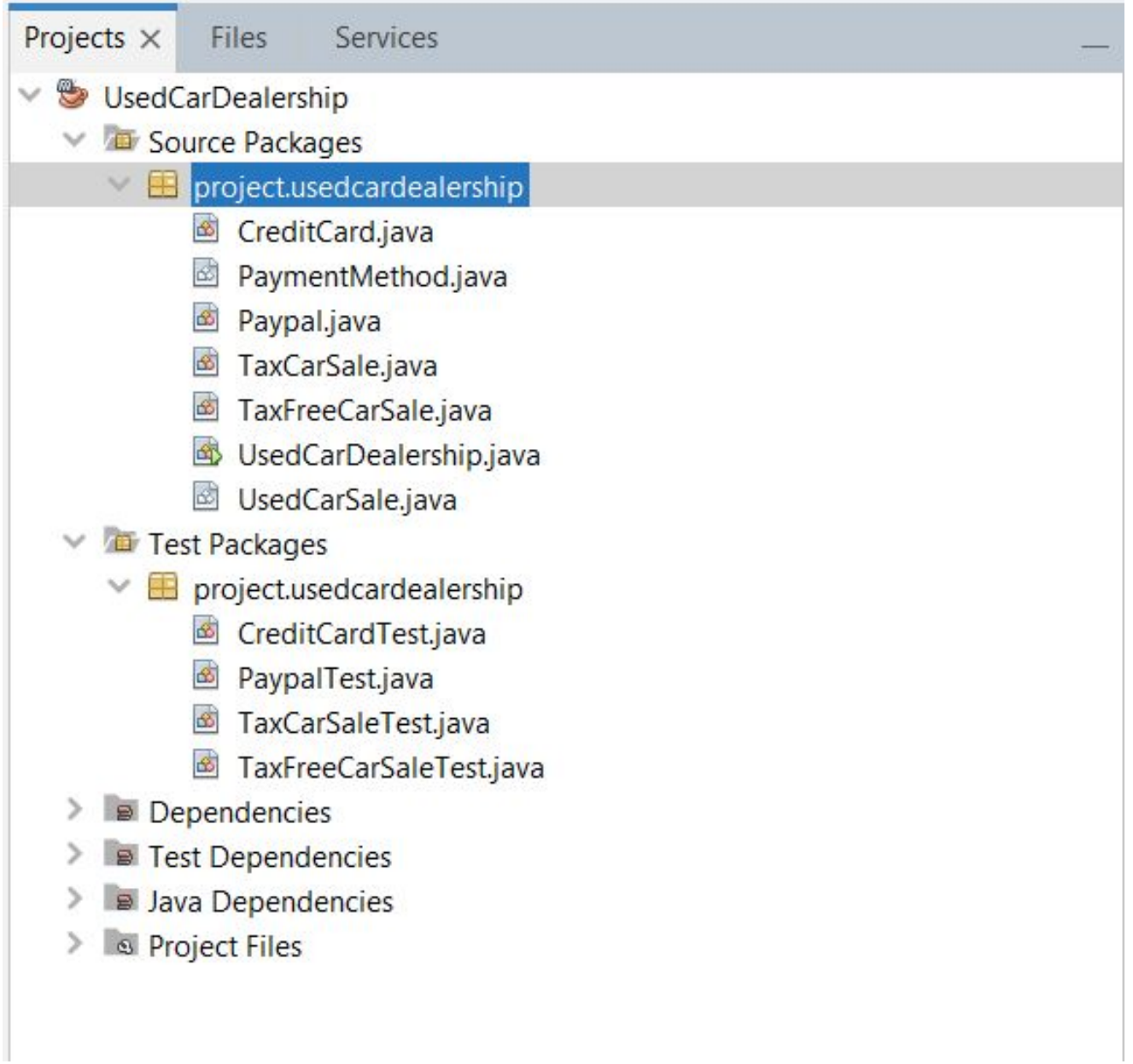
Credit Card and Paypal for Payment Method

Concrete Classes

Tax Car Sale and Tax-Free Car Sale for Used Car Sale

UML Class Diagram

Code Structure



Code - Sale

```
public abstract class UsedCarSale {
   public double CarPrice = 0;
   public String Color = "Red";
   public int YearOfManufacturing = 2020;
   public String Model ="AB231";
   public String StateOfSale ="Ohio";
   public PaymentMethod PaymentType;
public abstract void PriceCalculation();
}
public class TaxCarSale extends UsedCarSale{

  public double TaxPercent;
  public double TaxPrice;
  @Override
  public void PriceCalculation() {
    if(TaxPercent == 0)
    {
       System.out.println("Tax cannot be zero percent. Go For Tax Free Sales"); return;
    }
    TaxPrice = CarPrice*TaxPercent/100;
    CarPrice = TaxPrice+CarPrice;
     System.out.println("Price To Be Paid With Tax: "+ CarPrice);
  }
}
public class TaxFreeCarSale extends UsedCarSale{

  @Override
  public void PriceCalculation() {
    System.out.println("Price To Be Paid for Tax Free Car Sale: "+ CarPrice);
  }
}
```

Code – Payment Methods

```
    {
        if(UserName.matches("^(.+)@(.+)$"))
        System.out.println(Amount+" Paid by PayPal User: "+UserName);

        else{
                System.out.println("Username must be a valid email. Invalid Username: "+UserName);

        }
    }
}
```

```
public class CreditCard extends PaymentMethod{
    public int CVV;
    public String CardNumber;

        public void pay()
    {
        if(CardNumber.length() == 16){
            if(CVV <99 &&CVV>999){
        System.out.println(Amount+" Paid by Credit Card Number: "+CardNumber);
            }
            else{
```

