ASSIGNMENT

CE/CZ2002: Object-Oriented Design and Programming

Building an OO Application

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING NANYANG TECHNOLOGICAL UNIVERSITY

<u>Declaration of Original Work for SC/CE/CZ2002 Assignment</u>

We hereby declare that the attached group assignment has been researched, undertaken, completed and submitted as a collective effort by the group members listed below.

We have honoured the principles of academic integrity and have upheld Student Code of Academic Conduct in the completion of this work.

We understand that if plagiarism is found in the assignment, then lower marks or no marks will be awarded for the assessed work. In addition, disciplinary actions may be taken.

Name	Course (CE2002 or CZ2002)	Lab Group	Signature/Date
Pareena Kaur D/O Harcharan Singh	CZ2002	SSP5	[illegible] 12/11/22
Kwok Ying Jie Hillary	CZ2002	SSP5	[illegible] 12/11/22
Ananthan Srinath Adhvait	CZ2002	SSP5	[illegible] 12/11/22
Arun Ezekiel S/O Richard	CZ2002	SSP5	
Rishabh Alexander John	CZ2002	SSP5	[illegible] 12/11/22

Design Consideration

1.1 Overview

This project aims to design and develop a Console-based application for Movie Booking and Listing Management. The project is designed to make online booking and purchase of movie tickets to be done easier and more efficiently for movie-goers and cinema staff.

1.2 Approach taken

We first break down the project requirements needed and get an overview of this project. The next step was to decide on the approach we will be using, which is the model-view-controller approach.

1.3 Assumptions

a. This application is for use by a single user and concurrent access is not considered
b. For demonstration purposes, three cineplexes have been created
c. The currency utilised is Singapore Dollar (SGD) and it is inclusive of Good and Services Tax (GST).
d. A simplified login system has been created for use by cinema staff only
e. Once the booking has been confirmed, the payment is always successful.
f. No interfaces made with external systems, e.g. Payment, printer, etc.
g. Discounted tickets for students and senior citizens can be purchased online without validation of their age. These will be verified at the entrance of the cinema.
h. Public holidays do not extend beyond 1 day. In the case of a multi-day holiday, the name of the holiday will be the same for each day.

OOP Concepts

2.1 Inheritance

A core concept in OOP where new classes known as subclasses are derived from existing classes known as superclasses. The subclass absorbs or inherits the attributes and behaviours of the superclass, and new capabilities are also added. This allows code to be reused which prevents code duplication and redundancy.

For example, **PriceTicket** inherits **PriceType**. **PriceType** contains the prices for different types of tickets, which are the attributes for the class, and all **PriceTicket** "is a" **PriceType**. Hence, it is appropriate to use Inheritance in this case.

2.2 Encapsulation

Encapsulation restricts access to private data, keeping them secure from unauthorised parties and preventing unwanted interference. This is achieved by bundling data with methods that operate on the data. The implementation details of the classes are also hidden from general users. This concept is applied in most of the classes in our application.

For example, the class **MovieCineplexModel** contains a hashmap, **movieList**, of each movie in which methods are needed to access and modify this hashmap. Thus, it was appropriate to bundle the necessary attributes and methods together in a single class that exposes certain methods i.e. the hashmap can only be accessed using the "get" method (**GetMovies()**) and modified using the "set" method (**setMovies()**).

2.3 Polymorphism

Polymorphism allows users to access different types of objects via the same interface, and each type can have its own implementation of the interface.

For example, we define **SelectMoviePage** to extend the abstract class **MenuPage** in which **MenuPage** aids in creating and navigating the menu for booking a movie. **SelectMoviePage** holds the attributes of the **MenuPage**, and specifically allows users to select movies and view related details. The class uses the same methods defined in **MenuPage** but has its own implementation to execute its function. For example, the **Update()** method has been implemented differently to print details of a movie on selection, but it retains the same method name previously defined in **MenuPage**.

2.4 Abstraction

Abstraction aids in reducing complexity by concealing unnecessary information such as the implementation of the code from the user, and only shows essential information such as the methods necessary to the user. This concept was utilised when our application was separated into several different sections such as Data, Model, Controller and View classes, which caters to the specific needs of different users.

