SC/CE/CZ2002 Object-Oriented Design & Programming Assignment

**APPENDIX B:**

**Declaration of Original Work for SC2002 Assignment**

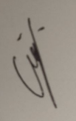
We hereby declare that the attached group assignment has been researched, undertaken, completed, and submitted as a collective effort by the group members listed below.

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.

A close up of a signature

AI-generated content may be incorrect.A black signature on a white background

AI-generated content may be incorrect.

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Course** | **Lab Group** | **Signature / Date** |
| BRYANT KARSTEN LEE | DSAI | FDAE | 24 Apr 2025 |
| DAMIEN TAN YI EN | DSAI | FDAE | 24 Apr 2025 |
| MEGAN SARAH CHUA EN QI | DSAI | FDAE | A signature on a wood surface  AI-generated content may be incorrect.24 Apr 2025 |
| TEE JIA HONG | DSAI | FDAE | 24 Apr 2025 |

Important notes:

1. Name must **EXACTLY MATCH** the one printed on your Matriculation Card.
2. Student Code of Academic Conduct includes the latest guidelines on usage of Generative AI and any other guidelines as released by NTU.

# Design Considerations

## Understanding the Problem and Requirements

We began by reading through the BTO project specification document line by line, highlighting all stated use cases and system requirements. From this, we identified the main problem domain as building a command-line based BTO Management System that caters to multiple user roles—Applicants, HDB Officers, and HDB Managers—each with distinct permissions and workflows. Explicit requirements such as user login using NRIC and password, role-based access, project visibility toggling, enquiry handling, flat application and booking, and officer/manager approval workflows were clearly defined. Additionally, we inferred several implicit expectations, including the need for data validation (e.g., NRIC format, age restrictions), and maintaining user state such as applied status and password changes. Where requirements were ambiguous—such as the scope of visibility effects or flat type selection limits—we discussed them as a team and referred to real-world HDB processes for clarity. These insights allowed us to define the feature scope accurately and establish a solid foundation for our system design.

## 1.2 Features and Scope of project

We grouped features into three categories: core, optional, and excluded. Our goal was to ensure that the core features demonstrated strong object-oriented principles without making the system overly complex or unmanageable within the given timeline. After analyzing the specification, we identified a list of potential features, including user login and authentication, role-based dashboards, BTO project creation and management, application submission, enquiry handling, flat booking, visibility toggling, password change, and withdrawal requests.

Core features were selected based on their essential role in fulfilling the system’s primary use cases. These included user login, role-based navigation, submitting and viewing enquiries, applying for a BTO, flat booking workflows, and manager/officer approval features. Optional features included functions like changing passwords, deleting enquiries, and viewing past applications, as they were useful but not strictly required for demonstrating the core functionality. Features we excluded included email notifications and advanced reporting, as they were beyond the scope of a CLI-based system and would have significantly increased development time without adding proportional learning value.

We prioritized our development by balancing importance and feasibility, focusing first on user flows and data entities that supported the core logic. This structured approach ensured we could complete a meaningful, testable version of the system before considering enhancements.

## 1.3 Analysis of Class Stereotypes

### 1.3.1 Boundary Classes

Boundary classes are the interface between the system and its external environment, handling communication and data transfer between the system and users.

Classes such as user implements the Login interface to login, and will then be directed to their respective user group based on their roles (HDBOfficer, HDBManager and Applicant). This design demonstrates a strong adherence to the principles of modularity and separation of concerns. Each boundary class encapsulates the specific interface and user experience relevant to its user role, ensuring that the logic for handling inputs, displaying outputs, and managing role-specific workflows remains organized and independent. By isolating the UI behavior per user type, our system becomes easier to maintain, extend, and test — changes to the officer interface, for example, do not risk unintended effects on the applicant’s experience. This clear separation also makes the system more scalable, allowing future roles or interface enhancements to be implemented with minimal disruption to the existing structure. Overall, the use of multiple boundary classes reflects a well-structured, user-centric approach to system design.

|  |  |
| --- | --- |
| **Class Name** | **Purpose** |
| Applicant\_UI | Handles applicant interactions by displaying a menu and directing user actions to ApplicantController |
| HDB\_Manager\_UI | Displays a welcome message for the HDB Manager interface. |
| HDB\_Officer\_UI | Facilitates user interaction for officers by displaying options and delegating actions to ApplicantController. |
| Login\_UI | Handles the login process and delegates authentication to Login\_Controller. |

### Control Classes

Control classes are responsible for coordinating and controlling the flow of data between the user interface, entity classes (which represents the system’s data), and any external systems.

The controller classes such as ApplicantController, HDBOfficerController, and Login\_Controller are responsible for managing business logic independently of the UI layer, promoting separation of concerns and aligning with the Single Responsibility Principle. They interact with utility classes like CSVReader and CSVWriter, which encapsulate external file handling responsibilities. This abstraction shields the system’s internal logic from low-level I/O operations, enhancing modularity and testability. By decoupling the UI from the business and data access layers, the design adheres to several SOLID principles—particularly Dependency Inversion and Open/Closed Principles—ensuring that each component remains maintainable, extensible, and easy to test in isolation. Doing it this way ensures that each class have their own responsibility, also fulfilling Single Responsibility Principle.

|  |  |
| --- | --- |
| **Class Name** | **Purpose** |
| ApplicantController | Handles business logic for applicant-related actions like submitting and managing enquiries or BTO applications. |
| HDBOfficerController | Manages operations specific to HDB officers, such as processing applications or managing data access. |
| Login\_Controller | Handles user authentication logic, such as verifying credentials during login. |
| CSVReader & CSVWriter | Manage reading from and writing to CSV files respectively, acting as utility/helper classes. |
| HDBManagerController | Manages operations specific to HDB managers, such as approving registration, application and to create BTO project. |

### Entity Classes

Entity classes are usually persistent and encapsulate the attributes and behaviors of the entities within the system. Within our system, some entity classes are Applicant; BTOApplication; BTOProject; Registration; Enquiry; FlatBooking; HDBManager; HDBOfficer; Receipt; User. These classes set their attributes as private to prevent direct access and provide controlled access through getters and setters.

## Java Package

To ensure package cohesion, we organized the classes into packages based on the type of classes they belong to such as Boundary, Controllers and UI. Having a good package design ensures that each package represents a cohesive set of related classes that work together to implement a specific goal. Classes within the same package should be closely related in terms of functionality and work together as a cohesive unit to achieve a specific goal.

## Assumptions made

* The HDB officer can be officer of multiple projects, as long as there is no overlap in application opening and closing dates. This is so that HDB officer will be able to apply for a new project after his current project’s application closing date.
* All user’s names are distinct.
* All project names are distinct.
* Users already know their given role to log in to the system.

## Design Tradeoffs

* Any officer or manager can reply to enquiry which will overwrite the previous one.
* Using CSV to save does not allow us to save the whole project, only able to save an attribute of the project.
* If a user replies with a ‘,’ it will corrupt the entire CSV file.

## UML Class Diagram

Based on the above class stereotypes that we have identified above; we are able to better identify how different classes are linked together. We identified these classes by different characteristics and verbs in the project outline. After listing down these classes and its potential method and attribute, we are then able to create a UML class diagram on Visual Paradigm. Doing it this way allows us to be able to attain a code skeleton based off our diagram, and we can continuously refine it along the way.

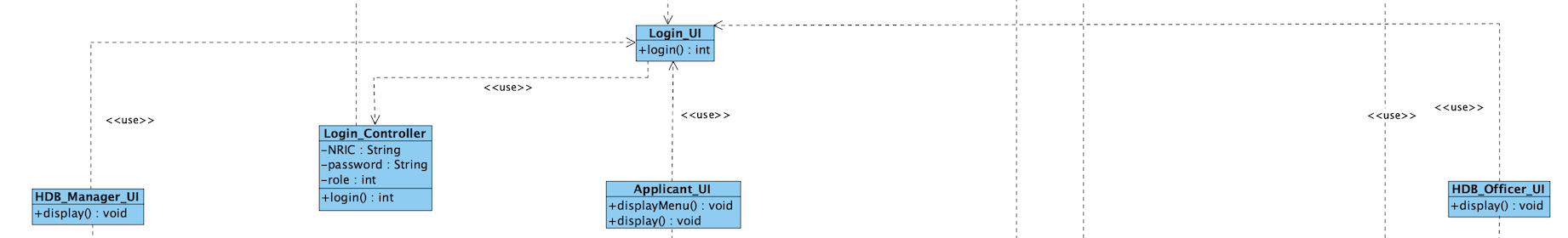
## Sequence Diagram

When creating the sequence diagram, we had to first decide what process in the project that we may want to use. Given the quick deadline of the project and smaller group size, we focus on the two main sequences required from us, namely HDB Officer applying for a project and HDB officer registering to be officer of a project. When creating the diagram, we must account how the different classes interact with one another as well as the correct sequence. We also have to account for the requirements in ensuring that the appropriate requirement are met before the officer can apply or register for project.

# 2.0 Design Principles Implemented

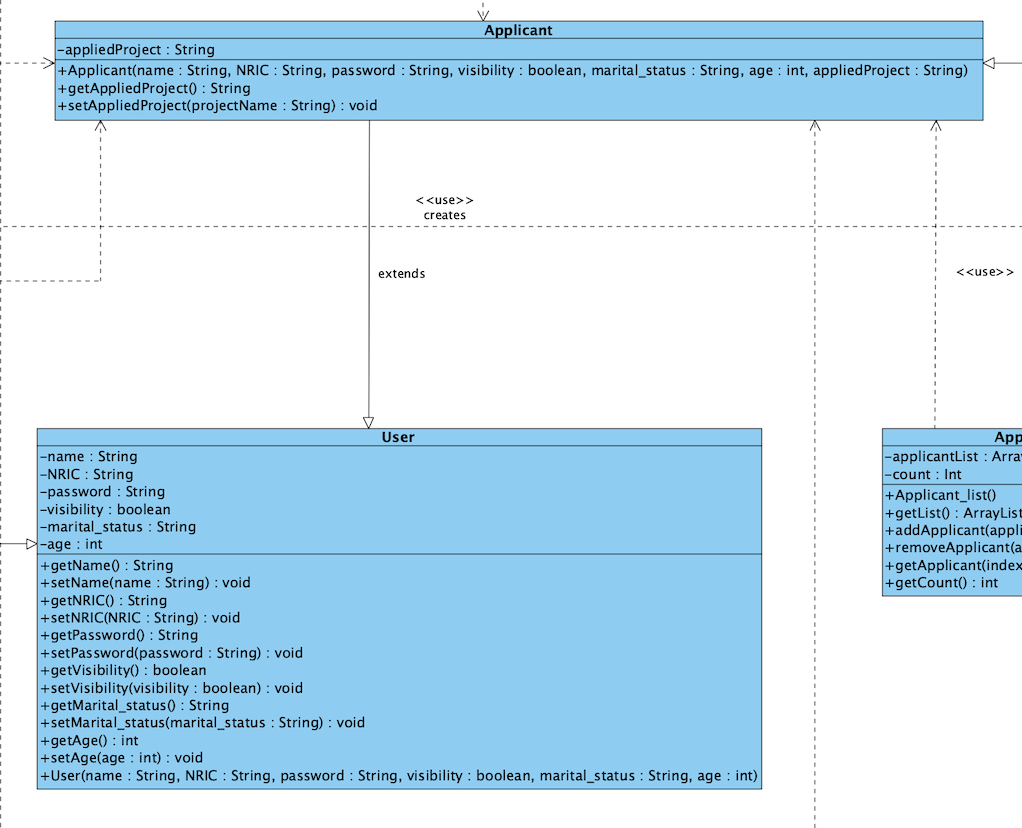
## 2.1 Single Responsibility Principle

The Single Responsibility Principle states that each class should only have one responsibility. This means that a class will only have a singular, well-defined purpose or job and thus should only have one reason to change. By adhering to the SRP, it ensures that class remains cohesive, where changes made will be restricted to its own domain. For example, by splitting the different login UI of each user class, we ensure that each class has a single responsibility and that it fulfills this responsibility to the best of its extent.



## 2.2 Open/Closed Principle

The Open/Closed Principle states that the entities should be open for extension but closed for modification. This means that we should be able to add in new functionality to the class by extending the class without making any modifications to the original class source code.



An instance where OCP is incorporated is through our use of inheritance, where the Applicant class extends the base Users class. This allows us to introduce additional behaviors and properties specific to Applicant without altering the existing User class, thereby keeping it closed for modification but open to extension through subclassing, allowing for future enhancements.

## 2.3 Liskov Substitution Principle

The Liskov Substitution Principle states that superclass objects should be replaceable with subclass objects without affecting the correctness of the program. This ensures that the subclasses are compatible with the base class and can be used interchangeably.

public static void changePassword(){

User currentUser = LocalData.getCurrentUser();

Scanner sc = new Scanner(System.in);

System.out.println("Enter your new password");

String newPassword = sc.nextLine();

currentUser.setPassword(newPassword);

// need to change password here

System.out.println("Password saved successfully");

}

The Applicant class extends User, gaining all its properties and methods (getName(), getNRIC()), while preserving the expected behavior of a User, thus adhering to the Liskov Substitution Principle (LSP). The Applicant class does not override any method in a way. It introduces additional attributes specific to applicants, such as appliedProject, without modifying or restricting the base functionality inherited from User. The changePassword() method in the ApplicantController demonstrates the Liskov Substitution Principle by treating the currently logged-in user as a generic User object. Although the actual object may be an Applicant, it can still be used wherever a User is expected. The call to setPassword() operates uniformly across all user types, confirming that subclasses conform to the expected behavior of their superclass without modification.

