Name: Aravind Anand
Student ID: 030821269
Title: Assignment Unit 4 – Behavioral Design Pattern

Question 1: Find a compelling scenario where you can apply the template method design pattern to the food delivery system. Draw the corresponding class diagram and sequence diagram and implement in either Java or Python (50 points)

Template Method Design Pattern:

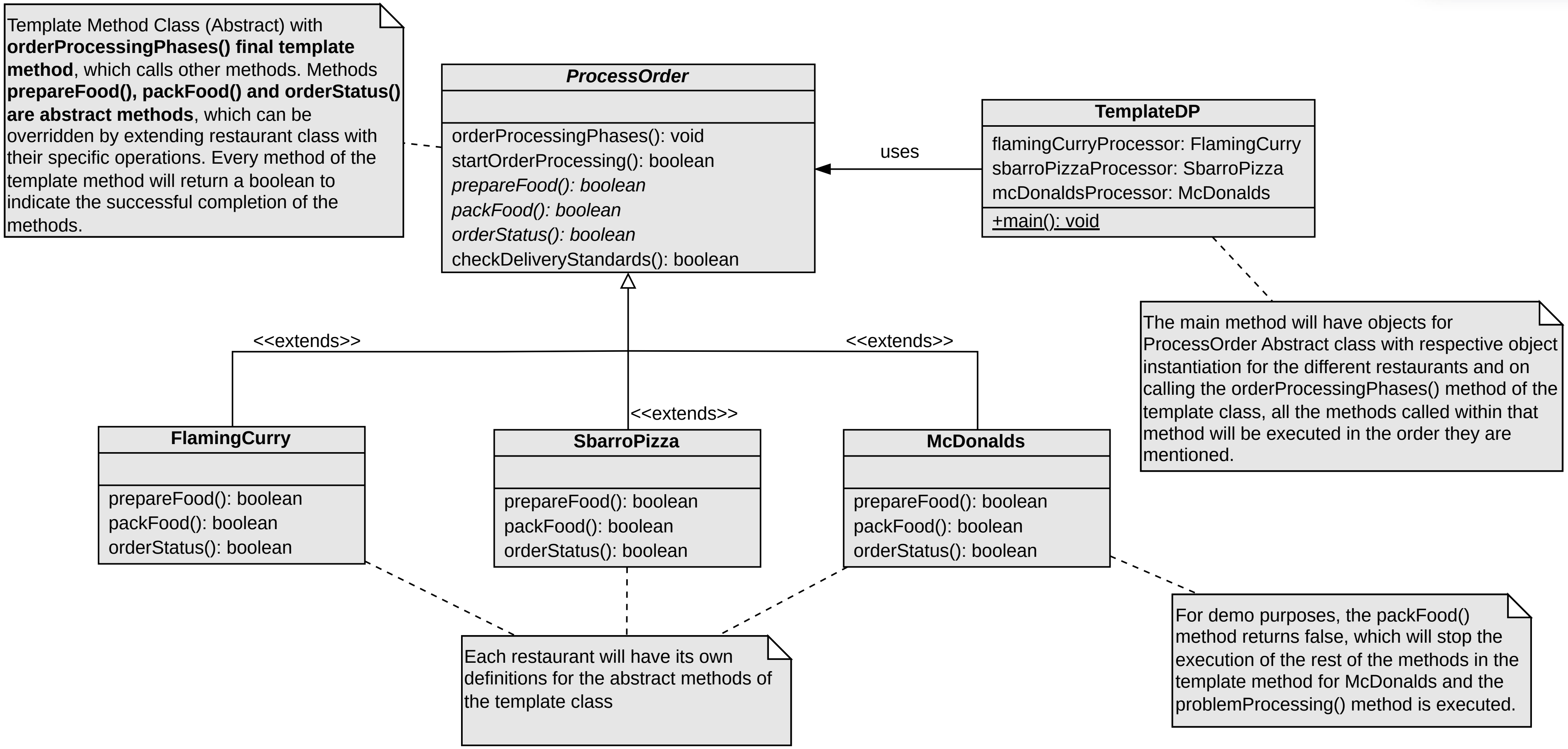
The template method design pattern (a behavioral design pattern) aids in creating a set of methods that can be used as a template for different scenarios where each scenario requires all the methods in the template to be executed. In other terms, it creates a framework for a set of methods in a parent class and the child classes can have their own implementation without altering the overall structure.

In the food delivery system, after processing the payment, there are a set of common phases involved in processing the order, but each restaurant has its own way of implementing those phases. So, the below scenario of template method design pattern has a template method - orderProcessingPhases() in the parent class (superclass) ProcessOrder, which has a set of abstract methods - prepareFood(), packFood(), orderStatus() and common methods - startOrderProcessing(), checkDeliveryStandards(). The abstract methods are overridden by different restaurant child classes (subclasses) such as FlamingCurry, SbarroPizza, and McDonalds.

In addition, the methods in the template method class validate whether each phase is successfully completed, and if there is any problem in the middle of any phase, that must be addressed by the application. To implement this, the return type of the methods is set to boolean and on returning true in one phase the next phase will be executed. If there was any issue with one phase, it will return false, and execution of the next phase will be stopped, and a problem processing procedure must be implemented. For example, assuming that there is an issue with the packFood() phase of McDonald's order, the method is returning false, so, problemProcessing() is called, and the remaining phases in the template method - orderStatus() and checkDeliveryStandards() methods will not be executed. This scenario is explained below with the respective class diagram, sequence diagram, and code snippets.

Note: The restaurant names in the example are used for educational purposes only.

Question 1: Template Method Design Pattern - Class Diagram



Template Method Class (Abstract) with **orderProcessingPhases() final template method**, which calls other methods. Methods **prepareFood(), packFood() and orderStatus() are abstract methods**, which can be overridden by extending restaurant class with their specific operations. Every method of the template method will return a boolean to indicate the successful completion of the methods.

ProcessOrder

orderProcessingPhases(): void
startOrderProcessing(): boolean
prepareFood(): boolean
packFood(): boolean
orderStatus(): boolean
checkDeliveryStandards(): boolean

TemplateDP

flamingCurryProcessor: FlamingCurry
sbarroPizzaProcessor: SbarroPizza
mcDonaldsProcessor: McDonalds

+main(): void

The main method will have objects for ProcessOrder Abstract class with respective object instantiation for the different restaurants and on calling the orderProcessingPhases() method of the template class, all the methods called within that method will be executed in the order they are mentioned.

FlamingCurry

prepareFood(): boolean
packFood(): boolean
orderStatus(): boolean

SbarroPizza

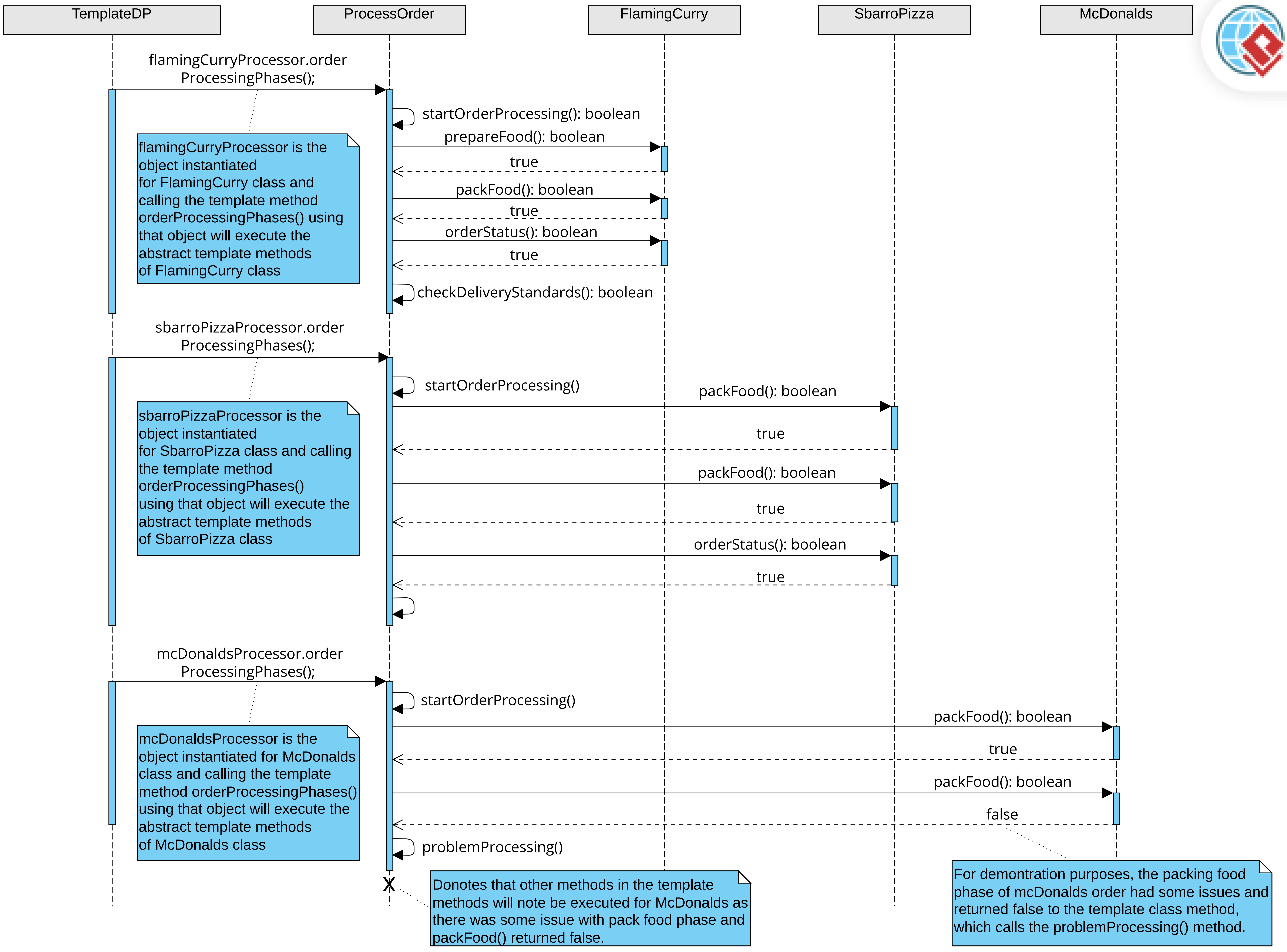
prepareFood(): boolean
packFood(): boolean
orderStatus(): boolean

McDonalds

prepareFood(): boolean
packFood(): boolean
orderStatus(): boolean

Each restaurant will have its own definitions for the abstract methods of the template class

For demo purposes, the packFood() method returns false, which will stop the execution of the rest of the methods in the template method for McDonalds and the problemProcessing() method is executed.



flamingCurryProcessor is the object instantiated for FlamingCurry class and calling the template method orderProcessingPhases() using that object will execute the abstract template methods of FlamingCurry class

sbarroPizzaProcessor is the object instantiated for SbarroPizza class and calling the template method orderProcessingPhases() using that object will execute the abstract template methods of SbarroPizza class

mcDonaldsProcessor is the object instantiated for McDonalds class and calling the template method orderProcessingPhases() using that object will execute the abstract template methods of McDonalds class

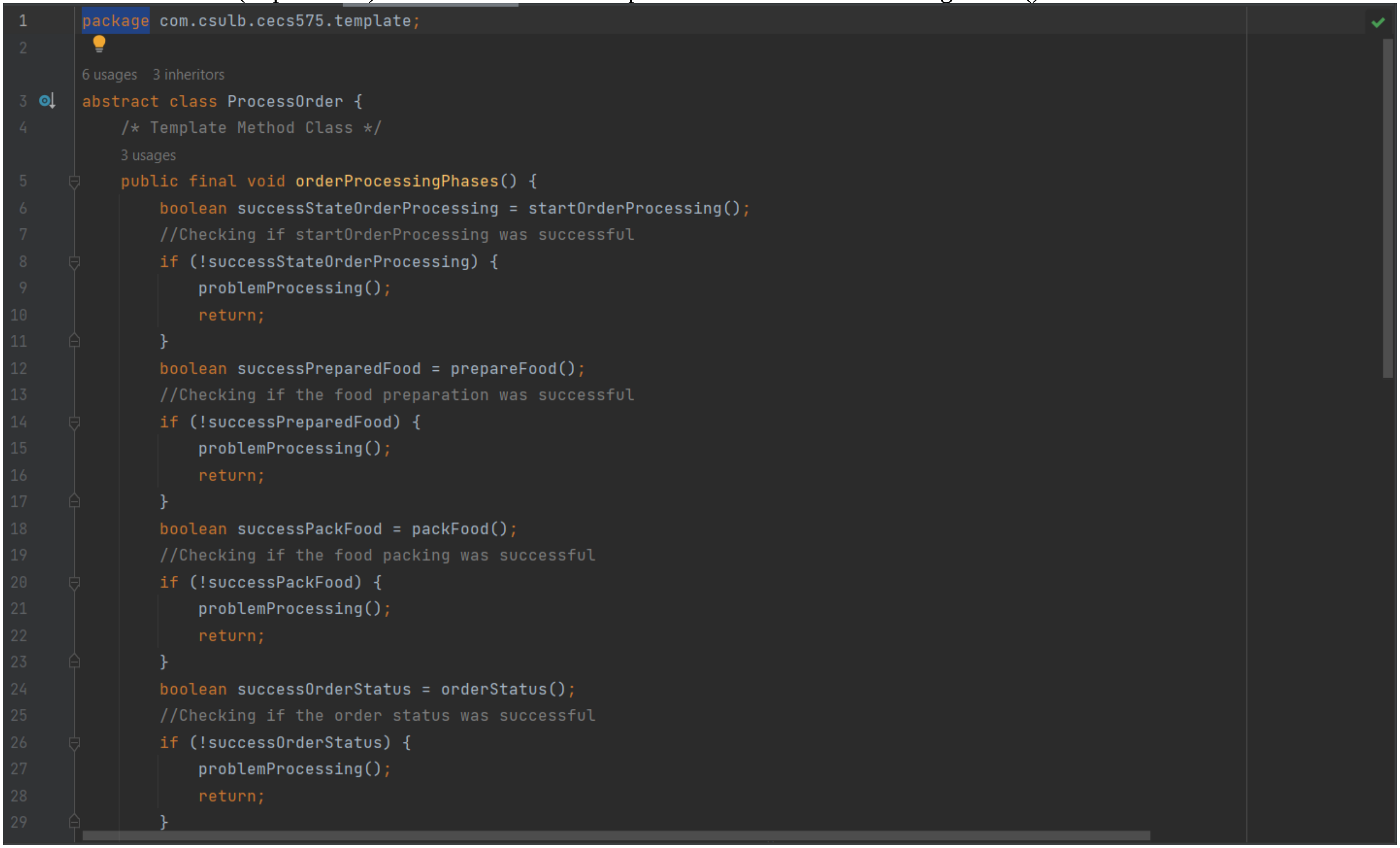
Donotes that other methods in the template methods will note be executed for McDonalds as there was some issue with pack food phase and packFood() returned false.

