Online Clothing Store

Fashion Inc. (FI).

Name: Design Patterns

Duaa(Captain)-Observer

Shahnia-Factory

Jonathan-Decorator

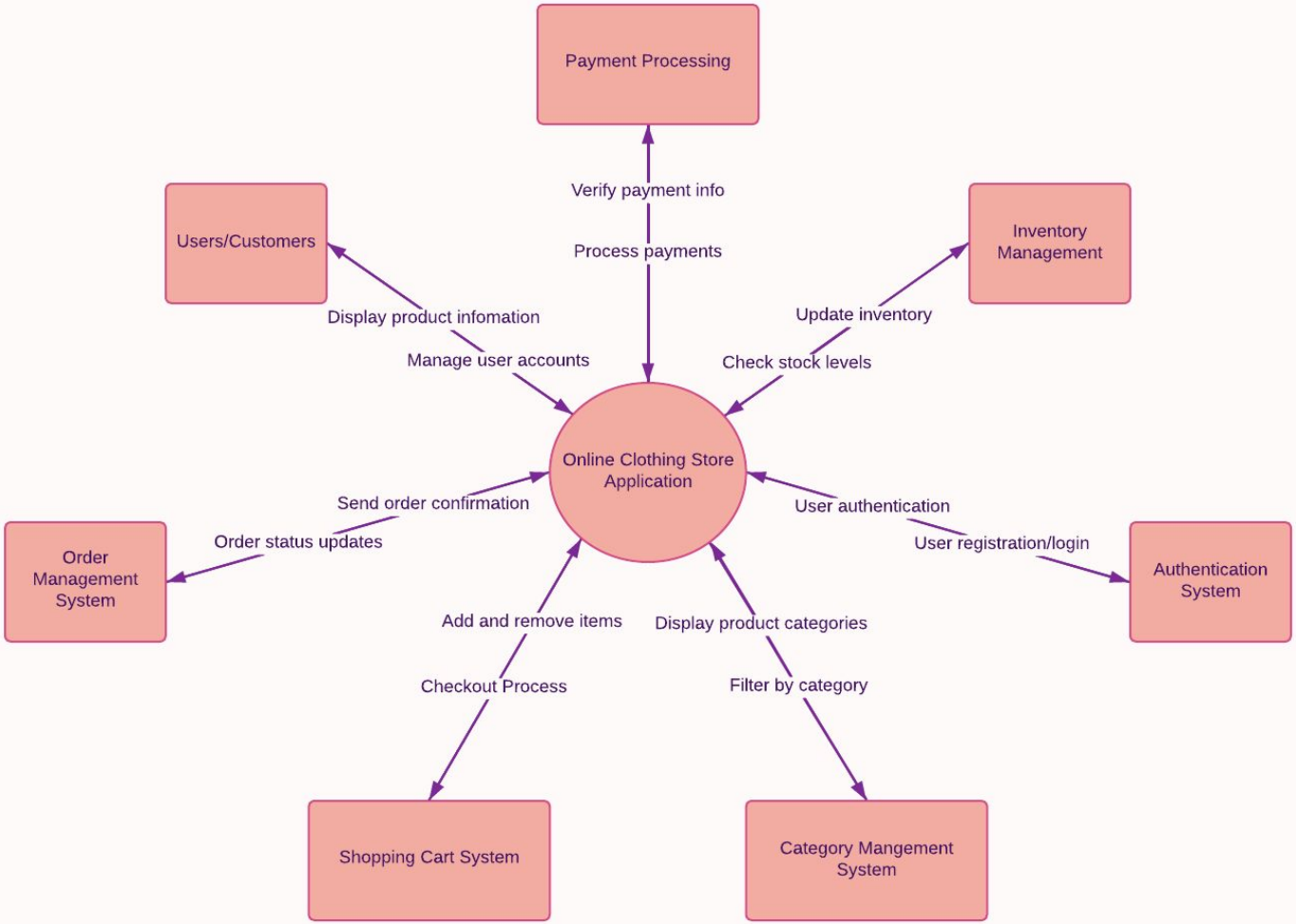
Elan-Singleton

Winston-State

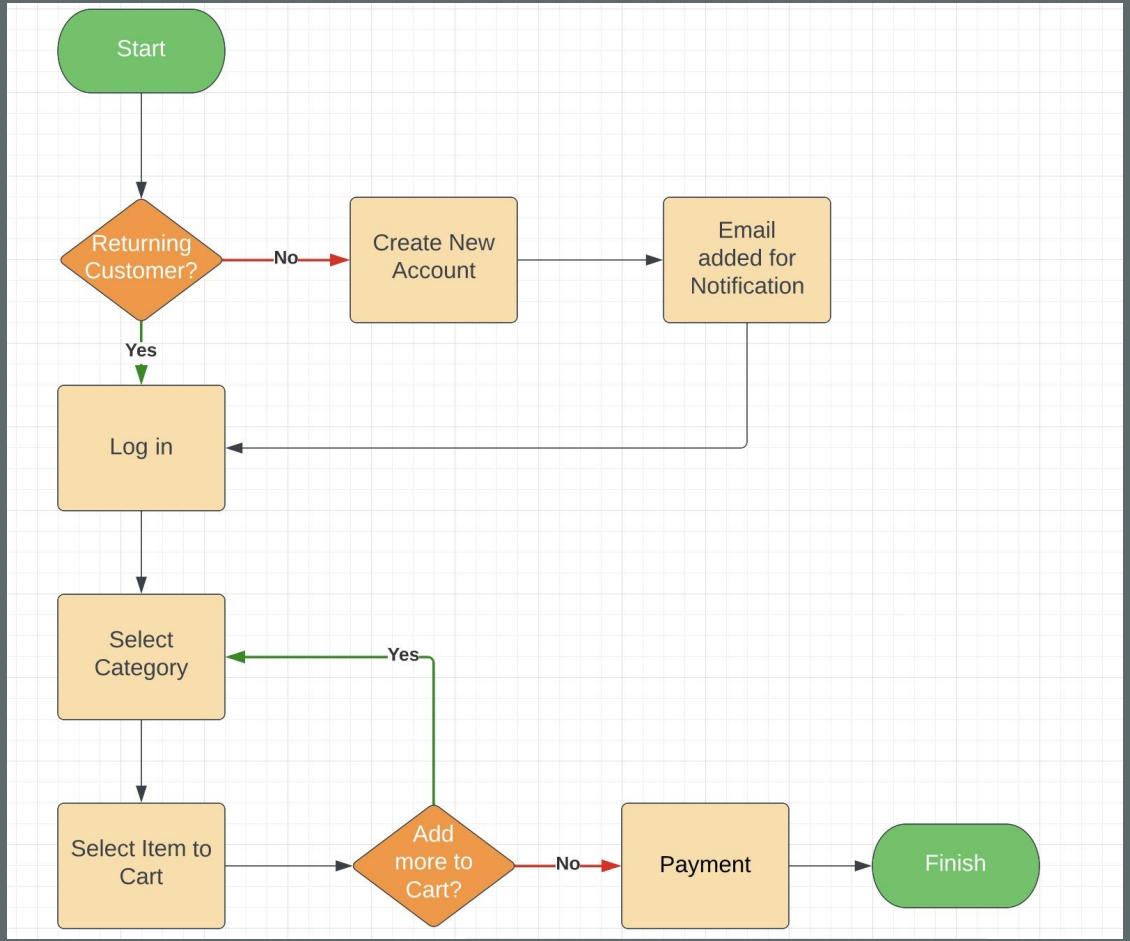
Introduction

- Online shopping is perfect for customers looking for a more convenient way of navigating through and finding their desired products.
- Design Patterns is a traditional online clothing store allowing customers to scroll through various categories and items of clothing.
- Users are able to customize their clothing with preferred features such as color, size and material.
- They can add their items to a shopping cart and pay after.
- Customers will also have the option to be notified of item availability as well as any store discount.

