







NANYANG
TECHNOLOGICAL
UNIVERSITY

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 the 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
Andrew Wee Chin Ho	CZ2002	SS3	 16/11/2019
Larry Lee Ken Ann	CZ2002	SS3	 16/11/2019
Siek Ming Kang	CZ2002	SS3	 16/11/2019
Hussain Khozema Kheriwala	CZ2002	SS3	 16/11/2019
Jeremy Book Kay Yip	CZ2002	SS3	 16/11/2019
Wong Ying	CZ2002	SS3	 16/11/2019

Design Considerations in MOBLIMA

Encapsulation - Booking, Cinema, Cineplex, Movie, MovieShowing, Payment, Review, Seat

Encapsulation protects an object's private data from illegal access by other classes. We enforced the concept of encapsulation into our application design by making attributes within a class private. We allow access to the attributes through public setter and getter methods in the object's class. For example, in the Movie class, if you would like to add the Movie's Synopsis, you would have to go through the setSynopsis method as it is the access point to the instance variable in the class.

Reusability - PersonMgr, PersonInterface, Booking, Payment, Review, Seat

PersonMgr is a general boundary class that houses common methods like calculateRating. This allows the movieGoerMgr and staffMgr to gain access to such functions which are needed by both the classes.

Booking, Payment, Review are common entity classes with common attributes and general methods that would likely appear in other programs such as an application for booking a seat in a restaurant. Hence, the existing code can be effectively used when developing code for future implementation while ensuring it maintains its usability.

Loose Coupling & High Cohesion - Payment, Review, Seat

Our classes are created with specific functions in mind. This allows our code to attain high cohesion & loose coupling. For example, in our Seat class, the Seat is not coupled to any movie theatre or cinema as there was no need for the seat to know about the theatre. Hence, while ensuring that there is high cohesion, we have also managed to attain low coupling.

Abstraction - PersonInterface, MovieGoerInterface, StaffInterface

Abstraction is the process of selecting important details of an object while hiding the insignificant ones. Abstraction helps reduce the complexity of the design and implementation process. In our application, PersonInterface is an abstract class that cannot be instantiated. MovieGoerInterface and StaffInterface inherit from PersonInterface. PersonInterface is a general boundary class that houses all the printing and scanning methods with the sole purpose of displaying and accepting inputs in the appropriate controllers. Specific methods like scanLine and displayLine are reused by multiple classes in the package. Hence, by using abstraction, we can encourage reusability while keeping our design simple and clean.

Polymorphism & Inheritance - PersonInterface, MovieGoerInterface, StaffInterface

Polymorphism allows an object to take on many forms. It occurs when we have many classes that are related to each other by inheritance. Our application adopts the polymorphism and inheritance concept which allows a new class to inherit all non-private attributes and methods from an existing class. For example, the StaffInterface and MovieGoerInterface class inherit from the abstract class PersonInterface. The concept allows us to build on existing work without having to create classes from scratch.

Singleton Design Pattern - MovierGoerMgr, StaffMgr

Singleton design pattern allows for only one instance of an object to be instantiated for the entire application lifecycle and provides a global access to that instance. Our application uses the *lazy initialization tactic* for MovierGoerMgr and StaffMgr. Each time the function getInstance() is called, if an instance of the object already exists, we will return the existing object. If the object does not already exist, we would proceed with creating a new instance of that object. This follows the Singleton Design Pattern as we can restrict the number of instantiating calls, we make throughout the application lifecycle. Hence, the entire application is made to share the manager object created, which removes the potential risk of having multiple instances of the same manager class.

Single Responsibility Principle - Entities: Seat, Review, MovieShowing, Payment

All our entities are only responsible for the attributes related to them. The entity classes are simply there to return values. For example, the MovieShowing entity class is responsible to return the date and time of the selected movie. This principle is well followed through throughout all entities, ensuring that a class should have one, and only one, reason to change.

Single Responsibility Principle - Controllers: StaffMgr, MovieGoerMgr

Their main function is to process the staff and moviegoer choice of action respectively by calling the relevant entity classes. For example, when a staff decides to update ticket pricing, it will call the Movie entity class to set the new price of a selected movie category. Or when a moviegoer wants to book a movie, it will call the Booking entity class and MovieShowing entity class to display relevant options for the customer.