For demontration purposes, the packing food phase of mcDonalds order had some issues and returned false to the template class method, which calls the problemProcessing() method.

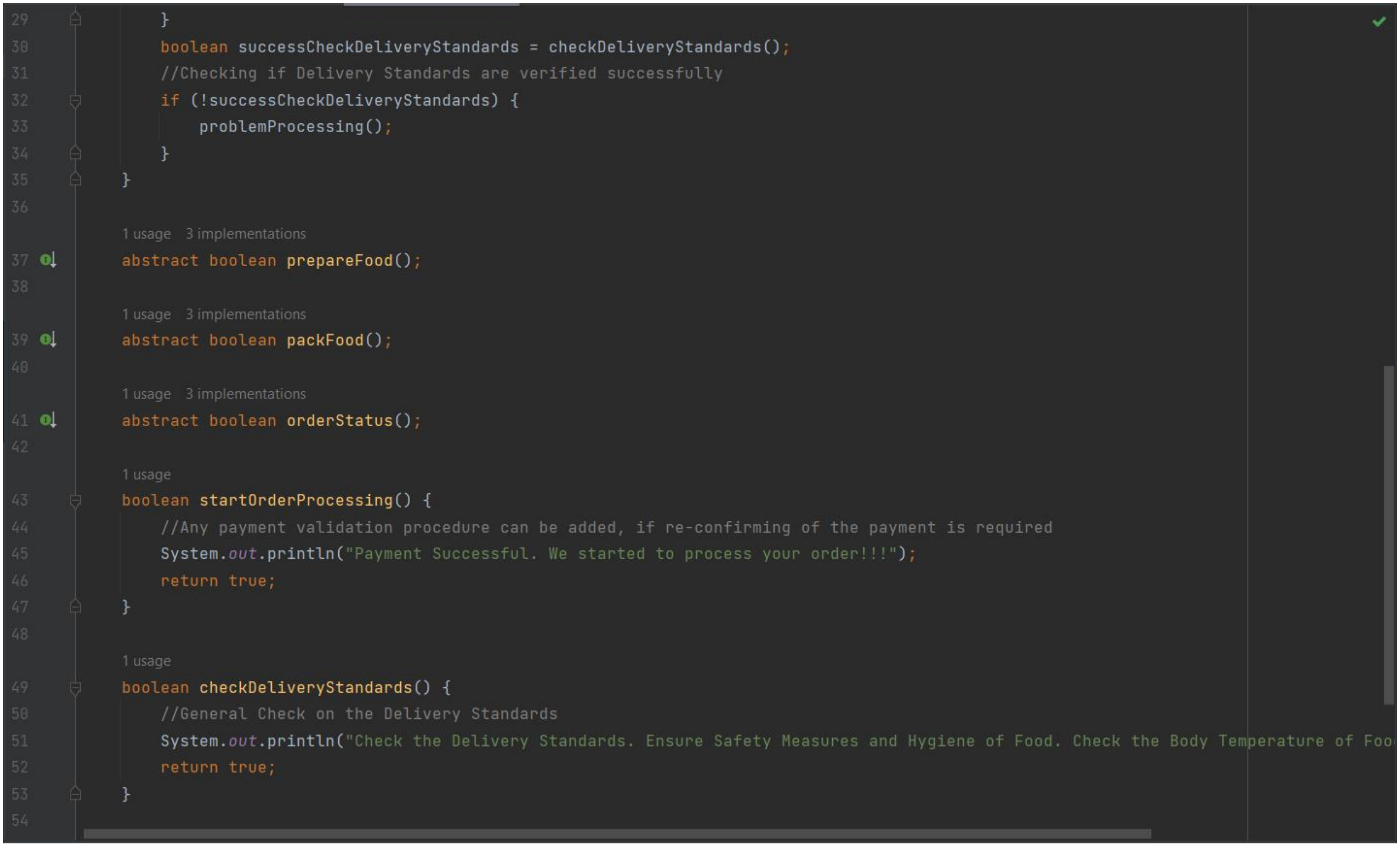
Question 1: Template Method Design Pattern - Sequence Diagram

Template Design Pattern: Order Processing – Code Snippets:

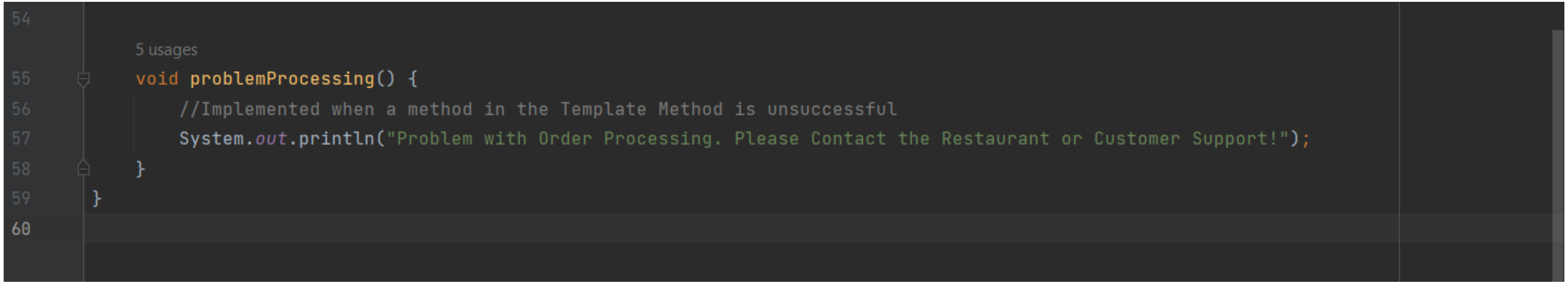
- Abstract Class (Super Class) ProcessOrder with Template Method - orderProcessingPhases() which has other methods:



```java
package com.csulb.cecs575.template;

abstract class ProcessOrder {
    /* Template Method Class */
    public final void orderProcessingPhases() {
        boolean successStateOrderProcessing = startOrderProcessing();
        //Checking if startOrderProcessing was successful
        if (!successStateOrderProcessing) {
            problemProcessing();
            return;
        }
        boolean successPreparedFood = prepareFood();
        //Checking if the food preparation was successful
        if (!successPreparedFood) {
            problemProcessing();
            return;
        }
        boolean successPackFood = packFood();
        //Checking if the food packing was successful
        if (!successPackFood) {
            problemProcessing();
            return;
        }
        boolean successOrderStatus = orderStatus();
        //Checking if the order status was successful
        if (!successOrderStatus) {
            problemProcessing();
            return;
        }
```

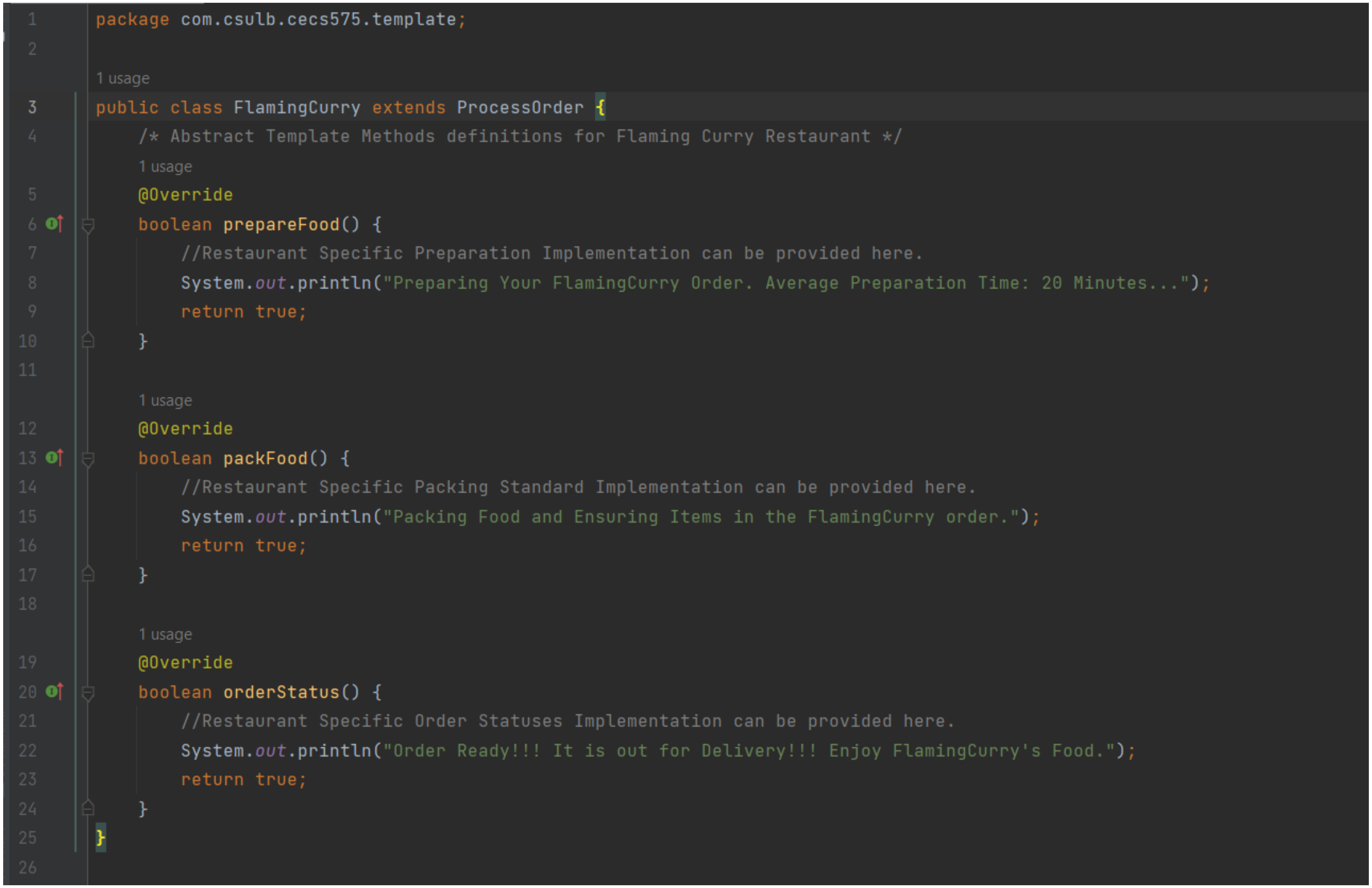


```java
        }
        boolean successCheckDeliveryStandards = checkDeliveryStandards();
        //Checking if Delivery Standards are verified successfully
        if (!successCheckDeliveryStandards) {
            problemProcessing();
        }
    }

    1 usage   3 implementations
    abstract boolean prepareFood();

    1 usage   3 implementations
    abstract boolean packFood();

    1 usage   3 implementations
    abstract boolean orderStatus();

    1 usage
    boolean startOrderProcessing() {
        //Any payment validation procedure can be added, if re-confirming of the payment is required
        System.out.println("Payment Successful. We started to process your order!!!");
        return true;
    }

    1 usage
    boolean checkDeliveryStandards() {
        //General Check on the Delivery Standards
        System.out.println("Check the Delivery Standards. Ensure Safety Measures and Hygiene of Food. Check the Body Temperature of Foo
        return true;
    }

