# 1. Classification of Design Patterns

12 January 2024 21:46

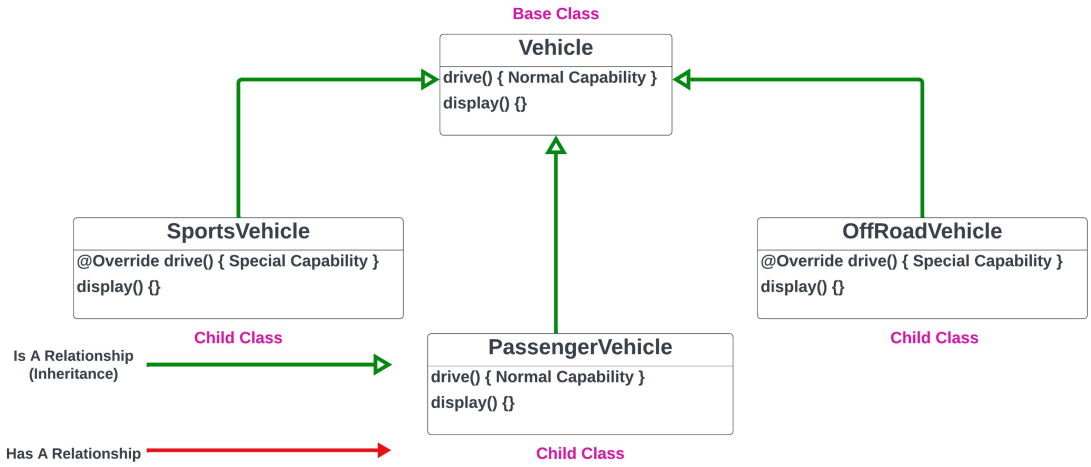
- **Creational Design Patterns**
  - **Abstract Factory**
  - **Builder**
  - **Factory Method**
  - **Prototype**
  - **Singleton**
- **Structural Design Patterns**
  - **Adapter**
  - **Composite**
  - **Bridge**
  - **Decorator**
  - **Façade**
  - **Flyweight**
  - **Proxy**
- **Behavioral Design Patterns**
  - **Chain of Responsibility**
  - **Interpreter**
  - **Iterator**
  - **Mediator**
  - **Memento**
  - **Observer**
  - **State**
  - **Strategy**
  - **Template Method**
  - **Visitor**

# 2. Strategy Design Pattern

12 January 2024 22:39

**Pre-requisites:**

1. We have a Vehicle Base class which have drive and display methods
2. **SportsVehicle** and **OffRoadVehicle** classes extending Vehicle class need Special Drive Capabilities so overrides drive method
3. **PassengerVehicle** class needs Normal Drive Capability so does not override drive method

