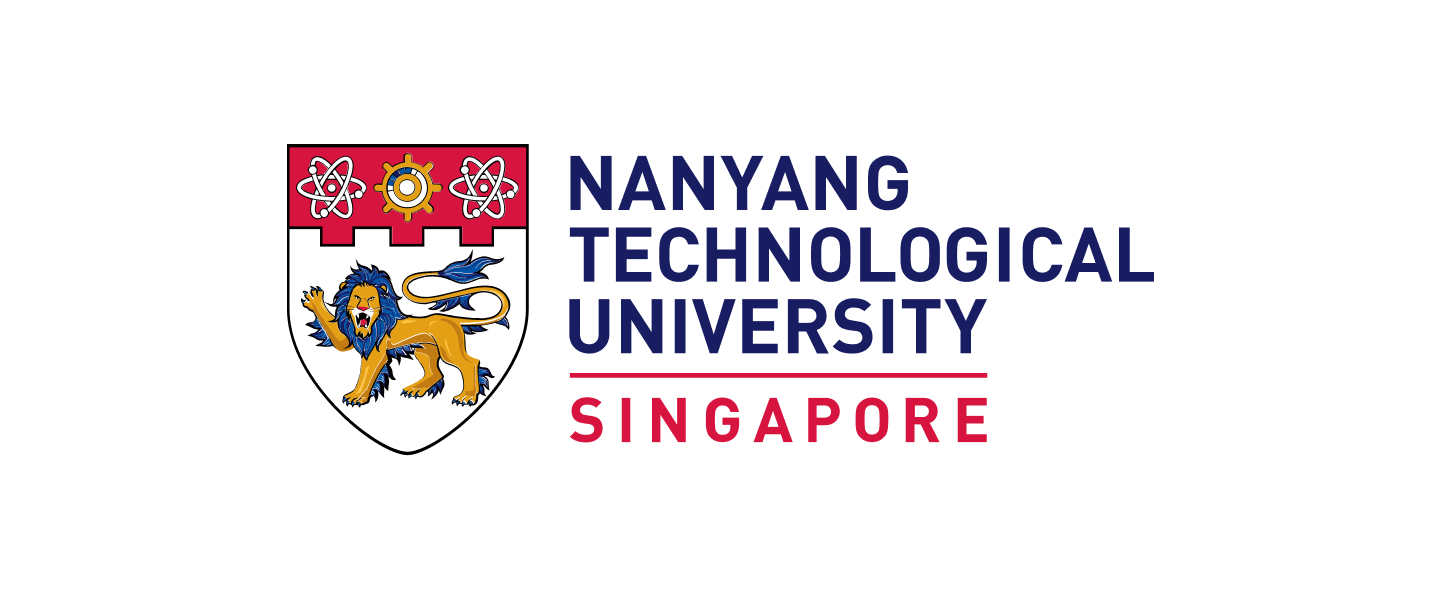
**NANYANG TECHNOLOGICAL UNIVERSITY**



**ASSIGNMENT**

**SC/CE/CZ2002: Object-Oriented Design & Programming**

***MO****vie* ***B****ooking and* ***LI****sting* ***M****anagement* ***A****pplication (MOBLIMA)*

**LAB: SS9**

**GROUP: 3**

|  |  |
| --- | --- |
| **Group Members** | **Matriculation Number** |
| Jabez Ng Yong Xin | U2120757D |
| Leong Jun Yi | U2120828J |
| Ler Lian Ping | U2122530F |
| Lim Jin Feng, Alexus | U2121689H |
| Lim Zheng Yong Shawn | U2121495D |

**TABLE OF CONTENTS**

[**Report Overview**](#_38n9inhrigzo) **4**

[**Design Considerations**](#_dlryt0f7o1qi) **4**

[**Design Principles**](#_miyzr4nk8bg1) **4**

1. [Single Responsibility Principle (SRP)](#_9fv1svwvb0b8) 4
2. [Open-Closed Principle (OCP)](#_mfja1pbrz12e) 5
3. [Liskov Substitution Principle (LSP)](#_mjwefp8nz5ha) 5
4. [Interface Segregation Principle (ISP)](#_fhiw1iq7e3u9) 5
5. [Dependency Injection Principle (DIP)](#_obnt78iz6jpc) 5

[**Assumptions**](#_29g0hnp6zb7h) **6**

[**Use Of OODP Concepts**](#_ae0947ogot2w) **7**

1. [Abstraction](#_r6rc6nsnbl8p) 7
2. [Encapsulation and Information Hiding](#_lzubhigrs7yg) 7
3. [Inheritance](#_f61q5k1hpd7p) 8
4. [Polymorphism](#_wd18znjo3115) 9

