Text

Description automatically generated

**SC2002 Object- Oriented Design & Programming**

**Project Report – MOvie Booking and LIsting Management Application**

**Group Assignment on MOBLIMA**

**AY22/23 SEM 1 | MACS, GROUP 1**

**Instructors: Dr Li Fang and Dr Zhang Jie**

**Declaration of Original Work for 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 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 |
| Khor Pock Xuan | SC2002 | MACS | pockxuan |
| Tan Jun Hong | SC2002 | MACS | junhong |
| Tan Jun Xiong | SC2002 | MACS | junxiong |
| Wesley Low | SC2002 | MACS | wesley |
| Yeo Zong Han | SC2002 | MACS | zonghan |

Table of Contents

[Introduction 3](#_Toc119249041)

[Design Considerations 3](#_Toc119249042)

[Approach Taken 3](#_Toc119249043)

[Assumptions Made 4](#_Toc119249044)

[Use of Object Oriented Concepts 5](#_Toc119249045)

[Abstraction 5](#_Toc119249046)

[Encapsulation/Information Hiding 5](#_Toc119249047)

[Inheritance 6](#_Toc119249048)

[Polymorphism 6](#_Toc119249049)

[Use of Object-Oriented Design Principles 7](#_Toc119249050)

[Single Responsibility Principle (SRP) 8](#_Toc119249051)

[Open-Closed Principle (OCP) 8](#_Toc119249052)

[Liskov Substitution Principle (LSP) 8](#_Toc119249053)

[Interface Segregation Principle (ISP)  8](#_Toc119249054)

[Dependency Inversion Principle (DIP)  9](#_Toc119249055)

[Applying Entity-Control-Boundary (ECB) 9](#_Toc119249056)

[Future Considerations 10](#_Toc119249057)

# Introduction

**MOvie Booking and LIsting Management Application (MOBLIMA)** is a console-based application to computerize online bookings, purchase of movie tickets, listing of movies, and sale reporting. It will be used by both movie-goers and cinema staff. The program has multiple interfaces including modifying movie and cinema listings, setting date and time, and listing the top five movies by either sales or reviews.

This objective of this report is to explain and demonstrate how key Object-Oriented concepts and design considerations were used to implement the application. This includes our approach, a UML Class Diagram, and a UML Sequence Diagram to show how objects interact with each other. This report will demonstrate several test cases to illustrate the functionality of our application.

# Design Considerations

## Approach Taken

We intended to design an application that is simple and user-friendly yet as realistic as possible. Therefore, the MOBLIMA application reflects and simulate a real-life application as closely as possible.

In an actual movie booking, it is important for data to be stored after each day. We created a database to store all information even after the application closes. Prior to start up, the system automatically recalls the most recent updates from its database so that both staff and moviegoers can access the latest information.

Our admin interface allows staff members to update, remove, or create a new movie or cinema listing. They can add a new movie to the database of known titles and update the movies screened at a cineplex. The showing period of popular movies can also be easily extended, which further simulates actual cinema operation. Staff members can also easily add (or remove) screenings of any movie to (or from) any cinema hall in any cineplex.

The booking interface allows customers the flexibility to book multiple tickets at once. Before finalizing their purchase, there will be a confirmation page that shows the total price of the tickets. They can also view their booking history to see which movies they previously booked. For every ticket purchased, they are entitled to one review and rating for that movie, for which they can check how many reviews they have left and edit them whenever they wish.

Each ticket contains identifiable information such as a seat ID and cinema code to allow them to easily find their seats. The ticket prices are differentiated between age groups (student, adult and senior citizen) and special days (such as public holidays).

Our application also accounts for adversarial input, so that a user who accidentally or deliberately makes invalid inputs would not inadvertently crash our programme. With all this, the application is designed to encompass real life situations and provides ease of usage.

