Project Title: Build-To-Order (BTO) Management System

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 honoured 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	Student ID	Course	Lab Group	Roles	Signature	Date
Kong Fook Wah	U2421655E	DSAI	FDAB	Project Manager/ UML Designer		25/4/2025
Kris Khor Hai Xiang	U2421377C	DSAI	FDAB	Documentation		25/4/2025
Lin Zeshen	U2421421J	CSC	FDAB	Lead Tester/ Developer		25/4/2025
Mau Ze Ming	U2421176G	DSAI	FDAB	Documentation Lead/ UML Designer		25/4/2025
Zheng Nan	U2422815K	CSC	FDAB	Lead Developer		25/4/2025

Chapter 1: Requirement Analysis & Feature Selection

1.1 Understanding the Problem and Requirements

To identify the main problems, our team started with an initial reading to give us a general understanding of the system requirements. We then proceed with a more intentional reading to uncover deeper insights. We actively highlighted and annotated key terms, with a particular focus on the main actors (Applicant, HDB Officer, HDB Manager) and the specific actions that each role can do (e.g. View, Apply, Register). By systematically extracting the entities and their associated responsibilities, we dissected the problem into atomic components and created a high-level functional requirement list. This list served as a reliable reference throughout the project and helped maintain alignment across the team, especially when designing our conceptual diagram.

Explicit Requirement

The provided Venn diagram had outlined clearly the overlaps and differences, suggesting the similarity and differences in each role's capability. Notably, the HDB Officer role extends the capabilities of Applicants, making it a natural candidate for inheritance. This diagram also guided our decision to design modular controller classes like ApplicantController and OfficerController, which aligns with key OO principles like the separation of concerns to ensure that responsibilities are segregated.

Implicit Expectation

While the assignment did not ask for a scalable system, the nature of the module and knowledge acquired thus far does call for our system to be modular and extensible with good OO design.

Ambiguities & Interpretation

Ambiguity	Reason	Interpretation & Resolution
Officers Flat Booking Role	The assignment mentions that *"only HDB Officers can help book a flat"* for successful applicants but does not specify whether the officer must be assigned to that project.	We assumed that only officers assigned to a specific project are allowed to manage its flat bookings. This maintains role accountability and prevents unauthorised access.
Officer's Project Registration Conflict	HDB Officers are disallowed from applying for a project they want to handle — but it's unclear whether this applies after their registration is rejected.	We decided to enforce a stricter policy — once an officer attempts to register for a project, whether approved or not, they forfeit the right to apply for it as an applicant. This simplifies validation and prevents conflict of interest.

1.2 Deciding on Features and Scope

We analysed the assignment document to identify all potential features of user roles. Then, we grouped them into core, optional and excluded categories:

- **Core:** Essential; Required by the assignment
- **Optional:** Not essential, but adds polish if time and complexity permit
- **Excluded:** Too complex, out of scope or not aligned with assignment goals

We prioritise features based on two factors:

- **Importance:** How essential is it to have the feature?
- **Feasibility:** How realistic is it to implement the feature in time?

With the categorisation, it helps us avoid over-engineering and stay focused on the actual requirement. Those with low feasibility and importance were automatically excluded.

Core Features

A. Common Functionality + System-Wide Functionality

Feature Description	Roles Involved	Importance	Feasibility
NRIC Login	Applicant + HDB Officer + HDB Manager	High	High
Change Password	Applicant + HDB Officer + HDB Manager	High	Moderate
CLI-based navigation	-	High	-
File-based program Initalisation	-	High	-
Input Validation	-	High	Moderate

B. User Role Functionality

Feature Description	Roles Involved	Importance	Feasibility
View BTO projects (based on age/marital status + visibility)	Applicant + Officer	High	Moderate
Apply for a flat	Applicant + Officer	High	Moderate
View Application Status	Applicant + Officer	High	High
Request Withdrawal of Application	Applicant + Officer	High	High
Enquiry CRUD operation	Applicant + Officer	High	High
Register to be an officer for a project with conditions	Officer	High	Moderate
View own's registration status	Officer	High	High
View project details (even if visibility is off)	Officer	High	High
Reply to enquiries about the projects they handle	Officer + Manager	High	High
Book flats for successful applicant	Officer	High	Moderate
Generate receipts for flat bookings	Officer	High	High
Project CRUD operation	Manger	High	Moderate
Approve / Reject Registration	Manager	High	Moderate
Approve / Reject BTO application	Manager	High	Moderate
Approve / Reject Withdrawal	Manager	High	Moderate
Generate projects report	Manager	High	High

