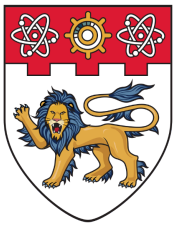


NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

SC2002: OBJECT-ORIENTED DESIGN AND PROGRAMMING

AY 24/25 SEMESTER 2 GROUP PROJECT

Build-To-Order (BTO) Management System

DATE OF SUBMISSION: 24/4/25

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	Role
Low Zhen Jie, Alden	SC2002	FCSI - Group 4	Alden 24/4/25	Project Manager Report Lead Developer
Wang Cheng	SC2002	FCSI - Group 4	Wang Cheng 24/4/25	UML Designer (Class Diagram) Lead Developer Tester
Han Sheng Jie, Philip	SC2002	FCSI - Group 4	Philip 24/4/25	UML Designer (Sequence Diagram) Developer Testing Lead
Sean Winston Susanto	SC2002	FCSI - Group 4	Sean 24/4/25	UML Designer (Sequence Diagram) Developer

1. Requirement Analysis & Feature Selection

1.1 Understanding the Problem and Requirements

We began by analysing the assignment document, highlighting the system requirements and use cases for each of the different user types. Based on the requirements given, we noted down the different methods needed for each user, for example, requestWithdrawal() for applicants and assistBookingFlat() for officers. Hence, our designed system's main aim was to minimise coupling by promoting clear separation of responsibilities among the layers. However, right off the bat, we found that there were some missing requirements regarding the storage of data and how to ensure consistency of user decisions across logins.

1.2 Deciding on Features and Scope

We grouped features by their user type, and within each user type, we categorised them into core and bonus features. All users have login and change password features, along with being able to view their own profiles. The core features are below:

- Applicants' features will be accessed through the ApplicantMainMenu with the following features:
    - View Booking: Monitor booking status
    - View Available Projects: View projects with filtering capabilities
    - Applicants can manage their project application through ProjectAppMenu:
        - Apply for Project: Apply for a project with filters
        - Withdraw Application
        - View Project Details: Able to view project details like flat type and application status
        - View Booking Receipt: View the full details for their booked flat
    - Applicants can manage their enquiries through the applicantEnquiryMenu:
        - Submit, Edit and Delete Enquiries
- Officers' features will be accessed through the OfficerMainMenu with the following features:
    - Officers have all of the Applicant capabilities in their main menu
    - Register to Handle a Project: Officers can register to handle a project (only 1 project at any point)
    - View Registration Status: View the status of their project registration
    - View the Project they are handling
    - Assist in Flat Booking: Help applicants book a flat once the manager approves the application
    - Generate Booking Receipt: View the booking receipts by applicant NRIC
    - Officers can manage their project's enquiries through the officerEnquiryMenu:
        - View and respond to enquiries about the project they are in charge of
- Manager's features will be accessed through the ManagerMainMenu with the following features:
    - Process Join Requests: Handle the applications to join the project as officers
    - Process Project Application: Handle the applications for flats from applicants

- Process Withdrawal Requests: Handle applicants' withdrawals regardless of applicant status
- Generate Report: Able to see the receipts of all bookings with a filter for only their project
- View Project Listing: Able to view all project listings with a filter for only their project
- Create, Edit and Delete Project Listings
- Managers can manage their project's enquiries through the managerEnquiryMenu:
  - View and respond to enquiries about the project they are in charge of

Additionally, we included bonus features like:

- Option to filter the manager's viewed enquiries based on whether they are from his/her managing project
- Additional project filters for viewing, like flat type, neighbourhood and room type.

2. System Architecture & Structural Planning

2.1 Planning the System Structure

The BTO Management System implements a public housing application and allocation platform for three types of users: **Applicants**, **HDB Officers**, and **HDB Managers**. We designed the system using a modular, object-oriented approach, and it is divided into high-level and low-level packages and the service layer.

