Design Considerations

The goal of our design is to make the application easy to maintain, modify and extend. This can be done by managing the dependencies between classes and packages to minimise the impact of change on other parts of the software.

SOLID Design Principles

In this section, we will analyse how our application design exhibits high cohesion and low coupling using the five design principles.

Single Responsibility Principle

This principle aims to achieve high cohesion within a class. In our project, the classes are divided into 3 packages (model, view and controller). The classes in the model, controller and view packages are the entity, control and boundary classes respectively. Within these 3 packages, each class is designed to perform a single role. Take the scenario below as an example.

> When the user wants to make a booking, he will firstly login to the system and the `MovieGoer` class handles password validation. The control is then transferred to `BookingController` (handles the logic), `BookView` (handles the display) and the model classes (handles the data) to handle the booking process. When `BookingController` needs data stored in files, it calls `DataManager` class to retrieve necessary information from `DataStore` object.

The scenario shows that `BookingController` does not need to know about login. It also does not need to know about how the information is displayed to and retrieved from the user. It also does not need to know about the data structure and how to handle it because the `DataManager` and `DataStore` take care of it. Therefore, `BookingController` only has a single responsibility which is to control the logic flow of the booking process. The same concept applies to all the other classes.

Open-Closed Principle

The principle highlights that a module should be open for extension but closed for modification and abstraction is the key. In our application, the controller classes (in controller package) implement the `Controller` interface and the interface acts as an abstraction layer. It allows us to add new classes to the application without causing a cascade of subsequent changes.

For instance, if the cineplexes want to add new functionality for `MovieGoer` (e.g. displaying Promotion and Event), we can just create a new controller class which implements the `Controller` interface for the functionality and add an additional option to the switch case to initialize the new class in `MovieGoerController` class. Nothing else needs to be changed because the `NavigationController` will handle all the implementations of the `Controller` interface the same way.

Liskov Substitution Principle

This principle states that subtypes must be substitutable for their base types. The `Controller` interface is the generalisation of all the specialised controller classes such as `BookingController, MovieController` and `BookingHistoryController`. The `NavigationController` class has a method to load different control classes: `load(Controller controller)`. This method will be able to load any implementation of the `Controller` interface, regardless of which instance we send to it

Interface Segregation Principle

For this principle, we will demonstrate that no client class should be forced to depend on methods it does not use. This can be achieved by creating client-specific interfaces in order to minimize coupling between classes. The `BookShow` interface consists of 3 methods: `getLayout(), checkAvail(boolean [][] selectedSeat)` and `getSeatAvailabilities()`. It is implemented by the `ShowTime` class. The class `BookView`, which uses only the methods in the `Bookshow` interface, uses the `Bookshow`

interface instead of depending on the whole `ShowTime` class. Hence, if changes are made to the `ShowTime` class, we do not need to recompile BookView class, thus decoupling the classes.

Dependency Injection Principle

This principle emphasizes that high-level modules and low-level modules should not depend on each other but they should both depend on abstraction. In our application, the `LabelledItem` interface is an abstraction between view classes (high-level module) and model classes (low-level module). So, the `MenuView`'s `getLabelledItem(String title, List<T> labelledItems)` method uses the `LabelledItem` interface to handle any object of a class that implements the `LabelledItem` interface. For example, the BookingController class uses this method in its `selectShowTime()` method for List objects with different data types. The relevant statements are shown below.

```
Cineplex cineplex = MenuView.getLabelledItem("Select a
Cineplex", cineplexList);
Movie movie = MenuView.getLabelledItem("Select a movie",
movieList);
ShowTime showTime = MenuView.getLabelledItem("Select a Show
Time", movieShowTimeList);
```

Hence, the model classes, `Cineplex`, `Movie` and `ShowTime`, and the view class, `MenuView`, both depend on the abstraction, `LabelledItem` interface. Hence, the high-level modules are independent of the low-level modules, so changes can be made to the low-level module without affecting the high-level module and vice-versa.

Object-Oriented Concepts

Abstraction

Abstraction refers to the act of representing essential features without including background details or explanations. Using model classes, such as `Movie`, `ShowTime` and

`CinemaStaff` allows us to create an abstract representation of real world objects. For example, a cinema staff member can be represented by a `CinemaStaff` object which contains their `username` and `password`, as well as a password checking method, `login(String password)`.

Encapsulation and Information Hiding

Encapsulation is building a barrier to protect an object's private data. In our application, attributes are set to private and they can only be accessed through public methods that retrieve or modify them. This way we can limit access to the private data of an object. For example, the `cinemaCode` of `Cinema` objects can only be accessed through the `getCinemaCode()` method and it cannot be modified.

Information hiding is hiding the complex inner workings of the methods. So, the method can be called without the user needing to know the implementation details of the method. In our application, this achieved through methods such as `IOController`'s `displayMessage(String message)` method, which client classes can use without needing to know how the `IOController` displays the message.

