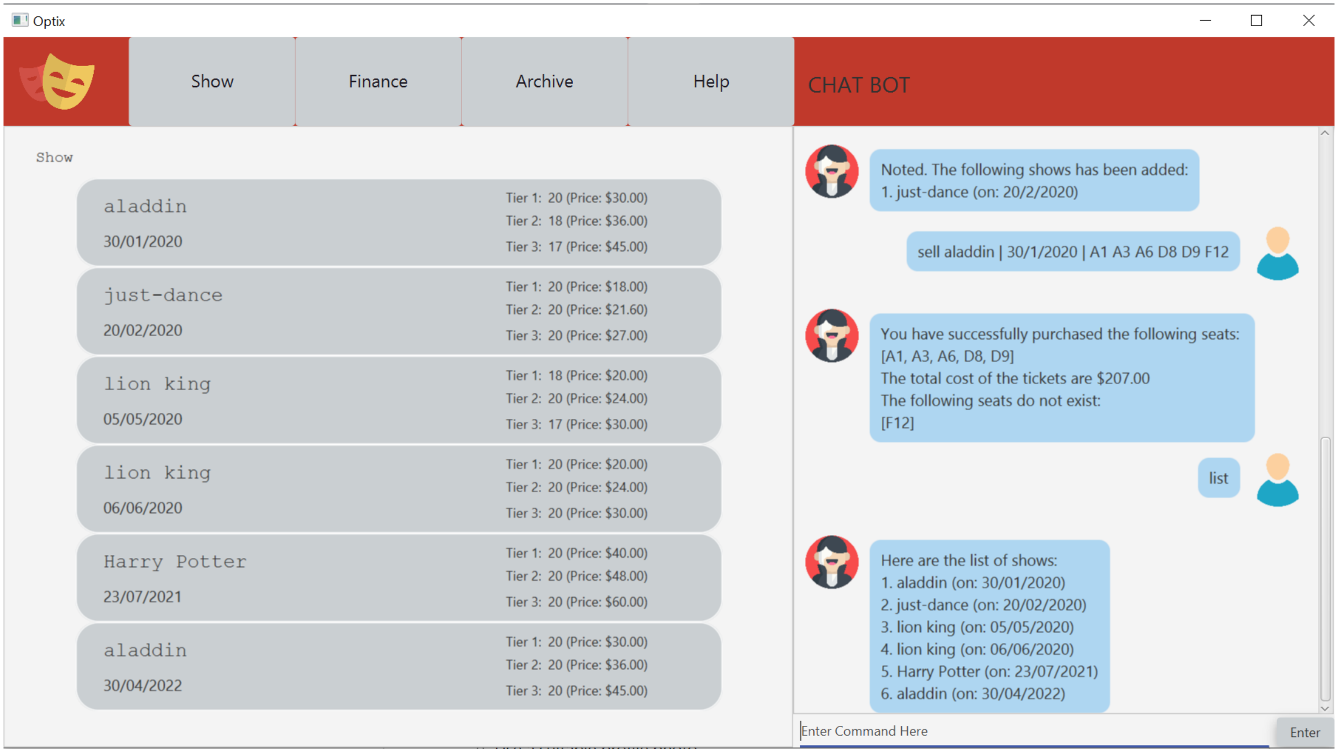
Brian Lim — Project Portfolio for Optix

About the project

My team of 3 software engineering students and I were tasked with enhancing a basic command line interface desktop task manager application for our Software Engineering project. We chose to morph it into a show and ticketing management system called Optix. This enhanced application enables theatre and concert hall managers to schedule shows, manage ticket purchase and account for their finance all within a single application.