Unit Test 1 : Tax Car Sale Strategy

```
41      */
42      @Test
43      public void testPriceCalculation() {
44          System.out.println( x: "PriceCalculation");
45
46
47          TaxCarSale instance = new TaxCarSale();
48          instance.TaxPercent= 4;
49          instance.CarPrice = 1000;
50          instance.Color= "White";
51          instance.PriceCalculation();
52      }
53
54  }
55
```

project.usedcardealership.TaxCarSaleTest › testPriceCalculation ›

Test Results ×

project:UsedCarDealership:jar:1.0-SNAPSHOT (Unit) ×

Tests passed: 100.00 %

The test passed. (0.0 s)

project.usedcardealership.TaxCa

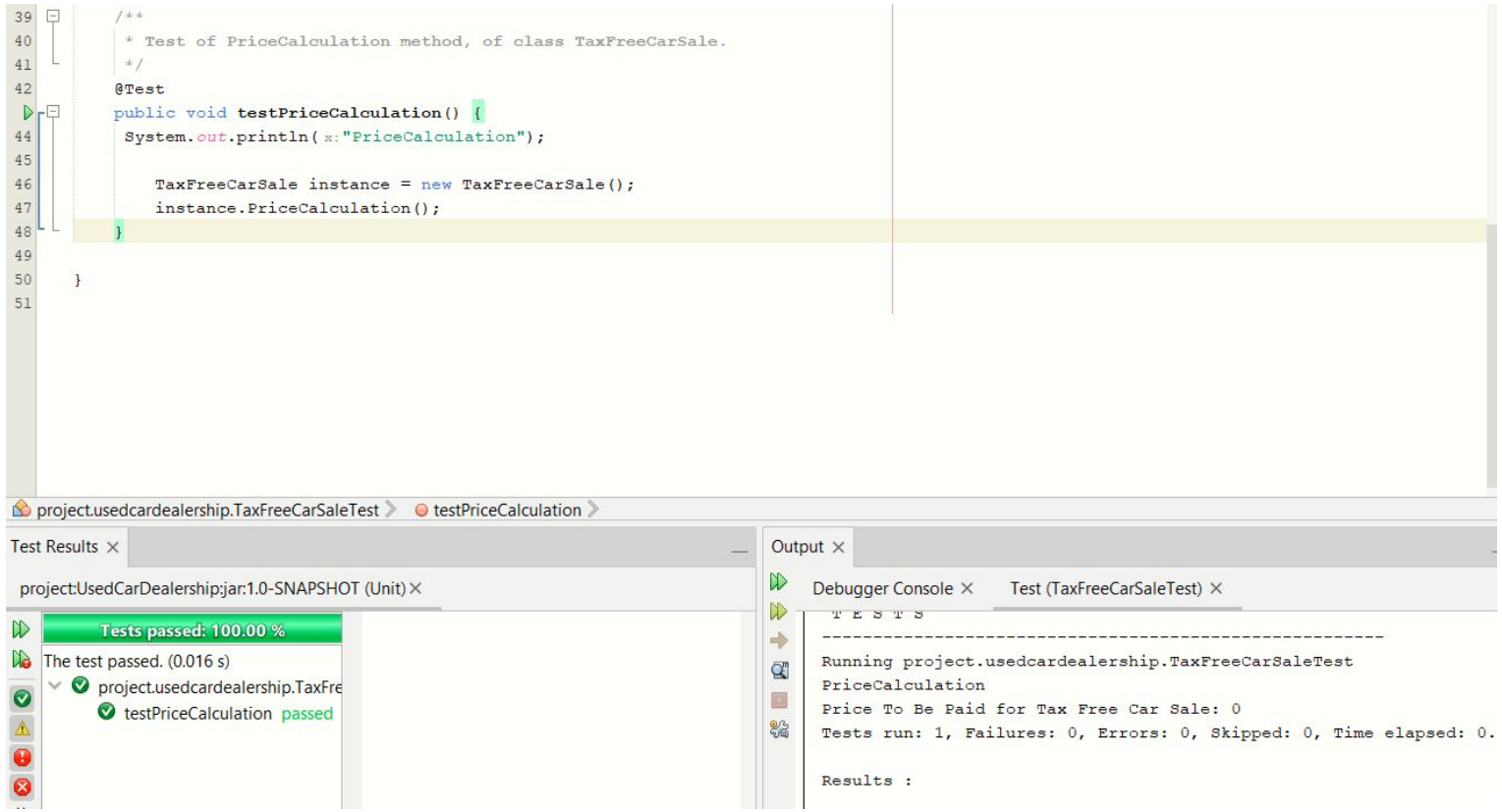
testPriceCalculation passed

Output ×

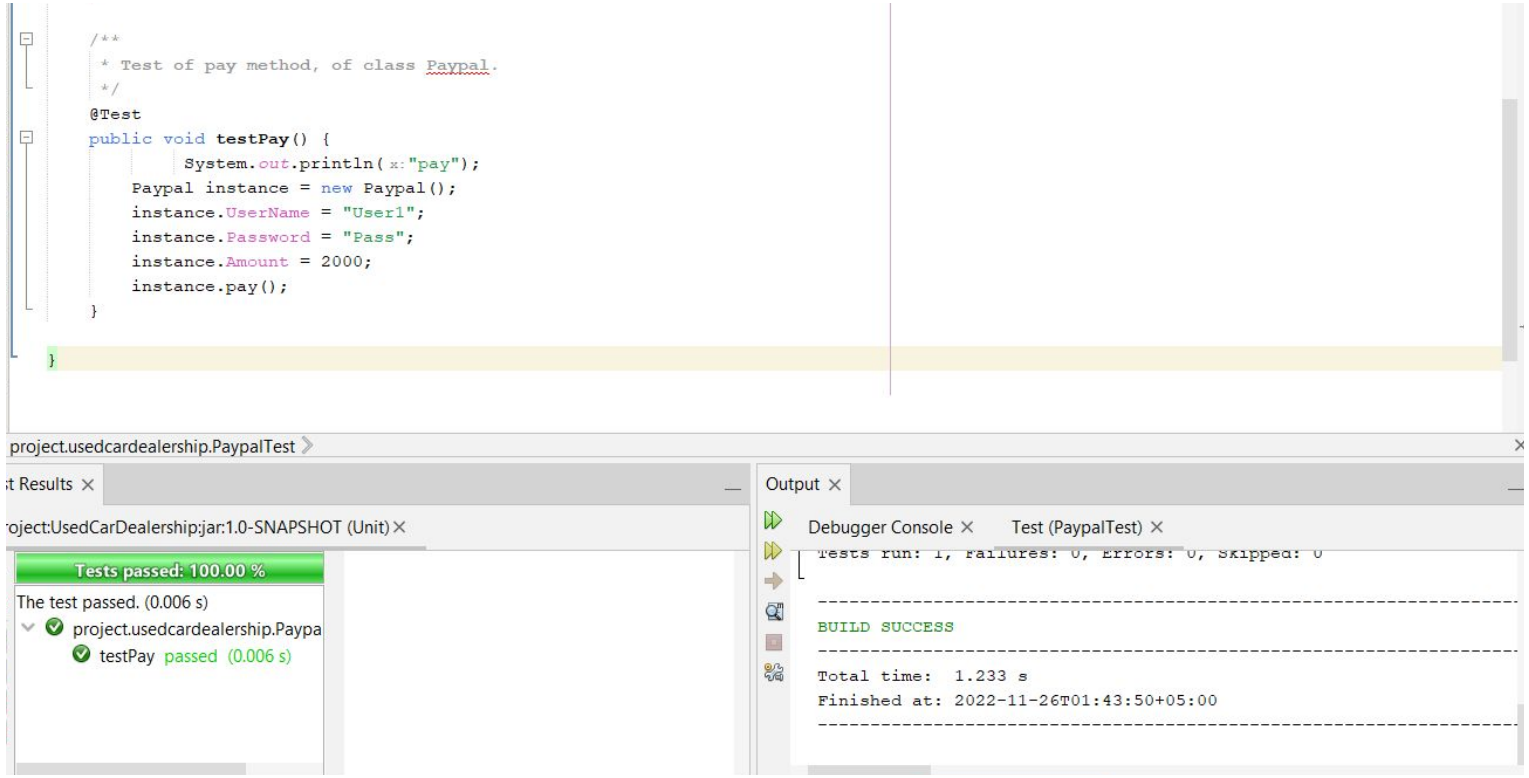
Debugger Console × Test (TaxCarSaleTest) ×

```
 T E S T S
-------------------------------------------------------
Running project.usedcardealership.TaxCarSaleTest
PriceCalculation
Price To Be Paid With Tax: 1040
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.(

Results :
```

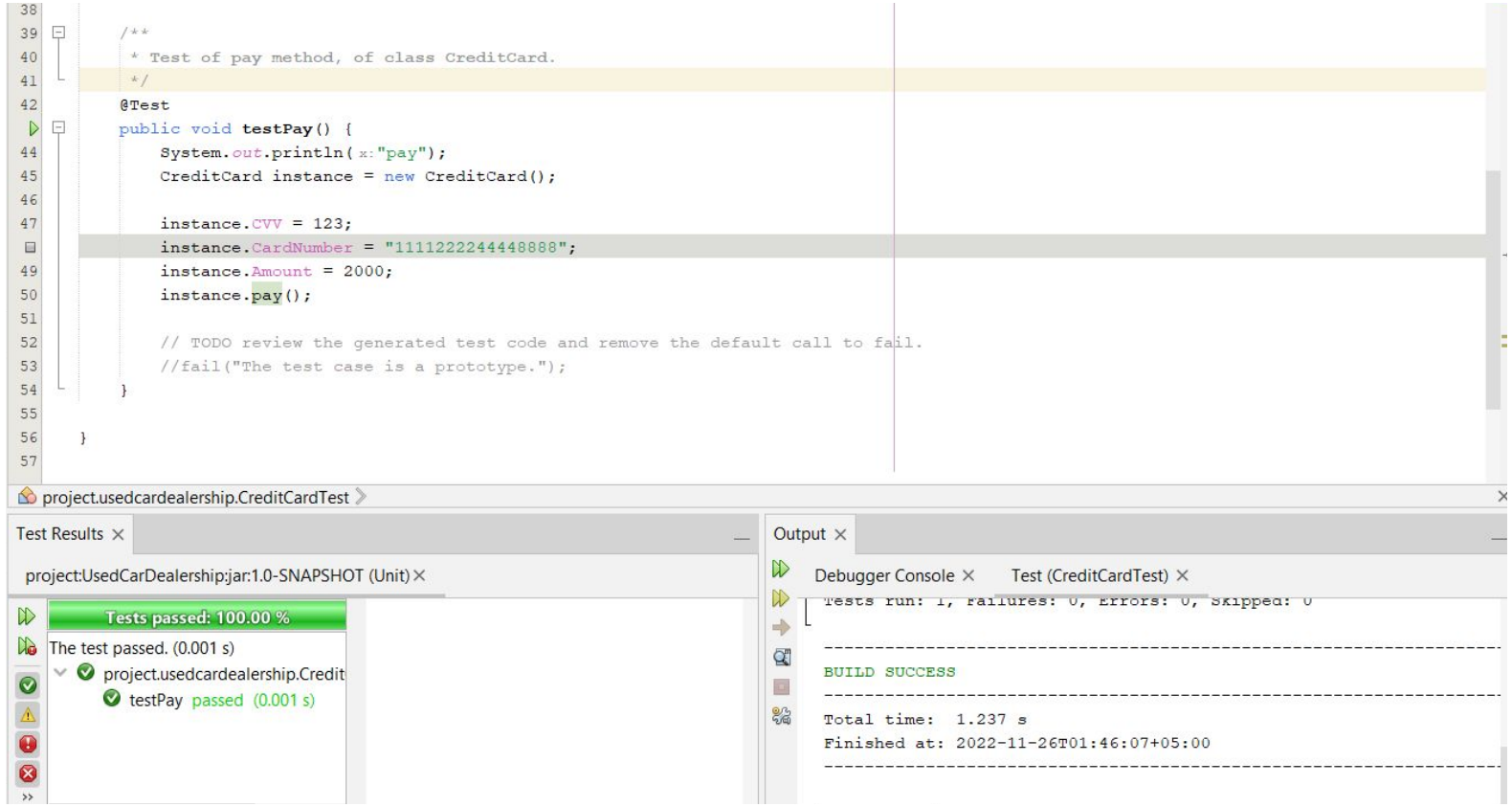
Unit Test 2 : Tax Free Car Sale Strategy



