C322 Final Project

Christina, Maddi, James

Online Clothing Shop

We have created an online customizable clothing shop!

Our shop is responsible for creating the clothes.

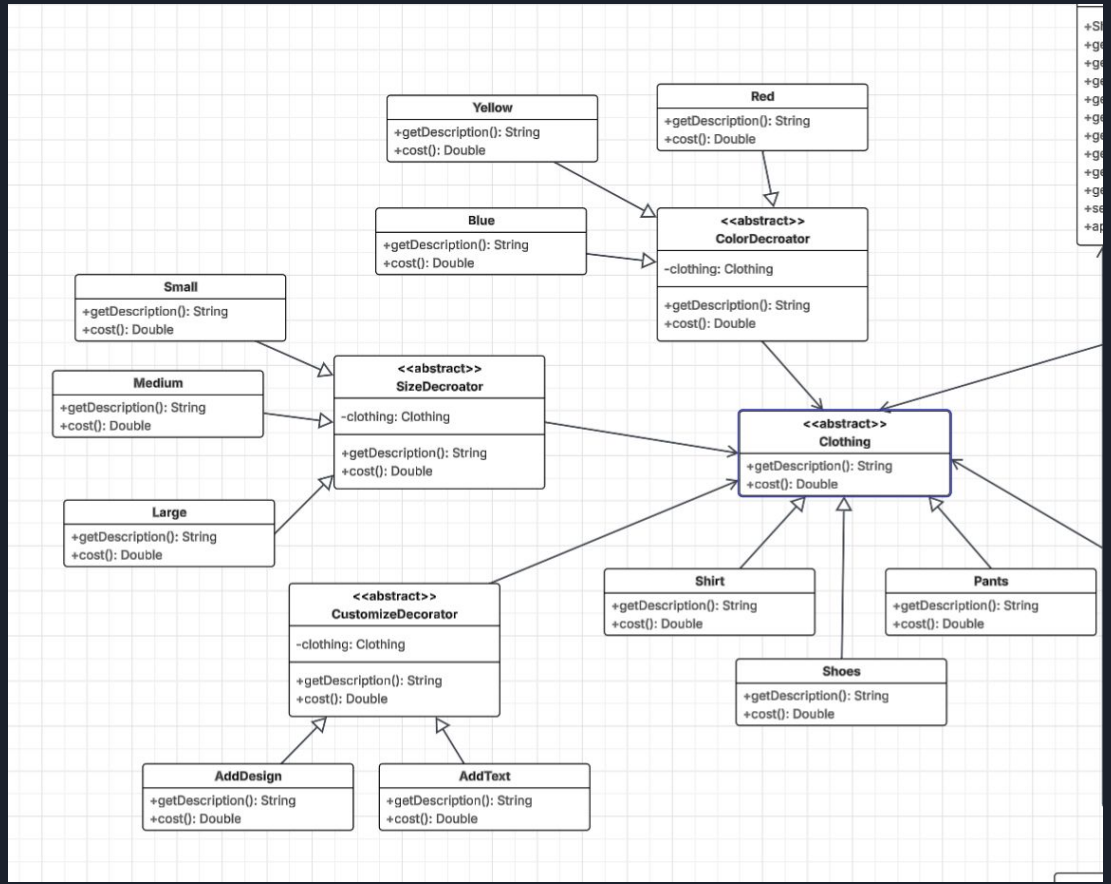
Users are able to:

- Sign up for our service and notifications
- Order and pay for different clothing combinations

Our shop utilizes the decorator, observer, factory, and strategy design patterns.