[**Proposal for Possible New Features**](#_b68qaihii383) **9**

[**UML Class Diagram**](#_6m1d54ivx42s) **10**

[**Test Cases (not shown in video)**](#_3n518324g0r) **14**

**Declaration of Original Work for SC/CE/CZ2002 Assignment**

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 honored the principles of academic integrity and have upheld the 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** |
| Jabez Ng Yong Xin | **SC2002** | **SS9** |  |
| Leong Jun Yi | **SC2002** | **SS9** |  |
| Ler Lian Ping | **SC2002** | **SS9** |  |
| Lim Jin Feng, Alexus | **SC2002** | **SS9** |  |
| Lim Zheng Yong Shawn | **SC2002** | **SS9** |  |

Important notes:

1. Name must **EXACTLY MATCH** the one printed on your Matriculation Card

**We have uploaded our video presentation on YouTube.**  
Link to our presentation: https://www.youtube.com/watch?v=M5aEbe4fcL8

# Report Overview

This report outlines the design process when developing the MOBLIMA application. By utilising design principles and concepts on Object-Oriented Programming in our considerations, we first designed a UML class diagram using Visual Paradigm’s Online Collaboratory Drawing tool before we developed the Cinema Operator Application using Java on VSCode and IntelliJ IDEs whilst collaborating on GitHub. This application manages the user and database needs of a fictional Cinema Chain, MOBLIMA.

# Design Considerations

Our group’s objective was to build an application that is easy to maintain and improve on in the future. We first divided the classes into 3 main package folders namely, *controller*, *model* and *view* based on their general functionalities. *Controller* contains the classes that contain the necessary logic implementation for user actions. M*odel* contains the entity classes and *view* contains the classes crucial for user interface displays such as the movie booking menu.

# Design Principles

We will elaborate on how the design of our application achieves high cohesion and low coupling based on the 5 design principles learned.

## Single Responsibility Principle (SRP)

Classes are designed to perform a single role, without mixing responsibilities. Therefore there should never be more than one reason for a class to change. This thus allows us to attain high cohesion within a class. For instance, when the user intends to view movie details, the user would first need to log in using the *MovieGoerMenu* class before selecting the movie to view. The *MovieMenu* class is then used to view movie details. After accessing *MovieMenu*, *MovieController* will be accessed to display the movie details for the selected movie through the *printMovieDetails* method. In this example, it can be seen that each class (menus and controllers in our case) fulfills its own responsibilities. *MovieMenu* does not need to know how to handle the login and the displaying of movie details as it is *MovieGoerMenu’s* and *MovieController’s* responsibility.

## Open-Closed Principle (OCP)

OCP underscores the importance that a module should be open for extension but closed for modification. Abstraction allows us to achieve this. For example, menu classes implement the *Menu* interface. The *Menu* interface functions as a layer of abstraction, allowing for a convenient extension if we were to add new menu classes such as a Food and Drinks menu. This removes the need to directly modify one general *Menu* class which would result in a large ripple effect where affected classes have to be modified.

## Liskov Substitution Principle (LSP)

LSP highlights that subtypes must be substitutable for base types. In the model package, certain model entity classes such as *Movie*, *Cinema* and *Cineplex* implement the *ItemTag* interface which is the generalisation for these classes. *Movie*, *Cinema* and *Cineplex* classes are specialised and have their own additional responsibility-specific functions. While they all utilise the *getTag* method from the *ItemTag* interface, they vary in their method implementation. This follows the notion of LSP with the subtypes expecting no more from the base type and providing no less.

## Interface Segregation Principle (ISP)

ISP emphasises that client-specific interfaces are better than a single general-purpose interface. In essence, no class should be forced to depend on methods that it does not use. This principle is demonstrated with the use of the *ShowTimeBooking* interface which is implemented by the *ShowTime* class. This interface allows the *BookingController* class to utilise only the required methods, *getLayout*, *getSeatAvailabilities* and *checkAvail* without the need to import *ShowTime* to access those methods*.*

## Dependency Injection Principle (DIP)

DIP states that high and low-level modules should depend on abstractions rather than one another. Details should depend upon abstractions but abstractions should not depend on details. Referencing the menu classes, the *menu* interface functions as a layer of abstraction between a high-level module like *MovieGoerMenu* and a low-level module like *movieGoer*. Both of these classes are dependent on abstraction but not the other way around. This removes the danger of dependencies between high and low-level modules. As such, modifications to low-level classes will not affect the high-level classes.

# Assumptions

For our project we followed these assumptions provided in the Assignment PDF:

(1) This is a single-user application and there is no need to consider concurrent access.

(2) THREE cineplexes will be created for the demonstration.

(3) The currency will be in Singapore Dollars (SGD) and inclusive of Good and Services Tax

(GST).

(4) A simple login for cinema staff is sufficient.

(5) Payment will always be successful.

(6) There is no need to interface with an external system, e.g. Payment, printer, etc., but you can

consider it in your design.