High-level packages include the **menu (UI) classes**, which serve as the primary point of interaction between the user and the system. These classes manage user navigation menus, handle input via scanners, and delegate logic to the corresponding service layers. Each user type has a dedicated main menu interface that streamlines access to their specific functionalities.

Low-level packages are composed of the **entity and repository classes**, which mirror real-world objects such as users, projects, and booking receipts. Entity classes serve as the data model, representing objects such as Applicant, Officer, Project, Receipt, etc. Repositories store collections of these objects and provide access methods to retrieve or modify them in memory.

The **service layer** acts as the middle layer, encapsulating business logic such as handling bookings, managing join requests, generating receipts, and validating user roles and permissions. These services ensure proper workflow execution and error handling while keeping the UI and data model layers decoupled.

2.2 Reflection on Design Trade-Offs

During the planning stages, we met 3 main challenges. Below are our solutions and lessons learnt from each:

1. Managing state across services without a database.
   a. Used Repository to simulate persistent data stores.
   b. Importance of **object-oriented design** for scalability and maintainability.
2. Ensuring menu navigation flows smoothly using Scanner inputs.
   a. Used layered architecture to isolate concerns.
   b. Benefits of separating UI and business logic.
3. Designing for different user roles without duplicating logic.

a. Designed helper methods to simplify code.
b. How to manage complex user interactions in a CLI-based system.

We also considered some trade-offs during the planning stage:

- Unless explicitly handled through serialisation, all data is lost on shutdown. Using an actual DB would greatly enhance functionality.
- Role promotion is not supported: For example, a user can't switch from applicant to officer or view combined dashboards — roles are rigid.
- Due to implementation limitations, we are limited to a Command-Line Interface rather than a more user-friendly Graphical User Interface.

3. Object-Oriented Design

3.1 UML Class Diagram

Identification of Entity Classes and Class Relationships

The class diagram was the first thing we created in this project, so that we could have a general model of the system, to which we could add more details later. We started by reading through the required functionalities and identified a list of frequently occurring nouns, which might become classes later. Applicant, Officer and Manager were the obvious classes that we needed to have. Looking more closely, we realised they shared some common attributes, such as name and age. Therefore, we created an abstract class called User, and Applicant and Manager classes extended User. Since an officer possessed all the capabilities of an applicant, we let the Officer class extend the Applicant class. It is worth mentioning that we initially identified a report class, which was an aggregation of project applications. However, later we concluded that the only purpose of a report was for a manager to view flat bookings, and this could be achieved by printing the relevant information to the command line. Therefore, there was no need to let it be a stand-alone class, so we removed the report class.

Identification of Class Responsibilities

After we have identified the list of classes, we proceeded to decide what the classes should be responsible for, i.e. their methods. Initially, we did this by identifying any verbs in the required functionalities. For example, since an applicant was able to apply for a project, we added applyProject() to the Applicant class. However, we realised at a later stage that it made the Applicant, Officer and Manager classes too big and too complex. For instance, the Manager class needed to handle roughly 10 functionalities. Therefore, we decided not to let Applicant, Officer and Manager classes have any methods. They became pure entity classes to only store information. All functionalities were delegated to service classes, and each service class took charge of one functionality. For example, ProjectListingService enabled managers to create new project listings, and ProjectApplicationService enabled applicants to apply for a project.

Creation of Repository Classes and Menu Classes

Since we had entity classes, we needed to find a way to organize and store objects. For this purpose, we created one repository class for each entity class. For example, all Applicant objects were stored in ApplicantRepository. ApplicantRepository was responsible for saving objects into a local file. It also enabled searching for a particular applicant by their NRIC.

There should also be some higher level classes that read inputs from whoever is using the system, and depending on user inputs, these classes would call the functions from service classes and repository classes. For this purpose, we created the Menu package, which included ApplicantMainMenu, OfficerMainMenu, ManagerMainMenu and submenus such as the EnquiryMenu. Now we had four packages, namely the menu, service, repository and entity. They formed a layered architecture that provided a good organization of classes.

