NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

School of Computer Science and Engineering

SC2002: Object-Oriented Design & Programming

Assignment: Building an OO Application

MOBLIMA Report

AY2022/2023 Semester 1

Lab Group	SE4
Team Number	1

Members:

HASHIL JUGJIVAN	U2120599F
JONATHAN LIM JUN WEI	U2121770L
RYAN TEO CHER KEAN	U2122540D
ALOYSIUS TAY ZEN	U2121353L

Content

1. ***Declaration of Original Work***

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 honoured the principles of academic integrity and have upheld the Student Code of AcademicConduct 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	Lab Group	Signature & Date
Hashil Jugjivan	SC2002	SE4	[illegible] 13/11/2022
Jonathan Lim Jun Wei	SC2002	SE4	[illegible] 13/11/2022
Ryan Teo Cher Kean	SC2002	SE4	[illegible] 13/11/2022
Aloysius Tay Zen	SC2002	SE4	[illegible] 13/11/2022

2. ***Assumptions Made***

The assumptions made when developing the Movie Booking and Listing Management Application (MOBLIMA) are as follows:

a. The currency will be in Singapore Dollars (SGD) and inclusive of Goods and Services Tax (GST).
b. There is no need to interface with external systems.
c. Payment is always successful.
d. There is no need for the customer to log in.
e. A simple login for staff is sufficient.
f. Three cineplexes will be created for demonstration.
g. The application is made for only one outlet of the cinema.
h. Senior citizens can purchase tickets online without validation of identity or age.
i. The customer booking a movie will have at least one credit or debit card to make the purchase
j. There are no refunds or cancellations of movie bookings.
k. Each cineplex will have the same number of seats.
l. The ticket prices for the different categories are fixed.
m. The movie timings are fixed once posted onto the application.

3. ***Approach Taken***

In order to build an efficient and reliable MOBLIMA application system, we broke the application down into smaller parts and started working up from there onwards.

First, a database was built up to be used by the application. All movies including - movie details such as cast, rating, reviews, stars etc, types of tickets and their pricing, dates of holidays, and customer details including - the reference number, first and last name, email address etc, were stored in text files to be used by our application and for easy access as well. Staff login details were stored in a .csv file too.

Secondly, for each aspect of the cinema, a Manager Class was created with specific methods relating to that specific function of the cinema. For instance, a CinemaManager Class was created with the functionality of creating different cineplexes, deleting cineplexes, editing cineplexes, creating and deleting cinemas and editing the seating of each cinema. A TicketManager was created with the functionality of creating different categories of tickets, setting ticket prices, changing ticket prices and viewing all tickets. Likewise, BookingManager, CustomerManager, HolidayManager, MovieManager, SettingsManager, ShowtimeManager, StaffManager and TransactionManager were all created with specific tasks and functions pertaining to each class.

Lastly, we linked all these managers with their related packages and classes to build up the MOBLIMA Application and developed a MainApp for staff and customers to access different parts of the application required by them.

4. ***MOBLIMA Prototype Video***

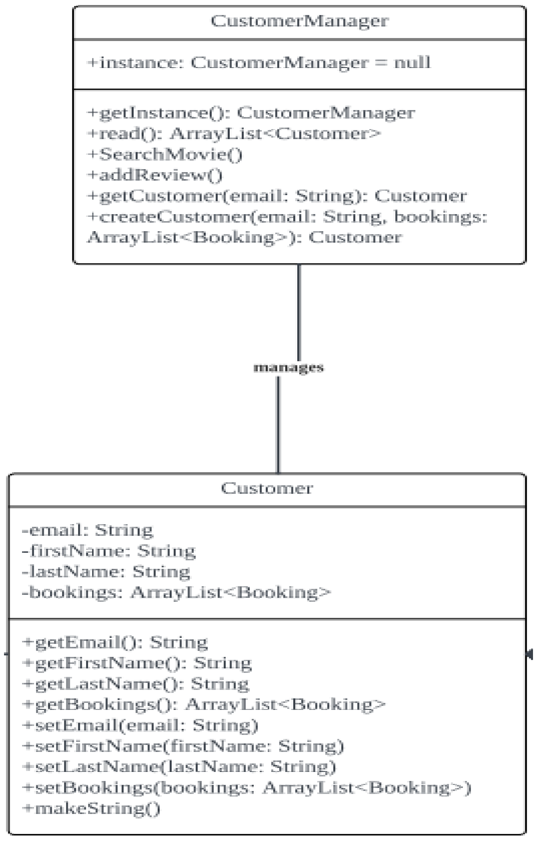
https://www.youtube.com/watch?v=5c_rKAbEkwU&feature=youtu.be