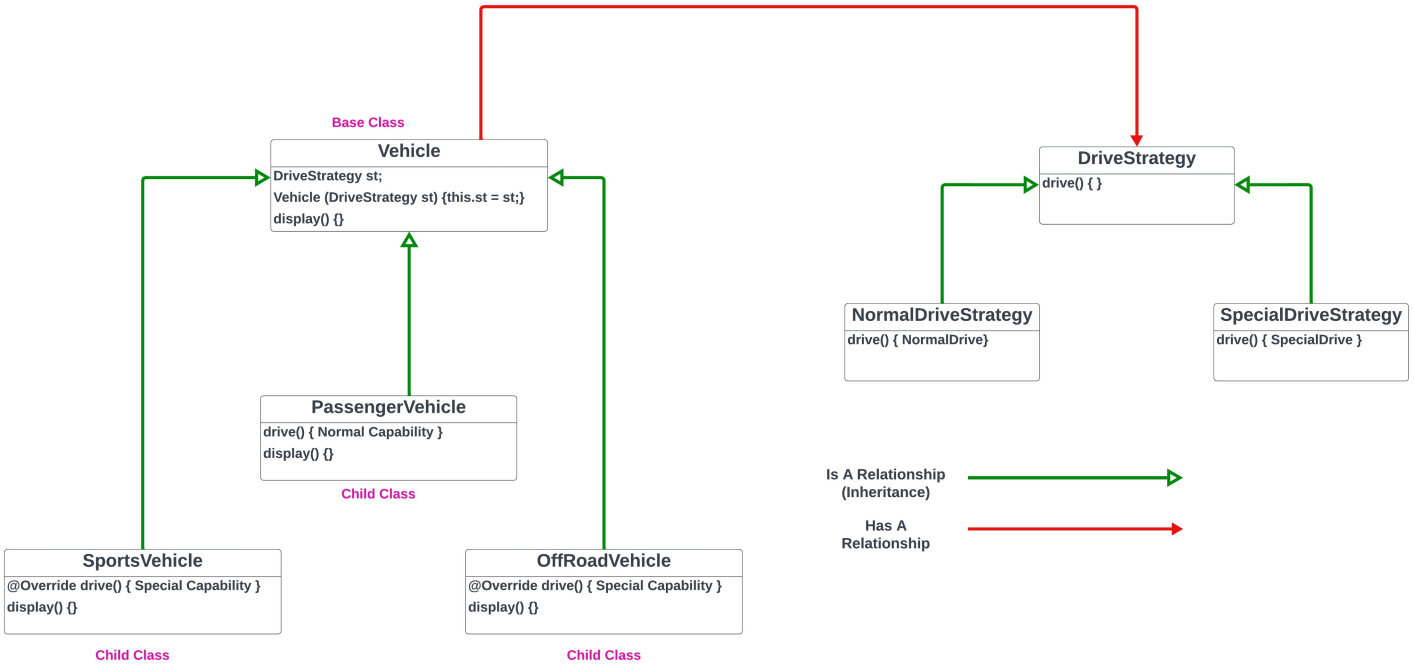



**Problem:**

1. Here both **SportsVehicle** and **OffRoadVehicle** class needs Special Drive capabilities and their functionality is different from Base class functionality so may result into duplication of code.
2. This duplication of code may increase when we need additional TypesVehicle classes

**Solution:**

1. This can be resolved using Strategy Design Pattern
2. Create a **DriveStrategy** Interface with concrete classes implementing the same as **NormalDriveStrategy** and **SpecialDriveStrategy**
3. In Vehicle class, use a variable of **DriveStrategy**
4. Now the individual classes have the responsibility to pass the Strategy to **Vehicle** class




**Strategy Designs Implementation**

```
public interface DriveStrategy {
    public void drive();
}
```

```
public class NormalDriveStrategy implements DriveStrategy {

    @Override
    public void drive() {
        System.out.println(x:"Normal Drive Capability");
    }

}
```

```
public class SportsDriveStrategy implements DriveStrategy {

    @Override
    public void drive() {
        System.out.println(x:"Special Drive Capability");
    }

}
```

**Vehicle Classes Modification to use the above Strategy**

```
public class Vehicle {

    private DriveStrategy strategy;

    public Vehicle(DriveStrategy strategy) {
        this.strategy = strategy;
    }

    public void drive() {
        strategy.drive();
    }

    public DriveStrategy getDriveStrategy() {
        return this.strategy;
    }
}
```

```
public class OffRoadVehicle extends Vehicle {
    public OffRoadVehicle() {
        super(new SportsDriveStrategy());
    }
}
```

```
public class PassengerVehicle extends Vehicle {
    public PassengerVehicle() {
        super(new NormalDriveStrategy());
    }
}
```

```
public class SportsVehicle extends Vehicle {
    public SportsVehicle() {
        super(new SportsDriveStrategy());
    }
}
```

