<u>Declaration of Original Work for SC/CE/CZ2002 Assignment</u>

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.

Name	Course	Lab Group	Signature	Date
LEE JUIN	SC2002	SSP2		11/11/2022
OI YEEK SHENG	SC2002	SSP2		11/11/2022
NG HONG JIN, JONATHAN	SC2002	SSP2		11/11/2022
JERICK LIM KAI ZHENG	SC2002	SSP2		11/11/2022
LEE CI HUI	SC2002	SSP2		11/11/2022

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.

1.   Introduction

1.1   Class Model

The MOBLIMA Application is developed using Object-oriented Design Strategies. To begin development, the team started by constructing a concise and comprehensive class model, which comprises of two sub-models —— Entity Class model and Control Boundary Class model. The former helped the team in developing persistent data relationships while the latter depicted a general overview of the program flow.

The Class Models are presented as follows. A supplementary copy of each class model diagram is provided **independent** of this document for clarity.

1.1.1   Entity Class Model

The Entity classes represent the persistent data in the database and are modelled as Model in the Model-View-Controller (MVC) architectural design pattern. They contain the constructor, accessor and mutator methods as well as relevant enumerations. The relationships between entity classes (Dependency, Association and Inheritance) are depicted in the Entity Class Model diagram below labelled Figure 1.

Figure 1

1.1.2   Control Boundary Class Model

The Control-Boundary Class Model is depicted in Figure 2. The class model adopts the Model-View-Controller architectural design.

The Control classes are translated into Managers in code. Essentially, each Manager is responsible for all business logic related to a single Entity class. For instance, the `CinemaManager` is responsible for handling logic related to the `Cinema` class. On top of that, the Manager also serves as the middleman between the Entity class and the View class.

The access to Model class is abstracted by using a database as depicted in the figure. This is consistent with the Facade design pattern where a unified interface is provided to allow easier code readability and maintainability.
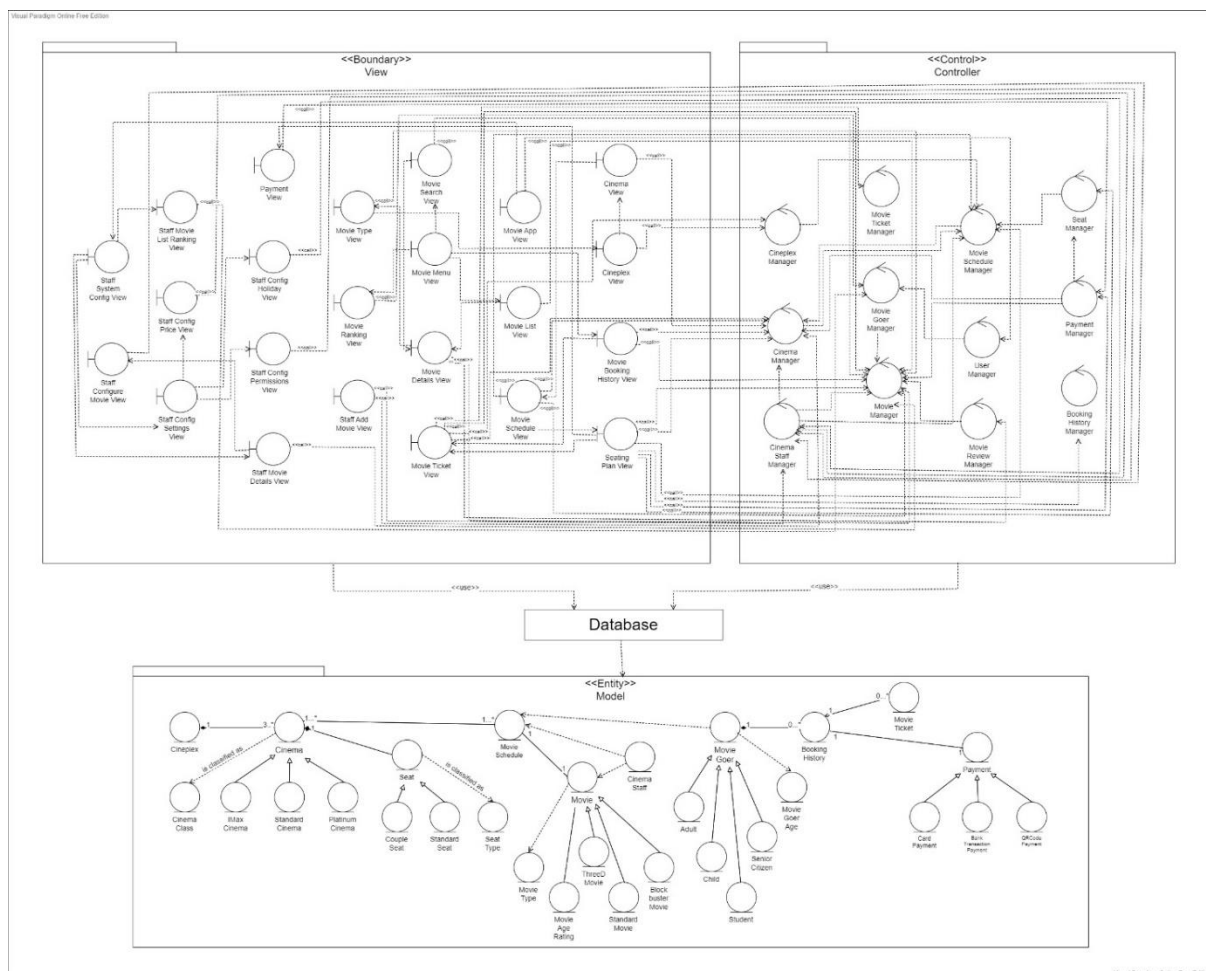

