****

**SC2002 OBJECT ORIENTED DESIGN & PROGRAMMING**

**Build-To-Order (BTO) Management System**

**Report of Project Structure Design & Functionality**

**AY 24/25 SEM 2 | FCS1 Group 3**

**GitHub Main Page:**  [github.com/jacktan130802/SC2002-Assignment-](http://github.com/jacktan130802/SC2002-Assignment-)

**Declaration of Original Work for SC2002 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** |
| Cher Yue Yang | SC2002 | FCS1\_G3 |  |
| Goh Jia Hui | SC2002 | FCS1\_G3 |  |
| Jack Tan Xin Jie | SC2002 | FCS1\_G3 |  |
| Tan Wei Song Russell Bryan | SC2002 | FCS1\_G3 |  |

# **Requirement Analysis & Feature Selection**

**1.1 Understanding the Problem and Requirements**

In this assignment, we aim to create a Build-To-Order (BTO) System, a system for applicants, HDB officers, managers to view, apply and manage for BTO projects. It is designed in an object oriented way, having to achieve loose coupling and high cohesion. We began by reading through the BTO document line-by-line, highlighting all use cases and system requirements. The system is then implemented with specific capabilities for specific roles to enhance usability and security. Explicit requirements we enforce our system to implement are the sample test cases in the SC2002 2024S2 Assignment BTO.pdf. Implicit expectations include implementing such that the system acts like an actual BTO system, which also saves and persists data even when a user logs out. Any ambiguous and missing parts have been resolved by clarifying with Prof. For example, what does it mean that a Manager will not be able to create a new project if he/she already has an active project within the application period and what does active mean? We have taken additional efforts for clarification and then implemented it so that our system is complete as a whole.

**1.2 Deciding on Features and Scope**

After understanding the assignment thoroughly, we have identified the core and additional features to implement. The features are as follows:

|  |  |
| --- | --- |
| **Core features** | Valid User Login  Project Visibility Based on User Group and Toggle  Project Application  Viewing Application Status after Visibility Toggle Off  Single Flat Booking per Successful Application Applicant’s enquiries management  HDB Officer Registration Eligibility  HDB Officer Registration Status  Project Detail Access for HDB Officer  Response to Project Enquiries  Flat Selection and Booking Management  Create, Edit, and Delete BTO Project Listings  Toggle Project Visibility  View All and Filtered Project Listings  Manage HDB Officer Registrations  Approve or Reject BTO Applications and Withdrawals |
| **Other features (of less importance as compared to Core)** | Invalid NRIC Format  Incorrect Password  Password Change Functionality  Restriction on Editing Project Details  Single Project Management per Application Period  Receipt Generation for Flat Booking  Generate and Filter Reports |
| **Bonus features** | Centralised Login Menu  Mixed Filters  Saved Receipt Generation |

More details are elaborated in Section 5: Testings

The core features are chosen to prioritize the main structure of the system. After implementing them, we proceed to implement the other required features. This is to ensure that the concrete skeleton exists before adding specific details onto the core features, maintaining an error-free implementation of essential user experience. With the remaining time, the other features were put in place. We managed to successfully build all the features before the deadline, thus we have decided to add bonus features into our system, making the system more user-friendly.

# **System Architecture & Structural Planning**

# **2.1 Planning the System Structure**

Before implementation, our team has come up with the overall system layout. We broke down the system into multiple logical components, identifying main modules Applicants, HDB Officers, HDB Managers, Project, etc. In addition, we have identified other modules such as FlatBooking, ApplicationStatus, NRICValidator,etc. After identifying them, we list down all possible use cases and mapped them to the components accordingly. We have developed early visual flowcharts and diagrams via Visual Paradigm, outlining the basic classes and behaviours, grouping various components into entities, controllers and boundaries. With those drafts, we have a clearer idea of the important details to insert in the UML class and sequence diagrams.

## **2.2 Reflection on Design Trade-off**

## During the design of the BTO application system, we carefully considered several trade-offs, particularly our initial goal to apply SOLID principles across most classes using interfaces and abstract classes. However, given the limited time frame, implementing a large number of classes was not feasible. As a compromise, we chose to focus on five key components of the system where we could effectively demonstrate the SOLID principles while keeping the overall design manageable and maintainable.

