NANYANG TECHNOLOGICAL UNIVERSITY

CZ2002 Object-Oriented Design and Programming

Moblima

Movie Booking and Listing Management System

SS5 | GROUP 6

Norman (U1820628C)

Sven (U1820068B)

Randy (U1821394D)

Swee Sen (U1821542D)

APPENDIX B:

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
Koh Swee Sen	CZ2002	SS5	[illegible] 17/11/19
Ng Yi Liang, Randy	CZ2002	SS5	[illegible] 17/11/19
Sven Tan Wei Jie	CZ2002	SS5	[illegible] 17/11/19
Norman Fung Zi Jie	CZ2002	SS5	[illegible] 17/11/19

Important notes:

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

Introduction

MOBLIMA is an application designed to facilitate bookings and purchases of Movie Tickets as well as administrative purposes for admin users.In this report , we will show various design considerations and principles as well as object-oriented concepts which we have implemented during the development of this application.Detailed UML Class diagram as well as UML Sequence Diagram are also included for viewing to further illustrate the flow of the application.

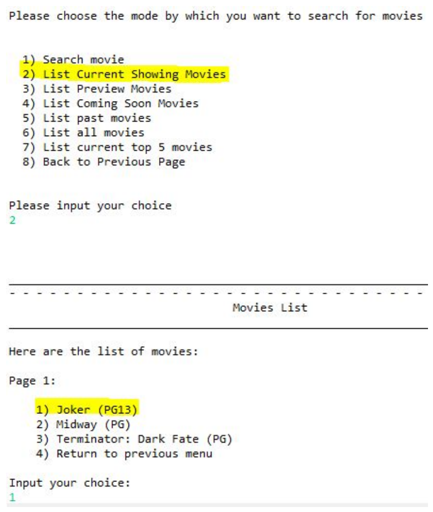
Essential Test Cases

Our video highlights the majority of the test cases, including:

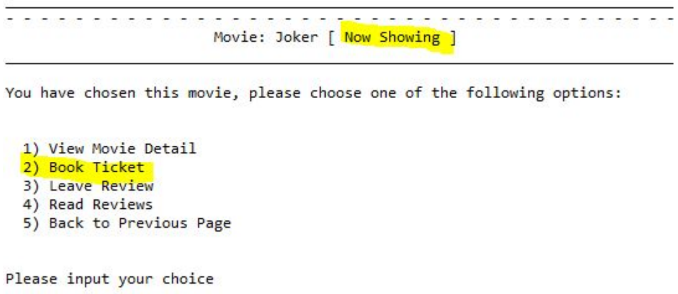
- Giving an admin the ability to set a holiday date, and ticket prices are updated when a booking is made on that date
- Having different ticket prices when bookings are made on a weekend compared to on a weekday, as well as when different cinema classes are chosen
- Changing the movie status to “End of Showing” and not allowing movie-goers to book tickets for that movie any longer.

Our demonstration also includes movie-goers booking tickets for movies that have the “Now Showing” status, and movies that have the “Coming Soon” status cannot be booked. However, for completeness’ sake, we will present the booking capabilities for movies of all statuses here.

<u>"Now Showing" movies</u>



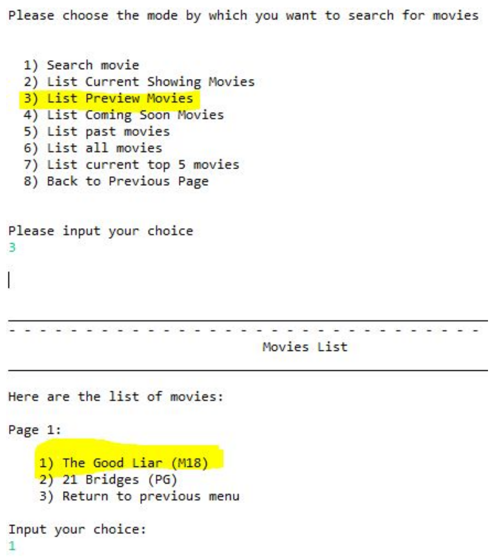
```
Please choose the mode by which you want to search for movies

  1) Search movie
  2) List Current Showing Movies
  3) List Preview Movies
  4) List Coming Soon Movies
  5) List past movies
  6) List all movies
  7) List current top 5 movies
  8) Back to Previous Page

Please input your choice
2

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
                         Movies List

Here are the list of movies:

Page 1:

    1) Joker (PG13)
    2) Midway (PG)
    3) Terminator: Dark Fate (PG)
    4) Return to previous menu

Input your choice:
1
```