This section is created in case the size file for the video is too large to be submitted through NTULearn.

5. Design Considerations

5.1 OO Concepts

5.1.1 Encapsulation

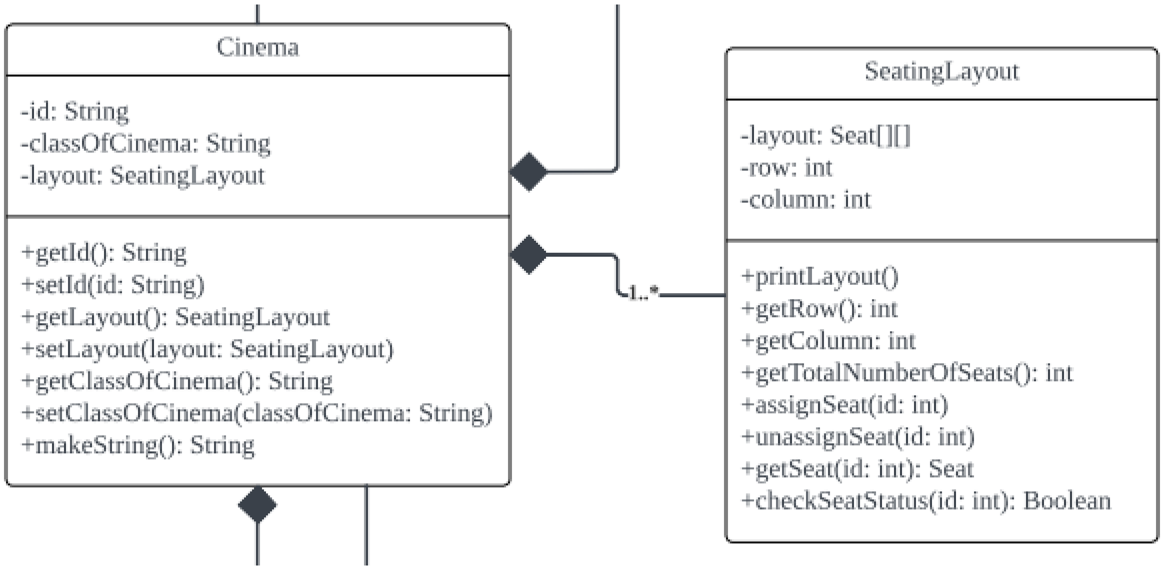

CustomerManager
+instance: CustomerManager = null
+getInstance(): CustomerManager
+read(): ArrayList<Customer>
+SearchMovie()
+addReview()
+getCustomer(email: String): Customer
+createCustomer(email: String, bookings: ArrayList<Booking>): Customer
manages
Customer
-email: String
-firstName: String
-lastName: String
-bookings: ArrayList<Booking>
+getEmail(): String
+getFirstName(): String
+getLastName(): String
+getBookings(): ArrayList<Booking>
+setEmail(email: String)
+setFirstName(firstName: String)
+setLastName(lastName: String)
+setBookings(bookings: ArrayList<Booking>)
+makeString()

Encapsulation in OOP, is the concept of binding fields (object state) and methods (behaviour) together as a single unit.

Encapsulation is also restricting direct access to certain components of an object so that users can’t access the state values for all variables of a particular object. Therefore, encapsulation can be used to hide data members and functions associated with an instantiated class or object.