```



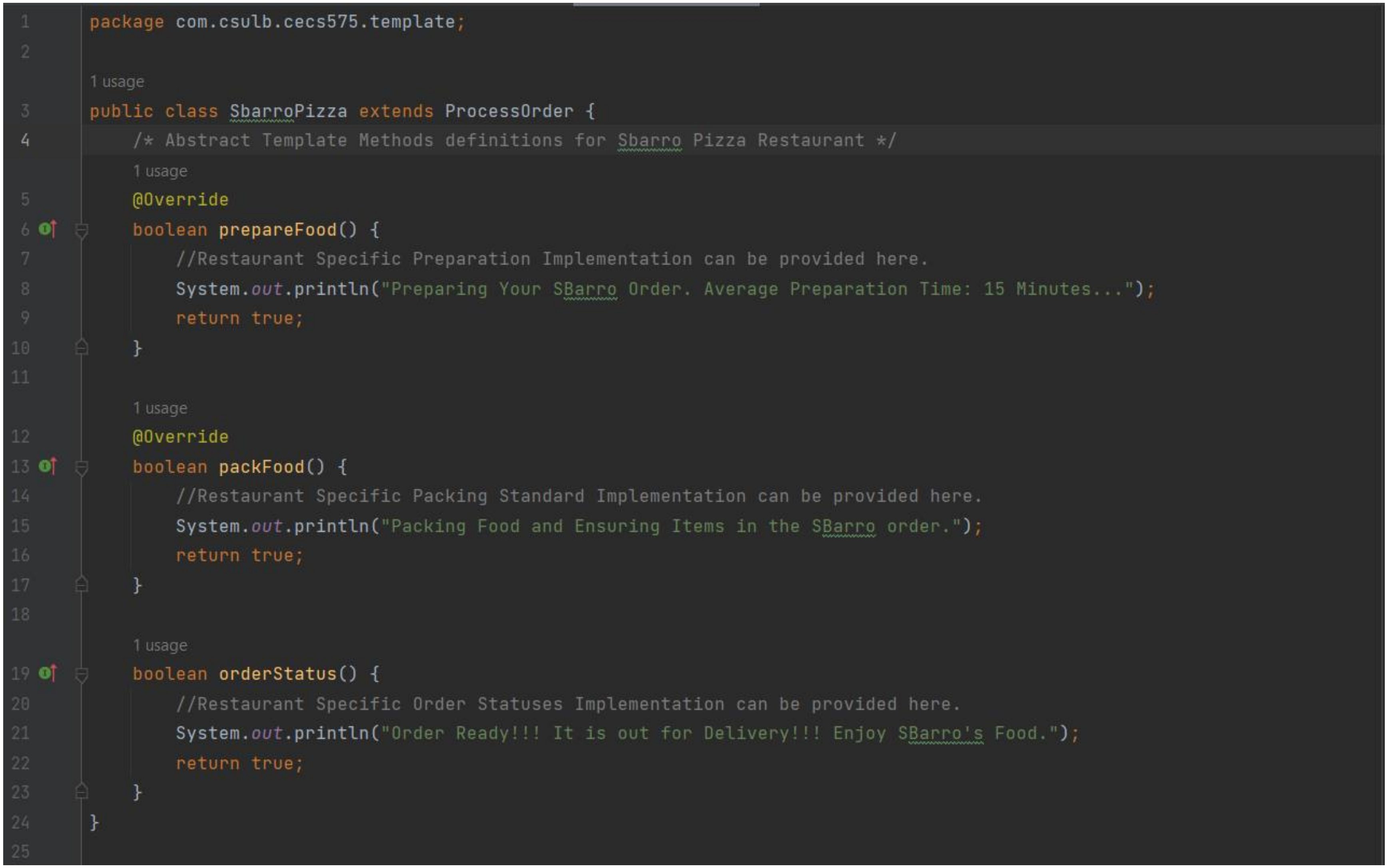
```java

    5 usages
    void problemProcessing() {
        //Implemented when a method in the Template Method is unsuccessful
        System.out.println("Problem with Order Processing. Please Contact the Restaurant or Customer Support!");
    }
}

```

- Abstract Methods defined for FlamingCurry Restaurant:



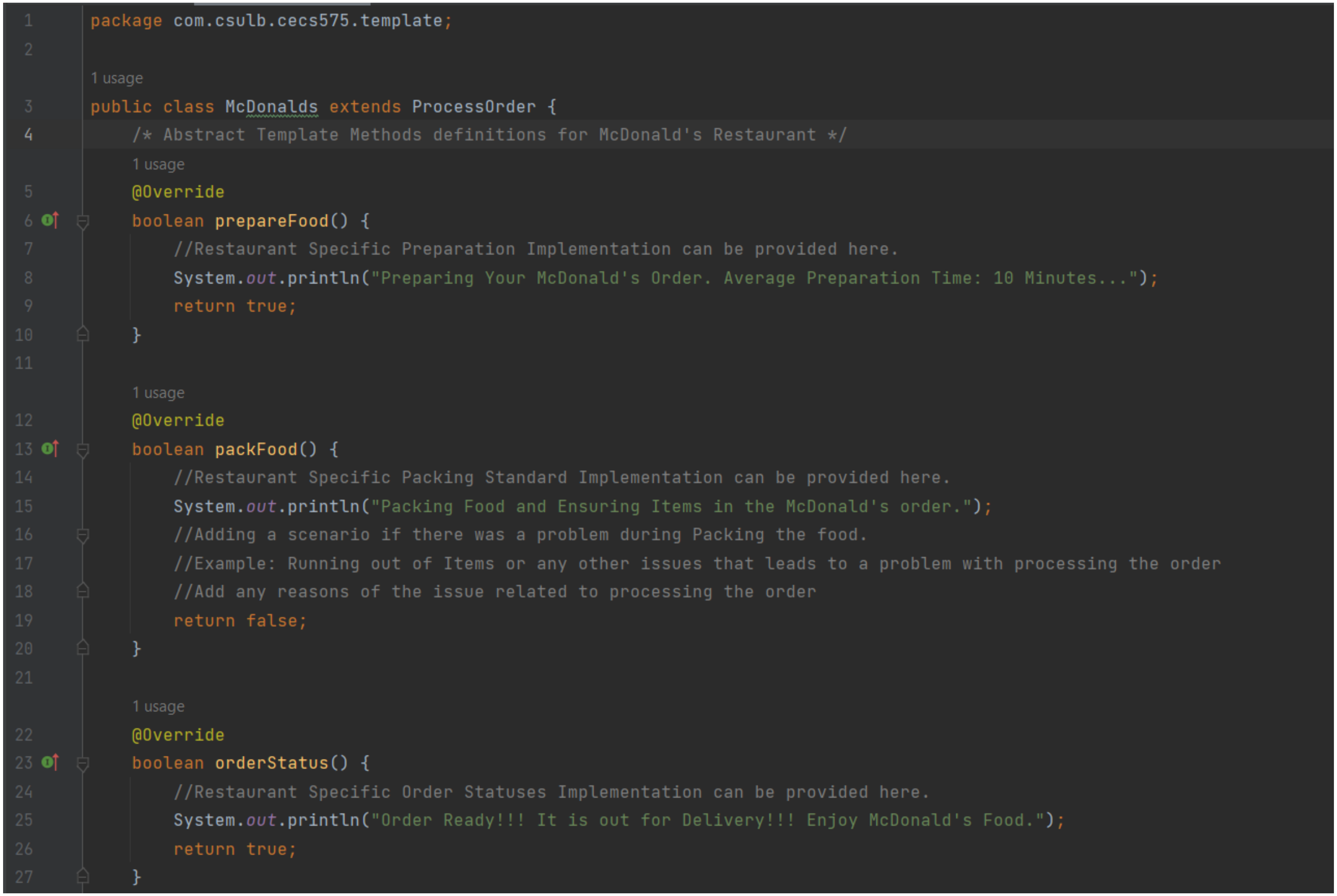
```
package com.csulb.cecs575.template;

1 usage
public class FlamingCurry extends ProcessOrder {
    /* Abstract Template Methods definitions for Flaming Curry Restaurant */
    1 usage
    @Override
    boolean prepareFood() {
        //Restaurant Specific Preparation Implementation can be provided here.
        System.out.println("Preparing Your FlamingCurry Order. Average Preparation Time: 20 Minutes...");
        return true;
    }

    1 usage
    @Override
    boolean packFood() {
        //Restaurant Specific Packing Standard Implementation can be provided here.
        System.out.println("Packing Food and Ensuring Items in the FlamingCurry order.");
        return true;
    }

    1 usage
    @Override
    boolean orderStatus() {
        //Restaurant Specific Order Statuses Implementation can be provided here.
        System.out.println("Order Ready!!! It is out for Delivery!!! Enjoy FlamingCurry's Food.");
        return true;
    }
}

```

- Abstract Methods defined for Sbarro Restaurant:



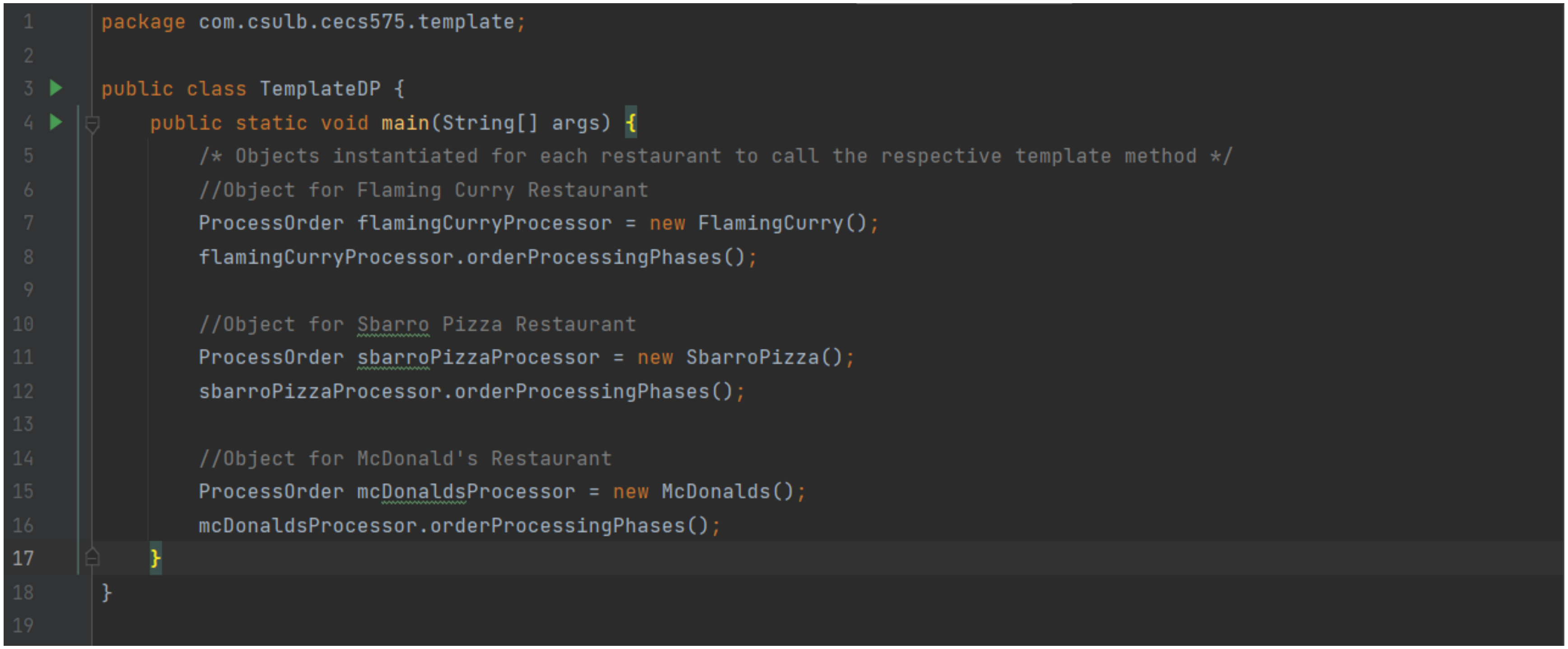
```java
package com.csulb.cecs575.template;

1 usage
public class SbarroPizza extends ProcessOrder {
    /* Abstract Template Methods definitions for Sbarro Pizza Restaurant */
    1 usage
    @Override
    boolean prepareFood() {
        //Restaurant Specific Preparation Implementation can be provided here.
        System.out.println("Preparing Your SBarro Order. Average Preparation Time: 15 Minutes...");
        return true;
    }

    1 usage
    @Override
    boolean packFood() {
        //Restaurant Specific Packing Standard Implementation can be provided here.
        System.out.println("Packing Food and Ensuring Items in the SBarro order.");
        return true;
    }

    1 usage
    boolean orderStatus() {
        //Restaurant Specific Order Statuses Implementation can be provided here.
        System.out.println("Order Ready!!! It is out for Delivery!!! Enjoy SBarro's Food.");
        return true;
    }
}

```

- Abstract Methods defined for McDonalds Restaurant with a negative scenario in the packFood() method, which will call the problemProcessing() method and stops the execution of other methods in the order of the Template method:



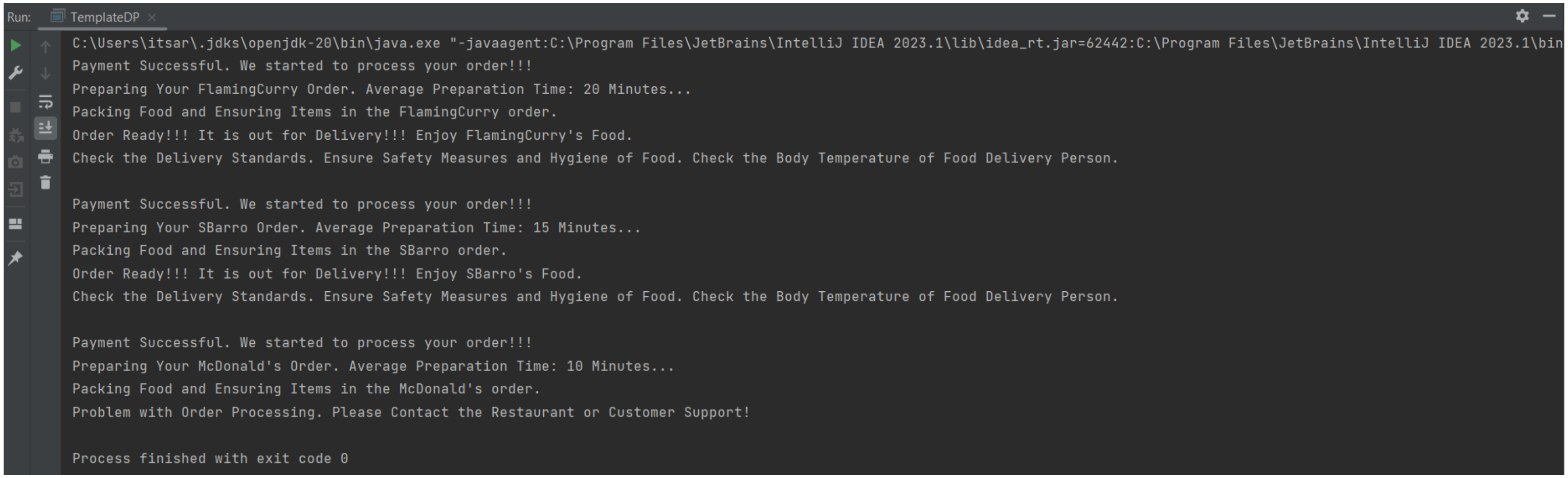
```java
package com.csulb.cecs575.template;

1 usage
public class McDonalds extends ProcessOrder {
    /* Abstract Template Methods definitions for McDonald's Restaurant */
    1 usage
    @Override
    boolean prepareFood() {
        //Restaurant Specific Preparation Implementation can be provided here.
        System.out.println("Preparing Your McDonald's Order. Average Preparation Time: 10 Minutes...");
        return true;
    }

    1 usage
    @Override
    boolean packFood() {
        //Restaurant Specific Packing Standard Implementation can be provided here.
        System.out.println("Packing Food and Ensuring Items in the McDonald's order.");
        //Adding a scenario if there was a problem during Packing the food.
        //Example: Running out of Items or any other issues that leads to a problem with processing the order
        //Add any reasons of the issue related to processing the order
        return false;
    }

    1 usage
    @Override
    boolean orderStatus() {
        //Restaurant Specific Order Statuses Implementation can be provided here.
        System.out.println("Order Ready!!! It is out for Delivery!!! Enjoy McDonald's Food.");
        return true;
    }
```

- TemplateDP class with the main method that has the object instantiation for the different restaurants and calls the template method for those restaurants using their respective objects:



```java
package com.csulb.cecs575.template;

public class TemplateDP {
    public static void main(String[] args) {
        /* Objects instantiated for each restaurant to call the respective template method */
        //Object for Flaming Curry Restaurant
        ProcessOrder flamingCurryProcessor = new FlamingCurry();
        flamingCurryProcessor.orderProcessingPhases();

        //Object for Sbarro Pizza Restaurant
        ProcessOrder sbarroPizzaProcessor = new SbarroPizza();
        sbarroPizzaProcessor.orderProcessingPhases();

        //Object for McDonald's Restaurant
        ProcessOrder mcDonaldsProcessor = new McDonalds();
        mcDonaldsProcessor.orderProcessingPhases();
    }
}
```

Output:



```
C:\Users\itsar\.jdks\openjdk-20\bin\java.exe "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 2023.1\lib\idea_rt.jar=62442:C:\Program Files\JetBrains\IntelliJ IDEA 2023.1\bin
Payment Successful. We started to process your order!!!
Preparing Your FlamingCurry Order. Average Preparation Time: 20 Minutes...
Packing Food and Ensuring Items in the FlamingCurry order.
Order Ready!!! It is out for Delivery!!! Enjoy FlamingCurry's Food.
Check the Delivery Standards. Ensure Safety Measures and Hygiene of Food. Check the Body Temperature of Food Delivery Person.

Payment Successful. We started to process your order!!!
Preparing Your SBarro Order. Average Preparation Time: 15 Minutes...
Packing Food and Ensuring Items in the SBarro order.
Order Ready!!! It is out for Delivery!!! Enjoy SBarro's Food.
Check the Delivery Standards. Ensure Safety Measures and Hygiene of Food. Check the Body Temperature of Food Delivery Person.

Payment Successful. We started to process your order!!!
Preparing Your McDonald's Order. Average Preparation Time: 10 Minutes...
Packing Food and Ensuring Items in the McDonald's order.
Problem with Order Processing. Please Contact the Restaurant or Customer Support!

Process finished with exit code 0
```

- The Template methods are successfully executed for FlamingCurry and Sbarro. For McDonalds, the assumption of a problem occuring with the packFood() phase (returning false) has stopped the execution of other methods of the template method for McDonalds and called the problemProcessing() method, which notifies the customer to contact the restaurant or customer support.

Question 2: Find a compelling scenario where you can apply the Observer Design Pattern to the food delivery system. Draw the corresponding class diagram and sequence diagram and implement in either Java or Python (50 points)

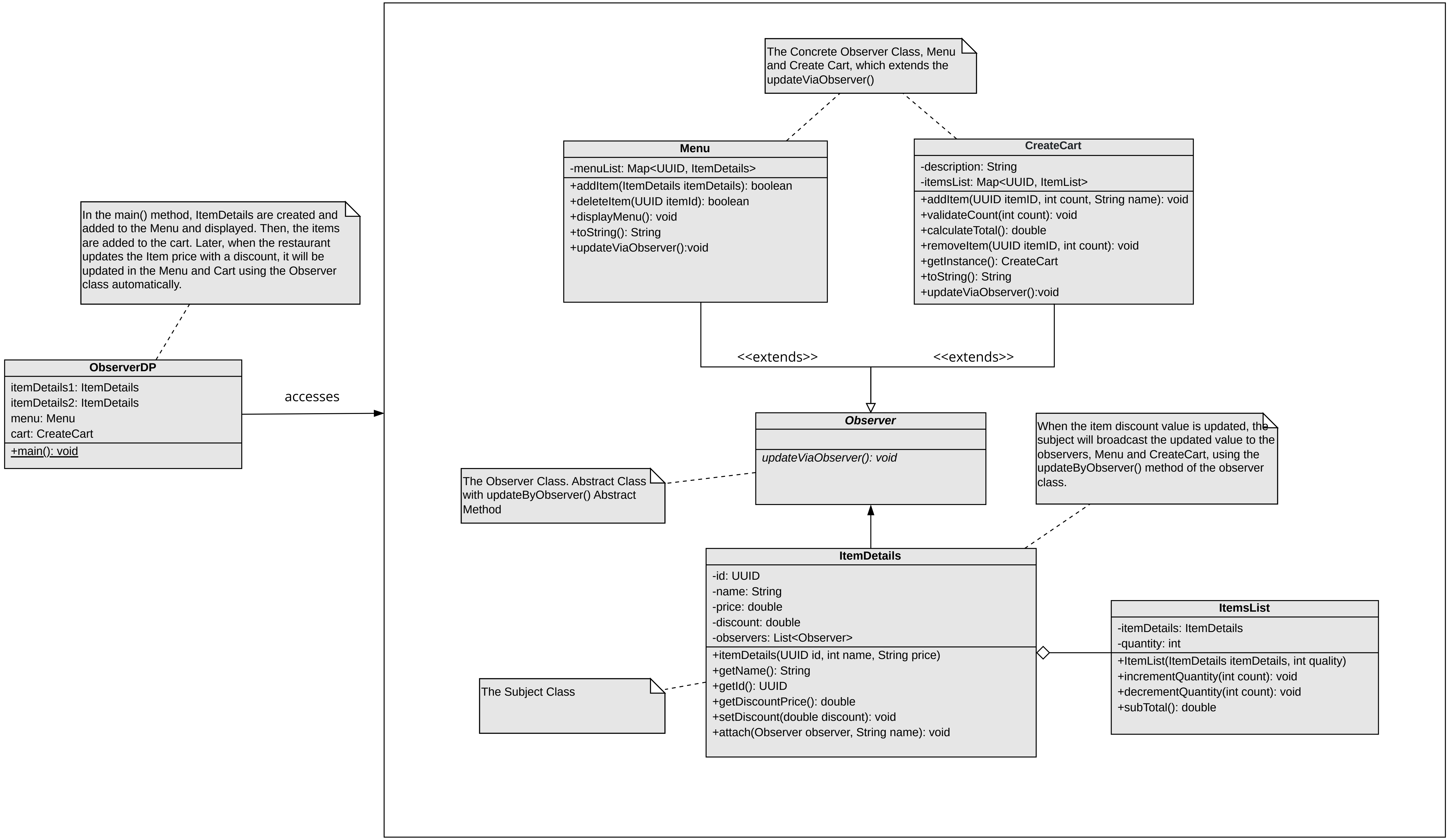
Observer Design Pattern:

The Observer pattern (a behavioral design pattern) defines a one-to-many dependency between objects so that when one object changes state, all its dependents are automatically notified and updated. In other words, it allows multiple objects to observe and react to changes in another object. It helps in maintaining the consistency of data. Instead of directly coupling the objects, the Observer pattern uses an intermediate object called the "Subject" that maintains a list of its dependents, called "Observers". When the Subject changes its state, it notifies all its Observers, triggering their corresponding update methods. In other words, the Observer pattern is implemented by defining two types of objects: the Subject and the Observer. The Subject maintains a list of its Observers and provides methods to attach, remove, and notify Observers. The Observers who need to listen to the Subject will get attached to the Subject and the Subject will update its list of Observers when there is a change.

In the food delivery system, the item details can be updated by the restaurant regularly, say with discounted pricing, increased pricing, etc. Some restaurants are offering discounted pricing on their items during some time frame in a day, say 6 PM – 9 PM. So, assuming a scenario where a restaurant has a Menu and the customer is in the process of cart creation using that menu and adding items to the cart, and when the restaurant adds a discount to the price of the items at that instance, the Menu of the Restaurant and the Cart of the customer must have the discounted price and the cart must perform the recalculation again.

An implementation of this scenario with ItemDetails as the Subject, and Menu and CreateCart as the Concrete Observer Classes which extends the Observer abstract class, involves the process of setting discounts to the item prices and that gets updated in the Menu and CreateCart automatically is explained below with the respective class diagram, sequence diagram, and code snippets.

Question 2: Observer Pattern - Class Diagram


Menu
-menuList: Map<UUID, ItemDetails>
+addItem(ItemDetails itemDetails): boolean
+deleteItem(UUID itemId): boolean
+displayMenu(): void
+toString(): String
+updateViaObserver():void
CreateCart
-description: String
-itemsList: Map<UUID, ItemList>
+addItem(UUID itemID, int count, String name): void
+validateCount(int count): void
+calculateTotal(): double
+removeItem(UUID itemID, int count): void
+getInstance(): CreateCart
+toString(): String
+updateViaObserver():void
<<extends>>
<<extends>>
ObserverDP
itemDetails1: ItemDetails
itemDetails2: ItemDetails
menu: Menu
cart: CreateCart
+main(): void
accesses
Observer
updateViaObserver(): void
ItemDetails
-id: UUID
-name: String
-price: double
-discount: double
-observers: List<Observer>
+itemDetails(UUID id, int name, String price)
+getName(): String
+getId(): UUID
+getDiscountPrice(): double
+setDiscount(double discount): void
+attach(Observer observer, String name): void
ItemsList
-itemDetails: ItemDetails
-quantity: int
+ItemList(ItemDetails itemDetails, int quality)
+incrementQuantity(int count): void
+decrementQuantity(int count): void
+subTotal(): double