Timeline

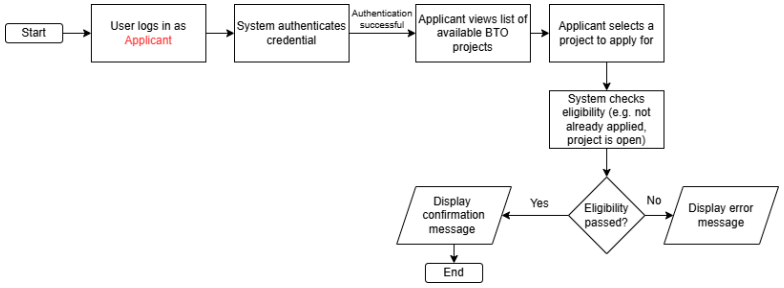
To manage our time well, we adopted the Scrum methodology to break down this system into manageable sprints and prioritised features to be completed first. This ensures that core features were addressed early and that inter-feature dependencies were properly managed.

S/N	Item	Class Involved?	Responsibility	Start Date	Expected Date	Progress	Remarks
	Meeting on 19/4/2025						
1	Able to view the list of projects (for open projects)	Applicant	Kris	28/3/2025	6/4/2025	100%	Done by Kong, yet to test it out
2	Able to view the list of projects (for applied projects)	Applicant	Kris	28/3/2025	6/4/2025	100%	Done by Kong, yet to test it out
3	View all projects	HDB Manager	Zheng Nan	28/3/2025	6/4/2025	100%	
4	View all enquiries	HDB Manager	Kong	28/3/2025	6/4/2025	100%	
5	Allowed to request withdrwal before/after flat booking	Applicant	Kris	28/3/2025	6/4/2025	100%	
6	Able to submit enquiries	Applicant	Zeshen	28/3/2025	6/4/2025	100%	
7	Able to view all projects that he is handling	HDB Officer	Ze Ming	28/3/2025	6/4/2025	100%	yet to test
	Meeting on 06/4/2025						
8	Register to join a project if criterias are met	HDB Officer	Zeshen	6/4/2025	9/4/2025	100%	

Chapter 2: System Architecture & Structural Planning

2.1 Planning the System Structure

Before jumping into implementation, we focused on laying out a clear architecture to guide our development. We first decomposed requirements into local components (e.g. classes) and analysed their core functions. For each core function (e.g. project application), we identified the relevant actors and their interactions within the system. Using tools like *draw.io*, we also sketched out flowcharts for those core functions. These flowcharts showed interactions between the user and system, helping us surface the expected behaviour and uncover early edge cases. Through this, we also derived several use cases, which laid the foundation for class diagram.



We translated these flows into a high-level architectural model, which included entity *(model)*, boundary *(view)*, and controller *(controller)* classes. This initial model gave our team a shared understanding of the interaction of components and allocation of roles.

2.2 Reflection on Design Trade-offs

In the process of planning our system, we made several conscious design decisions and trade-offs to balance development speed, code clarity and long-term extensibility.

One of the key debates our teams had was whether to combine or separate role-specific functionalities into a unified controller. Combining them into a single controller would reduce

the number of classes which speeds up development but it risks creating a bloated, hard-to-maintain logic block. Moreover, it introduces duplicated logic and tight coupling, especially when handling different user roles or adding more features.

Nevertheless, our team wants to keep the system modular and loosely coupled. Hence, despite the tight timeline, we ultimately chose to separate controllers for each role (e.g. ApplicantController, OfficerController) to ensure clarity and better alignment with design principles like the Single Responsibility Principle.

Chapter 3: Object-Oriented Design

3.1 Class Diagram

To translate the system layout into an object-oriented model, our team focused on breaking down the requirements into logical entities, responsibilities and relationships.

