

**NANYANG
TECHNOLOGICAL
UNIVERSITY**

SINGAPORE

SC2002 - Object-Oriented Design & Programming

Final Report

Lab SSP5 Assignment Group 4

Names:

Student ID:

KENNY SEAH YONG JIE

U2121539K

RUNGTA DHAIRYA

U2120483K

TEOH SHUN JET

U2020579C

TIWANA TEG SINGH

U2122816B

WONG JIA JI

U2140413K

Table of Content

1 Declaration of Original Work	2
2 Design Considerations, Principles & OO Concepts	3
2.1 Introduction	3
2.2 Design Considerations	3
2.2.1 Boundary Classes	3
2.2.2 Entity Classes	4
2.2.3 Control Classes	5
2.3 Design Principles	6
2.3.1 Single Responsibility Principle (SRP)	6
2.3.2 Open-Closed Principle (OCP)	6
2.3.3 Liskov Substitution Principle (LSP)	6
2.3.4 Interface Segregation Principle (ISP)	7
2.3.5 Dependency Injection principle (DIP)	7
2.4 Object-Oriented Concepts	7
2.4.1 Abstraction	7
2.4.2 Encapsulation	7
2.4.3 Inheritance	8
2.4.4 Polymorphism	8
2.5 Assumptions Made	8
3 Test Cases	9
4 Future Improvements	9
4.1 MovieGoer Module	10
4.1.1 Food Options	10
4.1.2 Better Case-Matching and Search Using Regular Expression	10
4.1.3 Seat Blocking Mechanism	10
4.2 Admin Module	10
4.2.1 Time Controller	10
5 UML Diagram	11

1 Declaration of Original Work

SC/CE/CZ2002 Object-Oriented Design & Programming
Assignment






APPENDIX B:

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 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
KENNY SEAH YONG JIE	SC2002	SSP5	 13/11/2022
RUNGTA DHAIRYA	SC2002	SSP5	 13/11/2022
TEOH SHUN JET	SC2002	SSP5	 13/11/2022
TIWANA TEG SINGH	SC2002	SSP5	 13/11/2022
WONG JIA JI	SC2002	SSP5	 13/11/2022

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

2 Design Considerations, Principles & OO Concepts

2.1 Introduction

MOBLIMA is an all-in-one movie booking application which comprises several features such as the management of accounts, the booking of movie tickets and the updating of movie sessions. To implement such a complex system, the object-oriented design paradigm is ideal as compared to sequential programming as it models after real-life entities and their interactions with each other. However, due to the need to implement multiple classes for a multi-use-case application, simply implementing the software using object-oriented design paradigm would make the implementation process complicated and hard to debug. Thus, we have also employed the use of Boundary-Control-Entity architectural pattern as well as the SOLID design principles to aid us in implementing the software. In addition, we have also created some custom exceptions to deal with runtime exceptions, as well as going through test cases to ensure the code runs as expected.

2.2 Design Considerations

In designing our application, we have opted to use the Boundary-Control-Entity (BCE) architectural pattern fitted to use-case driven object-oriented applications such as MOBLIMA. This means that all component classes in the application developed by our team are either boundary, control, or entity classes, and these 3 categories of classes share this simple relationship.



2.2.1 Boundary Classes

Boundary classes interact with external actors, in this case users (moviegoers and administrators) of the MOBLIMA application. In our source code, the user interfaces of the application take on the role of boundary classes and have their class name titled appropriately as “XXXUI”, where “XXX” refers to the type of user interface.

The first and main user interface that would be called upon for our application is the MainMenuUI. MainMenuUI will then display options for users to choose from and prompts, get user input, and call upon other user interfaces or control classes at the request of the users.

As illustrated above, the main responsibilities of boundary classes are to:

1. Interact with external actors (i.e. users)
2. Get user input.

In our application, we have implemented the following boundary classes:

Boundary class	Function
MainMenuUI	Main function to start the MOBLIMA program; Displays all main options for the user to choose from
AccountSettingUI	Handles all editing and setting functions related to the Account created by admin or MovieGoer
BookBuyUI	Purchase and book seats for selected movie and calls the transaction class
ChangePriceUI	Display the UI for the admin to change and update price
MovieBookingViewUI	To display and get user input to select their movie and calls the BookBuyUI
MovieDetailViewUI	Display Movie Details in a string format
MovieSearchUI	To search and display Movie by any search attributes or ID