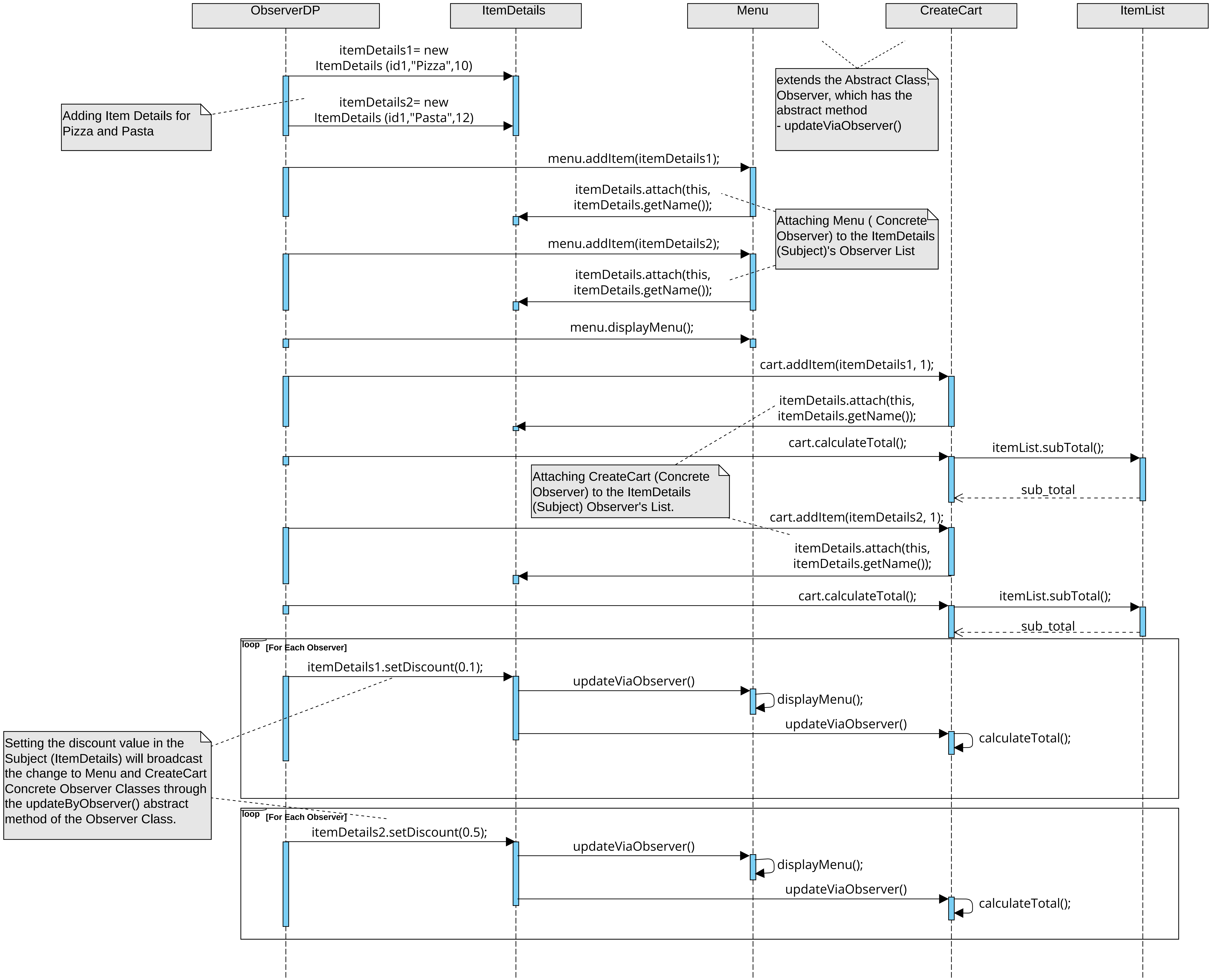
The Concrete Observer Class, Menu and Create Cart, which extends the updateViaObserver()

In the main() method, ItemDetails are created and added to the Menu and displayed. Then, the items are added to the cart. Later, when the restaurant updates the Item price with a discount, it will be updated in the Menu and Cart using the Observer class automatically.

When the item discount value is updated, the subject will broadcast the updated value to the observers, Menu and CreateCart, using the updateByObserver() method of the observer class.

The Observer Class. Abstract Class with updateByObserver() Abstract Method

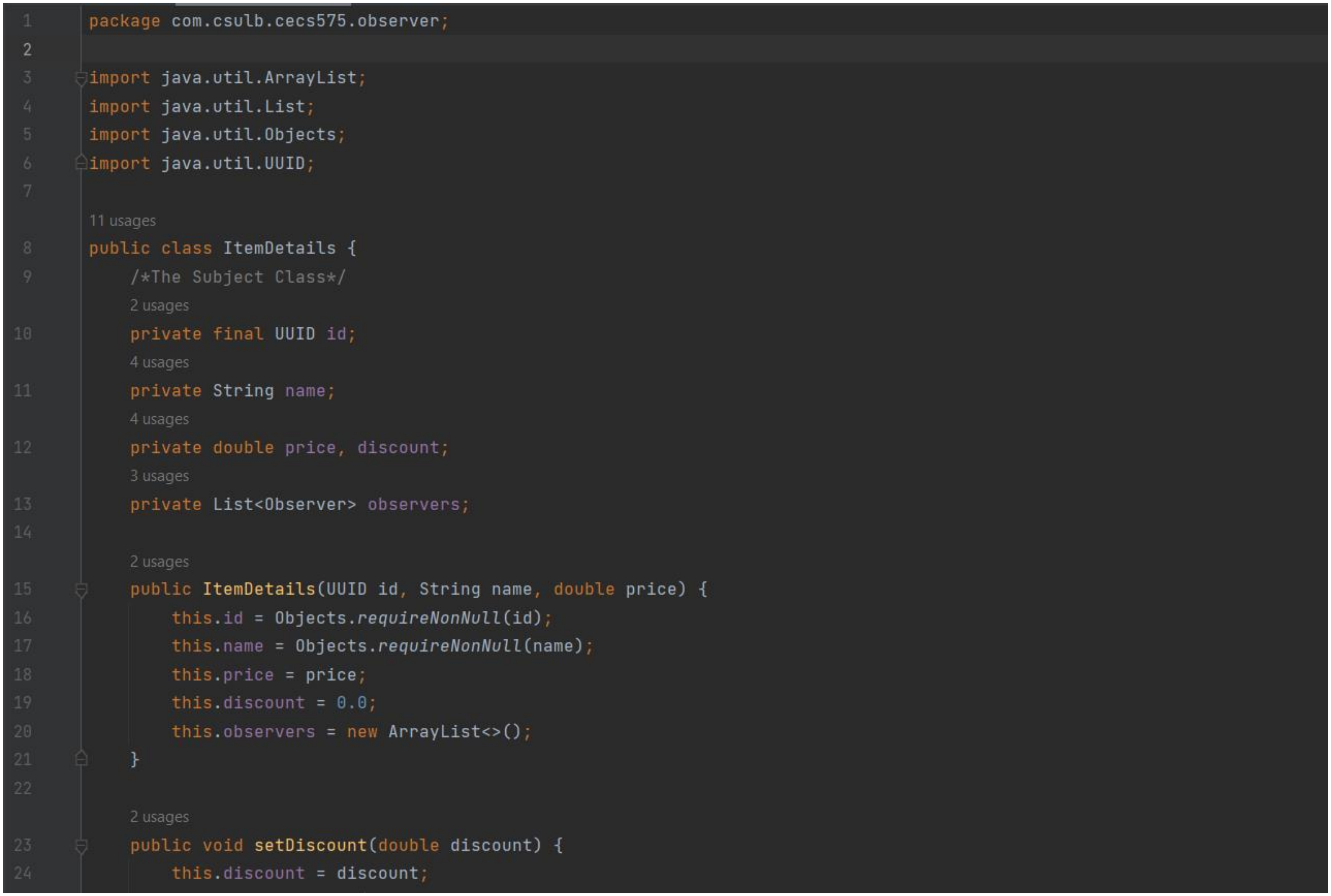
The Subject Class


ObserverDP
ItemDetails
Menu
CreateCart
ItemList
extends the Abstract Class, Observer, which has the abstract method - updateViaObserver()
itemDetails1= new ItemDetails (id1,"Pizza",10)
Adding Item Details for Pizza and Pasta
itemDetails2= new ItemDetails (id1,"Pasta",12)
menu.addItem(itemDetails1);
itemDetails.attach(this, itemDetails.getName());
Attaching Menu (Concrete Observer) to the ItemDetails (Subject)'s Observer List
menu.addItem(itemDetails2);
itemDetails.attach(this, itemDetails.getName());
menu.displayMenu();
cart.addItem(itemDetails1, 1);
itemDetails.attach(this, itemDetails.getName());
cart.calculateTotal();
itemList.subTotal();
Attaching CreateCart (Concrete Observer) to the ItemDetails (Subject) Observer's List.
sub_total
cart.addItem(itemDetails2, 1);
itemDetails.attach(this, itemDetails.getName());
cart.calculateTotal();
itemList.subTotal();
sub_total
loop [For Each Observer]
itemDetails1.setDiscount(0.1);
updateViaObserver()
displayMenu();
updateViaObserver()
calculateTotal();
Setting the discount value in the Subject (ItemDetails) will broadcast the change to Menu and CreateCart Concrete Observer Classes through the updateByObserver() abstract method of the Observer Class.
loop [For Each Observer]
itemDetails2.setDiscount(0.5);
updateViaObserver()
displayMenu();
updateViaObserver()
calculateTotal();

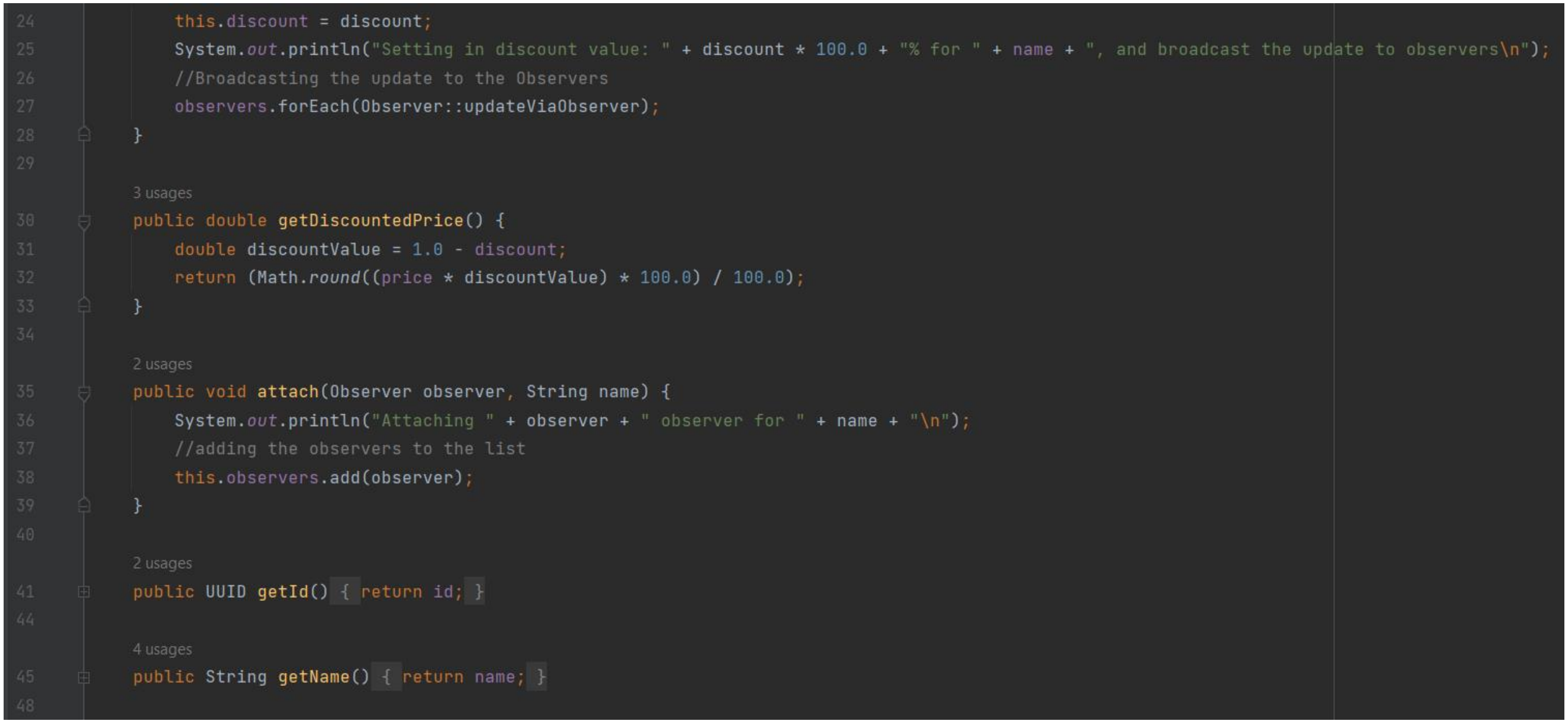
Question 2: Observer Pattern - Sequence Diagram

Observer Design Pattern: Applying discount to the Items will broadcast the details to Menu and Cart – Code Snippets:

- The Subject Class – ItemDetails with the list of Observers, attach and update methods:

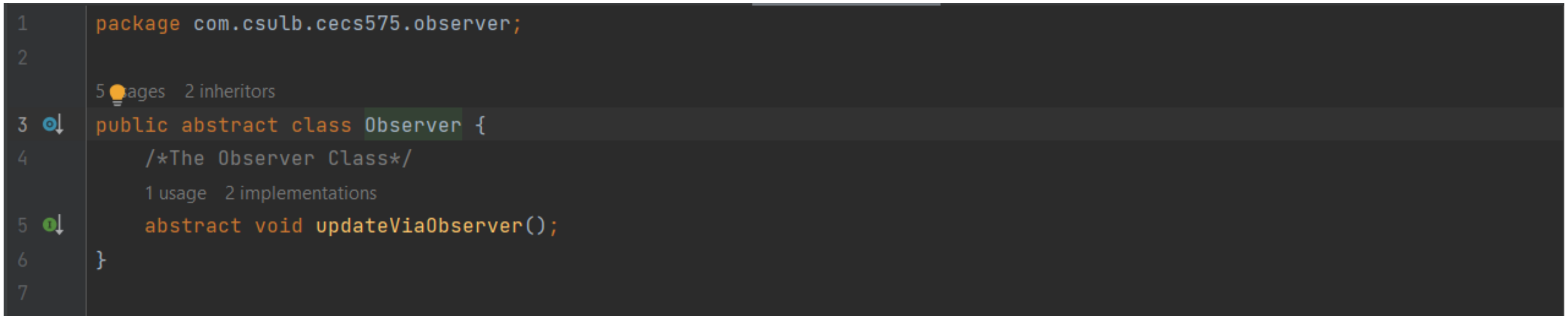


```
package com.csulb.cecs575.observer;

import java.util.ArrayList;
import java.util.List;
import java.util.Objects;
import java.util.UUID;

11 usages
public class ItemDetails {
    /*The Subject Class*/
    2 usages
    private final UUID id;
    4 usages
    private String name;
    4 usages
    private double price, discount;
    3 usages
    private List<Observer> observers;

    2 usages
    public ItemDetails(UUID id, String name, double price) {
        this.id = Objects.requireNonNull(id);
        this.name = Objects.requireNonNull(name);
        this.price = price;
        this.discount = 0.0;
        this.observers = new ArrayList<>();
    }

    2 usages
    public void setDiscount(double discount) {
        this.discount = discount;
```