Decorator Pattern UML

- Allows for easy customization
- Wraps objects around other objects to modify their behavior



Decorator Pattern

Clothing is our abstract base class, from there, there are three classes that extend this: Shirt, Pants, and Shoes

```
public abstract class Clothing {
    4 usages
    String description = "clothing";

    11 overrides   new *
    public String getDescription() {
        return description;
    }

    11 implementations   new *
    public abstract double cost();

}
```

```
public class Shirt extends Clothing {
    1 usage   new *
    public Shirt() { description = "Shirt "; }

    new *
    public double cost() { return 25.00; }
}
```

Decorator Pattern

There are three decorator classes: size decorator, customize decorator, and color decorator

Users are able to choose three sizes: small, medium, or large

They can pick three colors for their clothing: red, yellow, or blue

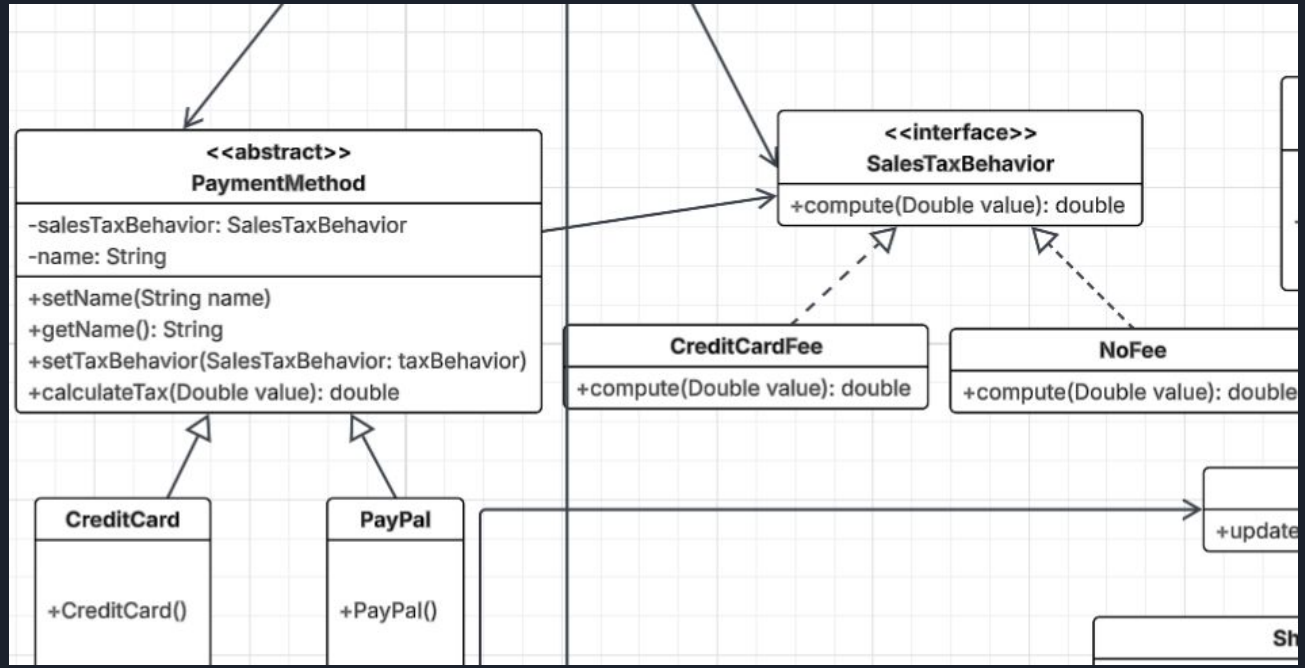
They can also add a custom design or text to their clothing

```
public abstract class SizeDecorator extends Clothing {
    9 usages
    Clothing clothing;

    3 implementations   new *
    public abstract String getDescription();
}
```

```
public class Small extends SizeDecorator {
    1 usage   new *
    public Small(Clothing clothing) { this.clothing = clothing; }

    new *
    public String getDescription() { return clothing.getDescription() + "in the size small "; }

    new *
    @Override
    public double cost() { return clothing.cost(); }

}
```

Strategy Design Pattern

- Users are able to change behavior at runtime
- Benefits from loose coupling



Strategy Design Pattern

Payment Method is our abstract class, we have two payment methods available: paypal and credit card