Context Diagram

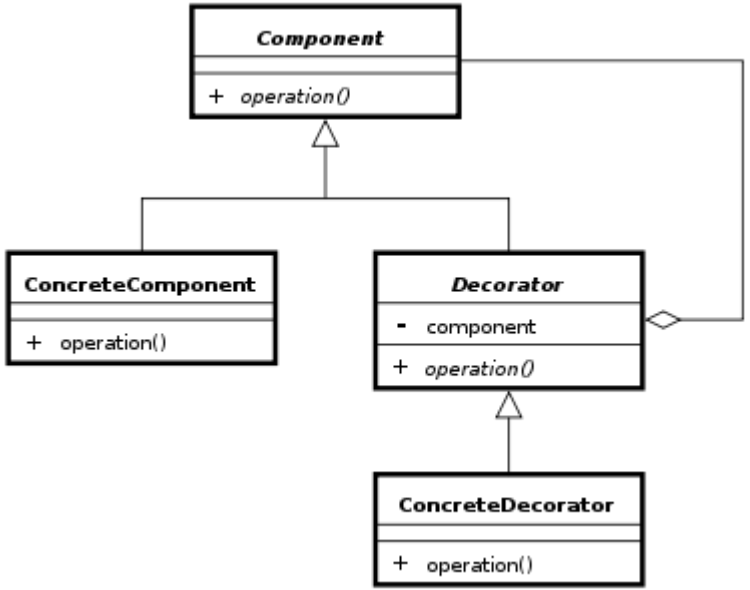


Process Flow Diagram

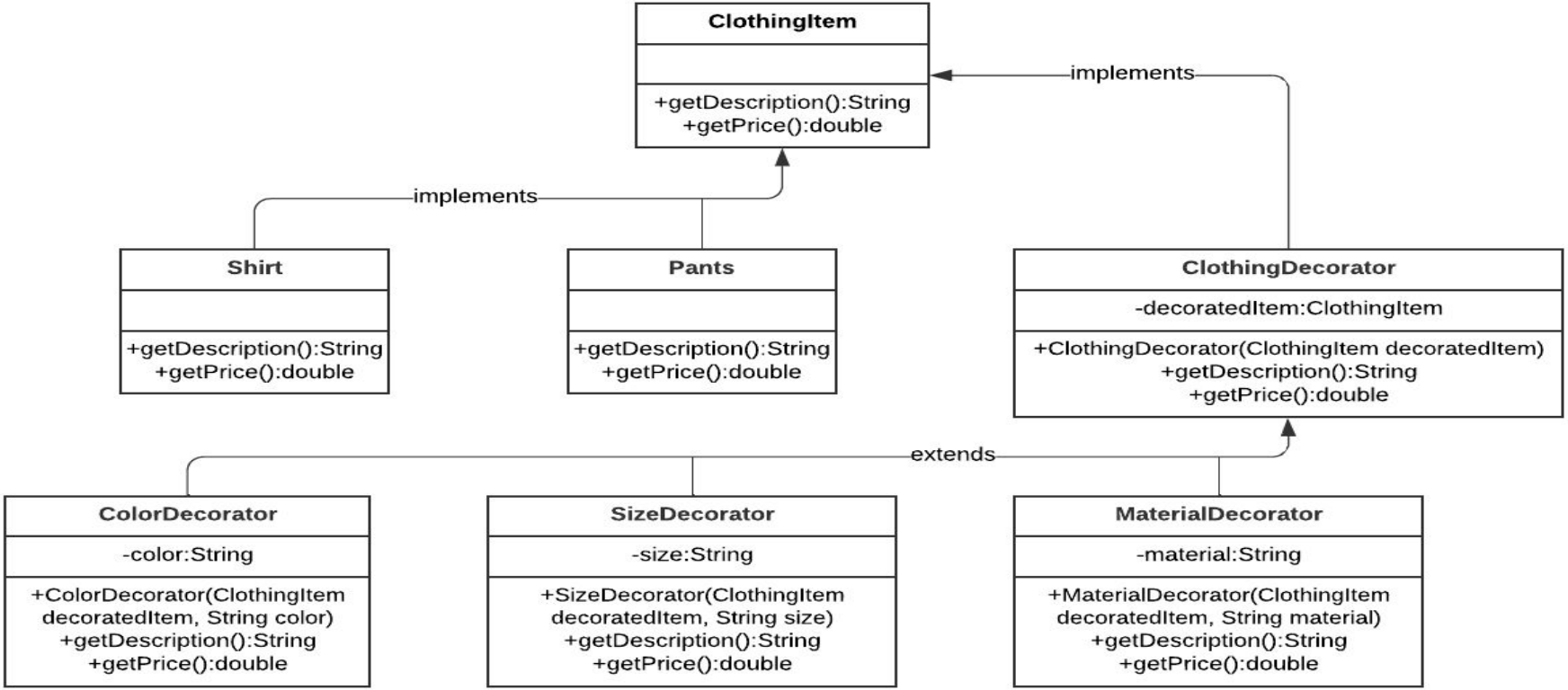


Jonathan - Decorator Pattern

- Adds behavior to an object dynamically
- Consists of a component interface, concrete component classes, an abstract decorator class, and concrete decorator classes
- Follows the open-closed principle
- In our case, the decorator pattern will be used to add color, size, and material to the clothing items.



Decorator Pattern - UML Diagram



Decorator Pattern - Java Code

The ClothingItem interface defines the methods all clothing items must have

```java
public interface ClothingItem {
    // gets the description and price of a ClothingItem
    public String getDescription();
    public double getPrice();
}
```

- Two concrete classes “Shirt” and “Pants”
- Implements the ClothingItem interface

```java
public class Shirt implements ClothingItem {
    // Implements the getPrice and getDescription methods
    @Override
    public double getPrice() {
        return 9.99;
    }
    @Override
    public String getDescription() {
        return "Regular Shirt";
    }
}
```

```java
public class Pants implements ClothingItem {
    // Implements the getPrice and getDescription methods
    @Override
    public double getPrice() {
        return 19.99;
    }
    @Override
    public String getDescription() {
        return "A Pair of Pants";
    }
}
```

