SOFT252 OPP with Design Patterns

COURSEWORK REPORT

Student ID: 10518778

2018/19

Link to GitHub:

https://github.com/TKCSeow/Library-Management-System

1 Introduction

The task was to create a system for a library that can manage the borrowing and returning of resources. This program was to be made using the Java coding language in Netbeans. This is a reflection of how I design the system and want to could have improved on.

2 Design

2.1 Initial Design Thoughts

Before starting any design or coding, I sat down and gave thought to what design patterns were appropriate for a library system. I had concluded that a state pattern for resources would be appropriate as a resource could have different states like, “Available” or “Borrowed” which both would need to react to the same actions as each other which the state pattern deals with nicely. The observer pattern I had thought would work between a resource and client. The idea was that a client would observer a resource, where if some changed about the resource, for example becoming overdue, it would notify the client. However, this idea was abandoned during the implementation stage due to the added complexity it added to the existing code and was applied to something more simple. The observer pattern was instead used between the newsletter and client, where the client would observe the newsletter. This work where, whenever the newsletter would be updated the client would be notified by a pop up message, but I would only pop up once.

2.1 Item Class

The “Item Class” would hold all basic information about a resource. ID and title attributes where used however I believed I could have added more. At the beginning I was unsure what to add as this class would have sub-classes of each resource type “book”, “DVD” etc, and they my hold information that was unique to them. I could at the minimum added a description attribute but ran out of time to implement it. As mentioned before, subclasses were made from this class. These were “Book”, “DVD”, “Newspaper”, “Magazine”.

Each Item Class would hold a “BorrowingInformation” class that would hold all the information about borrowing. This was to separate this information from the actual item as to make it less “messy”.

2.2 User Class

As before this would have sub-classes of “Client” and “Admin”. When it came to the attributes for this class I had only thought about the minimum needed for it to meet the requirements and work. So I ended up with only “ID”, “Password”, “First Name” and “Last Name”. I could have however added extra attributes like “phone number” or “address” which a real library would likely require.

2.3 Design Pattern Implementation

2.3.1 State Pattern

This was applied to resources which would have “Available” or “Borrowed” states. Depending on the state, the resource would react differently to the same action. So for example, in “Available” state, I book may be borrowed. However if the borrow action was used when the resource was in “Borrowed” state, it would be refused. I felt my utilisation of this pattern was as good as I would have hoped. Although it was working, I felt it amounted to basically being an “if statement”. Perhaps if I added an extra state like “Overdue” it would have helped it feel more integral to the system.

2.3.2 Observer Pattern

As stated before this pattern was moved on the Newsletter. The idea was that the client would hold a “isRead” boolean that tells the system if the user has read the newsletter, i.e logged in for the first time after the newsletter has been updated. So, when a newsletter was updated, it would notify all clients to update their “isRead” boolean. The client would be added to the observer list on creation so all clients will be updated.

2.3.3 Singleton Pattern

The Singleton pattern was applied on the Newsletter. This would allow easy access to it by both clients and admin, as the newsletter would not contain any sensitive data and would only have one instance, I felt it would be able to use this pattern.

2.4 Other

Unfortunately, I was not able to fully understand how to implement the MVC architecture in my program although I did give it a try. In the examples given in the practicals, it shows the controller listening for button presses, however this was for only one button. So I was unclear how it would work for multiple buttons and instead went the opposite direction, having the “View” directly calling the controller for any methods it needed. I am unsure if this is a correct implementation but it was better than writing all the methods directly in the “View”.

3 Conclusion

In the end I felt that my program went well, I was able to implement all the requirements to some degree while successfully implementing some design patterns. However, I feel that the design patterns feel tacked onto the design and not integral to how the system worked. Although I understood the patterns in theory, when it came to implementing it to my system it was difficult and cost me a lot of time figuring how it code them it, fortunately I was able to understand them in the end. This may be one of those times where it is hard to implement something even if you know the theory until you actually give it a try and understand all the quirks it may have by implementing in my own code.