Each payment method will have a different tax behavior

```java
public abstract class PaymentMethod {
    2 usages
    SalesTaxBehavior salesTaxBehavior;
    2 usages
    String name;

    new *
    public String getName() { return name; }

    new *
    public void setName(String name) { this.name = name; }

    2 usages  new *
    public void setTaxBehavior(SalesTaxBehavior taxBehavior) { this.salesTaxBehavior = taxBehavior; }

    2 usages  new *
    public double calculateTax(Double value) {
        return salesTaxBehavior.compute(value);
    }
}
```

```java
public class PayPal extends PaymentMethod {
    1 usage  new *
    public PayPal() { this.setName("paypal"); }
}
```

```java
public class CreditCard extends PaymentMethod{
    1 usage  new *
    public CreditCard() { this.setName("creditCard"); }
}
```

Strategy Design Pattern

There are two different tax behaviors: a 5% fee for purchases made with a credit card, and no fee for all other options

```
public class NoFee implements SalesTaxBehavior {
    1 usage   new *
    public double compute(Double value) { return 0.0; }
}
```

```
public interface SalesTaxBehavior {
    1 usage   2 implementations   new *
    double compute (Double value);
}
```

```
public class CreditCardFee implements SalesTaxBehavior{
    1 usage   new *
    @Override
    public double compute(Double value) { return value * 0.05; }
}
```

Observer Pattern Code

- **Observer Interface:** defines update method for classes to receive notifications
- **Subject Interface:** registers, removes, and notifies shoppers
- **Shopper Class:** ensures each shopper receives updates from ShopperNotifications
- **ShopperNotifications:** stores ArrayList of shoppers and notifies them

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

```java
public interface Subject {
    void registerObserver(Observer o);
    void removeObserver(Observer o);
    void notifyObservers(String message);
}
```

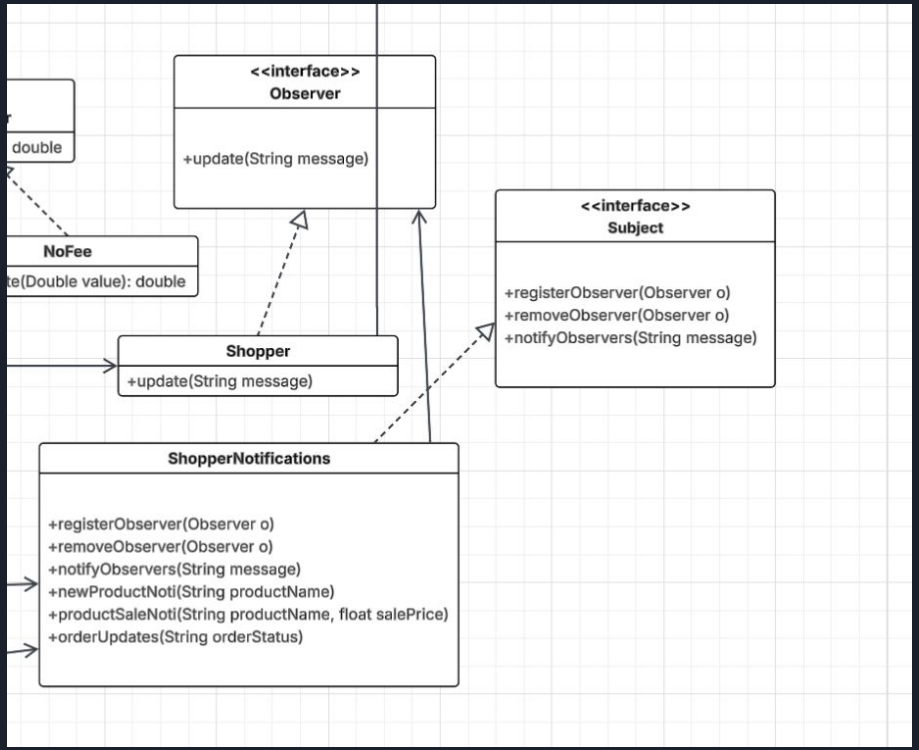
```java
public class Shopper implements Observer{
    private String name;
    private ShopView view;

    public Shopper(String name, ShopView view) {
        this.name = name;
        this.view = view;
    }

    @Override
    public void update(String message) {
        if (view != null) {
            view.appendNotification("Hello " + name + "! " + message);
        }
```

```java
public class ShopperNotifications implements Subject {
    private List<Observer> shoppers;

    public ShopperNotifications() {
        shoppers = new ArrayList<>();
    }

    @Override
    public void registerObserver(Observer o) {
        shoppers.add(o);
    }

    @Override
    public void removeObserver(Observer o) {
        shoppers.remove(o);
    }

    @Override
    public void notifyObservers(String message) {
        for (Observer o : shoppers) {
            o.update(message);
        }
    }

    public void newProductNoti(String productName){
        notifyObservers("New product available: " + productName + "!");
    }

    public void productSaleNoti(String productName, float salePrice){
        notifyObservers(productName + " is currently on sale for $" + salePrice);
    }

    public void orderUpdates(String orderStatus){
        notifyObservers("Order update: " + orderStatus);
```