For example, the CustomerManager and Customer class. The CustomerManager will not be able to access the Customer’s class’ attributes, such as email, firstName, lastName and booking directly. Using the get and set methods implemented in the Customer Class, allows CustomerManager to retrieve the private variables of the Customer object. This protects the data from unwanted access from the client.

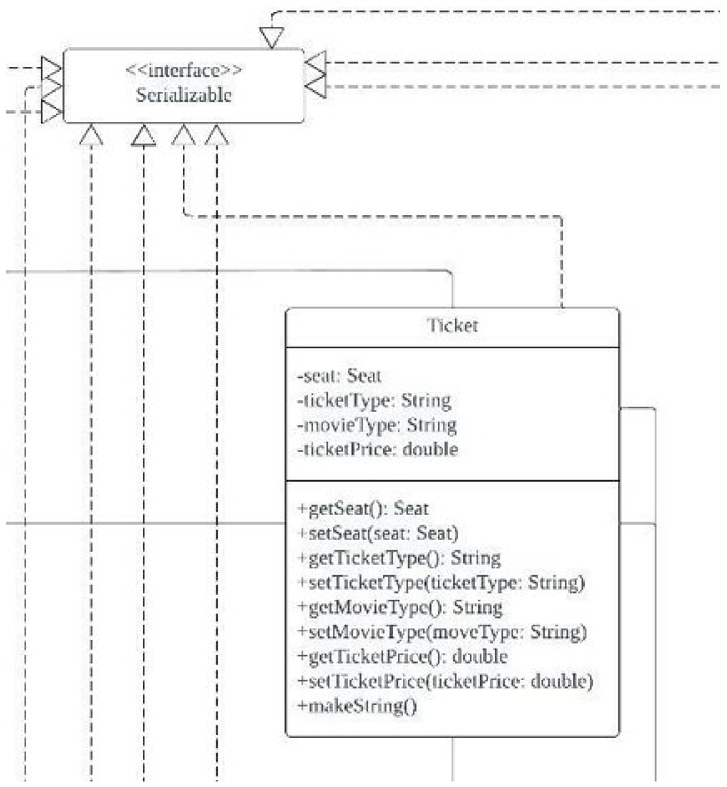
5.1.2 Abstraction



Abstraction refers to hiding unnecessary information and showing only relevant and necessary details to external objects or users, reducing the complexity of the problem. Hence, an external user does not need to understand the implementation but still makes use of complex logic inside an object. For example, in Cinema's relationship with SeatingLayout, Cinema only requires the data essential for getLayout. Furthermore, when calling printLayout, the users can only see the seating layout of that particular cinema and if the seats have already been assigned or not. All the other irrelevant implementation details of Seat and SeatingLayout are abstracted away. Abstraction has the advantage of reducing complexity, avoiding code duplication, and improving the usability of data.

5.1.3 Inheritance

As a result of inheritance, we can create child classes that have attributes and methods inherited from their parents. In terms of an interface, a class can implement an interface which could also be considered an inheritance.`