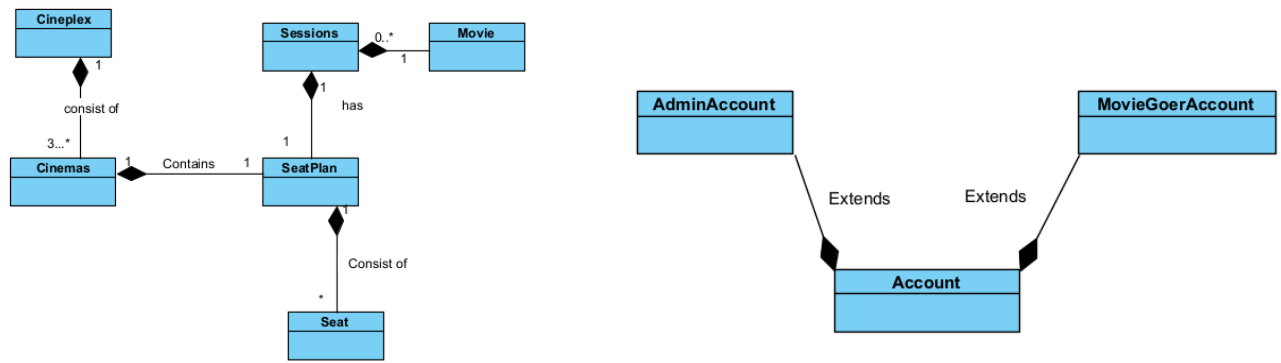
2.2.2 Entity Classes

Entity classes are those that represent long-lived persistent information and are derived from real world objects. Examples of entity classes in our application include Account, Movies, Cinemas, et cetera. Since the information needs to be persistent, we have opted for entity classes to implement the Serializable interface. This way, objects of entity classes may be easily read (written) to plain text files using FileInput(OutputStream) and ObjectInput(OutputStream). For instance, since the Account class and its subclasses implements the Serializable interface, we can read and write objects of those classes easily using the DataManager class.

In our application, we have implemented the following boundary classes:

Entity class	Function
Account	Represent either an Admin or a moviegoer
Cinemas	Represent a Cinemas
Cineplex	Represent a cineplex
Holiday	Stores the dates of holidays
Movie	Represent and contain the detail of a single Movie
Price	List of prices representing the different category of users and day
Review	Contain a rating and review
Seat	Represent a single seat
SeatPlan	Represent a 2D Array of Seat
Sessions	Represent a movie showtime
Transaction	Represent a single Transaction when a ticket is brought

Like how in real life objects can contain other object, entity classes can be a part of another entity, In MOBLIMA these few classes share a is-a/has-a relationship as represented by the UML Diagram below:



2.2.3 Control Classes

Control classes bridge the gap between boundary classes and entity classes. They encapsulate the logic and algorithms required to execute the different use cases. In our source code, the control classes are aptly named “XXXController” or “XXXManager”, where “XXX” refers to the different types of control classes. For example, continuing with the above illustrations, the control class that bridges the MainMenuUI and the Account classes for the use-case of registering new moviegoer accounts is the RegistrationManager. To register an movie goer account, MainMenuUI creates an instance of RegistrationManager and calls upon its public createMovieGoerAccount() method without having to know the code implementation.

In our application, we have implemented the following control classes:

Control class	Function
AccountManager	Keeps track of active account
AccountSetting	Contains logic to update account settings
CinemasContoller	To read cinemas and returns the data from the database
CineplexController	To read cineplex data and return the data from the database
DataManager	To read and write account data
HolidaysCtrl	To add new date that represent holidays
LoginManager	Contains logic for users to login in
MoviesCtrl	Read and write to database, creates/update/remove movie
PriceCtrl	Read and write to database, update the pricing of the movie
RegistrationManager	To write new accounts to database
SessionsCtrl	Read and write to database, create/update/remove sessions
TransactionsCtrl	To write to database, creates new transaction when booking is done.

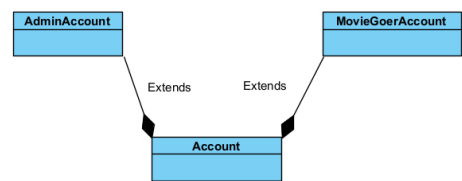
2.3 Design Principles

Our team also implemented the SOLID principles, which is an essential object-oriented approach intended for any form of software structure design or code, in conjunction with an extensive test suite. This ensured that the MOBLIMA app had minimal software rot, whilst establishing a more readable, modular and maintainable app that is easy to refactor and debug.