- The abstract ClothingDecorator class
- Implements the ClothingItem interface
- Serves as a base class for our color, size, and material decorators to extend

```java
public abstract class ClothingDecorator implements ClothingItem {
    // The ClothingItem which is being customized
    public ClothingItem decoratedItem;

    // Constructor
    public ClothingDecorator(ClothingItem decoratedItem) {
        this.decoratedItem = decoratedItem;
    }

    // gets the price of the customized ClothingItem
    @Override
    public double getPrice() {
        return decoratedItem.getPrice();
    }

    // gets the description of the customized ClothingItem
    @Override
    public String getDescription() {
        return decoratedItem.getDescription();
    }
}
```

- The concrete ColorDecorator class
- Extends the ClothingDecorator class
- Decorates a ClothingItem with a valid color
- Applies an additional cost based on the color

```java
import java.util.Arrays;
import java.util.List;
public class ColorDecorator extends ClothingDecorator {
    // The final color of the customized ClothingItem
    private String color;
    public ColorDecorator(ClothingItem decoratedItem, String color) {
        // Calling the parent constructor
        super(decoratedItem);
        // Throw an exception if color is not in the list of validColors
        List<String> validColors = Arrays.asList("Red", "Blue", "Purple", "Black", "Green");
        if (!validColors.contains(color)) {
            throw new IllegalArgumentException("Illegal color: " + color);
        }
        // If valid, assign this color to the ColorDecorator
        this.color = color;
    }
    // Adds the color to the item description
    @Override
    public String getDescription() {
        return decoratedItem.getDescription() + ", Color: " + color;
    }
    // Return the price of the ClothingItem based on its color
    @Override
    public double getPrice() {
        if (color.equals("Red")) {
            return decoratedItem.getPrice() + 2.00;
        } else if (color.equals("Blue")) {
            return decoratedItem.getPrice() + 3.00;
        } else {
            return decoratedItem.getPrice();
        }
    }
}
```

- The concrete SizeDecorator class
- Extends the ClothingDecorator class
- Decorates a ClothingItem with a valid size
- Applies an additional cost based on the size

```java
import java.util.Arrays;
import java.util.List;
public class SizeDecorator extends ClothingDecorator {
    // The final size of the customized ClothingItem
    private String size;
    public SizeDecorator(ClothingItem decoratedItem, String size) {
        // Calling the parent constructor
        super(decoratedItem);
        // Throw an exception if size is not in the list of validSizes
        List<String> validSizes = Arrays.asList("Small", "Medium", "Large");
        if (!validSizes.contains(size)) {
            throw new IllegalArgumentException("Illegal size: " + size);
        }
        // If valid, assign this size to the SizeDecorator
        this.size = size;
    }
    // Adds the size to the item description
    @Override
    public String getDescription() {
        return decoratedItem.getDescription() + ", Size: " + size;
    }
    // Return the price of the ClothingItem based on its size
    @Override
    public double getPrice() {
        if (size.equals("Medium")) {
            return decoratedItem.getPrice() + 2.00;
        } else if (size.equals("Large")) {
            return decoratedItem.getPrice() + 4.50;
        } else {
            return decoratedItem.getPrice();
        }
    }
```

- The concrete MaterialDecorator class
- Extends the ClothingDecorator class
- Decorates a ClothingItem with a valid material
- Applies an additional cost based on the material

```java
import java.util.Arrays;
import java.util.List;
public class MaterialDecorator extends ClothingDecorator {
    // The final material of the customized ClothingItem
    private String material;
    public MaterialDecorator(ClothingItem decoratedItem, String material) {
        // Calling the parent constructor
        super(decoratedItem);
        // Throw an exception if material is not in the list of validMaterials
        List<String> validMaterials = Arrays.asList("Cotton", "Leather", "Silk", "Wool");
        if (!validMaterials.contains(material)) {
            throw new IllegalArgumentException("Illegal material: " + material);
        }
        // If valid, assign this material to the MaterialDecorator
        this.material = material;
    }
    // Adds the material to the item description
    @Override
    public String getDescription() {
        return decoratedItem.getDescription() + ", Material: " + material;
    }
    // Return the price of the ClothingItem based on its material
    @Override
    public double getPrice() {
        if (material.equals("Silk")) {
            return decoratedItem.getPrice() + 3.00;
        } else if (material.equals("Wool")) {
            return decoratedItem.getPrice() + 5.00;
        } else {
            return decoratedItem.getPrice();
        }
    }
}
```

- The main function
- Decorates one Shirt and one Pants object with color, size and material

```
public class Main {
    public static void main(String[] args) {
        // Decorates a Shirt object with color, size, and material
        ClothingItem shirt = new Shirt();
        shirt = new ColorDecorator(shirt, color: "Red");
        shirt = new SizeDecorator(shirt, size: "Medium");
        shirt = new MaterialDecorator(shirt, material: "Silk");

        // Print out the description and price of the shirt
        System.out.println("Item Description: " + shirt.getDescription());
        System.out.println("Price: $" + (String.format("%.2f", shirt.getPrice())));

        // Decorates a Pants object with color, size, and material
        ClothingItem pants = new Pants();
        pants = new ColorDecorator(pants, color: "Blue");
        pants = new SizeDecorator(pants, size: "Large");
        pants = new MaterialDecorator(pants, material: "Wool");

        // Print out the description and price of the pants
        System.out.println("Item Description: " + pants.getDescription());
        System.out.println("Price: $" + (String.format("%.2f", pants.getPrice())));
    }
}
```

```
Item Description: Regular Shirt, Color: Red, Size: Medium, Material: Silk
Price: $16.99
Item Description: A Pair of Pants, Color: Blue, Size: Large, Material: Wool
Price: $32.49
```

Unit tests

- Tests the color, size, and material decorators
- Checks if the updated price of the decorated clothing item is correct

```java
import static org.junit.jupiter.api.Assertions.*;

public class ClothingDecoratorTest {
    @org.junit.jupiter.api.Test
    public void ColorDecoratorTest() {
        ClothingItem shirt = new Shirt();
        shirt = new ColorDecorator(shirt, color: "Blue");
        assertEquals( expected: 12.99, shirt.getPrice(), delta: 0.001);
    }

    @org.junit.jupiter.api.Test
    public void SizeDecoratorTest() {
        ClothingItem pants = new Pants();
        pants = new SizeDecorator(pants, size: "Medium");
        assertEquals( expected: 21.99, pants.getPrice(), delta: 0.001);
    }

    @org.junit.jupiter.api.Test
    public void MaterialDecoratorTest() {
        ClothingItem pants = new Pants();
        pants = new MaterialDecorator(pants, material: "Silk");
        assertEquals( expected: 22.99, pants.getPrice(), delta: 0.001);
    }
}
```

Component Test

The customer selects a shirt and customizes it with the color “Blue”, which adds an additional $3.00 to the base price. He or she then chooses the size “Medium”, which costs an additional $2.00. Finally, the customer decides he/she wants a “Wool” shirt, which costs an additional $5.00. The base price of a shirt is $9.99. Therefore, with these customizations, the total cost for this shirt will equal $19.99.