The changePassword() method in the ApplicantController demonstrates the Liskov Substitution Principle by treating the currently logged-in user as a generic User object. Although the actual object may be an Applicant, it can still be used wherever a User is expected. The call to setPassword() operates uniformly across all user types, confirming that subclasses conform to the expected behavior of their superclass without modification.

This ensures that an Applicant can be used wherever a User is expected, allowing the system to remain robust and extensible without introducing errors or inconsistent behavior.

## 2.4 Interface Segregation Principle

Interface Segregation Principle states that a class should not be forced to implement interfaces it does not use. To ensure this, we segment our interface into more specific interfaces. By separating into separate specific interfaces, ViewEnquiryInterface, we can prevent ‘FAT’ interfaces. It ensures that the classes are only dependent on the interfaces that they use. For example, the Enquiry class main’s function is to allow user to store and view their enquiry and hence only implement the ViewEnquiryInterface.

A screen shot of a computer

AI-generated content may be incorrect.

## 2.5 Dependency Inversion Principle

Dependency Inversion Principle states that high-level modules should not depend on low-level modules, but both should depend on interfaces or abstract classes. Abstractions should not depend on details; details (concrete implementations) should depend on abstractions.

import Entity.User;

public interface ViewEnquiryInterface {

void viewEnquiry(User user);

}

To achieve dependency inversion, Java interfaces can't have static methods that are intended for instance polymorphism. So we need to make viewEnquiry an instance method, not static. We then updated the controller classes that uses this interface to implement the instance method properly.

import Interface.ViewEnquiryInterface;

public class ApplicantController implements ViewEnquiryInterface {

public void viewEnquiry(User user) {

boolean found = false;

int count = 1;

System.out.println("Your Enquiries:");

We then Inject the controller as the interface (abstraction) and call it from there. Since we are working with static UI methods, we'll instantiate the interface once in the UI class.

import Interface.ViewEnquiryInterface;

public class Applicant\_UI {

ViewEnquiryInterface viewInterface = new ApplicantController();

viewInterface.viewEnquiry(currentUser);