# 3. Observer Design Pattern

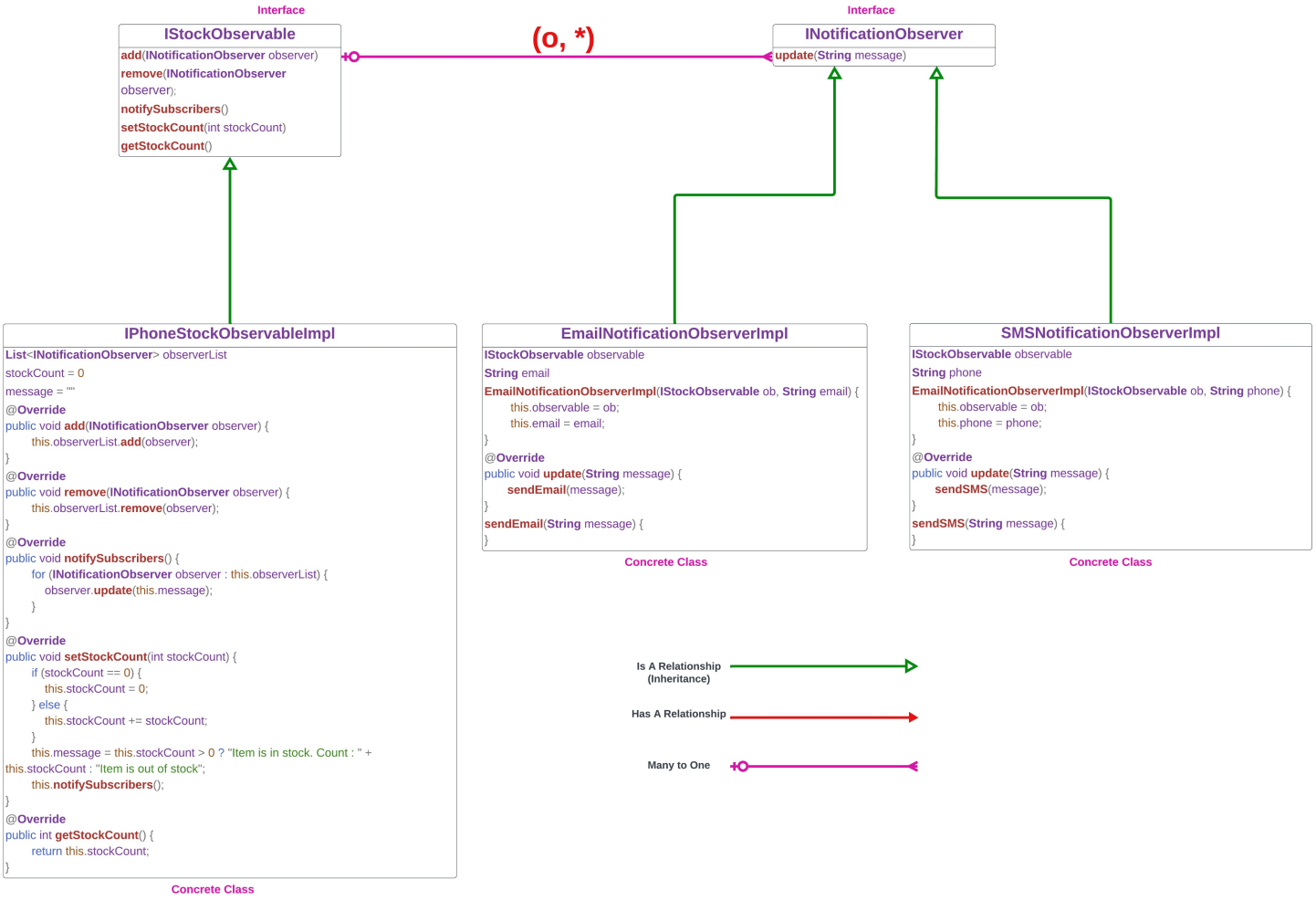
14 January 2024 22:25

**Use-cases**

- Amazon notifying customers once product is back in stock
- Weather App notifying users about change in temperature

**Amazon Back in Stock Notification**

The below ER Diagram shows how Observer design pattern would work in this scenario.




