COE528 Final Project

Instructor: Olivia Das
Teaching Assistant: Messiah Abolfazli Esfahani
Term: Winter 2025

Role	Name	Student ID	Email	Section	Signature
Member	Kelin Mathew Jacob	501242848	kelin.mathewjacob@torontomu.ca	8	[illegible]
Member	Dev Brahmbhatt	501235383	dev.brahmbhatt@torontomu.ca	8	[illegible]
Member	Moiz Choudhary	501247129	m1choudhary@torontomu.ca	8	[illegible]

We affirm that this project is original and is our own work.

Table of Contents

Introduction

The culminating project of the course, COE528: Object-Oriented Engineering Analysis and Design, required students to develop a JavaFX-based Bookstore Application designed to facilitate book management and customer transactions within a bookstore environment. Developed as a single-window GUI, the application switches between screens within the same window instead of opening multiple ones. The application is programmed using the State Design Pattern, allowing for seamless transitions between different states. The application supports two types of users: an Owner, who manages books and customer accounts, and Customers, who can browse the bookstore and purchase books while earning reward points. Upon exiting the application, all relevant user and bookstore data is saved, and when the application is launched again, the data is reloaded. This report details a use-case of the application and the rationale behind using the State Design Pattern.

Use-Case Description

The table below shows a use-case description that follows the format specified for COE528

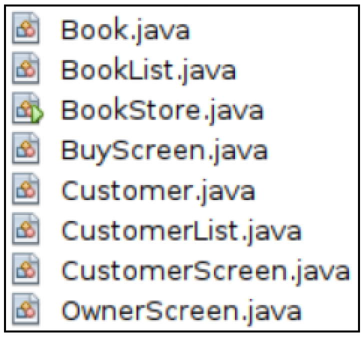
Use Case	Buying a book
Participating actors	Customer
Entry conditions	A customer must be logged in and have at least one book checked
Flow of events	1. Customer has one book selected when clicking the “Buy” button on their screen. 2. The Customer has their points changed such that they keep the same amount of points plus 10 points per dollar of the total price of the selected books. 3. The Customer will have their status updated depending on whether or not their total number of points exceeds or is equal to 1000. 4. The Customer will be shown the total price, their updated points and status values, and a logout button.
Exit conditions	The Customer is shown the total price, their updated points and status values, and a logout button. The Customer is shown an error that tells them that they did not select at least one book.
Exceptions	The Customer gets an error due to not having at least one book selected.
Special requirements	There must be at least one book available to select.

State Design Pattern Rationale

The use of the State Design pattern is a rational choice for the Bookstore Application because it simplifies managing the different states (or screens) of the application without relying on cumbersome if-else chains. Given that the application has the following distinct user states,

- Login-Screen
- Owner Screens (owner-start-screen, owner-books-screen, owner-customers-screen)
- Customer Screens (customer-start-screen, customer-cost-screen)

Using the state design pattern allows us to represent each of the above screens as separate state objects, ultimately making the transition between them easier. Our program makes use of this functionality, having separate classes for the different screens:



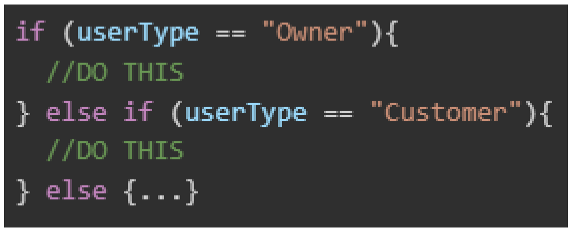
```
Book.java
BookList.java
BookStore.java
BuyScreen.java
Customer.java
CustomerList.java
CustomerScreen.java
OwnerScreen.java
```

Figure 1. Classes representing the different screens (states) in the Book Store Application

The bookstore application also changes its behavior based on whether the user is logged in as an owner or customer. For example,

- If logged in as the owner, they can add/delete books and customers
- If logged in as a customer, they can buy books and redeem points

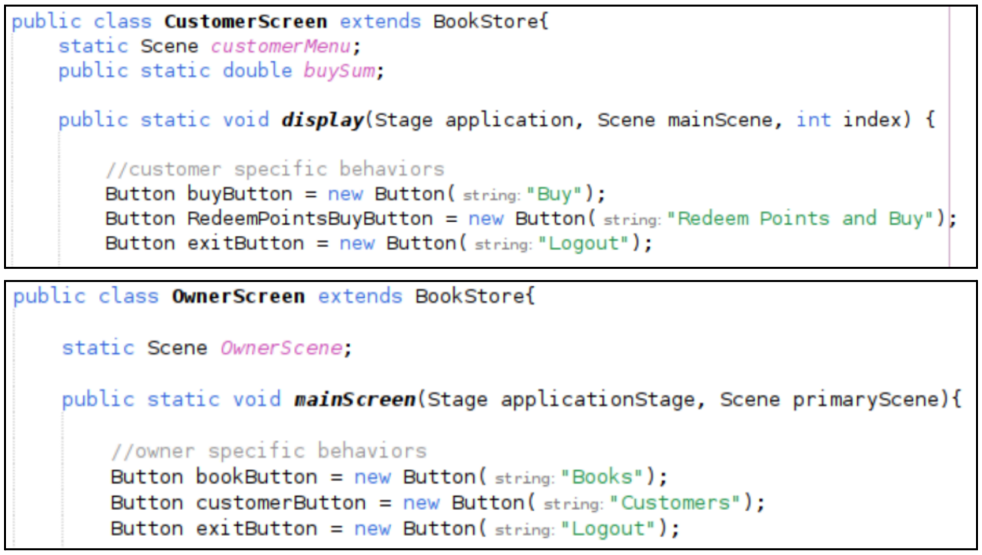
Instead of using conditional blocks, like below:



```
if (userType == "Owner"){
  //DO THIS
} else if (userType == "Customer"){
  //DO THIS
} else {...}
```