Unit Test 3 : Payment Method PayPal Strategy



Unit Test 4 : Payment Method Credit Card Strategy



Component Test

..ge | UsedCarDealership.java | TaxCarSale.java | TaxFreeCarSale.java | PaymentMethod.java | Paypal.java

Source | History

```
17      public static void RunComponentTest()
18      {
19          // Paypal Object Creation
20          Paypal paypal = new Paypal();
21          paypal.Amount= 1500;
22          paypal.UserName = "UserX";
23          paypal.Password = "Passsword1223";
24
25          //CreditCard Object Creation
26          CreditCard cc = new CreditCard();
27          cc.Amount = 3000;
28          cc.CVV = 233;
29          cc.CardNumber="1234656909386565";
30
31          TaxCarSale car1 = new TaxCarSale();
32          car1.CarPrice= 1500;
33          car1.PaymentType = paypal;
34          car1.TaxPercent= 5;
35          car1.PriceCalculation();
36          car1.PaymentType.pay();
37
38          TaxFreeCarSale car2 = new TaxFreeCarSale();
39          car2.CarPrice=3000;
40          car2.PaymentType= cc;
41
42          car2.PriceCalculation();
43          car2.PaymentType.pay();
```

project.usedcardealership.UsedCarDealership > RunComponentTest >

Output

Debugger Console | Run (UsedCarDealership)

```
--- exec-maven-plugin:3.0.0:exec (default-cli) @ UsedCarDealership ---
Price To Be Paid With Tax: 1575
1500 Paid by PayPal User: UserX
Price To Be Paid for Tax Free Car Sale: 3000
3000 Paid by Credit Card Number: 1234656909386565
------------------------------------------------------------------------
BUILD SUCCESS
```

Negative Test

```
    public static void RunNegativeTest()
{
    System.out.println( x: "\nNegative Test\n------------------------");
    // Paypal Object Creation
    Paypal paypal = new Paypal();
    paypal.Amount= 1500;
    paypal.UserName = "UserX";
    paypal.Password = "Passsword1223";

    //CreditCard Object Creation
    CreditCard cc = new CreditCard();
    cc.Amount = 3000;
    cc.CVV = 233;
    cc.CardNumber="123465609386565";

    CreditCard cc2 = new CreditCard();
    cc2.Amount = 3000;
    cc2.CVV = 23;
    cc2.CardNumber="1234656909386565";
    cc2.pay();

    TaxCarSale car1 = new TaxCarSale();
    car1.CarPrice= 1500;
    car1.PaymentType = paypal;
    car1.TaxPercent= 0;
    car1.PriceCalculation();
    car1.PaymentType.pay();

    TaxFreeCarSale car2 = new TaxFreeCarSale();
    car2.CarPrice=3000;
    car2.PaymentType= cc;
    car2.PriceCalculation();
    car2.PaymentType.pay();
}
```

Output

```
Negative Test
------------------------
Invalid CVV 23
Tax cannot be zero percent. Go For Tax Free Sales
Price To Be Paid With Tax: 1500
Username must be a valid email. Invalid Username: UserX
Price To Be Paid for Tax Free Car Sale: 3000
Invalid Card Number 123465609386565
------------------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------------------
```