Observer Pattern UML

- Shopper implements Observer interface
- ShopperNotifcations implements subject interface

- Registers and maintains a list of shoppers
- Notifies shoppers for sales, new products, and order updates



Factory Pattern Code

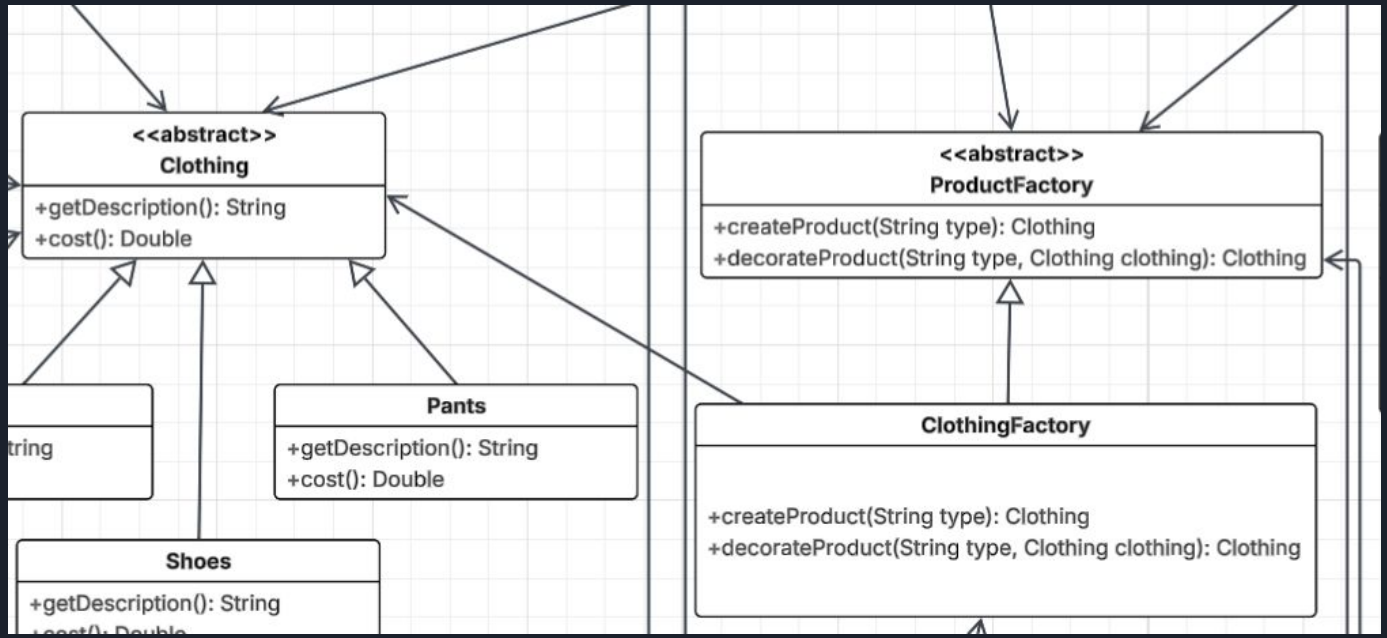
- ProductFactory defines behavior for other factories to create and customize products
- ClothingFactory extends ProductFactory and creates and allows for clothing to be created and decorated

```java
import java.util.List;

public abstract class ProductFactory {
    public abstract Clothing createProduct(String type);
    public abstract Clothing decorateProduct(String[] type, Clothing clothing);

    public abstract List<String> getProductNames();
    public abstract List<String> getSizeOptions();
    public abstract List<String> getColorOptions();
    public abstract List<String> getDesignOptions();
    public abstract List<String> getTextOptions();
}
```

```java
import java.util.List;
public class ClothingFactory extends ProductFactory {
    private final List<String> productNames = List.of("shirt", "pants", "shoes");
    private final List<String> sizeOptions = List.of("small", "medium", "large");
    private final List<String> colorOptions = List.of("red", "blue");
    private final List<String> designOptions = List.of("design", "none");
    private final List<String> textOptions = List.of("text", "none");

    public Clothing createProduct(String type) {
        return switch (type.toLowerCase()) {
            case "shirt" -> new Shirt();
            case "pants" -> new Pants();
            case "shoes" -> new Shoes();
            default -> throw new IllegalArgumentException("Invalid input for createProduct() - " + type);
        };
    }

    public Clothing decorateProduct(String[] decoratorType, Clothing clothing) {
        return switch(decoratorType[0].toLowerCase()) {
            case "small"  -> new Small(clothing);
            case "medium" -> new Medium(clothing);
            case "large"  -> new Large(clothing);
            case "red"    -> new Red(clothing);