2.3.1 Single Responsibility Principle (SRP)

Description:	SRP states that every class should ideally be only responsible for a single functionality of a system and thus all class members should be corresponding to that singular purpose. As classes now only have one reason to change, it ensures a higher level of class cohesion.
Example:	In accordance with the BCE principles, we have implemented each boundary and control classes to correspond with one use-case. It is shown in the examples above and our detailed UML Diagram below.

2.3.2 Open-Closed Principle (OCP)

Description:	OCP ties back down to the concept of Abstraction and it states that we have to be able to extend a class's behaviour without directly modifying its source code.	 <pre> classDiagram class Account class AdminAccount class MovieGoerAccount Account < -- AdminAccount Account < -- MovieGoerAccount </pre>
Example:	In our application, MovieGoerAccount and AdminAccount extends directly from the base Account class. Should there be any need to introduce different types of account, we can simply extend from the Account class without changing its implementation.	

2.3.3 Liskov Substitution Principle (LSP)

Description:	LSP states that all subclasses of a superclass should function the same manner as its superclass. Thus, if we were to substitute the objects of a base class, with the objects from a derived subclass, we should be able to do so without disrupting the functionality of MOLIBA. This related back to Inheritance that we will explain in the next section.	<pre> public void displayAccountDetail(){ Account tempAccount = accountMgr.getActiveAccount(); if (tempAccount.getAccountType() == AccountType.ADMIN) displayAdminAccountDetail((AdminAccount)tempAccount); else displayMovieGoerAccountDetail((MovieGoerAccount) tempAccount); } </pre>
Example:	In our application, we are able to call getAccountType() to determine to account type despite not knowing whether the Account object being referenced to is an AdminAccount or MovieGoerAccount.	

2.3.4 Interface Segregation Principle (ISP)

Description:	ISP states that we should ideally split larger, general-purpose interfaces into smaller, more client-specific interfaces. This made sure that all our implementing classes did not need to depend on interfaces that were irrelevant to them or that they did not need to use. This principle hopes to achieve the same goal as the SRP.
Example:	As there was no need to implement interfaces in our code, this principle is not applicable to our application and we were able to prevent 'fat' interfaces with multiple methods.

2.3.5 Dependency Injection principle (DIP)

Description:	DIP makes sure we utilize the concept of abstraction, thus interfaces and abstract classes, over concrete implementations. Any high-level module should also not be dependent on a lower-level module, and both should depend on abstraction.
--------------	---

Example:	As there was no need to implement interfaces or abstract classes in our code, this principle is not applicable to our application.
----------	--

2.4 Object-Oriented Concepts

2.4.1 Abstraction

Description:	Abstraction is the process of removing characteristics from an object, in order to reduce it to a set of essential characteristics required. This can be seen in the classes of our application, in which all but the relevant attributes and methods of each object are hidden for the goal of reducing complexity and increasing efficiency.	<pre>switch(choice) { case 1: accountSettingMgr.updateUserName(); displayAccountDetail(); break; case 2: accountSettingMgr.updatePassword(); displayAccountDetail(); break; case 3: accountSettingMgr.updateEmail(); displayAccountDetail(); break; case 4: accountSettingMgr.updateMobileNumber(); displayAccountDetail(); break; case 5: accountSettingMgr.deleteAccount(); break; case 6: break; }</pre>
Example:	In our application, the UI classes simply have to call upon controller classes and use its public methods to execute use-cases without knowing how the code was implemented behind the scenes. For example, the UI class responsible for updating account settings can simply call upon the methods provided by the AccountSettingManager class without knowing the underlying implementation.	

3.4.2 Encapsulation

Description:	The idea of encapsulation is to hide the variables or data of a class from any other class and can only be accessed through any member function of its own class in which it is declared. This restricts the user's direct access to an object's private data.	<pre>public class Movie implements Serializable{ /** * This movie's unique ID */ private int id; /** * This movie's title */ private String title; /** * This movie's MovieType */ private MovieType type; }</pre>
Example:	Within our application, all attributes of entity classes are declared as private or protected (for classes that need to be extended). Furthermore, Public methods like the getters and the setters are created in order to access the object's private or protected data.	

2.4.3 Inheritance

Description:	Inheritance is the mechanism used to inherit and thus reuse features of an existing/parent class into another new/child class. This is beneficial for code reusability and efficiency as the new class can reuse the methods and fields of the parent class, while also being able to add new features into this child class.	<pre>public class Account implements Serializable{ // ... } public class AdminAccount extends Account{ // ... } public class MovieGoerAccount extends Account{ // ... }</pre>
Example:	In our application, we have two different types of account- one for Admins and Movie Goers. Since they share similar features such as username and password, we extended AdminAccount and MovieGoerAccount from the base Account class.	