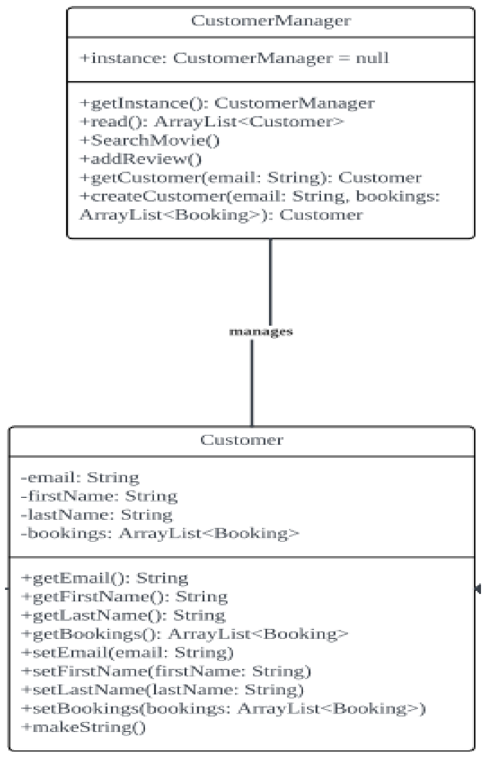


For example, the interface Serializable, the function of this interface was to read and write to text files in our database. Since the ticket class had to retrieve data from the database and write to the database when setting the TicketType, Ticket class implemented the Serializable interface. This allows reusability and prevents extensive duplication of code.

5.2 OO Principles

5.2.1 Single Responsibility Principle

In the Single Responsibility Principle, each class has only one responsibility and does not have more than one objective. Each class has specific responsibilities dedicated to its specific functionality. For example, CustomerManager and Customer class.


CustomerManager
+instance: CustomerManager = null
+getInstance(): CustomerManager
+read(): ArrayList<Customer>
+SearchMovie()
+addReview()
+getCustomer(email: String): Customer
+createCustomer(email: String, bookings: ArrayList<Booking>): Customer
manages
Customer
-email: String
-firstName: String
-lastName: String
-bookings: ArrayList<Booking>
+getEmail(): String
+getFirstName(): String
+getLastName(): String
+getBookings(): ArrayList<Booking>
+setEmail(email: String)
+setFirstName(firstName: String)
+setLastName(lastName: String)
+setBookings(bookings: ArrayList<Booking>)
+makeString()

In CustomerManager class, it contains methods to handle customer-related functionality such as creating customers, adding reviews and searching for movies while in the Customer class its sole responsibility is to contain its attributes. Thus, this would also mean that it has low coupling. In the event that CustomerManager is modified, it would not affect the Customer class. The sole reason that this is possible is that both classes have specific and different responsibilities that will not affect each other.

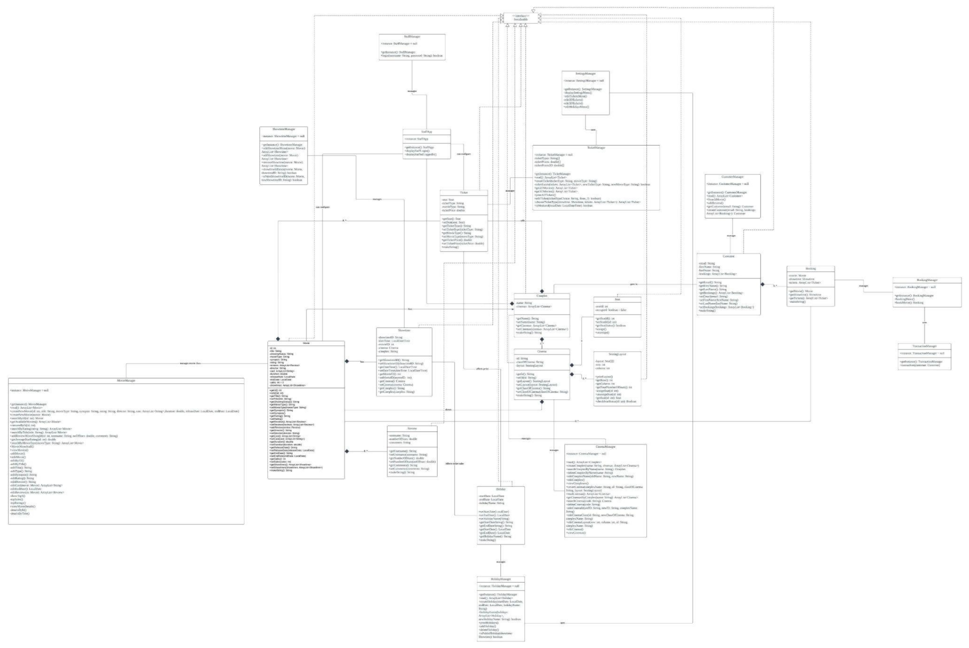
5.2.2 Interface Segregation Principle

The principle embodies the idea that having interfaces and methods that are more client-specific may be better than general ones since a client should not be implementing any interfaces or methods that it does not require. We avoided creating a general-purpose application and instead, created two application classes that are used for two different purposes. One for the customer and one for the staff.