OOP Solid Principles

3.1 Single Responsibility Principle (SRP)

The main idea behind SRP is that every class or module should have one purpose in the program. This allows the code implemented to be reused and makes it easier to implement or debug in the case of future changes. In the Model, View and Controller, each of them serve only as one purpose. What we have used includes:

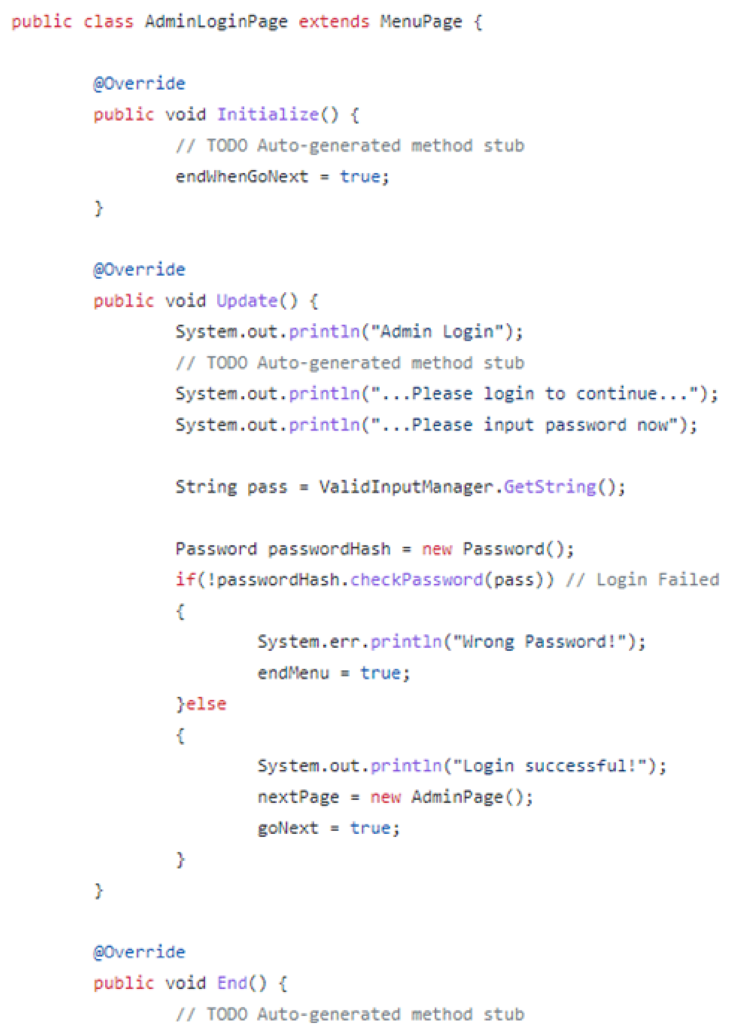
1. **Model:** Defines data structure and allows us to update the application
2. **View:** Display items for the user
3. **Controller:** Controls the logic and data
4. **ExceptionHandler:** it decides what a program will do if an anomaly happens that disrupts the process
5. **Data:** e.g. for updating and reading

3.2 Open-closed principle (OCP)

This allows for the behaviour to be extended without modifying the source code. It should allow us to add fields or elements to the data structure. For example, our Menu classes extends **MenuPage** which opens menu pages used in Menu classes for extension. However, the new Menu classes implement **MenuPage** without changing the logic.

3.3 Liskov Substitution Principle (LSP)

Liskov Substitution Principle (LSP) states that the objects of a superclass should be substitutable with the objects of its subclasses. This means that the objects of the subclasses will act in a similar way as the objects of the superclass that we have used. This will be useful to us because our code will be more easily modified if there is a new code added to it. In our code, we made a parent class called menupage and then “extends” it to various other classes such as AdminLoginPage, AdminPage and ConfigureTicketPrices. Furthermore, we used the same argument functions as menupage which is Initialise( ), Update( ) and End( ).