(7) Senior citizens can be purchased online without validation of identity or age. The

validation will be done upon entering the cinema.

For (1), we used our Driver class to push and pop menus from a Stack of Menus, with the recently-called menus placed at the top of the stack and the older menus at the bottom. This allows the program to efficiently navigate between the different menus of the program in the application.

For (2), we initialised 3 Cineplexes with 3-4 Cinemas each.

For (3), the pricing of each booking was done in our BookingMenu class and we multiplied the total price by 1.07 to include GST.

For (4), we made a login method where the input username and password strings are compared with the actual usernames and passwords in the user objects. A successful login implies that the provided usernames and passwords were correct.

For (5), we ensured that payment is always successful.

For (6), we encapsulated all of our input/output methods under the IO class. This helps to make the coding process simpler and cleaner as fewer lines of code are needed. Doing this also allowed for reusability as many functions would require the same lines of code to fulfill the same purpose, such as reading a String input or prompting the user for a “Yes” or “No” response.

For (7), in addition to Senior Citizens, we made it such that no validation is required for all age groups (children and adults). All identity and age validation would be done in person at the cinemas for the moviegoers’ convenience.

# Use Of OODP Concepts

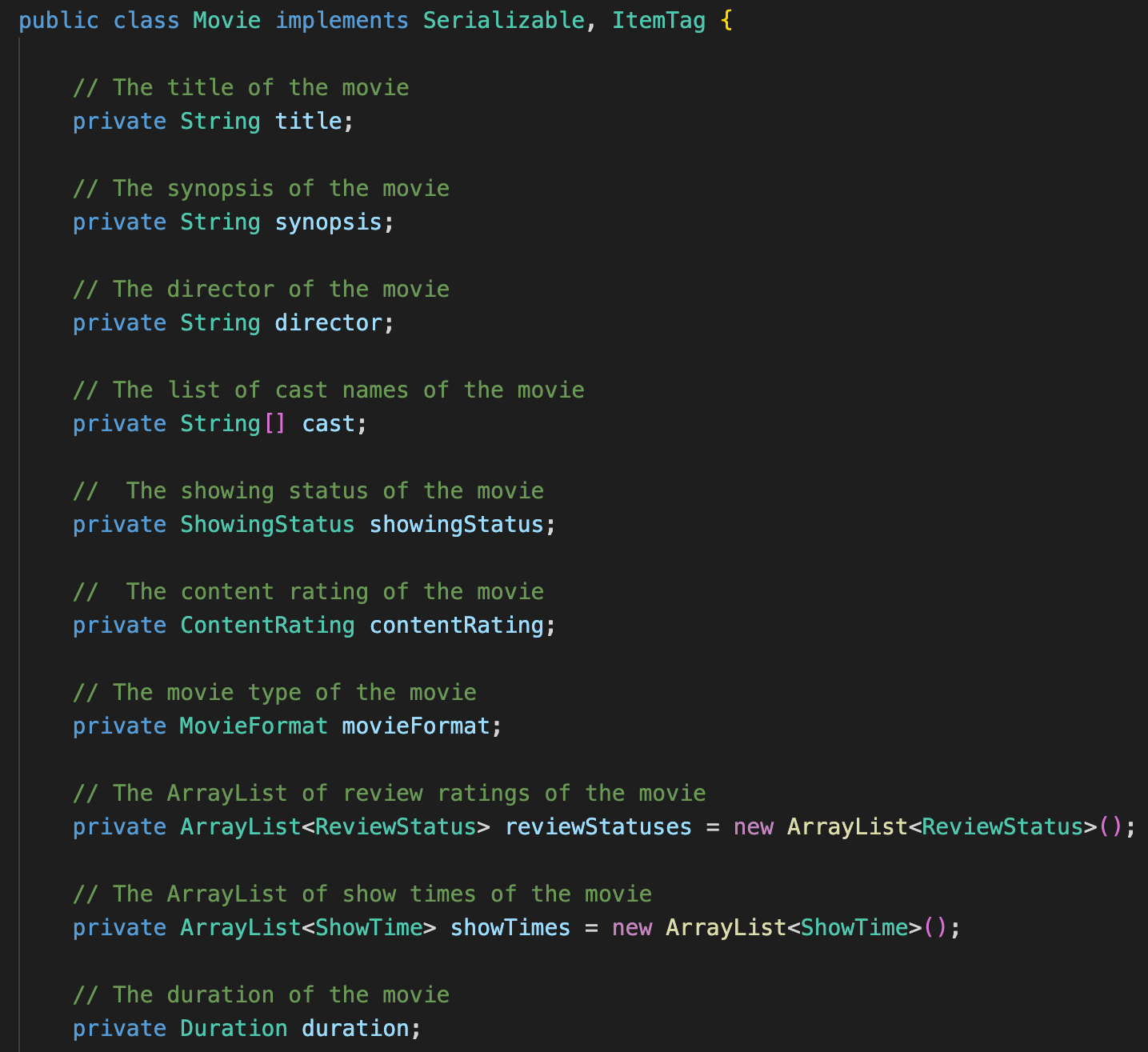
The 4 Object-Oriented Programming Concepts learned in this module are Abstraction, Encapsulation, Inheritance and Polymorphism. We will now explain the use of these concepts in our program design.