# 3.0 Test cases

We applied black box testing techniques when testing the test cases. We first identified key features in our project, stating the scenario, input we will put and the expected result. Once manually inputted, we will see if the actual output matches the expected result.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test ID | Scenario | Input | Expected  Result | Actual  Result |
| 1 | Valid User Login | 1 | User able to  access their dashboard  based on their roles. | User able to  access their dashboard  based on their roles. |
| 2 | Invalid NRIC Format | S1234A | User receives a  notification about  incorrect NRIC format | User receives a  notification about  incorrect NRIC format |
| 3 | Incorrect Password | hello | System should deny  access and alert the  user to incorrect  password | System should deny  access and alert the  user to incorrect  password |
| 4 | Password Change Functionality | Newpassword | System updates  password, prompt re-  login and allows login  with new credentials | System updates  password, prompt re-  login and allows login  with new credentials |
| 5 | Project Visibility Based  on User Group and  Toggle | Logged in as married applicant James | Projects are visible to  users based on their  age, marital status and  the visibility setting | Projects are visible to  users based on their  age, marital status and  the visibility setting |
| 6 | Project Application | Apply for project as married applicant | Users can only apply  for projects relevant to  their group or when  visibility is off | Users can only apply  for projects relevant to  their group or when  visibility is off |
| 7 | Viewing Application  Status after Visibility  Toggle Off | Visibility turn off for applicant | Applicants continue to  have access to their  application details  regardless of project  visibility. | Applicants continue to  have access to their  application details  regardless of project  visibility. |
| 8 | Single Flat Booking per  Successful Application | Applicant book one flat and tries to book again | System allows  booking one flat and  restricts further  bookings | System allows  booking one flat and  restricts further  bookings |
| 9 | Applicant’s enquiries  management | Applicant submits enquiry and can view it | Enquiries can be  successfully  submitted, displayed,  modified, and  removed. | Enquiries can be  successfully  submitted, displayed,  modified, and  removed. |
| 10 | HDB Officer Registration  Eligibility | Valid HDB Officer register | System allows  registration only under  compliant conditions. | System allows  registration only under  compliant conditions. |
| 11 | HDB Officer Registration  Status | HDB Officer view status | Officers can view  pending or approved  status updates on their  profiles. | Officers can view  pending or approved  status updates on their  profiles. |
| 12 | Restriction on Editing  Project Details | HDB officer tries to edit project detail | Edit functionality is  disabled or absent for  HDB Officers. | Edit functionality is  disabled or absent for  HDB Officers. |
| 13 | Response to Project  Enquiries | HDB Manager can respond to any enquiry. | Officers & Managers  can access and  respond to enquiries  efficiently. | Officers & Managers  can access and  respond to enquiries  efficiently. |
| 14 | Flat Selection and  Booking Management | Registered HDB Officer able to select and book the flat | Officers retrieve the  correct application,  update flat availability  accurately, and  correctly log booking  details in the  applicant’s profile. | Officers retrieve the  correct application,  update flat availability  accurately, and  correctly log booking  details in the  applicant’s profile. |
| 15 | Receipt Generation for  Flat Booking | Registered HDB Officer able to generate receipt | Accurate and complete  receipts are generated  for each successful  booking. | Accurate and complete  receipts are generated  for each successful  booking. |
| 16 | Create, Edit, and Delete  BTO Project Listings | HDB Manager create listing, then is able to delete and edit. | Managers should be  able to add new  projects, modify  existing project  details, and remove  projects from the  system | Managers should be  able to add new  projects, modify  existing project  details, and remove  projects from the  system |
| 17 | Manage HDB Officer  Registrations | HDB Manager approves officer registration. | Managers handle  officer registrations  effectively, with  system updates  reflecting changes  accurately | Managers handle  officer registrations  effectively, with  system updates  reflecting changes  accurately |
| 18 | Approve BTO  Applications | HDB Manager approves a BTO application. | Approvals are  processed correctly,  with system updates to  reflect the decision | Approvals are  processed correctly,  with system updates to  reflect the decision |
| 19 | Generate and Filter  Reports | HDB Manager generates a report | Accurate report  generation with  options to filter by  various categories. | Accurate report  generation with  options to filter by  various categories. |
| 20 | Toggle Project Visibility | Manager changes the visibility. | Project visibility updated. | Project visibility updated. |

# 4.0 Reflection

## 4.1 Difficulties Encountered

### 4.1.1 Getting Started:

At the start of the project, we struggled to decide if we should start with the UML diagram first or go straight into coding. We decided to establish a base idea first and draw up a brief UML as starting guidelines for developing our code. We then proceeded to start coding.

### 4.1.2 Coding as a group:

We initially assigned different tasks and parts of code to each other. However, there were often overlaps where two group mates were working on the same thing. To solve this issue, we decided to use GitHub to collaborate on our project. With GitHub, we were able to implement the necessary changes and resolve any conflicts in our codes smoothly. By using Git commands, we were able to push and pull our code and remain in good sync with each other.

### 4.1.3 Designing the code with SOLID in mind:

Understanding that our code needed to be optimal, we had to think of a way to start coding with these principles in mind. It was crucial that we thought of it early before coding so that later on we would face lesser difficulties in tweaking our code to optimise it. We started off with the Single Responsibility Principle as it was one of the most crucial and straightforward principles to implement. The other principles slowly came along to ideation as we coded further.

### 4.1.4 Consolidating our UML diagram and report:

Fine-tuning our UML diagrams requires good class diagram understanding. We needed to spend time learning how to use the visual paradigm. Including all the details also made it prone to make errors. Hence, it had to be done carefully and by solidifying our understanding.

### 4.1.5 Preparing the Sequence diagram:

Preparing the sequence diagram was deemed challenging as we were unsure of the requirements of the report, as well as if we should do one or two diagrams. We had to re-read the project guideline to get a better sense of what was asked from us.

## 4.2 Improvements to be made

We could have started on the project earlier and experimented with using more interfaces. This could have made the code more loosely coupled and less dependent on each other. By starting earlier, we could also potentially have a better sensing on how many things we need to do in terms of the code as well as documentation, allowing us to continuously refactor the code when necessary.

# 5.0 Attachments

Refer to the diagrams in the folder titled “SC2002 Diagrams”

Figure 1: UML Class Diagram

Figure 2: Sequence Diagram (HDB Officer register to handle a project)

Figure 3: Sequence Diagram (HDB Officer apply for a BTO)

Figure 4: Sequence Diagram (HDB Manager approving application for BTO)

Refer to folder named Javadoc for our Javadoc of the project.

# 6.0 Github Link

<https://github.com/JayHaych/SC2002_BTO-Project>