Object-Oriented Software Method Course Project Final Report

Maddi Acton
Object-Oriented Software Methods
Indiana University
Bloomington, Indiana
actonm@iu.edu

James Petersen
Object-Oriented Software Methods
Indiana University
Bloomington, Indiana
jampete@iu.edu

Christina Melnic
Object-Oriented Software Methods
Indiana University
Bloomington, Indiana
cmelnic@iu.edu

Abstract— **This paper explains the design and implementation of a customizable online store inspired by platforms like Etsy. The system allows users to choose from basic products and then customize them with options like color, text, size, and design. It also offers multiple shipping options and payment methods. It provides updates and notifications to users. The system is built in Java and uses five design patterns Factory, Decorator, Strategy, Observer, and Model-View-Controller to ensure flexibility, ease of maintenance, and future expansion. The paper outlines how each pattern was used, shows how the components interact, and explains the benefits of each pattern-based approach.**

Keywords—Factory pattern, Decorator pattern, Strategy pattern, Observer pattern, Model-View-Controller pattern

I. Introduction

The project addresses the need for flexible product customizations, payment options, and notifications, while ensuring the system remains easy to manage and update. It achieves this by building an online shop in Java utilizing design patterns. By separating concerns such as product creation, behavior extension, payment method selection, and notification management, the system can change without causing issues in other areas. This paper explores the approaches used, and details how the design patterns are applied, and the benefits of using each pattern for the scalability and flexibility of the overall system.

II. Implementation

A. Factory Pattern

The core of the program is the Factory pattern. It is utilized to standardize the creation of our Clothing objects. Rather than instantiating concrete classes directly within the main code, we defined an abstract ProductFactory interface with a createProduct method. This is implemented by ClothingFactory which produce instances of Shirt, Pants, and Shoes, all of which conform to a common Clothing interface. This approach decouples object creation from object use, making it straightforward to introduce entirely new categories of products-such as accessories or decoration simply by adding new factory implementations without touching the existing codebase.

B. Decorator Pattern

Building upon this foundation, we integrated the Decorator pattern to provide an extensible customization mechanism. An abstract class, CustomizeDecorator, implements the Clothing interface and holds a reference to another Clothing object, thereby "wrapping" it. Concrete decorators such as ColorDecorator, AddText, and SizeDecorator, extend CustomizeDecorator to alter the attributes of the underlying clothing object. For example, the ColorDecorator can apply a red or blue tint, while the TextDecorator overlays a custom text typed by the user. By layering multiple decorator instances around a single product, we enable users to have a variety of customizations without a massive number of classes. Each decorator is focused on a single customization option and new customization options can be introduced by simply writing an additional decorator subclass.

C. Strategy Pattern

To manage differences in payment processing, we implemented the Strategy Pattern. We defined a PaymentMethod interface that encapsulates the steps required to complete a transaction. Namely the setTaxBehavior and calculateTax methods. Concrete strategy classes, such as CreditCardPayment and PaypalPayment, each implement this interface and contain logic specific to the payment method. The payment methods carry variable fees. They both delegate the computation for the fee to the SalesTaxBehavior interface, which is implemented in CreditCardFee and NoFee, which

allows for more dynamic substitution of different payment methods. The dynamic substitution of the payment methods makes the system flexible and scalable.

D. Observer Pattern

We incorporated the Observer pattern to handle notifications. A Subject interface defines method to register, remove, and notify observers. A concrete subject ShopNotifications is defined to be triggered when events such as the addition of new products or order confirmations occur. Observers implement an Observer interface. For our purposes, the Shopper class implements the Observer interface and receives all notifications relevant to its orders. When the system state changes, such as when an order is placed, the ShopNotifications object broadcasts an update to all registered Shopper observers. By decoupling the event sources from what receives the event, we ensure that the notification system can grow independently as we introduce more observers or subjects.

E. Model-View-Controller Pattern

We applied the Model-View-Controller pattern for our graphical interface. The ShopModel encapsulates the application's data. Such as the current clothing item being ordered. The ShopView handles all logic for presentation. So it creates all of the text boxes and buttons needed for the graphical interface. The ShopController serves to interpret user actions and invoke the appropriate methods on the model or updating the view. This separation keeps the codebase organized and assists with keeping the system flexible and scalable.

III. Challenges

One of the primary challenges was enabling the system to handle a variety of product customizations without making the codebase overly complex or difficult to maintain. As new customization options, such as color, text, and size were added, the number of potential combinations increased. To address this, we used the Decorator pattern. This pattern allowed us to introduce new customizations as independent decorators that could be stacked on top of each other. This approach kept the code modular and easy to extend. Another challenge was since each individual was responsible for implementing different design patterns, we faced challenges when merging our individual work. Specifically, discrepancies in class names and behaviors when integrating. For example, one developer used the class name Product while another used Clothing to represent the same concept. This inconsistency required time and effort to resolve. In the future, clearer communication could help avoid these issues. Finally, it is important to note that one of our group members, Jack Schwartz, did not participate in the project. The active contributors to this project were Maddi Acton, James Petersen, and Christina Melnic.