- Thank you

FACTORY PATTERN

EASE IN CREATING OBJECTS

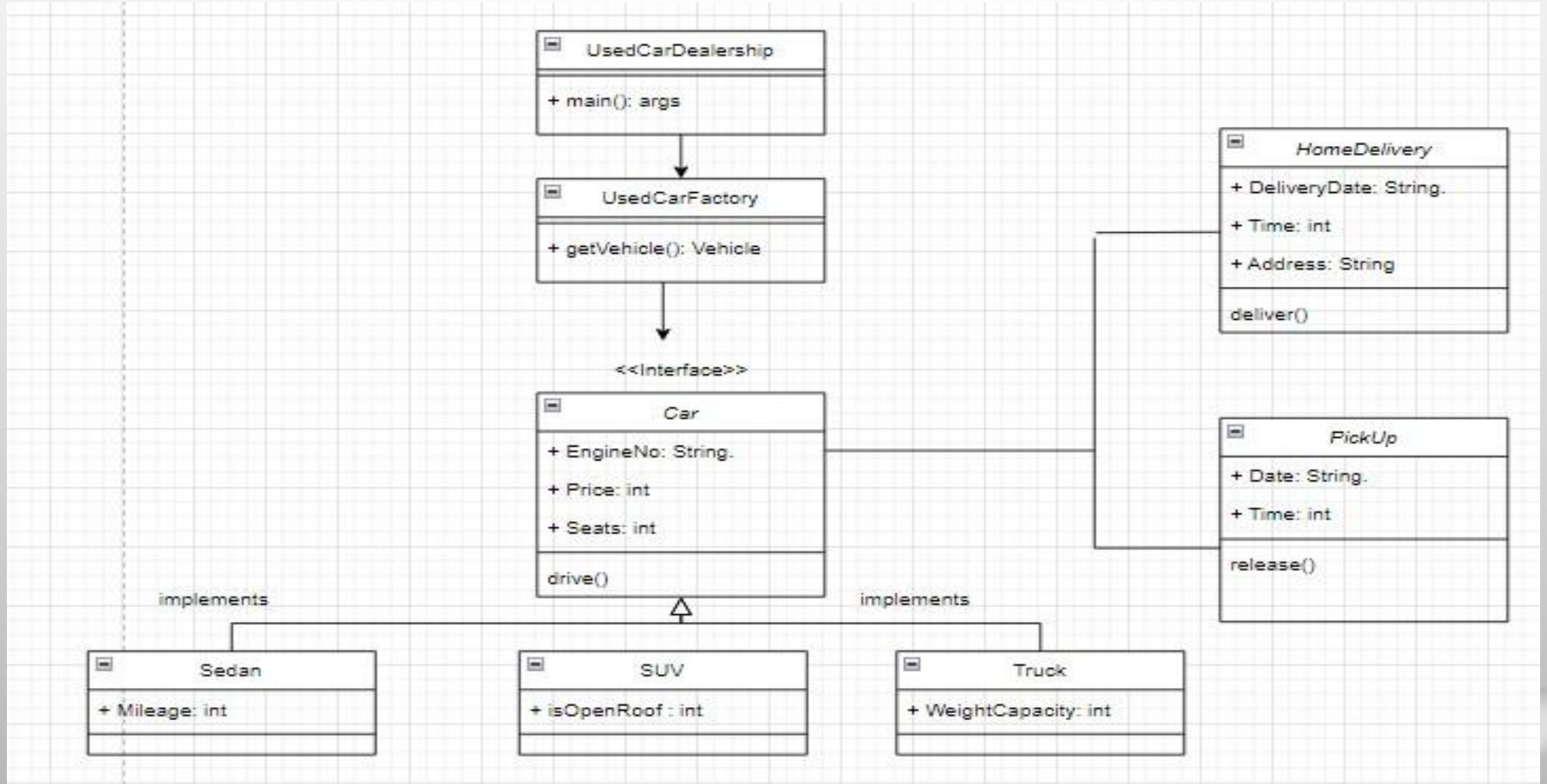
INTRODUCTION TO FACTORY PATTERN

- IT'S A CREATIONAL DESIGN PATTERN
- THIS PATTERN STATES THAT DEFINE AN INTERFACE FOR CREATING OBJECTS AND LET THE SUBCLASSES DECIDE WHICH CLASS TO INSTANTIATE.
- HELPS IN FUTURE SCALABILITY OF SYSTEM.

BENEFITS OF FACTORY PATTERN

- FACTORY METHOD PATTERN **ALLOWS THE SUB-CLASSES TO CHOOSE THE TYPE OF OBJECTS TO CREATE**.
- IT PROMOTES THE LOOSE-COUPLING BY ELIMINATING THE NEED TO BIND APPLICATION-SPECIFIC CLASSES INTO THE CODE.

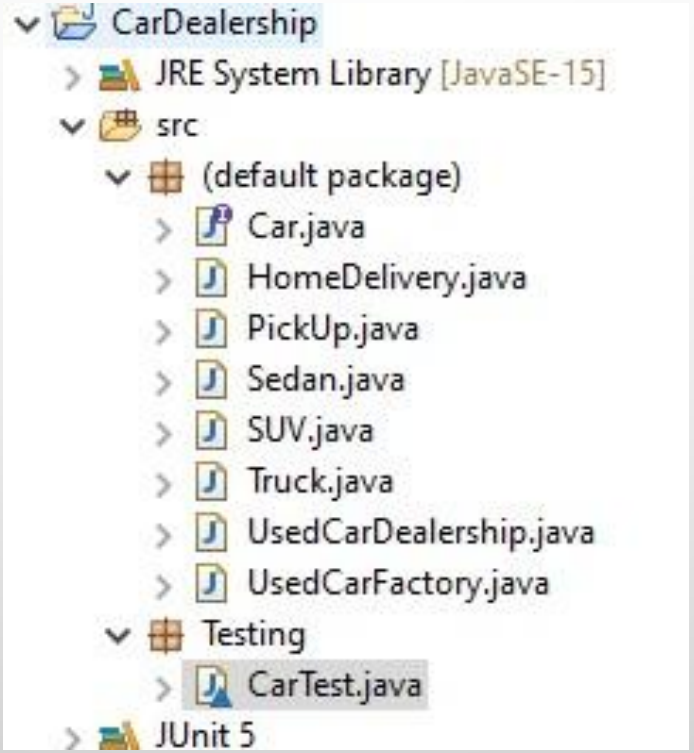
UML CLASS DIAGRAM



UML DESCRIPTION

- IN THE ABOVE MENTIONED UML CLASS DIAGRAM, A INTERFACE NAMED ‘CAR’ IS IMPLEMENTED
- IT LETS ITS SUBORDINATE CLASSES TO IMPLEMENT IT AND TO ACCESS COMMON FUNCTION WITH DIFFERENT FUNCTIONALITIES CLASS WISE.
- MOREOVER, I IMPLEMENTED THREE CONCRETE CLASSES OF DIFFERENT TYPES OF CARS.
- THEN, THERE IS A FACTORY CLASS WHICH DECIDES WHICH TYPE OF OBJECT IS TO BE CREATED BASED ON THE DATA RECEIVED BY THE MAIN CLASS.
- TWO ADDITIONAL CLASSES IMPLEMENTING THE DELIVERY MODULE HAVE BEEN INCLUDED WHICH ARE CLOSELY WORKING ALONG WITH OTHER CLASSES.