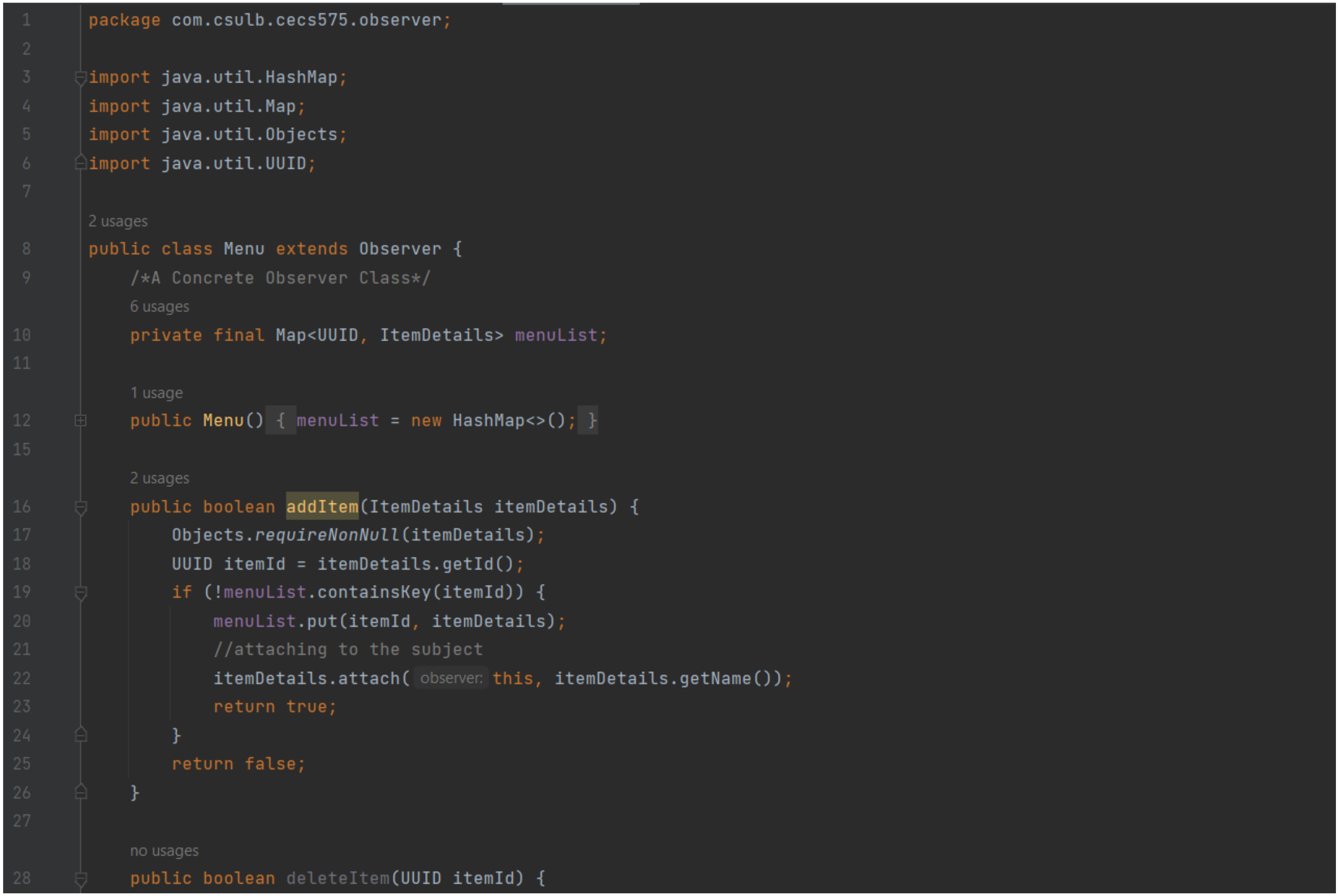
```
        this.discount = discount;
        System.out.println("Setting in discount value: " + discount * 100.0 + "% for " + name + ", and broadcast the update to observers\n");
        //Broadcasting the update to the Observers
        observers.forEach(Observer::updateViaObserver);
    }

    3 usages
    public double getDiscountedPrice() {
        double discountValue = 1.0 - discount;
        return (Math.round((price * discountValue) * 100.0) / 100.0);
    }

    2 usages
    public void attach(Observer observer, String name) {
        System.out.println("Attaching " + observer + " observer for " + name + "\n");
        //adding the observers to the list
        this.observers.add(observer);
    }

    2 usages
    public UUID getId() { return id; }

    4 usages
    public String getName() { return name; }
```

- The Observer abstract class (parent class) with updateViaObserver():



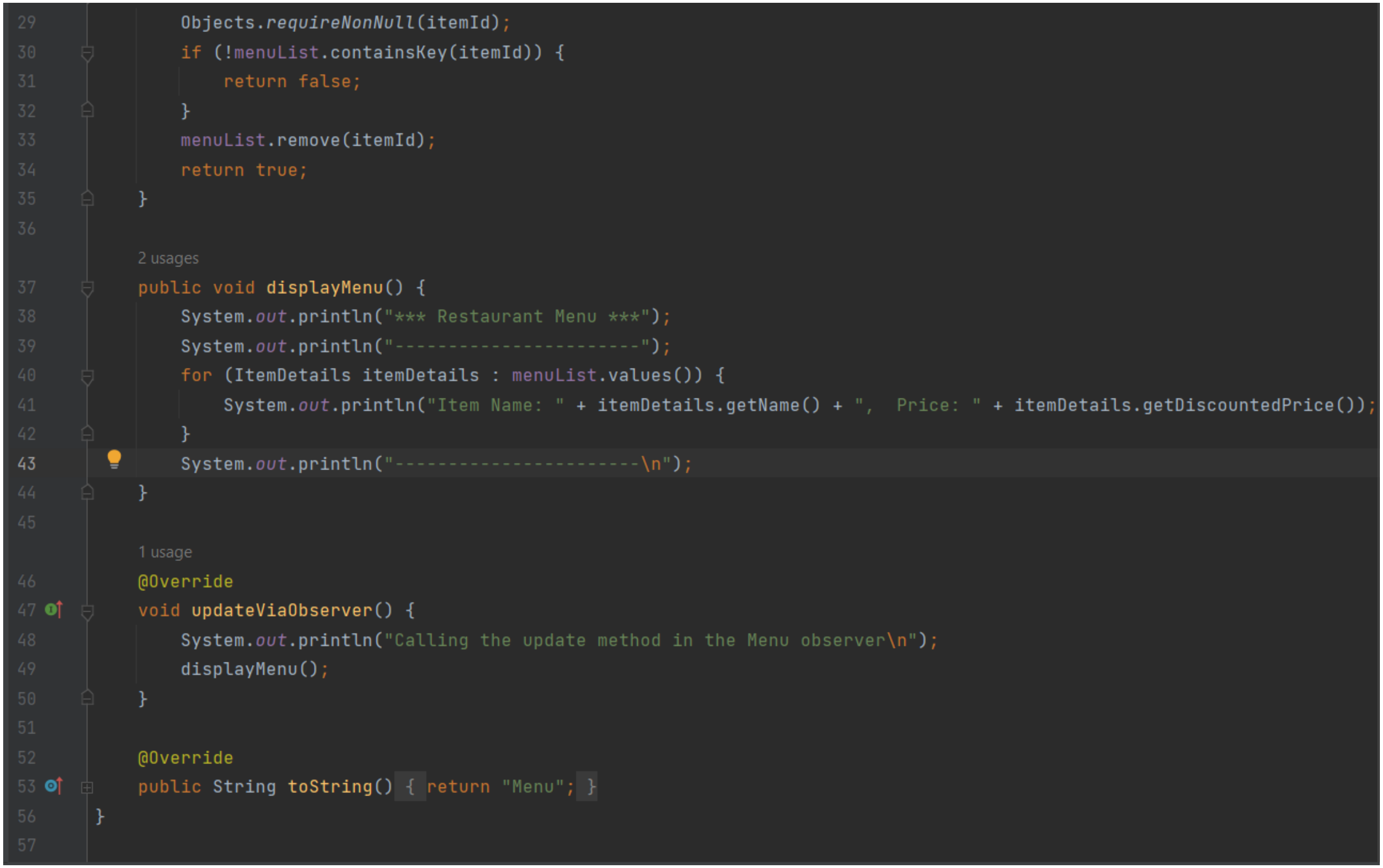
```
package com.csulb.cecs575.observer;

5 ages   2 inheritors
public abstract class Observer {
    /*The Observer Class*/
    1 usage   2 implementations
    abstract void updateViaObserver();
}
```

- Menu class in which the add item method will attach it to the Observer List in the ItemDetails Class (Subject):



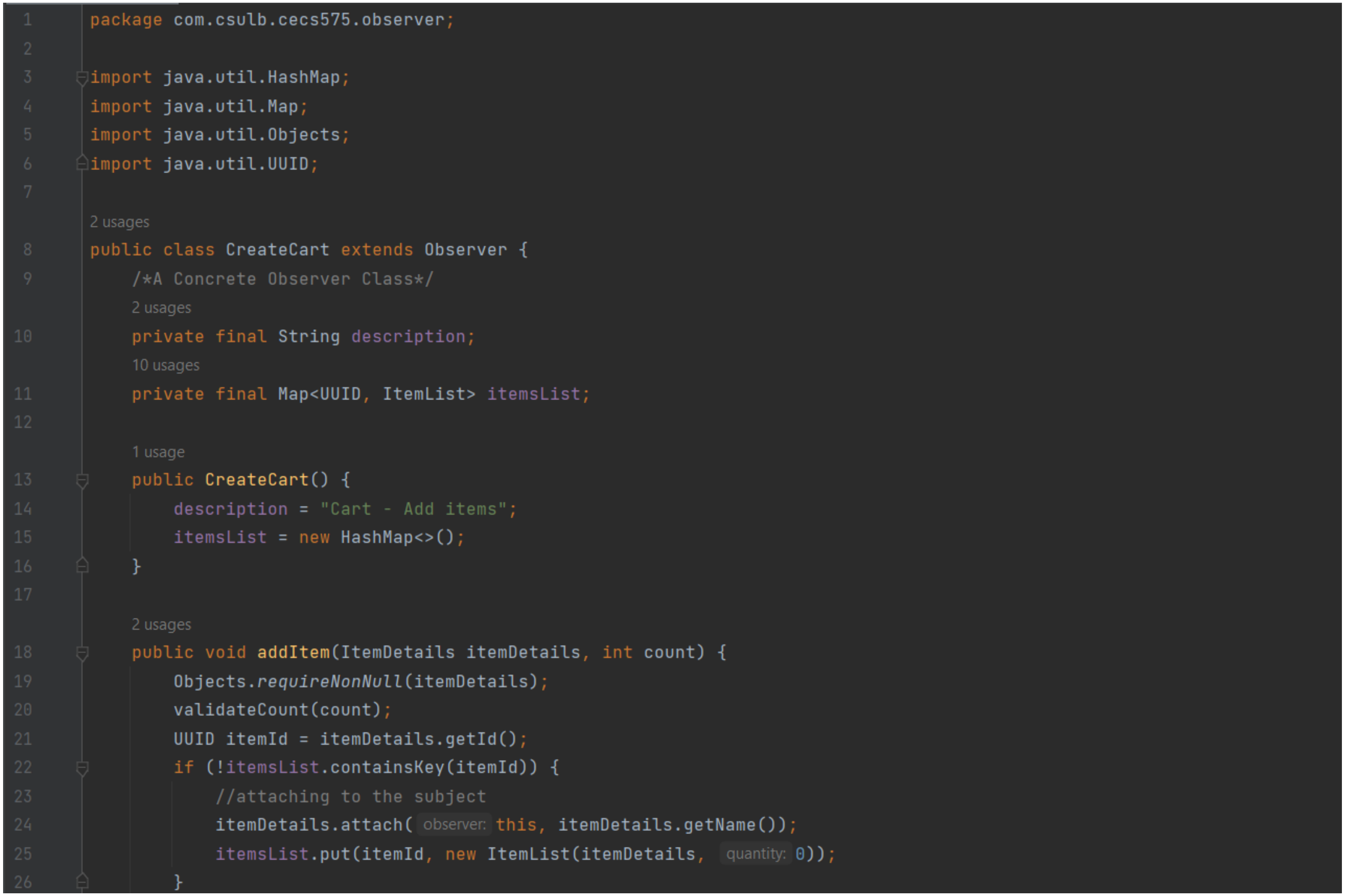
```java
package com.csulb.cecs575.observer;

import java.util.HashMap;
import java.util.Map;
import java.util.Objects;
import java.util.UUID;

2 usages
public class Menu extends Observer {
    /*A Concrete Observer Class*/
    6 usages
    private final Map<UUID, ItemDetails> menuList;

    1 usage
    public Menu() { menuList = new HashMap<>(); }

    2 usages
    public boolean addItem(ItemDetails itemDetails) {
        Objects.requireNonNull(itemDetails);
        UUID itemId = itemDetails.getId();
        if (!menuList.containsKey(itemId)) {
            menuList.put(itemId, itemDetails);
            //attaching to the subject
            itemDetails.attach( observer: this, itemDetails.getName());
            return true;
        }
        return false;
    }

    no usages
    public boolean deleteItem(UUID itemId) {
```