2.4.4 Polymorphism

Description:	Polymorphism allows a class to provide different implementations of a method. It is a way to dynamically decide which version of a function will be invoked. This creates a system that is extendable as it facilitates the adding of new classes into the application with minimal modifications to the system's code.	<pre> public void displayAccountDetail(){ Account tempAccount = accountMgr.getActiveAccount(); if (tempAccount.getAccountType() == AccountType.ADMIN) displayAdminAccountDetail((AdminAccount)tempAccount); else displayMovieGoerAccountDetail((MovieGoerAccount) tempAccount); } </pre>
Example:	An example in our application is assigning the current active account into a temporary Account variable and subsequently calling the getAccountType() method on the variable without knowing whether it's a AdminAccount or MovieGoerAccount to determine the AccountType.	

2.5 Assumptions Made

Beyond the assumptions given in the assignment document, our group has made the following additional assumptions:

1)	User Input Matching:	User's input must match the records in the system database for the application to return a successful result. There is no need to deal with cases such as casing and similarity in the search input.
2)	Number of Cineplexes created:	For demonstration purposes, only 3 cineplexes were created, with varied Cinema Theatres inside each of the Cineplexes.
3)	Manual Updating of Movie Status:	Administrators have to manually update the movie statuses.
4)	Age Verification for ticket purchase:	Student or Senior Citizen can purchase their respective age category tickets without verification. Actual verification of identities can be done physically at the Cineplex Entrance by the staff who are checking the tickets.
5)	Login for MovieGoers:	Movie goers have to login or create an account in order to book tickets, give reviews and ratings, as well as to view their respective booking history.
6)	Application Type:	MOBLIMA is assumed to be a single-user type of application so multi-user access to the app is not considered.

3 Test Cases

All Essential Test cases were covered by our group video: (<https://youtu.be/VPL5AwTIUao>). But we had additional Exemption cases that under our Custom Exceptions section to do all Exception Handling.

Exceptions cases (When invalid input is given):

Test Case Descriptions:	Test Inputs/Outputs:	
Empty String	<div>Enter movie title</div> <div>Cannot be empty, try again!</div>	<div>Enter movie synopsis:</div> <div>Cannot be empty, try again!</div>

Invalid input for selection	<pre> Select movie types: 1. 2D 2. 3D Enter option: ----- 4 Wrong input! Going back ===== </pre>
Overlapping Session	<pre> Enter Session Start Date and Time YYYY, dd/MM/yyyy HH:mm Monday, 14/11/2022 12:00 Show cannot be created (due to conflicting datetime) </pre>
Negative Price	<pre> Enter new student price -5.0 Sorry updated price cannot be negative, terminating update, returning to main menu </pre>
Movie end date is before start date	<pre> Enter movie release date (DD/MM/YYYY) : 12/12/2022 ----- Enter movie end date (DD/MM/YYYY) : 11/11/2022 Invalid Movie End Date! Returning to menu </pre>
Movie not found	<pre> Movie Could Not Be Found... Movie does not Exist in the database Exiting now. </pre>

4 Future Improvements

For future improvements, we propose the following further enhancements to be made to the application, for each of the MovieGoer and the Admin module:

4.1 MovieGoer Module

4.1.1 Food options

Add a Food interface, which has the parameters: name (String) and price (double). This interface will be implemented by the different food object we can have the user buy. This can be done with the principle of Interface segregation as all food items share the common parameter of food name and price. We can then implement a FoodManager class to handle the create/ update and remove food items from the menu to ensure that will use the interface class. This creates dependency on the interface instead of the class, ensuring that the new feature is bug free when new food category is added, example drinks and combo sets.

4.1.2 Better Case-Matching and Search Using Regular Expression

We can make use of Regular Expression (Regex), like we did for our email inputs, to enable the application to prompt user inputs of certain requirements, such as requiring the alphanumeric symbols or different types of character types in passwords.

In addition, it enables the application to return better search results, since currently the users' input has to match the system record exactly for the application to return a search result. We can have a control class with methods for getting different types of input as well as search using Regex to ensure reusability.

4.1.3 Seat Blocking Mechanism

As seen during the Covid-19 pandemic, some seats were blocked for booking due to safe-distancing measures. As our current design stands, it will be very easy for us to implement seat blocking mechanism in our application as we only have to change the implementation for the SeatPlan class without touching any other classes.