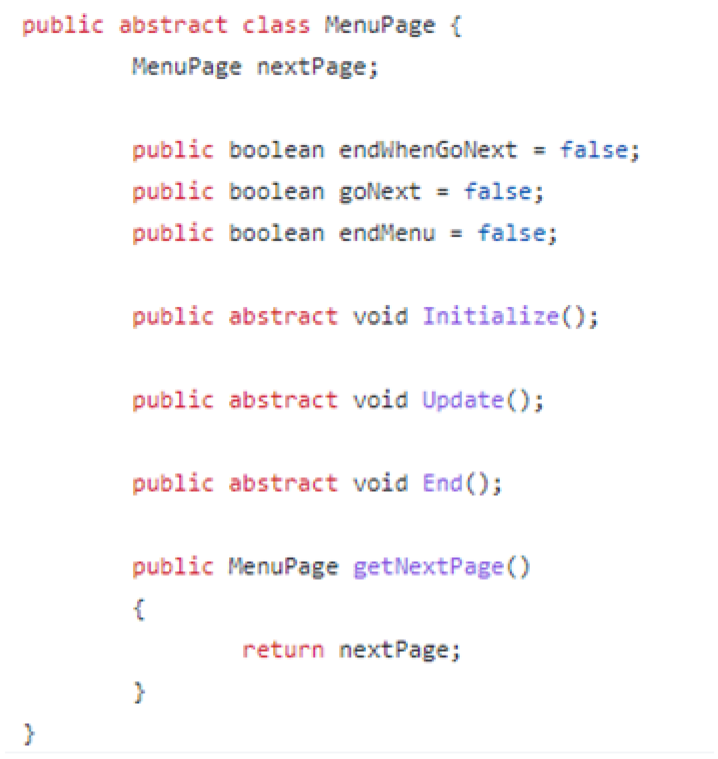
```java
public class AdminLoginPage extends MenuPage {

	@Override
	public void Initialize() {
		// TODO Auto-generated method stub
		endWhenGoNext = true;
	}

	@Override
	public void Update() {
		System.out.println("Admin Login");
		// TODO Auto-generated method stub
		System.out.println("...Please login to continue...");
		System.out.println("...Please input password now");

		String pass = ValidInputManager.GetString();

		Password passwordHash = new Password();
		if(!passwordHash.checkPassword(pass)) // Login Failed
		{
			System.err.println("Wrong Password!");
			endMenu = true;
		}else
		{
			System.out.println("Login successful!");
			nextPage = new AdminPage();
			goNext = true;
		}
	}

	@Override
	public void End() {
		// TODO Auto-generated method stub
```

3.4 Interface Segregation Principle (ISP)

ISP states that the client should not be implementing an interface or methods it does not use. This helps to reduce the negative effects of large interfaces by breaking them into smaller ones and making the interface more effective. For example, in our code we implemented **MovieCineplexModel** as a primary model for storing movie and cineplex data. It enables cinema staff to access and make simple modifications to movies and cineplexes stored in HashMap List, such as addition or deletion of a particular movie or cineplex from the list. However, the detailed modifications are done in their respective interface classes such as **CineplexIO** and **MovieIO** using methods that are relevant and specific to the modifications in each case.

3.5 Dependency Injection Principle (DIP)

DIP states that high-level modules should be easily reusable and will not be affected by changes in low-level modules. The goal is to have the high-level module and the low-level module to depend on the same abstraction. For **MenuPage**, we made it an abstract class and then declared different abstract functions inside that will be inherited by other classes that use the **MenuPage**.



```
public abstract class MenuPage {
        MenuPage nextPage;

        public boolean endWhenGoNext = false;
        public boolean goNext = false;
        public boolean endMenu = false;

        public abstract void Initialize();

        public abstract void Update();

        public abstract void End();

        public MenuPage getNextPage()
        {
                return nextPage;
        }
}
```