Identifying Main Classes

We identified main classes by analysing the commonly stated nouns in the problem description, such as Applicant, HDBOfficer, HDBManager, Project, Application and Enquiry. Then, depending on the behaviour of classes, we came up with methods that we think are needed for implementation. These were also cross-referenced against our functional requirements list and gradually mapped into Entity, Boundary and Control classes in our implementation.

Key Classes Responsibilities

We made a table listing the key responsibilities of each class to refer to easily (see Appendix).

Determining the Relationships

We used OODP practices to better determine the relationship between classes.

1. **Inheritance vs Association:** We used inheritance only when a subclass needed to extend shared behaviour and data (e.g. initial User subclasses like Applicant and HDBManager) and association when one class simply held or interacted with another (e.g. Project has a list of Enquiries and Applicants).
2. **Attributes vs Classes:** For attributes, we modelled them as simple fields (e.g. age, maritalStatus) when they were purely descriptive, but used separate classes when the data requires structure or behaviour of its own (e.g. Housing, BookingManager)

These relationship decisions helped shape a modular and maintainable system, ensuring that each component had a clear responsibility and interacted cleanly with others.

Trade-Offs

Discussions grew heated when debating how to balance the relationships between components. Ultimately, we had to make some key trade-offs:

- We accepted slightly more complexity in setup (e.g. introducing the UserFactory and individual UserCreator classes per role) for long-term maintainability and extensibility.
- We prioritised clarity over premature optimisation so that we can understand each other's code quickly. By modelling BookingManager and Housing as dedicated classes rather than cramming their data into simpler attributes, we kept the classes to only focus on their own responsibility.

This gave us a modular, role-aware system that is open to change yet robust against regressions.

3.2 Sequence Diagrams

One of the more complex use cases in our system is an Officer applying for a BTO project as it involves multiple objects and conditional logic and requires status updates based on approval.

Officer Application for Project They are Not Handling

This flow captures how an HDB Officer requests to apply for a project that they are not handling. We chose this use case because

- It spans multiple layers of our system architecture: UI → Controller → Entity (OfficerInCharge in Project)
- It contains conditional logic (approve/reject, checking concurrent projects)
- It demonstrates role-based logic (only managers can approve)

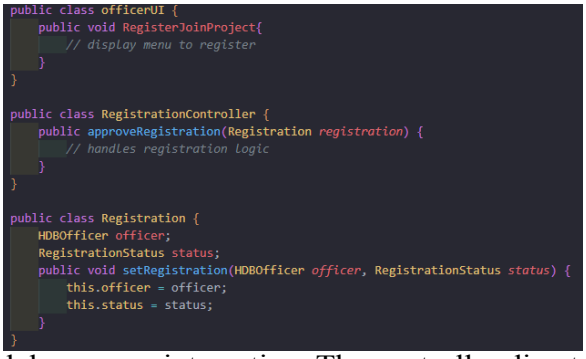
Reasons	**Pattern / System Design Illustrated**
Spans multiple layers	Show how roles interact with different layers of the system *View → Controller → how controller handles logic → Entity*
Contains conditional logic	Show how objects communicate with one another through class relationships and controller logic
Role-based logic	Validate our design decisions around single responsibility, role segregation and data integrity

Using this scenario, we were able to visualise and validate the intended behaviour of our system architecture. Ultimately, it served as a bridge between high-level design and practical implementation, ensuring we built a system that was both functional and structurally sound.

3.3 Application of OOD Principles (SOLID)

Single Responsibility Principle (SRP):

To prevent tight coupling and promote maintainability, we applied SRP to the registration approval flow, which involves multiple system components. By clearly defining each class's responsibility, we avoided scenarios where objects handle logic outside their scope.



```
public class officerUI {
    public void RegisterJoinProject{
        // display menu to register
    }
}

public class RegistrationController {
    public approveRegistration(Registration registration) {
        // handles registration logic
    }
}

public class Registration {
    HDBOfficer officer;
    RegistrationStatus status;
    public void setRegistration(HDBOfficer officer, RegistrationStatus status) {
        this.officer = officer;
        this.status = status;
    }
}
```

The UI layer is focused solely on user interaction. The controller directed the flow without enforcing business rules, and the entity layer handles persistence only.

Open/Closed Principle (OCP)

Instead of modifying core classes like Applicant, we introduced a UserCreator interface with dedicated implementations such as ApplicantCreator, HDBOfficerCreator and HDBManagerCreator. Each creator encapsulates the construction logic specific to its user role, promoting cleaner separation of concerns.

```
public interface UserCreator {
    User createUser(String userID, String password, String name, int age, MaritalStatus maritalStatus);
}

public class ApplicantCreator implements UserCreator {
    @Override
    public Applicant createUser(String userID, String password, String name, int age, MaritalStatus maritalStatus) {
        return new Applicant(userID, password, name, age, maritalStatus);
    }
}

public class UserFactory {
    private final Map<UserRole, UserCreator> creators = new HashMap<>();

    public UserFactory() {
        // Register default creators
        creators.put(UserRole.APPLICANT, new ApplicantCreator());
        creators.put(UserRole.HDB_OFFICER, new HDBOfficerCreator());
        creators.put(UserRole.HDB_MANAGER, new HDBManagerCreator());
    }
}
```

While using the factory pattern complicates our system, this design allows the system to be extended, such as adding new roles like External Audit, by simply introducing a new creator class and registering it, without altering any existing logic. This ensures our codebase remains open for extension but closed for modification.

Liskov Substitution Principle (LSP)

Since HDBOfficer is a subclass of Applicant, we applied LSP here so that HDBOfficer can seamlessly replace its parent without breaking functionality. For example, BookingManager accepts Applicant instances, but HDBOfficer can still pass in without errors or behavioural changes, preserving the superclass contract. One special note to take is that we modelled the BookingCapable interface and embedded it in the inheritance chain. This avoids tight coupling and also keeps responsibilities isolated.

```
public interface BookingCapable {
    void bookFlat(Project project, String housingType)
}

public class Applicant extends User implements BookingCapable {
    public void bookFlat(String housingType) {
        // booking Logic
    }
}

public class HDBOfficer extends Applicant {
    // inherits booking capability
}

// HDBOfficer can replace Applicant
public class BookingManager {
    public void processBooking(Applicant applicant, String housingType) {
        applicant.bookFlat(housingType)
    }
}
```

```
// HDB officer can stand in for Applicant with no error
public class ApplicantUI {
    protected flatBooking {
        BookingManager bookingManager = new BookingManager();
        bookingManager.processBooking(applicant);
        bookingManager.processBooking(officer);
    }
}
```

Interface Segregation Principle (ISP)

Considering that there may be other user types in the future who may want to use the system, such as system users, we followed ISP by designing focused and role-specific interfaces such as Authenticatable and PersonalProfile. This prevents users from being forced to implement methods that they do not need, promoting cleaner code that aligns with ISP.

```
public interface Authenticatable {
    String getUserID();
    String getPassword();
    void changePassword(String password);
}

public interface PersonalProfile {
    String getName();
    int getAge();
    MaritalStatus getMaritalStatus();
}
```

```
public abstract class User implements Authenticatable, PersonalProfile{
    private String userID;
    private String name;
    public String getUserID() {return this.userID; }
    public String getName() { return this.name; }
}

// In the future, if we have a class for system user e.g. data admin
public abstract class DataAdmin implements Authenticatable {
    //Login logic
}
```

All user types could implement Authenticatable for login functionality, but only relevant roles need to implement PersonalProfile, as system users like administrators do not need to have personal information in the system. This approach cleanly complies with ISP.

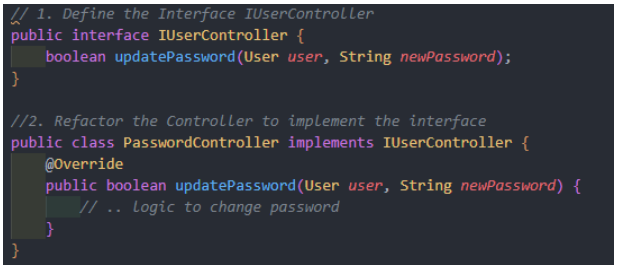
Dependency Inversion Principle (DIP)