Figure 2

1.2   Testing

After development, the MOBLIMA Application has undergone several Unit Testing and Integration Testing. This section covers the techniques used during the testing phase. Supplementary copies of the testing results in the form of pictorial screenshots are provided **independent** of this document.

1.2.1   Grey-box Testing

Functionalities	Input	Expected output	Actual output
Login as MovieGoer	Correct username and password	Login is successful	Login is successful
Login as admin	Incorrect username and password	Login is unsuccessful.	Login is unsuccessful.
Login as admin	Correct username and password	Login is successful.	Login is successful.
Change MovieGoer permission to list Top 5 movies by ticket sales/overall reviews.	Select option to opt out MovieGoer permission to list Top 5 movies by ticket sales/overall reviews.	MovieGoer cannot view Top 5 movies by ticket sales/overall reviews.	MovieGoer cannot view Top 5 movies by ticket sales/overall reviews.
Book ticket at different cinema.	Book ticket at Standard Cinema. Book ticket at Platinum Cinema.	Booking at Standard Cinema show standard price. Booking at Platinum Cinema show premium price.	Booking at Standard Cinema show standard price. Booking at Platinum Cinema show premium price.
Configure movie status to End of Showing	Select option to update movie details. Select option to update showing status. Select End of Showing.	Movie is not listed for MovieGoer. Movie is listed for admin.	Movie is not listed for MovieGoer. Movie is listed for admin.

List Movie. **Book ticket.** **View booking history.**	Select option to view list of movies. Select movie to book ticket. Select option to view booking history.	Movies are listed. Tickets are booked. Booking history is shown.	Movies are listed. Tickets are booked. Booking history is shown.
Configure holiday date. **Book ticket.** **Ticket shows different price.**	Add 25th December 2022 to holiday. Book ticket for movie to be shown on 25th December 2022.	Holiday is added successfully. Ticket price reflects the holiday ticket price.	Holiday is added successfully. Ticket price reflects the holiday ticket price.
Attempt to book movie on Coming Soon status.	Select movie that is currently on Coming Soon status. Select "Booking Query" to book movie.	Prompted with "This movie is currently unavailable for booking right now."	Prompted with "This movie is currently unavailable for booking right now."

1.2.1 Additional Test Cases

(a) Movie "Ticket to Paradise" changed Showing Status to "End-of-Showing"

    (i) MovieGoer unable to view on Movie Listing, but Cinema Staff able to

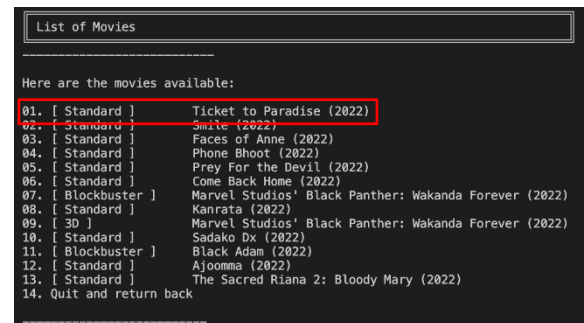*Figure 3 MovieGoer's Listing View*

Figure 4 Staff's Listing View

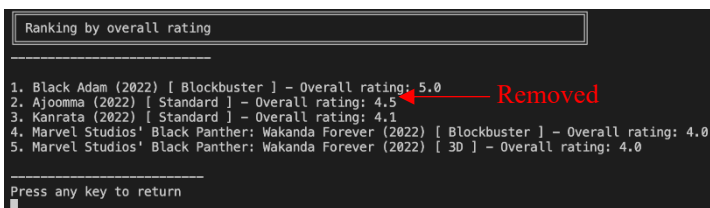   (ii) MovieGoer unable to view on Top 5 Ranking, but Cinema Staff able to
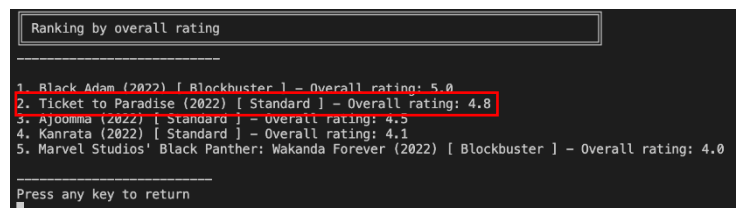

Figure 5 MovieGoer's Listing View

Figure 6 Staff's Listing View

(b)   Movies of Showing Statuses "Preview" available for booking, but not for "Coming Soon"
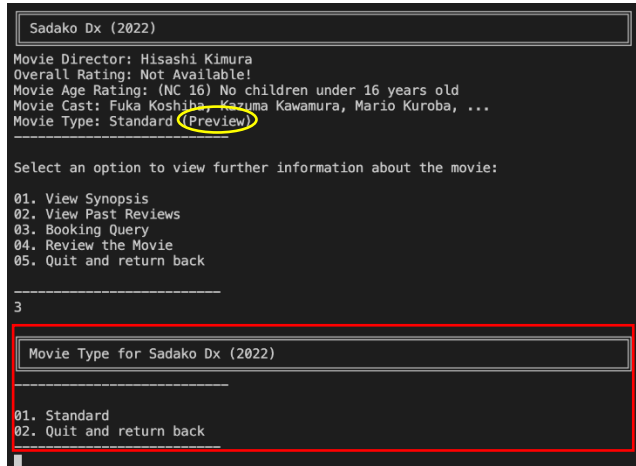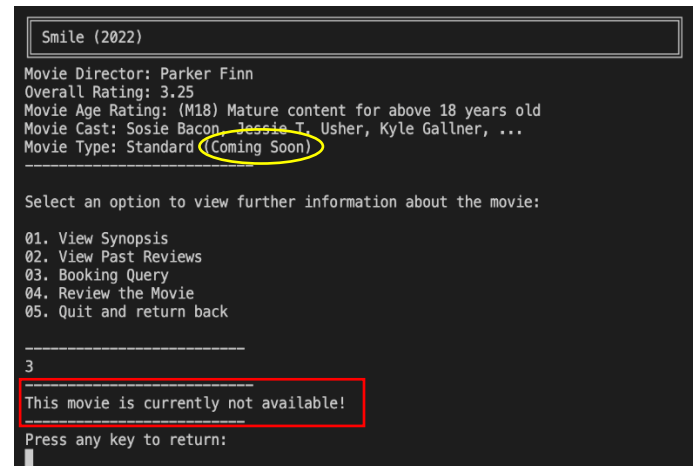

Figure 7 Booking Query on "Preview" Movie

Figure 8 Booking Query on "Coming Soon" Movie

1.3   Assumptions & Dependencies

1.3.1   Assumptions

- The MOBLIMA Application is designed with a pseudo-manual notion. No external server is hosting the MOBLIMA Application. Hence, actions such as removing a showing time can only be done manually upon start-up of the application by the administrator.

- The MOBLIMA Application is designed with a simple login feature. No help is provided for lost account retrieval.

- The MOBLIMA Application does not verify the legitimacy of Movie Goers who self-proclaimed as either Student, Child, or Senior Citizen. The verification shall be done by the vendor instead.

- The MOBLIMA Application does not verify the legitimacy of Movie Goers' payment information.

- The MOBLIMA Application assumes that a movie may have multiple showing types, i.e., a movie can be showed as a 3D movie, or a standard movie.

- The MOBLIMA Application assumes that a movie's review is independent of its showing type. Hence, all reviews under the same movie title are displayed to the user upon query, regardless of the showing types available.

- The MOBLIMA Application assumes that a movie's title is always unique. Hence, no exception handling regarding movie of the same name is done.

1.3.2   Dependencies

Java support	The MOBLIMA Application is developed under Java SE 7.
Command-line Interface (CLI) support	The MOBLIMA Application requires a terminal window to operate. The Command Prompt from Windows or Terminal on MacOS will suffice for the application.

2.   Design Standards

2.1   Software Engineering Principles

2.1.1   Model-View-Controller (MVC) Design Pattern

The MOBLIMA Application adopts a Model-View-Controller (MVC) system architectural design pattern. The Model component holds all persistent data of the application, while the Controller component represents all the business logic behind the application. Finally, the View component is the CLI presented to the Movie Goers. The file structure of MVC design pattern is illustrated in Figure 9.

The development team decides to adopt the MVC architectural pattern as it allows segregation of code between components. This allowed each developer to work independently and simultaneously on different components, significantly speeding up the development process.
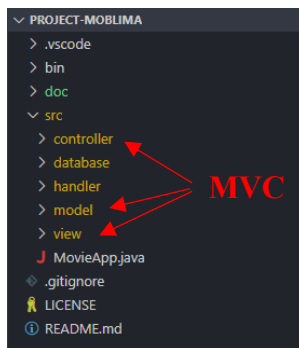

Figure 9

2.1.2   Version Control

The development team uses GitHub for version control. Adopting the practice of version control allows the development team to rollback any breaking changes whenever necessary, on top of that allowing the team to work simultaneously on the same files using different branches.

2.2   Object-oriented Design Principles

2.2.1   Liskov Substitution Principle

In the View component, the development team applied the Liskov Substitution Principle. Each View class inherits from the `MainView` class. The children View classes implements the `printMenu()` and `appContent()` methods without reducing any functionalities, and thus behaving the same way as `MainView` class. Hence, the `MainView` class is replaceable by its children classes.

A code snippet of the `MainView` class is shown in Figure 10.

```java
public abstract class MainView {
    public MainView() {}
    public abstract void printMenu();
    public abstract void appContent();
    public static void printBoilerPlate(String content) {
        String spaces = String.format("%" + (65 - content.length()) + "s", ...args: "");
        System.out.println(
            x: "┌──────────────────────────────────────────────┐");
        System.out.println("║ " + content + spaces + "║");
        System.out.println(
            x: "└──────────────────────────────────────────────┘");
    }
    public static void printMenuContent(String content) {
        System.out.println(
            x: "--------------------------");
        System.out.println(content);
        System.out.println(
            x: "--------------------------");
    }
}
```

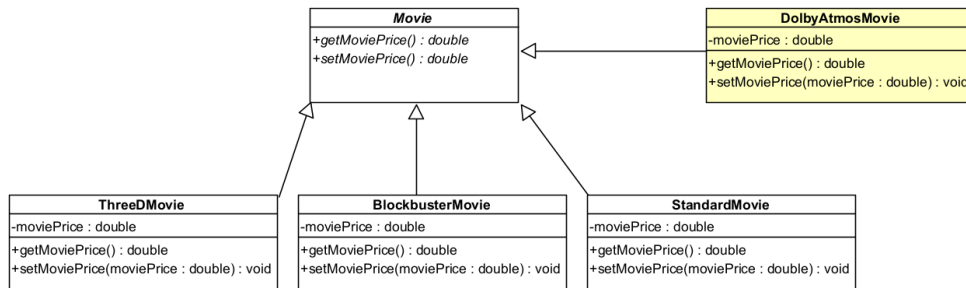*Figure 10*

2.2.2   Open-Closed Principle

Figure 11

In the Model component, the development team adopted the Open-Closed Principle when building the model for Movie, Cinema and Movie Goer. A section of the modified Entity Class Diagram for Movie is shown in Figure 11.

Suppose we wish to support a new Movie type, Dolby Atmos movie, in our application. To do so, we only need to create a new class called `DolbyAtmosMovie` and inherit from the `Movie` class. Since methods such as `getMoviePrice` and `setMoviePrice` are abstract methods to be realized, we do not need to modify the `Movie` class to include the price for Dolby Atmos movies. On top of that, suppose that Dolby Atmos movies have additional attributes or methods, we can directly implement those attributes and methods in the `DolbyAtmosMovie` class without modifying the Movie class.

Hence, the Movie class is effectively *closed* for modification, but *open* for extension to support more types of movies.

2.2.3   Dependency Inversion Principle

Figure 12

```
public static BookingHistory createBookingHistory(ArrayList<MovieTicket> movieTicket, Payment payment, MovieGoer movieGoer) {
    String UUID = String.format(format: "BH%04d", DatabaseHandler.generateUUID(Database.BOOKING_HISTORY));
    BookingHistory bookingHistory = new BookingHistory(UUID, movieTicket, payment);
    movieGoer.addBookingHistory(bookingHistory);
    DatabaseManager.saveUpdateToDatabase(UUID, bookingHistory, Database.BOOKING_HISTORY);
    DatabaseManager.saveUpdateToDatabase(movieGoer.getUUID(), movieGoer, Database.MOVIE_GOER);
    return bookingHistory;
}
```
Depend on abstract classes

Building on top of the concept of Open-Closed Principle, the development team also applied the Dependency Inversion Principle such that the business logic found in all the Controller classes depend on the abstract and interface classes, instead of the concrete classes.

An example is shown in Figure 12 and Figure 13. As mentioned, the development team applied the Open-Closed Principle when building models. As such, the model `MovieGoer` is abstracted and extended by concrete classes such as `Adult`, `Child`, `Student` and `SeniorCitizen`.
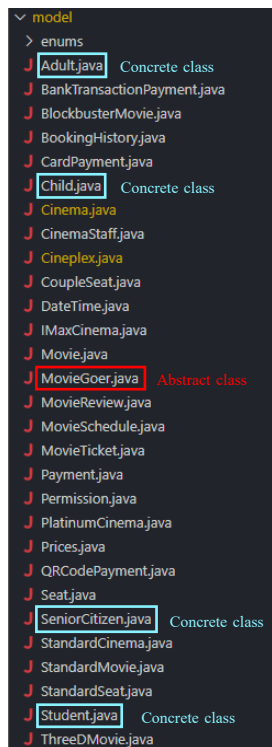
However, all Controller and View classes uses and depends on the abstract class `MovieGoer` instead, as shown in Figure 12. This allows looser coupling between the Controller classes and the Model classes, as instead of depending on the concrete classes like Adult or Student, the Controller class only interacts with the abstract `MovieGoer` class. Hence, when there are changes to be made in the concrete classes such as `Adult`, we do not have to change anything in the Controller class.

```
∨ model
  > enums
  J Adult.java       Concrete class
  J BankTransactionPayment.java
  J BlockbusterMovie.java
  J BookingHistory.java
  J CardPayment.java
  J Child.java       Concrete class
  J Cinema.java
  J CinemaStaff.java
  J Cineplex.java
  J CoupleSeat.java
  J DateTime.java
  J IMaxCinema.java
  J Movie.java
  J MovieGoer.java   Abstract class
  J MovieReview.java
  J MovieSchedule.java
  J MovieTicket.java
  J Payment.java
  J Permission.java
  J PlatinumCinema.java
  J Prices.java
  J QRCodePayment.java
  J Seat.java
  J SeniorCitizen.java  Concrete class
  J StandardCinema.java
  J StandardMovie.java
  J StandardSeat.java
  J Student.java     Concrete class
  J ThreeDMovie.java
```