This is what our project looks like:



aladdin	30/01/2020	Tier 1: 20 (Price: $30.00) Tier 2: 18 (Price: $36.00) Tier 3: 17 (Price: $45.00)
just-dance	20/02/2020	Tier 1: 20 (Price: $18.00) Tier 2: 20 (Price: $21.60) Tier 3: 20 (Price: $27.00)
lion king	05/05/2020	Tier 1: 18 (Price: $20.00) Tier 2: 20 (Price: $24.00) Tier 3: 17 (Price: $30.00)
lion king	06/06/2020	Tier 1: 20 (Price: $20.00) Tier 2: 20 (Price: $24.00) Tier 3: 20 (Price: $30.00)
Harry Potter	23/07/2021	Tier 1: 20 (Price: $40.00) Tier 2: 20 (Price: $48.00) Tier 3: 20 (Price: $60.00)
aladdin	30/04/2022	Tier 1: 20 (Price: $30.00) Tier 2: 20 (Price: $36.00) Tier 3: 20 (Price: $45.00)

```
Noted. The following shows has been added:
1. just-dance (on: 20/2/2020)

sell aladdin | 30/1/2020 | A1 A3 A6 D8 D9 F12

You have successfully purchased the following seats:
[A1, A3, A6, D8, D9]
The total cost of the tickets are $207.00
The following seats do not exist:
[F12]

list

Here are the list of shows:
1. aladdin (on: 30/01/2020)
2. just-dance (on: 20/02/2020)
3. lion king (on: 05/05/2020)
4. lion king (on: 06/06/2020)
5. Harry Potter (on: 23/07/2021)
6. aladdin (on: 30/04/2022)
```

Figure 1. The Graphical User Interface of Optix

My role was to design Commands and Models that manages ticket purchase and to ensure that my team adheres to the coding standards. Two prominent features that I have designed are the sell seat and view seat features. Once the main features were up, I took on the additional role of designing and implementing the Graphical User Interface for Optix. The following sections illustrate these enhancements in more detail, as well as the relevant documentation I have added to the user and developer guides in relation to these enhancements.

Summary of contributions

This section shows a summary of my coding, documentation, and other helpful contributions to the team project.

Enhancement added: I added the ability for user to sell seats and view seating arrangements.

- **What it does**: The sell command allows the user to manage ticket purchase and the view command allows user to see the seating arrangement for a show.
- **Justification**: For the application to fulfil its purpose, it is necessary for it to be able to keep track of the number of tickets that can be sold for each showing instead of just scheduling shows.
- **Highlights**: This enhancement works with existing as well as future commands. The sell feature implementation was particularly challenging because there were several implementations that we could explore but each implementation has their own challenges which will be discussed in the Contributions to the Developer Guide section.
- **Code contributed**: Please click the link to see a sample of my code: [Functional Code] [Test Code]

Other Code contributions: I helped to set up the base code for Optix (Pull request: #10) and subsequently added most of the basic command so that my group could work on the features for Optix.

Other contributions:

- Project management:
  - Ensured that coding standards were adhered to. (Pull request: #87, #96)
  - Updated and managed issue tracker by assigning issues to ensure milestones are met.
  - Structured Command Class and Test code guidelines for my team so that the code is standardised. (Pull request: #87, #97)
- Enhancements to existing features:
  - AddCommand and DeleteCommand to enable bulk process. (Pull request: #87)
    - Justification: Shows are normally scheduled for consecutive days, hence the enhancement to allow bulk process reduces the amount of work for managers to schedule the shows. Similarly, if the trope is unable to perform due to unforeseen circumstances, the manager can bulk delete the shows that cannot be performed.
  - ListShowCommand and ListDateCommand. (Pull request: #14 and #75 respectively)
    - Justification: To act as filters when shows scheduled increases.
  - Added Graphical User Interface for Optix. (Pull request: #89)
    - Justification: To make the application more user-friendly as Graphical User Interface is more interactive as compared to command line.
- Documentation:
  - Ensured that the stylings of the user guide are aligned and separated commands into various headers to make it reader-friendly.
- Tools:
  - Integrated a third-party library (JFoenix) to the project. (Pull request: #87)

Contributions to User Guide

The following section shows my contribution to the user guide.

While updating the user guide, I noticed that as more commands were added it made locating a specific command within the user guide to be extremely tedious. This made it non-user-friendly.