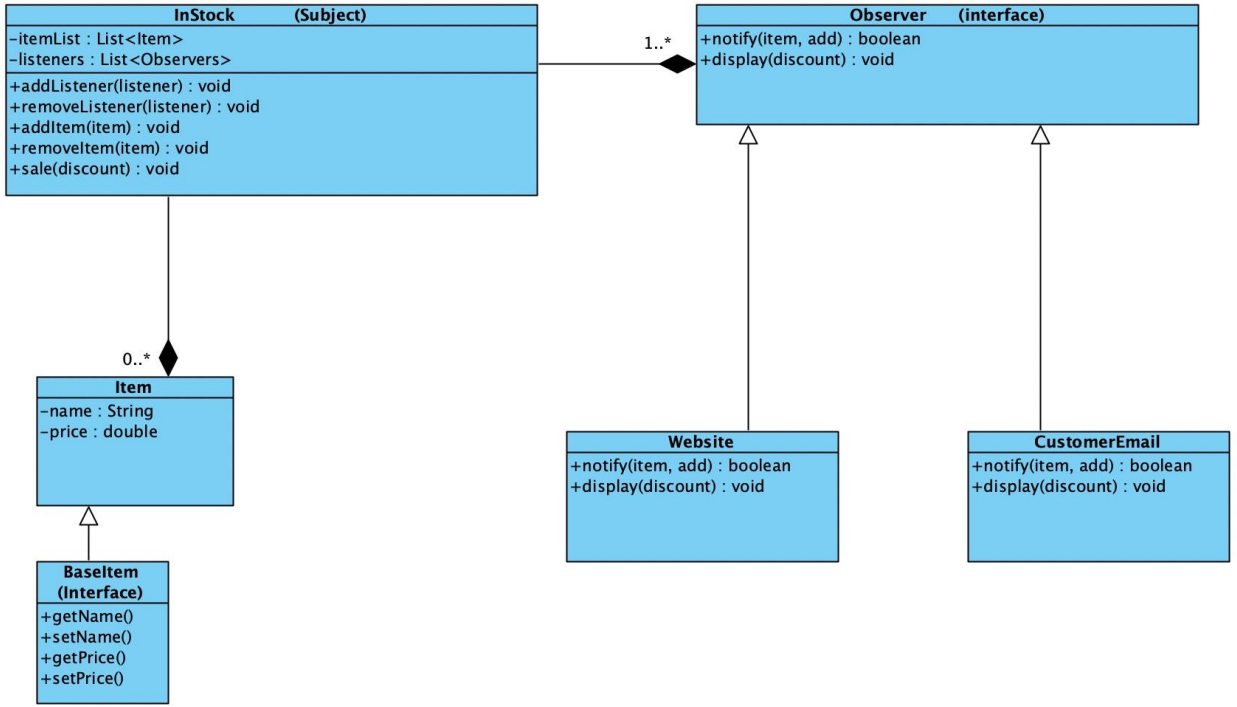
Observer Pattern (+UML Diagram)

Goals:

-Notify customers about an item's availability status

-Notify customers if there is a store sale, and update the site

-Note: In the current plan, we will consider one website and multiple customer emails as the observers



Observer Pattern-Java Code

Item class implements a BaseItem interface. Each item consists of a name and a price, allowing the use of these variables in the rest of the program.

```java
1 usage  1 implementation
public interface BaseItem {
    no usages  1 implementation
    void setName(String name);
    2 usages  1 implementation
    String getName();
    no usages  1 implementation
    void setPrice (double price);
    2 usages  1 implementation
    double getPrice();
}

//Behavior of a base item that will be implemented by class Item.
```

```java
10 usages
public class Item implements BaseItem {
    3 usages
    private String name;
    3 usages
    private double price;     // Item will be identified by its name and price
    3 usages
    public Item (String name, double price) {
        this.name= name;
        this.price= price;
    }
    //add getter and setters for both
    2 usages
    @Override
    public String getName() {
        return name;
    }
    no usages
    @Override
    public void setName(String name) {
        this.name =name;
    }
    2 usages
    @Override
    public double getPrice() {
        return price;
    }
    no usages
    @Override
    public void setPrice(double price) {
        this.price =price;
    }
}
```

The Subject- Java Code

The InStock class keeps track of changes in the store such as item availability. It also allows observers to be notified of such changes. We require a list of items(in stock) and a list of listeners:Observers. Methods include adding, removing either items or listeners, and tracking any discounts.

```java
import java.util.ArrayList;
import java.util.List;
6 usages
public class InStock {
    4 usages
    public final List<Item> itemList = new ArrayList <>();              //declare list of items
    5 usages
    private final List<Observers> listeners = new ArrayList <>();      //declare list of observers

    //methods to add+remove listeners such as website and emails
    4 usages
    public void addListener(Observers listener) { listeners.add(listener); }
    no usages
    public void removeListener(Observers listener) { listeners.remove(listener); }
```

```java
    //methods to add+remove items to stock
    //notify observers of the same
    2 usages
    public void addItem(Item item)
    {
        itemList.add(item);
        for(Observers observer:listeners)
            observer.notify(item, add: true);
    }
    2 usages
    public void removeItem(Item item)
    {
        itemList.remove(item);
        for(Observers observer:listeners)
            observer.notify(item, add: false);
    }
    // notify observers if there will be a sale and inform them of the discount
    3 usages
    public void sale (int discount) {
        for(Observers observer:listeners)
            observer.display(discount);
    }
}
```