## Abstraction

Abstraction is defined as distinguishing objects based on their different essential characteristics. We are then able to provide well-defined conceptual boundaries, relative to the perspective of the viewer of our code. We performed abstraction by creating an interface *menu* which is implemented by other classes such as *RateAppMenu*, *MovieGoerMenu* and *ShowTimeMenu*. A user who intends on utilising the method *execute* will not know the information regarding the implementation of *execute* in the class. This hides the complexity of the various implementations, achieving 100% abstraction.

## Encapsulation and Information Hiding

Encapsulation hides and protects an object’s private data. Access to private data is done via the class’ public getter methods. Such data can also be modified using the public setter methods. To attain robust data security, we made the instance variables of all our classes private. Taking our *Movie* class, for example, the *title*, *synopsis*, *director* and other information about the movie are declared as private:



This prevents modification unless performed through the proper public getter and setter methods for these variables.

Information hiding conceals the details and implementation of the class from users. It lets users call methods without knowing how it exactly works. Our group performed extensive information hiding for our input/output methods under our *IO* class. Instead of having to print a message that prompts the user for input, create a Scanner object and then scan the input into a variable, we are able to do all that work in a single simple *readInt(message)* call, where it prints the message and scans for an integer input. Users can call this method without needing to know how it exactly works. Doing this allows for reusability as many methods require similar functions, such as reading a String input or prompting the user for a “Yes” or “No” response.

## Inheritance

Inheritance allows a new class to inherit the properties and behaviors of a parent class. Inheritance also allows for the reusability of code, where subclasses can reuse code from their superclasses. In our program, we have a *Menu* interface class that consists of a *execute* method. This interface is then implemented by other menus used throughout the program and the *execute* method is also inherited by them. This *execute* method can then be reused by redefining it to fulfill specific functions per each menu’s requirements.

## Polymorphism

Polymorphism refers to the ability to take more than one form and an operation may exhibit different behaviors under different instances. Polymorphism is used to augment the use of inheritance and is executed through method overriding in this case. Interfaces are also necessary to implement polymorphism. Many of our classes implement the *ItemTag* interface, which is used for objects with tags attached to them. Examples include our *Cinema*, *CinemaClass* and *Cineplex* classes. Each of these classes shares a *getTag* method that returns the tag attached to that object.

