SWENG304: Software Design and Architecture
Shopping Mall Java Project

Yahya AlAyaseh
Supervisor: Dr. Yousef Hassouneh

Second Semester 2024/2025

ShoppingCart Refactoring and Design Pattern Application

Problem Review

Given the initial implementation of the `ShoppingCart` class:

```
public class ShoppingCart {
    private List<Item> items = new ArrayList<>();

    public void addItem(Item item) {
        items.add(item);
        System.out.println("Item added: " + item.getName());
        sendNotification("Added " + item.getName());
    }

    public void checkout() {
        processCreditCardPayment();
        sendNotification("Order placed.");
    }

    private void sendNotification(String message) {
        System.out.println("Email to user: " + message);
    }
```

```
    private void processCreditCardPayment() {
        System.out.println("Paid with Credit Card");
    }
}
```

Problems Identified

1. **Tightly coupled implementation**: Notification and payment mechanisms are hardcoded, making the class rigid and inflexible to change.
2. **Violates the Open/Closed Principle**: To add new behaviors (e.g., SMS notifications or PayPal payment), we must modify the existing `ShoppingCart` class.
3. **Low testability**: Since the class is not modular, it's difficult to substitute components for testing or mocking purposes.

Refactoring Using Strategy and Observer Patterns

Strategy Pattern: Payment Handling

PaymentStrategy Interface

```
public interface PaymentStrategy {
    void pay(double amount);
}
```

CreditCardPayment Implementation

```
public class CreditCardPayment implements PaymentStrategy {
    public void pay(double amount) {
        System.out.println("Paid $" + amount + " using Credit Card");
    }
}
```

PayPalPayment Implementation

```
public class PayPalPayment implements PaymentStrategy {
    public void pay(double amount) {
        System.out.println("Paid $" + amount + " using PayPal");
    }
}
```

Observer Pattern: Notifications

CartObserver Interface

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

EmailNotifier Implementation

```
public class EmailNotifier implements CartObserver {
    public void update(String message) {
        System.out.println("Email to user: " + message);
    }
}
```

SMSNotifier Implementation

```
public class SMSNotifier implements CartObserver {
    public void update(String message) {
        System.out.println("SMS to user: " + message);
    }
}
```

Refactored ShoppingCart Class

```
public class ShoppingCart {
    private List<Item> items = new ArrayList<>();
    private List<CartObserver> observers = new ArrayList<>();
    private PaymentStrategy paymentStrategy;

    public void addItem(Item item) {
        items.add(item);
        notifyObservers("Added " + item.getName());
    }

    public void setPaymentStrategy(PaymentStrategy strategy) {
        this.paymentStrategy = strategy;
    }

    public void addObserver(CartObserver observer) {
        observers.add(observer);
    }
```

```
    private void notifyObservers(String message) {
        for (CartObserver observer : observers) {
            observer.update(message);
        }
    }

    public void checkout() {
        double total = items.stream().mapToDouble(Item::getPrice).sum()
        if (paymentStrategy != null) {
            paymentStrategy.pay(total);
            notifyObservers("Order placed.");
        } else {
            System.out.println("No payment method selected.");
        }
    }
}
```

Demonstration

```
public class Main {
    public static void main(String[] args) {
        ShoppingCart cart = new ShoppingCart();

        cart.addObserver(new EmailNotifier());
        cart.addObserver(new SMSNotifier());

        cart.addItem(new Item("Book", "B001", "S001", 50.0));
        cart.setPaymentStrategy(new CreditCardPayment());
        cart.checkout();

        System.out.println("\nSwitching payment strategy to PayPal...\n

        cart.setPaymentStrategy(new PayPalPayment());
        cart.checkout();
    }
}
```

Reflection

1. How did using design patterns improve the flexibility of your code?

Using the Strategy and Observer patterns allowed the separation of concerns and enabled dynamic behavior configuration at runtime. New payment methods or notification types can be added and injected without modifying the core class, increasing maintainability and scalability.

2. Would this design make testing easier? Why or why not?

Yes. This design facilitates unit testing of each component (payment strategies, notifiers) in isolation. It also allows mocking of interfaces during testing and simulates different behaviors without rewriting the `ShoppingCart` logic. The modular nature improves overall testability and code quality.