Figure 13

2.3   Proposed Features

2.3.1   Food Payment System

The development team proposes a food payment system to be included as future features. The added food payment system allows Movie Goer to pay for snacks/food to be consumed during their movie watching through the MOBLIMA Application. With this, Movie Goers only needs to collect their snacks/food at the box office prior to entering the cinema without queueing.

To accommodate the new future feature, we can implement `IPaymentFood` and `IPaymentTicket` interfaces on the different payments. Implementing interfaces separately satisfies the Interface Segregation Principle, as a vendor may choose to include a new payment type in the future which can only be used to purchase tickets without rewriting any classes (instead, the new payment type class will only need to implement the `IPaymentTicket` interface). The modified section of the entity class diagram is shown in Figure 14.
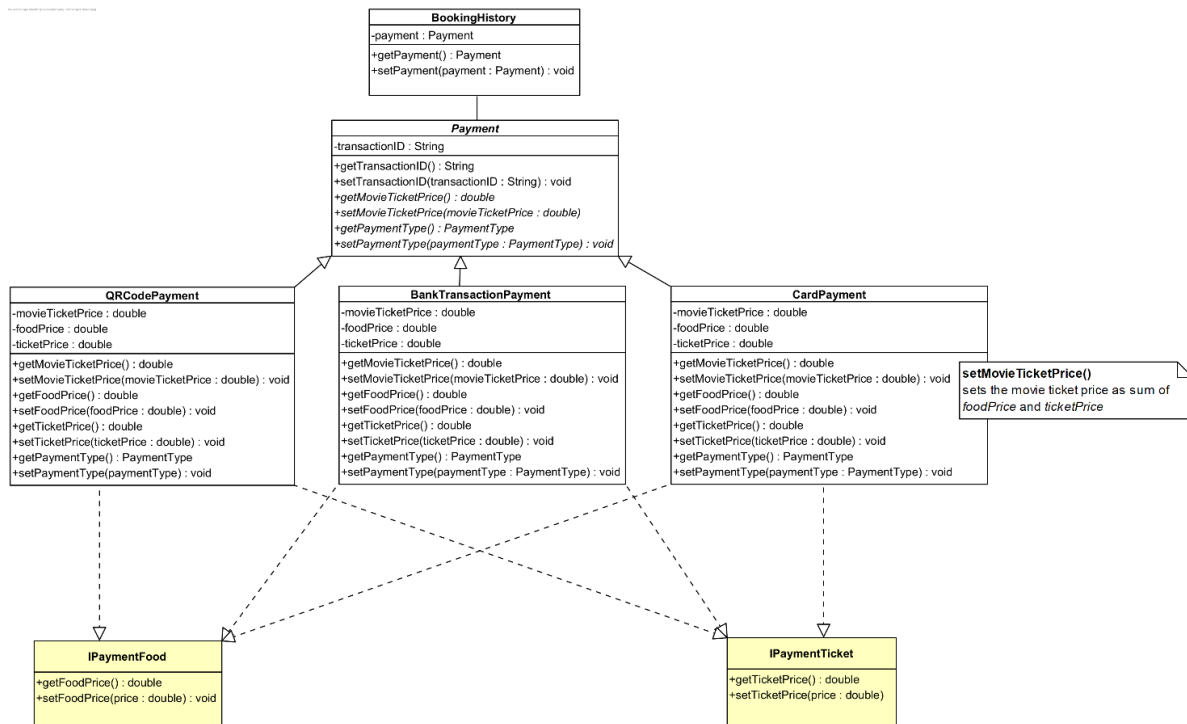