# Proposal for Possible New Features

1. **Food and Drinks Menu**

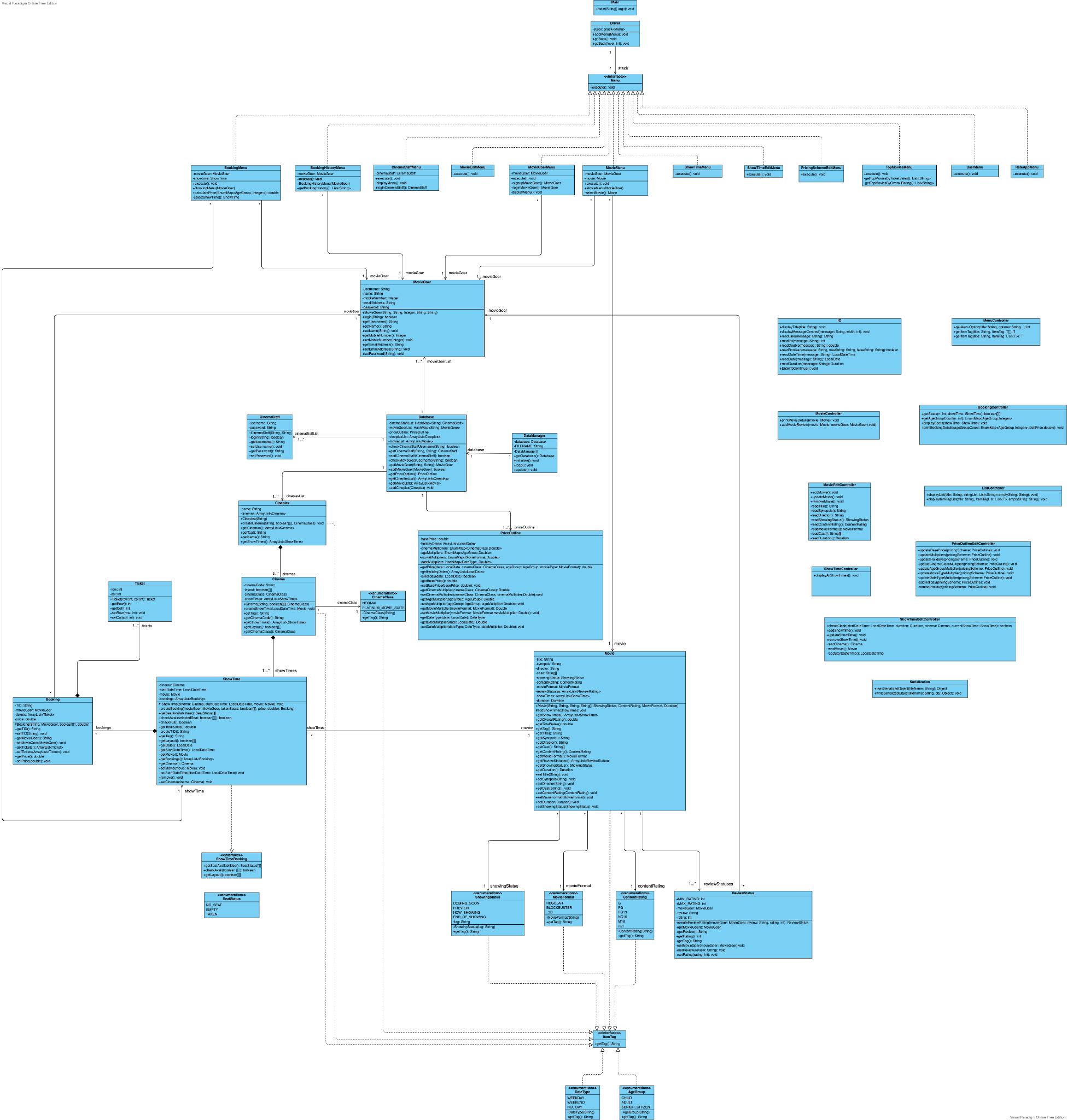
As mentioned previously under the Open-Close Principle section, our group had the idea of creating new Menu classes such as a Food and Drinks menu. Being able to buy Food and Drinks alongside Movie Tickets during the Booking Menu is more convenient for moviegoers as opposed to them having to queue up and buy them when they arrive at the cinemas on the actual day of the movie. Adding this menu would not be difficult to implement as the menu just has to implement the *Menu* interface. Moreover, changes to other classes will not be required.

1. **Menu navigation**

Another possible addition is to implement a method that allows the user to navigate to different menus, specifically to be able to navigate back to menus at any point in the program execution. Currently, we are only able to navigate back via the *Exit* option when the options page is displayed. Because *Menu* is an interface that can be implemented, the added *navigateBack* method can be implemented in the respective menus that require this navigation function. Due to Dependency-Injection-Principle, high-level specialised menu classes and low-level model classes will not be greatly affected by this addition of a new interface method.

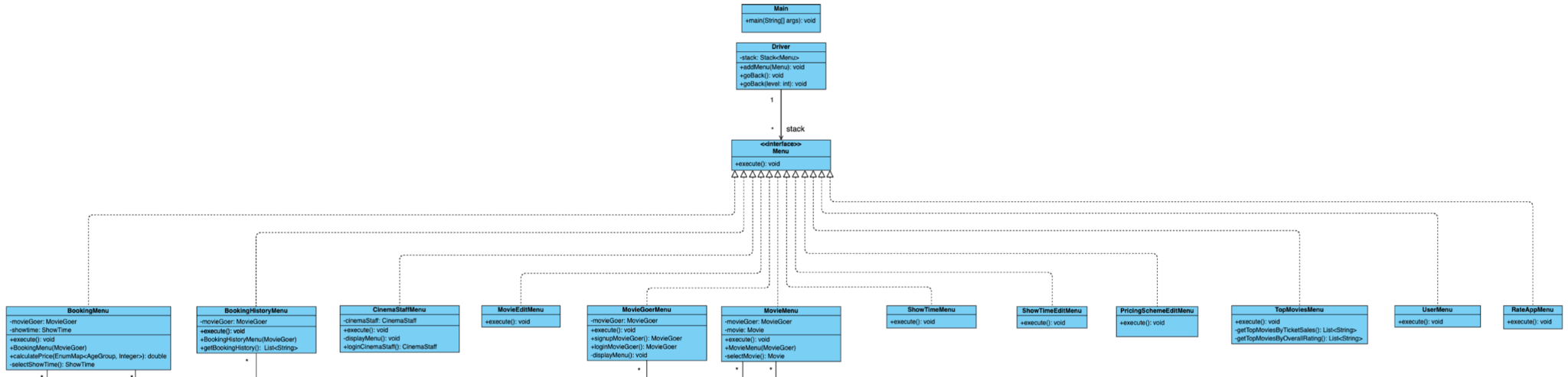
# UML Class Diagram

The UML Class Diagram below has been further sectioned into the 3 packages we created namely, *View*, *Model* and *Controller*.