## **Object-Oriented Design**

## **3.1 Detailed UML Class Diagram**

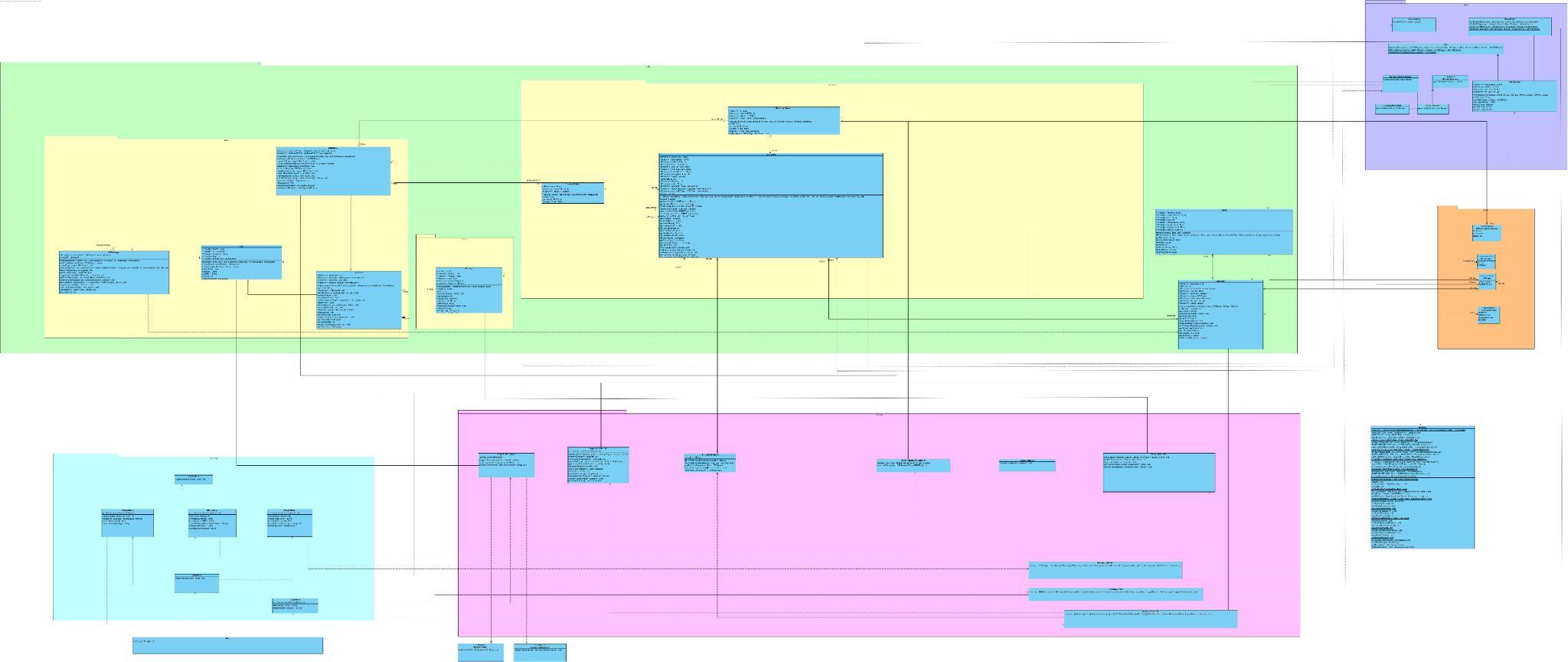
**3.1.1 Overview**

The UML Class Diagram has 3 main types of classes:

* **Entity Classes:** Represents the users and domain objects. In the context of this project, the users are Applicant, HDBManagers, HDBOfficers and the domain objects are Application, BTOProject.
* **Boundary Classes:** Handles the communication between users or other systems. Acts like an interface. (LoginMenu, ApplicantMenu)
* **Controller Classes:** Manages logic and acts as a coordinator between entity and boundary classes. (ManagerController, LoginAuthController)

**3.1.2 Diagram Description and Notes**

* **Relationships between Classes:**
  + Our class diagram shows the relationship between classes such as association, dependency and generalisations. E.g (Use generalisation for Applicant and HDBOfficer as Applicant is a parent class for HDBOfficer)
* **OO Principles:**
  + Annotations are made in our UML Class Diagram showing the different OO Principles such as encapsulation, inheritance, polymorphism and abstraction.