FILE STRUCTURE



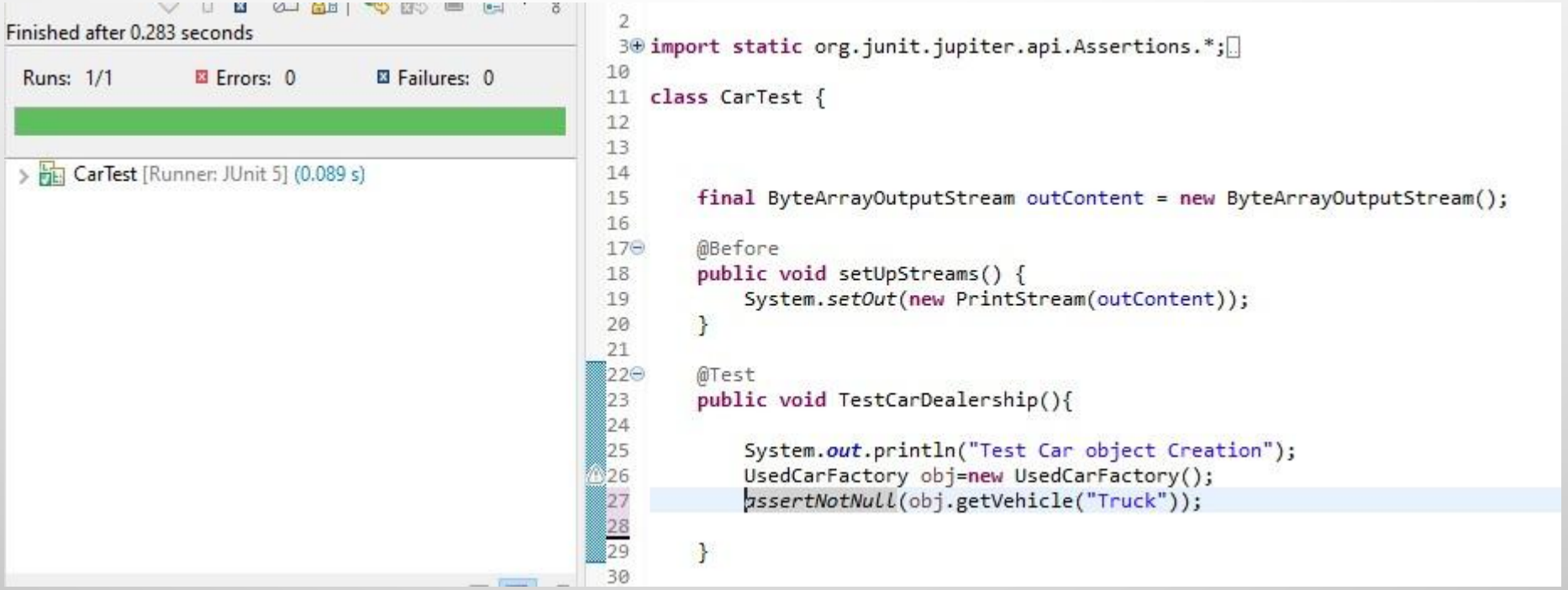
CODE SCREENSHOTS

```java
public class HomeDelivery {

    String Address;
    int time;
    String deliveryDate;

    public String getAddress() {
        return Address;
    }

    public void setAddress(String address) {
        Address = address;
    }

    public int getTime() {
        return time;
    }

    public void setTime(int time) {
        this.time = time;
    }

    public String getDeliveryDate() {
        return deliveryDate;
    }

    public void setDeliveryDate(String deliveryDate) {
        this.deliveryDate = deliveryDate;
    }

    public HomeDelivery(String address, int time, String deliveryDate) {
        super();
        Address = address;
        this.time = time;
        this.deliveryDate = deliveryDate;
    }
```

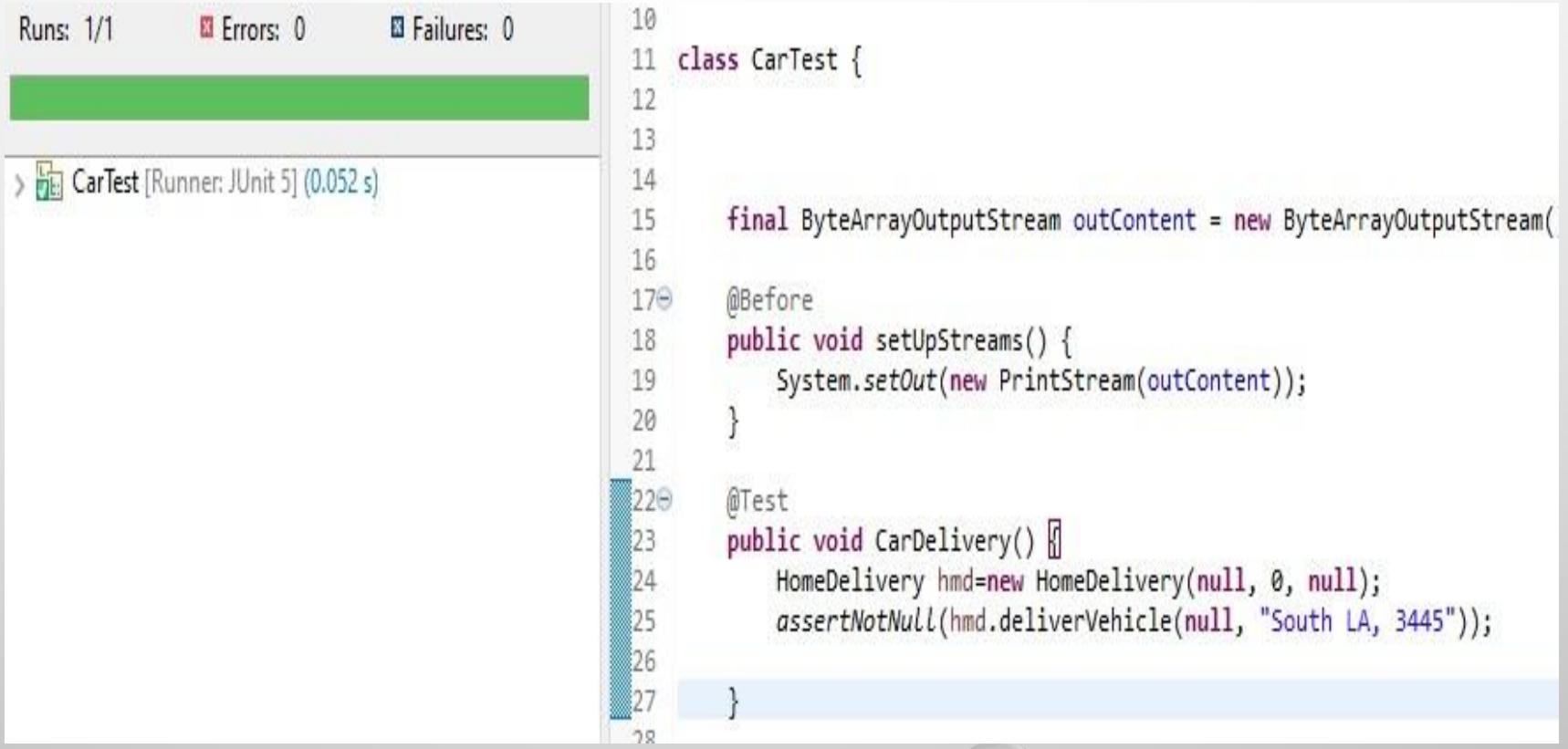
```java
public class UsedCarDealership {

    public static void main(String[] args) {

        UsedCarFactory carFactory = new UsedCarFactory();
        Car newCar=carFactory.getVehicle("SUV");
        newCar.drive();
    }
}
```

```java
public class UsedCarFactory {

    public Car getVehicle(String typeOfCar) {

        if(typeOfCar == null || typeOfCar.isEmpty()) return null;

        switch(typeOfCar) {
        case "Sedan":
            return new Sedan();
        case "SUV":
            return new SUV();
        case "Truck":
            return new Truck();
        default:
            throw new IllegalArgumentException("Unknown type of car -> "+typeOfCar);
        }

    }
}
```