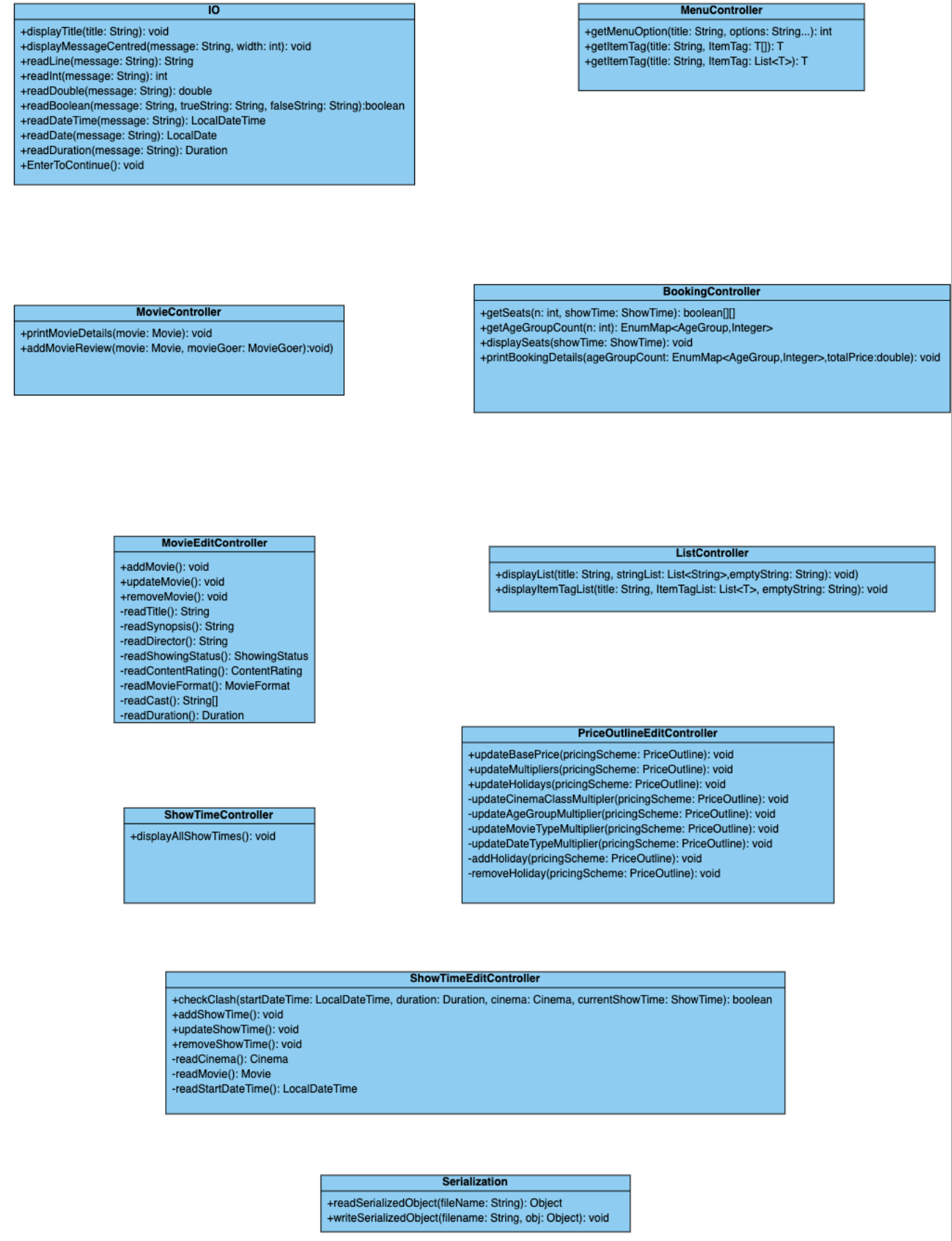


**(Please refer to the file labeled “UML\_Class\_Diagram.pdf” in the submitted folder for a higher-resolution image).**

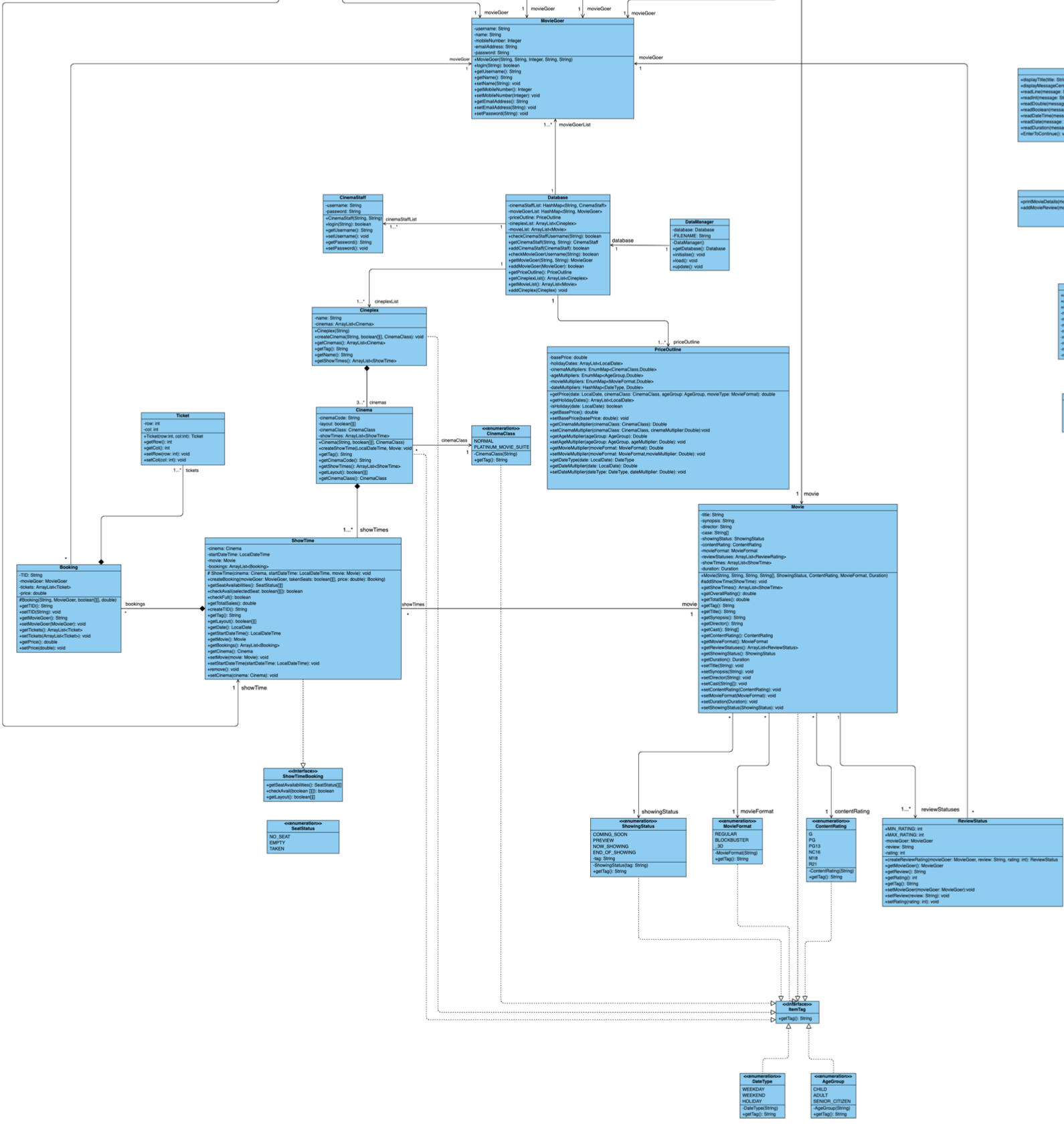
**View**

****

**Controller**

****

**Model**

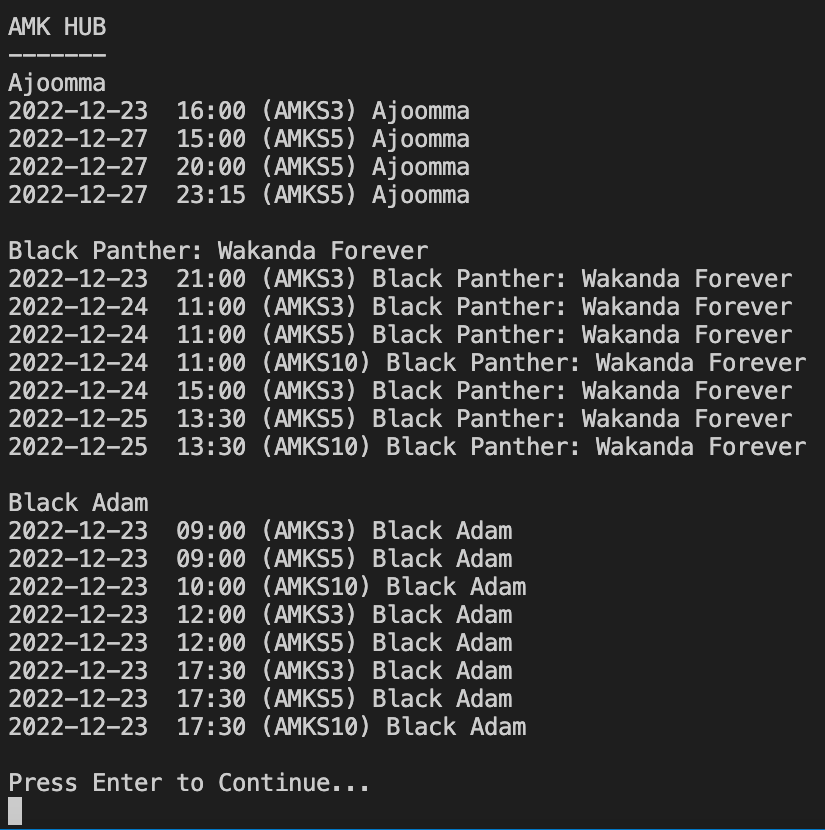


# 

# Test Cases (not shown in video)

**CinemaStaff User**

This will be the reference showtimes that we will use to test:



1. Adding an existing showtime

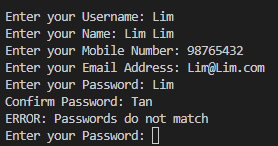


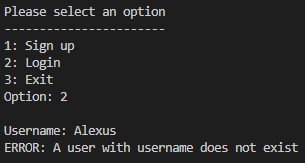
**MovieGoer User**

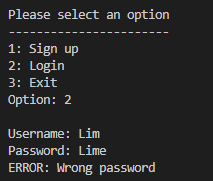
1. Exception when we try to create an account with a username of another pre-existing account.

****

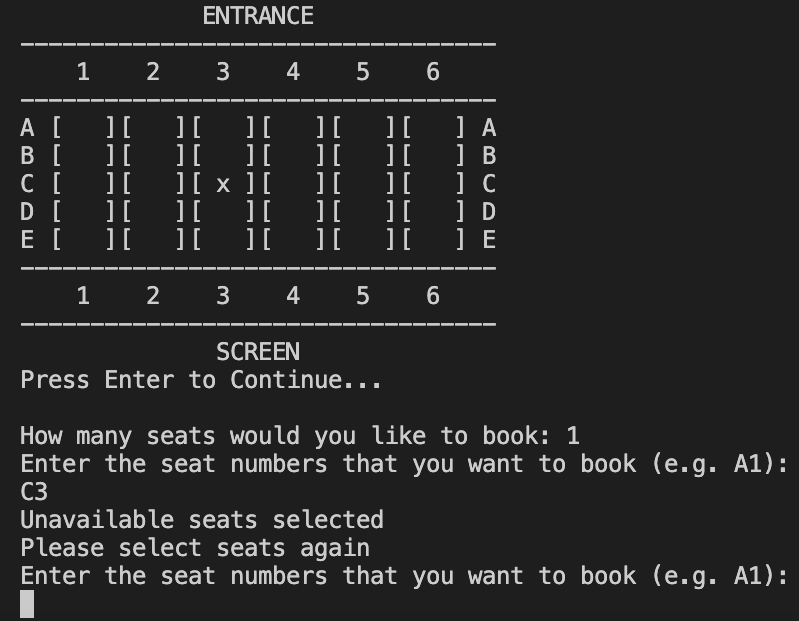
1. Exception when we enter the wrong password when confirming the password. The user will be prompted to re-enter the password.