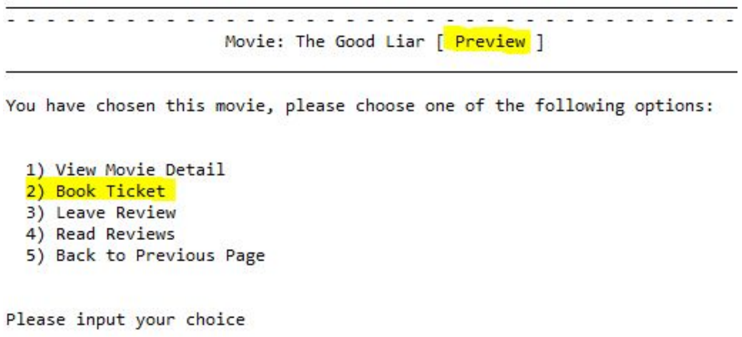
```
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
                    Movie: Joker [ Now Showing ]

You have chosen this movie, please choose one of the following options:

  1) View Movie Detail
  2) Book Ticket
  3) Leave Review
  4) Read Reviews
  5) Back to Previous Page

Please input your choice
```

Booking can be made for the movie "Joker" of status "Now Showing".

<u>"Preview" movies</u>



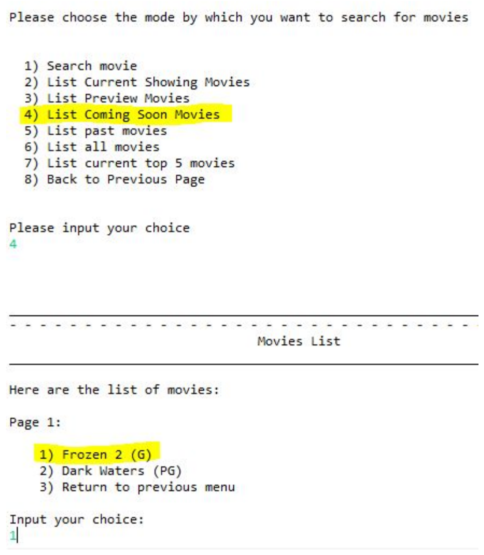
```
Please choose the mode by which you want to search for movies

  1) Search movie
  2) List Current Showing Movies
  3) List Preview Movies
  4) List Coming Soon Movies
  5) List past movies
  6) List all movies
  7) List current top 5 movies
  8) Back to Previous Page

Please input your choice
3

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
                         Movies List

Here are the list of movies:

Page 1:

    1) The Good Liar (M18)
    2) 21 Bridges (PG)
    3) Return to previous menu

Input your choice:
1
```



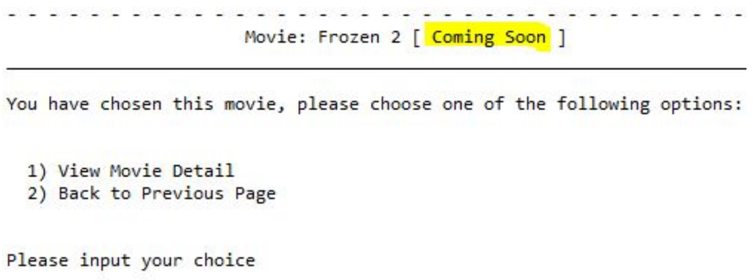
```
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
                 Movie: The Good Liar [ Preview ]

You have chosen this movie, please choose one of the following options:

  1) View Movie Detail
  2) Book Ticket
  3) Leave Review
  4) Read Reviews
  5) Back to Previous Page

Please input your choice
```

Booking can be made for the movie "The Good Liar" of status "Preview".

<u>“Coming Soon” movies</u>



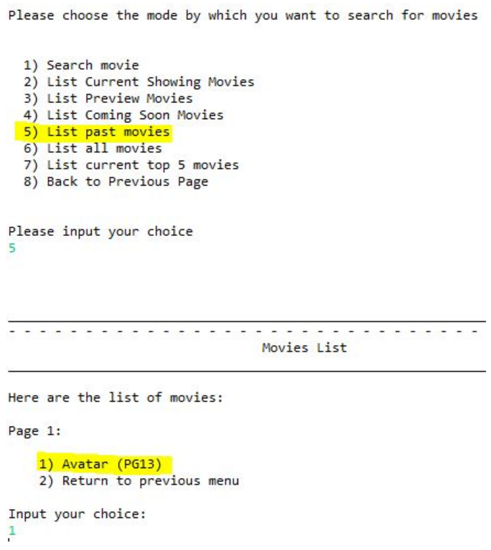
```
Please choose the mode by which you want to search for movies

  1) Search movie
  2) List Current Showing Movies
  3) List Preview Movies
  4) List Coming Soon Movies
  5) List past movies
  6) List all movies
  7) List current top 5 movies
  8) Back to Previous Page

Please input your choice
4

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
                         Movies List

Here are the list of movies:

Page 1:

    1) Frozen 2 (G)
    2) Dark Waters (PG)
    3) Return to previous menu

Input your choice:
1
```



