



## **ASSIGNMENT**

### **SC2002: Object-Oriented Design & Programming**

#### *Building an OO Application*

#### **Lab Group: REP**

#### **Group 4**

Aaron Chua Cher Jin (U2121565G)

Chay Hui Xiang (U2122080L)

Ivan Loke Zhi Hao (U2120789B)

S Jivaganesh (U2120817J)

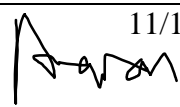
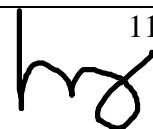



Ang Kai Jun (U2122649H)

### **Declaration of Original Work for SC2002 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 (SC2002)	Lab Group	Signature/ Date
Aaron Chua Cher Jin	SC2002	REP	 11/11/2022
Chay Hui Xiang	SC2002	REP	 11/11/2022
Ivan Loke Zhi Hao	SC2002	REP	 11/11/2022
S Jivaganesh	SC2002	REP	 11/11/2022
Ang Kai Jun	SC2002	REP	 11/11/2022

## Table of Contents

<b>UML Class Diagram .....</b>	<b>4</b>
<b>Test Cases .....</b>	<b>5</b>
<b>Diagram .....</b>	<b>5</b>
<b>Screenshots for Test Case 1: .....</b>	<b>5</b>
<b>Screenshots for Test Case 2: .....</b>	<b>8</b>
<b>Write-Up .....</b>	<b>9</b>
<b>Design Considerations: .....</b>	<b>9</b>
Low coupling: .....	9
High Cohesion: .....	9
Single Responsibility Principle: .....	10
Open-Closed Principle: .....	10
Interface Segregation Principle: .....	10
<b>Additional Features .....</b>	<b>11</b>
Feature 1- Membership System: .....	11
Feature 2 – Addition of new cinema class with different seating arrangement: .....	11

We have also submitted an SVG file and two PNG files of the class diagram.



## **Test Cases**

### **Diagram**

We performed thirteen test cases on MOBLIMA. The first to seventh test cases are the given required test cases while the eighth to thirteenth test cases are those that we have come up with.

<b>No.</b>	<b>Description</b>	<b>Description</b>
1	Configuring a holiday date and the ticket price is shown correctly when booking is done on that date	Demonstrated in Screenshots below
2	Booking on a different day of the week to demonstrate the differences in prices	Demonstrated in Screenshots below
3	Booking different type of cinema to demonstrate the differences in prices	Demonstrated in Video
4	Configuring “End of Showing” date and the movie should not be listed for booking	Demonstrated in Video
5	Booking only allowed for “Preview” and “Now Showing” status	Demonstrated in Video
6	Pricing of the different age groups and timings and should be reflected in the transaction	Demonstrated in Video
7	Pricing should be different based on the type of movie and reflected in the transaction (example: Regular, Blockbuster, 3D)	Demonstrated in Video
8	Should not be able to book seats that are taken (should be able to update once seat is taken)	Demonstrated in Video
9	Searching for movie title, should list every movie based on valid keywords (example: capitalisation should not matter)	Demonstrated in Video
10	Booking should be able to handle errors from user inputs (example: keying seats outside of the given array)	Demonstrated in Video
11	Once showing has been declared “End of Showing” by the admins, they should no longer be able to edit or view it	Demonstrated in Video
12	Once the prices has been edited, admins should be able to see the corrected prices in Display All Prices	Demonstrated in Video
13	Admins should be able to view any changes to the holiday dates	Demonstrated in Video

Screenshots for Test Case 1:

- Test Case: Configuring a holiday date and the ticket price is shown correctly when booking is done on that date

#### List of prices and multipliers configured by Admin:

- Holiday: Multiplier of 1.25 times on the base price

```
*****
MOBLIMA -- Admin -- Settings Module (Display All Prices):
Movie Types      | Cinema Classes | Movie Goer Age | Days | Seat Types
REGULAR: $10.0   | GOLD_CLASS: x2.5 | CHILD: x0.7    | WEEKEND: x1.25 | REGULAR: x1.0
THREE_D: $15.0   | DELUXE: x1.5    | ADULT: x1.0    | WEEKDAY: x1.0  | COUPLE: x1.0
BLOCKBUSTER: $12.0 | REGULAR: x1.0   | SENIOR: x0.5   | PUBLIC_HOLIDAY: x1.25 | ELITE: x1.4
                                                           | ULTIMA: x1.4
*****
```

