SC2002 : OBJECT ORIENTED DESIGN AND PROGRAMMING
AY22/23 SEMESTER 1 GROUP ASSIGNMENT

MOvie Booking and LIsting Management Application (MOBLIMA)
DATE OF SUBMISSION : 13 NOVEMBER 2022

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	Lab Group	Signature/Date
Mishra Apurva (U2120474C)	SC2002	SSP1	Apurva 13/11/2022
Gambhir Dhruv (U2120075F)	SC2002	SSP1	Dhruv 13/11/2022
Jadhav Chaitanya Dhananjay (U2121503D)	SC2002	SSP1	Chaitanya 13/11/2022
Najah Ismail (U2120555F)	SC2002	SSP1	Najah 13/11/2022
Ng Tze Kean (U2121193J)	SC2002	SSP1	Tze Kean 13/11/2022

Table of Contents

1. Design Considerations 3

1.1 Approach 3

1.2 Assumptions 3

1.3.1 SOLID Design Principles 3

Single Responsibility Principle 3

Open Closed Principle 4

Liskov Substitution Principle 4

Interface Segregation Principle 4

Dependency Injection Principle 5

1.3.2 Additional Design Principles 5

Singleton Implementation 5

Enumeration-Based State Tracking 6

1.4 Object-Oriented Programming Principles 6

Abstraction 6

Encapsulation 6

Polymorphism 6

1.5 Proposed New Features 6

Privileged Staff and Customer Accounts: 7

2. UML Class Diagram 7

2.1 Explanation 7

3. Test Case Demonstration 8

1. Design Considerations

1.1 Approach

The MOBLIMA system implements a movie booking and management system for two users: Movie Staff and Movie Goers. The system implementation is split into high and low-level packages. High level packages are the view classes, which provide UI for Staff and Customer to interact with the system. Low level packages are object entities which are supposed to mirror real-world objects which the user can interact with. Low-level objects act like a model, and Managers store an array of these models and manage the interaction of the high-level classes with them. Managers also enforce error checking and exception catching to ensure that models are being manipulated as designed. Our system was designed with classes within each package to be encapsulated from each other. Interaction between classes is handled by interfaces to reduce the overall coupling in the system.

1.2 Assumptions

The following are the assumptions made while designing the system:

1. Staff accounts have fixed usernames and passwords.
2. Every movie has a base price which is manipulated by:
    i. Customer Attributes - if they are a Student / Child / Senior Citizen / have discount
    ii. Showtime Attributes - if the showtime is scheduled on a holiday/cinema type
3. Couple Seats are only available in the last row C of Platinum Cinema Halls.
4. Elite Seats pricing is the same as normal seats to prevent discrimination but comfortable
5. Users are allowed to make anonymous reviews for movies on the platform.
6. Users can anonymously browse the website until the booking payment stage, after which they are required to enter their details for instantiating the booking object and calculating price.
7. Only one seat can be booked at a time.
8. Customers above the age of 65 are automatically given senior citizen discounts.
9. Customers under the age of 12 are automatically given a child discount.
10. Customers can view details about Coming Soon movies but are unable to select them for booking.
11. Staff can view and access Movie and Cineplex objects via their unique ID, while Customers can only interact with them by specifying their String values. This prevents Customers from having access to details about how objects are stored in our Manager Classes.

1.3 Design Principles

1.3.1 SOLID Design Principles

Single Responsibility Principle

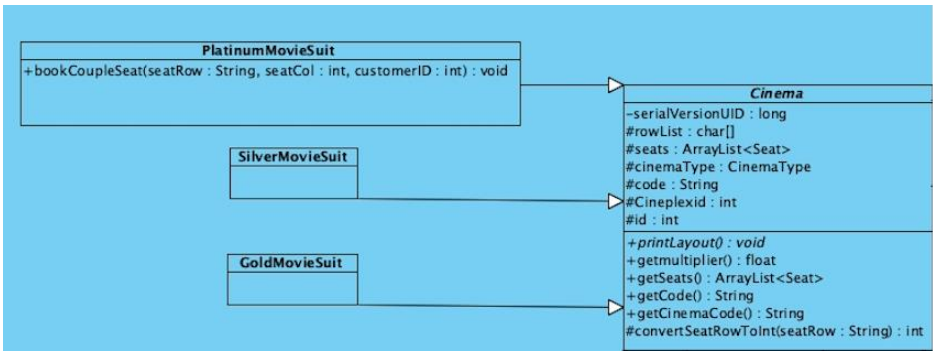
Each package has model classes which perform a single responsibility and composited classes encapsulate internal implementations via API function calls. Manager classes are also implemented with the responsibility of managing the model classes composited within them.

For example, within the Movie Package, the Movie class is used to store the relevant attributes and enumeration values. The MovieManager class implements the required functions to get and set values for the Movie objects. The movie interfaces call functions in MovieManager to enable functionality for the various actors' requirements.

Open Closed Principle

Open Closed Principle as described by Meyer means that the entities created including classes, functions and modules should be unmodifiable but extendable i.e., new functions are added without changing existing ones.

For instance, Cinema classes Platinum, Gold and Silver have different functionalities (an example is their printLayout function may be different). We declare the Cinema class as an abstract class and close it to modification. We open the class for extension to allow the subclasses to inherit Cinema class for method overriding and new feature implementation. Similar implementations can be seen in the other packages.



Furthermore, all our views inherit from the main View class, which must contain a printMenu() and start() method. The content of the parent View class is unmodifiable but easily extensible as additional views can be easily created for customers and staff by implementing these functions.

Liskov Substitution Principle

Liskov Substitution Principle (LSP) states that objects of a superclass should be replaceable with objects of its subclasses without breaking the application. This is primarily used in the View and Cinema classes.

In MOBLIMA we substituted the Cinema class for specialized Movie Suites which perform the same error checking and functionality. These specialized classes were upcasted as Cinema Type in MovieManager, but do not break the design by contract in their overridden functions.

Interface Segregation Principle

The Interface Segregation Principle specifies that specific interfaces are better than a general interface. Interfaces in MOBLIMA have been segregated to suit the abstraction of the classes and managers from the view class to ensure that the view class only uses the functions provided in the interface.

For example, customers use the IMovie and IReview interface to interact with the movie objects, but do not have access to the ISales methods to perform staff activities. This increases the security of the system and eliminates unintended misuse. A similar design structure is implemented through all interfaces in our packages.

Another example is the IShowtime interface and IShowtimeSystem interface. The IShowtimeSystem interface provides more system admin privilege functions which should not be accessed by normal users. The IShowtime interface provides basic access to showtimes, which will be used by the movie-goer. Since Staff should be also able to perform the actions of a moviegoer, IShowtimeSystem extends IShowtime interface. This increases the security of the system and avoids misuse.

Dependency Injection Principle

Dependency Injection Principle is a design philosophy in which an object or function can receive other objects or functions that it depends on. Dependency Injection Principle states that high-level modules should not depend on low-level modules but instead on abstraction. The Client, Customers and Staff are provided dependencies through external code that it is not aware of.

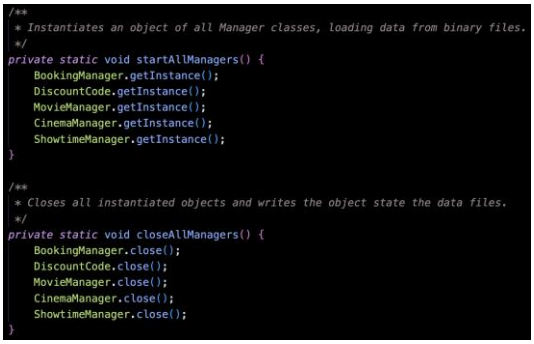
This is made possible by creating an interface between the high-level module and the low-level module. An example in MOBLIMA is the view package classes using interfaces to perform actions on the Manager classes. This implies that the view classes do not know the actual implementation of the Managers nor the other functions available in the class, providing the abstraction from the low-level module. The low-level module will also not know how these functions are going to be used, but only know that such a functionality is needed.

This is illustrated in the Movie Package. For instance, the StaffMovie class, which handles Movie related admin functions, uses the declared IMovie interface to interact with the Managers through abstraction. StaffMovie is hence able to call all relevant methods to modify movie data. However, because of our interface instantiation, the StaffMovie class is only concerned with the existence of the function and not about the methodology behind how the corresponding Movie object is manipulated.

This allows for the specific details in each class to be implemented based on abstraction, making the high and low-level modules more flexible and reducing the coupling between each class.

1.3.2 Additional Design Principles

Singleton Implementation



```
/**
 * Instantiates an object of all Manager classes, loading data from binary files.
 */
private static void startAllManagers() {
    BookingManager.getInstance();
    DiscountCode.getInstance();
    MovieManager.getInstance();
    CinemaManager.getInstance();
    ShowtimeManager.getInstance();
}

/**
 * Closes all instantiated objects and writes the object state the data files.
 */
private static void closeAllManagers() {
    BookingManager.close();
    DiscountCode.close();
    MovieManager.close();
    CinemaManager.close();
    ShowtimeManager.close();
}
```

Singleton design pattern has also been considered for each manager class to allow for only a single instance of a Manager such that there will not be duplicated incorrect data or concurrent access to the information by multiple classes which may lead to data inconsistency. This is made possible through our getInstance() method, which returns a static object of the Manager class serialized with our pre-loaded data. At the end of MOBLIMA running, the close() method is called and changes to object states are written into our .dat files to guarantee persistence.

Enumeration-Based State Tracking

Enums were also used in our application to enforce a state within the function. This allowed users to backtrack to different states and allows for easy debugging and code readability.

```
private static void createCineplex() {
    enum createCineplexState {
        NAME, LOCATION, CREATE
    }
    ;
    ICineplex cineplexHandler = CineplexManager.getInstance();
    createCineplexState state = createCineplexState.NAME;
    String name = null;
    String location = null;
    boolean complete = false;
    sc = new Scanner(System.in);

    System.out.print(s: "\033[H\033[2J");
    System.out.println(x: "-------------------------------------");
    System.out.println(x: "\tCreating Cineplex");
    System.out.println(x: "-------------------------------------");

    while (!complete) {
        switch (state) {
            case NAME: ···
            case LOCATION: ···
            case CREATE: ···
        }
    }
    System.out.println(x: "-------------------------------------");
    System.out.println(x: "\tNew Cineplex Created");
    System.out.println(x: "-------------------------------------");
    waitForEnter(message: null);
}
```

1.4 Object-Oriented Programming Principles

Abstraction

The Interfaces access only the required functions, abstracting all the details and method implementations not required by the actors. This maintains simplicity in interacting with the methods and increases the security of the system.

Encapsulation

Encapsulation is used to promote data protection. The data is accessible and modifiable only through the respective get and set functions within the Manger classes while all member attributes are declared private, restricting direct access. Additionally, the constructor of the Manager classes is set to private and only accessible through the getInstance() function. This is to ensure that only one object of the Managers exists in the respective functions.

Polymorphism

In the IShowtime Interface, the getPrice() function is overloaded taking into account the presence/absence of a Discount Code. The respective prices can be retrieved effectively by method overloading. A similar implementation is seen in the printCinemaLayout() function.

1.5 Proposed New Features

Booking Cancellation:

We propose the addition of a feature called "Cancel Booking" that allows the customer to cancel their booking 5 days before the showtime. This functionality can also be extended to allow staff to cancel bookings at any time to deny entry for any valid reason such as Cinema Maintenance, etc.

Implementation:

A function to cancel the booking can be added to the IShowtime interface allowing the customer view class to utilize this function. As for the low-level module, the only change needed is to add one new function which checks if the booking being cancelled is valid five days before the showtime. The overall change to code is minimal as DIP was applied before and coupling between classes is low, thus a change to the system does not drastically affect other classes.

Privileged Staff and Customer Accounts:

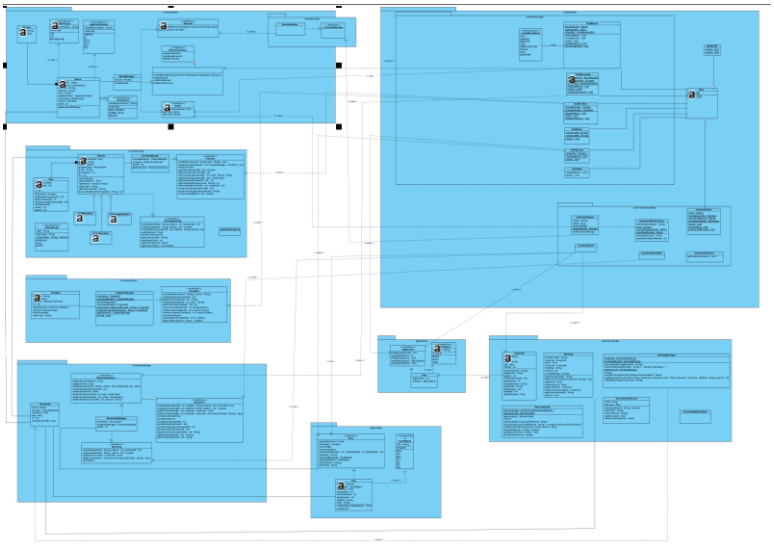
We propose the creation of customers and staff accounts with varying abilities and privileges. Currently, customers do not have accounts that they can log in to view personalized details about themselves. With account creation, customers must first log in with their email and password to view their purchase history, which provides more security and privacy. Once logged in, customers can also access more personalized features, such as booking cancellation as mentioned above.

Implementation:

Currently booking only takes in the Customer details without allowing the customer to create an account. An additional class called CustomerAccount can be created which has a composite Customer object within itself and has 2 other attributes called Username and Password. This allows customers to log in before making a booking or to view booking history. That way, the booking history will display entries booked by a specific account and prevent anonymous users from viewing other customer booking history.

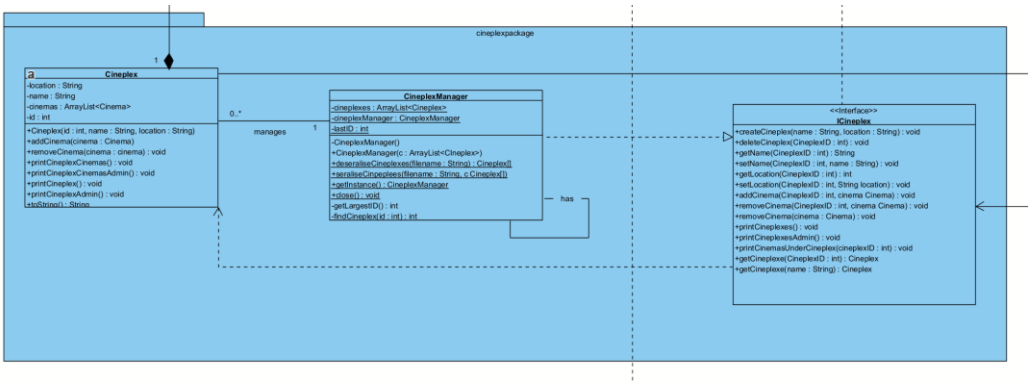
A login class can be implemented similar to StaffAuth, while a CustomerLoginView can be created to handle operations for logged-in customers. Lastly, an attribute "customerUsername" can also be added to the Review class to display the "reviewer".

2. UML Class Diagram



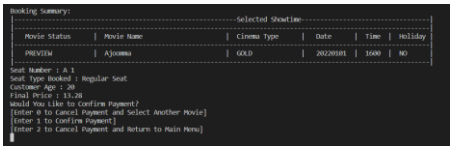
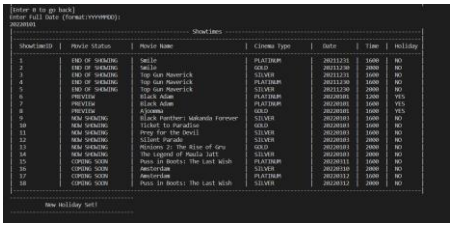
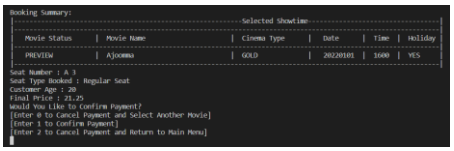
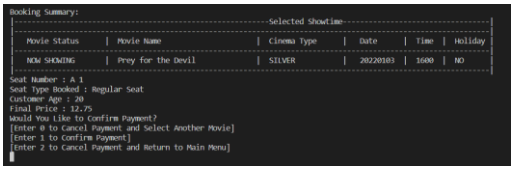
The .vpp and .jpg file for our MOBLIMA implementation are attached along with this report.

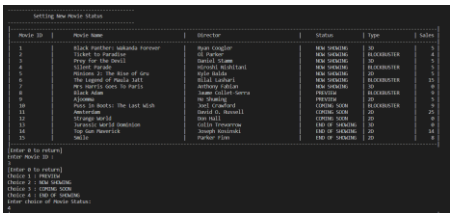
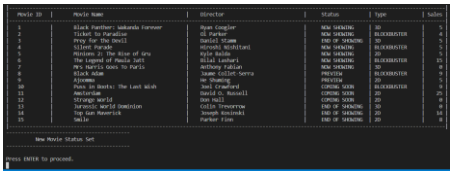
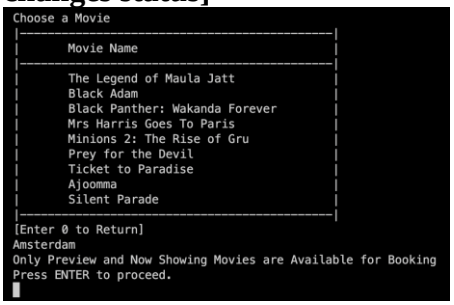
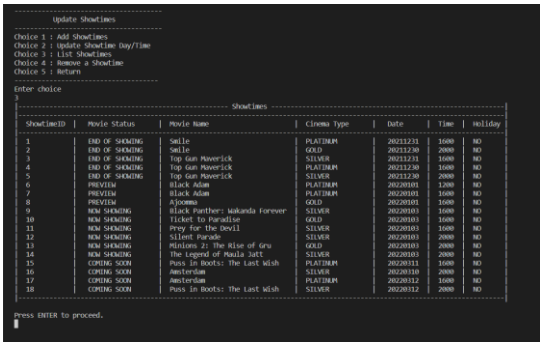
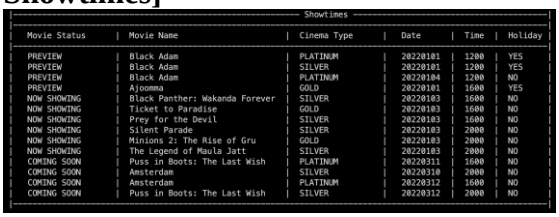
2.1 Explanation

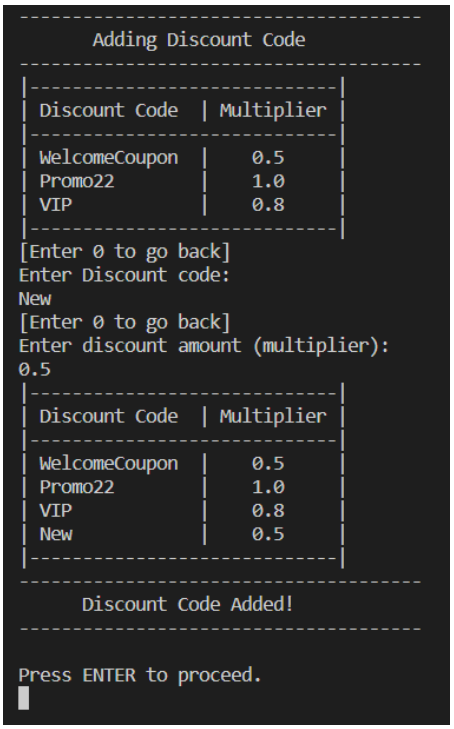
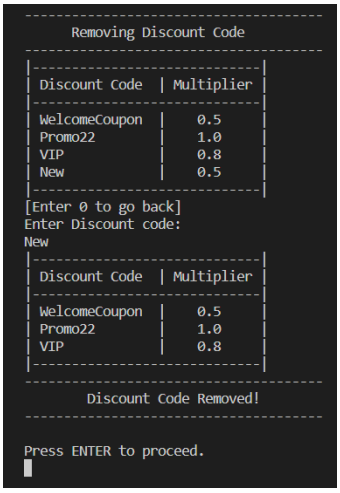


Packages are designed with a model class (Cineplex) and a manager (CineplexManager) in mind. Each package would also have an interface (ICineplex) which the Manager implements for other classes to interact with the cineplex object. Overall, this design allows for appropriate encapsulation for each object entity for other packages. Another consideration while designing the UML is the amount of coupling between classes. To reduce coupling, interfaces were created to reduce the relationship from association to dependency.

3. Test Case Demonstration

Test Case	Expected Outcome	Result
1. Configuring a holiday date and the ticket price is shown correctly when booking is done on that date.	The staff can view the available showtimes and set one of the dates as through the "Set Holiday" choice. **[Exception Handling]** On entering, Invalid Dates where showtimes don't exist or invalid format, an error message is displayed, and Staff are redirected to the Configure System menu. **[Back Tracking]** During Scanning, entering "0" allows the user to go back to the Update Showtime Menu. On entering a valid Date, it is set as a holiday and an updated Showtimes List is displayed. When booking the change in the price for that day is observed. Price calculation takes weekends and holidays into account with the use of multipliers. Regular Day: 1x Weekend: 1.25x Holiday: 1.5x Weekend AND Holiday: 2x	 **[Movie Price before Holiday Set = 13.28]**  **[20220101 Set as Holiday]**  **[Movie Price after Holiday is set = 21.25]**
2. Configuring the "End of Showing" date and the movie should not be listed for booking	All the existing Movie Listing details including MovieID, Movie name, Type, Status, Director and Sales are printed. **[Exception Handling]** Invalid Movie IDs and InputMismatchError are handled by returning to the Update Movie Menu. **[Back Tracking]**	 **[Customer is able to book Prey for the Devil before setting END_OF_SHOWING]**

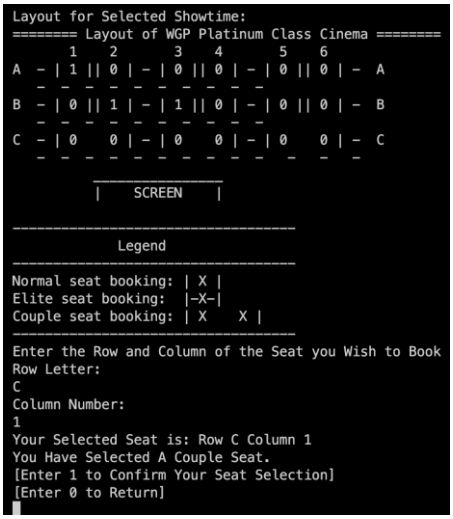
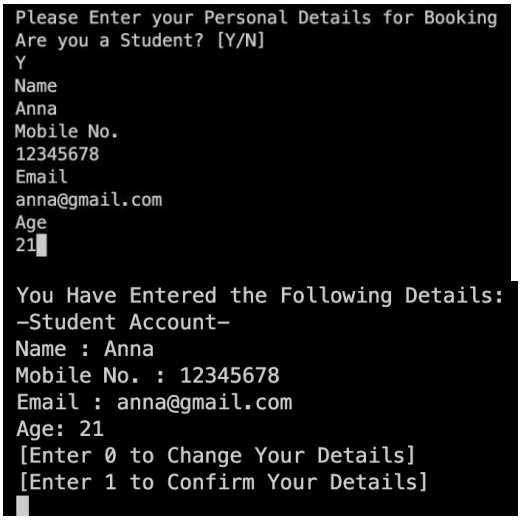
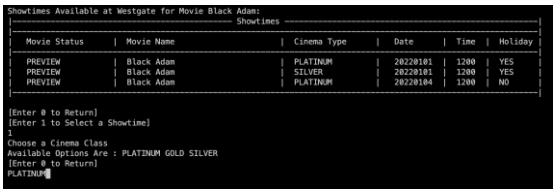
	During Scanning entering “0” allows the user to go to the previous entry to change their input. Entering “0” at the first variable i.e. Movie ID allows the user to go back to the Update Movie menu. The Enum MovieStatus is entered through integer choice 1-4. Invalid inputs are handled by allowing re-entry. When a valid MovieID is entered, the new Movie list with the changed Movie Status is printed as END_OF_SHOWING. When booking, the movie is no longer displayed in the available options to the customer. **[Exception Handling]** If the customer enters the name of a movie that is COMING SOON or END OF SHOWING, they are given an error message and returned to the select movie menu.	  **[The Movie Status change to END OF SHOWING is reflected when Staff changes status]**  **[“Amsterdam”, a COMING SOON movie is not an accepted input for Movie Selection while Booking.]**
3. Bookings only allowed for NOW SHOWING and PREVIEW status.	All the available showtimes are visible through the Staff Showtime Menu. When making a booking, movies with status COMING SOON or END OF SHOWING are not visible and thus cannot be booked. If a user tries to book it regardless, an exception is thrown, and movie has to be chosen again.	 **[Staff Showtime Menu can list all Showtimes]**  **[Customer Showtime Menu can view all PREVIEW, NOW SHOWING, and COMING SOON Showtimes]**

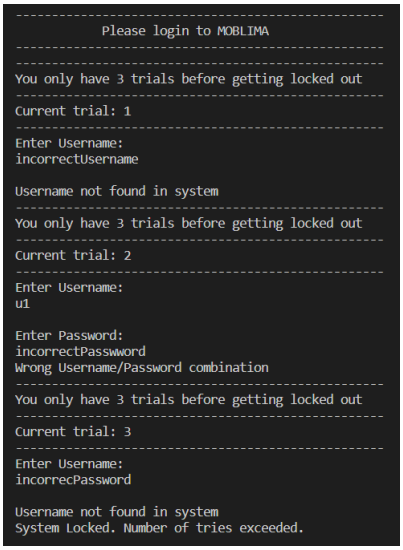
4. Discount Code: Creation, Deletion and Usage	When booking with a discount code, the discounted price is calculated according to the respective multiplier. The staff also has the privileges to add and remove discount codes as well as change the values of the respective multiplier. Add & Remove Discount Code is implemented in the Configure System Menu. **[Add Discount Code]** Prints all the existing Discount Codes and Multipliers. Scans new Code and Multiplier to add. **[Remove Discount Code]** Prints all the existing Discount Codes and Multipliers. Scans Code to remove Backtracking for reentry and InputMismatchException handling is also implemented for both classes. **[Usage]** After the check for isValidDiscountCode, the price is reduced to a fraction of the original price based on the multiplier corresponding to the discount code. For code VIP, a 0.8 multiplier results in a 20% price discount.	 **[Adding Discount Code by Staff]**  **[Remove Discount Code by Staff]**  **[Successful use of Discount Code by Customer]**

```
---------------------------------------
        Adding Discount Code
---------------------------------------
|-----------------------------|
| Discount Code  | Multiplier |
|-----------------------------|
| WelcomeCoupon  |    0.5     |
| Promo22        |    1.0     |
| VIP            |    0.8     |
|-----------------------------|
[Enter 0 to go back]
Enter Discount code:
New
[Enter 0 to go back]
Enter discount amount (multiplier):
0.5
|-----------------------------|
| Discount Code  | Multiplier |
|-----------------------------|
| WelcomeCoupon  |    0.5     |
| Promo22        |    1.0     |
| VIP            |    0.8     |
| New            |    0.5     |
|-----------------------------|
---------------------------------------
      Discount Code Added!
---------------------------------------

Press ENTER to proceed.
```

```
---------------------------------------
      Removing Discount Code
---------------------------------------
|-----------------------------|
| Discount Code  | Multiplier |
|-----------------------------|
| WelcomeCoupon  |    0.5     |
| Promo22        |    1.0     |
| VIP            |    0.8     |
| New            |    0.5     |
|-----------------------------|
[Enter 0 to go back]
Enter Discount code:
New
|-----------------------------|
| Discount Code  | Multiplier |
|-----------------------------|
| WelcomeCoupon  |    0.5     |
| Promo22        |    1.0     |
| VIP            |    0.8     |
|-----------------------------|
---------------------------------------
       Discount Code Removed!
---------------------------------------

Press ENTER to proceed.
```

```
Booking Summary:
|-------------------------------------------------------Selected Showtime-------------------------------------------------------|
|-------------------------------------------------------------------------------------------------------------------------------|
|  Movie Status  |  Movie Name  |  Cinema Type  |  Date  |  Time  |  Holiday  |
|-------------------------------------------------------------------------------------------------------------------------------|
|  PREVIEW  |  Black Adam  |  PLATINUM  |  20220101  |  1200  |  YES  |
|-------------------------------------------------------------------------------------------------------------------------------|
Cineplex Name : Westgate
Seat Number : C 1
Seat Type Booked : Couple Seat
Customer Age : 21
Final Price : 16.07
Discount Code Used : Yes
Student Pricing Used : Yes

Would You Like to Confirm Payment?
[Enter 0 to Cancel Payment and Select Another Movie]
[Enter 1 to Confirm Payment]
[Enter 2 to Cancel Payment and Return to Main Menu]
```

5. Couple Seat Booking	In state **SELECTSEAT** of the CustomerBook menu, if row C is selected in **PLATINUM** cinema type, display: "You Have Selected A Couple Seat." Next, confirmation options shown: "[Enter 1 to Confirm Your Seat Selection]" "[Enter 0 to Return]" Variable customerCoupleSeat is set to true upon confirmation, which is used further for price calculation and booking seat.	
6. Booking for Different Customer types	In state **CUSTOMERDETAILS** of the customer book menu, User's response to "Are you a Student? [Y/N]" sets variable **customerIsStudent** to true or false accordingly. Prompt to enter details — Name (string), Mobile No. (int), Email (string), and lastly Age(int) — which is further converted to Age object that returns Age multiplier. Error checking is done at every step for invalid inputs Final Price becomes a multiple of the standard based on multiplier: Student: 0.8 Senior (age >=65): 0.5 Child (age <= 12): 0.7	
7. Booking for Different Cinema Types	In state SELECTCINEMATYPE of the customer book menu, the user is prompted to enter CinemaType for the showtime they have selected. Lists out options for CinemaType to choose from (PLATINUM, GOLD, SILVER.) Variable cinemaTypeInput is assigned. The following multipliers are used for price calculation: Silver: 1 Gold: 1.25 Platinum: 1.5	 User is prompted to enter the details of the showtime that they wish to book. **[Error Checking]** If customers enter an invalid cinema type, they are returned to the select showtime menu

8. Staff Auth-Locked out of System	By choosing the Staff option, the user is redirected to a login page before accessing the Staff Admin functions. There are **three** maximum login attempts before the user is locked from the system. An unsuccessful login attempt is characterized by: 1. Username that doesn't exist in the Database 2. Wrong password/username combination In case the user takes up all three login attempts the User is redirected back to the MOBLIMA menu.	 **[Example of System Lock]**

Additional test cases here:

Staff test cases: Staff test case Annex.pdf

Customer test cases: Customer test case Annex.pdf