****

**Figure 1: Class Diagram**

# **3.2 Detailed UML Sequence Diagram of Stated Function**

The BTO system consists a many flows, including

A screenshot of a computer program

AI-generated content may be incorrect.

**Figure 2:** **Officer Booking Flat for Applicant**

This sequence diagram illustrates the flow of officer booking flats for applicants. This flow is crucial as it interacts with many entities including Flat, Applicant, Project and more. It reflects one of the core duties an Officer can perform. Officer can book flats on behalf of applicants, and displays the appropriate validations, whether the officer is handling a project, and checks whether applicant status is successful. With this flow, we can perform technical implementations fluently.

A diagram of a project

AI-generated content may be incorrect.

**Figure 3: Officer Registration for a Project**

This sequence diagram demonstrates the flow of HDB Officers registering for a project. This process validates that an HDB Officer cannot register a project that he has applied for, a project with an overlapping timeline, or a registered project. It checks that the system creates getRegisteredProject with the correct criterias. This shows the system process flow that we would need to implement to have a smooth and controlled flow.

A screenshot of a project

AI-generated content may be incorrect.

**Figure 4: Applicant Login**

This sequence diagram displays the detailed flow of user login validation. This flow is the first entry point for every user. Without this diagram for a clearer technical implementation, the other flows in the system might be affected. In this diagram, it illustrates user NRIC and password validation flow, maintaining a fluent flow of the authentication process and the display of the menu base of the user’s role. This ensures security and a comprehensive experience from the beginning.

With the help of these diagrams, we can perform in-depth technical implementations.

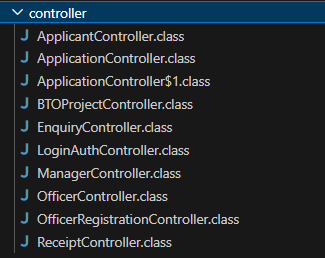
# **3.3 Application of OOD Principles (SOLID)**

Single Responsibility Principle (SRP)

The SRP states that one class should only have one main responsibility and should only require one reason for it to be changed.

Application in the project:

Under “Src/Controller”, various controller classes can be seen. Each managing a specific task.



**Figure 5**

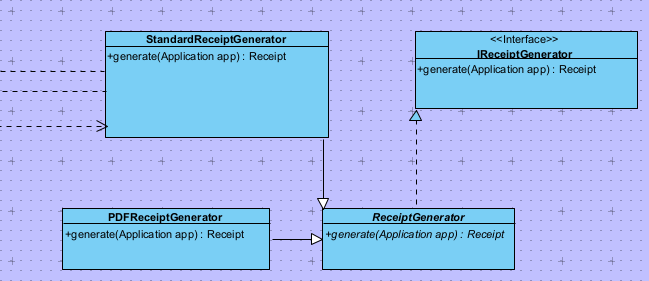
For instance, the ApplicantController, OfficerController and ManagerController classes each manage the flow for 3 different types of users (Applicant, Officer, Manager) in the BTO system respectively, instead of clustering all the flows and logic for different users in one class. By following SRP, we ensure that any changes made to the code will have minimal impact on other classes.

Open/Closed Principle (OCP)

The OCP states that software entities should be open for extension but closed for modification.

Application in the project:

The OCP is demonstrated by the ability to introduce new receipt generator classes (like PDFReceiptGenerator) by extending ReceiptGenerator without modifying any of the existing classes. This makes the system more maintainable and extensible.



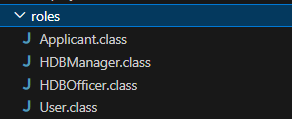
**Figure 6**

Liskov Substitution Principle

The LSP states that sub-types must be able to replace their super-types without breaking the program execution.