Below is a screenshot of the previous version of our user guide:

[cheesengg UG Old] | *../images/team/cheesengg_UG_Old.png*

Figure 2. Old User Guide Format

Since there was no structure to how the commands were listed in our user guide, I noticed that it is hard for the user to find specific commands to use. I also noticed that as the user guide was not updated constantly after every feature, it posed as a challenge for us when we were trying to update the user guide since the commands were not categorised properly.

Since each Optix's command deals with a specific component in the program, I categorised the commands to be representative of the different components that they deal with in the program.

Below is the screenshot of the changes that I made for the user guide:

[cheesengg UG New] | *../images/team/cheesengg_UG_New.png*

Figure 3. New User Guide Format

Contributions to Developer Guide

The following section shows my contribution to the developer guide.

Below details my contributions for Section 2 of developer guide.

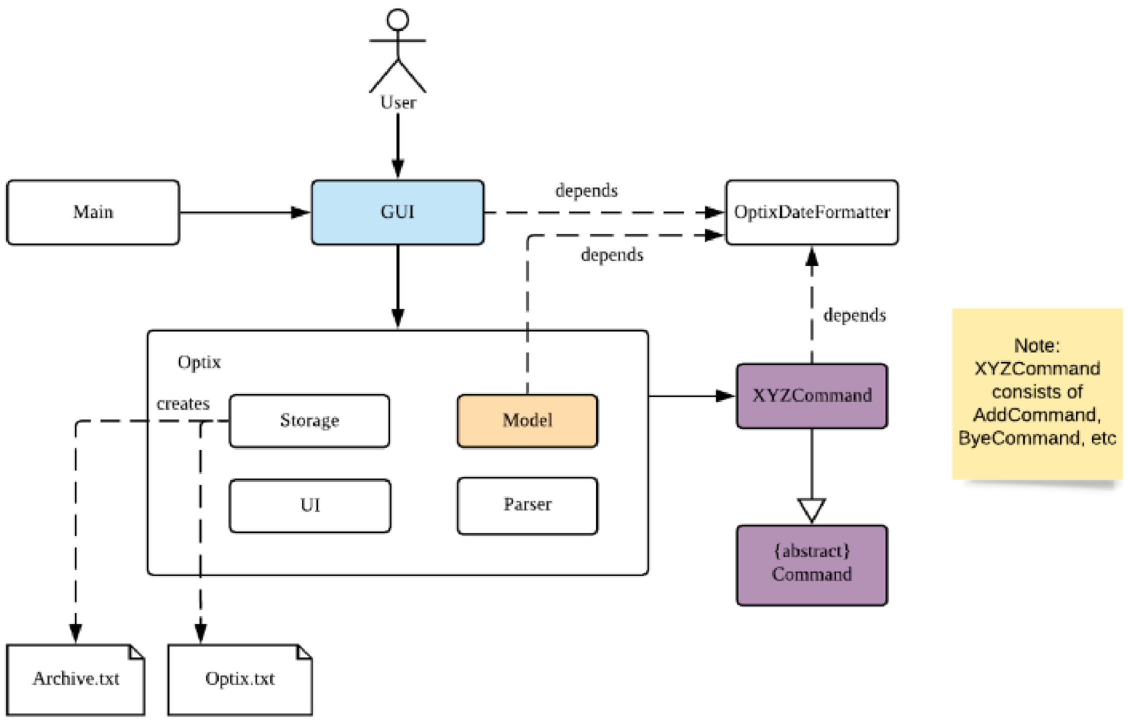
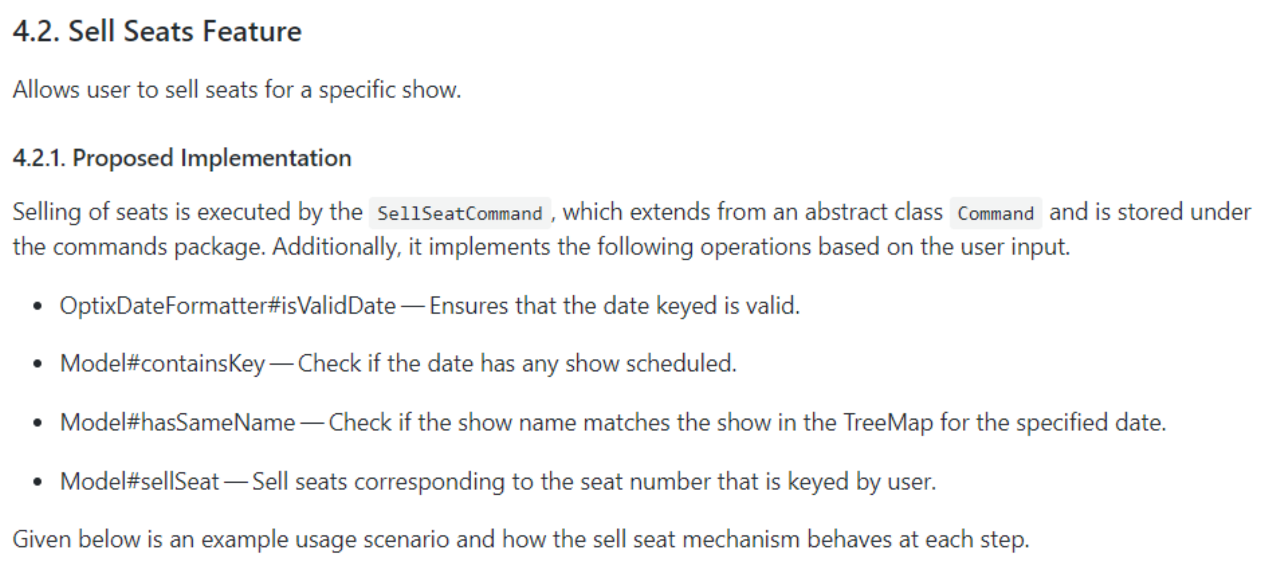