4. Future enhancements

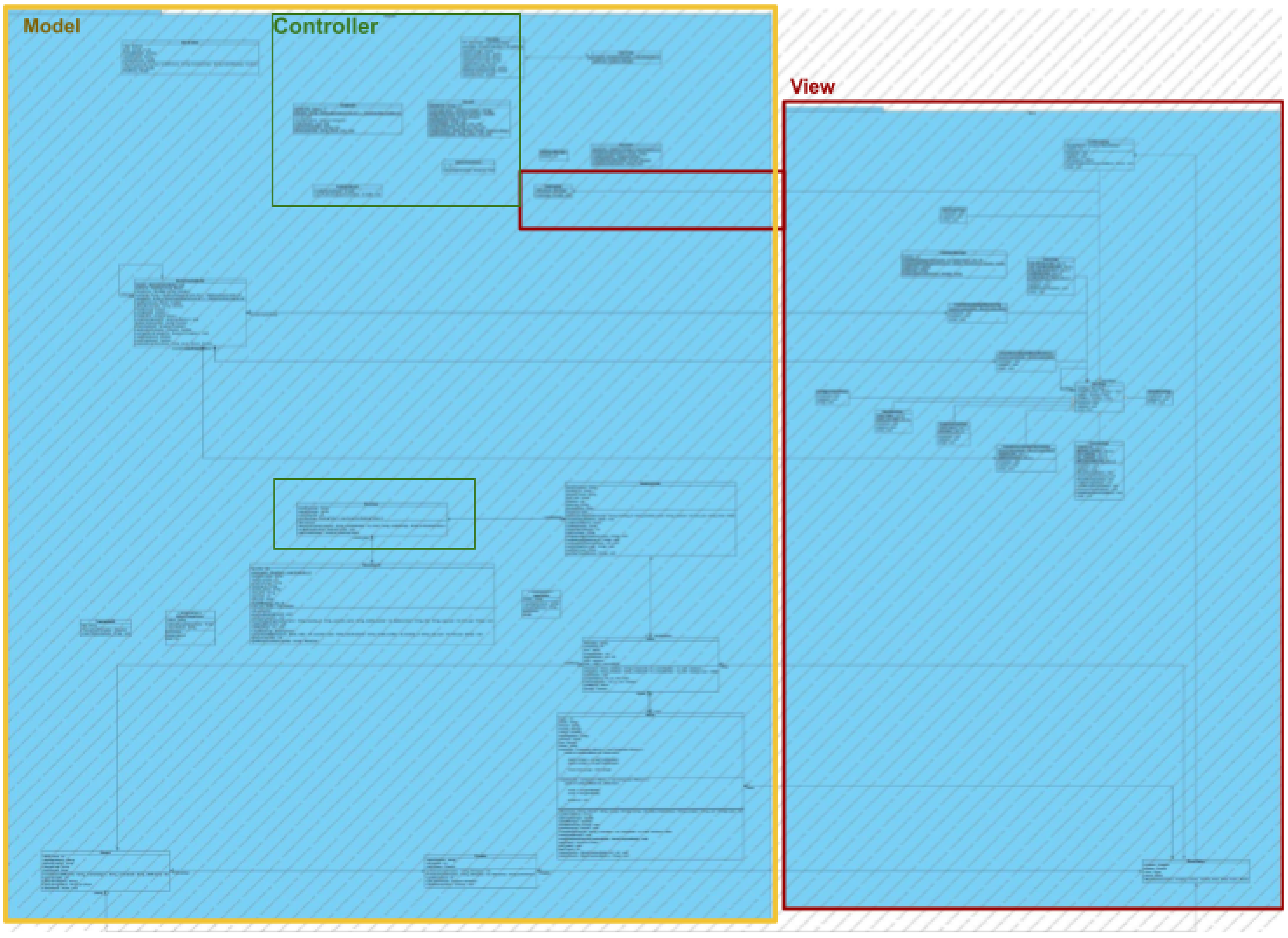
4.1 Reminder system

A reminder system can be created to inform customers of their bookings. This will make it more convenient and accessible for the users . This can be done by creating a **Reminder** class in which it will retrieve booking details made by customers and check for upcoming bookings. The booking details contain the customer's email as well as the showtime for the movie booked that were written to a data file upon completion of a booking. This has already been implemented in the **BookingTicket** class. Hence, the **Reminder** class can deliver an email to customers to remind them of their booking. Furthermore, as all our classes follow the Single Responsibility Principle, this implementation will not affect our **BookingTicket** class or other related classes that deal with movie bookings made by customers.