Trade-offs in The Class Diagram

All individual repositories could have been combined into a centralized repository, which would have saved some storage space and made repository management easier. However, we decided not to combine them, in favour of separate repositories, because each repository had their own specific methods that were only applicable to that particular type of entity class. For example, the ApplicantRepository had a method to search for applicants by their NRIC, but the ProjectRepository obviously did not need this method, because NRIC is not an attribute of a project. Separating repositories allowed each repository to serve its own purpose.

3.2 UML Sequence Diagram

Sequence diagrams show how objects collaborate over time to execute real use cases and verify our layered design. We selected scenarios that combine multiple components, data persistence, conditional rules, and retry loops to exercise core features—authentication, filtering, and role-based access control—without mixing UI and data logic.

Before selecting any sequence diagram, we applied three criteria. First, we focused on scenarios featuring multi-step logic (such as eligibility checks and retry loops), interactions across multiple services and repositories, and clear branching for both success and failure. Second, we ensured each scenario exercised our three architectural layers—menu classes, service classes and repository/domain classes—while preventing any direct persistence calls from the user interface. Third, we chose flows that highlight key patterns (authentication, search and filter logic, capacity enforcement and full CRUD lifecycles). These criteria guarantee that our design meets real-world requirements and remains easy to maintain.

BTO Application as an Officer

We chose this sequence to illustrate both the application submission flow and our system's design choice of allowing officers to inherit the capabilities of applicants. In our class structure, the Officer class extends Applicant, which in turn extends User. This decision allows an Officer to perform any function available to an Applicant, including submitting a BTO application. Applying for an application is one of the core features of the system, we are able to showcase the full flow of how applications are handled—starting from the user interface, flowing through the service layer, and finally being processed and saved via the relevant repository.

Additionally, the sequence captures eligibility checks that occur when an officer wants to submit an application. These checks ensure that the officer meets the required criteria for application submission, such as age, marital status, and other requirements.

Officer Join-Request

We selected this sequence to show how an officer submits a join request to handle a project, including rejection scenarios like having an active project or exceeding capacity limits. It also highlights the approval process, where the request must pass through a manager for final approval. This demonstrates how the system enforces role-based constraints and ensures requests are validated in the service layer before being saved into the repository. By capturing this flow, we showcase how the service and repository layers handle business logic and maintain data consistency, while the UI focuses on user interaction.

Enquiry Process

We chose this sequence to demonstrate the full CRUD lifecycle for project enquiries, showing how both applicants and managers interact with the system via a shared enquiry repository. Applicants can submit, view, edit, and delete enquiries, while managers can view all enquiries regardless of project boundaries and respond to those regarding their active handling project. The diagram also includes enquiry filtering by project, ensuring both roles access and manage relevant information based on their permissions.

While officers can also reply to enquiries, we didn't include them in this sequence because their role is more restricted—they can only respond to enquiries related to the projects they handle. Managers, on the other hand, encompass a larger flow since they can manage all enquiries, not limited by project constraints. By integrating both applicant and manager roles, we show how the system efficiently manages shared resources like the enquiry repository, maintaining clear role-based boundaries and ensuring data consistency.

3.3.1 SOLID Design Principles

Our system design follows the SOLID principles to ensure that the application is modular, extensible, and easy to maintain:

S – Single Responsibility Principle (SRP)

Each class in the system has one and only one reason to change. The Officer, Applicant, and Manager entity classes are solely responsible for storing user-specific attributes. Service classes like JoinRequestService, BookingService, and ViewProjectService handle logic related to their own domain. Interface classes (OfficerMainMenu, ApplicantMainMenu, etc.) are responsible only for user interaction, not logic or storage.

O – Open/Closed Principle (OCP)

The system is open for extension but closed for modification. New user types or functionalities (e.g., an Auditor role) can be added with minimal changes to existing code. For instance, if a new project status needs to be introduced, it can be added via enums or handled within services without modifying existing logic-heavy classes.