Figure 4. Architecture Diagram

I drew the System Architecture for the developer guide to show the general overview for the relationship between components in Optix.

Additionally, I color-coded the main components that will be discussed and will be drawn in subsequent subsections. This allows readers to recognise which component is being discussed in the subsection and its role in the system easily.

Next the following details my documentation of the enhancement feature for Optix, SellSeatCommand.



4.2. Sell Seats Feature

Allows user to sell seats for a specific show.

4.2.1. Proposed Implementation

Selling of seats is executed by the `SellSeatCommand`, which extends from an abstract class `Command` and is stored under the commands package. Additionally, it implements the following operations based on the user input.

- OptixDateFormatter#isValidDate — Ensures that the date keyed is valid.
- Model#containsKey — Check if the date has any show scheduled.
- Model#hasSameName — Check if the show name matches the show in the TreeMap for the specified date.
- Model#sellSeat — Sell seats corresponding to the seat number that is keyed by user.

Given below is an example usage scenario and how the sell seat mechanism behaves at each step.

Figure 5. SellSeatCommand Overview

The first part of the section states clearly the location of the feature and the relevant components to take note of so that developers reading the guide can zoom in on these components and take note of them when looking at the code.

Step 1
The user executes `sell Phantom of the Opera|5/5/2020|C1 D6 E10` command to sell the following seats C1 D6 E10 for the show Phantom of the Opera on 5th May 2020. The `SellSeatCommand` command calls `OptixDateFormatter#isValidDate(String date)` to first check if the given date is a valid date.

Step 2
Once verified, the `SellSeatCommand` command calls `Model#containsKey(LocalDate date)` and `Model#hasSameName(LocalDate date, String showName)` to check if the show in query exist within `Model` .

Step 3
Once it has been confirmed that the show exist, the `SellSeatCommand` command calls `Model#sellSeats(LocalDate date, String[] seats)` to query if the seats have been booked. Whenever a seat has been purchased successfully, the revenue obtained from the show will then be updated accordingly.

The following activity diagram summarizes what happens when a user executes sell seat command.