            case "blue"   -> new Blue(clothing);
            case "text"   -> new AddText(clothing, decoratorType[1]);
            case "design" -> new AddDesign(clothing);
            default -> throw new IllegalArgumentException("Invalid input for createProduct() - " + decoratorType[0]);
        };
    }
```

Factory Pattern UML

- ClothingFactory extends ProductFactory
- ClothingFactory is associated with and produces Clothing types
- ShopModel, ShopController, and ShopView all use ProductFactory



Model-View-Controller Pattern (GUI)

Implemented GUI with
Model-View-Controller Pattern.

```
public class ShopView extends JFrame {
    JComboBox<String> productCombo;
    JTabbedPane tabbedPane;
    JComboBox<String> sizeCombo;
    JComboBox<String> colorCombo;
    JComboBox<String> designCombo;
    JTextField textField;

    JComboBox<String> paymentMethodCombo;
    JComboBox<String> shippingMethodCombo;

    JButton orderButton;
    JTextArea outputArea;
    JTextArea notificationArea;

    public ShopView(ProductFactory factory) {
        super("Custom Shop");
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(960, 540);
        setLayout(new BorderLayout(5, 5));

        JPanel north = new JPanel(new FlowLayout(FlowLayout.LEFT));
        north.add(new JLabel("Product:"));
        productCombo = new JComboBox<>(factory.getProductNames().toArray(new String[0]));
        north.add(productCombo);
        add(north, BorderLayout.NORTH);
        tabbedPane = new JTabbedPane();
```

```
public class ShopModel {
    private Clothing current;

    public void setBaseProduct(String type, ProductFactory factory) {
        current = factory.createProduct(type);
    }

    public void addCustomization(String[] decorator, ProductFactory factory) {
        current = factory.decorateProduct(decorator, current);
    }

    public String getDescription() { return current == null ? "" : current.getDescription(); }

    public double getCost() { return current == null ? 0.0 : current.cost() ; }

    public double getTax(double clothingCost, String paymentMethod, String shippingMethod) {...}
```

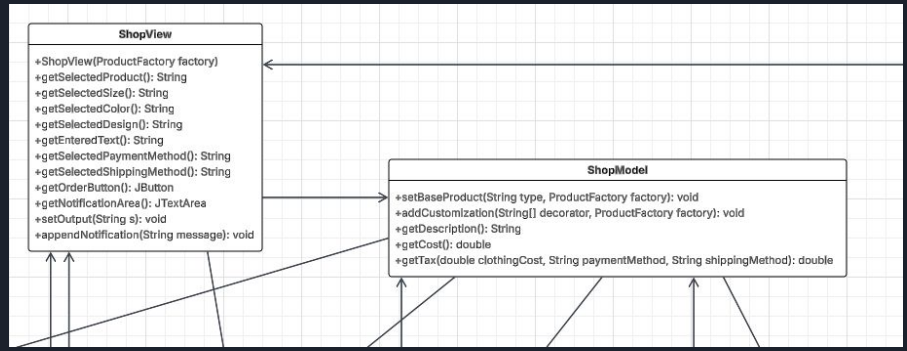
```
public class ShopController {
    private final ShopModel model;
    private final ShopView view;
    private final ProductFactory factory;

    ShopperNotifications store;

    public ShopController(ShopModel model, ShopView view, ProductFactory factory, ShopperNotifications store) {...}
```

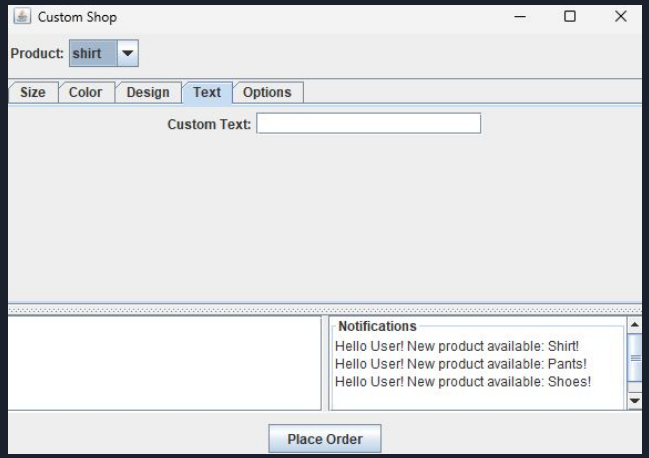
Model-View-Controller Pattern (GUI)

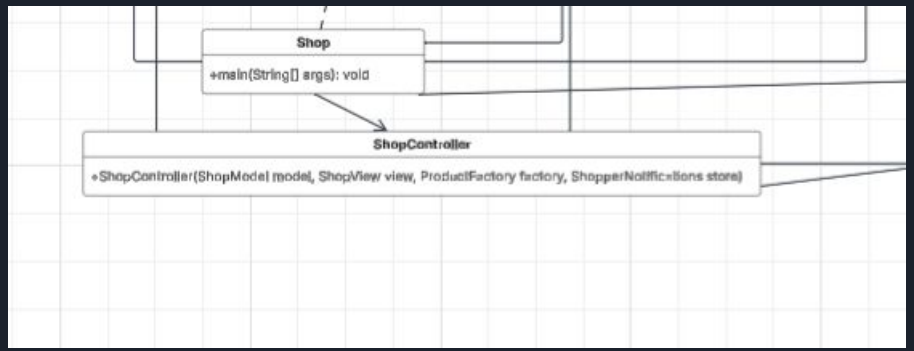
Model: manages logic and rules of application

View: displays data to user

Controller: handles user input, updates model







UML Overview