The Observers-Java Code

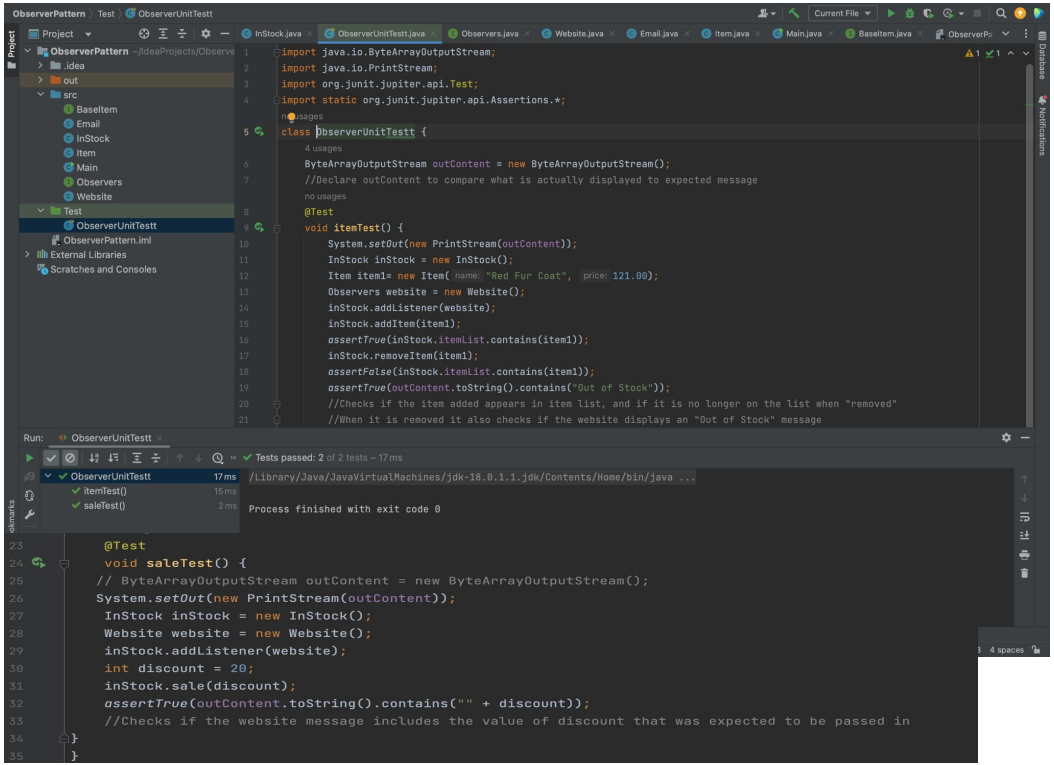
We have an observer interface defining the common behavior of all our observers for the online clothing store. One type of observer is customer email which will display specific notifications to customers. Another is the online site itself which will also display a message with any update.

```java
9 usages  2 implementations
public interface Observers {

    //observers will need to know what the item is and whether it was added or not
    //observers will also be notified of a discount
    2 usages  2 implementations
    boolean notify(Item item, Boolean add);
    1 usage  2 implementations
    void display(int discount);
}
```

```java
1 usage
public class Email implements Observers {
  //There may be multiple emails due to multiple customers.
    2 usages
    @Override
    public boolean notify(Item item, Boolean add) {
        //display the following messages
        System.out.println("Email Notification");
        System.out.println(item.getName() + "=" + item.getPrice());
        if (add == true)
            System.out.println("In Stock");
        else System.out.println("Out of Stock");
        return false;
    }
    1 usage
    @Override
    public void display(int discount) {
        //send a message with the discount that was passed in
        System.out.println("New Email Notification");
        System.out.println("Exciting Offer: "+discount+"% OFF Just For You!");
    }
}
```

```java
4 usages
public class Website implements Observers {
    2 usages
    @Override
    public boolean notify(Item item, Boolean add) {
        //item and whether it was added or not, is passed into the method

        System.out.println("Website Update");     //alert for an update
        System.out.println(item.getName() + "=" + item.getPrice());
        if (add == true)
            System.out.println("In Stock");         //display these messages
        else System.out.println("Out of Stock");
        return false;
    }

    1 usage
    public void display(int discount) {
        //display message with whatever the given discount is
        System.out.println("Website Update");
        System.out.println("Exciting Offer:"+discount+"% OFF");
    }
}
```

Observer Pattern- Unit Tests



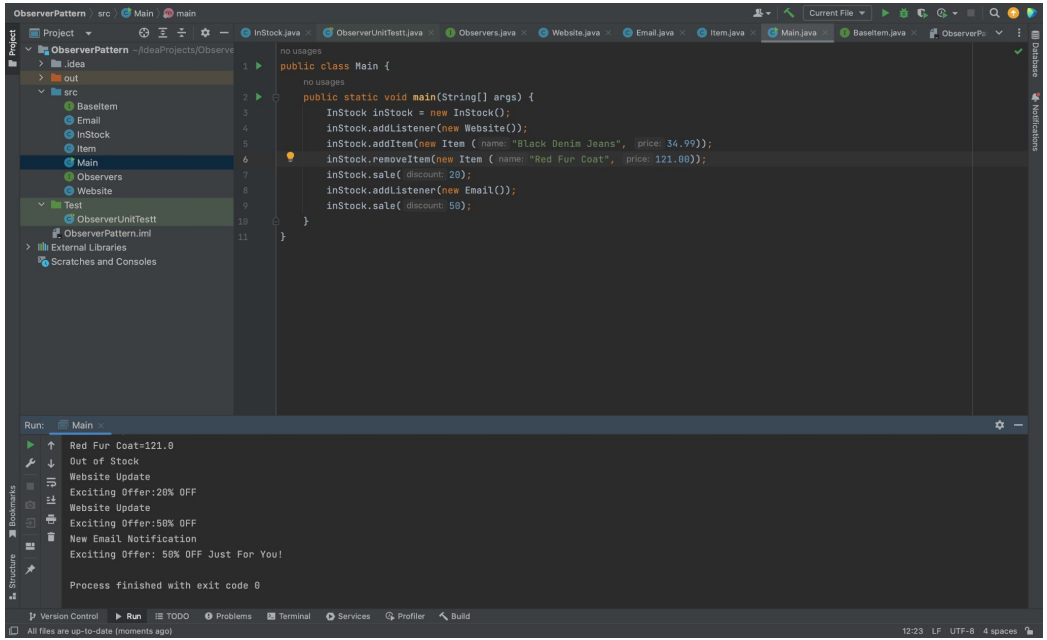
Unit Test 1:

Testing if items are added and removed appropriately, as well as if it displays an “out of stock” message on website.

Unit Test 2:

Testing if the value of discount is correct in the expected message displayed on the website.