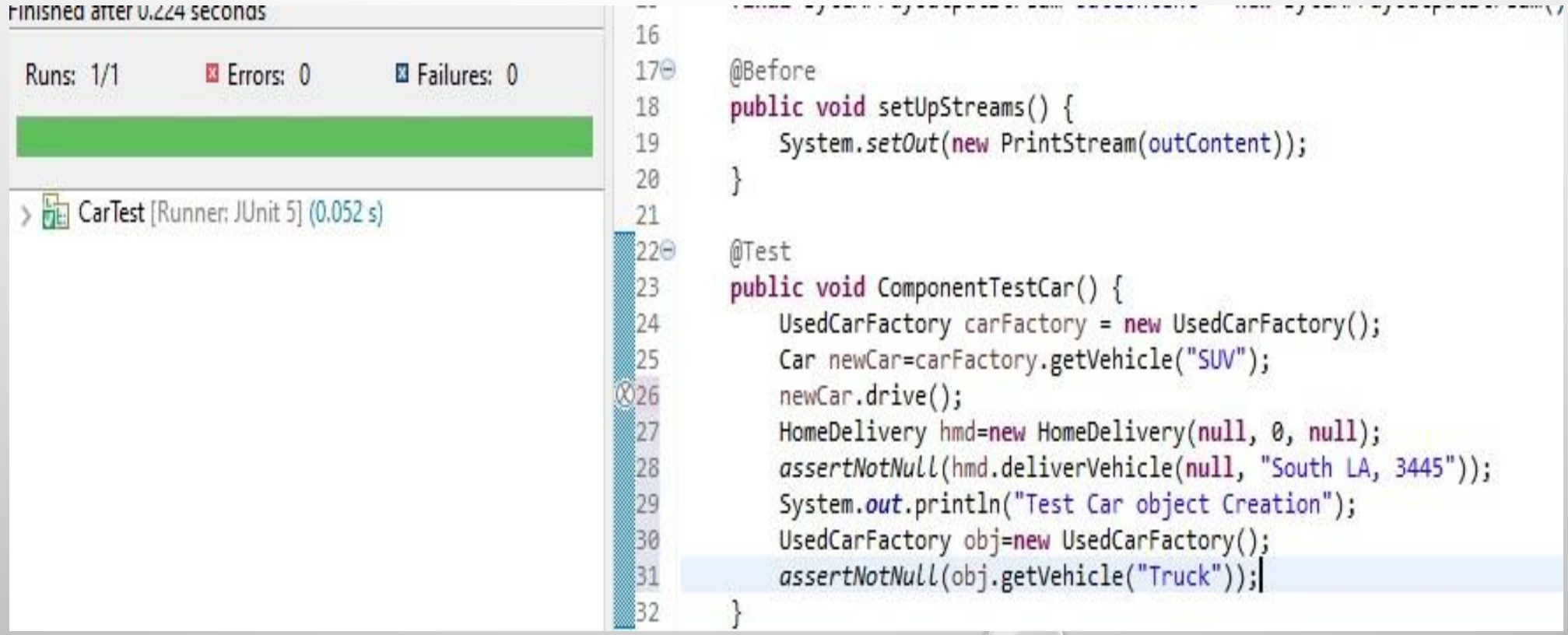
TESTCASE 1


Finished after 0.283 seconds
Runs: 1/1
Errors: 0
Failures: 0
CarTest [Runner: JUnit 5] (0.089 s)
import static org.junit.jupiter.api.Assertions.*;
class CarTest {
final ByteArrayOutputStream outContent = new ByteArrayOutputStream();
@Before
public void setUpStreams() {
System.setOut(new PrintStream(outContent));
}
@Test
public void TestCarDealership(){
System.out.println("Test Car object Creation");
UsedCarFactory obj=new UsedCarFactory();
assertNotNull(obj.getVehicle("Truck"));
}

TEST-CASE 2


Runs: 1/1
Errors: 0
Failures: 0
CarTest [Runner: JUnit 5] (0.052 s)
class CarTest {
final ByteArrayOutputStream outContent = new ByteArrayOutputStream(
@Before
public void setUpStreams() {
System.setOut(new PrintStream(outContent));
}
@Test
public void CarDelivery() {
HomeDelivery hmd=new HomeDelivery(null, 0, null);
assertNotNull(hmd.deliverVehicle(null, "South LA, 3445"));
}