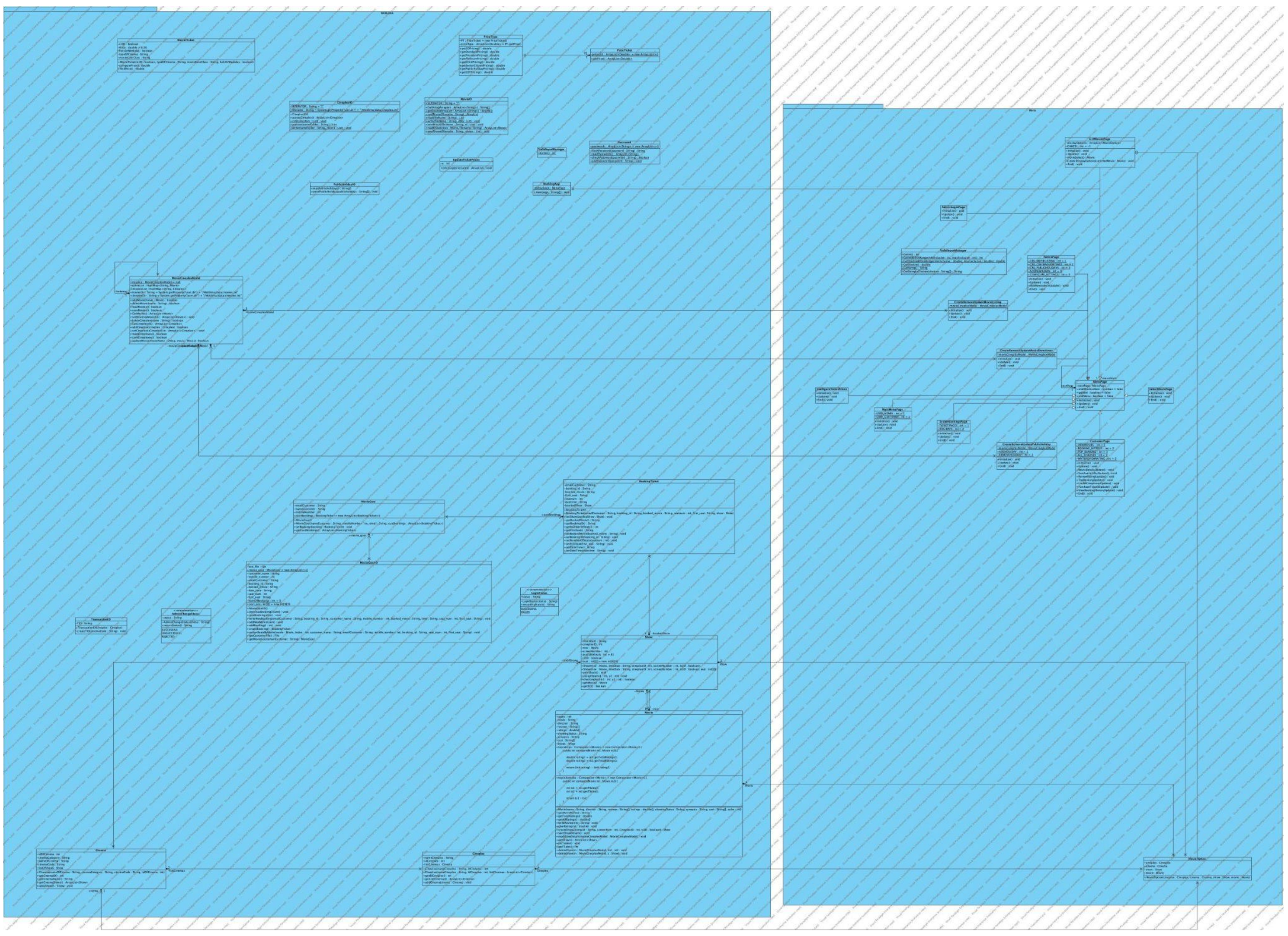
4.2 Membership Pricing

As of now, we only have prices catered for different age groups and show types. We can also implement a membership program that provides a discount based on the number of movies that have been booked by a customer, given that the customer is a member. This would involve creating a **Member** class that stores a list of customers who are a member, and adding an additional method to **PriceType** that specifies the fixed amount or percentage of the overall price that is to be discounted. However, modifications have to be made to the implementation of **PriceTicket** as there is a need to confirm that a customer is a member first before discounting the price. This would involve retrieving the list of bookings made to find the customer id and compare against the membership list to determine if the price should be discounted. Regardless, most of the required classes can be reused and only an extension is needed to be added to facilitate this new implementation.