Observer - Component Test

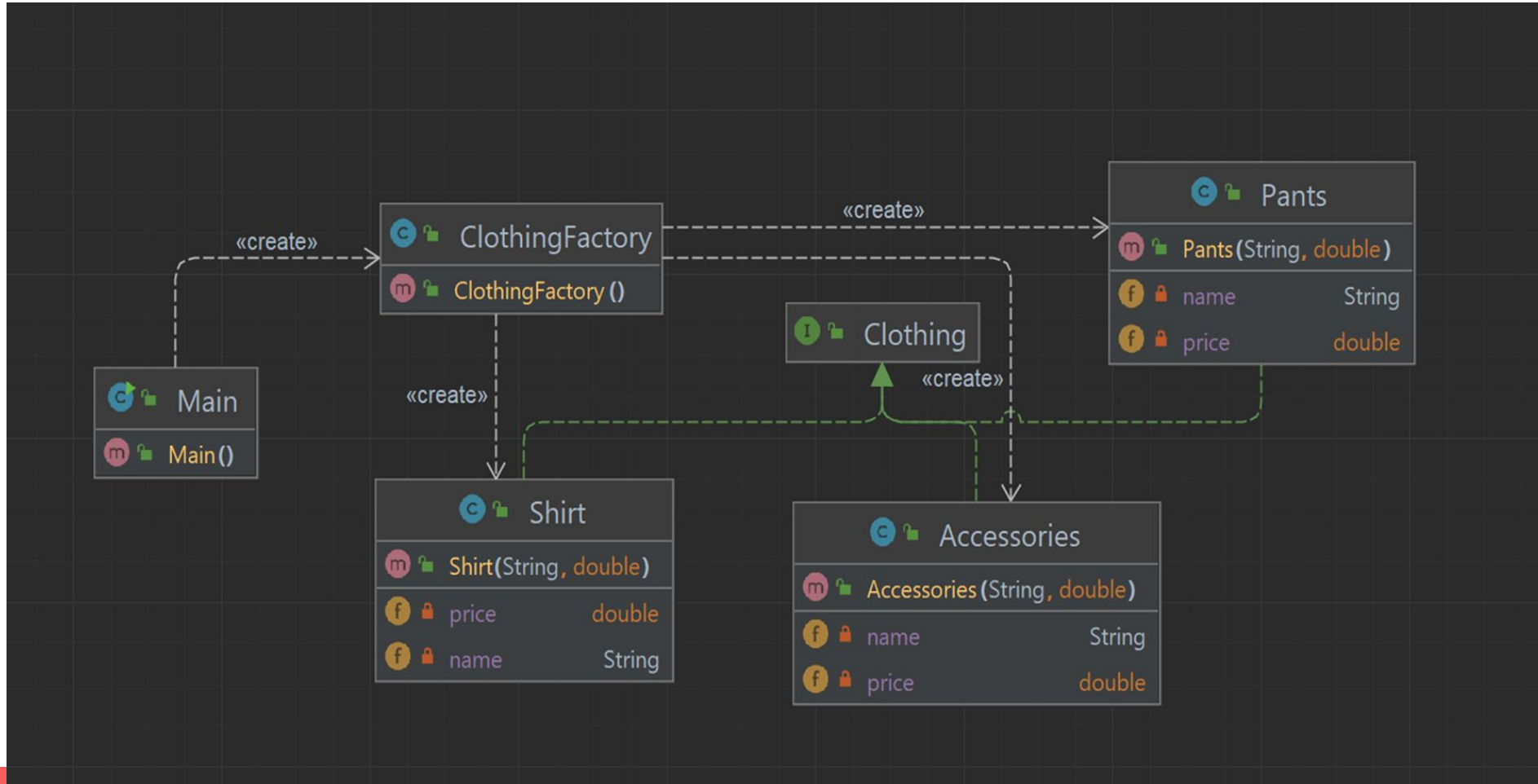


When “Black Denim Jeans” are added and “Red Fur Coat” is removed, the website displays “Black Denim Jeans” = 34.99 “In Stock” and for the latter, “121.00” and “out of stock”. For discount, only website is updated to display a 20 percent discount while both website and email get notified of a 50 percent discount after.

Shahnia khan-Factory Pattern

1. Define an abstract class or interface that contains the common methods for the objects you want to create.
2. Create concrete classes that implement the abstract class or interface, providing their own implementation of the methods.
3. Create a factory class with a method that accepts parameters to determine which concrete class to instantiate and returns an object of the abstract class or interface type.
4. The factory method returns an object of the abstract class or interface type, which can then be used by the client code without needing to know the concrete class.

UML Diagram - Factory



Main class

```java
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        ClothingFactory factory = new ClothingFactory();
        Scanner scanner = new Scanner(System.in);

        System.out.println("Welcome to our Online Clothing Store!");

        while (true) {
            System.out.println("What are you looking for today? please enter shirt/pants/accessories/exit");
            String input = scanner.nextLine();

            if (input.equalsIgnoreCase( anotherString: "exit")) {
                break;
            }

            Clothing clotheObject = null;

            switch (input.toLowerCase()) {
                case "shirt":
                    clotheObject = factory.createClothing("shirt", "White Shirt", 10.99);
                    break;
                case "pants":
                    clotheObject = factory.createClothing("pants", "Black Jeans", 59.99);
                    break;
                case "accessories":
                    clotheObject = factory.createClothing("accessories", "Brown Belt", 15.99);
                    break;
                default:
                    System.out.println("Invalid input. Please try again.");
                    continue;
            }

            System.out.println("You have selected: " + clotheObject.getName() + ", And the price is: $" + clotheObject.getPrice());
        }

        scanner.close();
        System.out.println("Thank you for shopping with us!");
    }
}
```

Clothing Interface

1. Promoting loosely coupling.
2. This ‘Clothing’ interface specifies two methods: getName() and getPrice(). These methods are implemented in each of the concrete clothing classes (Shirt, Pants, and Accessories), ensuring that they all have these same behaviors, even though they represent different types of clothing.

Shirt, Pants and the Accessories classes

```java
public class Shirt implements Clothing {
    2 usages
    private String name;
    2 usages
    private double price;

    public Shirt(String name, double price) {
        this.name = name;
        this.price = price;
    }

    public String getName() { return this.name; }

    public double getPrice() { return this.price; }
}
```

```java
public class Pants implements Clothing {
    2 usages
    private String name;
    2 usages
    private double price;

    public Pants(String name, double price) {
        this.name = name;
        this.price = price;
    }

    public String getName() { return this.name; }

    public double getPrice() { return this.price; }
}
```

```java
public class Accessories implements Clothing {
    2 usages
    private String name;
    2 usages
    private double price;
    public Accessories(String name, double price) {
        this.name = name;
        this.price = price;
    }

    @Override
    public String getName() { return name; }
    @Override
    public double getPrice() { return price; }
}
```

Clothing Factory

```
public class ClothingFactory {

    public Clothing createClothing(String type, String name, double price) {


        if (type.equalsIgnoreCase( anotherString: "shirt")) {
            return new Shirt(name, price);

        }

        else if (type.equalsIgnoreCase( anotherString: "pants")) {
            return new Pants(name, price);
        }

       else if (type.equalsIgnoreCase( anotherString: "accessories")) {
          return new Accessories(name, price);
       }

        else {
            return null;
        }
    }

}
```

Unit Tests

```java
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

class PantsTest {

    @Test
    void getName_ReturnsCorrectName() {

        String name = "Black Jeans";
        double price = 59.99;
        Pants pants = new Pants(name, price);
        String result = pants.getName();
        assertEquals(name, result);
    }

    @Test
    void getPrice_ReturnsCorrectPrice() {
        String name = "Black Jeans";
        double price = 59.99;
        Pants pants = new Pants(name, price);
        double result = pants.getPrice();
        assertEquals(price, result);
    }
}
```

```java
port org.junit.jupiter.api.Test;
port static org.junit.jupiter.api.Assertions.*;

ass ShirtsTest {


  @Test
  void getName_ReturnsCorrectName() {
      String name = "White Shirt";
      double price = 10.99;
      Shirt shirt = new Shirt(name, price);
      String result = shirt.getName();
      assertEquals(name, result);
  }

  @Test
  void getPrice_ReturnsCorrectPrice() {
      String name = "White Shirt";
      double price = 10.99;
      Shirt shirt = new Shirt(name, price);
      double result = shirt.getPrice();
      assertEquals(price, result);
  }
```

Component Test

1. Once the code ran, it welcomes the user and ask what they are looking for

2. It suggests the user to enter Shirt/Pants/Accessories/Exit

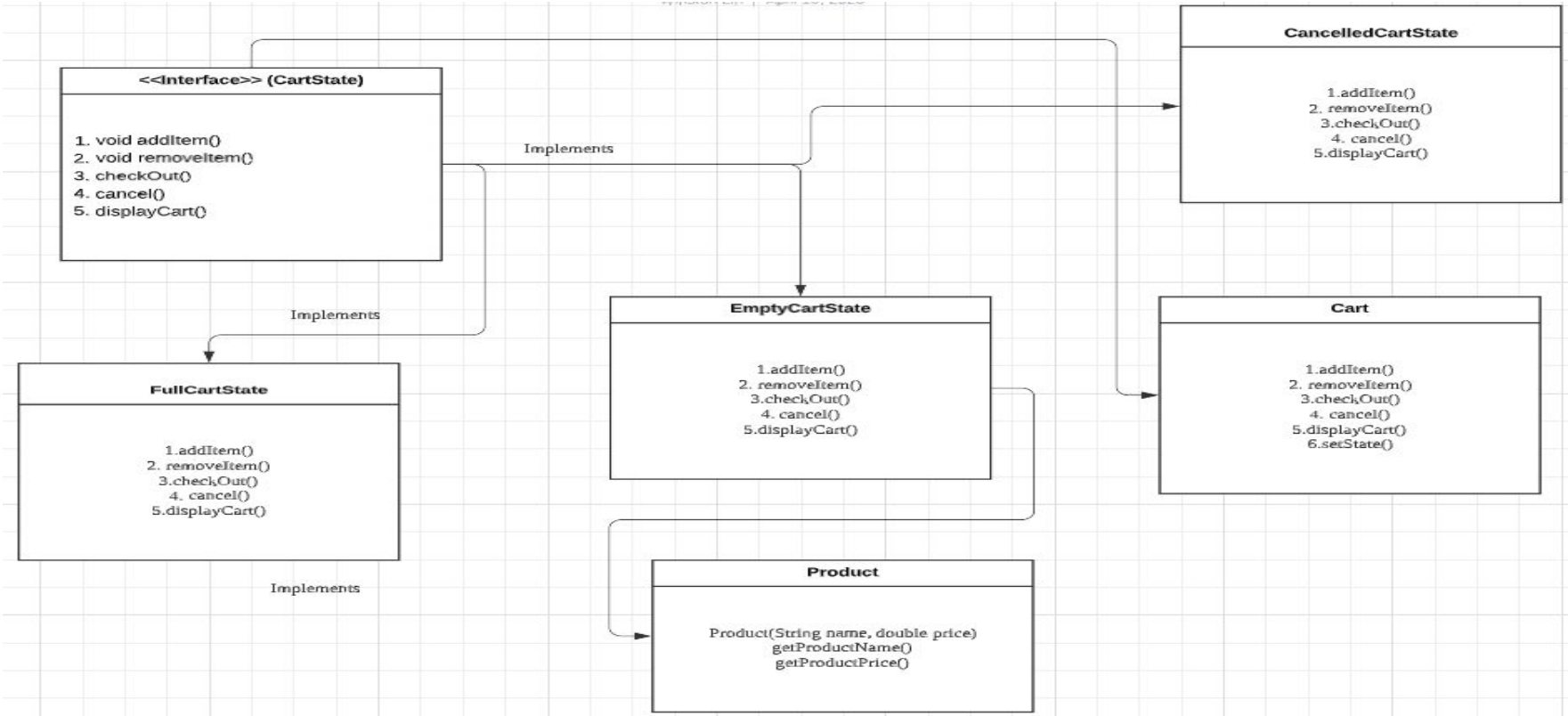
3. When Pants or Shirt or Accessories are entered as user input, the console displays what the customer has selected and the price of the. The price for white “Shirt” the price will show $10.99, for “Black Jeans” it is $59.99 and “Brown Belt” will be $15.99. If the user type exit the code will break an thank the customer

What is State Pattern?

Winston Lin

In programming, State Pattern is a behavior software design pattern that allows object to alter its behavior when its internal state changes.

State Pattern UML Diagram



CartState

This CartState interface act as a blueprint for other cartstates.

```
1   public interface CartState {
2
3       void addItem(Item product); // Add Product to Shopping cart.
4       void removeItem(Item product); // Remove the selected product from shopping cart.
5       void checkout(); // Check out all the product in the shopping cart.
6       void cancel(); // Cancel the current cart. -> Null
7       void displayCart(); // Display all the product in the current cart.
8
9
10  }
```

EmptyState

```java
public class EmptyCartState implements CartState {

    private Cart cart;

    public EmptyCartState (Cart cart) {
        this.cart = cart;
    }

    @Override
    public void addItem(Item product) {
        cart.setState(new FullCartState(cart)); // Once an product is added, it is no longer empty.
        cart.addProduct(product);
        System.out.println(product.getName() + " added successfully to the cart."); // Notify
    }

    @Override
    public void removeItem(Item product) {
        System.out.println(x:"Cannot remove item. Cart is empty.");
    }

    @Override
    public void checkout() {
        System.out.println(x:"Cannot checkout. Cart is empty.");
    }

    @Override
    public void cancel() {
        System.out.println(x:"Cannot cancel. Cart is already empty.");
    }

    @Override
    public void displayCart() {
        System.out.println(x:"Your cart is empty.");
    }

}
```

FullCartState

```java
public class FullCartState implements CartState {

    private Cart cart;

    public FullCartState(Cart cart){
        this.cart = cart;
    }

    @Override
    public void addItem(Item product) {
        cart.addProduct(product);
        System.out.println("Product " + product.getName() + " is added successfuly");
    }

    @Override
    public void removeItem(Item product) {

        cart.removeProduct(product);

        if (cart.isEmpty()) { // After the removal check the cart state.
            cart.setState(new EmptyCartState(cart)); // Set the state if the current state is empty.
        }
        System.out.println(product.getName() + " has been removed from the cart successfuly."); // Ale

    }

    @Override
    public void checkout() {
        System.out.println(x:"Cart has been checked out successfully");
        cart.setState(new CancelledCartState(cart));

    }

    @Override
    public void cancel() {
        cart.setState(new EmptyCartState(cart));
        System.out.println(x:"Shopping cart is being cancelled...");


    }

    @Override
    public void displayCart() {
        cart.display();
    }

}
```

CancelledCartState

```java
public class CancelledCartState implements CartState {
    private Cart cart;

    public CancelledCartState(Cart cart) {
        this.cart = cart;
    }

    @Override
    public void addItem(Item product) {
        System.out.println(x:"Error: Unable to add product...");
        System.out.println(x:"Cart Cancelled...");
    }

    @Override
    public void removeItem(Item product) {
        System.out.println(x:"Error: Unable to remove product...");
        System.out.println(x:"Cart Cancelled...");
    }

    @Override
    public void checkout() {
        System.out.println(x:"Error: Unable to checkout product...");
        System.out.println(x:"Cart Cancelled...");
    }

    @Override
    public void cancel() {
        System.out.println(x:"Error: Unable to complete task.");
        System.out.println(x:"Cart has already been cancelled.");
    }

    @Override
    public void displayCart() {
        System.out.println(x:"Error: Unable to display the task.");
        System.out.println(x:"Cart has been cancelled.");
    }
}
```

Unit Test

```java
public class CartTest {

    //@Test
    public void testEmptyCart() {
        Cart cart = new Cart();
        System.out.println(cart.isEmpty());
        cart.displayCart();
    }

    //@Test
    public void testAddItem() {
        Cart cart = new Cart();
        Item item = new Item(name:"Product 1", price:10.0);
        cart.addItem(item);

        cart.displayCart();
    }

    //@Test
    public void testRemoveItem() {
        Cart cart = new Cart();
        Item item1 = new Item(name:"Product 1", price:10.0);
        Item item2 = new Item(name:"Product 2", price:15.0);
        cart.addItem(item1);
        cart.addItem(item2);
        cart.removeItem(item1);

        cart.displayCart();
    }

    //@Test
    public void testCheckout() {
        Cart cart = new Cart();
        Item item1 = new Item(name:"Product 1", price:10.0);
        Item item2 = new Item(name:"Product 2", price:15.0);
        cart.addItem(item1);
        cart.addItem(item2);
        cart.checkOut();
        cart.displayCart();
    }

    //@Test
    public void testCancel() {
        Cart cart = new Cart();
        Item item1 = new Item(name:"Product 1", price:10.0);
        Item item2 = new Item(name:"Product 2", price:15.0);
        cart.addItem(item1);
        cart.addItem(item2);
        cart.cancel();
        cart.displayCart();
    }
}
```