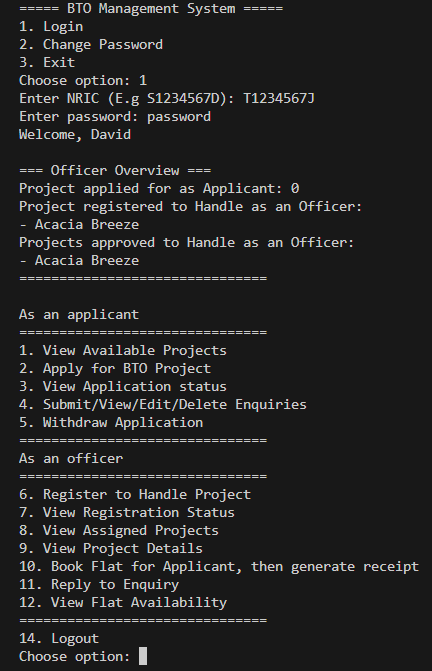
Application in the project:

Under “Src\entity\roles”, we would like to point out the LSP between Applicant and HDBOfficer classes.



**Figure 7**

A HDB Officer can act like an Applicant as long as the officer does not apply for the same project (as an applicant) as the project he is registering to handle (as an officer). Additionally, the HDB Officer has other features that the officer can do. As such, we implement HDB Officer as a subclass of the Applicant class such that a HDB Officer can substitute an Applicant in applying for the project. Attached below is the menu of the HDB Officer as evidence to justify that we kept the applicant portion for the officer. We also ensure that the rest of the additional features of the officer does not break or interfere with the superclass (Applicant).



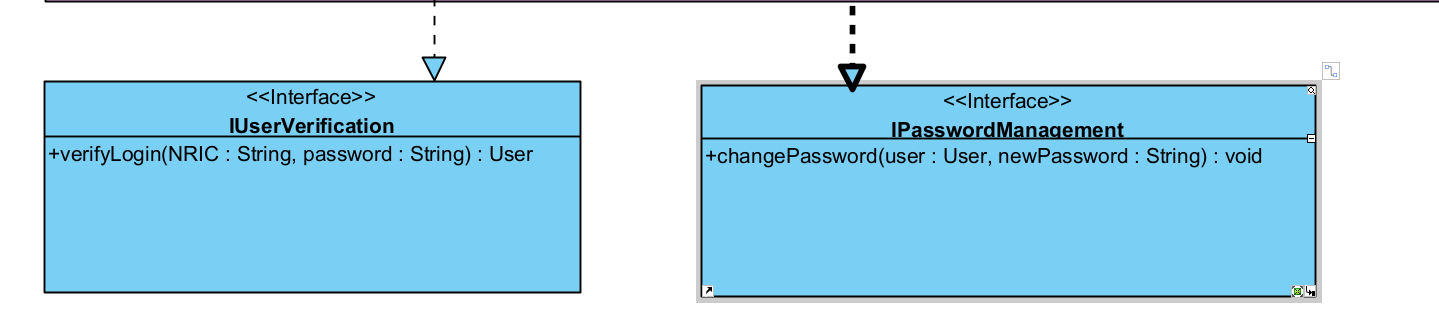
**Figure 8**

Interface Segregation Principle (ISP)

The ISP states that a client should not be forced to implement an interface it doesn’t use, and shouldn't be forced to depend on methods it does not use.

Application in the project:

Initially, the LoginAuthController Class depended on a large interface and other classes to implement the logic for logging in. This would then cause the LoginAuthController class to depend on other methods that it does not need. We improved by applying ISP to split the interfaces into smaller interfaces such as IPasswordManagement (changing of password) and IUserVerification (authentication). This ensures that the LoginAuthController only needs to depend on the interfaces and methods that it needs.



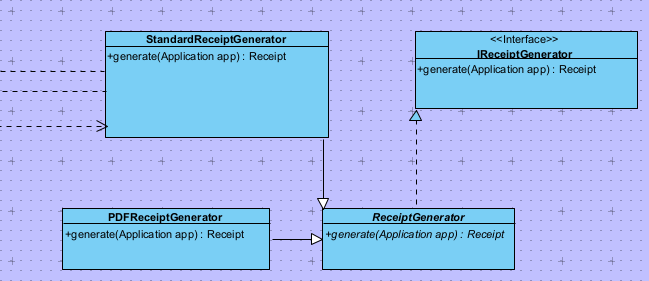
**Figure 9**

Dependency Inversion Principle (DIP)

The DIP states that a high level module should not depend on a low level module. Both should depend on abstractions.

Application in the project:

The DIP is achieved by ensuring that high-level classes depend on the abstraction IReceiptGenerator, rather than specific implementations. This allows flexibility and reduces coupling.



**Figure 10**

## **Implementation**

**4.1 Tools Used:**

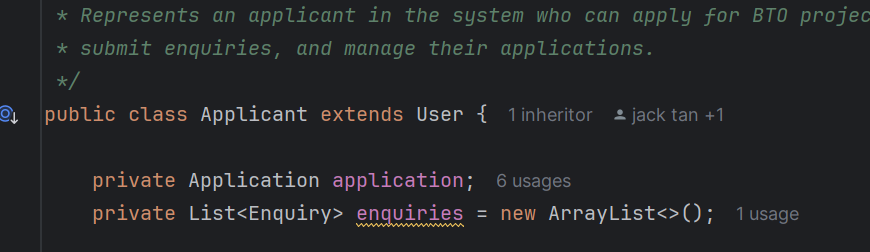
Java 17

Intellij/VS code

Version Control: Github, Git

## **4.2 Design Considerations - OO**

Inheritance



**Figure 11**

The Applicant class demonstrates inheritance by extending the User superclass. This allows Applicants to reuse attributes and methods defined in User.

Polymorphism

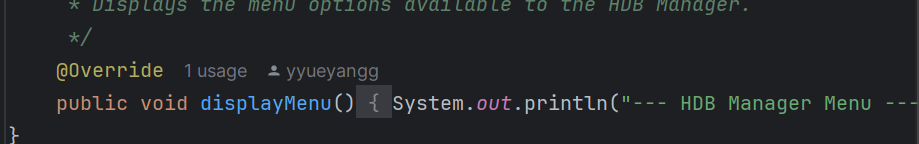
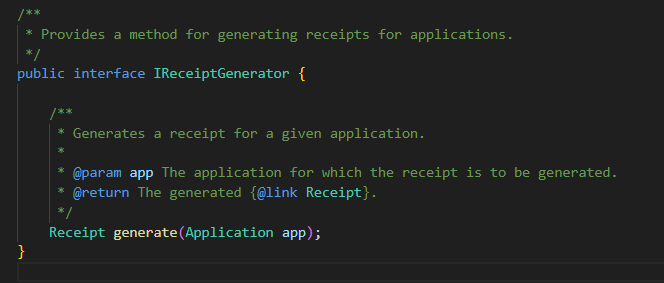


Figure 12

The User class is an abstract class with an abstract method displayMenu().

The HDBManager class, which extends User, overrides the displayMenu() method to provide a specific implementation for HDB managers.

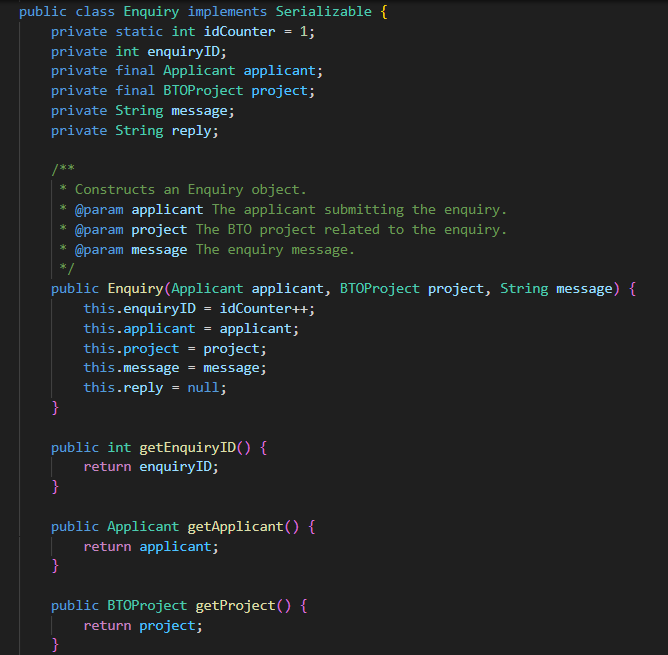
The other different roles, such as officer and manager menu also overrides the displayMenu() method in parent class so that they can display their respective menu.



**Figure 13**

Abstraction

The IReceiptGenerator interface illustrates abstraction by defining a common contract for generating receipts without exposing the implementation details. This allows for multiple interchangeable implementations and enables high level modules to depend on abstractions rather than concrete classes. This enhances flexibility and modularity.

  
**Figure 14**

Encapsulation

Classes such as Enquiry have private fields like age, enquiryID, applicant etc that should only expose what is necessary through getter methods. This ensures external classes cannot change the internal state of an Enquiry object.

## **Testings (ALL 23 SAMPLE TEST CASES PASSED)**

## **5.1 Test Strategy**

We tested our code using manual functional testing where we tested the code based on the different test cases provided. We manually interacted with the application by performing actions based on the different users, verifying the outcomes, error messages, and data updates. Manual functional testing focuses on overall system behavior, validating that each feature meets its functional requirements without diving into the code. This is useful for catching usability issues and ensuring the system flows logically.

## **5.2 Test Cases**

**Test cases can be viewed here:**

[**https://docs.google.com/document/d/104vc\_ylwUSdBVIQdPLJ4BXuTVly8kWqUAyt6OYnYLl4/edit?usp=sharing**](https://docs.google.com/document/d/104vc_ylwUSdBVIQdPLJ4BXuTVly8kWqUAyt6OYnYLl4/edit?usp=sharing)

## **Documentation**

**6.1 JavaDoc**

The JavaDoc can be found on Github. Please refer to “JavaDoc/index.html”

**6.2 Developer Guide:**

Please refer to our Github repository’s README at “How to Run the Application:”

## **Reflection**

Through developing this application, our team gained a deeper understanding of the different design principles and object-oriented concepts and how to design our code to adhere to them. Although we initially found some principles challenging and a chore to implement, we slowly recognised their benefits and purpose as development progressed. When implementing new features, these concepts allowed us to build modular, reusable code, reducing redundancy when multiple classes needed similar functionality. This modularity also made it easier to isolate and address bugs, and enabled us to add planned features with minimal changes to existing code, ultimately simplifying the development process, on top of making our system easier to maintain in the future.

Although implementing these principles initially felt challenging, tedious, and required considerable planning and foresight, we came to realise that this groundwork ultimately saved our team significant time and effort in the long run, while also simplifying collaboration. Initially, our team struggled with adhering to the single responsibility principle, as we found it challenging to define a clear, single responsibility for each class in a way that would aid our development. Eventually, we adopted a layered architecture, similar to MVC but with additional packages, to guide our code structure, helping us organise responsibilities more effectively and maintain a clear separation of concerns.

Additionally, our team found that one of the most challenging aspects of this project was creating the UML diagram, as it required careful identification of the necessary entities and clear understanding how they interacted with each other. However, once this foundation was established, we found that delegating segments for each team member to code, as well as implementing the required methods was much more straightforward. By referring to the completed UML diagram, each team member was aligned in terms of implementation, and our code worked together seamlessly despite the fact that we were working independently on distinct parts of the application. This was especially important given the tightly interconnected requirements of the application. This experience helped us realise the value of thoroughly planning with a complete UML diagram before beginning the implementation process

To further improve the program, our team feels that we can implement more abstract classes and interfaces. This is for loose coupling between classes, increasing modularity. We also feel that we could strengthen security by hashing the passwords before storing them in the respective CSV.

## **Appendix**

Our team used GitHub to facilitate collaboration. The github repository link can be seen below.

[github.com/jacktan130802/SC2002-Assignment-](http://github.com/jacktan130802/SC2002-Assignment-)