UML Diagram (With Model-View-Controller Design Pattern)



UML Diagram



Reflection

This project has taught us the importance and usefulness of using Object-Oriented Programming (OOP) design in the real world. As we do not have much prior experience aside from the lab exercises we have done in OOP, we were unsure about how we could implement such a large scale application. Initially, we wanted to split the application into just three areas: movie-goers, cineplexes, movies. However, we realised there were many functions to consider and simply forcing all related functions into one class would be inefficient and would not work. We then spent time doing in depth research, and proceeded to apply the OOP design principles to determine an appropriate method to demonstrate our code clearly. Applying the Single Responsibility Principle, we found the Model-View-Controller method to be the most appropriate. We then used a similar process to brainstorm specialised classes for each element and distributed the work accordingly.

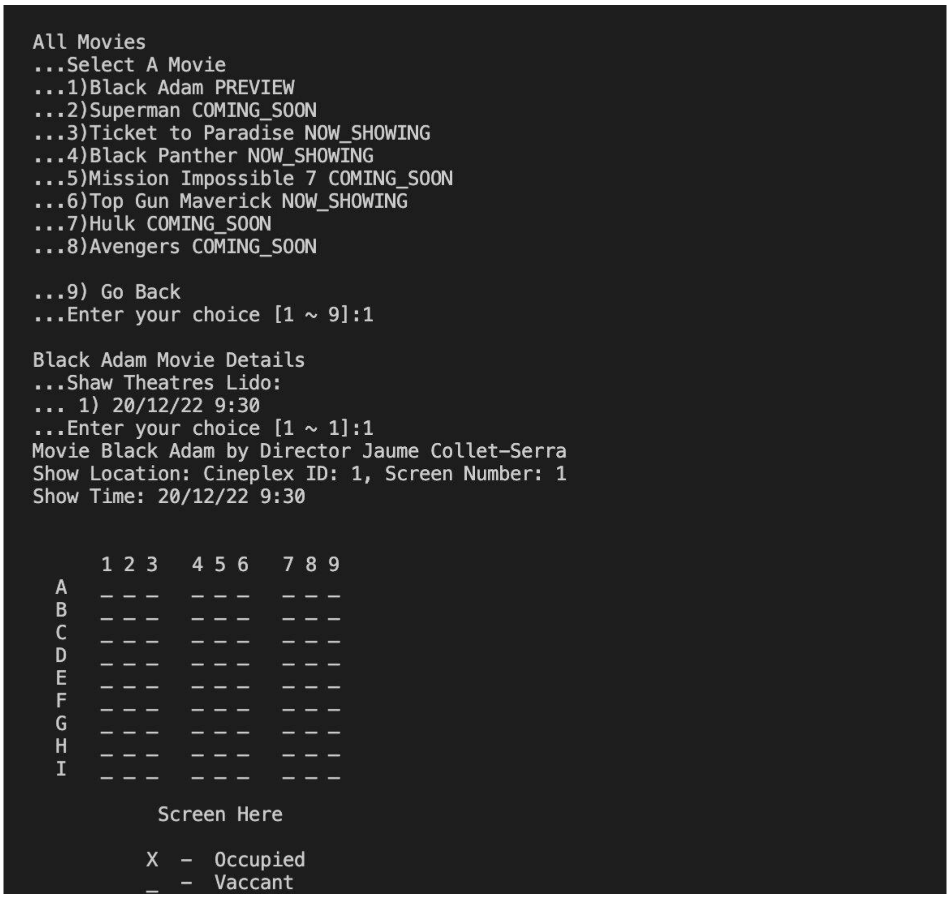
Over the course of our project, we faced numerous challenges. In the beginning stages, we faced repeated errors in combining various data types into a single array and attempting to use methods to operate on them. Additionally, we wanted to use HashMaps to speed up the efficiency of our application but we were unfamiliar with its implementation and uses, thus we faced trouble implementing them successfully. We had to be patient and resilient as we made constant attempts to tackle these various problems by doing further research and communicating our ideas with each other. Given the way we distributed the work, we also ran into many formatting errors when we combined our classes together and attempted to run the application. This was due to the inconsistency in the package names, classes names as well as variables used which we needed to modify repeatedly. It was through overcoming these challenges that we learnt the importance of doing more comprehensive planning beforehand and working together as a team to deal with problems efficiently.