Originally, our system suffered from tight coupling, where classes directly instantiated other components, such as PasswordController. For instance, Applicantui directly instantiated ChangePasswordUI, which internally depended on the concrete Password Controller.

```
public class ChangePasswordUI extends UI {
    protected void displayChangePasswordMenu() {
        // .. display logic
        successful = PasswordController.updatePassword(currentUser, password);
    }
}

public class PasswordController {
    public static boolean updatePassword(User user, String newPassword) {
        //..Logic to change password
    }
}

public class ApplicantUI extends UI {
    private final ChangePasswordUI changePasswordUI = new ChangePasswordUI();

    protected displayMenu() {
        // logic to display menu
        changePasswordUI.displayChangePasswordMenu();
    }
}
```

This structure violates the Dependency Inversion Principle (DIP), as ChangePasswordUI is tightly coupled to the concrete implementation of PasswordController. This makes testing, swapping implementations and scaling more difficult. What if one day we want to have an AdminPasswordController instead of PasswordController? We have to change the source code.

To improve our system, we introduce an interface IUserController and inject it into ChangePasswordUI. This decouples the UI from the concrete controller implementation. ChangePasswordUI now depends on the IUserController interface rather than a concrete implementation, allowing for different password controller implementations to be injected. ChangePasswordUI no longer knows about PasswordController directly, but knows about the IUserController interface. In future, we can flexibly swap our implementation without changing ChangePasswordUI.



```
// 1. Define the Interface IUserController
public interface IUserController {
    boolean updatePassword(User user, String newPassword);
}

//2. Refactor the Controller to implement the interface
public class PasswordController implements IUserController {
    @Override
    public boolean updatePassword(User user, String newPassword) {
        // .. Logic to change password
    }
}
```

```
//3. Inject the Dependency into ChangePasswordUI
public class ChangePasswordUI extends UI {
    private final IUserController userController;

    public ChangePasswordUI(IUserController userController) {
        this.userController = userController;
    }

    protected void displayChangePasswordMenu(){
        // .. display logic
        successful = userController.updatePassword(currentUser, password);
    }
}

//4. Contruct with Injection
public class ApplicantUI extends UI {
    IUserController controller = new PasswordController();
    private final ChangePasswordUI changePasswordUI = new ChangePasswordUI(controller);
}
```

Chapter 4: Implementation (Java)

4.1 Tools Used

- Java 17
- IDE: Visual Studio Code
- Version Control: GitHub

4.2 Sample Code Snippets

Encapsulation

```java
public class Enquiry {
    private Applicant applicant;
    private String message;
    private String reply = "-";
    private Project project;
    private boolean replied = false;

    public Enquiry(Applicant applicant, Project project, String message) {
        this.applicant = applicant;
        this.message = message;
        this.project = project;
    }
    public String getMessage() {return this.message; }
    public String getReply() {return this.reply; }
    public void setReply(String reply) { this.reply = reply; replied = True; }
}

public class EnquiryController {
    public static void getEnquiriesbyApplicant() {
        //Logic to get enquiries by an applicant
        if(...) { return enquiry.getMessage();}
    }
```

Polymorphism

```java
public abstract class UserUI implements UserInterface{
    protected void viewOpenProjects() {
        ...
    }
}

public class OfficerUI extends UserUI{

    protected void viewOpenProjects() {
        System.out.println(x:"This functionality is only for viewing Projects open for Appli
        System.out.println(x:"Please use Option 1 to view projects you're handling.");
    }
}
```

```java
public class ApplicantUI extends UserUI{

    protected void viewOpenProjects() {
        System.out.println(x:"List of Open Projects: ");
        List<Project> projectList = getApplyProjectList();

        if (projectList.isEmpty()) { System.out.println(x:"No visible project to view based on your
        displayProjectList(projectList);

        try {
            int projIndex = getIntInput("Select the project to view details for: ") - 1;
            if (projIndex >= 0 && projIndex < projectList.size()) {
                Project project = projectList.get(projIndex);
                ProjectController.displayProjectDetails(project);

            } else { System.out.println(x:"Invalid project selection."); }

        } catch (Exception e) { System.out.println("Error viewing project: " + e.getMessage());}
    }
}
```