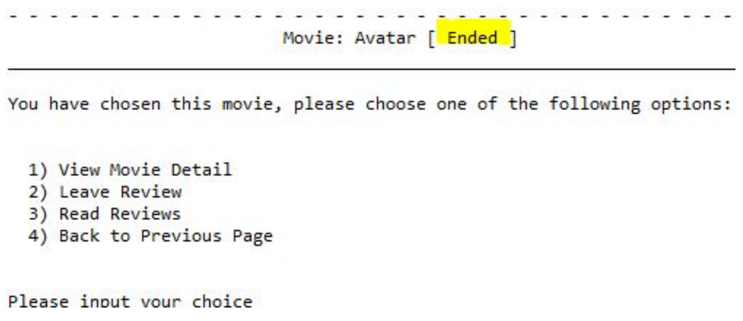
```
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
                 Movie: Frozen 2 [ Coming Soon ]

You have chosen this movie, please choose one of the following options:

  1) View Movie Detail
  2) Back to Previous Page

Please input your choice
```

As seen in the screenshot on the right, there is no “Book Ticket” option available for the movie “Frozen 2” of status “Coming Soon”.

<u>“End of Showing” movies</u>



```
Please choose the mode by which you want to search for movies

  1) Search movie
  2) List Current Showing Movies
  3) List Preview Movies
  4) List Coming Soon Movies
  5) List past movies
  6) List all movies
  7) List current top 5 movies
  8) Back to Previous Page

Please input your choice
5

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
                         Movies List

Here are the list of movies:

Page 1:

    1) Avatar (PG13)
    2) Return to previous menu

Input your choice:
1
```



```
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
                   Movie: Avatar [ Ended ]

You have chosen this movie, please choose one of the following options:

  1) View Movie Detail
  2) Leave Review
  3) Read Reviews
  4) Back to Previous Page

Please input your choice
```

As seen in the above screenshot, there is no “Book Ticket” option available for the movie “Avatar” of status “End of Showing”

Design Considerations

Design Patterns

1. MVC Design Pattern

MVC design pattern states that our classes in the source code should be organised into these three categories: data models, views which contains the presentation information, as well as controller which contains all the control classes.

- **Model:** We categorised our entity classes to be under the Model category. This includes classes like the Movie class and the Cineplex class.

- **View:** In our console application, each of the menus in the application is considered to be a view, and hence handled by one of the view classes. Hence in this View category, we have many classes such as the **ListMoviesView** and the **AdminConfigureSystemSettingsView**.

- **Controller:** We have a total of 4 control classes in our source code.

    1. **DatabaseManager**
    - This class is responsible for any interaction with our database (txt file). This class mainly consists of static methods which helps to assist in the process of saving data into the text files as well as retrieving data from the database.

    2. **IOManager**
    - This class is responsible for any interaction with the user of the application. It consists of static methods which mainly deals with either getting input from the user or output information onto the console screen. The static methods of this class also helps to handle the exceptions that might occur while getting input from the user, such as wrong input type.

3. **ViewsManager**

- This class is responsible for managing the flow of the different View classes during the application. Some processes that are handled by this class includes navigating to the next view, as well as going back to the previous views. This is done by maintaining a stack that contains the presented views. Going to the previous view is done simply by popping the current view from the views stack, while going to the next view is done simply by pushing new view onto the stack.

4. **PriceManager**

- This class is responsible for the logic of calculation of prices. This includes both the calculation of ticket cost as well as the calculation of an order made by the movie-goer. Factors of ticket and order price such as age, movie type, cinema type, day-of-week, and GST are taken into account to calculate the prices.

2. Singleton Design Pattern

- We use the singleton design pattern cautiously because if used incorrectly, it will lead to the increased coupling among the classes. This is because all the classes can access and modify the single shared instance of this class and classes may rely on this shared instance to pass data instead of using dependency injection, making it more difficult for testing and making the code less maintainable.