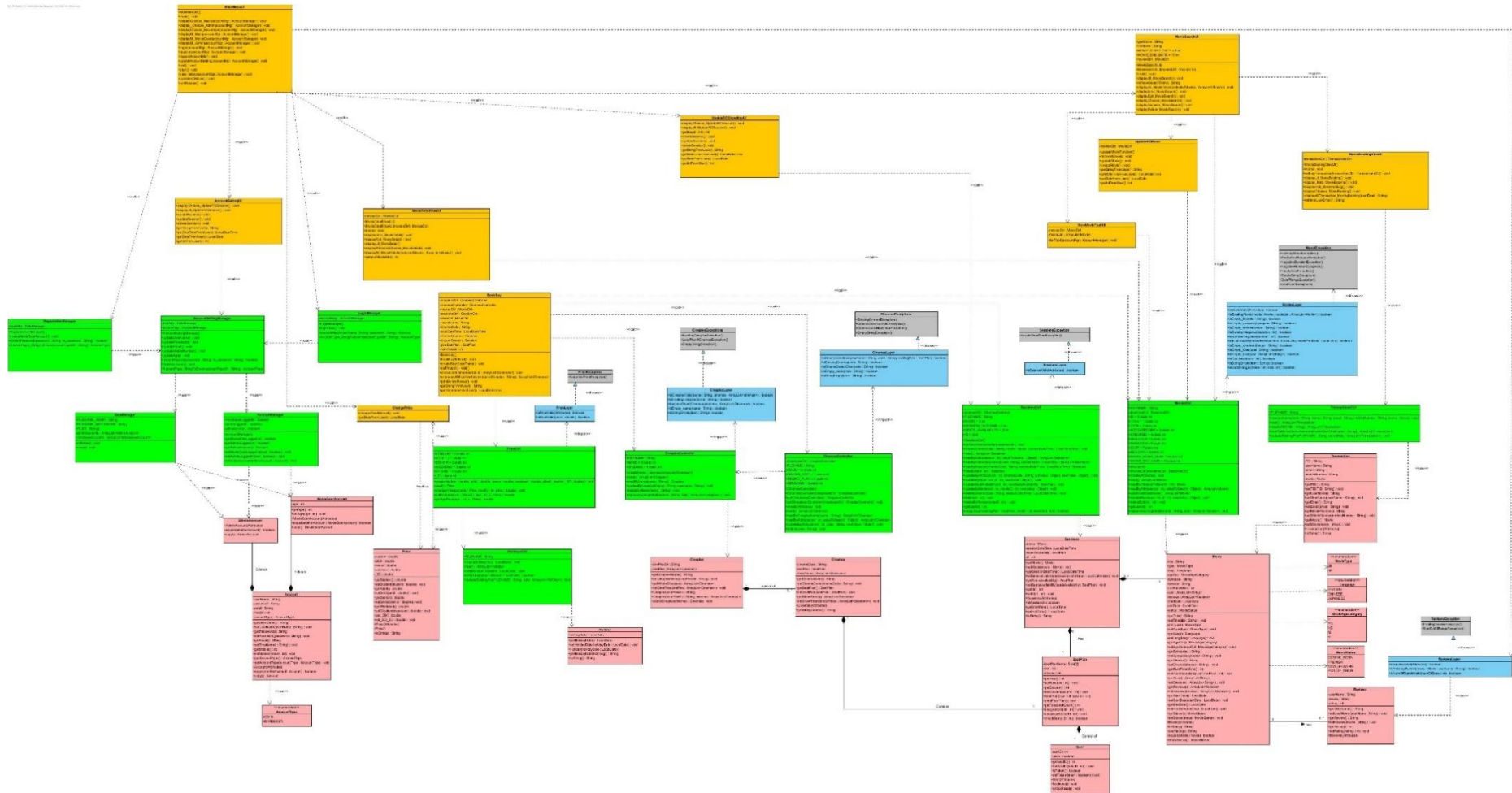
4.2 Admin Module

4.2.1 Time Controller

Currently, the administrators have to manually edit the movie status. This can become quite troublesome when the cineplex expands its operations. Thus, we can create a time controller class that constantly checks the system date daily and automatically update the movie status to "Preview", "Now Showing" and "End of Showing" when the system's date meets the conditions for the various statuses.

5 UML Class Diagram

Note: High resolution class diagram image can be found in the zip folder or [this google drive link](#).



Legend:

