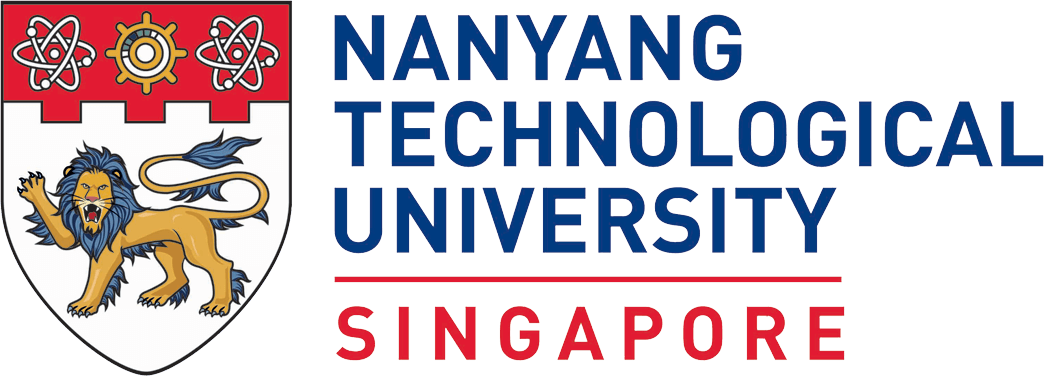
****

**College of Engineering**

**School of Computer Science & Engineering**

**CZ2002**

**Object Oriented Design & Programming**

**AY 2019/20 S1**

**Building an OO Application**

**Project Report**

|  |  |
| --- | --- |
| **Member’s Names** | **Matriculation Number** |
| Eric Lee Jun Hong | U1820995G |
| Cecil Street |  |
|  |  |
|  |  |
|  |  |
|  | |
| **Tutorial Class, Group :** | **SS2, Group 1** |
| **Date of Submission :** | **16/11/2019** |

Contents

[1. Introduction 3](#_Toc24755336)

[2. UML Class Diagram 5](#_Toc24755337)

[3. UML Sequence Diagram 6](#_Toc24755338)

[4. Design Principles, Considerations & Object-Oriented Concepts 7](#_Toc24755339)

[a. Design Considerations 7](#_Toc24755340)

[i. Data Access 7](#_Toc24755341)

[ii. Loose Coupling 7](#_Toc24755342)

[iii. High Cohesion 7](#_Toc24755343)

[b. SOLID Principles 8](#_Toc24755344)

[i. Single Responsibility Principle 8](#_Toc24755345)

[ii. Open-Closed Principle 8](#_Toc24755346)

[iii. Liskov Substitution Principle 8](#_Toc24755347)

[iv. Interface Segregation Principle 8](#_Toc24755348)

[v. Dependency Inversion Principle 8](#_Toc24755349)

[c. Object-Oriented Concepts 9](#_Toc24755350)

[I. Object-Oriented Abstraction (Detail Hiding) 9](#_Toc24755351)

[II. Object-Oriented Encapsulation (Information Hiding) 9](#_Toc24755352)

[III. Inheritance & Polymorphism 9](#_Toc24755353)

[5. Test Case 10](#_Toc24755354)

# Introduction

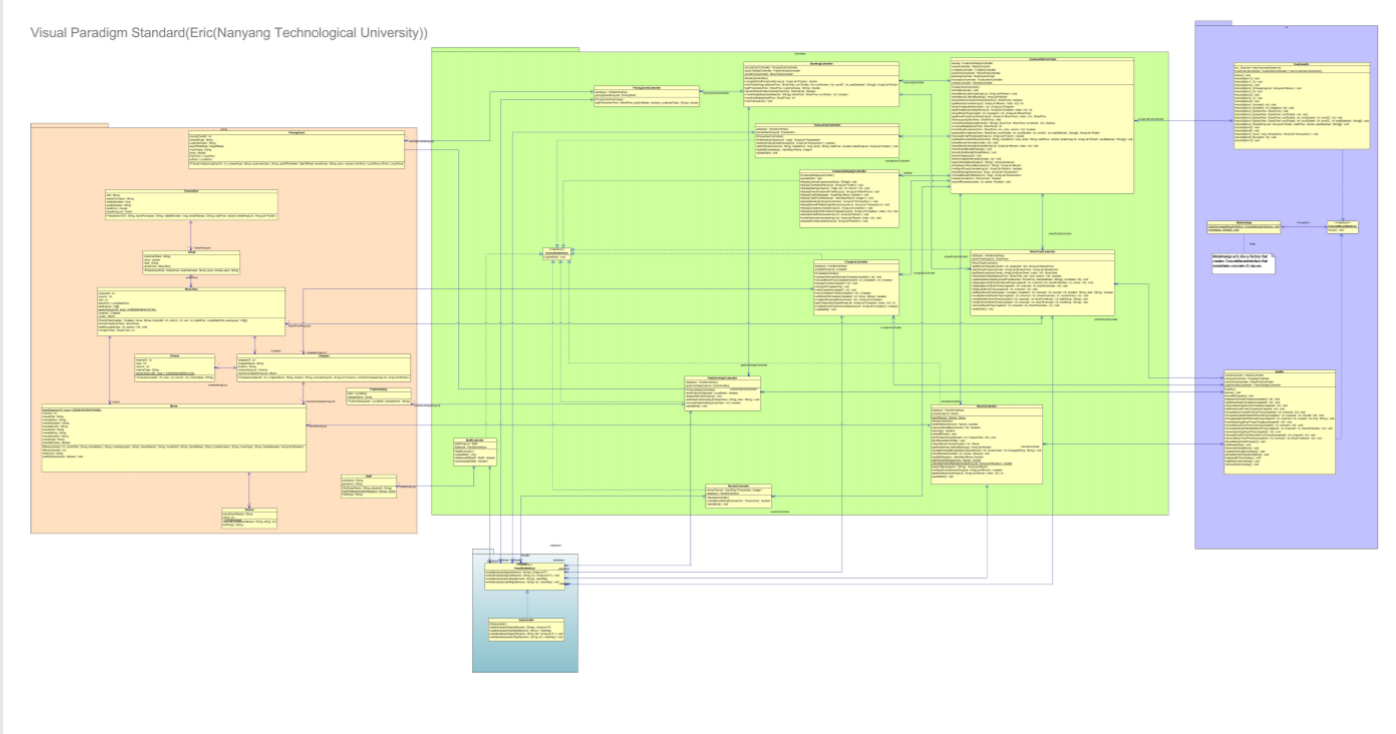
This assignment report will describe and explain the functionalities of our Console-based application, Movie Booking and Listing Manager Application (MOBLIMA). It will also include a UML class diagram, sequence diagrams and explanations of some of the design principles, considerations and object-oriented concepts that we have incorporated into our application.

Moblima is an application that is geared towards both users and staff alike. It allows users to view available movie and showtimes and make a Movie booking through the application. It also allows cinema staff to modify movie listings and showtimes.

As stated in the assignment requirements, here are some of the assumptions we follow while building this application.

* + 1. This is a single-user application and there is no need to consider concurrent access
    2. Three cineplexes will be created.
    3. Currency of transaction will be in Singapore Dollar (SGD) and inclusive of Good and Services Tax (GST).
    4. Payment will always be successful.
    5. There is no need to interface with external system, e.g. Payment, printer, etc., but you can consider it in your design.
    6. Senior citizen can be purchased online without validation of identity or age. The validation will be done upon entering the cinema.

# UML Class Diagram



A clearer combined class diagram can be found in the folder!

# UML Sequence Diagram

# Design Principles, Considerations & Object-Oriented Concepts

## Design Considerations

The way we designed our program encompasses the flow of the Entity-Boundary-Control (EBC) design. The EBC design is a form of stereotyping which helps users to easier understand the flow of the code. More on these design considerations will be documented below!

### Data Access

For our data files, we store and retrieve data from a file format. We make use of FileOutputStream to serialize our data to be stored in the file format. For example, in our application, we make use of a single concrete class DataHandler, which implements HandlerInterface, to read and write our data which is stored as a collection of ArrayList<T> through the generic methods of readSerializedObject and writeSerializedObject.

### Loose Coupling

To achieve loose coupling, we aim to reduce the dependencies of a class using a different class directly. For example, in our UI package, MoblimaApp class makes use of concrete class CustomerUi, StaffUi’s method show() to display information to the user. Without loose coupling, whenever a developer chooses to add a new UI class, modification must be made in MoblimaApp.

However, since we have designed CustomerUi and StaffUi to implement ConsoleBasedInterface that has a method show(), this allows the developer to easier add a new concrete Ui Class without changing much of original code. This is because MoblimaApp doesn’t target a concrete UI class, but rather an interface, this reduce the tight coupling and allows for easy extension (part of the open-closed principle).

### High Cohesion

To achieve High Cohesion, we must ensure that we follow the Single Responsibility Principle. Every class should only have one responsibility. In the grand scheme of things, this mean that we need to subdivide the program into different level of modules, namely the boundary classes, control class and entity class. For example, our boundary class is in charge of displaying and receiving the user’s choice. The control class will then receive this input and process this input to correctly access the right entity through a data access object (DAO) namely DataHandler. As such, there will not be a “swiss army knife” class that does everything.

The advantage of having a high Cohesion is that we can isolate the functionality into different classes, thereby reducing the complexity of each class and increase the system maintainability and reduce undesired bugs.

## SOLID Principles

### Single Responsibility Principle

As previously mentioned, single responsibility principle indicates that a Class should only have one responsibility and we shouldn’t have a “God” class that does everything. This means ensuring that our UI classes only have UI functionality and not have any logical processing functionality.

### Open-Closed Principle

Most of our top-level modules are designed to be implemented by concrete classes. We often make use of Abstract Classes or Interfaces to make sure that if there is a need to re-implement a feature or add a new Class, the base class will not need to be modified, hence adhering to the “closed for modification” principle. Furthermore, since new concrete classes can easily implement our top-level Abstract Classes/Interfaces, the program will be easily and readily “opened for modification”.

### Liskov Substitution Principle

When designing a class that will inherit/override features from our superclass/abstract class, we are always mindful to ensure that our subclass will never be more restrictive than the behavior of the superclass. This is important since if the subclass methods are more restrictive than the superclass/abstract class, then abstraction might not work since a lot of our classes rely on abstraction to adhere to dependency inversion principle. If Liskov substitution principle is not adhere to, a class associated with the superclass/abstract class’s subclass might not know how to handle the more restrictive method. Even worst, it might force the class to know how the restrictive method works, thus introducing tight coupling while breaking the rule of abstraction.

### Interface Segregation Principle

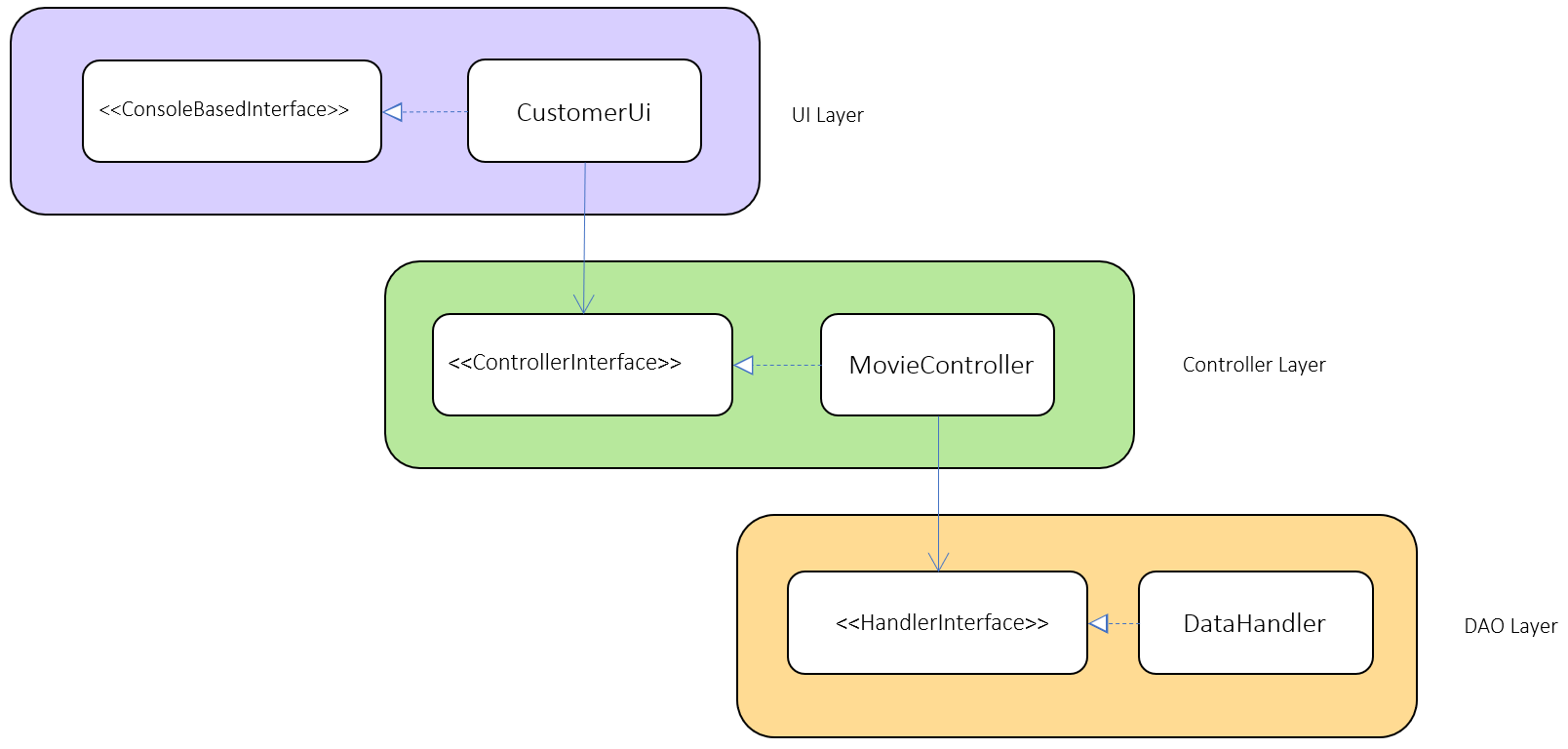
Interface segregation principle is an extension to the single responsibility principle for Interfaces. For example, most of our controller class such as CineplexController implements ControllerInterface. However, certainly controllers such as BookingControllers do not implement ControllerInterface because it doesn’t make use of any methods from ControllerInterface, thereby adhering to the interface segregation principle which states that no clients(Class) should be forced to depend on methods it doesn’t make use.

### Dependency Inversion Principle

Dependency inversion principle is a huge consideration for our project, since we deal with a 4-tier system architecture. This means that across layers, we must not make the mistake of allowing Classes from different tiers to be tightly coupled and depend on concretion. Hence to solve this issue, we introduce multiple Interface such:

1. Higher-Level modules do not depend on low-level modules. They depend on abstraction! (Interface)
2. Abstraction do not depend on the details(concrete implementation), the concrete implementation depends on the abstraction!

Hence with such a design, Dependency Inversion Principle is achieved.



## Object-Oriented Concepts

### Object-Oriented Abstraction (Detail Hiding)

Abstraction is a property where by only the essential details are displayed to the user(or framework or another class). Such example can be seen in the UI package. We have 2 concrete classes, namely CustomerUi and StaffUi that implements ConsoleBasedInterface. Moblima uses the method show(), found in CustomerUi and StaffUi. However, it doesn’t need to know what each method’s individual implementation does, it only needs to be aware of a show() method that was implemented from ConsoleBasedInterface.

### Object-Oriented Encapsulation (Information Hiding)

Encapsulation is a property where by the data and inner implementation of a class is hidden to other classes. Through the appropriate setting of access modifiers for classes, methods and attributes, we can ensure the data integrity of a class’s object since other classes cannot directly access and modify another’s class data! This allows for easy re-usability and data integrity as well as making the encapsulated code easier to test for unit testing.

### Inheritance & Polymorphism

Inheritance and polymorphism are an important concept in our project that allows one class to inherit the features of another class. Using the right polymorphic behavior, it allows our program to implement the correct design patterns/strategy. For example, strategy pattern allows a set of algorithms to be interchangeable. For example, our Interface, ControllerInterface has a method called updateDat(). Using the appropriate inheritance and polymorphism, it allows multiple class to implement this method to achieve multiple functionality based on run-time requirements.

It also allows for the usage of factory pattern, which is an important concept in our application. Since we do not know which classes needs to be instantiated until runtime, we can use the factory pattern, who relies heavily on polymorphic behavior, to achieve dynamic instantiation while hiding creational logic. This makes a program secure and easily usable by other external modules.

# Test Case

**Assumptions**

**New Features**

**Source Code**

The source code is exported

**Javadoc**

The Javadoc is exported

**Video Demonstration**

The video demonstration is saved

**UML Diagram Class**

**UML Sequence Diagram**

**Test Cases & Results**

|  |  |  |  |
| --- | --- | --- | --- |
| **Test Case 1 –** | | | |
| **Test Case** | | **Expected Outcome** | **Actual Outcome** |
|  |  |  |  |
| **Test Case 2 –** | | | |
| **Test Case** | | **Expected Outcome** | **Actual Outcome** |
|  |  |  |  |

**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** | **Lab Group** | **Signature** | **Date** |
|  |  |  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

~End of Report~