- However, we think that there is one part of our project in which we think singleton design pattern is suitable, and that is our **PriceConfiguration** class. This class stores all information that are related to price, such as the base price of a movie ticket depending on age, as well as the amount of price increment for gold class ticket. Since there is only a single instance of this class, it is ensured that there is only a single source of truth for the prices information and that the price information is always up to date. The admin module can make use of this shared instance to modify the price information, and the changes are immediately available for the movie-goer module, thus keeping the prices information in sync between the two modules.

Implementation of SOLID design principles

1. Single Responsibility Principle

- Single responsibility principle states that there should never be more than one reason for a class to change. To achieve this, a class should never assumes more than one responsibility since each responsibility is an axis of change. Each class should have high cohesion.

- There are many examples of the usage of this design principles in our source code. One of the examples is getting data from database (text files) to present on the views. Instead of implementing the code in the individual views class that directly interacts with the text file to retrieve information, the actual implementation of code to get data from database is done in the **DatabaseManager** class instead, and the individual views class make use of the **DatabaseManager** class to obtain data from database. This is because the views classes is only responsible for presenting the information to the users, and thus should not assume the responsibility of interacting with the database.

2. Open-Closed Principle

- The open-closed principle states that a module should be open for extension but closed for modification. In this way, we will be able to change what the module does by extending on the module, without changing the source code of the module.

- There are many examples of the usage of this design principles in our source code. One of the examples is our View classes. All of the views are subclasses of the base view class called **BaseView.** The **BaseView** class defines some default properties of a view, such as having a title, view content as well as options for users. If we wish to customise the views or provide more functionality for the views, we extend from the **BaseView** class and do the necessary modification. Hence, in this case, our **BaseView** class is closed for modification, but open for extension to provide more functionalities.

3. Liskov Substitution Principle

- The Liskov substitution principle states that subtypes must be substitutable for their base types. This means that a user of a base class should continue to function properly even if a derivation of that class is passed into it.

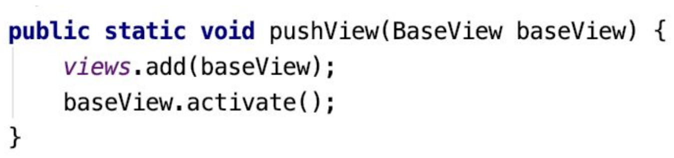
- One example of the usage of this principle in our project is our views classes. In this case, the user of our **BaseView** class is our **ViewsManager** Class. The **ViewsManager** class contains many methods which requires **BaseView** object. We ensured that the subclasses of the **BaseView** class provides no less functionalities than the base class, and the subclasses do not expect more than their base class. Hence, all of our views classes derived from **BaseView** is substitutable for the base class and can be used by the **ViewsManager** class.

4. Interface Segregation Principle

- The interface segregation principle states that many client specific interfaces are better than one general purpose interface. We did not use any custom interface in our source code.

5. Dependency Injection Principle

- The dependency injection principle states that a high level module should not depend upon low level modules, both should depend on abstractions. The abstractions should not depend upon details, details should depend upon abstractions. One such example is in the **pushView** method in the **ViewsManager** class:



```
public static void pushView(BaseView baseView) {
    views.add(baseView);
    baseView.activate();
}
```

- In this case, instead of relying on the high level module, **ViewsManager,** to call the individual methods in the derived classes of BaseView to print the view content or manage user interaction, we make use of a single method in **BaseView** called

activate(). This method acts as an abstraction layer for any users of this class to "activate" the view. Once the **activate()** method is called, the view will manage by itself all the necessary steps to transform into active state, such as printing out the view content onto the screen and getting input from the application user.

Object-Oriented Concepts

1. Inheritance

- Proper usage of inheritance helps to reduce code duplication. This is because if a class behaves very similarly to another class but provides additional functionality, this class can be made to be a subclass of that class to utilise the code written in the superclass.

- For example in our project, both the admin module and the movie-goer module needs to make use of a View that list the available movies at some point in the application. Hence, we created a class called **BaseListMoviesView,** which provides a basic view that helps to list movies. Then, for the admin and movie-goer module, we extend on this base class and created **MovieGoerListMoviesView** class and **AdminListMoviesView** to extend on the functionalities of the basic list movies view.

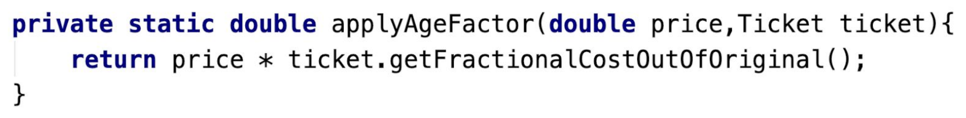
2. Encapsulation

- Encapsulation helps to build a barrier to protect an object's private data. We utilised encapsulation in all of our classes, where all the attributes are made to be private and access and modification to these attributes are done through the getter and setter methods.

3. Polymorphism

- Polymorphism is the ability of an object reference being referred to different types, knowing which method to apply depending on where it is in the inheritance hierarchy.

- For example, we have a class called **Ticket** which contains an abstract method called **getFractionalCostOutOfOriginal().** Subclasses such as **SCTicket** (senior citizen ticket) and **ChildTicket** will override the **getFractionalCostOutOfOriginal()** method which returns the fraction of the total price of the ticket that this ticket cost after their entitled discounts. Hence, in our **PriceManager** class is responsible for calculating prices, it contains the following method:



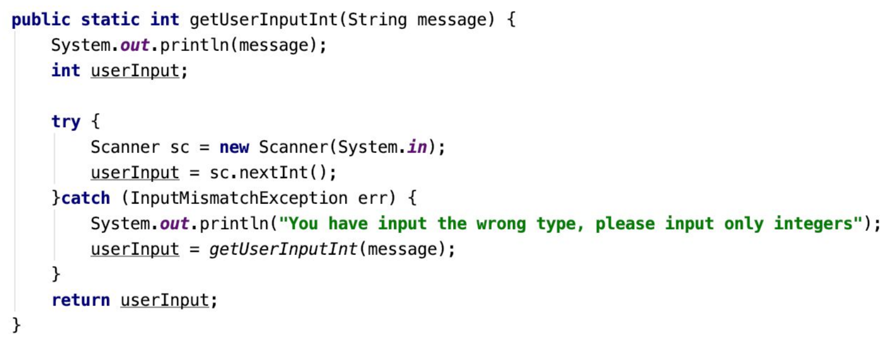
```java
private static double applyAgeFactor(double price,Ticket ticket){
    return price * ticket.getFractionalCostOutOfOriginal();
}
```

- This is an example of polymorphism because depending on the actual object type of **Ticket** that is passed into this function, the respective method implementation of **getFractionalCostOutOfOriginal()** of that object type will be called instead.

4. Abstraction

- One example where we made use of the concept of abstraction is our **IOManager** class, which is responsible for getting input from the user as well as output information onto the console for the user. In particular, the concept of abstraction is used for the process for getting input from the user.

- Getting input from the user can be prone to errors. For example, our program may run into error if we expect an integer input from the user but the user input a characters or strings instead. Hence, the typical usage of the **Scanner** object involves a lot of error handling code. Since many of our View classes rely on getting input from the user, this getting input and error handling logic can be abstracted and be handled by a single class, **IOManager.** Our View classes will make use of the methods by **IOManager** class to get input from user instead. Below is an example of a method in **IOManager** class to get integer input from user.



```java
public static int getUserInputInt(String message) {
    System.out.println(message);
    int userInput;

    try {
        Scanner sc = new Scanner(System.in);
        userInput = sc.nextInt();
    }catch (InputMismatchException err) {
        System.out.println("You have input the wrong type, please input only integers");
        userInput = getUserInputInt(message);
    }
    return userInput;
}
```

Further Enhancements (2 new features)

1. Admin Feature: We can allow admin to add new cinema to existing cineplex with customisable seat layout. For every cinema in the cineplex, the admin can customise the layout of the seats such as the number of rows and columns, the number of aisles, or even the addition of new seat types such as couple seat or wheelchair seat.

   This feature is possible in our design because we followed the Single Responsibility Principle closely as well as making use of the concept of abstraction. Instead of making our **Cinema** class be responsible for the management of its own seat layout, we abstracted the logic of seat layout management code into a new class called **SeatLayout.** An instance of **SeatLayout** is then injected into the **Cinema** class as a dependency. Not only does this promote Single Responsibility Principle, the greater benefit of this design is reusability of the seat layout and the loose coupling between the seat layout class and the cinema class. The seat layout of a cinema can also be easily changed and swappable by simply injecting a different instance of **SeatLayout** into the **Cinema** class. For example in this case, if we want to create a new cinema with a seat layout that has two aisles instead of one, we can simply subclass the **SeatLayout** and override the **printSeatLayout()** method to print the layout with two aisles instead of one, and then inject the instance of this subclass into the **Cinema** class.

2. Movie-goer feature: We can allow movie-goers to be able to compare the prices across different types of movies, cinema class and day of the week combinations, as well as ticket types to allow them to better analyse the price differences between two different circumstances to enable them to choose a combination that is best suited for them.

For example, after choosing their tickets for a Normal cinema class, they can choose to see the prices for the same movie screening at a Platinum cinema class instead. They can then decide if the premiums they pay for viewing at a Platinum class cinema is worth it and can choose to place an order there instead.

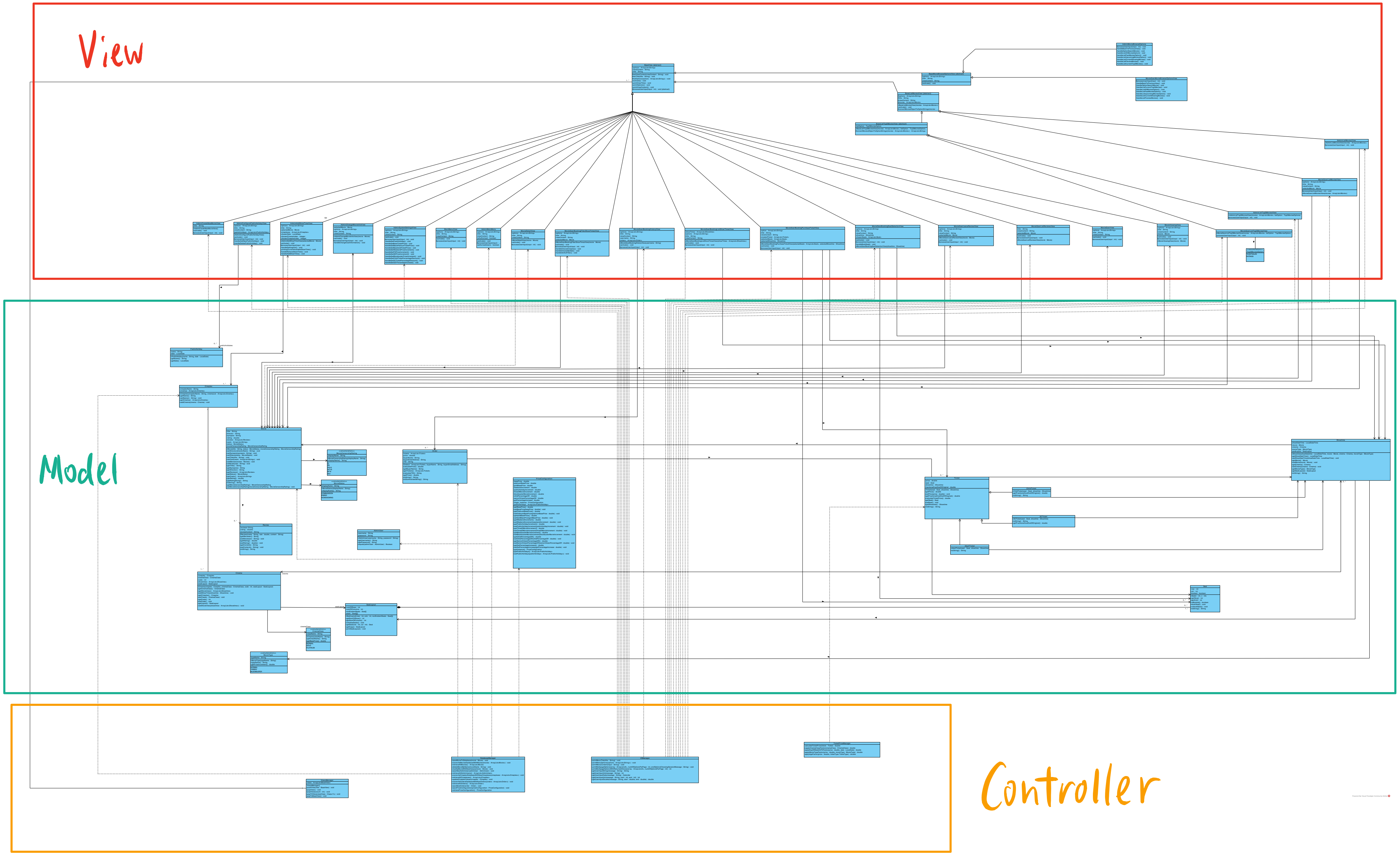
This feature is supported by the open-closed principle of our design, as it makes use of already existing model classes such as **Showtime**, **Cinema** and **PriceConfiguration** to calculate ticket prices. Therefore, only a simple extension of the application for movie-goers to choose to compare between ticket combinations, and to display order prices together for them. This can be done without having to change the existing modules and classes.