We follow a layered design as in Figure 1.Diagram

Description automatically generated In this architecture, only downward dependency is permitted, and upward dependency is not allowed. For example, the “Application” classes depend on the “Manager” classes which interacts with the

Figure 1 “Database Storage Systems”.

## Assumptions Made

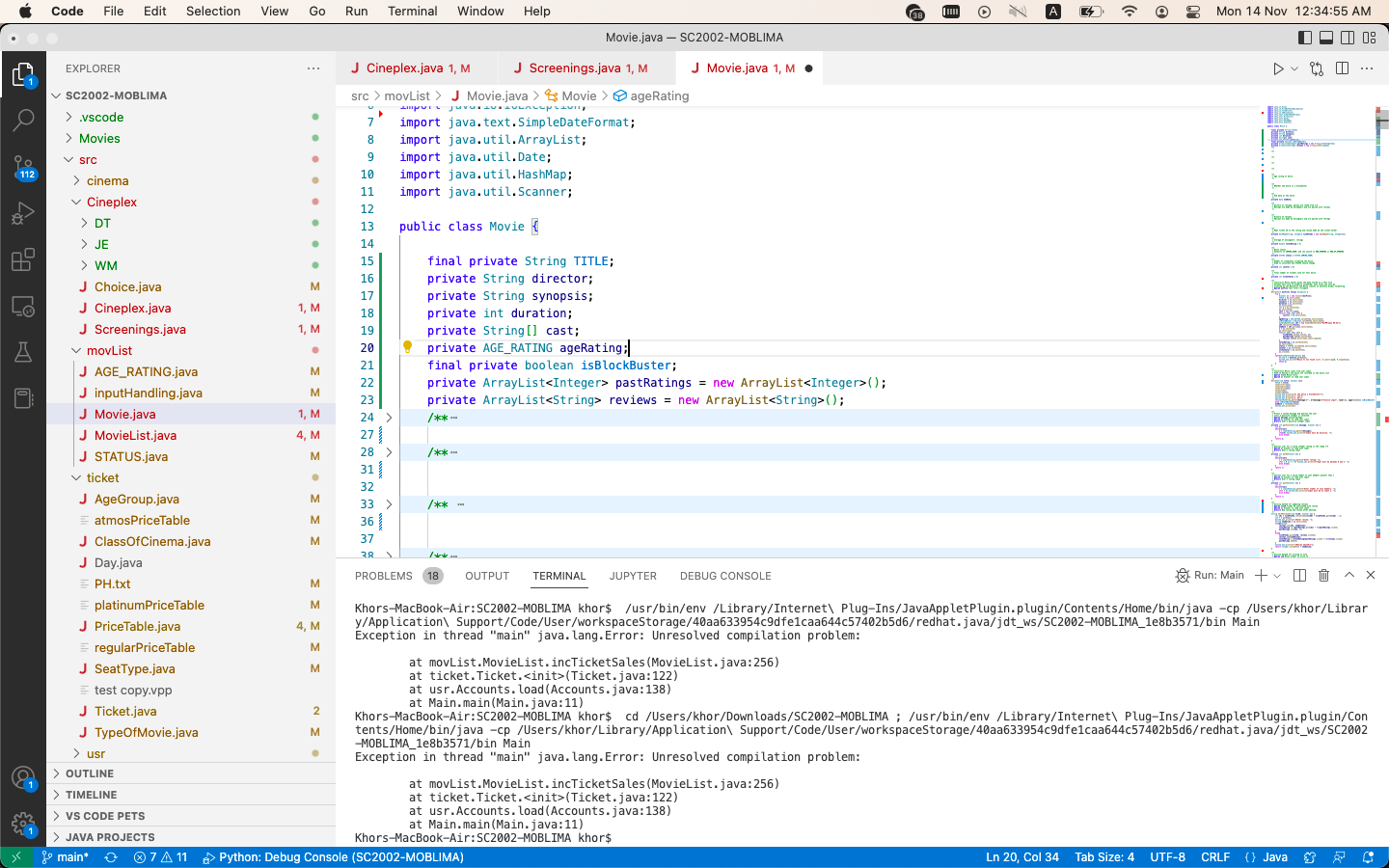
|  |  |
| --- | --- |
| 1 | This is a single-user application and there is no need concurrent access. |
| **2** | Only THREE cineplexes will be created. |
| **3** | The currency will be in Singapore Dollar (SGD) and inclusive of Good and Services Tax (GST). |
| **4** | Payment will always be successful. |
| **5** | Admin should always input correct movie titles, although they can edit other movie attributes |
| **6** | Time is in 24-hour format, cinema operates 24 hours each day and is open everyday. Local Date Time will be used. |
| **7** | There is no interface with peripherals, e.g. Payment, printer, etc. |
| **8** | Senior citizens and students can purchase online without validation of identity or age. The validation will be done upon entering the cinema. |