Figure 14

2.3.2 Loyalty Program

Building on top of the first proposed feature, the development team also proposes to add a loyalty program to the MOBLIMA application. This feature allows Movie Goer to collect loyalty points for every movie ticket they purchase with MOBLIMA. The Movie Goers will then be able to redeem these loyalty points when purchasing future tickets.

Since `Payment` is an abstracted class, we only need to add a new `LoyaltyPointsPayment` class without modifying the existing `Payment` class. On top of that, if we were to implement both the first and second features, we do not need to change anything other than to implement `IPaymentFood` and `IPaymentTicket` on the `LoyaltyPointsPayment` class. This is because the existing sub-system on Payment satisfies the Open-Closed principle, where the `Payment` class is closed for modification and opened for extension for different types of payment.

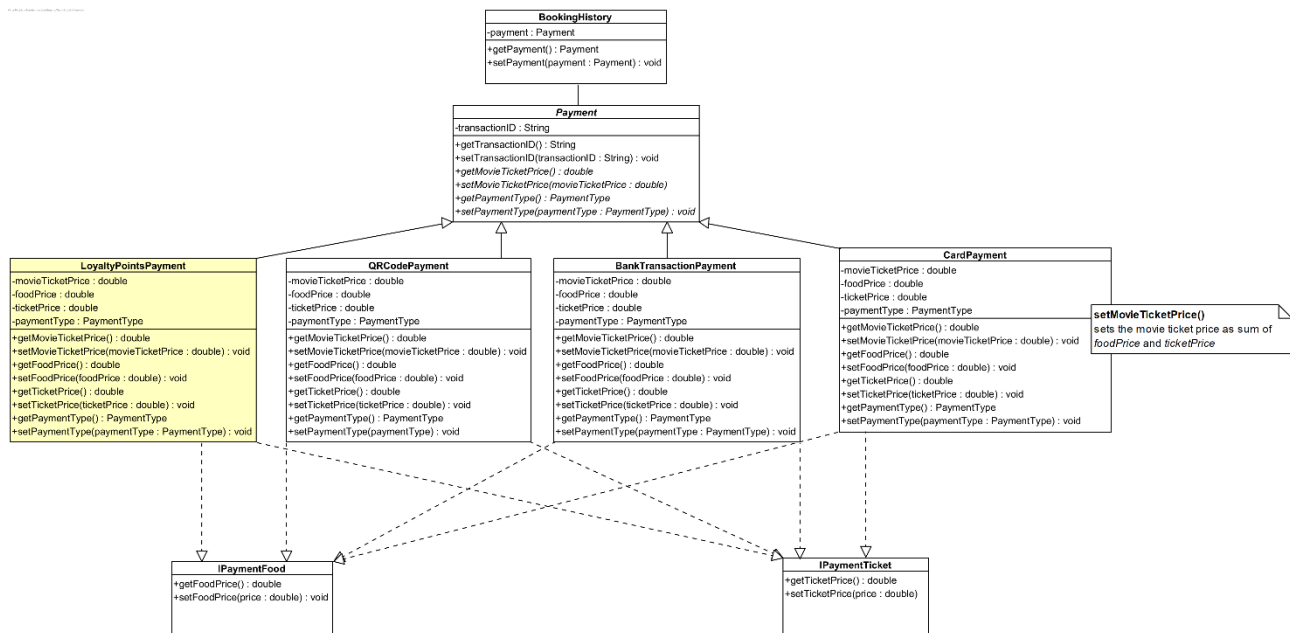The modified section of the entity class diagram is shown in Figure 15.

Figure 15

3.   Appendix

Link to Demo Video:
https://drive.google.com/file/d/1MSkguctkESqwN6FAZpG1zfYTokUy1y2o/view?usp=sharing