Strategy Design Pattern

Here we are going to understand about design patterns which are one of the most important aspects about Low Level Design (LLD). But before that we need to understand the design patterns.

- **What are Design Patterns?**

First we will understand the definition which is accepted by the world wide organisation for software engineering and designing for softwares.
'**Design Patterns are general repeatable solutions for a common problem occurring in the software designing**'

Now let me explain the design patterns in the simpler terms which can be understood by anyone.
Design Patterns can be considered as the principles for object oriented programming which are used for Software development by applying the concept of code reusability for development and maintenance.

First of all, we will understand the notation of inheritance

(1) Has - a composition
(2) IS -a inheritance

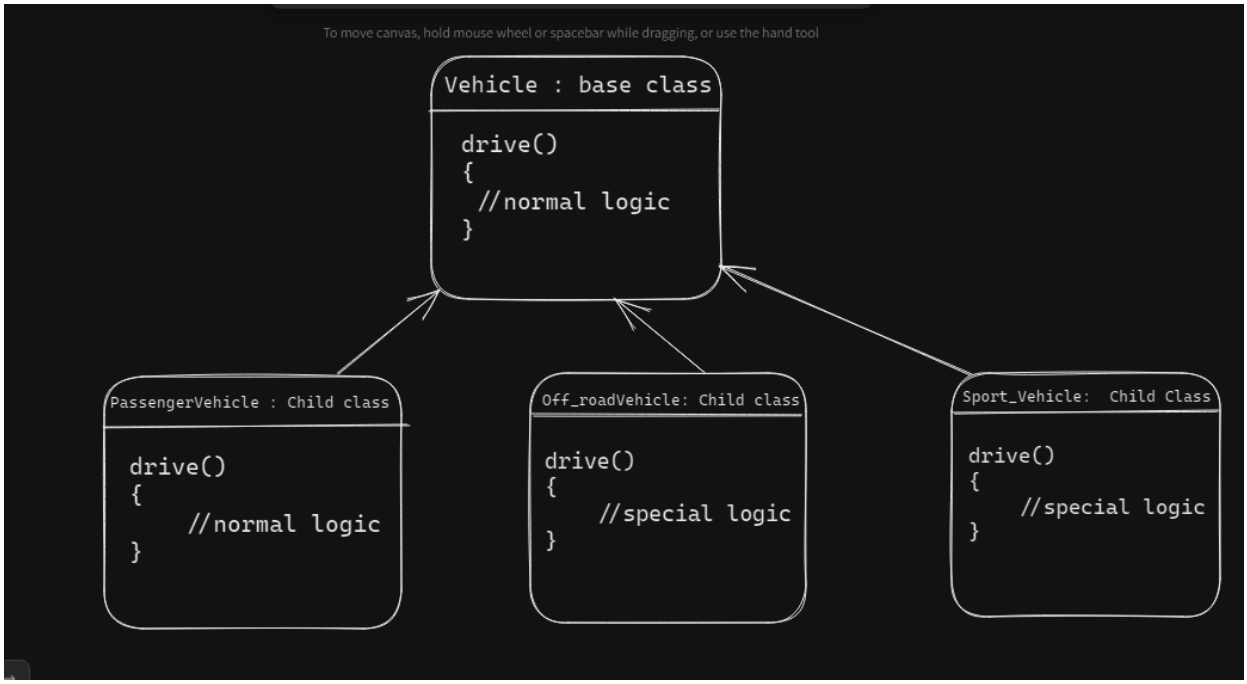
Composition(HAS-A) simply means the use of instance variables that are references to other objects. For example, a Maruti has an Engine, or a House has a Bathroom.

In object-oriented programming, **the concept of IS-A is totally based on Inheritance**, which can be of two types Class Inheritance or Interface Inheritance. It is just like saying "A is a B type of thing". For

example, Apple is a Fruit, Car is a Vehicle etc. Inheritance is unidirectional. For example, House is a Building. But Building is not a House.

We will try to understand when the scalability problem in the inheritance occurs so that we need to use design patterns.

Let us understand this scenario by an example which I find easy to explain and study the strategy design pattern.



Now as we can see the diagram. There is a base class named **vehicle**
which implements drive() method with some normal logic

There are total 3 child class which extends the properties of the base class

(1) **Passenger Class ->** Implements normal logic from base class
(2) **Off_roadVehicle Class ->** can not implement normal logic as it needs some special logic for implementing
(3) **Sport_Vehicle ->** same as the Off_roadVehicle Class

Now here we can understand that the 2 classes which need special logic can not implement the inherited method from the base class so they will override the method. Now for this case, the requirements of the classes are the same but not available in the base class so it is vulnerable to code duplication so it reduces the performance and scalability.