****

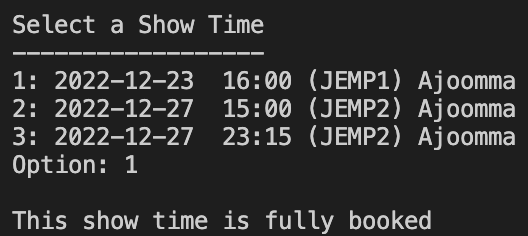
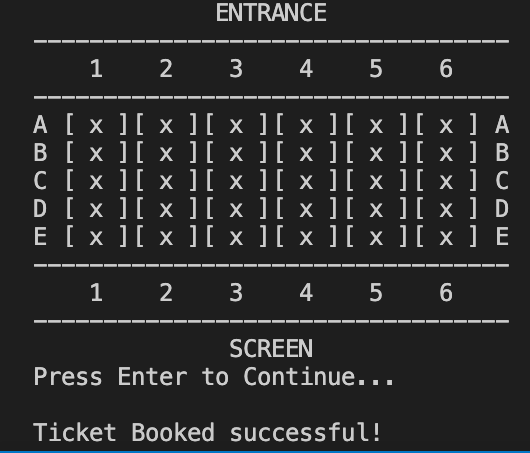
1. Trying to log in to an account that does not exist
2. Entering the wrong password when logging in



1. Selecting an already taken seat

****

1. Selecting showtime with a full booking



**Rating Application**

1. Any value that is not between the given option range of 1-5 will result in an exception.