**Observable Code:**

This is mainly the service which is responsible to notify the subscribed customers.

**Observable Interface:**

```
public interface IStockObservable {
    public void add(INotificationObserver observer);

    public void remove(INotificationObserver observer);

    public void notifySubscribers();

    public void setStockCount(int stockCount);

    public int getStockCount();
}
```

**Observable Concrete Class Implementation:**

```java
public class IPhoneStockObservableImpl implements IStockObservable {

    List<INotificationObserver> observerList = new ArrayList<INotificationObserver>();
    int stockCount = 0;
    String message = "";

    @Override
    public void add(INotificationObserver observer) {
        this.observerList.add(observer);
    }

    @Override
    public void remove(INotificationObserver observer) {
        this.observerList.remove(observer);
    }

    @Override
    public void notifySubscribers() {
        for (INotificationObserver observer : this.observerList) {
            observer.update(this.message);
        }
    }

    @Override
    public void setStockCount(int stockCount) {
        if (stockCount == 0) {
            this.stockCount = 0;
        } else {
            this.stockCount += stockCount;
        }
        this.message = this.stockCount > 0 ? "Item is in stock. Count : " + this.stockCount : "Item is out of stock";
        this.notifySubscribers();
    }

    @Override
    public int getStockCount() {
        return this.stockCount;
    }

}
```

**Observer Code:**

This is mainly the service on the client side which will be called in case any changes happen to Observable code.

**Observer Interface:**

```java
public interface INotificationObserver {
    public void update(String message);
}
```

**Observer Concrete Classes Implementation:**

```java
public class EmailNotificationObserverImpl implements INotificationObserver {
    @SuppressWarnings("unused")
    private IStockObservable observable;
    private String email;

    public EmailNotificationObserverImpl(IStockObservable ob, String email) {
        this.observable = ob;
        this.email = email;
    }

    @Override
    public void update(String message) {
        sendEmail(message);
    }

    private void sendEmail(String message) {
        System.out.println("Notified via SMS to phone number : " + this.email + ". Message : " + message);
    }
}
```

```java
public class SMSNotificationObserverImpl implements INotificationObserver {
    @SuppressWarnings("unused")
    private IStockObservable observable;
    private String phoneNumber;

    public SMSNotificationObserverImpl(IStockObservable ob, String phoneNumber) {
        this.observable = ob;
        this.phoneNumber = phoneNumber;
    }

    @Override
    public void update(String message) {
        sendSMS(message);
    }

    private void sendSMS(String message) {
        System.out.println("Notified via SMS to phone number : " + this.phoneNumber + ". Message : " + message);
    }

}
```

**Main Store Class which will make changes to Observable**

```java
public class Store {
    Run | Debug
    public static void main(String[] args) {
        IStockObservable iphoneObservable = new IPhoneStockObservableImpl();

        INotificationObserver observer1 = new EmailNotificationObserverImpl(iphoneObservable, email:"abc.xyz@ymail.com");
        INotificationObserver observer2 = new EmailNotificationObserverImpl(iphoneObservable, email:"pqr.xyz@ymail.com");
        INotificationObserver observer3 = new SMSNotificationObserverImpl(iphoneObservable, phoneNumber:"+91-8456325478");
        INotificationObserver observer4 = new SMSNotificationObserverImpl(iphoneObservable, phoneNumber:"+91-9569852633");

        iphoneObservable.add(observer1);
        iphoneObservable.add(observer2);
        iphoneObservable.add(observer3);
        iphoneObservable.add(observer4);

        iphoneObservable.setStockCount(stockCount:10);
        iphoneObservable.setStockCount(stockCount:0);
        iphoneObservable.setStockCount(stockCount:100);
        iphoneObservable.setStockCount(stockCount:20);
    }
}
```