5.2.3 Dependency Inversion Principle

According to the Dependency Inversion Principle, classes should depend on abstraction instead of concretion, so that high-level modules can be reused without being affected by low-level module changes. Our application consists of a MainApp which is the main interface with the user of the MOBLIMA App and contains all the boundary classes of other modules. Since the MainApp is used to interface with the users and is only able to receive the input, it does not know how the input will be used or processed by the control classes. This acts as a layer of abstraction between the classes. Moreover, some of the Classes have their own app (Customer has CustomerApp, Staff has StaffApp) and they too include all the boundary classes of other modules which further supports the Dependency Inversion Principle. Both of these App's are then implemented into the MainApp. Thus, any changes in the low-level modules will not affect the high-level modules at all

6. UML Class Diagram



*Separate .jpeg file attached externally

<u>7. Two New Features</u>

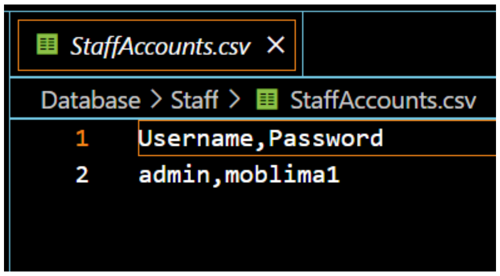
The first feature that could be implemented is allowing the customer to create a customer account for easier booking of movies as their data would already be stored in the system and only be available to them after successful login. As such, when booking the tickets, their information would already be filled up for them and only require confirmation by the user thus saving the user time. Since our code follows the SOLID principle as discussed above, we will easily be able to add new functions to create a customer account and login system without having to modify the entire program code as each function is independent of the other. We can make use of the CustomerApp which has already been implemented to get the user's information from the system using the MainApp. Similar to the StaffApp, the customer will have to log in to the system before they can access the features of MOBLIMA and their details could be stored in a .csv file similar to that of the StaffAccounts. As such, all we will need to do is to implement a login function or method in the CustomerManager Class and implement it into the CustomerApp thus adding this new functionality to our application.

The second feature that could be implemented is allowing the customer to pre-purchase movie snacks such as popcorn and soft drinks after they have selected which movie they want to watch and their desired seating to save them the time of queueing when they arrive at the cinema to watch their movie. Since our code follows the SOLID principle as discussed above, we will easily be able to add new functions to purchase movie snacks without having to modify the entire program code as each function is independent of the other. All that needs to be done to implement this feature is to implement a SnacksManager to show the user what snacks are currently available for pre-purchase and allow the user to purchase snacks in the BookingManager. We can then use the TransactionManager which has already been implemented to link the purchase of the snacks to the movie booking reference number which can be shown at the cineplex for collection as each customer would get a unique reference number linked to their purchase.

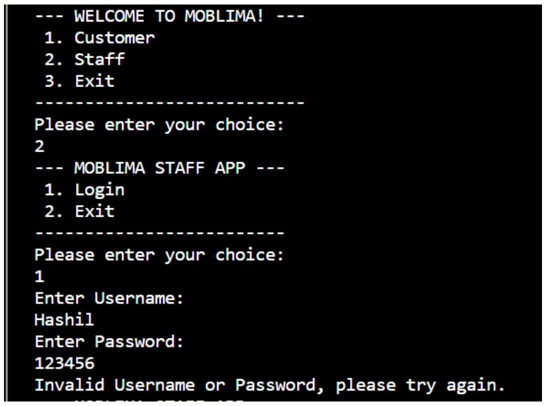
<u>8. Additional Testing</u>

<u>8.1 Staff members without valid or approved accounts are unable to access the MOBLIMA Staff App.</u>