Inheritance

```java
public class ManagerUI extends UserUI {
    // code showing how to get user's input and show menu
}


public abstract class UserUI implements UserInterface{
    protected User currentUser;
    protected static final Scanner scanner = new Scanner(System.in);
    protected static final DateTimeFormatter DATE_FORMATTER = DateTimeFormatter.ofPattern("yyyy-MM-d

    protected int getValidIntInput(int min, int max) { return UIUtils.getValidIntInput(min, max); }
    protected String getStringInput(String prompt) { return UIUtils.getStringInput(prompt); }
    protected float getFloatInput(String prompt) { return UIUtils.getFloatInput(prompt); }
    protected LocalDate getDateInput(String prompt) { return UIUtils.getDateInput(prompt); }
    protected abstract int getMaxMenuOption();
    public void showMenu() {...}
}
```

Interface Use

```java
public interface Authenticatable {
    String getUserID();
    String getPassword();
    void changePassword(String password);
}

public interface PersonalProfile {

    String getName();
    int getAge();
    MaritalStatus getMaritalStatus();
}
```

```java
public abstract class User implements Authenticatable, PersonalProfile{
    private String userID;
    private String password;
    private String name;
    private int age;
    private MaritalStatus maritalStatus;
    private UserRole userRole;

    @Override
    public String getUserID() { return this.userID; }

    @Override
    public String getPassword() { return this.password;}

    @Override
    public String getName() { return this.name; }

    @Override
    public MaritalStatus getMaritalStatus() { return this.maritalStatus; }

    @Override
    public int getAge() { return this.age; }
}
```

Error Handling

```java
public class OfficerUI extends UserUI{
    protected void viewProjects() {
        try {
            int projIndex = getIntInput("Select the project to view details for: ") - 1;
            Project project = projectList.get(projIndex);

            if (project != null) {
                ProjectController.displayProjectDetails(project);
            }
        } catch (Exception e) {
            System.out.println("Error viewing project: " + e.getMessage());
        }
    }
}
```

Chapter 5: Testing

5.1 Test Strategy

We adopted a two-pronged approach, Unit Testing and Manual Functional Testing, to ensure that our system met both functionality and usability requirements.

1. **Unit Testing:** We developed unit tests to verify the correctness of each method in the system. This ensured that each method created performs exactly as we designed.
2. **Manual Functional Testing:** We manually tested user flows and operations using the various user profiles to simulate real-world usage scenarios.

Tools Used: We used JUnit for automated unit testing of Java classes and methods. This helped us quickly identify logical errors and validate the functionalities of each method.

5.2 Test Case Table

No	Test Case Description	Input / Action Taken	Expected Output	Actual Output
1	Valid login credentials	NRIC: S1234567A Password: password	Login success, user sees User Menu	Login success, user sees User Menu
2	Invalid NRIC Format	NRIC: 1234567A	Error Message: "Invalid userID format. Try again."	Error message displayed
3	Incorrect password	NRIC: S1234567A Password: wrongpass	Error Message: "Invalid NRIC or password. Please try again."	Error message displayed
4a	Change password Old: password New: P@ssword!1234	Password: P@ssword!1234	Login success, user sees User Menu	Login success, user sees User Menu
4b		Password: password	Error Message: "Invalid NRIC or password. Please try again."	Error message displayed
5	Project visibility (Single user)	View visible projects (Single, Age: 35)	Sees only 2-Room **visible** projects	Sees only 2-Room **visible** projects
6	Invalid project application (Single below Age 35)	Tries to apply for project (Single, Age: 25)	Error Message: "Sorry. Currently no projects available for you to apply for."	Error message displayed
7	View application after visibility is off	Applicant view application after toggling visibility to Off	Application for project remained visible to applicant	Application for project remained visible to applicant
8	Multiple flat bookings	Applicant tries to book second flat	Error Message: "Flat already booked."	Error message displayed
9a	Applicant's enquiry management	Submit, view, edit, delete enquiry	Able to submit, view, edit, delete enquiry without causing error	Able to submit, view, edit, delete enquiry without causing error
9b		Edit enquiry after being replied	Error Message: "Cannot edit a replied enquiry."	Error message displayed
10	Officer registration and application conflict	Officer apply and register for same project	Project not present in list of project that can register / apply for	Project not present in list of project that can register / apply for
11	Officer registration status access	Officer view registration status	Registration status visible regardless of project visibility	Registration status visible regardless of project visibility
12	Officer project access	Officer access project	Can access assigned project regardless of project visibility	Can access assigned project regardless of project visibility
13	Unauthorised project editing	Officer tries to edit project	Edit functionality is absent for officer	Edit functionality is absent for officer