Open-Closed Principle - PersonInterface, MovieGoerInterface, StaffInterface

We created 2 different interfaces, which will both inherit from the PersonInterface. By doing so, more interfaces (eg. VIPInterface), can be created without having to change the existing base class in the future. Within each interface, specific methods can be added to it, allowing the additional method to be implemented to the original code without affecting the functionality of the existing code. This design makes the code extendable.

Liskov Substitution Principle - displayMenu()

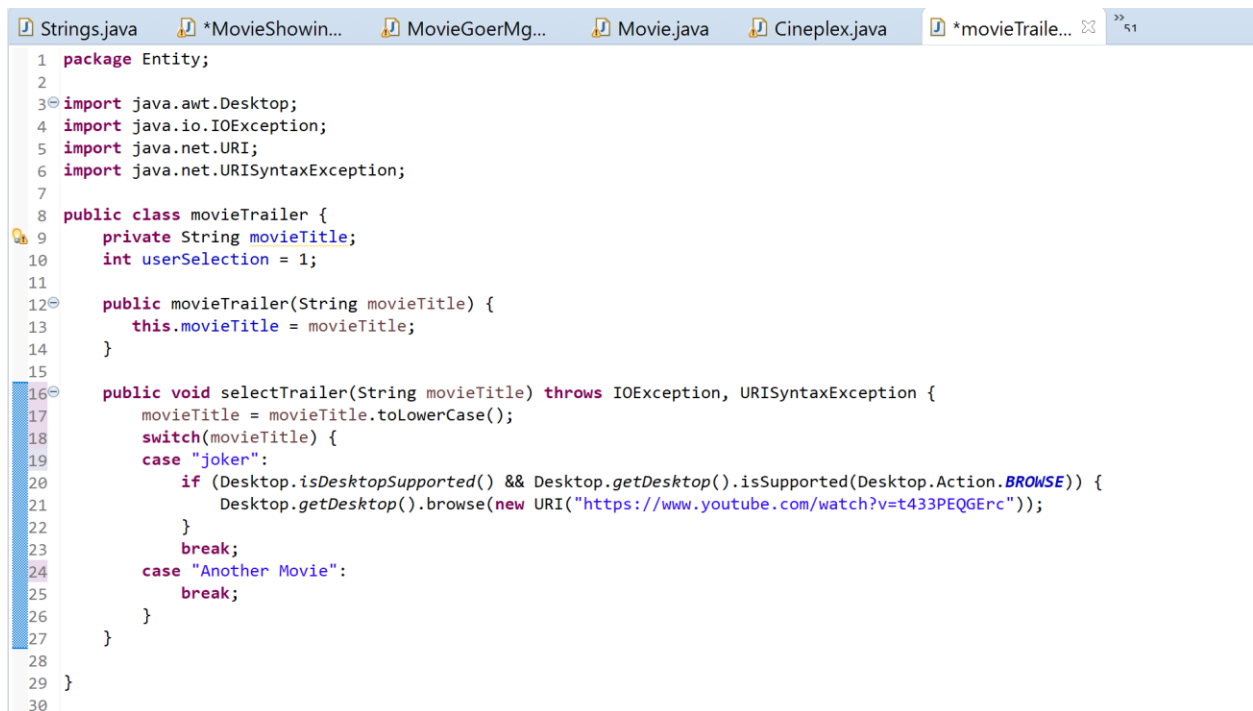
The Liskov Substitution Principle comes into play as methods used in the base class objects are also able to use the subclass objects without there being a difference. For example, displayMenu() is in all PersonInterface, StaffInterface and MovieGoerInterface. displayMenu() pre-conditions are no stronger than the base class method and its post-conditions are no weaker than the base class. This way, we are able to make our derived classes substitutable for their base classes and apply the Liskov Substitution Principle.

Future Feature Enhancements 1

Creating a Movie Trailer class

To further enhance the project, we would like to add an option for the user to watch the trailer for the movie that he is interested in. We can cater to the new design by following the Loose Coupling High Cohesion and Single Responsibility principles. We will create a new method that can be instantiated within movieGoerMgr for the sole purpose of redirecting users to the respective movie trailers. Since we are not directly editing any classes, we will be able to execute this implementation smoothly. An example of how the class will look like is attached.

Example:



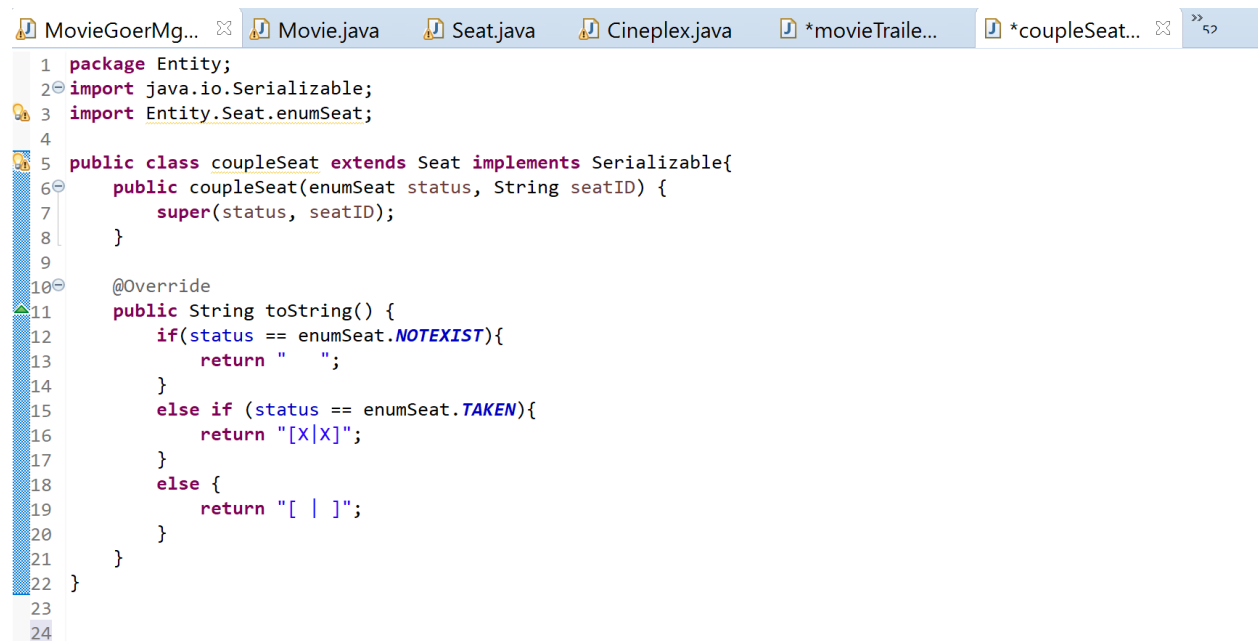
```
1 package Entity;
2
3 import java.awt.Desktop;
4 import java.io.IOException;
5 import java.net.URI;
6 import java.net.URISyntaxException;
7
8 public class movieTrailer {
9     private String movieTitle;
10    int userSelection = 1;
11
12    public movieTrailer(String movieTitle) {
13        this.movieTitle = movieTitle;
14    }
15
16    public void selectTrailer(String movieTitle) throws IOException, URISyntaxException {
17        movieTitle = movieTitle.toLowerCase();
18        switch(movieTitle) {
19            case "joker":
20                if (Desktop.isDesktopSupported() && Desktop.getDesktop().isSupported(Desktop.Action.BROWSE)) {
21                    Desktop.getDesktop().browse(new URI("https://www.youtube.com/watch?v=t433PEQGerc"));
22                }
23                break;
24            case "Another Movie":
25                break;
26        }
27    }
28 }
29
30 }
```

Future Feature Enhancements 2

Adding Couple Seats to the Booking Selection

To further replicate the real-life movie booking functions, we would include the option for users to purchase a couple seat in our application. We can cater to the new design by simply following Reusability, Inheritance and Open-Close principles. We will create a new class called coupleSeats. Because our previous Seat class followed the Single Responsibility Principle, we can easily extend and inherit the Seat class. We will then override certain methods to suit the needs of the couple seat. An example of how we will carry out the extension is attached.