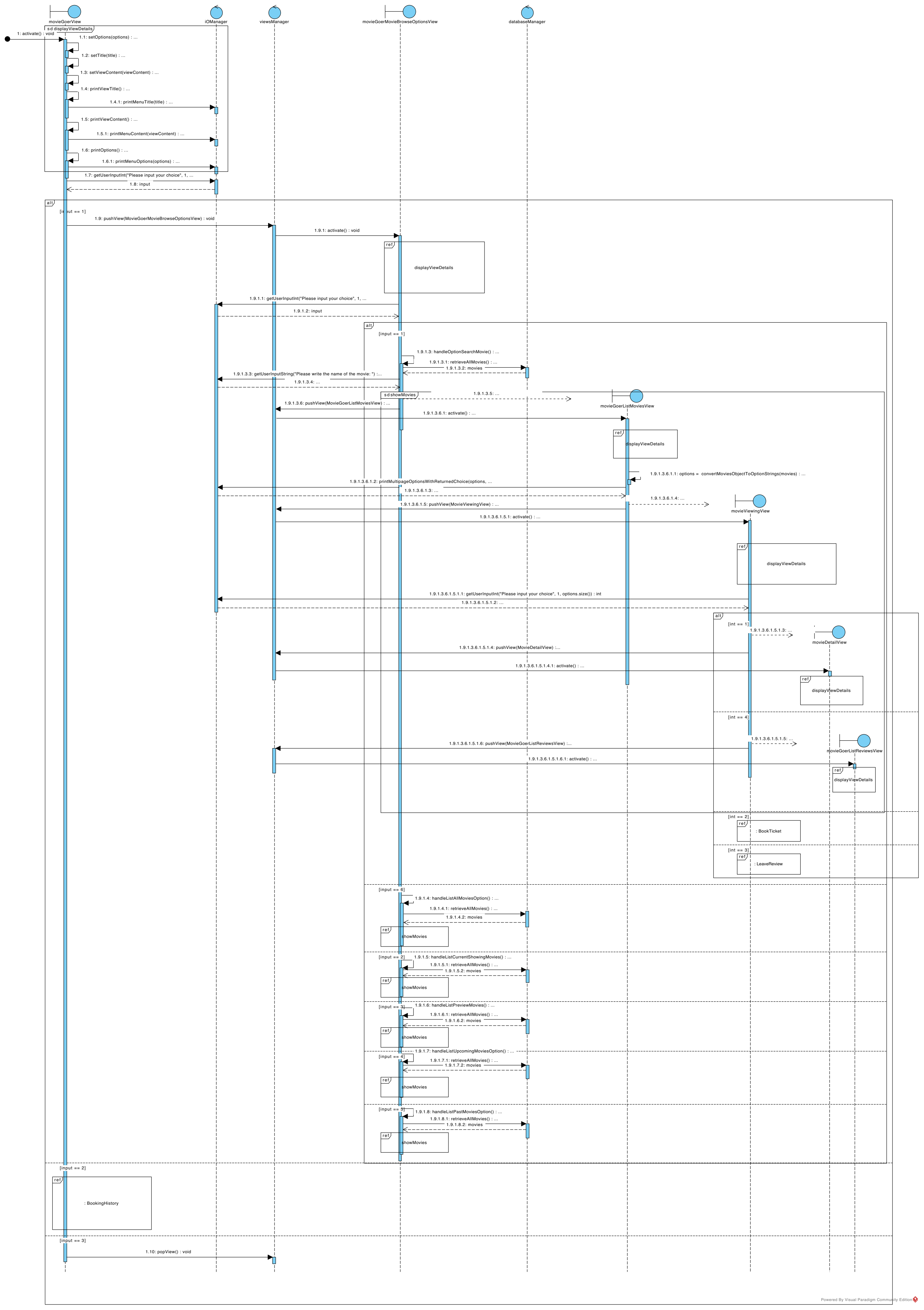
Class Diagram : Page 14

Sequence Diagram: Flow Description

The sequence diagram illustrates the classes involved when a movie-goer wants to view the details of a movie. First, once the the activate() method is called on the movieGoerView, it sets its attributes which will then be printed on the console showing options for the movie-goer to choose. The printing of view details as well as getting the user's input are all done by the methods of the IOManager. After getting the user's input to show movies, we push the next view, movieGoerMovieBrowseOptionsView, through the ViewsManager onto a stack. The activate() method is then called on the view, displaying details and options of the view.