Inheritance

Inheritance is the mechanism by which classes may inherit or acquire the properties and methods of other classes. In our application three interfaces are used, `LabelledItem`, `Controller` and `BookShow`. For example, `CinemaStaffController` implements the `Controller` class. It inherits its behaviour, so it has to define the `Controller`'s `start()` function.

Polymorphism

Polymorphism is used to describe situations in which something takes on several different forms. In Java, we can implement polymorphism using method overriding. The interface `LabelledItem` contains the abstract method `getLabel()`. Different classes implementing this interface, such as `Movie`, `Cineplex` and `AgeGroup`, have different implementations of the function as shown below. However, when this method is called, such

as in the `MenuView`'s `getLabelledItem` method, the correct implementation of the `getLabel()` method will be used depending on the class of the actual object.

```
public String getLabel() {
    return title;
}
```

getLabel() in the Movie class

```
public String getLabel() {
    return name;
}
```

getLabel() in the Cineplex class

```
public String getLabel() {
    return label;
}
```

getLabel() in the AgeGroup enum class

Proposals for New Features

Booking of Live Shows

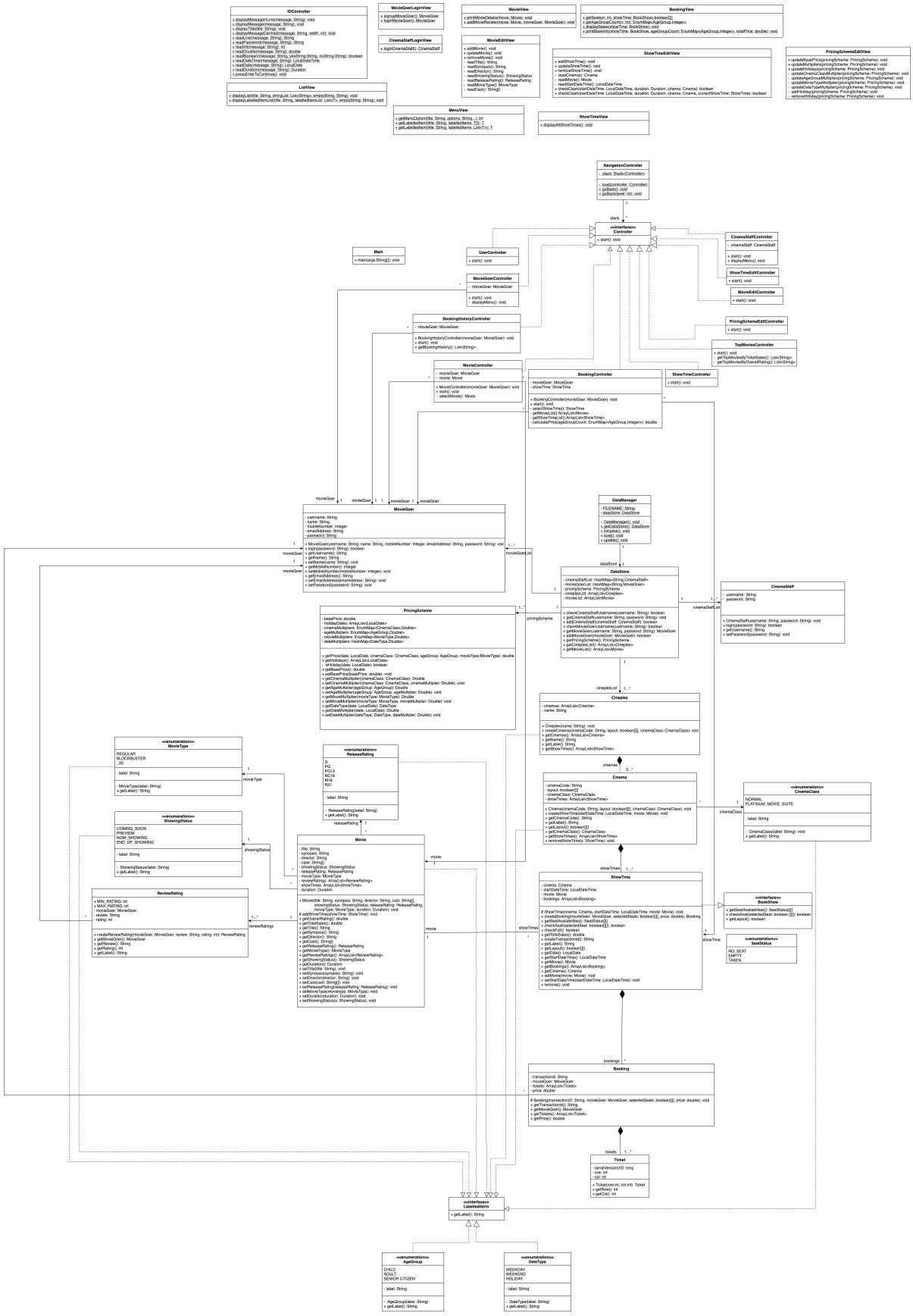
One possible new feature would be to allow for the booking of live shows, in addition to movies. The live shows may include performances such as theatre plays and live band shows. As mentioned under the Interface Segregation Principle section, the `BookShow` interface was used to decouple the `ShowTime` class and the `BookView` class. So, we can create another implementation of the `BookShow` interface for live show bookings. Then, we can reuse the same `BookView` class to manage the display and retrieval of information from the user.

