DESIGN PATTERNS PROJECT

GADGETS STORE MANAGEMENT SYSTEM

GROUP MEMBERS	Kulsoom Mateen (BESE-005) Maryam Nadeem (BESE-026) Hamdah Hamid (BESE-028) Syeda Muniba Faisal (BESE-030)
DEPARTMENT	Software Engineering
BATCH	2017-18
SUBJECT	Design Patterns
COURSE CODE	SE-408

Contents

GADGETS STORE MANAGEMENT SYSTEM

1. INTRODUCTION

1.1. PROBLEM:

Nowadays, gadgets and technology is something that attracts the mind of people regardless of age and gender. People are interested in gadgets and they have their choice of brands from where they prefer buying their favorite gadgets. Also when people have to buy particular gadget then they want to check that gadget from different brands/companies so that they will be able to pick the best one out of them. But mostly one store doesn’t have gadgets of different brands due to which people have to go to different stores to look for and to check other company’s gadgets features and to look for accessories of different brand’s gadgets is also difficult as a result of which customers don’t have flexibility to get the product from a single store.

1.2 SOLUTION:

So, we have come up with this project which will facilitate people to have a single gadget store which consists of different gadgets such as phones, laptops, tablets etc. from multiple different brands such as Apple, Lenovo, Samsung etc. to order them from a single place at a single platform. Also customers can get accessories for those different gadgets of different brands from a single store.

2. PURPOSE:

The purpose of this project is to build a gadget store management system for gadget order and store management having multiple gadgets of different variants and to facilitate the customers as well as gadget stores to have a well-managed and defined order placement mechanism.

3. INTENDED USERS:

The intended users of this application will be the people who like different gadgets of different variants and the gadget stores who will be able to manage complex orders in a well-defined and systematic way using this application.

4. PROJECT SCOPE:

The scope of this application is to ease the complicated steps involved in the order placement between customers and particular gadget factories. The main functionality of this application is in its ability which allows customers to order gadgets of their own choice from the brand of their choice as well and making the gadget stores able to deal with those orders with ease and in a well-managed manner.

5. TECHNOLOGY USED:

Operating system	Windows 10
Language	Java
IDE	IntelliJ Idea

6. SALIENT FEATURES:

- **Order Placement:**

  Our application enables users to place an order for different types of gadgets of different brands from our store.

- **Payment Mechanism:**

  Our application provides customers with different payment mechanisms to select from according to their own comfort.

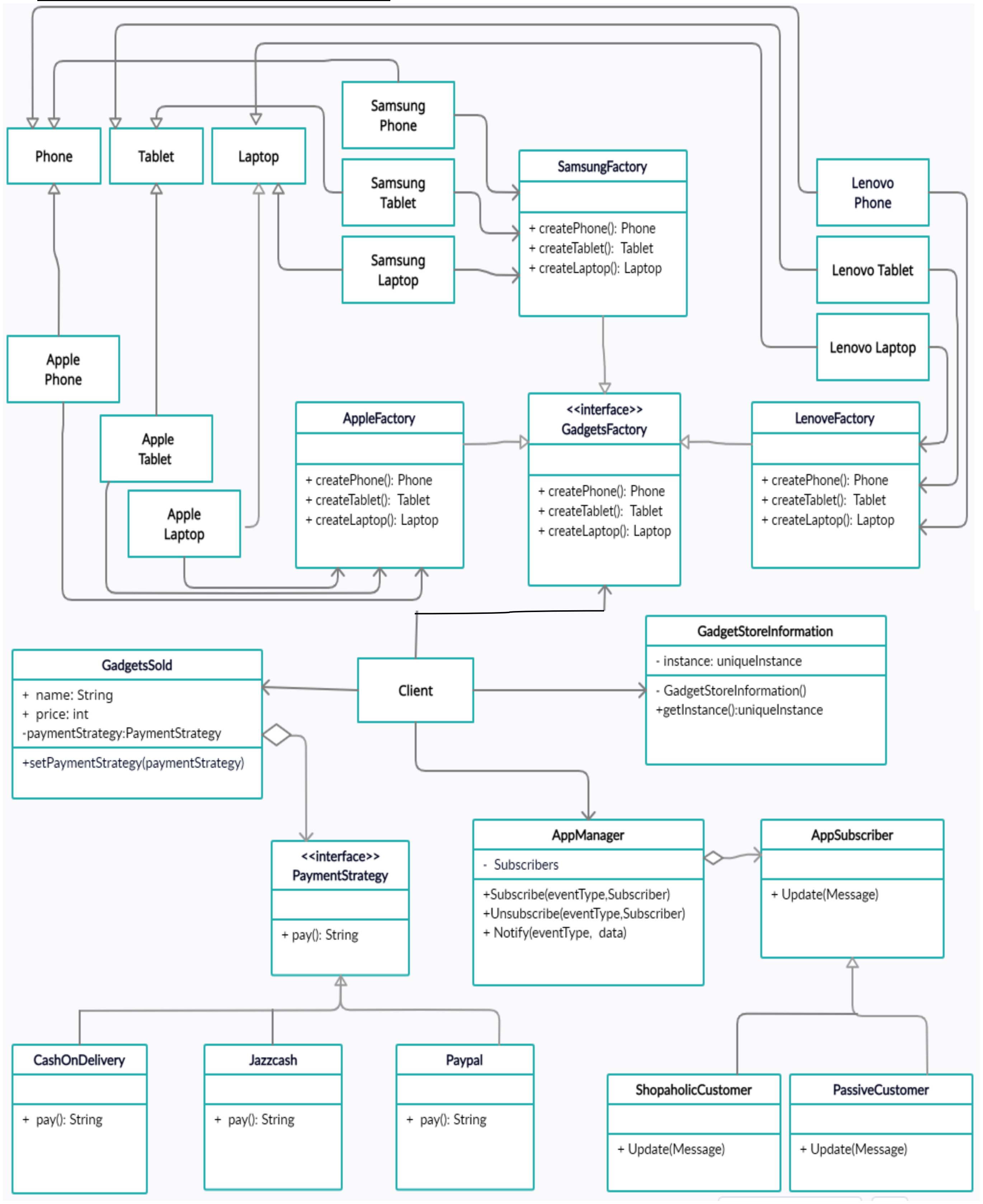
- **Gadget Store Information:**

  Our application will provide customers with adequate information of our gadget store on every item they order.

- **Subscription:**

  Our application will notify the subscribed users about the new arrivals of gadgets from different brands.

7. SYSTEM CLASS DIAGRAM:



8. DESIGN PATTERNS APPLICATIONS:

The design patterns we have used in our project are:

1) Abstract Factory Pattern:

Our application consists of a store that consists of multiple gadgets of different variants which will help customers to order different gadgets of their choice from variants of their own choice and to achieve this we have used an abstract factory design pattern in our project.

2) Strategy Pattern:

Our application requires to provide our customers to choose a payment method from a given list of payment methods at runtime. These payment methods are needed to separate from each other so for this purpose we used strategy pattern in our project.

3) Singleton Pattern:

Our application requires a unique information of our gadget store, which will be available on every package bought or sold from or store. To keep the information of our store that can only be initialized once we wrapped the Gadget Store Information class in a Singleton Pattern.

4) Observer Pattern:

There are some users who are waiting for new Arrivals of different gadgets in our gadget store. So there is a subscription mechanism in our application, by which users can get subscribed to our application and will get the latest updates of new Arrivals of gadgets.

8.1. ABSTRACT FACTORY PATTERN:

Intent:

Abstract factory is a creational design pattern that lets you create families of related objects without specifying their concrete classes.

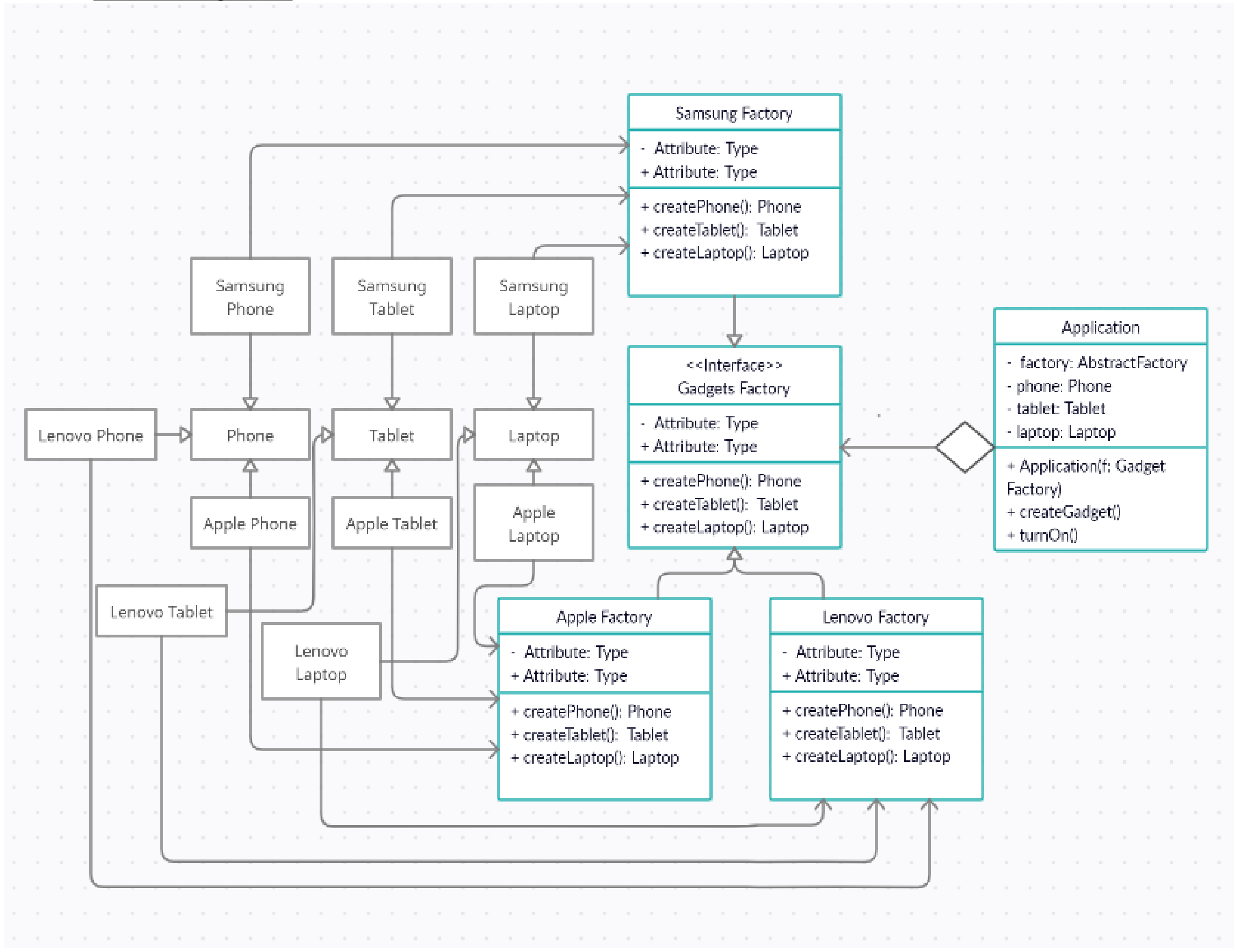
Need:
We needed to have a store that contained multiple gadgets from different factories to facilitate the customers to order their choice of gadget from their choice of factory. So, we used an abstract factory pattern.