At this stage, the movie-goer can then select whether he wants to search for a movie in which the input is 1, list movies depending on showing status, or list all movies in which the input is 6. The DatabaseManager.retrieveAllMovies() method is then called to get all the movies that is stored in the database, which will then be filtered based on either the user's search string, or showing type. The next view, movieGoerListMoviesView, is then shown for the user to choose which movie he wishes to see, and then movieViewingView is shown to allow him to either choose to see the movie details, see the reviews of the movie, book a ticket, or leave a review. The movie details and reviews are both presented in a new view each, movieDetailView and movieGoerListReviews respectively, and will be shown when the views activate().


View
Model
Controller


movieGoerView
iOManager
viewsManager
movieGoerMovieBrowseOptionsView
databaseManager
sd displayViewDetails
1: activate() : void
1.1: setOptions(options) : ...
1.2: setTitle(title) : ...
1.3: setViewContent(viewContent) : ...
1.4: printViewTitle() : ...
1.4.1: printMenuTitle(title) : ...
1.5: printViewContent() : ...
1.5.1: printMenuContent(viewContent) : ...
1.6: printOptions() : ...
1.6.1: printMenuOptions(options) : ...
1.7: getUserInputInt("Please input your choice", 1, ...
1.8: input
alt
[input == 1]
1.9: pushView(MovieGoerMovieBrowseOptionsView) : void
1.9.1: activate() : void
ref
displayViewDetails
1.9.1.1: getUserInputInt("Please input your choice", 1, ...
1.9.1.2: input
alt
[input == 1]
1.9.1.3: handleOptionSearchMovie() : ...
1.9.1.3.1: retrieveAllMovies() : ...
1.9.1.3.2: movies
1.9.1.3.3: getUserInputString("Please write the name of the movie: ") :...
1.9.1.3.4: ...
sd showMovies
1.9.1.3.5: ...
movieGoerListMoviesView
1.9.1.3.6: pushView(MovieGoerListMoviesView) : ...
1.9.1.3.6.1: activate() : ...
ref
displayViewDetails
1.9.1.3.6.1.1: options = convertMoviesObjectToOptionStrings(movies) : ...
1.9.1.3.6.1.2: printMultipageOptionsWithReturnedChoice(options, ...
1.9.1.3.6.1.3: ...
1.9.1.3.6.1.4: ...
movieViewingView
1.9.1.3.6.1.5: pushView(MovieViewingView) : ...
1.9.1.3.6.1.5.1: activate() : ...
ref
displayViewDetails
1.9.1.3.6.1.5.1.1: getUserInputInt("Please input your choice", 1, options.size()) : int
1.9.1.3.6.1.5.1.2: ...
alt
[int == 1]
1.9.1.3.6.1.5.1.3: ...
movieDetailView
1.9.1.3.6.1.5.1.4: pushView(MovieDetailView) :...
1.9.1.3.6.1.5.1.4.1: activate() : ...
ref
displayViewDetails
[int == 4]
1.9.1.3.6.1.5.1.5: ...
movieGoerListReviewsView
1.9.1.3.6.1.5.1.6: pushView(MovieGoerListReviewsView) :...
1.9.1.3.6.1.5.1.6.1: activate() : ...
ref
displayViewDetails
[int == 2]
ref
: BookTicket
[int == 3]
ref
: LeaveReview
[input == 6]
1.9.1.4: handleListAllMoviesOption() : ...
1.9.1.4.1: retrieveAllMovies() : ...
1.9.1.4.2: movies
ref
showMovies
[input == 2]
1.9.1.5: handleListCurrentShowingMovies() : ...
1.9.1.5.1: retrieveAllMovies() : ...
1.9.1.5.2: movies
ref
showMovies
[input == 3]
1.9.1.6: handleListPreviewMovies() : ...
1.9.1.6.1: retrieveAllMovies() : ...
1.9.1.6.2: movies
ref
showMovies
[input == 4]
1.9.1.7: handleListUpcomingMoviesOption() : ...
1.9.1.7.1: retrieveAllMovies() : ...
1.9.1.7.2: movies
ref
showMovies
[input == 5]
1.9.1.8: handleListPastMoviesOption() : ...
1.9.1.8.1: retrieveAllMovies() : ...
1.9.1.8.2: movies
ref
showMovies
[input == 2]
ref
: BookingHistory
[input == 3]
1.10: popView() : void
Powered By Visual Paradigm Community Edition