## Use of Object Oriented Concepts

### Graphical user interface, application Description automatically generatedAbstraction

Abstraction is the process of identifying the characteristics of an object that distinguishes them from other objects. For example, our Admin and Movie Goer classes extend from the User class because they contain similar methods such as choosing a cineplex. However, admin and movie-goers differ from each other in their functionalities. This allows for extensibility as we can further add new types of admin with different privileges without modifying the method that calls the class. This further reduces the complexity of the design and implementation, which aligns with the Open Closed principle of OOP. 

Encapsulation/Information Hiding

Encapsulation protects an object’s private data and enables access only via public get and set methods. Information hiding hides the details or implementations of a class from the user. This means the user of a class only need to know what a class does and how to call the methods to perform the tasks but need not know the implementation details for the method. For example, the movie class has private attributes as seen in the diagram. With information hiding, the admin classes can simply call these methods without understanding the implementation. Therefore, this helps to protect private data as only Movie class has full control of the private attributes and can prevent unauthorized access to the data.

### Text Description automatically generatedInheritance

Inheritance allows us to derive new classes from existing classes by taking their attributes and behaviours in addition to new capabilities. For example, the Admin class inherits all the attributes and methods of the User class – such as a global clock and methods for selecting a cineplex – but has the additional capability of changing the movie listing and cinema screenings. This makes our code more readable and reusable. If the attributes and methods of the parent class User change, we need not rewrite the code in the Admin module.

### Text Description automatically generatedText Description automatically generatedPolymorphism

Polymorphism is the ability of an object to take on different forms that behave differently. This improves functionality of our application as extending subclass behaviour can be done more easily in the future. We implement this in the Admin and Movie Goer classes where we inherit abstract methods from the User class, where the same method signature is defined appropriately for each class. In doing so, it allows our code to become more readable as we do not have to come up with different function names for methods that serve a similar purpose. This would further aid in ensuring that our code is readable and modifiable.

## Use of Object-Oriented Design Principles

We organise our application based on good design principles to manage dependencies by minimising the impact of change. Our application aims to achieve loose coupling and high cohesion by designing with reusability, extensibility, and maintainability in mind. For example, where Price Table computes ticket prices, changing the computation of a specific ticket price does not require any change to the Movie Goer class when booking a ticket.

Our program also has well-defined, simple, and independent units, which are each encapsulated and interact through well-designed interfaces. For instance, the Cineplex class does not require knowledge of how Movie List class works to use its functions. However, the classes are dependent such that information presented by Movie List, such as fetched movie attributes, still caters to the need of Cineplex class.

Our application is modelled based on the **SOLID** Design Principles.

### Single Responsibility Principle (SRP)

This principle advocates that each class should exist solely to perform a single responsibility. We incorporated this by creating different manager classes. For example, the Accounts class is solely for Users while the Movie List is solely for managing movies. As such, should any class require change, this mitigates the need to change many other classes, thus reducing functional overlap and achieving a low coupling effect. We ensure cohesion in our application by ensuring each class has strictly one responsibility.