L – Liskov Substitution Principle (LSP)

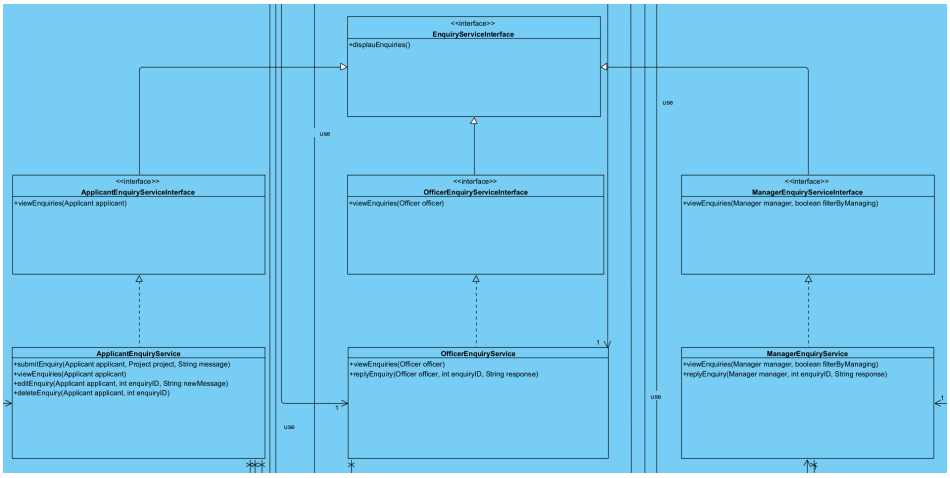
Subtypes can replace their base types without breaking the system. We enforce this through the use of a User superclass (or common interface), which allows polymorphism between Applicant, Officer, and Manager objects. Common methods like authentication or role-based access checks work uniformly across user subtypes.

I – Interface Segregation Principle (ISP)

The system follows ISP by designing focused interfaces tailored to each role's functionality. Instead of creating a single, large interface for managing enquiries, we split the concern into multiple role-specific interfaces:

- ApplicantEnquiryServiceInterface defines methods relevant only to applicants.
- OfficerEnquiryServiceInterface serves officers managing project enquiries.
- ManagerEnquiryServiceInterface focuses on enquiries at a higher level.

All these interfaces inherit from a common EnquiryServiceInterface that contains the default displayEnquiries() method for consistent output formatting.



This approach ensures:

- Role-specific services are not forced to implement unrelated methods.
- Each interface remains small, relevant, and easy to understand.
- Promotes code clarity and reduces coupling between different user functionalities.

This design also improves maintainability as adding a new method specific to applicants does not affect officer or manager services.

D – Dependency Inversion Principle (DIP)

The Dependency Inversion Principle is applied in our system to decouple **high-level modules** (such as services and interfaces) from **low-level modules** (like specific data repositories or entities), allowing both to depend on **abstractions** rather than concrete implementations. In our system, high-level menu classes (UI) depend on service abstractions rather than direct access to repositories or entity internals. For instance, OfficerMainMenu relies on JoinRequestService and ProjectRepository to retrieve or update information, promoting loose coupling and testability.

Service classes such as AccountService, ViewProjectService, and BookingService perform high-level operations like login, viewing/filtering projects, and assisting in flat bookings. Instead of these services being tightly coupled to data sources or direct database access, they rely on repository interfaces like ApplicantRepository and ReceiptRepository. The services interact with these repositories through abstract method calls such as findByNRIC(), save(), or getAllProjects(), without needing to know the underlying implementation details (e.g. whether data is stored in memory, a file, or a database).

The concrete implementations of the repositories **depend on the same abstractions**. If the data storage mechanism changes (e.g., from an in-memory list to a database), only the repository class needs to be modified. The service layer remains unchanged because it still calls the same interface-defined methods.