No	Test Case Description	Input / Action Taken	Expected Output	Actual Output
14	Officer response to enquiry	Officer respond to the selected enquiry of the project that he is incharge of	Enquiry's reply is submitted successfully	Enquiry's reply is submitted successfully
15	Flat booking updates	Officer helps Applicant books a flat	Flat count updates, Applicant marked "Booked"	Flat count updates, Applicant marked "Booked"
16	Receipt generation	Officer generate receipt of applicant that successfully booked a flat	Generate receipt with all correct details	Generate receipt with all correct details
17	Manager creating project	Valid project data	Project appears in csv file	Project appears in csv file
18	Manager edits project	Edit project with valid data	Project updates reflect in csv file	Project updates reflect in csv file
19	Manager handles >1 project in same period	Overlapping date with other projects handled	Error Message: "Manager can only handle one Project within each application period."	Error message displayed
20a	Manager toggle visibility	Toggle from On to Off	Project does not appear when Applicant tries to view	Project does not appear when Applicant tries to view
20b		Toggle from Off to On	Project appear when Applicant tries to view	Project appear when Applicant tries to view
21	Manager view filtered project list	Manager select filter (self-created projects)	Only Self-Created project appeared	Only Self-Created project appeared
22	Manager approves officer registration	Manager approve the registration to project	Officer successfully allocated	Officer successfully allocated
23a	Application withdraw management	Manager accept withdraw application	Application successfully withdrawn	Application successfully withdrawn
23b		Manager reject withdraw application	Application status successfully reverted	Application status successfully reverted
24	Report generation filter	Manager generate report with filter "Marital Status - Married"	Only application from married applicant shown	Only application from married applicant shown

Chapter 6: Reflection & Challenges

Our team collaborated effectively from the start, with clear task assignments and weekly meetings to check in on each other's progress. That kept our workflow efficient. Moreover, we maintained strong coding practices throughout such as naming conventions and writing clean and readable code (no spaghetti code). However, despite the much thoughtful planning, there were some inevitable hiccups that led to some backtracking and rework of the system. From the experience, we realised the usefulness of decomposing problems and drawing more diagrams to help with visualisation, as well as how applying SOLID principles resulted in a clearer and more maintainable code.

Appendix:

Github link: https://tinyurl.com/SC2002-FDAB-Group4

Table displaying the list of key responsibilities for each class:

Classes	Key Responsibilities
Applicant *(extends User)*	- View eligible projects, i.e. `viewOpenProject()` - Applies / Withdraw booking, i.e. `withdrawBooking()` - View application status, i.e. `viewAppliedProject()` - Submit / Edit / Delete enquiries, i.e. `editEnquiry()`
HDBOfficer *(extends Applicant)*	- Inherits Applicant methods - Register for project, i.e. `registerJoinProject()` - Book flat for applicants i.e. `flatBooking()` - Update flat unit count - Respond to enquiries, i.e. `viewAndReplyEnquiries()` - Generate booking receipt, i.e. `generateReceipts()`
HDBManager *(extends User)*	- Create / Edit / Delete projects i.e. `createProject()` - Approve / Reject officer registration i.e. `approveRejectOfficer()` - Approve / Reject Application / Withdrawal i.e. `approveRejectWithdrawal()` - Generate filterable reports i.e. `generateReports()`
Project	- Store project info, i.e. `Project` Entity Class - Add / Remove application, i.e. `setApplicant()` - Add assigned officer to project, i.e. `setOfficersInCharge()` - Update unit availability
Application	- Store applicant-project link and status, i.e. `bookingDetails: Map<Project,String>` - Update and retrieve status summary, i.e. `getBookingDetails()`
Enquiry	- Store message and response i.e. `Enquiry` Entity Class - Allow editing and responding to enquiry, i.e. `replyToEnquiries()` - Provide formatted enquiry details, i.e. `viewEnquiry()`