Implementation:
In our project, this pattern creates different variants of multiple products which in our case we have a gadget store as our abstract factory that consists of methods for concrete factories to implement. For our products we have three interfaces which will be implemented by concrete product classes. Our client application can create multiple

products of a single factory at a time by specifying the factory in the application constructor.
The pattern consists of the following:

- An abstract factory interface called gadgets factory which contains methods for product creation and their return type same as product interface name.
- Concrete factory classes called Apple factory, Lenovo factory and Samsung factory which will implement the methods of abstract factory interface.
- Product interfaces called phones, tablets and laptops which consist of methods for implementation by concrete product classes.
- Concrete product classes called Apple phones, Apple tablets, Apple laptops, Lenovo phones, Lenovo tablets, Lenovo laptops, and Samsung phones, Samsung tablets and finally Samsung laptops for concrete implementation of product interfaces.
- Finally a client application class that contains a private constructor that takes the factory class and creates gadgets of that factory class.

<u>Class Diagram:</u>



8.2.STRATEGY PATTERN:

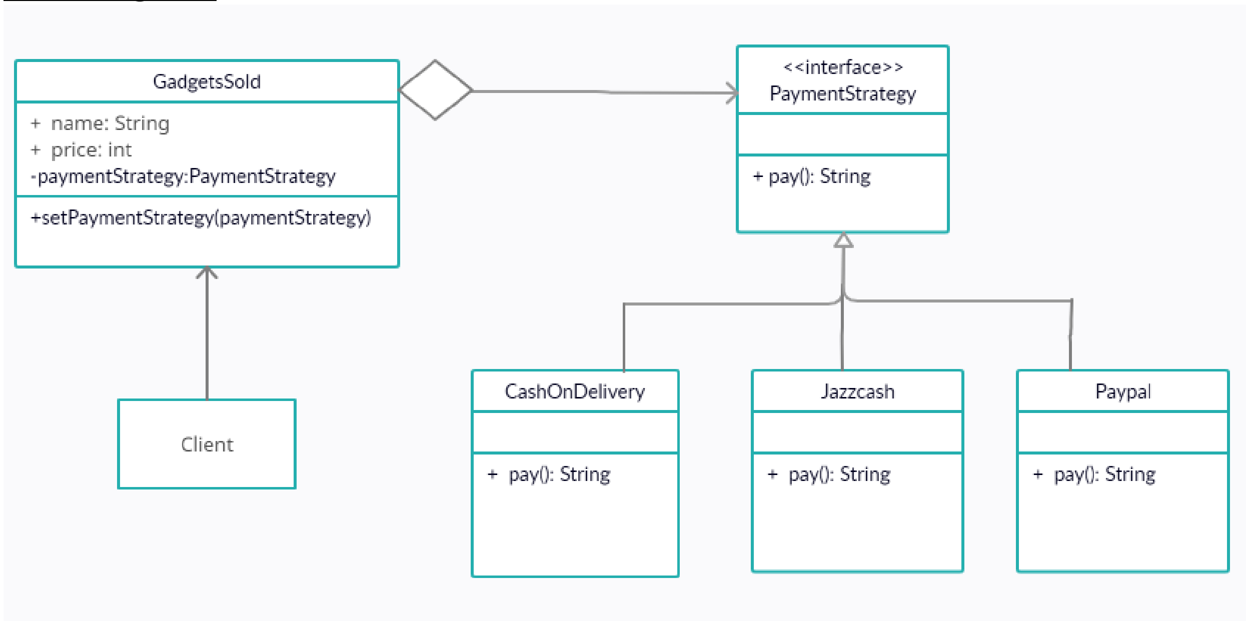
Intent:
Strategy pattern is a behavioral design pattern that lets you define a family of algorithms, put each of them into a separate class, and make their objects interchangeable.
Here we used this method to pay money by choosing payment method from a given list of payment methods at runtime easily.

Need:
We needed to provide our customers to choose a payment method from a given list of payment methods at runtime. These payment methods are needed to separate from each other because every payment method has a different process but they do the same action (make the payment) and as strategy pattern suggest to take that class that does something specific in a lot of different ways and extract all these in different classes (called strategies) so, we used strategy pattern.

Implementation:
In our project, this pattern will help the customers to make payment but it is changed on the runtime according to the customer's payment selection option. For this an interface is used to create the opportunity to dynamic behavioral changes and this interface will be implemented by different payment methods into their own set of classes like Cash on delivery, Jazzcash, Paypal etc.
The pattern consists of the following:

- The context is a GadgetsSold class that maintains a reference to one of the concrete payment strategies and communicates with this object only via the Strategy interface.
- The PaymentStrategy interface is common to all concrete payment strategies. It declares a method the context uses to execute a strategy.
- Concrete payment strategies which are CashOnDelivery, Jazzcash and Paypal classes will implement different variations of payment the GadgetsSold class will use.
- The context calls the execution method on the linked strategy object each time it needs to run the algorithm. The context doesn't know what type of payment strategy it works with.
- The client creates a specific payment strategy object ad passes it to the context. The context exposes a setter which lets clients replace the strategy associated with the context at runtime.

Class Diagram:



8.3. SINGLETON PATTERN:

Intent:
Singleton is a creational design pattern that lets you ensure that a class has only one instance, while providing a global access point to this instance.

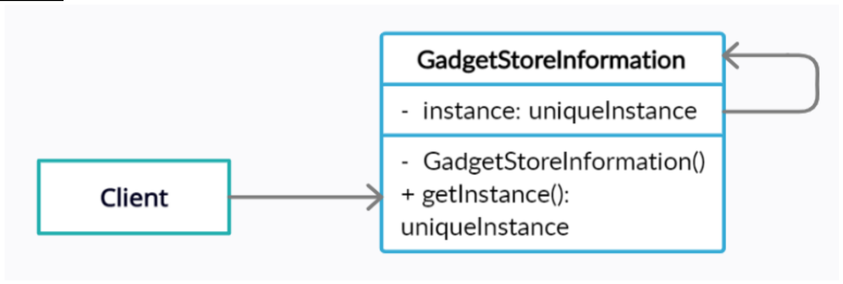
Need:
We needed a unique information for our Gadget Store, for uniquely identifying it. The object of Gadget Store Information class was required to be one at any time and all other classes need a global access to it. So, we used a Singleton.

Implementation:
In our project, this pattern creates an instance of the information at the beginning of the program. Whenever the customers buy a gadget/s from our store, the unique information instance of our store will be added to the packaging. As a result, it will uniquely identify our gadget store that this particular gadget/s has been brought from this store.

The pattern consists of the following:

- A GadgetStoreInformation class is a Singleton class. It wraps a reference to itself and the constructor is made private to avoid other classes from creating another instance of it.
- The GadgetStoreInformation class declares the static method getInstance that returns the same instance of its own class i.e., uniqueInstance.
- The GadgetStoreInformation's constructor should be hidden from the client code. Calling the getInstance method should be the only way of getting the GadgetStoreInformation object.

Class Diagram:


GadgetStoreInformation
- instance: uniqueInstance
- GadgetStoreInformation()
+ getInstance(): uniqueInstance
Client

8.4. OBSERVER PATTERN:

Intent:
Observer is a behavioral design pattern that lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they're observing.

Need:
As there are some users who wants to know latest updates about our store. And if we send email messages to our all users so it will be spam for those users who don't want to know about latest updates. So through observer pattern we will introduce subscription mechanism so that only subscribed customers will get notifications about new updates and our latest arrivals.

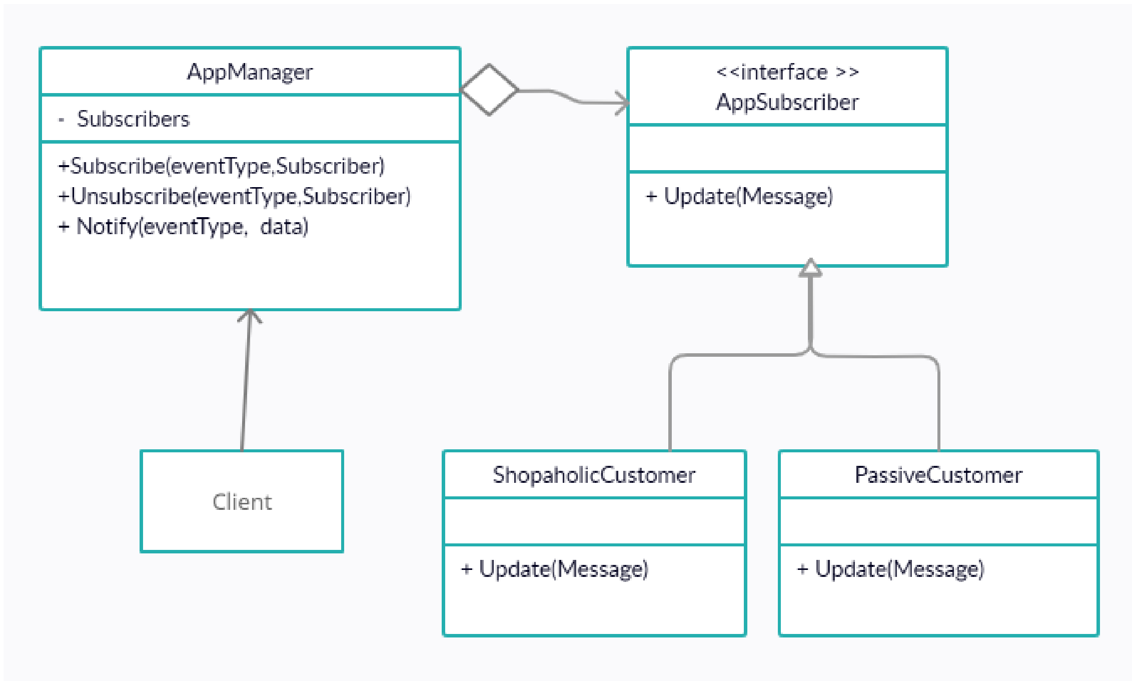
Implementation:
In our project, there is a main interface i.e Gadgets factory, and there is a separate class of Publisher called App Manager who will deal with the subscription module of the app. Also there is a separate interface for subscription called App Subscriber and there are concrete subscriber classes like Shopaholic Customer and Passive Customer, so that different users can get notification and they will act in different ways according to their interests.