3.3.2 Additional Design Patterns

Serialisable Implementation

We initially considered saving our data into .csv files after the termination of the program, in order to achieve data persistence. This would work if the objects contained only primitive data types as attributes. For example, assume a Project object has attributes String name, String neighbourhood, etc. To save the object into a .csv file, simply let the file contain the corresponding columns, namely name, neighbourhood, etc.

Problems will arise when the object has an attribute that is another object, i.e. object composition. For example, a Project object will have a manager attribute, which is an instance of Manager. In this case, it is not possible to save the Manager into a .csv file, because it cannot be expressed in the form of plain text.

Our solution is to use serialisation. Serialisation is able to convert an object into binary data and save it in a .ser file. We created an abstract class called Repository, which has two methods, namely saveToSer() and importFromSer(). All of our repositories, such as the applicant repository and the manager repository, extend Repository.

Every time before the program terminates, all repository objects will be saved as .ser files. Then, at the beginning of the next start-up of the program, all .ser files will be read and converted back to repository objects. In this way, we have achieved data persistence while allowing for object composition.

Singleton Implementation

The repository classes in our project adopt the Singleton design pattern to ensure that only one instance of each repository exists throughout the application's lifecycle. This is achieved using a private static **instance** variable and a public static **getInstance()** method, which initialises the repository only once and provides global access to it. During the application's first run, getInstance() is used to instantiate and populate the repositories from CSV files. In subsequent runs, the repositories are restored from serialised .ser files to retain user data and project state. However, deserialisation normally bypasses the constructor, which can break the singleton guarantee. To resolve this, each repository implements the readResolve() method. This special method is invoked automatically during deserialisation and returns the existing singleton instance via getInstance(), effectively replacing the newly deserialised object. This ensures that the Singleton pattern remains intact even

after loading data from disk, maintaining consistency across the entire runtime and data lifecycle of the application.

4. Implementation (Java)

4.1 Tools Used

We used Java 17 on the Eclipse IDE, and handled version control using GitHub.

4.2 Sample code snippets

Encapsulation

In classes like ProjectApp, Applicant, and Officer, the attributes are marked as private and can only be accessed or modified using getters and setters. Methods like setWantToWithdraw() in ProjectApp ensure that sensitive information (e.g., whether an applicant wants to withdraw their application) is only changed in a controlled manner, preventing unauthorised or unintended changes.

```
public class Applicant extends User{
    /**
     *
     */
    private static final long serialVersionUID = 2241078442498755535L;
    private ProjectApp projectApp;


    public Applicant() {
        super();
        projectApp = null;
    }

    public Applicant(String name, String NRIC, int age,
            boolean isMarried, String password) {
        super(name, NRIC, age, isMarried, password);

        this.projectApp = null;
    }

    public ProjectApp getProjectApp() {
        return projectApp;
    }

    public void setProjectApp(ProjectApp projectApp) {
        this.projectApp = projectApp;
    }
}
```

Inheritance

In our system, there are several roles, such as Applicant, Officer, and Manager. These roles share common functionalities (e.g., login, registration) but have unique behaviours. Applicant and Officer both extend from the User class, which contains common methods and attributes like NRIC and password. This allows for shared functionality such as authentication while allowing each subclass to introduce specific behaviour for different roles.

```
public abstract class User implements Serializable {
    /**
     *
     */
    private static final long serialVersionUID = 4624565131962188794L;
    protected String name;
    protected String NRIC;
    protected String password;
    protected boolean isMarried;
    protected int age;

    public User() {
        this.name = null;
        this.NRIC = null;
        this.password = null;
        this.isMarried = false;
        this.age = -1;
    }

    public User(String name, String NRIC, int age,
            boolean isMarried, String password) {
        this.name = name;
        this.NRIC = NRIC;
        this.password = password;
        this.isMarried = isMarried;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
```

Polymorphism

Method Overriding: The UserMainMenu class has a method like viewProfile(), which is overridden in the subclasses (ApplicantMainMenu, OfficerMainMenu, ManagerMainMenu) to handle role-specific login behaviours. Here, the viewProfile() method is polymorphic. The appropriate version of the method is called based on the type of user logging in, even though the method signature is the same across all subclasses.

```
@Override
public void viewProfile() {
    System.out.println("\n--- Profile ---");
    System.out.println("Name: " + currentSessionApplicant.getName());
    System.out.println("NRIC: " + currentSessionApplicant.getNRIC());
    System.out.println("Age: " + currentSessionApplicant.getAge());
    System.out.println("Marital Status: " +
    (currentSessionApplicant.isMarried() ? "Married" : "Single"));
}
```

Method Overloading: We have different versions of a method based on input parameters, like the viewProject() method, which shows projects in different formats for different user types.

```
public void viewProjectsAsOfficer(Officer officer) {
    List<Project> handledProjects = allProjects.stream()
            .filter(p -> p.getOfficers().contains(officer))
            .collect(Collectors.toList());
    showFilterMenu(false);
    List<Project> filtered = applyFilters(handledProjects);
    printProjects(filtered);
}
```

Interface Use

We used a layered architecture to link between systems, such that certain complex features would have a separate interface accessible through the three main user menus. For example, an applicant will go from ApplicantMainMenu to ProjectAppMenu when they choose to manage their project application.

```
=== Applicant Main Menu ===
1. View Profile
2. Change Password
3. View Booking
4. Manage your project application
5. Manage your enquiries
0. Logout
Enter choice: 4
You are now managing your project application.
To choose an option, input the corresponding number.


=== Project Application Menu ===
1. View Available Projects
2. Apply for Project
3. Withdraw Application
4. View Project Details
5. View Booking Receipt.
0. Return to Applicant Menu
Enter choice:
```

Error Handling

Throughout our system, we implemented basic input validation (e.g., NRIC format, date checks, numeric entries) and handled edge cases like empty lists or overbooking. Errors were managed through conditional checks and user prompts, favouring clarity over complex exception handling to keep the CLI user-friendly.

```
while (true) {
        try {
        choice = Integer.parseInt(sc.nextLine());
        if (choice == 1 || choice == 2 || choice == 3) {break;}
        else {System.out.println("Invalid input. Please enter 1 or 2 or 3. ");}
    } catch (NumberFormatException e) {
        System.out.println("Please enter a valid number.");
    }
}
```

5. Testing

5.1 Test Strategy

To ensure that our program functions as intended, we used manual functional testing with a variety of test cases, comparing the results of executing the test cases with the expected results. No additional testing tools were used.

5.2 Test Case Table

Our test cases include all those available in the assignment, which we updated to fit our program, as well as four new cases we created to properly test the functionalities of our program. The test cases are shown in the table below, with new test cases highlighted in yellow.

No.	Test Cases	Expected Behaviour	Failure Indicators
1	Valid User Login	User can access their dashboard based on their roles	User cannot log in, receives incorrect error messages, or logs into wrong dashboard
2	Invalid NRIC or incorrect password	User is notified about incorrect NRIC or password and prompted to retry	User successfully logs in with invalid NRIC or incorrect password
3	Password Change Functionality	System updates password, prompts re-login and allows login with new credentials	System does not update password, denies access with the new password, or allows login with the old password
4	Project Visibility Based on User Group and Toggle	Projects are visible to users based on their age, marital status, and the visibility setting	Users see projects not relevant to their group or when visibility is off
5	Project Filtering for Users	In addition to the details in (5), projects are filtered based on users' filter choices for neighbourhood and flat type	Users see projects not in the filtered categories, or do not see projects in the filtered categories