Whenever the feature increases like new methods like Display() or if the child classes increase, then, whenever child classes use the capabilities or methods which are not in base class but common among the child classes. At that particular time we can use the **Strategy Design Pattern**.

```
public class vehicle
{
    public void drive()
    {
        system.out.println("Normal Drive logic");
    }
}
```

Base Class

```
public class PassengerVehicle extends vehicle
{
    public void drive()
    {
        system.out.println("Normal Drive logic");
    }
}
```

Child class which can directly use method from Base

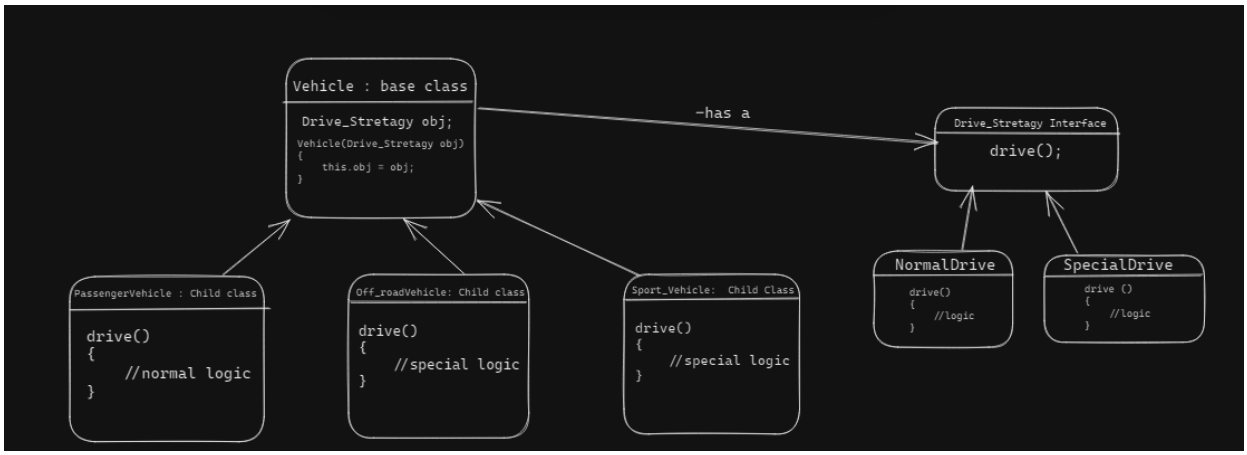
```
public class OffRoadVehicle extends vehicle
{
    public void drive()
    {
        system.out.println("special Drive logic");
    }
}

public class SportVehicle extends vehicle
{
    public void drive()
    {
        system.out.println("Special Drive logic");
    }
```

Child classes which needs the special logic and can not inherit method from the base class

Now how we can implement the solution so that we can implement strategy design pattern it is shown below:

==========

We will define an interface named which is considered as a strategy from which we will define the object.



Here we defined the Drive_Strategy interface and concrete classes implementing the Drive_Strategy interface.

```
//here we are implementing the interface which is known as
    stretagy
public interface Drive_Stretagy
{
    drive();
}
```

Now below are the concrete classes which are implementing the interface

```
//these are the concrete classes which are implementing the
    stretagy
class normalDrive implements Drive_Stretagy
{
    drive()
    {
        system.out.println("Normal Drive logic");
    }
}

class SpecialDrive implements Drive_Stretagy
{
    drive()
    {
        system.out.println("Special Drive logic");
    }
}
```

Now we will connect the base class with the interface using the composition property of inheriting.

```
class Vehicle implements Drive_Stretagy
{
    Drive_Stretagy obj;

    //this is known as the constructor injection
    public Vehicle(Drive_Stretagy obj)
    {
        this.obj = obj;
    }

    public void drive()
    {
        obj.drive();
    }
}
```

Now we will see the dynamical approach of how the child class call the superclass constructor and can implement the functionality which is appropriate for the particular class

```
class Off_roadVehicle extends Vehicle
{
    off_roadVehicle()
    {
        super(new SpecialDriveStretagy());
    }
}
```

So this is how we can design the particular strategy so that we can increase code reusability instead of code duplication.