- displayMenu() and updateViaObserver() methods which will be called by the Subject on update:



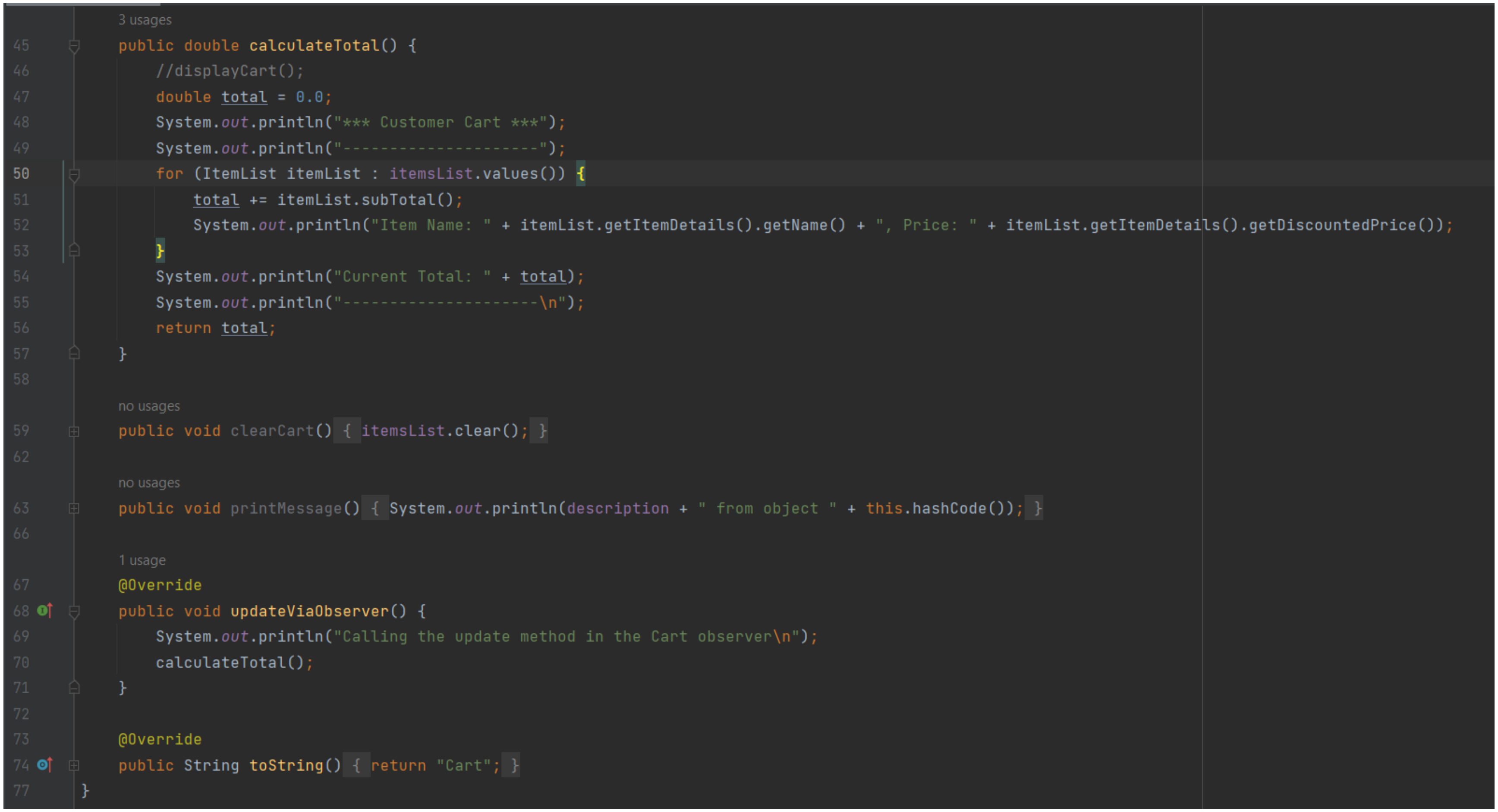
```
        Objects.requireNonNull(itemId);
        if (!menuList.containsKey(itemId)) {
            return false;
        }
        menuList.remove(itemId);
        return true;
    }

    2 usages
    public void displayMenu() {
        System.out.println("*** Restaurant Menu ***");
        System.out.println("-----------------------");
        for (ItemDetails itemDetails : menuList.values()) {
            System.out.println("Item Name: " + itemDetails.getName() + ",  Price: " + itemDetails.getDiscountedPrice());
        }
        System.out.println("-----------------------\n");
    }

    1 usage
    @Override
    void updateViaObserver() {
        System.out.println("Calling the update method in the Menu observer\n");
        displayMenu();
    }

    @Override
    public String toString() { return "Menu"; }
}
```

- CreateCart class in which the add item method will attach it to the Observer List in the ItemDetails Class (Subject):



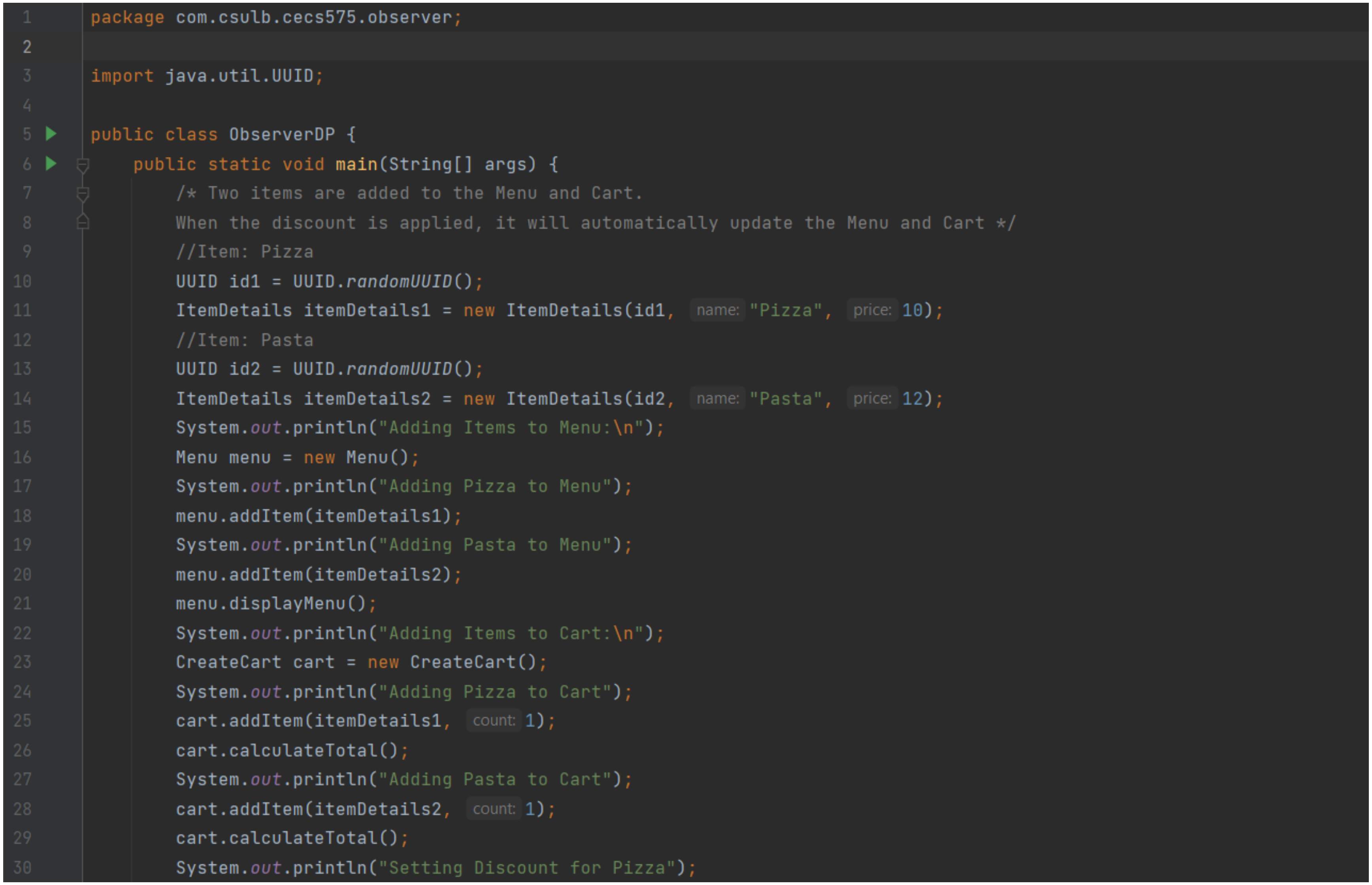
```
package com.csulb.cecs575.observer;

import java.util.HashMap;
import java.util.Map;
import java.util.Objects;
import java.util.UUID;

2 usages
public class CreateCart extends Observer {
    /*A Concrete Observer Class*/
    2 usages
    private final String description;
    10 usages
    private final Map<UUID, ItemList> itemsList;

    1 usage
    public CreateCart() {
        description = "Cart - Add items";
        itemsList = new HashMap<>();
    }

    2 usages
    public void addItem(ItemDetails itemDetails, int count) {
        Objects.requireNonNull(itemDetails);
        validateCount(count);
        UUID itemId = itemDetails.getId();
        if (!itemsList.containsKey(itemId)) {
            //attaching to the subject
            itemDetails.attach( observer: this, itemDetails.getName());
            itemsList.put(itemId, new ItemList(itemDetails,  quantity: 0));
        }
```

- calculateTotal() and updateViaObserver() method which will be called by the Subject on update:

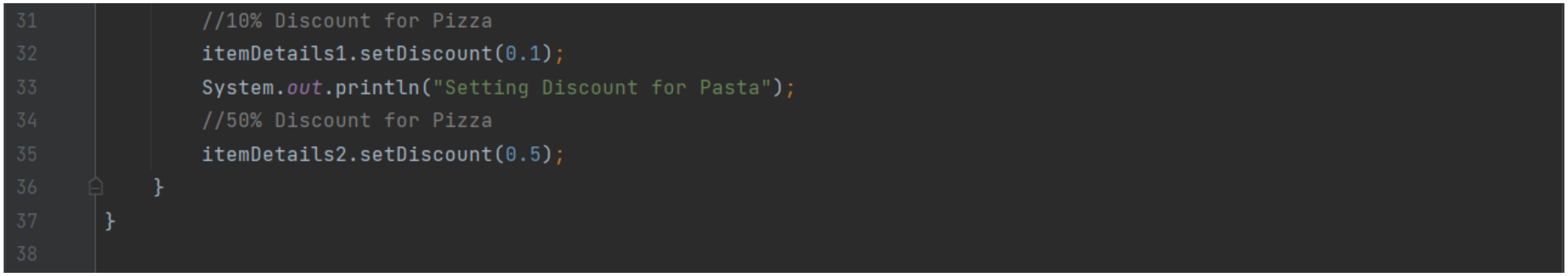


```
    3 usages
    public double calculateTotal() {
        //displayCart();
        double total = 0.0;
        System.out.println("*** Customer Cart ***");
        System.out.println("---------------------");
        for (ItemList itemList : itemsList.values()) {
            total += itemList.subTotal();
            System.out.println("Item Name: " + itemList.getItemDetails().getName() + ", Price: " + itemList.getItemDetails().getDiscountedPrice());
        }
        System.out.println("Current Total: " + total);
        System.out.println("---------------------\n");
        return total;
    }

    no usages
    public void clearCart() { itemsList.clear(); }

    no usages
    public void printMessage() { System.out.println(description + " from object " + this.hashCode()); }

    1 usage
    @Override
    public void updateViaObserver() {
        System.out.println("Calling the update method in the Cart observer\n");
        calculateTotal();
    }

    @Override
    public String toString() { return "Cart"; }
}
```

- ObserverDP Class with the main() method adds Pizza and Pasta to the Menu and then to the Cart, which will add the Menu and Cart to the observer(s) list of those items. Setting up discount values for those items will be automatically updated in the Menu and Cart:

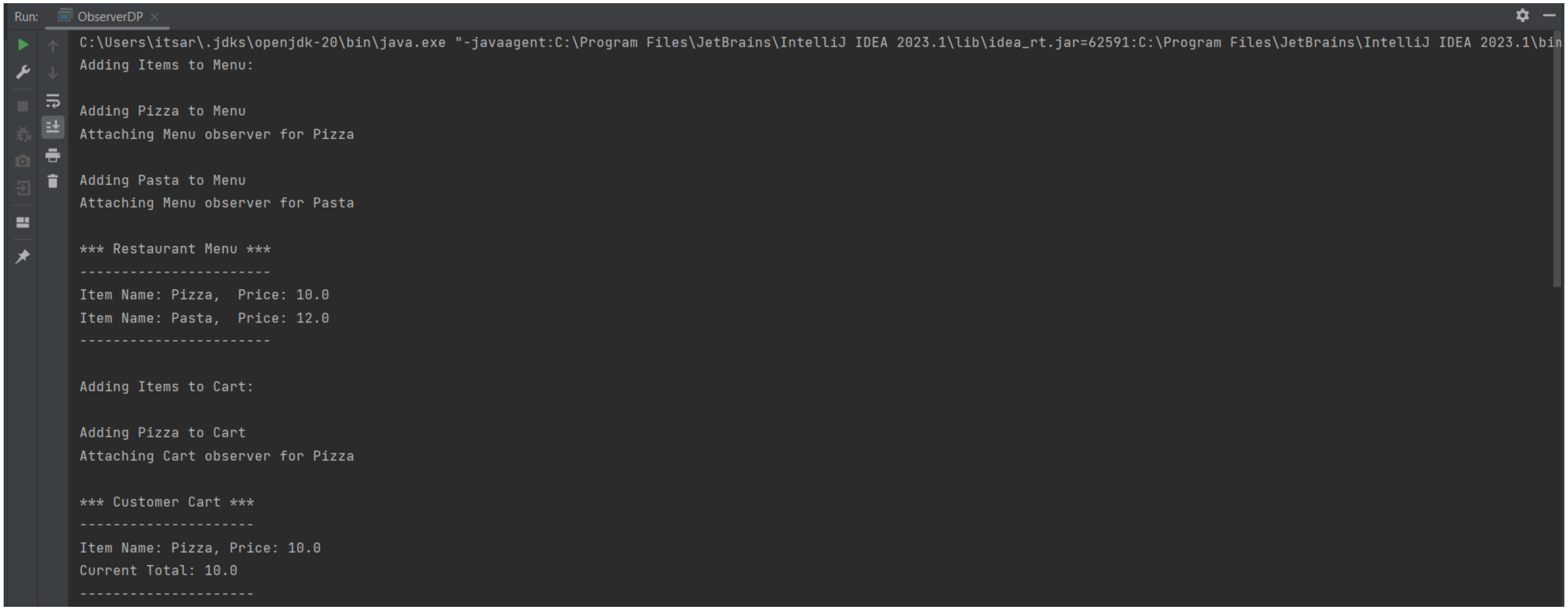


```java
package com.csulb.cecs575.observer;

import java.util.UUID;

public class ObserverDP {
    public static void main(String[] args) {
        /* Two items are added to the Menu and Cart.
        When the discount is applied, it will automatically update the Menu and Cart */
        //Item: Pizza
        UUID id1 = UUID.randomUUID();
        ItemDetails itemDetails1 = new ItemDetails(id1,  name: "Pizza",  price: 10);
        //Item: Pasta
        UUID id2 = UUID.randomUUID();
        ItemDetails itemDetails2 = new ItemDetails(id2,  name: "Pasta",  price: 12);
        System.out.println("Adding Items to Menu:\n");
        Menu menu = new Menu();
        System.out.println("Adding Pizza to Menu");
        menu.addItem(itemDetails1);
        System.out.println("Adding Pasta to Menu");
        menu.addItem(itemDetails2);
        menu.displayMenu();
        System.out.println("Adding Items to Cart:\n");
        CreateCart cart = new CreateCart();
        System.out.println("Adding Pizza to Cart");
        cart.addItem(itemDetails1,  count: 1);
        cart.calculateTotal();
        System.out.println("Adding Pasta to Cart");
        cart.addItem(itemDetails2,  count: 1);
        cart.calculateTotal();
        System.out.println("Setting Discount for Pizza");
```



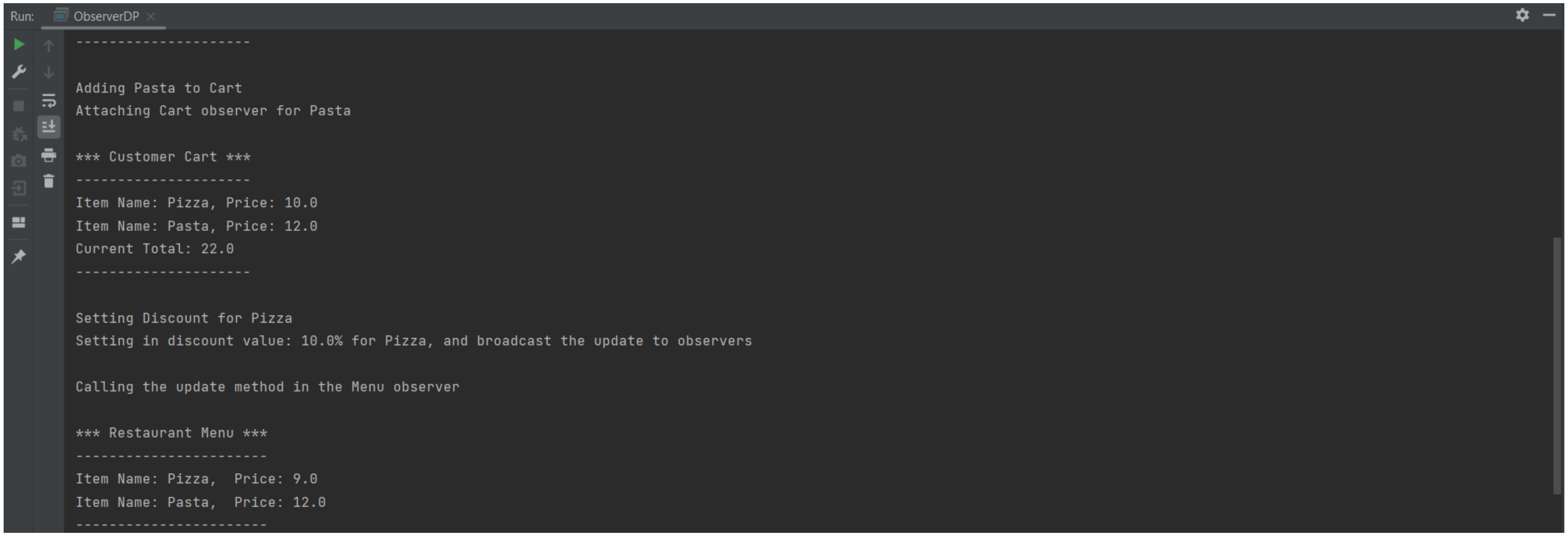
```java
            //10% Discount for Pizza
            itemDetails1.setDiscount(0.1);
            System.out.println("Setting Discount for Pasta");
            //50% Discount for Pizza
            itemDetails2.setDiscount(0.5);
        }
    }

```

Output:



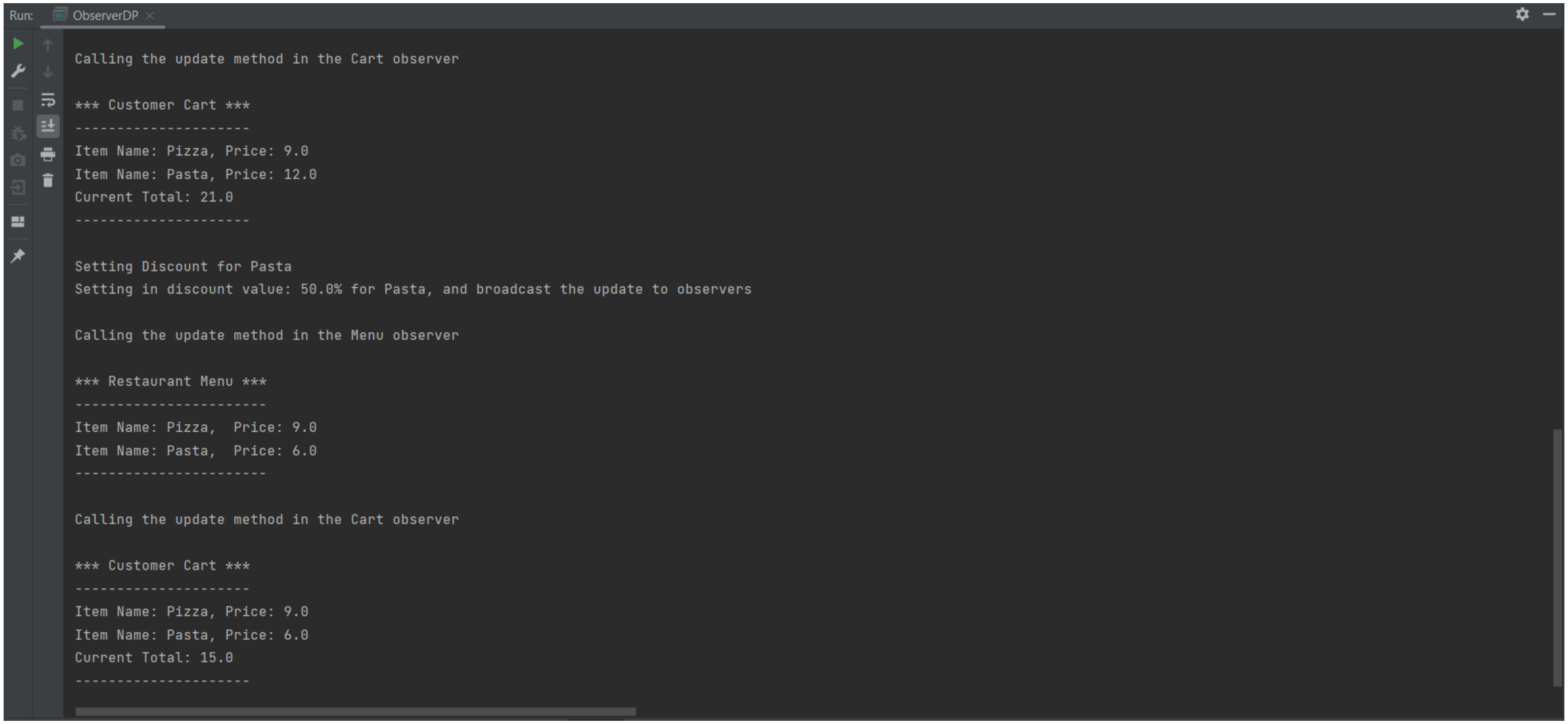
```
C:\Users\itsar\.jdks\openjdk-20\bin\java.exe "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 2023.1\lib\idea_rt.jar=62591:C:\Program Files\JetBrains\IntelliJ IDEA 2023.1\bin
Adding Items to Menu:

Adding Pizza to Menu
Attaching Menu observer for Pizza

Adding Pasta to Menu
Attaching Menu observer for Pasta

*** Restaurant Menu ***
-----------------------
Item Name: Pizza,  Price: 10.0
Item Name: Pasta,  Price: 12.0
-----------------------

Adding Items to Cart:

Adding Pizza to Cart
Attaching Cart observer for Pizza

*** Customer Cart ***
---------------------
Item Name: Pizza, Price: 10.0
Current Total: 10.0
---------------------
```

- Two items, namely Pizza and Pasta, were added to the Menu, which will attach Menu to the Observers list of both the items in the Subject (ItemDetails), and the Menu of the Restaurant is displayed. Also, the item Pizza is added to the cart, which will attach Cart to the Observers list of Pizza items and the current total of the cart is calculated and displayed.



```
---------------------

Adding Pasta to Cart
Attaching Cart observer for Pasta

*** Customer Cart ***
---------------------
Item Name: Pizza, Price: 10.0
Item Name: Pasta, Price: 12.0
Current Total: 22.0
---------------------

Setting Discount for Pizza
Setting in discount value: 10.0% for Pizza, and broadcast the update to observers

Calling the update method in the Menu observer

*** Restaurant Menu ***
-----------------------
Item Name: Pizza,  Price: 9.0
Item Name: Pasta,  Price: 12.0
-----------------------
```

- The item Pasta is then added to the cart, which will attach Cart to the Observers list of Pasta item and the current total of the cart is recalculated and displayed. Now, a discount of 10% is set for Pizza and a broadcast of this update is sent to Observers of ItemDetails, which includes the Menu and Cart. The discounted price is updated in the Menu is displayed.



```
Calling the update method in the Cart observer

*** Customer Cart ***
--------------------
Item Name: Pizza, Price: 9.0
Item Name: Pasta, Price: 12.0
Current Total: 21.0
--------------------

Setting Discount for Pasta
Setting in discount value: 50.0% for Pasta, and broadcast the update to observers

Calling the update method in the Menu observer

*** Restaurant Menu ***
----------------------
Item Name: Pizza,  Price: 9.0
Item Name: Pasta,  Price: 6.0
----------------------

Calling the update method in the Cart observer

*** Customer Cart ***
--------------------
Item Name: Pizza, Price: 9.0
Item Name: Pasta, Price: 6.0
Current Total: 15.0
--------------------
```

- The discounted price for Pizza is updated in the Cart as well, and the current total is recalculated and displayed. Then, a discount of 50% is set for Pasta and a broadcast of this update is sent to Observers of the ItemDetails, which includes the Menu and Cart. The discounted price is updated in the Menu and Cart, and updated values are displayed.