The pattern consists of the following:

- There is a **Publisher** class called App Manager linked with main class of gadgets factory. Publishers contain a subscription infrastructure that lets new subscribers join and current subscribers leave the list.
- The **Subscriber** interface declares the notification interface. In most cases, it consists of a single update method. The method may have several parameters that let the publisher pass some event details along with the update.
- **Concrete Subscribers** called Shopaholic Customer and Passive Customer perform some actions in response to notifications issued by the publisher. All of these classes must implement the same interface so the publisher isn't coupled to concrete classes.

- The **Client** creates publisher and subscriber objects separately and then registers subscribers for publisher updates.

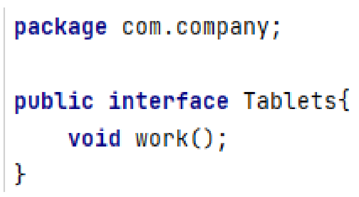
<u>Class Diagram:</u>



9. CLIENT-SIDE CODE:

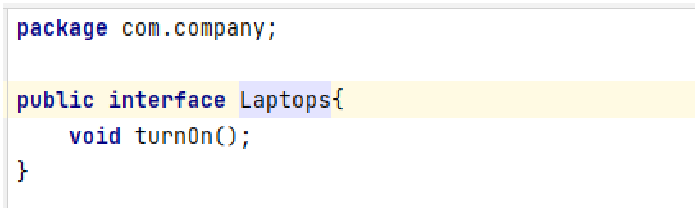
<u>Abstract Factory Pattern:</u>

Tablets.java (interface):



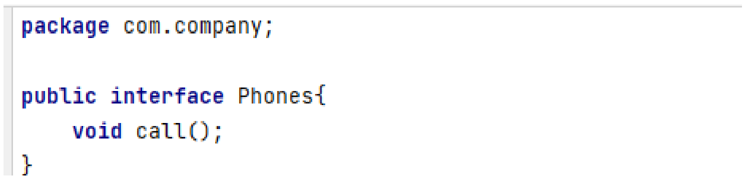
```
package com.company;

public interface Tablets{
    void work();
}
```

Laptops.java (interface):



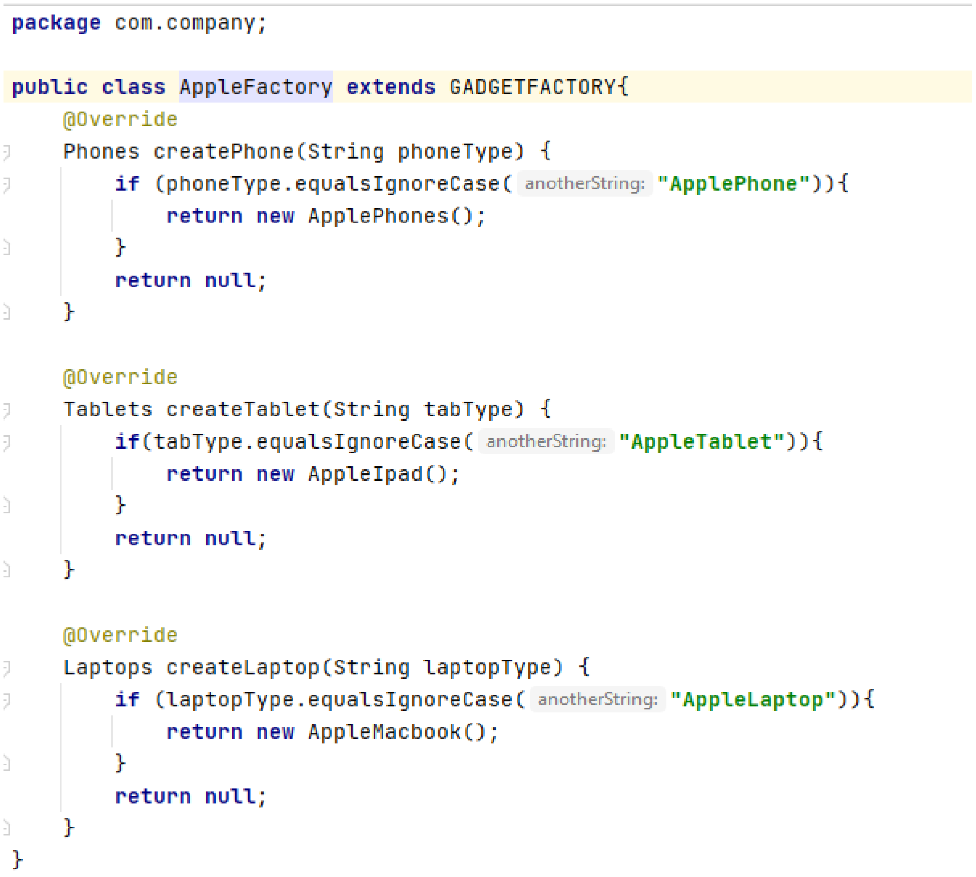
```
package com.company;

public interface Laptops{
    void turnOn();
}
```

Phones.java (interface):



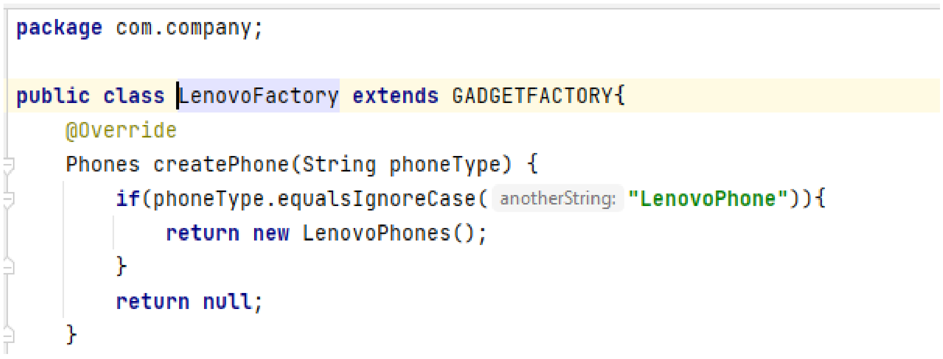
```
package com.company;

public interface Phones{
    void call();
}
```

AppleFactory.java:

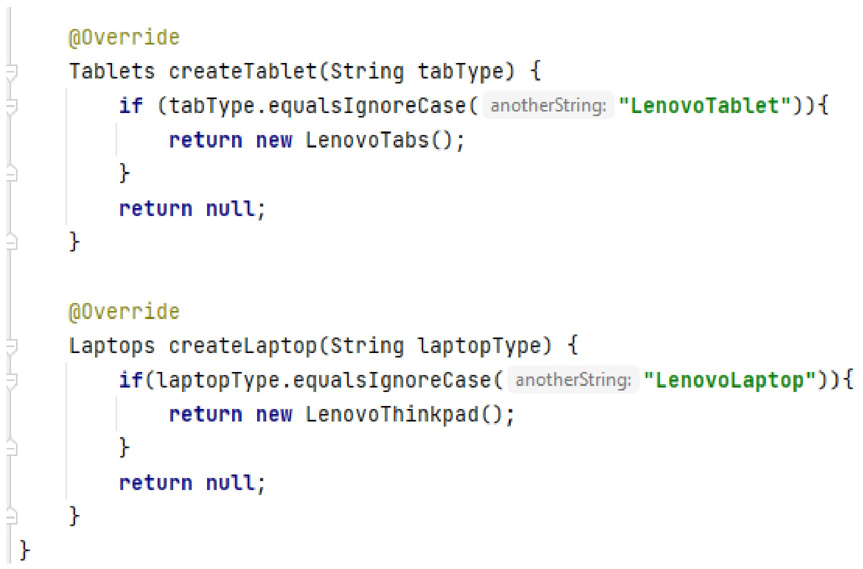


```
package com.company;

public class AppleFactory extends GADGETFACTORY{
    @Override
    Phones createPhone(String phoneType) {
        if (phoneType.equalsIgnoreCase( anotherString: "ApplePhone")){
            return new ApplePhones();
        }
        return null;
    }

    @Override
    Tablets createTablet(String tabType) {
        if(tabType.equalsIgnoreCase( anotherString: "AppleTablet")){
            return new AppleIpad();
        }
        return null;
    }

    @Override
    Laptops createLaptop(String laptopType) {
        if (laptopType.equalsIgnoreCase( anotherString: "AppleLaptop")){
            return new AppleMacbook();
        }
        return null;
    }
}
```

LenovoFactory.java:

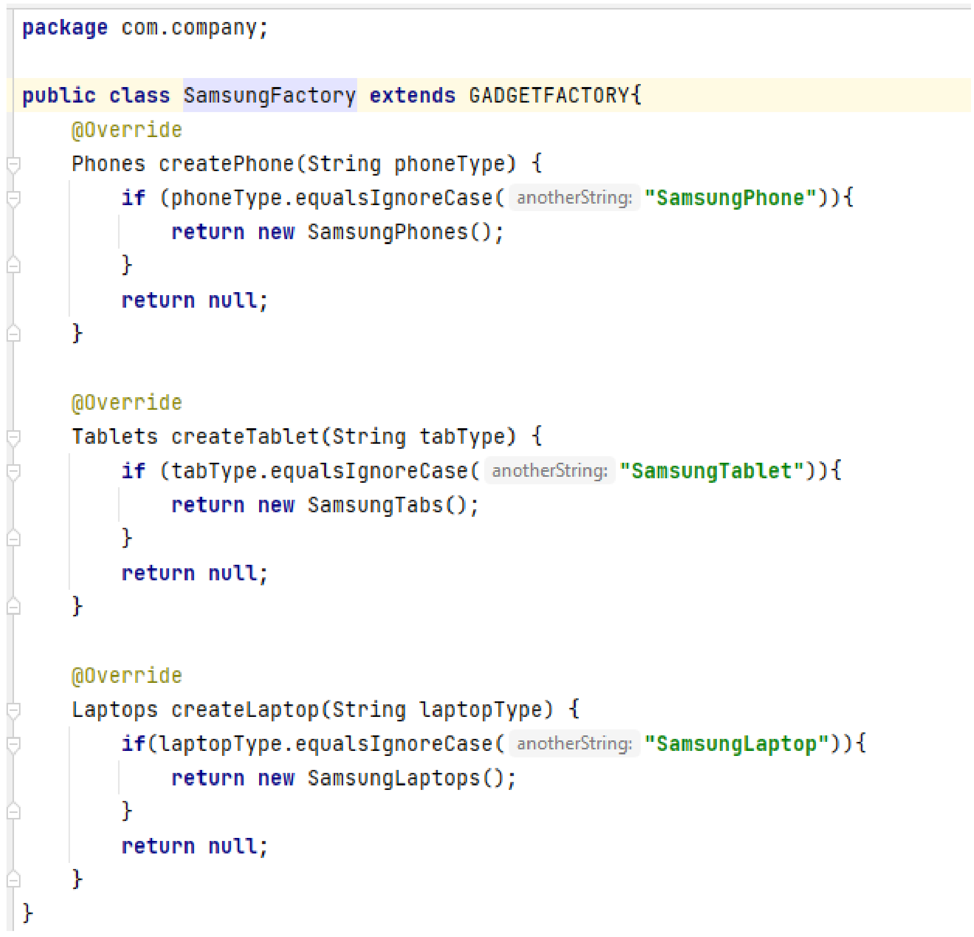


```
package com.company;

public class LenovoFactory extends GADGETFACTORY{
    @Override
    Phones createPhone(String phoneType) {
        if(phoneType.equalsIgnoreCase( anotherString: "LenovoPhone")){
            return new LenovoPhones();
        }
        return null;
    }
```



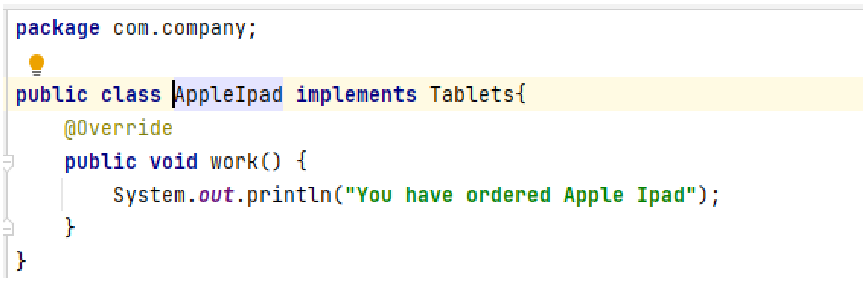
```
    @Override
    Tablets createTablet(String tabType) {
        if (tabType.equalsIgnoreCase( anotherString: "LenovoTablet")){
            return new LenovoTabs();
        }
        return null;
    }

    @Override
    Laptops createLaptop(String laptopType) {
        if(laptopType.equalsIgnoreCase( anotherString: "LenovoLaptop")){
            return new LenovoThinkpad();
        }
        return null;
    }
}
```

SamsungFactory.java:



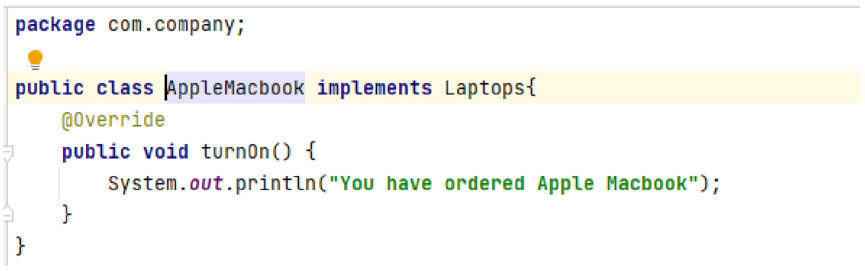
```
package com.company;

public class SamsungFactory extends GADGETFACTORY{
    @Override
    Phones createPhone(String phoneType) {
        if (phoneType.equalsIgnoreCase( anotherString: "SamsungPhone")){
            return new SamsungPhones();
        }
        return null;
    }

    @Override
    Tablets createTablet(String tabType) {
        if (tabType.equalsIgnoreCase( anotherString: "SamsungTablet")){
            return new SamsungTabs();
        }
        return null;
    }

    @Override
    Laptops createLaptop(String laptopType) {
        if(laptopType.equalsIgnoreCase( anotherString: "SamsungLaptop")){
            return new SamsungLaptops();
        }
        return null;
    }
}
```

AppleIpad.java:



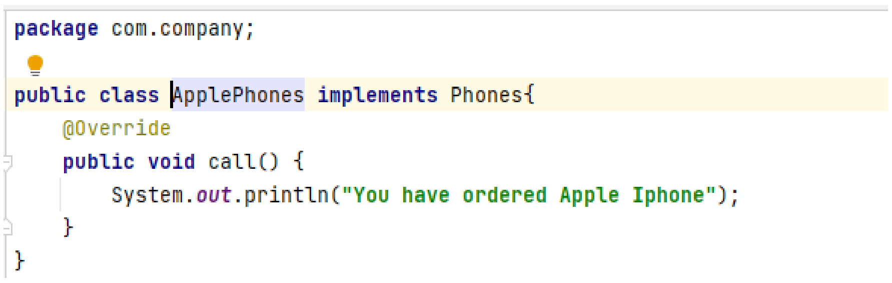
```java
package com.company;

public class AppleIpad implements Tablets{
    @Override
    public void work() {
        System.out.println("You have ordered Apple Ipad");
    }
}
```

AppleMacbook.java:



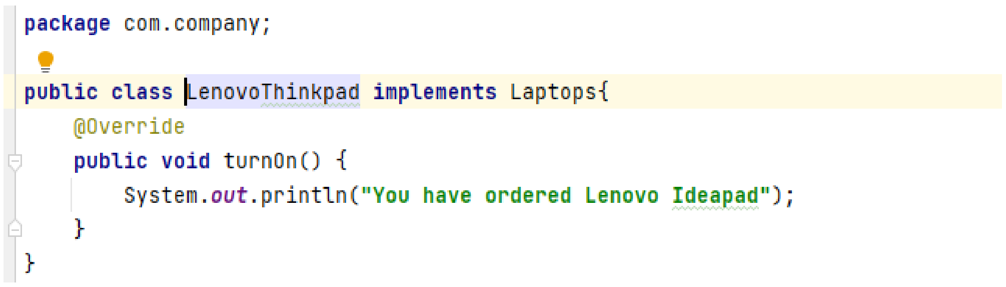
```java
package com.company;

public class AppleMacbook implements Laptops{
    @Override
    public void turnOn() {
        System.out.println("You have ordered Apple Macbook");
    }
}
```

ApplePhones.java:



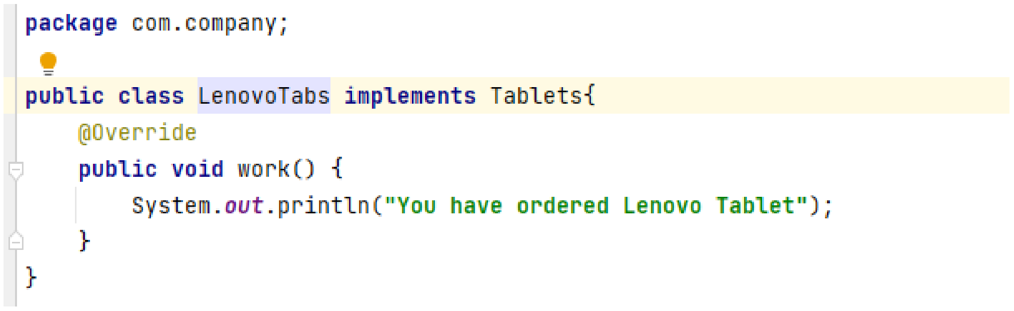
```java
package com.company;

public class ApplePhones implements Phones{
    @Override
    public void call() {
        System.out.println("You have ordered Apple Iphone");
    }
}
```

LenovoThinkpad.java:



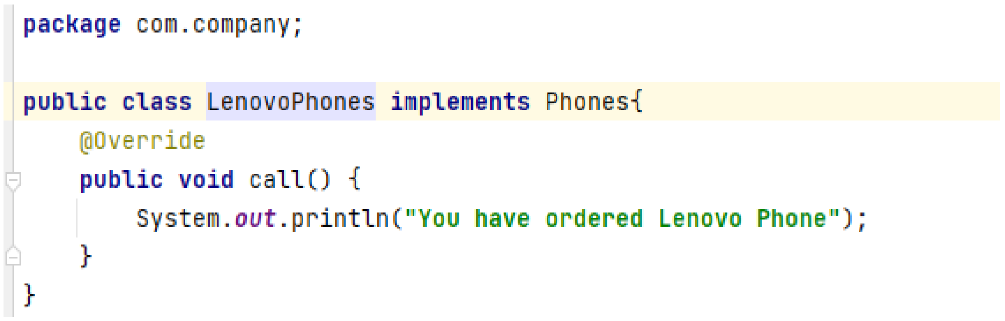
```java
package com.company;

public class LenovoThinkpad implements Laptops{
    @Override
    public void turnOn() {
        System.out.println("You have ordered Lenovo Ideapad");
    }
}
```

LenovoTabs.java:



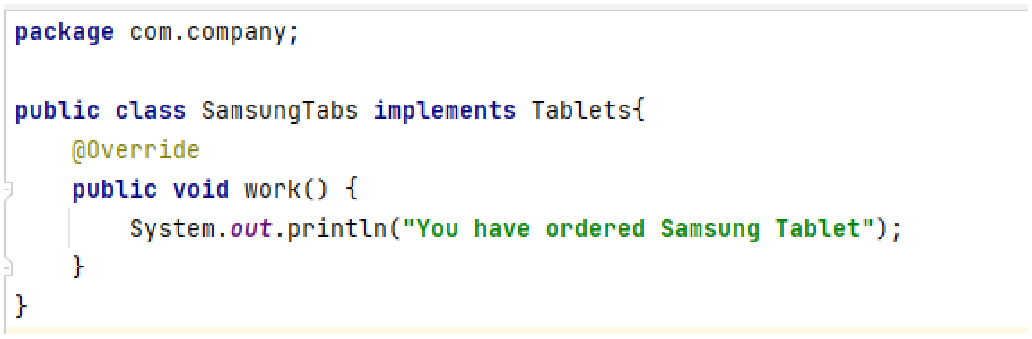
```java
package com.company;

public class LenovoTabs implements Tablets{
    @Override
    public void work() {
        System.out.println("You have ordered Lenovo Tablet");
    }
}
```

LenovoPhones.java:



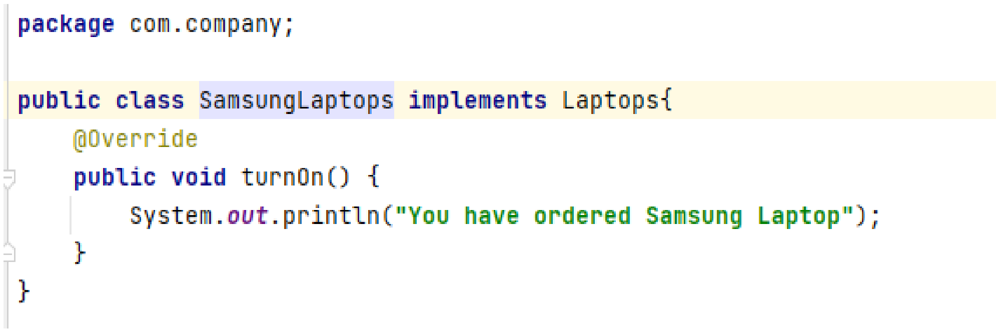
```java
package com.company;

public class LenovoPhones implements Phones{
    @Override
    public void call() {
        System.out.println("You have ordered Lenovo Phone");
    }
}
```

SamsungTabs.java:



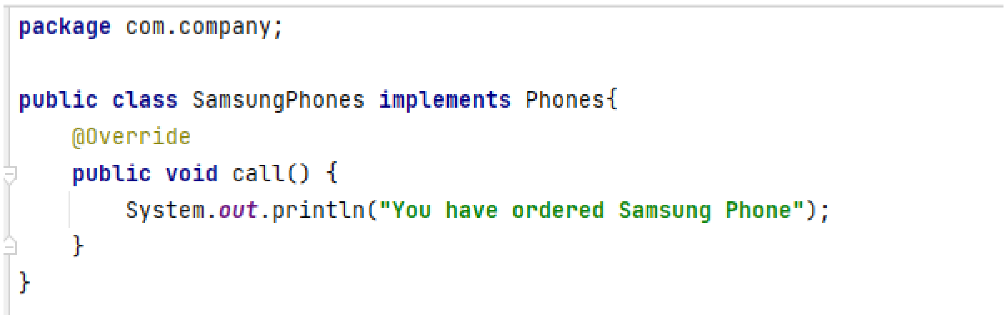
```java
package com.company;

public class SamsungTabs implements Tablets{
    @Override
    public void work() {
        System.out.println("You have ordered Samsung Tablet");
    }
}
```

SamsungLaptops.java:



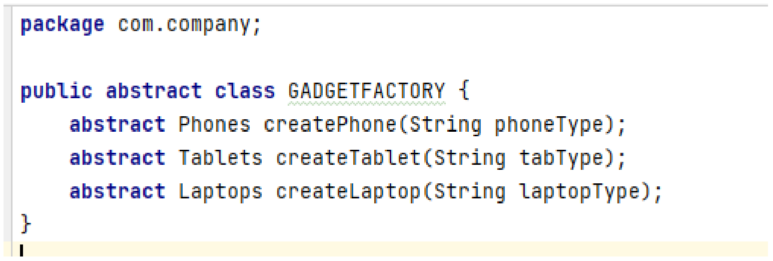
```java
package com.company;

public class SamsungLaptops implements Laptops{
    @Override
    public void turnOn() {
        System.out.println("You have ordered Samsung Laptop");
    }
}
```

SamsungPhones.java:



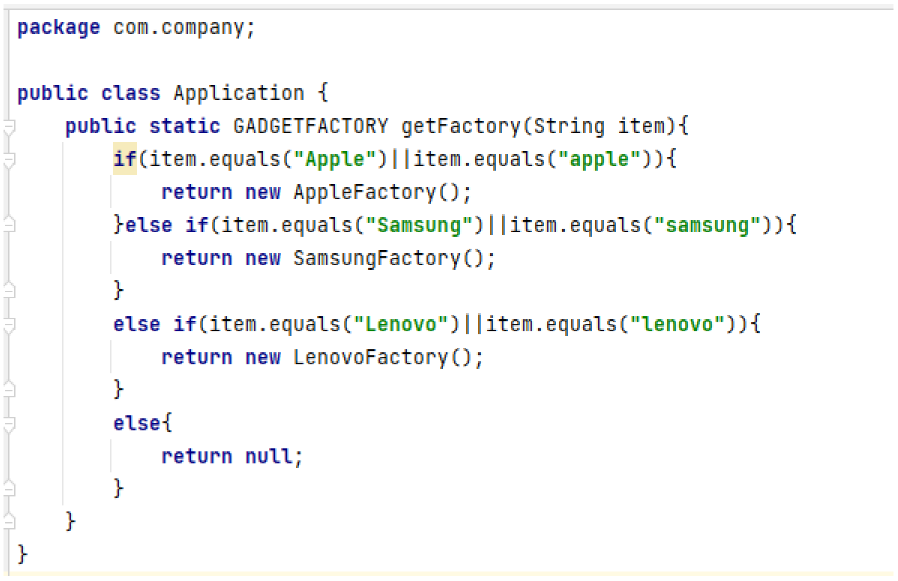
```java
package com.company;

public class SamsungPhones implements Phones{
    @Override
    public void call() {
        System.out.println("You have ordered Samsung Phone");
    }
}
```

GADGETFACTORY.java:



```java
package com.company;

public abstract class GADGETFACTORY {
    abstract Phones createPhone(String phoneType);
    abstract Tablets createTablet(String tabType);
    abstract Laptops createLaptop(String laptopType);
}
```

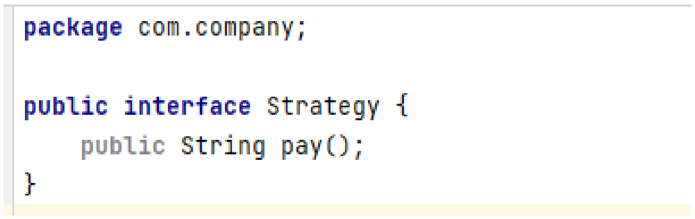
Application.java:



```java
package com.company;

public class Application {
    public static GADGETFACTORY getFactory(String item){
        if(item.equals("Apple")||item.equals("apple")){
            return new AppleFactory();
        }else if(item.equals("Samsung")||item.equals("samsung")){
            return new SamsungFactory();
        }
        else if(item.equals("Lenovo")||item.equals("lenovo")){
            return new LenovoFactory();
        }
        else{
            return null;
        }
    }
}
```

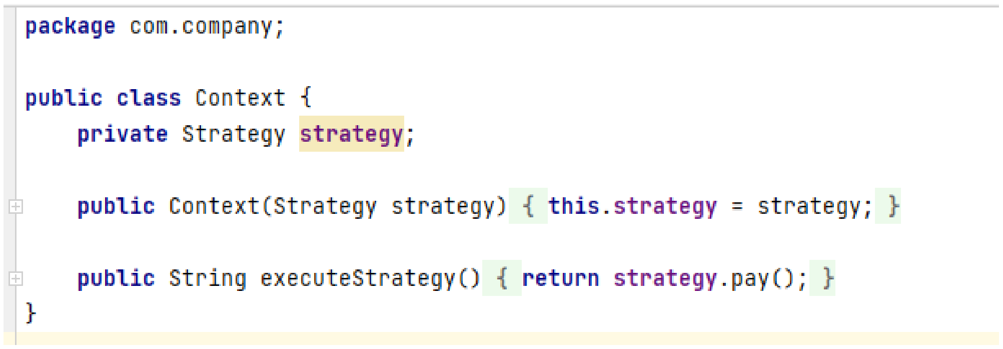
Strategy Pattern:

Strategy.java (interface):



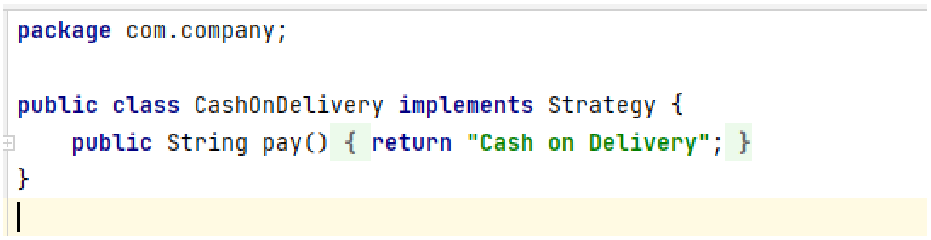
```java
package com.company;

public interface Strategy {
    public String pay();
}
```

Context.java:



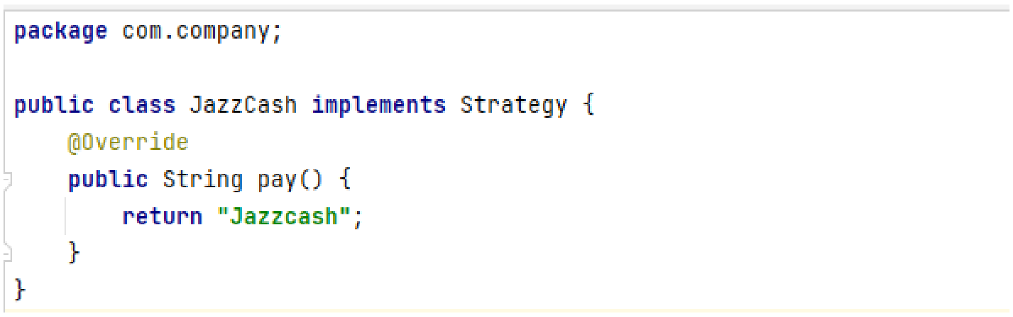
```java
package com.company;

public class Context {
    private Strategy strategy;

    public Context(Strategy strategy) { this.strategy = strategy; }

    public String executeStrategy() { return strategy.pay(); }
}
```

