**Homework 9 Write-Up**

Nathan Yap and Josey Vechey

August 10th, 2022

\*\* Please note we also wrote this in a .txt file so that you can view it through github)

**Part I: Explanation of the problem**

To solve this problem, we needed to construct seat objects, Row objects (ArrayLists of seats), a Theater object, a Reservation Service object, and a ReservationSystem (our main class). We would need the ReservationService to take in a Theater object and interact with the user. The main objectives were to be able to set up a theater that must have wheelchair rows and then a ReservationService would continuously take inputs from a user until the exit command “done” was entered. The user can “reserve #” of seats, and “show” the theater.

**Part II: Approach**

We considered using a comparator, but that would have involved creating multiple additional arrays based on the available seats and storing/accessing additional information in the theater and other classes.

Instead, as we were looping through the stored data to check if we should update the theater to reserve seats, it was decided this would be a possible approach to keep count as we iterate through the Theater row seats and check if we could reserve the row with a few additions including keeping track of available seats and then starting in the middle row and moving outward, because the middle row is the most desirable seat.

Essentially, a theater is an array list of rows and a row is an array list of seats. We broke the updating method up as much as possible, but keeping this approach allowed us to use iteration as our primary method for building and updating. We decided to not include the wheelchair accessible rows in this part of the algorithm, because those rows would be filled later if all the non-accessible rows were filled. As such, we built a different algorithm for the reservations of wheelchair accessible rows. Please note that for the reservation of the Non-wheelchair accessible seats, the algorithm we created is one that would reserve from the middle as the best row. However, for the wheelchair accessible, it will reserve from the first wheelchair accessible row closest to the screen and slowly move backwards. We designed it this way because we do not expect there to be more than 2 rows of wheelchair accessible seats although it can very well be possible.

This approach meant our Theater class was somewhat overextended in its functionality as the theater class was responsible for managing the many actions occurring in the program, but other classes were considerably cohesive. We also extensively tested edge cases and used information hiding as much as was appropriate. Please note however that ReservationService is the main class that controls the whole program.

**Part III: Other Decisions and Design Patterns**

To make this system function appropriately, we were not able to make a perfectly designed system. Ideally each component should be cohesive and belong/alter only the class to which it resides. However, most of our functionality required ample use of the Theater and the Reservation Service class. We kept the Seat and Row class completely isolated so that they only contained their information. The functionality for both Rows and Seats we needed was their ability to use their getters and setters for us to identify whether that seat is already taken or when we want to assign that seat to a customer. Next the ReservationSystem was kept minimal so that it only set up what was necessary, and all checks were performed in other classes.

ReservationService was isolated other than being coupled to Theater (required a completely built Theater to function). ReservationService being a public Boolean allowed us to use a while loop so that the service would continue to interact with the user until the exit command was entered. We ensured that it would read the nextline and return it in lowercase so we would only have to look for one option (a lowercase option). We also used private statements/options/question lists to keep the code concise and so that other statements could be adjusted/changed for future use. The ReservationService does rely on the theater class, but functionally the logic could be used in other designs.

Ultimately, the design pattern we used are the Singleton Pattern and the Builder Creational Design Pattern. We used the Singleton Pattern because we would first need to instantiate a Theater class and then instantiate a ReservationService class for the program to work. We though this was the most logical approach to take so that a programmer who takes over our code would be able to see how the objects are instantiated and how the ReservationService is controlling the program. Further we used the Builder Creational Design Pattern for our so that after it constructed itself it can move on to create the other classes. That is, we have the theater class constructor contain all of the information necessary for it to build itself and the other concrete classes included in our design (Row, seats) using an iterative approach to the problem. It also took on the role of checking and updating, another approach would be to separate this functionality and introduce an interface/another class. We added methods that could be used by any system such as returnAlphabet and we also used information hiding as much as possible in every class (print statements, helper methods, etc.) to improve the quality of our design. Again, aside from Theater the other classes were fairly cohesive aside from ReservationService requiring a Theater.