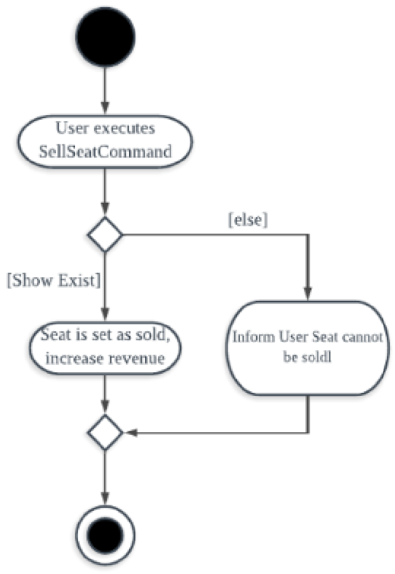
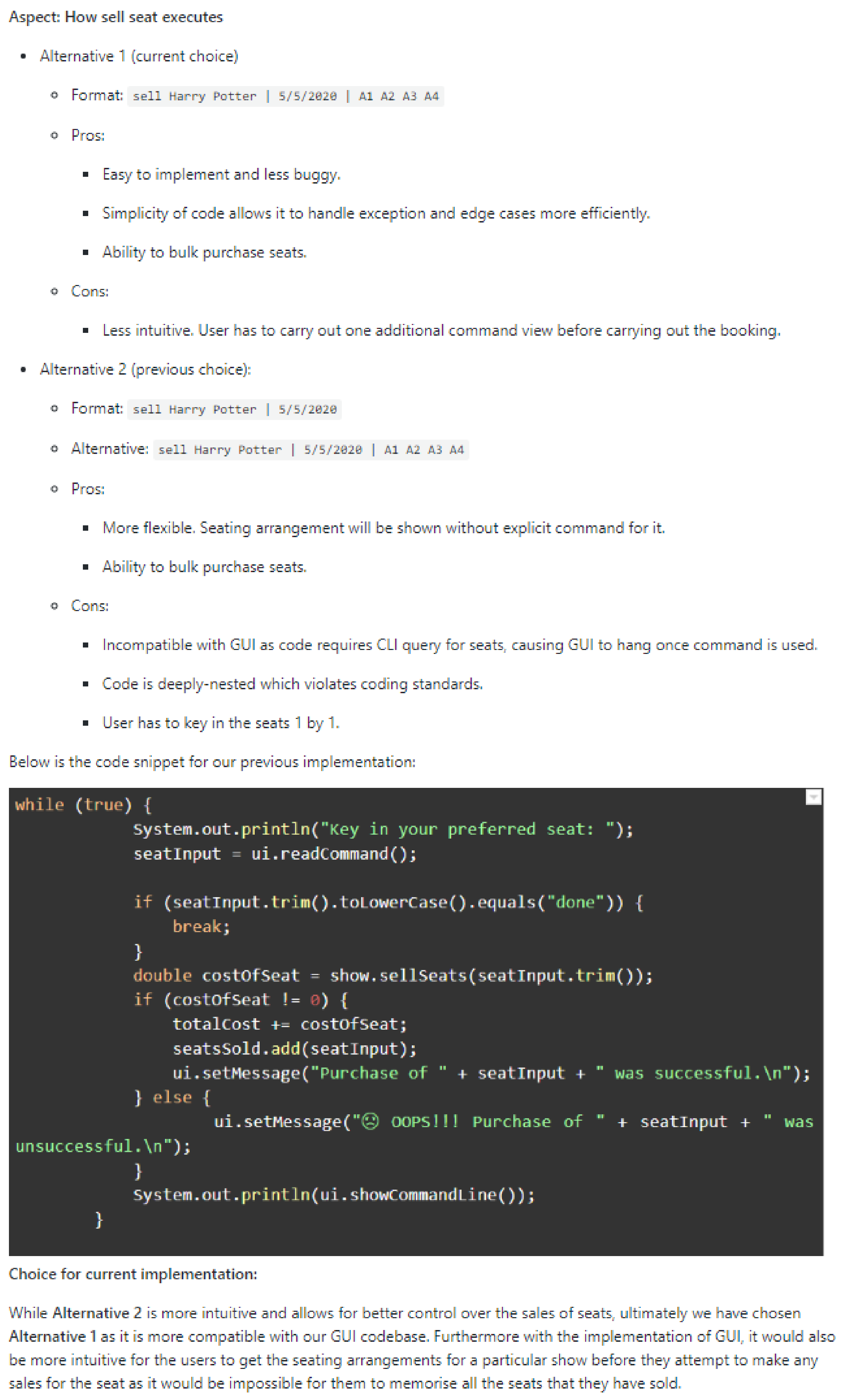