Promotions View

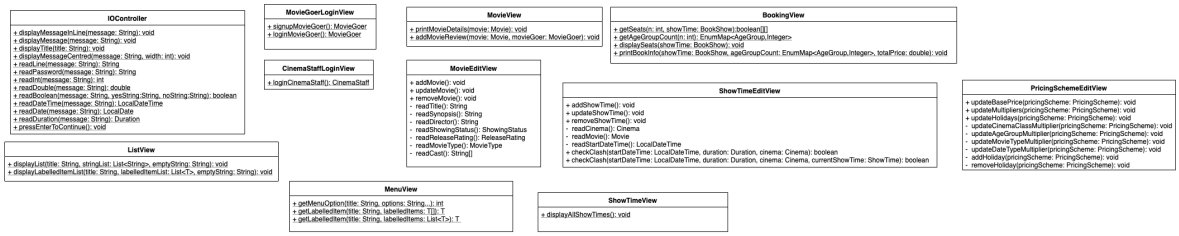
Another possible feature would be to display the promotions to the moviegoer. This display will allow the moviegoers to view the current promotions on movie bookings and food sold at the cineplexes. As mentioned under the Open-Closed Principle section, the `Controller`

interface allows for new controller classes to be created and added without requiring a cascade of changes.

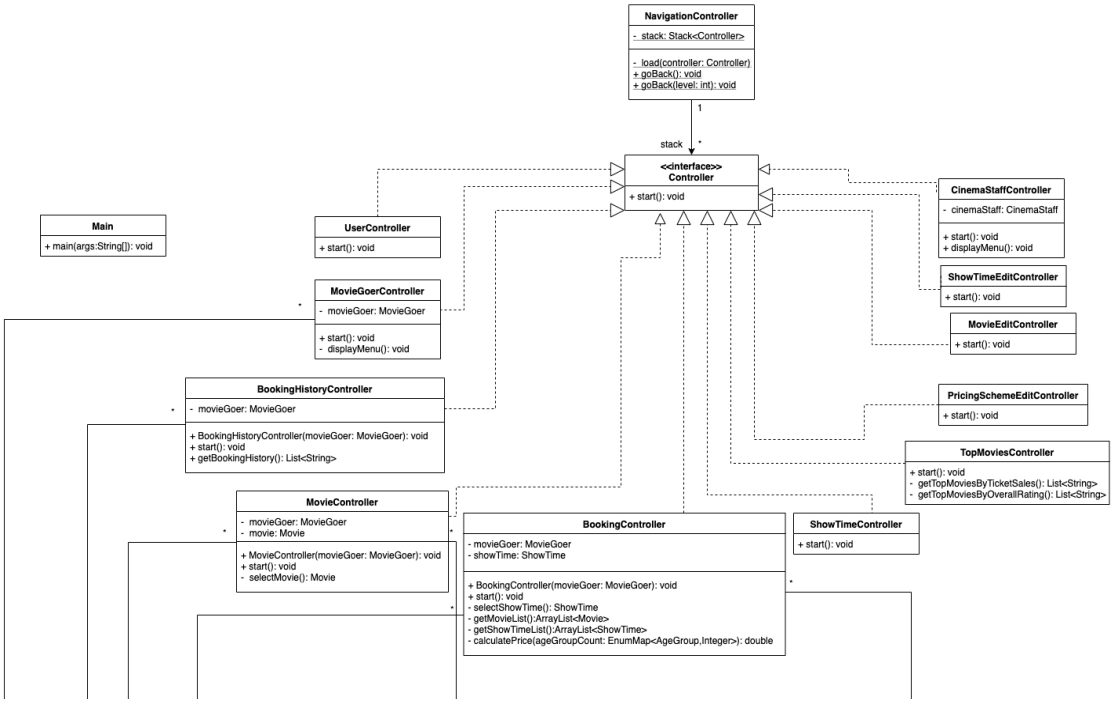
UML Class Diagram



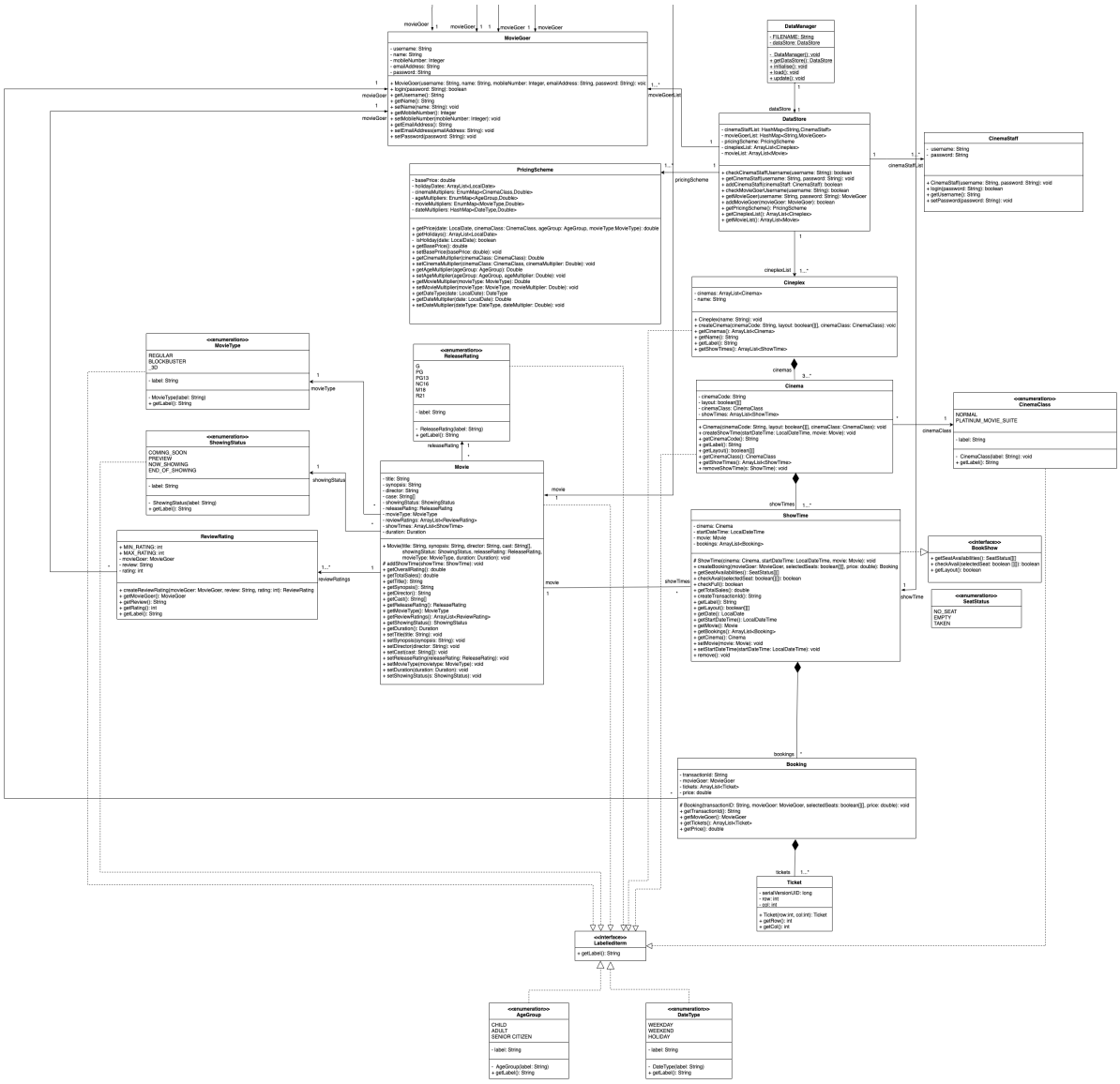
View



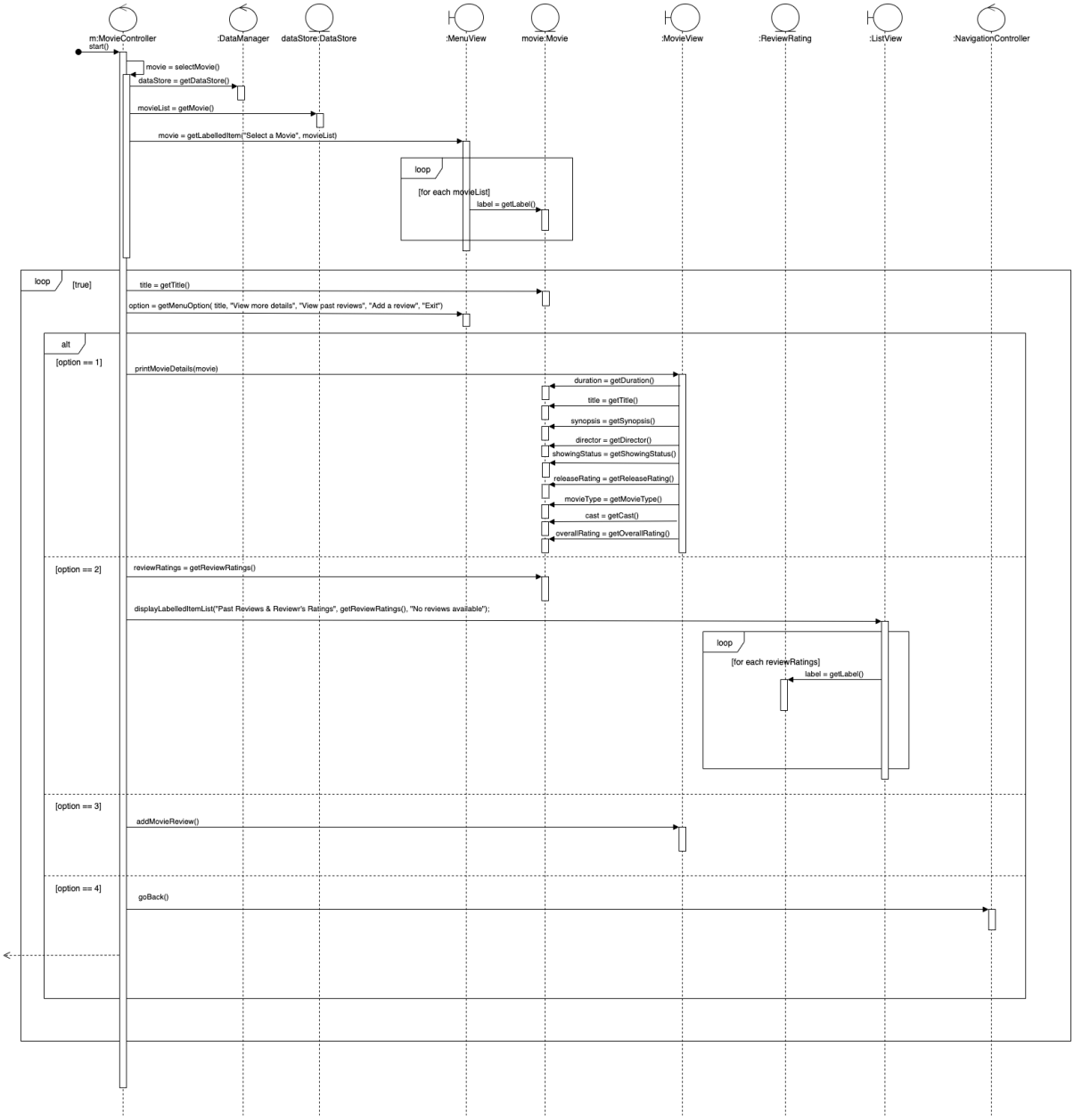
Controller



Model


MovieGoer
- username: String
- name: String
- mobileNumber: Integer
- emailAddress: String
- password: String
+ MovieGoer(username: String, name: String, mobileNumber: Integer, emailAddress: String, password: String): void
+ login(password: String): boolean
+ getUsername(): String
+ getName(): String
+ setName(name: String): void
+ getMobileNumber(): Integer
+ setMobileNumber(mobileNumber: Integer): void
+ getEmailAddress(): String
+ setEmailAddress(emailAddress: String): void
+ setPassword(password: String): void
DataManager
- FILENAME: String
- dataStore: DataStore
- DataManager(): void
+ getDataStore(): DataStore
+ initialise(): void
+ load(): void
+ update(): void
DataStore
- cinemaStaffList: HashMap<String,CinemaStaff>
- movieGoerList: HashMap<String,MovieGoer>
- pricingScheme: PricingScheme
- cineplexList: ArrayList<Cineplex>
- movieList: ArrayList<Movie>
+ checkCinemaStaffUsername(username: String): boolean
+ getCinemaStaff(username: String, password: String): void
+ addCinemaStaff(cinemaStaff: CinemaStaff): boolean
+ checkMovieGoerUsername(username: String): boolean
+ getMovieGoer(username: String, password: String): MovieGoer
+ addMovieGoer(movieGoer: MovieGoer): boolean
+ getPricingScheme(): PricingScheme
+ getCineplexList(): ArrayList<Cineplex>
+ getMovieList(): ArrayList<Movie>
CinemaStaff
- username: String
- password: String
+ CinemaStaff(username: String, password: String): void
+ login(password: String): boolean
+ getUsername(): String
+ setPassword(password: String): void
PricingScheme
- basePrice: double
- holidayDates: ArrayList<LocalDate>
- cinemaMultipliers: EnumMap<CinemaClass,Double>
- ageMultipliers: EnumMap<AgeGroup,Double>
- movieMultipliers: EnumMap<MovieType,Double>
- dateMultipliers: HashMap<DateType,Double>
+ getPrice(date: LocalDate, cinemaClass: CinemaClass, ageGroup: AgeGroup, movieType:MovieType): double
+ getHolidays(): ArrayList<LocalDate>
- isHoliday(date: LocalDate): boolean
+ getBasePrice(): double
+ setBasePrice(basePrice: double): void
+ getCinemaMultiplier(cinemaClass: CinemaClass): Double
+ setCinemaMultiplier(cinemaClass: CinemaClass, cinemaMultiplier: Double): void
+ getAgeMultiplier(ageGroup: AgeGroup): Double
+ setAgeMultiplier(ageGroup: AgeGroup, ageMultiplier: Double): void
+ getMovieMultiplier(movieType: MovieType): Double
+ setMovieMultiplier(movieType: MovieType, movieMultiplier: Double): void
+ getDateType(date: LocalDate): DateType
+ getDateMultiplier(date: LocalDate): Double
+ setDateMultiplier(dateType: DateType, dateMultiplier: Double): void
Cineplex
- cinemas: ArrayList<Cinema>
- name: String
+ Cineplex(name: String): void
+ createCinema(cinemaCode: String, layout: boolean[][], cinemaClass: CinemaClass): void
+ getCinemas(): ArrayList<Cinema>
+ getName(): String
+ getLabel(): String
+ getShowTimes(): ArrayList<ShowTime>
Cinema
- cinemaCode: String
- layout: boolean[][]
- cinemaClass: CinemaClass
- showTimes: ArrayList<ShowTime>
+ Cinema(cinemaCode: String, layout: boolean[][], cinemaClass: CinemaClass): void
+ createShowTime(startDateTime: LocalDateTime, movie: Movie): void
+ getCinemaCode(): String
+ getLabel(): String
+ getLayout(): boolean[][]
+ getCinemaClass(): CinemaClass
+ getShowTimes(): ArrayList<ShowTime>
+ removeShowTime(s: ShowTime): void
«enumeration» CinemaClass
NORMAL
PLATINUM_MOVIE_SUITE
- label: String
- CinemaClass(label: String): void
+ getLabel(): String
«enumeration» MovieType
REGULAR
BLOCKBUSTER
3D
- label: String
- MovieType(label: String)
+ getLabel(): String
«enumeration» ReleaseRating
G
PG
PG13
NC16
M18
R21
- label: String
- ReleaseRating(label: String)
+ getLabel(): String
«enumeration» ShowingStatus
COMING_SOON
PREVIEW
NOW_SHOWING
END_OF_SHOWING
- label: String
- ShowingStatus(label: String)
+ getLabel(): String
Movie
- title: String
- synopsis: String
- director: String
- cast: String[]
- showingStatus: ShowingStatus
- releaseRating: ReleaseRating
- movieType: MovieType
- reviewRatings: ArrayList<ReviewRating>
- showTimes: ArrayList<ShowTime>
- duration: Duration
+ Movie(title: String, synopsis: String, director: String, cast: String[], showingStatus: ShowingStatus, releaseRating: ReleaseRating, movieType: MovieType, duration: Duration): void
# addShowTime(showTime: ShowTime): void
+ getCinemaTotal(): double
+ getTotalSales(): double
+ getTitle(): String
+ getSynopsis(): String
+ getDirector(): String
+ getCast(): String[]
+ getReleaseRating(): ReleaseRating
+ getMovieType(): MovieType
+ getReviewRatings(): ArrayList<ReviewRating>
+ getShowingStatus(): ShowingStatus
+ getDuration(): Duration
+ setTitle(title: String): void
+ setSynopsis(synopsis: String): void
+ setDirector(director: String): void
+ setCast(cast: String[]): void
+ setReleaseRating(releaseRating: ReleaseRating): void
+ setMovieType(movietype: MovieType): void
+ setDuration(duration: Duration): void
+ setShowingStatus(s: ShowingStatus): void
ShowTime
- cinema: Cinema
- startDateTime: LocalDateTime
- movie: Movie
- bookings: ArrayList<Booking>
ShowTime(cinema: Cinema, startDateTime: LocalDateTime, movie: Movie): void
+ createBooking(movieGoer: MovieGoer, selectedSeats: boolean[][], price: double): Booking
+ getSeatAvailabilities(): SeatStatus[][]
+ checkAvail(selectedSeat: boolean[][]): boolean
+ checkFull(): boolean
+ getTotalSales(): double
+ getCinema(): Cinema
+ getLabel(): String
+ getLayout(): boolean[][]
+ getDate(): LocalDate
+ getStartDateTime(): LocalDateTime
+ getMovie(): Movie
+ getBookings(): ArrayList<Booking>
+ getCinema(): Cinema
+ setMovie(movie: Movie): void
+ setStartDateTime(startDateTime: LocalDateTime): void
+ remove(): void
«interface» BookShow
+ getSeatAvailabilities(): SeatStatus[][]
+ checkAvail(selectedSeat: boolean [][]): boolean
+ getLayout(): boolean
«enumeration» SeatStatus
NO_SEAT
EMPTY
TAKEN
ReviewRating
+ MIN_RATING: int
+ MAX_RATING: int
- movieGoer: MovieGoer
- review: String
- rating: int
+ createReviewRating(movieGoer: MovieGoer, review: String, rating: int): ReviewRating
+ getMovieGoer(): MovieGoer
+ getReview(): String
+ getRating(): int
+ getLabel(): String
Booking
- transactionId: String
- movieGoer: MovieGoer
- tickets: ArrayList<Ticket>
- price: double
Booking(transactionID: String, movieGoer: MovieGoer, selectedSeats: boolean[][], price: double): void
+ getTransactionId(): String
+ getMovieGoer(): MovieGoer
+ getTickets(): ArrayList<Ticket>
+ getPrice(): double
Ticket
- serialVersionUID: long
- row: int
- col: int
+ Ticket(row:int, col:int): Ticket
+ getRow(): int
+ getCol(): int
«interface» LabelledItem
+ getLabel(): String
«enumeration» AgeGroup
CHILD
ADULT
SENIOR CITIZEN
- label: String
- AgeGroup(label: String)
+ getLabel(): String
«enumeration» DateType
WEEKDAY
WEEKEND
HOLIDAY
- label: String
- DateType(label: String)
+ getLabel(): String