- Currently, 26-12-2022 is not a public holiday.

```
*****
MOBLIMA -- Admin -- Settings Module (Display Holiday Dates):
Current Holiday Dates are:
01-12-2022
01-01-2023
02-01-2023
*****
```

- The price for 2 ADULT BLOCKBUSTER tickets at a GOLD\_CLASS cinema with ULTIMA seats on a WEEKDAY will be \$84.

```
Ticket 1 | Please key in your preferred payment method: 2
Please confirm the details of your booking:
Movie:      Black Panther: Wakanda Forever
Cineplex:   Cathay@Orchard
Cinema:     1
Price:      84.0
Payment:     Debit Card
Time:       26-12-2022 13:00
Seats:      E1 E2
```

- As shown, 26-12-2022 has now been configured to be a holiday date.

```
*****
MOBLIMA -- Admin -- Settings Module (Display Holiday Dates):
Current Holiday Dates are:
01-12-2022
26-12-2022
01-01-2023
02-01-2023
*****
```

- Similarly, the same booking on 26-12-2022 (holidays) results in \$104.0 due to the 1.25 multiplier.

```
Ticket 1 | Please key in your preferred payment method: 2
Please confirm the details of your booking:
Movie:      Black Panther: Wakanda Forever
Cineplex:   Cathay@Orchard
Cinema:     1
Price:      104.0
Payment:    Debit Card
Time:       26-12-2022 13:00
Seats:      E1 E2
```

## Screenshots for Test Case 2:

- Test Case: Booking on a different day of the week to demonstrate the differences in prices

### List of prices and multipliers configured by Admin:

- Weekday (ie Monday - Friday): Multiplier of 1.0 times on the base price
- Weekend (ie Saturday, Sunday): Multiplier of 1.25 times on the base price

```
*****
MOBLIMA -- Admin -- Settings Module (Display All Prices):
Movie Types      | Cinema Classes | Movie Goer Age | Days           | Seat Types
REGULAR: $10.0    | GOLD_CLASS: x2.5 | CHILD: x0.7     | WEEKEND: x1.25 | REGULAR: x1.0
THREE_D: $15.0    | DELUXE: x1.5    | ADULT: x1.0     | WEEKDAY: x1.0  | COUPLE: x1.0
BLOCKBUSTER: $12.0 | REGULAR: x1.0   | SENIOR: x0.5    | PUBLIC_HOLIDAY: x1.25 | ELITE: x1.4
*****
ULTIMA: x1.4
```

- The price for 2 ADULT BLOCKBUSTER tickets at a GOLD\_CLASS cinema with ULTIMA seats on a WEEKDAY (30-11-2022, Wednesday) will be \$84.

```
Ticket 1 | Please key in your preferred payment method: 2
Please confirm the details of your booking:
Movie:      Black Panther: Wakanda Forever
Cineplex:   Cathay@Orchard
Cinema:     1
Price:      84.0
Payment:    Debit Card
Time:       30-11-2022 11:00
Seats:      E1 E2
```

- However, the same booking on a WEEKEND (31-12-2022, Saturday), results in \$104.0 due to the 1.25 multiplier.

```
Ticket 1 | Please key in your preferred payment method: 2
Please confirm the details of your booking:
Movie:      Black Panther: Wakanda Forever
Cineplex:   Cathay@Orchard
Cinema:     1
Price:      104.0
Payment:    Debit Card
Time:       31-12-2022 10:00
Seats:      E1 E2
```



## **Write-Up**

### **Design Considerations:**

In our code, there are 5 main types of classes/ enums/ interfaces.

1. **“database” classes:** These are low level classes that directly interact with the .dat files which store data. Each of these classes contain read and write methods that allow for data retrieval and data storage
2. **“object” classes:** These are low level classes which will be frequently instantiated as part of the functioning of the app.
3. **“module” classes:** These are high level classes that call upon the methods of “object” classes. For example, the AdminModule calls the methods from the Admin class.
4. **Enums:** These enums are used to store a list of constants for a particular category. For example, there are 4 constants stored in the MovieStatusType enum
5. **Interfaces:** These interfaces specify abstract methods which will need to be specified in a class that implements the interface

When planning our system design, we applied the following OO designs taught in the lectures:

### **Low coupling:**

Low coupling refers to a system where there are minimal links between classes, and that classes are only linked when required. In our design, “object” classes are intended to be low level classes which are standalone and not linked to each other, apart from enums. On the other hand, “module” classes are designed to handle app functions which require interaction between “object” classes, hence they are linked to one or more “object” classes. With such a system design, we are able to reduce the linkages between the “object” classes, while keeping the necessary links between the “module” classes and “object” classes.

### **High Cohesion:**

High cohesion refers to a design where similar methods are grouped together in the same class. In our design, each class serves its own purpose. For example, Booking functions are grouped under BookingModule, while Admin functions are grouped under AdminModule. Hence, by doing so, this allows us to achieve high levels of cohesion in our design.

### Single Responsibility Principle:

Single responsibility principle is similar to high cohesion. It states that each class should only have one responsibility and hence one reason for change. In our design, each class is designed with only one purpose in mind. For example, there are multiple .dat files which each stores different information. In order to ensure that each class is only responsible for one function, we created multiple “database” classes which only interact with their respective .dat files. Hence, by doing so, we thereby apply the Single Responsibility Principle.

### Open-Closed Principle:

The Open-Closed Principle states that classes should be open for extension but closed for modification, and one way to do it involves the implementation of interfaces and abstract classes. In our design, we have adopted the Open-Closed Principle for our DB classes. Each DB class extends the abstract class SerializeDB which contains 2 methods (read(), write()). By adopting such a design, each class will definitely inherit the 2 methods of read and write, but are free to add on more methods if required for their specific functions. For example, the CineplexDB class contains an additional method called generateShowingId() which is used to generate the showingID. Hence, this shows the benefit of implementing the Open-Closed Principle.

### Interface Segregation Principle:

Interface Segregation Principle refers to the principle of not having “fat” interfaces, and that each class should only implement interfaces with relevant methods, rather than one containing other irrelevant methods. To apply the Interface Segregation Principle, we implemented 2 interfaces for the “module” classes (LoginInterface, ModuleInterface). The LoginInterface contains the login() method, and is implemented by the AdminModule and MovieGoerModule, while the ModuleInterface contains the run() method and is implemented by all “module” classes. By splitting up the 2 methods into separate interfaces, classes that do not require the login() method will not need to implement them, thereby achieving Interface Segregation Principle.

## Additional Features

### Feature 1- Membership System:

We could implement an annual membership scheme for frequent movie goers who purchase tickets from MOBLIMA. This will be an annual subscription plan available to all movie goers in their interface and there will be 3 tiers to choose from: Basic, Advanced, Premium. These tiers will offer a discount of 10%, 15% and 20% respectively, on all ticket prices at all cineplexes for the movie goer who purchases them at any time of the day.

To incorporate this function, we will add a private attribute to the MovieGoer class type. This attribute will be an enum, MembershipStatus, with four possible values: Non, Basic, Advanced and Premium. Our design for high cohesion and loose coupling means that we are able to reuse the same MovieGoer class and extend its attributes easily by adjusting just one class. Similarly, using our single responsibility principle, the only changes we will make to reflect the change in prices will be in the SettingsModule and the BookingModule. We will add a single multiplicative factor based on the membership status of the movie goer into the SettingsModule which will be reflected when we use the calculate price method in BookingModule.

### Feature 2 – Addition of new cinema class with different seating arrangement:

Typically, the cinema will experience a high volume of eager MovieGoers on premiere day for blockbuster movies. Thus, to accommodate these large crowds, another possible feature we can implement is to have a new “PremierShowing” class which has twice the seating capacity of a “RegularShowing” class.

As a result of our utilisation of the high cohesion principle, each class has a specific function. Therefore, it is easy to identify the classes that needs to be modified. In our case, we only need to add a new showing subclass. This subclass will be extended from the current “Showing” abstract class. The use of abstract classes allows us to easily create new concrete subclasses without the other current concrete subclasses being affected. In this case, the creation of “PremierShowing” subclass will not impact the current “RegularShowing”, “DeluxeShowing” and “GoldShowing” subclasses and any methods which currently depend on these subclasses.