CashOnDelivery.java:



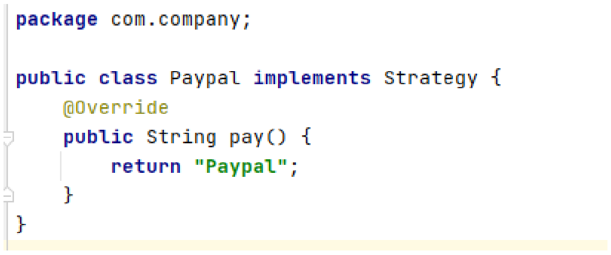
```java
package com.company;

public class CashOnDelivery implements Strategy {
    public String pay() { return "Cash on Delivery"; }
}
```

Jazzcash.java:



```java
package com.company;

public class JazzCash implements Strategy {
    @Override
    public String pay() {
        return "Jazzcash";
    }
}
```

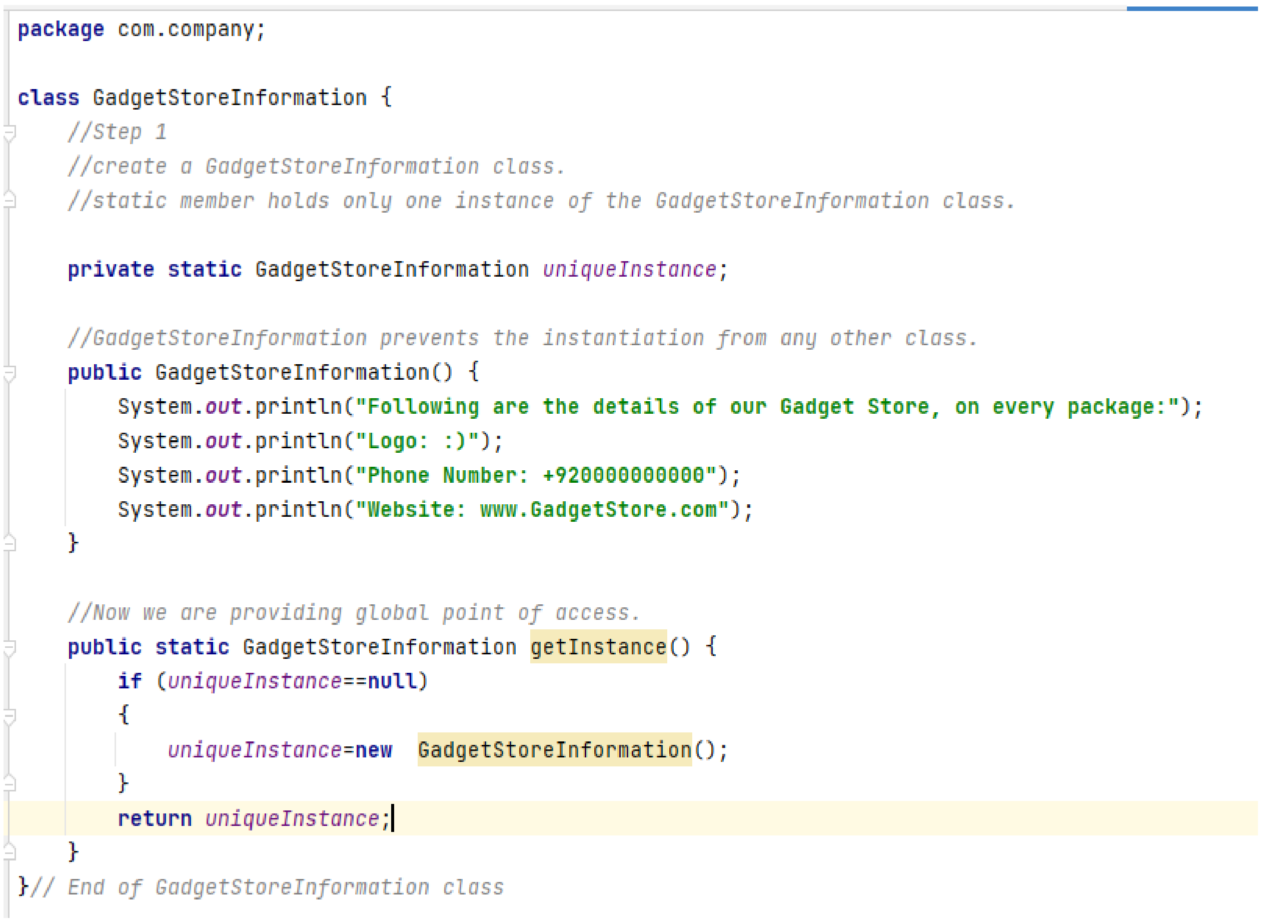
Paypal.java:



```java
package com.company;

public class Paypal implements Strategy {
    @Override
    public String pay() {
        return "Paypal";
    }
}
```

Singleton Pattern:

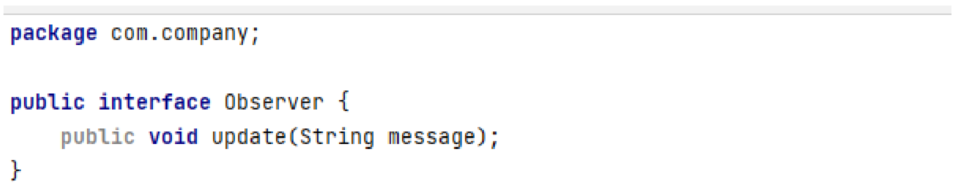
GadgetStoreInformation.java:



```java
package com.company;

class GadgetStoreInformation {
    //Step 1
    //create a GadgetStoreInformation class.
    //static member holds only one instance of the GadgetStoreInformation class.

    private static GadgetStoreInformation uniqueInstance;

    //GadgetStoreInformation prevents the instantiation from any other class.
    public GadgetStoreInformation() {
        System.out.println("Following are the details of our Gadget Store, on every package:");
        System.out.println("Logo: :)");
        System.out.println("Phone Number: +920000000000");
        System.out.println("Website: www.GadgetStore.com");
    }

    //Now we are providing global point of access.
    public static GadgetStoreInformation getInstance() {
        if (uniqueInstance==null)
        {
            uniqueInstance=new  GadgetStoreInformation();
        }
        return uniqueInstance;
    }
}// End of GadgetStoreInformation class
```

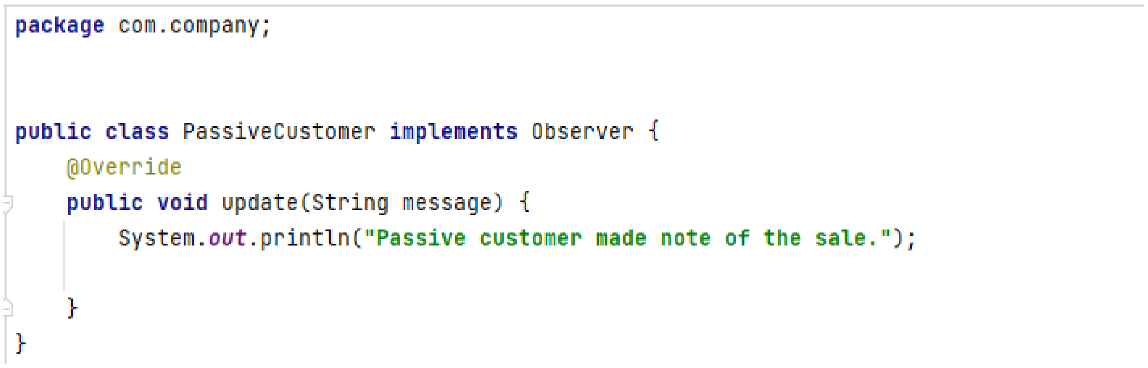
Observer Pattern:

Observer.java (interface):