Figure 2. Behavior defined using conditional statements

The state pattern allows each state class to define its own behavior, making the code easier to follow. An example from our implementation is shown below:

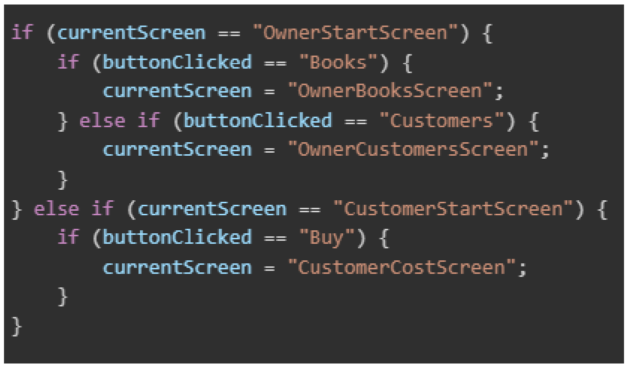


```
public class CustomerScreen extends BookStore{
    static Scene customerMenu;
    public static double buySum;

    public static void display(Stage application, Scene mainScene, int index) {

        //customer specific behaviors
        Button buyButton = new Button( string: "Buy");
        Button RedeemPointsBuyButton = new Button( string: "Redeem Points and Buy");
        Button exitButton = new Button( string: "Logout");
```

```
public class OwnerScreen extends BookStore{

    static Scene OwnerScene;

    public static void mainScreen(Stage applicationStage, Scene primaryScene){

        //owner specific behaviors
        Button bookButton = new Button( string: "Books");
        Button customerButton = new Button( string: "Customers");
        Button exitButton = new Button( string: "Logout");
```

Figure 3. Behaviors defined in their respective state classes

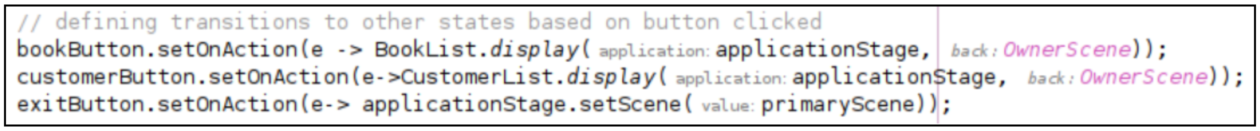
Lastly, without the state pattern, every action (clicking a button, switching screens, etc.) would require checking conditions (example shown is simplified):



```
if (currentScreen == "OwnerStartScreen") {
    if (buttonClicked == "Books") {
        currentScreen = "OwnerBooksScreen";
    } else if (buttonClicked == "Customers") {
        currentScreen = "OwnerCustomersScreen";
    }
} else if (currentScreen == "CustomerStartScreen") {
    if (buttonClicked == "Buy") {
        currentScreen = "CustomerCostScreen";
    }
}
```

Figure 4. State transitions using conditional statements

Overtime, as the application grows, the code structure shown in Figure 3 becomes poor since it will be hard to maintain. However, with the state pattern, each state handles transitions internally. For instance, the login-screen → owner-start-screen transition or the customer-start-screen → customer-cost-screen transition is handled inside the respective state class. Implementation from our program shown below:



```
// defining transitions to other states based on button clicked
bookButton.setOnAction(e -> BookList.display( application: applicationStage,  back: OwnerScene));
customerButton.setOnAction(e->CustomerList.display( application: applicationStage,  back: OwnerScene));
exitButton.setOnAction(e-> applicationStage.setScene( value: primaryScene));
```

Figure 5. OwnerScreen.java handling transitions to other states internally

Overall, the State Design Pattern is most appropriate for the Bookstore Application, allowing us to better organize the states and the transitions between them.

Conclusion

In conclusion, the Book Store application is successful in demonstrating the implementation of a JavaFX-based bookstore management system with a user-friendly, single-window GUI. By using the principles of the State Design Pattern, the application appropriately organizes and transitions between different screens (states) for both the owner and customers. In addition, the application maintains consistency between uses by saving relevant user and bookstore information. Overall, the project emphasizes key software engineering principles, including but not limited to UML and state pattern design. For future projects, the Book Store can be improved by adding support for multiple owners and adding more interactive elements within the UI to enhance the user experience.