UML Sequence Diagram



User Case Flow Description

Search/List Movie

When a user selects "View movie details" option in the movie goer menu, a `MovieController` object is instantiated and its `start()` method is called. Inside the `start()` method, the object calls its `selectMovie()` method. This method then calls the `DataManager` class to retrieve a list of all movies from the data store and the `movieList`

is returned. Then, the `MenuView` class is called to display the list of movies and get the selected movie from the user. The `MenuView` class calls each movie object's `getLabel()` method which returns the title of the movie object. It then displays titles of each movie. After the user selects a movie, the selected movie is returned to the `start()` method. Then, the `MenuView` class is called again to get input from the user.

View Movie Details

If the user selects the "View movie details" option (option 1), the `MovieView` class is called to print details of the movie using the `printMovieDetails(Movie selectedMovie)` method. The `MovieView` gets details about the movie by calling the selected movie object's get methods and displays them.

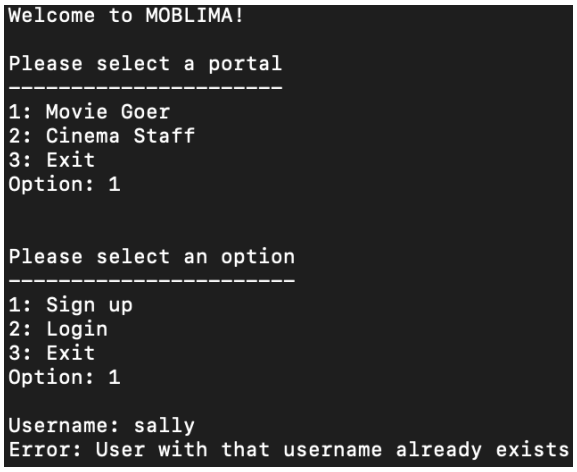
View Past Reviews and Ratings

If the user selects the "View past reviews & ratings" option (option 2), the selected movie object is called to get a list of all its `ReviewRating` objects. The objects are passed to the `ListView` class using the `displayLabelledItemList()` method. To get reviews and ratings as a `String`, the `ReviewRating` objects' `getLabel()` method is called. The `ListView` then display list of `String` objects to the user.

Test Cases and Results

Registering for an Existing Username

If a moviegoer tries to sign up with the same username as an existing moviegoer, then we will show an error message that will inform the user the username is not available. The screenshot below shows what happens when a user tries to sign up with the existing username “sally”.



Booking Full Show Times

If a moviegoer tries to book a show time with no more empty seats left, we will show an error message that will inform the user that the show time is full. The screenshot below shows that after the last seat is booked, no more users can book a seat in the same show time.

```
                                                SCREEN
-------------------------------------------------------------------------------------------------------------
    1    2    3    4    5    6    7    8    9    10   11   12   13   14   15   16   17   18   19
-------------------------------------------------------------------------------------------------------------
A [ x ][ x ][ x ][ x ]     [ x ][ x ][ x ][ x ][ x ][ x ][ x ][ x ][ x ][ x ]     [ x ][ x ][ x ][ x ] A
B [ x ][ x ][ x ][ x ]     [ x ][ x ][ x ][ x ][ x ][ x ][ x ][ x ][ x ][ x ]     [ x ][ x ][ x ][ x ] B
C [ x ][ x ][ x ][ x ]     [ x ][ x ][ x ][ x ][ x ][ x ][ x ][ x ][ x ][ x ]     [ x ][ x ][ x ][ x ] C
D [ x ][ x ][ x ][ x ]     [ x ][ x ][ x ][ x ][ x ][ x ][ x ][ x ][ x ][ x ]     [ x ][ x ][ x ][ x ] D
E [ x ][ x ][ x ][ x ]     [ x ][ x ][ x ][ x ][ x ][ x ][ x ][ x ][ x ][ x ]     [ x ][ x ][ x ][ x ] E
F [ x ][ x ][ x ][ x ]     [ x ][ x ][ x ][ x ][ x ][ x ][ x ][ x ][ x ][ x ]     [ x ][ x ][ x ][ x ] F
G                                                                                                   G
H [ x ][ x ][ x ][ x ]     [ x ][ x ][ x ][ x ][ x ][ x ][ x ][ x ][ x ][ x ]     [ x ][ x ][ x ][ x ] H
I [ x ][ x ][ x ][ x ]     [ x ][ x ][ x ][ x ][ x ][ x ][ x ][ x ][ x ][ x ]     [ x ][ x ][ x ][ x ] I
J [ x ][ x ][ x ][ x ]     [ x ][ x ][ x ][ x ][ x ][ x ][ x ][ x ][ x ][ x ]     [ x ][ x ][ x ][ x ] J
K [   ][ x ][ x ][ x ]     [ x ][ x ][ x ][ x ][ x ][ x ][ x ][ x ][ x ][ x ]     [ x ][ x ][ x ][ x ] K
-------------------------------------------------------------------------------------------------------------
    1    2    3    4    5    6    7    8    9    10   11   12   13   14   15   16   17   18   19
-------------------------------------------------------------------------------------------------------------
                                                ENTRANCE
Press enter to continue...

How many seats would you like to book: 1
Enter the seat no.s (e.g. A1):
K1
How many of each age group?
Child: 1
Adult: 0
Senior Citizen: 0
Confirm booking (y/n): y
```

```
                                                SCREEN
-------------------------------------------------------------------------------------------------------------
    1    2    3    4    5    6    7    8    9    10   11   12   13   14   15   16   17   18   19
-------------------------------------------------------------------------------------------------------------
A [ x ][ x ][ x ][ x ]     [ x ][ x ][ x ][ x ][ x ][ x ][ x ][ x ][ x ][ x ]     [ x ][ x ][ x ][ x ] A
B [ x ][ x ][ x ][ x ]     [ x ][ x ][ x ][ x ][ x ][ x ][ x ][ x ][ x ][ x ]     [ x ][ x ][ x ][ x ] B
C [ x ][ x ][ x ][ x ]     [ x ][ x ][ x ][ x ][ x ][ x ][ x ][ x ][ x ][ x ]     [ x ][ x ][ x ][ x ] C
D [ x ][ x ][ x ][ x ]     [ x ][ x ][ x ][ x ][ x ][ x ][ x ][ x ][ x ][ x ]     [ x ][ x ][ x ][ x ] D
E [ x ][ x ][ x ][ x ]     [ x ][ x ][ x ][ x ][ x ][ x ][ x ][ x ][ x ][ x ]     [ x ][ x ][ x ][ x ] E
F [ x ][ x ][ x ][ x ]     [ x ][ x ][ x ][ x ][ x ][ x ][ x ][ x ][ x ][ x ]     [ x ][ x ][ x ][ x ] F
G                                                                                                   G
H [ x ][ x ][ x ][ x ]     [ x ][ x ][ x ][ x ][ x ][ x ][ x ][ x ][ x ][ x ]     [ x ][ x ][ x ][ x ] H
I [ x ][ x ][ x ][ x ]     [ x ][ x ][ x ][ x ][ x ][ x ][ x ][ x ][ x ][ x ]     [ x ][ x ][ x ][ x ] I
J [ x ][ x ][ x ][ x ]     [ x ][ x ][ x ][ x ][ x ][ x ][ x ][ x ][ x ][ x ]     [ x ][ x ][ x ][ x ] J
K [ x ][ x ][ x ][ x ]     [ x ][ x ][ x ][ x ][ x ][ x ][ x ][ x ][ x ][ x ]     [ x ][ x ][ x ][ x ] K
-------------------------------------------------------------------------------------------------------------
    1    2    3    4    5    6    7    8    9    10   11   12   13   14   15   16   17   18   19
-------------------------------------------------------------------------------------------------------------
                                                ENTRANCE
```

```
Select a Show Time
------------------
1: 2019-12-01  09:30 (GV3)
2: 2019-12-01  11:15 (GV1)
3: 2019-12-01  12:00 (GV2)
4: 2019-12-02  16:00 (GV1)
5: 2019-12-02  17:45 (GV2)
6: 2019-12-03  13:00 (GV2)
7: 2019-12-03  14:15 (GV3)
Option: 2

Sorry, this show time is fully booked
```

Clashing Show Times

When a cinema staff attempts to create a new show time or modify an existing show time, we check whether the timing clashes with an existing show time in the same cinema. If there is a

clash, we don’t allow the cinema staff to change the show time. The screenshot below shows what happens when we try to change a show time’s timing from 09:00 - 10:00 to 11:30 - 12:30. Since there’s a clash with an existing show time in the same cinema (11:00 - 12:00), an error message is displayed and the show time is not updated.