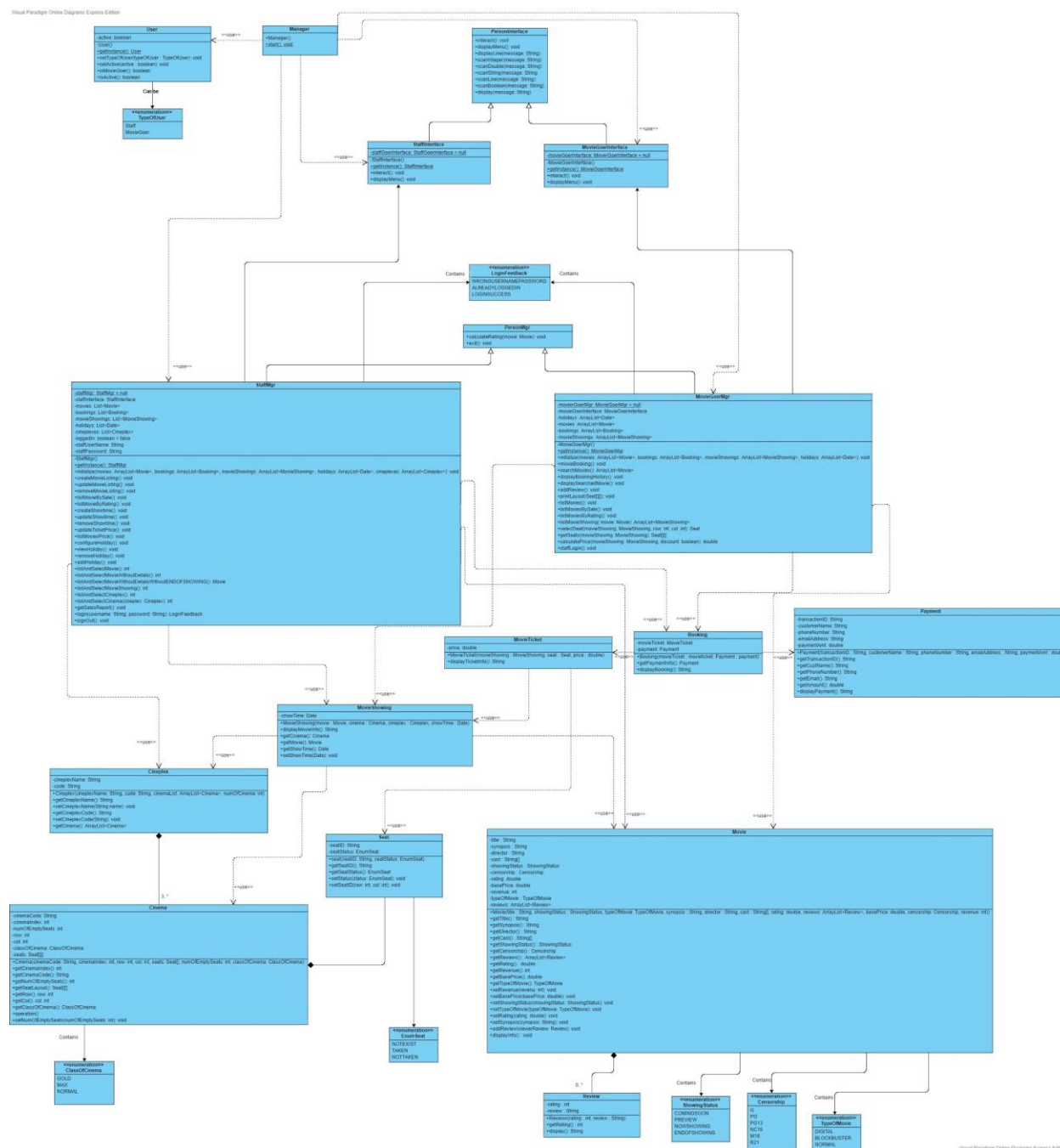
Example:



```
1 package Entity;
2 import java.io.Serializable;
3 import Entity.Seat.enumSeat;
4
5 public class coupleSeat extends Seat implements Serializable{
6     public coupleSeat(enumSeat status, String seatID) {
7         super(status, seatID);
8     }
9
10    @Override
11    public String toString() {
12        if(status == enumSeat.NOTEXIST){
13            return " ";
14        }
15        else if (status == enumSeat.TAKEN){
16            return "[X|X]";
17        }
18        else {
19            return "[ | ]";
20        }
21    }
22 }
```

Class Diagram

Visual Paradigm Online Diagrams Editors Edition



Visual Functions Online Dictionary Express Editor

Sequence Diagram

User case flow description

When MovieGoer user starts the application, MovieGoerInterface will call displayMenu().

MovieGoerInterface uses a switch statement to check which function MovieGoer wants to use.

To list movies, MovieGoer inputs 1 and MovieGoerMgr call listMovies(). MovieGoerMgr will loop through the movies stored in movie file, and for each movie, if ShowingStatus != ENDOFSHOWING, the movie will be added to ArrayList movie1. For each movie in movie1, the movie will call Movie's displayInfo() which will call getTitle(), getShowingStatus(), getDirector(), getCast(), getCensorship() and getSynopsis() to show movie details. It will call Movie's getReview() and return the ArrayList for reviews. For each review in reviews, it will call Review's own display() and getRating() to show reviews of the movie.

To search for movies, MovieGoer inputs 2 and MovieGoerMgr call displaySearchedMovie(). MovieGoerMgr will call searchMovies(). MovieGoer will input movieName. MovieGoerMgr will loop through the movies stored in movie file and for each movie, if movie.getTitle().contains(movieName) && (ShowingStatus != ENDOFSHOWING || ShowingStatus != COMINGSOON), the movie will be added into ArrayList searchResult. searchResult will then be returned to ArrayList movies. Similarly, for each movie in movies, the movie will call Movie's displayInfo() which will call getTitle(), getShowingStatus(), getDirector(), getCast(), getCensorship() and getSynopsis() to show movie details. It will call Movie's getReview() and return the ArrayList reviews. For each review in reviews, it will call Review's own display() and getRating() to show reviews of the movie.

TESTING

We created several customer accounts of varying age groups to test calculation of ticket price discounts.

Compulsory Test Cases

1. Configuring a holiday date and the ticket price is shown correctly when booking is done on that date

Logs in to Customer account, starts to book movie, get a quote for ticket prices.

Logs in to Staff account, creates public holiday on the same day.

Log in to Customer, starts to book movie, chooses the same movie and day and see an increase in price.

2. Booking on a different day of the week or holiday and type of cinema (eg Gold Class to demonstrate the differences in prices)
 1. show the different days of the week (Weekends more expensive than Weekdays)
 2. show different cinema type (Normal class cheaper than Gold class)

1. Log in to Customer account, book a movie on 17/11/2019 (Sunday) and quote price (\$12).

Log in to Customer account, book the exact same movie at the same location on 20/11/2019 (Monday) and quote price (\$10)

2. Log in to Customer account, book a movie showing in Normal Class cinema and quote price (\$10).

Log in to Customer account, book the same movie showing but in Gold Class cinema and quote price (\$12).

3. Configuring “End of Showing” date and the movie should not be listed for booking

Enter customer account and enter option 2, press enter again to list all movies and its showing status. (Avengers: Endgame is NOWSHOWING)

Enter staff account, update movie listing (Avengers: Endgame) to “End of Showing”.

Enter customer account, go to booking option, movie listing (Avenger: Endgame) is not available for booking.

4. Booking only allowed for “Preview” and “Now Showing” status.

Enter customer account, list all movies with option 1, we show the existence of movie listings with “Coming Soon” showing status (Zombieland: the next level, Parasite).

Enter the booking option, option 3, only movie listings with “Preview” and “Now Showing” showing status are listed. (Zombieland: the next level, Parasite is not listed)

Demonstrate app:

Movie-goer module

1. Attempt to book movie (Zombieland)
2. List all movies
3. List Top 5 by sales
4. List Top 5 by ratings
5. Search for movie (Crazy romance)
6. Book ticket (includes checking seat availability and selection of seats)
7. View booking history

Admin module

1. Login
2. Create, Update and Remove movie listing
3. Create, Update and Remove cinema showtimes and the movies to be shown
4. Configure system settings (adjust base price, holidays, return total number of tickets sold for the day)

Appendix A - Essential Information

Staff Account

<u>Username</u>	<u>Password</u>
1	1