COE528 Final Project Report

April 04, 2021

Moiz Kharodawala
Saiharan Muraleethasan
Ashreet Nagar
Atheesh Thanaseelan

Introduction

In this report, we will be discussing a use-case situation that our program has been designed to handle. Then, we will discuss our rationale for implementing the State Design Pattern in our application.

Use-case Scenario

The use-case we will describe is customerBuysBooksWithPoints.

The actors in this use-case are the customer, as well as the stock of the bookstore.

The entry condition for this use-case is that the customer must be registered with the bookstore, and that the books they want must also be in the bookstore. The files the program loads to get the customers and books must also match these conditions.

The flow of events are:

1) Customer logs into the bookstore interface
2) Customer selects the books they want to buy
3) Customer clicks “Redeem and Buy”
4) Customer is shown the cost of their purchase, as well as how many points they now have and their corresponding status.
5) Customer presses “Logout” and is logged out

The exit condition is that the customer has successfully logged out. On exit, the program will save the change made, by removing the book from the application, as well as adjusting the customer’s point balance to deduct the redeemed points.

One exception would be that the customer leaves without buying. Then, the customer would just be logged out, and their account status and the bookstore status would remain the same.
Another exception would be that the customer is not registered in the shop. In this case, they would have to speak with the store owner to have them added to the system.
Another exception would be the books not being in the shop. This would mean that the owner would need to either get the book in stock, or update the system to reflect that the book is in stock.
Another exception would be either the customer or book having invalid registrations. In this case, the store owner would need to repair the issue using the customer or book page.

A special requirement would be that the customer be able to afford the cost of the books.

Rationale for the State Design Pattern

State Design Pattern was used because it allows us to change the way the program behaves, depending on the state it is in.

When the user is unauthorized, we do not want them to be able to do anything but log in. When the user is logged in as a customer, we only want them to be able to view books, buy books, and redeem points. When the user is logged in as an admin, they will be able to view and make changes to customers and books.

To do this, the Backend class serves as the context, which holds a state class, which is the BackendState class. The three different states, UnauthorizedBackend, CustomerBackend, and AdminBackend. UnauthorizedBackend only implements the function needed to login. CustomerBackend only implements the functions needed to look at the list of books, as well as buy the books with or without points. AdminBackend implements the functions needed to list, add and remove customers and books.

If the state design pattern was not used, the complexity of the program would have increased. If a new state had to be added, such as a state for an employee, a large amount of changes would need to be made to the Backend class. With the State Design Pattern, all we would need to do is add an EmployeeBackend class, and implement any required functions.

Overall, using the State Design Pattern was helpful in making sure our program can be easily maintained and extended without having to edit pre existing classes.

Conclusion

In conclusion, our program is able to handle use-cases such as the one presented in a defined manner, and the use of Stage Design Pattern was appropriate for this application.