6	Project Application	Users can only apply for projects relevant to their group and the projects' visibility is on	Users can apply for projects not relevant to their group or when the projects' visibility is off
7	Viewing Application Status after Visibility Toggle Off	Applicants continue to have access to their application details regardless of project visibility	Application details become inaccessible once visibility is off
8	Project Application after Rejection	Applicants can apply for other projects after being rejected from one project	Applicants are unable to apply for other projects
9	Single Flat Booking per Successful Application	System allows booking one flat and restricts further bookings	Applicant is able to book more than one flat
10	Applicant Enquiry Management	Enquiries can be successfully submitted, displayed, modified, and removed	Enquiries cannot be submitted, modified, or removed, or do not display correctly
11	HDB Officer Registration Eligibility	System allows registration only under compliant conditions	Allows registration while the officer is an applicant of the same project, or an officer of another project in the same period
12	HDB Officer Registration Status	Officers can view pending or approved status updates for their join requests	Status updates are not visible or incorrect
13	Project Detail Access for HDB Officer	Able to view the details of the project he/she is handling, even when visibility is turned off	Project details are inaccessible when visibility is toggled off
14	Restriction on Editing Project Details	Edit functionality is not available for HDB Officers	Officers are able to make changes to project details
15	Response to Project Enquiries	Officers and Managers can access and respond to enquiries, and responses are visible to the enquiring applicant	Officers and Managers cannot see enquiries, or their responses are not recorded or visible to the applicant
16	Enquiry Filtering for HDB Manager	Enquiries are filtered based on whether the manager chooses to only see enquiries regarding their active project	Manager sees all projects when the filter for only active project enquiries is on, or sees only active project enquiries when the filter is off
17	Multiple Officers Handling Booking	All officers of a project are able to assist in flat booking, but once one does so, no others can do so for that application	Officer unable to assist in flat booking, or multiple officers assisting in flat booking for the same successful application
18	Flat Selection and Booking Management	Officer retrieves correct application, updates flat availability, and logs booking details in the applicant's profile	Incorrect retrieval or updates, or failure to reflect booking details accurately
19	Receipt Generation for Flat Booking	Accurate and complete receipts are generated for each successful booking	Receipts are incomplete, inaccurate, or fail to generate
20	Create, Edit, and	Managers are able to add new projects,	Inability to create, edit, or delete projects

	Delete BTO Project Listings	modify existing project details, and remove projects from the system	or errors during these operations
21	Single Project Management per Application Period	System prevents manager from creating more than one project listing within the same application dates	Manager is able to handle multiple projects simultaneously during the same period
22	Toggle Project Visibility	Changes in visibility should alter the project list visible to applicants	Visibility settings do not update or do not affect the project listing as expected
23	View All and Filtered Project Listings	Managers should see all projects and be able to apply filters to narrow down to their own projects	Inability to view all projects or incorrect filtering results
24	Manage HDB Officer Registrations	Managers handle officer registrations effectively, with system updates accurately reflecting changes	Mismanagement of registrations or slot counts do not update properly
25	Approve or Reject BTO Applications and Withdrawals	Approvals and rejections are processed correctly, with system updates to reflect the decision	Incorrect or failed processing of applications or withdrawals
26	Generate and Filter Reports	Accurate report generation with options to filter by various categories	Reports are inaccurate, incomplete, or filtering does not work as expected

6. Documentation

6.1 Javadoc

Refer to the attached Javadoc.

6.2 Developer Guide

Open Eclipse > File > Open Projects from File System… > Import Source > bto-application-system > Click Finish. Open the bto-application-system folder in the Eclipse Project Explorer. On the first start-up of the program, delete the save folder (if any). Then, open the menu folder. Right click on Main.java. Select run as a Java application. For all subsequent start-ups, do not delete the save folder.

7. Reflection & Challenges

From the past 7 weeks, some things that went well were: a clear modular structure using OOP principles, our role-specific interfaces enhanced realism and usability and our repository system streamlined data management. Some things that could be improved if we were to launch this system as an actual product include: Adding a persistent database, serialisation is limited. Using GUI instead of CLI as it works, but lacks user-friendliness. Some lessons learnt were that the proper use of SOLID design principles and modularity eases future changes. Also, trade-offs between simplicity and extensibility are key in ensuring the robustness of our system.
For individual contributions, refer to the cover page.

8. Appendix

Github link: https://github.com/CooledWater/SC2002-BTO (No external tools were used)