User executes SellSeatCommand
[else]
[Show Exist]
Seat is set as sold, increase revenue
Inform User Seat cannot be soldl

Figure 6. Flow of Algorithmn

Next, the flow of the feature is described in a logical manner so that when they read the code they will be able to visualise what is happening in the code. An activity diagram is then used to give a summary for the flow of the Command which enhances the developers understanding for the feature.



Aspect: How sell seat executes

- Alternative 1 (current choice)
  - Format: `sell Harry Potter | 5/5/2020 | A1 A2 A3 A4`
  - Pros:
    - Easy to implement and less buggy.
    - Simplicity of code allows it to handle exception and edge cases more efficiently.
    - Ability to bulk purchase seats.
  - Cons:
    - Less intuitive. User has to carry out one additional command view before carrying out the booking.
- Alternative 2 (previous choice):
  - Format: `sell Harry Potter | 5/5/2020`
  - Alternative: `sell Harry Potter | 5/5/2020 | A1 A2 A3 A4`
  - Pros:
    - More flexible. Seating arrangement will be shown without explicit command for it.
    - Ability to bulk purchase seats.
  - Cons:
    - Incompatible with GUI as code requires CLI query for seats, causing GUI to hang once command is used.
    - Code is deeply-nested which violates coding standards.
    - User has to key in the seats 1 by 1.

Below is the code snippet for our previous implementation:

```
while (true) {
            System.out.println("Key in your preferred seat: ");
            seatInput = ui.readCommand();

            if (seatInput.trim().toLowerCase().equals("done")) {
                break;
            }
            double costOfSeat = show.sellSeats(seatInput.trim());
            if (costOfSeat != 0) {
                totalCost += costOfSeat;
                seatsSold.add(seatInput);
                ui.setMessage("Purchase of " + seatInput + " was successful.\n");
            } else {
                    ui.setMessage("☹ OOPS!!! Purchase of " + seatInput + " was 
unsuccessful.\n");
            }
            System.out.println(ui.showCommandLine());
        }
```

Choice for current implementation:

While **Alternative 2** is more intuitive and allows for better control over the sales of seats, ultimately we have chosen **Alternative 1** as it is more compatible with our GUI codebase. Furthermore with the implementation of GUI, it would also be more intuitive for the users to get the seating arrangements for a particular show before they attempt to make any sales for the seat as it would be impossible for them to memorise all the seats that they have sold.

Figure 7. Design Considerations

Lastly, design considerations were included in the documentations, so that future developers would be able to understand the thought process for the implementations of the program as well as the various possibilities that the program could be structured. They would also be able to understand the strengths and weaknesses of the different design implementations and work on improving them. The code block at the very bottom of Figure 3 was included in the documentation so that if they would like to improve on our current design they would have a basic codebase to work with.

Conclusion

This portfolio provides a brief overview of my contributions to the code, User Guide and Developer Guide of the Optix software. Steps were taken to justify my design choices for the guide and code. Most of my contributions were towards designing a suitable model for our application such that there is high cohesion between our model and features implemented. It is designed with considerations of expanding it into a much larger projects in the future in mind.

My main takeaway for this project is that it is important to have a good domain knowledge of the problem that we want to solve. As none of us have any experience or knowledge on how ticketing is like in the real world, it was hard for us to come up with suitable data structure and model to manage the entire system and this led to constant refactoring of our model to suit the features. By having a strong understanding, we would then be able to design a model that is compatible with all the features that we have discussed in our user stories and be more efficient when coding the program.