### Open-Closed Principle (OCP)

The foundation this principle is Abstraction; this means a class should be open to extension but closed for modification. For example, our User is an abstract class that all users, admin or moviegoer, inherits. The use of abstract methods brings a degree of extensibility to our program. If new user account types need to be added in the future, we can easily implement this without changing other classes. This follows the OCP of flexibility, extensibility, and ease of maintenance.

### Liskov Substitution Principle (LSP)

According to Barbara Liskov: “A user of a base class should continue to function properly if a derivative of that base class is passed to it”. The subclass should be substitutable for the base class. Our application recognises that all users, admin or moviegoer, require common, basic methods as well as specialised methods. By having both inherit from a base User class, the account type can be initialised as either base or derived class without affecting user experience.

### Interface Segregation Principle (ISP)

This principle is built around the idea of having many specific interfaces are better than one general purpose interface, meaning subclasses should only implement methods that they will use. For example, we have multiple manager classes that performs different functionalities and are independent of the interfaces they do not require. Next, we have multiple application interfaces so that they do not depend on other applications that they do not require. By segregating the responsibilities of classes, we align with the ISP.

### Dependency Inversion Principle (DIP)

This principle states that both high- and low-level modules should depend on abstraction and that details should depend upon abstraction. For example, the user class methods are based on different classes of abstraction and the actual implementation depends on which class inherits. Treating the interaction itself as an abstract concept allows coupling of components to be reduced without introducing additional or specific coding patterns, allowing a lighter and less implementation-dependent interaction schema.

Throughout the design of the application, these design principles were consistently considered. The core design principles are implemented simultaneously, corresponding to the notion that these principles are usually concurrent and interleaving.

## Applying Entity-Control-Boundary (ECB)

In our MOBLIMA, we clearly define the role and relationship of every single class. Often, classes share a **has a** or **is a** relationship. By clearly identifying these factors, we can construct an elaborate structure for our project. A cinema **has** multiple screenings of various movies and each movie screening will **have a** set of seats.

In our MOBLIMA application, there are entities that hold tangible information (like Movie, Cinema and Ticket), and boundaries that interface with an external file system (like PriceTable) or directly with the user (like AdminApp and MovieGoerApp). Our control classes are coordination classes which comprise logic to coordinate the other classes and realise our use-cases. For example, Cineplex assists the user interface class User to check the movies that are screened in the database management class Cinema.

Ensuring that our application adheres to this principle so that entities and boundaries communicate only through controls will ensure a streamlined and efficient organisation. This allows for small modifications to our application to be easily made without having to change many classes.

# Future Considerations

Our application, while designed to be as realistic as possible, falls far short of reality. We make two proposals here to improve our system.

Firstly, we can enable concurrent access to our system. While concurrent accesses can be done by initializing multiple instances of the User class, the main concern would be synchronizing of data so that updates from one user would immediately reflect to the other. To improve running times, our application defers writing to files to when the user logs out. Working within this constraint, we can have each application module instantiated by a new programme startup module. Instead of initializing control classes from the User modules as they are now, they will instead be initialized by the startup so that each User uses the same pointers. Synchronicity for static attributes can then be provided by semaphores.

Secondly, we can consider a more automated and foolproof way to manage new movies. Instead of relying on the user to enter the correct details, we can instead access online movie databases such as IMDb to automatically fill details. This would make entering new movies easier and less prone to mistakes. While we could import the whole online database to the local repository, processing so much data would severely impact runtimes. Therefore, we recommend referencing online repositories only to add movies to the local database.

UML Class Diagram

Diagram

Description automatically generated

# Test Cases

Text

Description automatically generatedText

Description automatically generated

Text

Description automatically generated with low confidenceText

Description automatically generatedText

Description automatically generatedText

Description automatically generatedText

Description automatically generated

Text

Description automatically generatedText

Description automatically generatedText

Description automatically generatedText

Description automatically generated with medium confidenceText

Description automatically generated