The MOBLIMA staff app authenticates staff login by comparing the staff members' input with records in our database. If there is a match, the staff member will be able to log in.



```
Username,Password
admin,moblima1
```

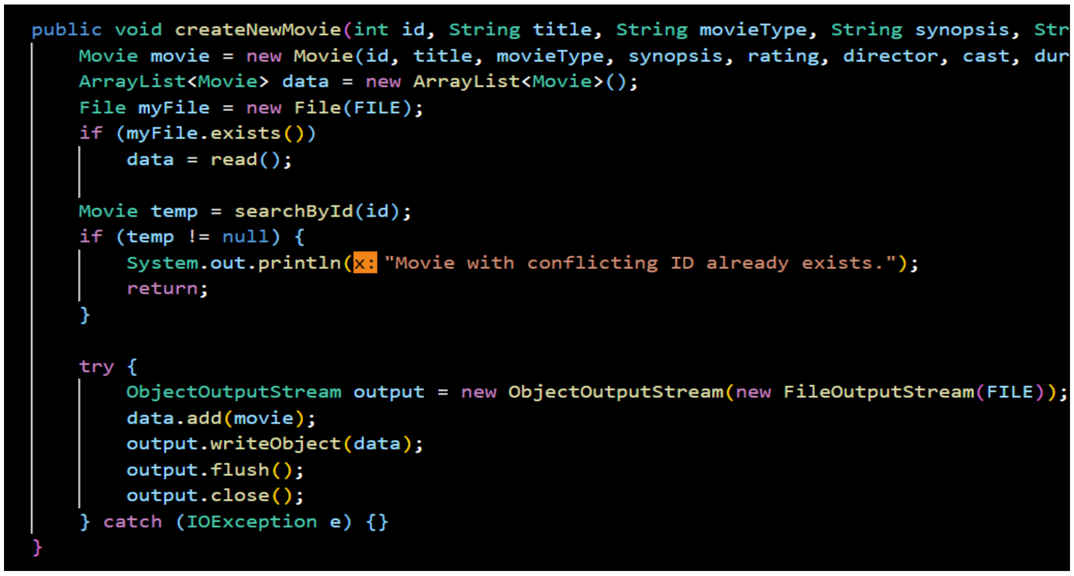


```
--- WELCOME TO MOBLIMA! ---
 1. Customer
 2. Staff
 3. Exit
---------------------------
Please enter your choice:
2
--- MOBLIMA STAFF APP ---
 1. Login
 2. Exit
-------------------------
Please enter your choice:
1
Enter Username:
Hashil
Enter Password:
123456
Invalid Username or Password, please try again.
```

Diagram 8.1 .csv file with approved staff accounts and unsuccessful login

8.2 Prevention of loss of data

A staff member may add new movies to the app as new movies are released. We do not want to lose this new movie data when the staff member logs out of the application. Thus, we have made use of Serialization in Java, which lets us convert the state of our movie object into a byte stream, thus enabling us to write to a data file when the app goes offline. Data read from the file is then deserialised for us to get back the movie object when a staff member logs in to the application thereby retrieving back this new movie data. The same has been done for many other classes in our MOBLIMA App.



```
public void createNewMovie(int id, String title, String movieType, String synopsis, Str
    Movie movie = new Movie(id, title, movieType, synopsis, rating, director, cast, dur
    ArrayList<Movie> data = new ArrayList<Movie>();
    File myFile = new File(FILE);
    if (myFile.exists())
        data = read();

    Movie temp = searchById(id);
    if (temp != null) {
        System.out.println(x: "Movie with conflicting ID already exists.");
        return;
    }

    try {
        ObjectOutputStream output = new ObjectOutputStream(new FileOutputStream(FILE));
        data.add(movie);
        output.writeObject(data);
        output.flush();
        output.close();
    } catch (IOException e) {}
}
```

Diagram 8.2 Serialization