Regardless, we believe that this was an enriching experience that allowed us to gain a deeper understanding of the OOP concepts and principles that were taught in lectures, and how they can effectively be applied to develop scalable, easy to read and maintainable code design.

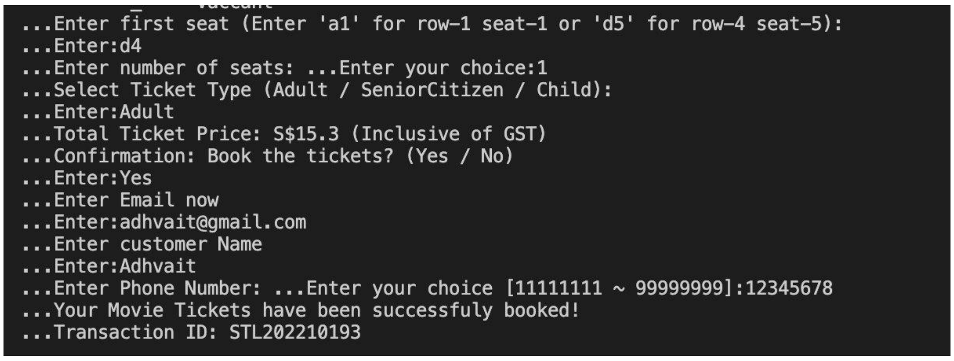
Appendix

Additional Test Cases

Booking Ticket for Movie listed as “PREVIEW”



```
All Movies
...Select A Movie
...1)Black Adam PREVIEW
...2)Superman COMING_SOON
...3)Ticket to Paradise NOW_SHOWING
...4)Black Panther NOW_SHOWING
...5)Mission Impossible 7 COMING_SOON
...6)Top Gun Maverick NOW_SHOWING
...7)Hulk COMING_SOON
...8)Avengers COMING_SOON

...9) Go Back
...Enter your choice [1 ~ 9]:1

Black Adam Movie Details
...Shaw Theatres Lido:
... 1) 20/12/22 9:30
...Enter your choice [1 ~ 1]:1
Movie Black Adam by Director Jaume Collet-Serra
Show Location: Cineplex ID: 1, Screen Number: 1
Show Time: 20/12/22 9:30


      1 2 3   4 5 6   7 8 9
  A   _ _ _   _ _ _   _ _ _
  B   _ _ _   _ _ _   _ _ _
  C   _ _ _   _ _ _   _ _ _
  D   _ _ _   _ _ _   _ _ _
  E   _ _ _   _ _ _   _ _ _
  F   _ _ _   _ _ _   _ _ _
  G   _ _ _   _ _ _   _ _ _
  H   _ _ _   _ _ _   _ _ _
  I   _ _ _   _ _ _   _ _ _

           Screen Here

          X  -  Occupied
          _  -  Vaccant
```



```
...Enter first seat (Enter 'a1' for row-1 seat-1 or 'd5' for row-4 seat-5):
...Enter:d4
...Enter number of seats: ...Enter your choice:1
...Select Ticket Type (Adult / SeniorCitizen / Child):
...Enter:Adult
...Total Ticket Price: S$15.3 (Inclusive of GST)
...Confirmation: Book the tickets? (Yes / No)
...Enter:Yes
...Enter Email now
...Enter:adhvait@gmail.com
...Enter customer Name
...Enter:Adhvait
...Enter Phone Number: ...Enter your choice [11111111 ~ 99999999]:12345678
...Your Movie Tickets have been successfuly booked!
...Transaction ID: STL202210193
```