Component Test

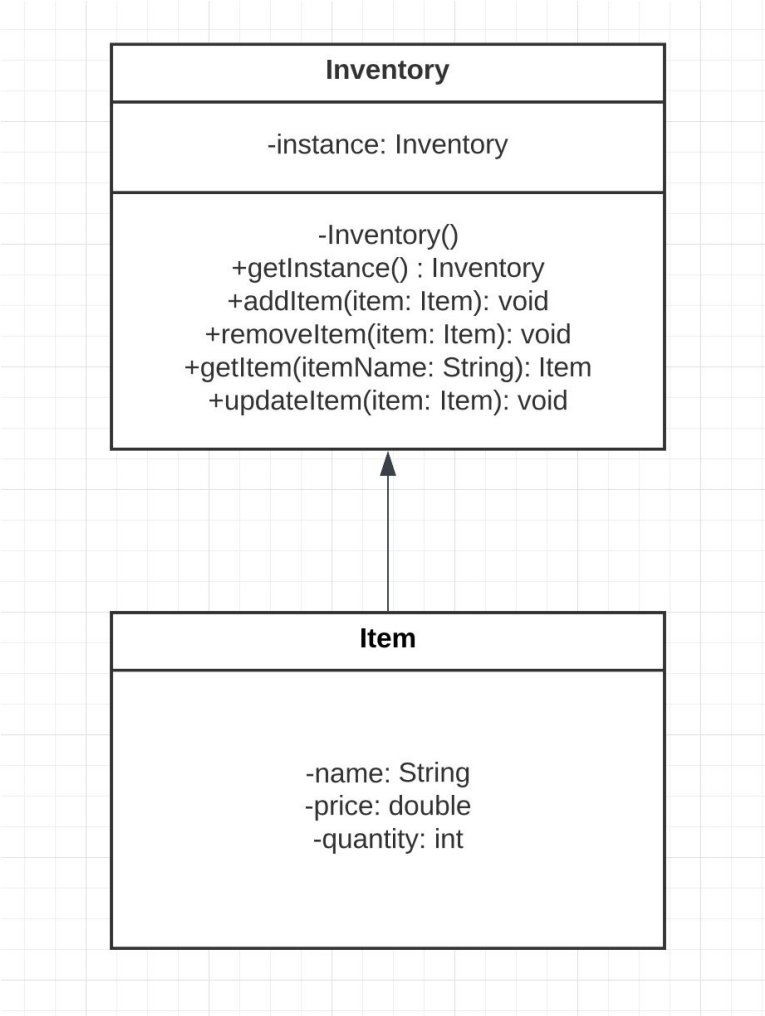
```java
public class Main {


    Run | Debug
    public static void main(String[] args) {

        Cart cart = new Cart();

        Item item1 = new Item(name:"Product 1", price:12.39); //
        Item item2 = new Item(name:"Product 2", price:13.00);
        Item item3 = new Item(name:"Product 3", price:100.23);

        System.out.println(cart.isEmpty()); // Current Cart state should be empty.
        cart.addItem(item1); // Add item1 to the cart
        cart.addItem(item2); // Add item2 to the cart

        System.out.println(cart.isEmpty()); // The cart has several item.

        cart.removeItem(item2); // Remove Item2 from the cart.
        cart.addItem(item3);    // Add item3 to the cart. Shopping Cart -> Product1, Product3.
        cart.cancel();  // Set the current cartState to empty.
        cart.displayCart(); // Display



    }


}

```

```
true
Product 1 added successfully to the cart.
Product Product 2 is added successfuly
false
Product 2 has been removed from the cart successfuly.
Product Product 3 is added successfuly
Shopping cart is being cancelled...
Your cart is empty.
```

Singleton Design Pattern - Elan Abramov

The Singleton Design pattern restricts class instantiation to a single instance. Instead of making a new instance each time, we update the existing instance. In this case the singleton pattern was used to make an inventory list of the items available.

UML Diagram of Singleton Design Pattern



Singleton Pattern Code

- Creates an instance of an Inventory List
- Adding items to that instance
- Removing items to that instance

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

public class InventoryList {
    private static InventoryList instance = null;
    private List<Item> itemList;

    private InventoryList() {
        itemList = new ArrayList<>();
    }

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

    public void addItem(Item item) {
        itemList.add(item);
    }

    public void removeItem(Item item) {
        itemList.remove(item);
    }
```

Singleton Pattern Code cont.

- Gets and item from the instance
- Update and item from the instance

```
    public Item getItem(String itemName) {
        for (Item item : itemList) {
            if (item.getName().equals(itemName)) {
                return item;
            }
        }
        return null;
    }

    public void updateItem(Item item) {
        for (int i = 0; i < itemList.size(); i++) {
            Item currentItem = itemList.get(i);
            if (currentItem.getName().equals(item.getName())) {
                itemList.set(i, item);
                break;
            }
        }
    }
}
```

Singleton Pattern Code cont.

- Item class that creates items to put in the Inventory List
- Creates a name of an item
- Creates a price for an item

```
public class Item implements BaseItem {
    private String name;
    private double price;     // Item will be identified by its name and price
    public Item (String name, double price) {
        this.name= name;
        this.price= price;
    }
    //add getter and setters for both
    @Override
    public String getName() {
        return name;
    }
    @Override
    public void setName(String name) {
        this.name =name;
    }
    @Override
    public double getPrice() {
        return price;
    }
    @Override
    public void setPrice(double price) {
        this.price =price;
    }
}
```

Unit Test

- Tests the inventory list by adding “Shirts” at a price of $20 to the inventory list
- Tests the inventory list by removing “Shirts” at the price of $20 from the inventory list

```java
import org.junit.Test;
import static org.junit.Assert.*;

public class InventoryListTest {

    //unit test to add items to the inventory list
    @Test
    public void testAddItem() {
        InventoryList inventoryList = InventoryList.getInstance();
        Item item = new Item("Shirt", 20.0);
        inventoryList.addItem(item);
        assertEquals(item, inventoryList.getItem("Shirt"));
    }


    //unit test to remove items from the inventory list
    @Test
    public void testRemoveItem() {
        InventoryList inventoryList = InventoryList.getInstance();
        Item item = new Item("Shirt", 20.0);
        inventoryList.addItem(item);
        inventoryList.removeItem(item);
        assertNull(inventoryList.getItem("Shirt"));
    }

}
```

Component Test

The owner would add clothes to their inventory list and it would be updated to the app/website so that users can see how many were added and what the price of each one would be. In this case, shirts were added at a price of 20 dollars each shirt. Then the second unit test removes the shirts that have a price of $20 and take it out of the inventory list.