POSITIVE COMPONENT TEST



NEGATIVE COMPONENT TEST

```
public void NegativeComponentTestCar() {
    UsedCarFactory carFactory = new UsedCarFactory();
    Car newCar=carFactory.getVehicle(435345);
    String returner = ((Object) newCar).drive();
    HomeDelivery hmd=new HomeDelivery(null, -5, null);
    ((Object) hmd).deliverVehicle(hmd,77585);
}
```

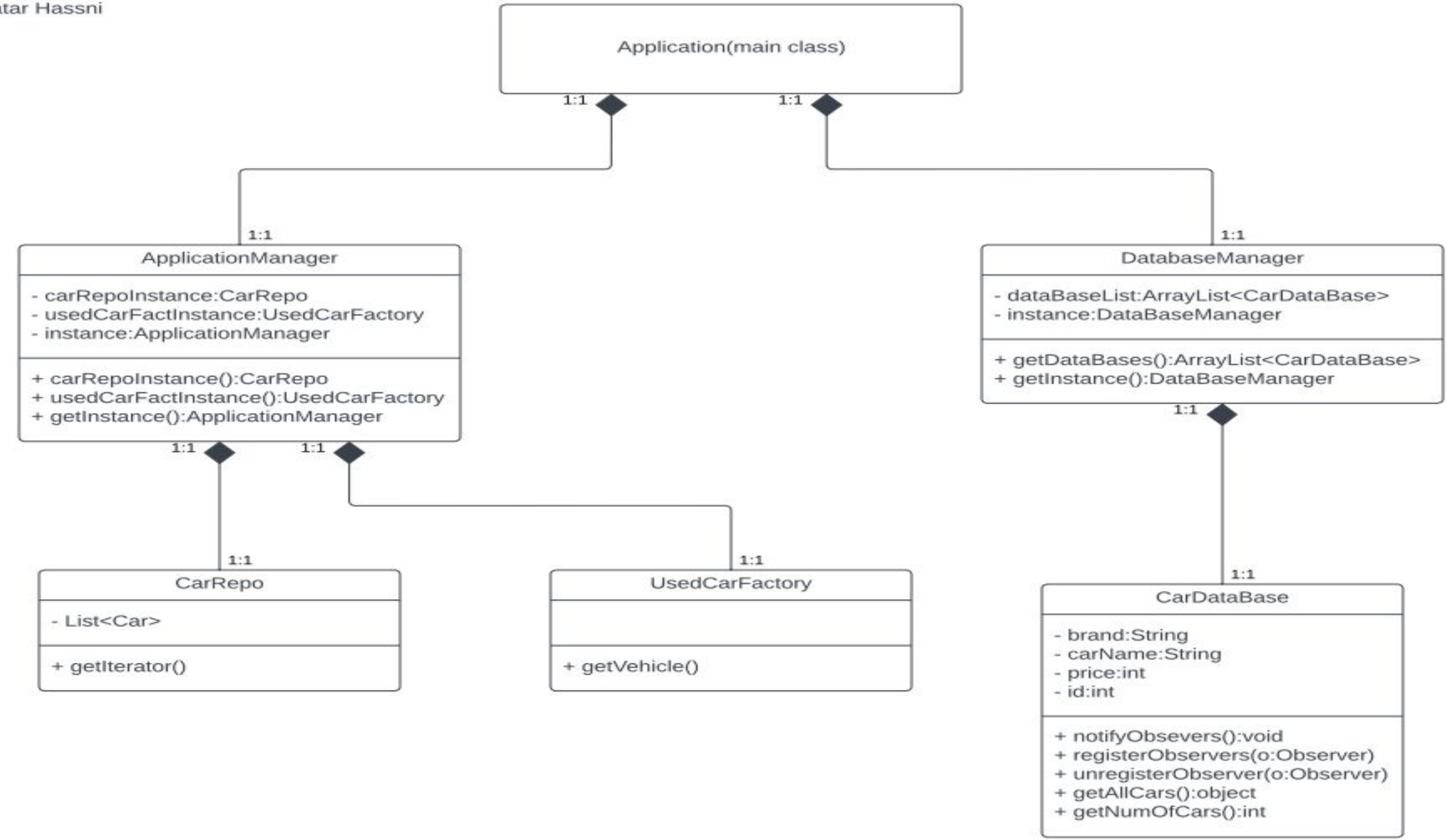
Singleton Pattern for Used Car System

By Satar Hassni

Which parts are required to be in the singleton pattern?

- The car repo.
- The used car factory.
- The car database.
- Because these parts of our application will only have one instance during run time.

Singleton pattern by
Satar Hassni



```
public class ApplicationManager {

    private CarRepo carRepoInstance;
    private UsedCarFactory usedCarFactory;
    private static ApplicationManager instance;

    public ApplicationManager() {
        this.carRepoInstance = new CarRepo();
        this.usedCarFactory = new UsedCarFactory();
    }

    public static ApplicationManager getInstance() {
        if (instance == null) {
            instance = new ApplicationManager();
        }
        return instance;
    }

    public CarRepo carRepoInstance() {
        return carRepoInstance;
    }

    public UsedCarFactory usedCarFactoryInstance() {
        return usedCarFactory;
    }
}
```

```
import java.util.ArrayList;

public class DataBaseManager {

    private List<CarDataBase> dataBaseList = new ArrayList<CarDataBase>();
    private static DataBaseManager instance;

    public DataBaseManager() {

    }

    public static DataBaseManager getInstance() {
        if (instance == null) {
            instance = new DataBaseManager();
        }
        return instance;
    }

    public List<CarDataBase> getDataBases() {
        return dataBaseList;
    }
}
```

Benefits?

- Although you can specifically make one instance in the main class this makes it cleaner and maintainable!
- Easier to modify one specific class then every place it's called.
- No need to directly call Main class with a static call to the instance of said object (generally very bad practice).