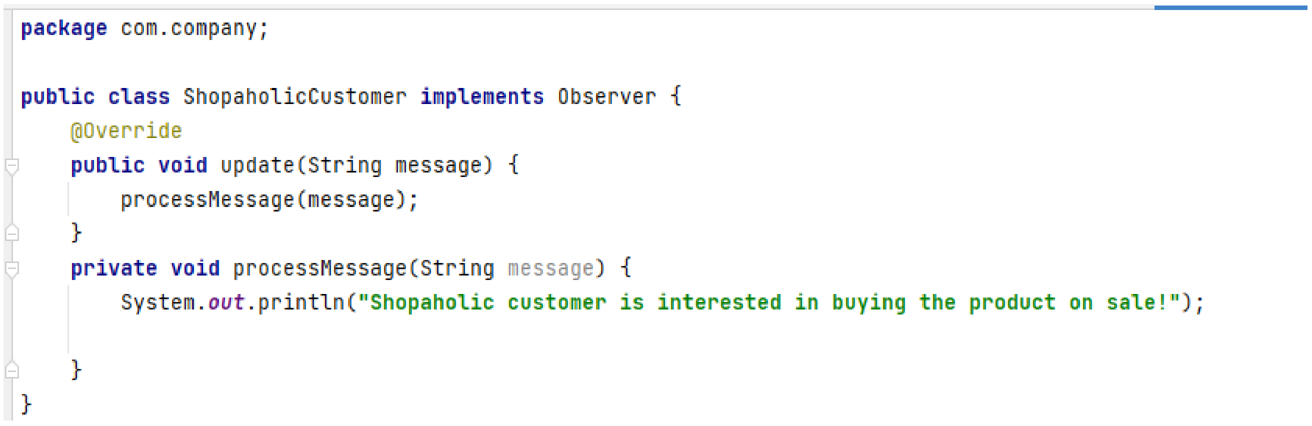
```
package com.company;

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

PassiveCustomer.java:



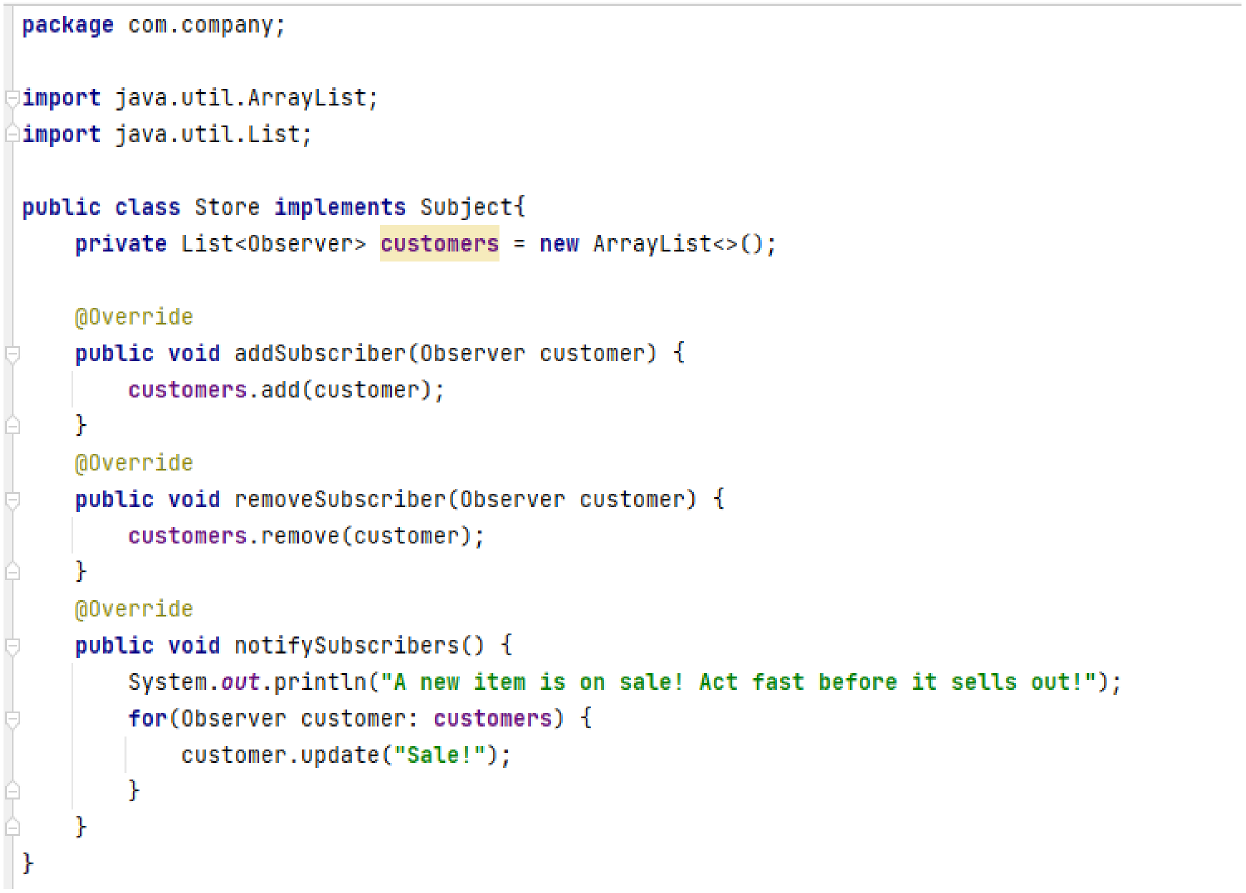
```
package com.company;


public class PassiveCustomer implements Observer {
    @Override
    public void update(String message) {
        System.out.println("Passive customer made note of the sale.");

    }
}
```

ShopaholicCustomer.java:



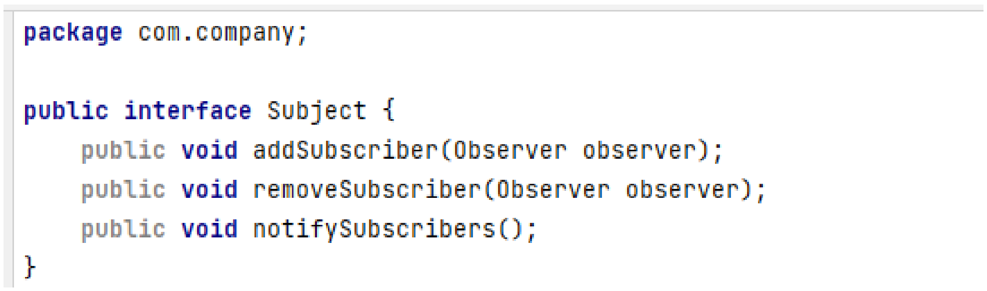
```
package com.company;

public class ShopaholicCustomer implements Observer {
    @Override
    public void update(String message) {
        processMessage(message);
    }
    private void processMessage(String message) {
        System.out.println("Shopaholic customer is interested in buying the product on sale!");

    }
}
```

Store.java:



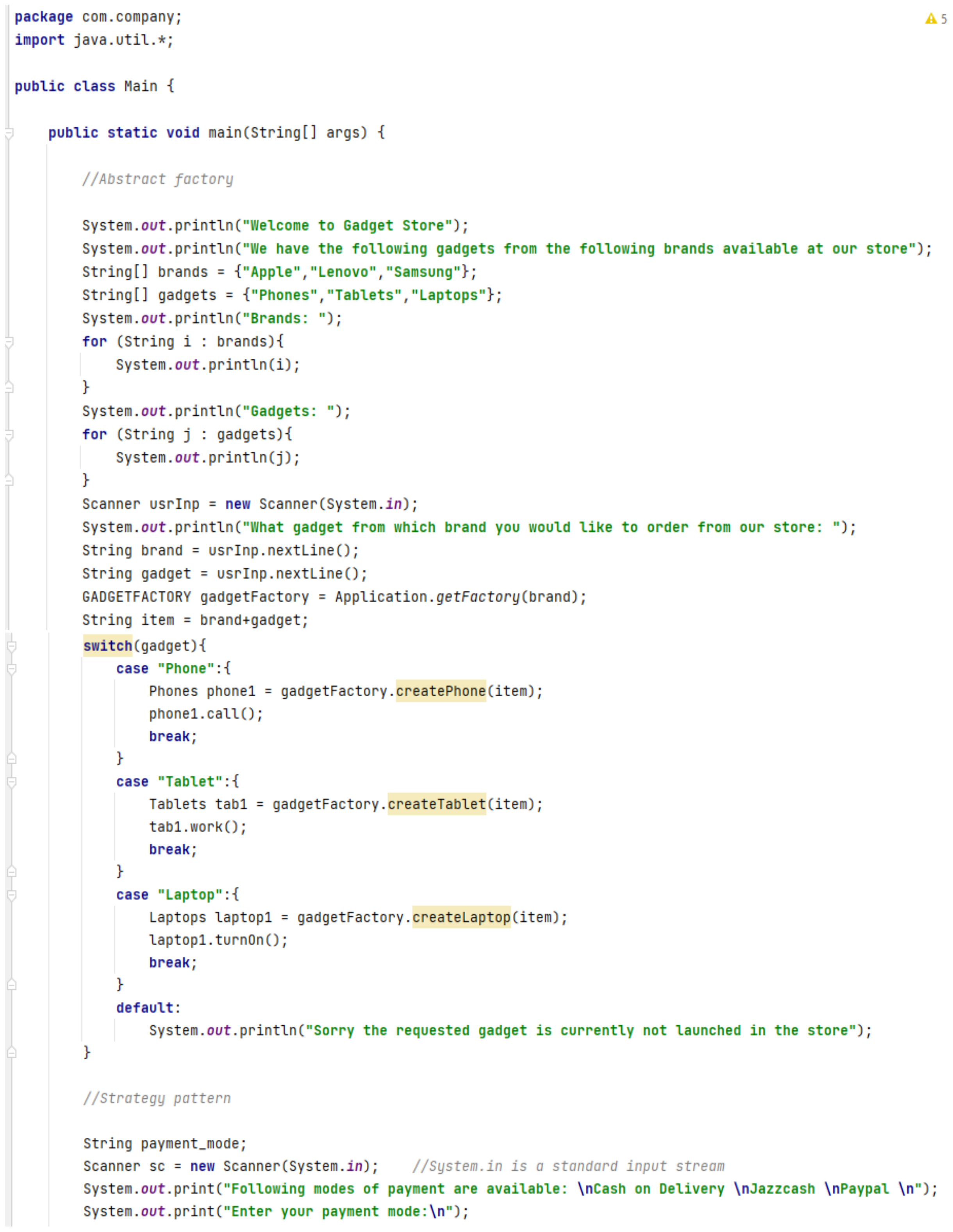
```java
package com.company;

import java.util.ArrayList;
import java.util.List;

public class Store implements Subject{
    private List<Observer> customers = new ArrayList<>();

    @Override
    public void addSubscriber(Observer customer) {
        customers.add(customer);
    }
    @Override
    public void removeSubscriber(Observer customer) {
        customers.remove(customer);
    }
    @Override
    public void notifySubscribers() {
        System.out.println("A new item is on sale! Act fast before it sells out!");
        for(Observer customer: customers) {
            customer.update("Sale!");
        }
    }
}
```

Subject.java (interface):

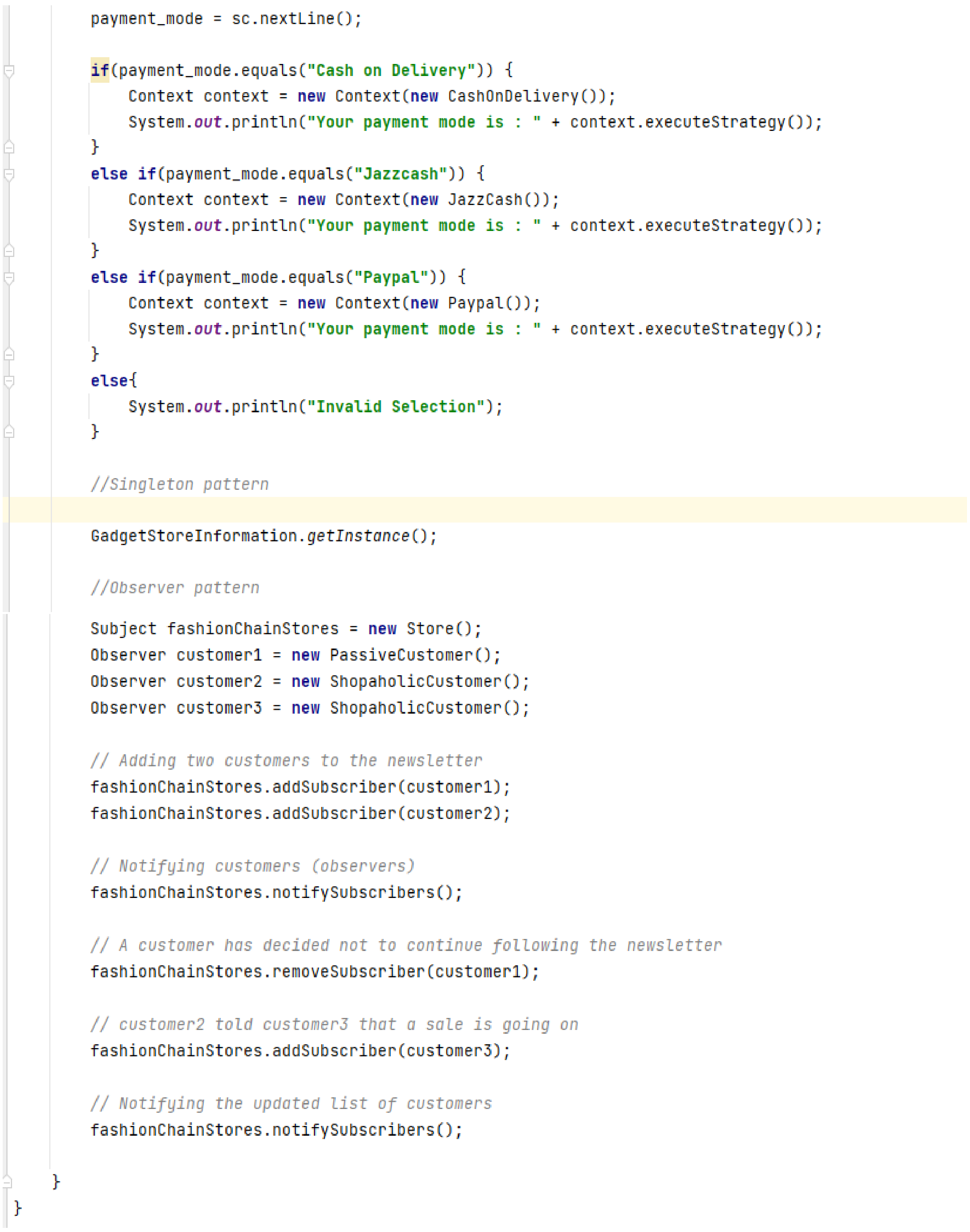


```java
package com.company;

public interface Subject {
    public void addSubscriber(Observer observer);
    public void removeSubscriber(Observer observer);
    public void notifySubscribers();
}
```

MAIN CLASS:

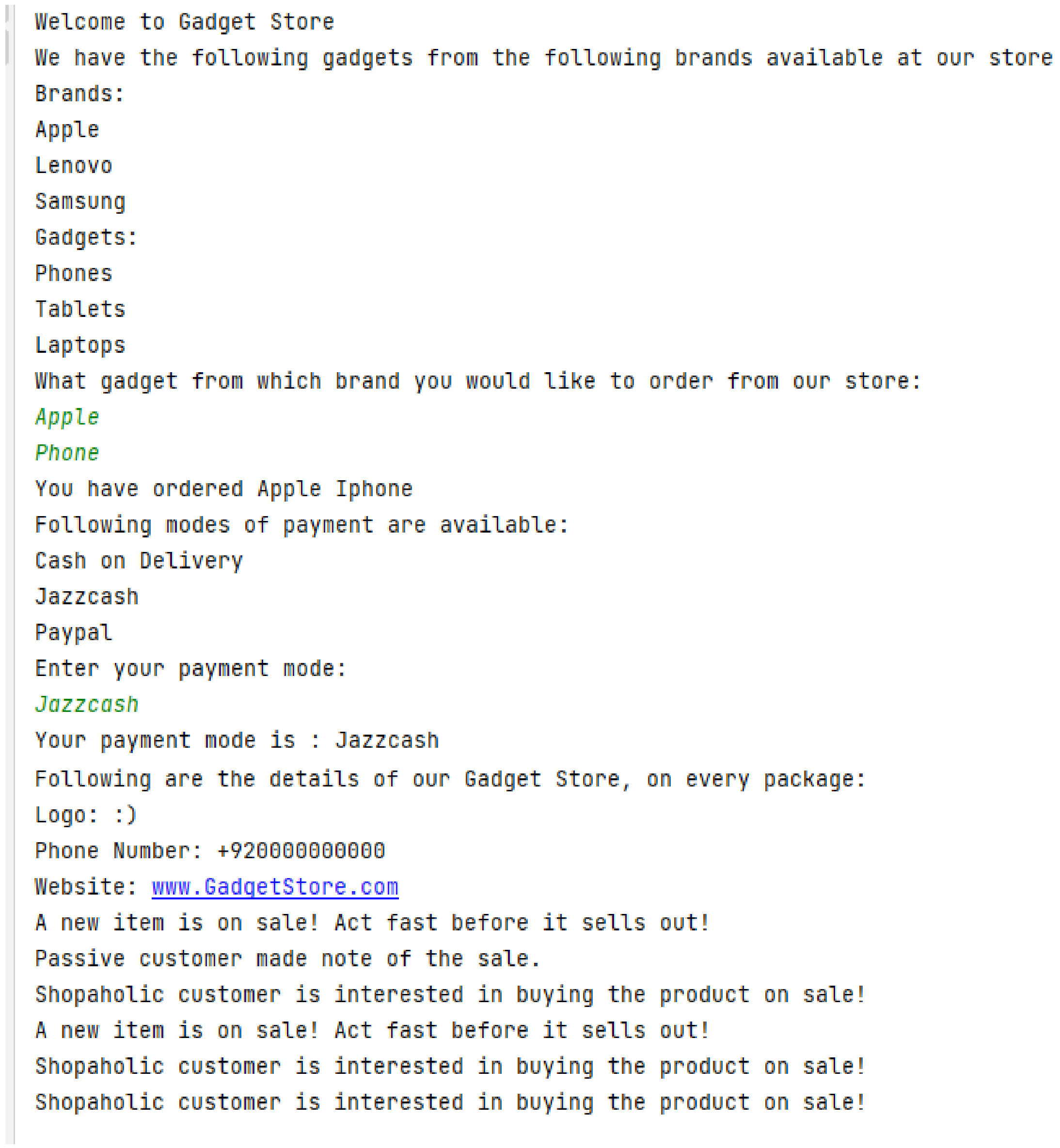


```java
package com.company;
import java.util.*;

public class Main {

    public static void main(String[] args) {

        //Abstract factory

        System.out.println("Welcome to Gadget Store");
        System.out.println("We have the following gadgets from the following brands available at our store");
        String[] brands = {"Apple","Lenovo","Samsung"};
        String[] gadgets = {"Phones","Tablets","Laptops"};
        System.out.println("Brands: ");
        for (String i : brands){
            System.out.println(i);
        }
        System.out.println("Gadgets: ");
        for (String j : gadgets){
            System.out.println(j);
        }
        Scanner usrInp = new Scanner(System.in);
        System.out.println("What gadget from which brand you would like to order from our store: ");
        String brand = usrInp.nextLine();
        String gadget = usrInp.nextLine();
        GADGETFACTORY gadgetFactory = Application.getFactory(brand);
        String item = brand+gadget;
        switch(gadget){
            case "Phone":{
                Phones phone1 = gadgetFactory.createPhone(item);
                phone1.call();
                break;
            }
            case "Tablet":{
                Tablets tab1 = gadgetFactory.createTablet(item);
                tab1.work();
                break;
            }
            case "Laptop":{
                Laptops laptop1 = gadgetFactory.createLaptop(item);
                laptop1.turnOn();
                break;
            }
            default:
                System.out.println("Sorry the requested gadget is currently not launched in the store");
        }

        //Strategy pattern

        String payment_mode;
        Scanner sc = new Scanner(System.in);    //System.in is a standard input stream
        System.out.print("Following modes of payment are available: \nCash on Delivery \nJazzcash \nPaypal \n");
        System.out.print("Enter your payment mode:\n");
```



```java
        payment_mode = sc.nextLine();

        if(payment_mode.equals("Cash on Delivery")) {
            Context context = new Context(new CashOnDelivery());
            System.out.println("Your payment mode is : " + context.executeStrategy());
        }
        else if(payment_mode.equals("Jazzcash")) {
            Context context = new Context(new JazzCash());
            System.out.println("Your payment mode is : " + context.executeStrategy());
        }
        else if(payment_mode.equals("Paypal")) {
            Context context = new Context(new Paypal());
            System.out.println("Your payment mode is : " + context.executeStrategy());
        }
        else{
            System.out.println("Invalid Selection");
        }

        //Singleton pattern

        GadgetStoreInformation.getInstance();

        //Observer pattern

        Subject fashionChainStores = new Store();
        Observer customer1 = new PassiveCustomer();
        Observer customer2 = new ShopaholicCustomer();
        Observer customer3 = new ShopaholicCustomer();

        // Adding two customers to the newsletter
        fashionChainStores.addSubscriber(customer1);
        fashionChainStores.addSubscriber(customer2);

        // Notifying customers (observers)
        fashionChainStores.notifySubscribers();

        // A customer has decided not to continue following the newsletter
        fashionChainStores.removeSubscriber(customer1);

        // customer2 told customer3 that a sale is going on
        fashionChainStores.addSubscriber(customer3);

        // Notifying the updated list of customers
        fashionChainStores.notifySubscribers();

    }
}
```

10.OUTPUT:



```
Welcome to Gadget Store
We have the following gadgets from the following brands available at our store
Brands:
Apple
Lenovo
Samsung
Gadgets:
Phones
Tablets
Laptops
What gadget from which brand you would like to order from our store:
Apple
Phone
You have ordered Apple Iphone
Following modes of payment are available:
Cash on Delivery
Jazzcash
Paypal
Enter your payment mode:
Jazzcash
Your payment mode is : Jazzcash
Following are the details of our Gadget Store, on every package:
Logo: :)
Phone Number: +920000000000
Website: www.GadgetStore.com
A new item is on sale! Act fast before it sells out!
Passive customer made note of the sale.
Shopaholic customer is interested in buying the product on sale!
A new item is on sale! Act fast before it sells out!
Shopaholic customer is interested in buying the product on sale!
Shopaholic customer is interested in buying the product on sale!
```