Project Title: HDB Project 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 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.

Group Members:

Name	Course	Lab Group	Signature	Student ID
CHOO KYE YONG	SC2002	SCE3	[illegible]	U2323595K
KEVIN LIM ERN KEE	SC2002	SCE3	[illegible]	U2321523C
MISHRA ADITI RAKESH	SC2002	SCE3	[illegible]	U2320817A
RAASHI SINGH	SC2002	SCE3	[illegible]	U2323855J
TOM TANG GUAN LIANG	SC2002	SCE3	[illegible]	U2422299J

Chapter 1: Requirement Analysis & Feature Selection

1.1 Understanding the Problem and Requirements

To identify the main problem domain, we first tried to break down what different kinds of roles there were, and what actions were key to their role. We understood that the main users we were working with were Applicant, Officer and Manager, by studying the key nouns of the requirements. There were specific use cases for each party, such as submitting an application for the project (user), being in charge of a BTO project (officer), being able to view, create and delete listings of BTO projects (manager).

The key explicit requirements we understood were:

1. Since users will login with their Singpass account, we should have User Authentication via NRIC and password, with their information of age and marital status being present
2. Applicants
    a. Able to only view listings available to their age group and marital status (i.e. singles cannot view listings for couples)
    b. Able to look at the status of their application
    c. If application status is “successful” - able to book for a flat
3. Officers
    a. Possess all of applicant’s functionalities
    b. Can register to join a project
    c. Able to view and respond to enquiries they are handling
    d. With an applicant’s successful BTO application, Officer can

        i. Update the number of flat for each flat type that are remaining
        ii. Retrieve applicant's BTO application with applicant's NRIC
        iii. Update applicant's profile with the type of flat (2-Room or 3- Room) chosen under a project
    e. Able to generate receipt of the applicants with their respective flat booking details – Applicant's Name, NRIC, age, marital status, flat type booked and its project details.
4. Managers
    a. Able to create, edit, and delete BTO project listings
    b. Able to toggle the visibility of the project to "on" or "off"
    c. Able to approve or reject HDB Officer's registration as the HDB Manager in-charge of the project
    d. Able to approve or reject Applicant's BTO application
    e. Able to generate a report of the list of applicants with their respective flat booking – flat type, project name, age, marital status
    f. Able to view enquiries of ALL projects

Here are the implicit requirements (as well as some ambiguous parts we tried to clear up) which we noted:

1. To guard against password leaks, we salted, and peppered stored passwords hashes.
2. Unique enquiry id assigned to facilitate and track better multiple enquiries per (User, Project) pair
3. Input validation on all user inputs
4. Stack-based menu navigation, allowing new menus to be extended easily. Facilitates backtracking.
5. We will also ensure to handle invalid or missing CSV data, like skipping bad lines, to maintain stability of the program.
6. We were told that applicants are allowed to edit their enquiry after submitting it, but were not told on how to handle the situation if an applicant wanted to edit their enquiry once the enquiry has been responded to. (Was an ambiguous part in the assignment)

1.2 Deciding on Features and Scope

Below is a full list of notable features:

1. User Management:

    We included User login with NRIC and password, Password hashing with salt and pepper, Password reset functionality, Role-based access for applicants, officers and managers, NRIC input and validation.

2. Project Management:

    We included Managers creating, editing and deleting projects, a project visibility toggle (public and private), Date formatting and input validation, Officer having limited slots per project, a Filtering and sorting of projects (by name, flat type, etc).

3. Housing Application System:

    We included applicants viewing eligible projects and applying, Applicants requesting withdrawal for applications, Manager approving request of withdrawal, Officer booking flats for applicants, Housing Request History

4. Enquiry System:

    We included applicants creating, editing and deleting enquiries, Officers and Managers responding to enquiries, Enquiries are given a unique ticket ID, Enquiries cannot be edited once Officer/Manager has responded.

5. Officer Assignment:

   We included officers registering for projects with available slots, managers approving officer requests, Officers unable to be assigned to more than a project per timeframe, assignment status tracking.
6. System Flow:

   We included Data initialisation from CSV files, Record Saving to CSV, CLI menu system with stack-based navigation, Filtering/sorting projects, Checking for time conflict for project overlaps.

We prioritized these features based on how important they were to the system's functionality, how practical it was to implement them in accordance with the team's technical abilities, as well as how much time it would take for us to implement them.

Here is the list of our core features, optional features and some features we thought to exclude:

Core Features:

Role-based login with password hashing, Manager project creation and visibility toggle, Application submission, status tracking, and withdrawal, Officer assignment workflow, Enquiry creation and response system, CLI interface with menu navigation, CSV read/write for data persistence, Validation of NRIC, project dates, and user inputs.

Optional Features:

Password reset functionality (PasswordResetHandler), Unique enquiry ticket ID system (Enquiry.totalTickets), Preventing enquiry edits after staff reply, User preference filtering and sorting options (UserPrefSorting), Conflict-of-interest logic for project applications (Project.conflictInterest), Formatted date/time using DateTimeFormat, Stack-based menu navigation.

Features we excluded (and why we left them out):

Excluded Features	And why we left them out
Visual reports, homepage in a dashboard format	Required by the assignment to be CLI based
Undo/redo actions for application or project changes	Would have gotten too complex and unnecessary

Chapter 2: System Architecture & Structural Planning

2.1 Planning the System Structure

Our system began with a basic structure centered around Main.java, but as our understanding of OOP matured, we adopted a layered architecture with Entity, Control, and Boundary packages. This allowed us to have better modularity and scalability, such as with the introduction of MenuNavigator. This allowed users to navigate menus with stack-based logic, which reduced hardcoding. We also mapped out user flows early, such as flat booking and officer registration, and aligned them with our components. Although we followed the layered structure, we allowed exceptions where the boundary classes accessed entity classes directly to streamline the process. By creating flowcharts and diagrams early into our design phase, we were able to visualize user interactions and help guide the creation of our modular components and imagine how features like enquiry editing would flow from start to end.

2.2 Reflection on Design Trade-offs

Designing the menu system revealed key trade-offs. Early versions were simple, but the addition of functions such as hiding options and dynamic updates led to more complex abstractions. We considered using patterns such as Factory or Builder to handle overloaded constructors but ultimately kept our existing structures to avoid major rewrites. Although this increased the chances of IDE issues (e.g. giving a code block instead of a lambda expression), it preserved functionality, We also prioritized modularity by using helper classes, and security by implementing password hashing. We also acknowledged the tension between data traceability and code simplicity. Maintaining lists such as HousingReqList and EnquiryList increased complexity but improved traceability. Our structural planning evolved alongside our OOP understanding, as we tried to adhere to the principles of modularity, maintainability and user experience.

3. Object-Oriented Design

3.1 Class Diagram (with Emphasis on Thinking Process)

We tried to identify the main classes by trying to extract the nouns and essential roles which came up more frequently than others in the problem description. Some of the main classes we ended up recognising were the main users; Applicant, Officer, Manager, User etc.

With these main users in hand, analysing the surrounding entities that these users are interacting with, we conducted actor-goal analysis, and recognised some main entity functions which came up consistently like the main Menu, Housing Application and Project.

If we thought an attribute had a more significant purpose, we elevated it to be a class; for example, rather than keeping enquiries as an attribute of the User, we recognised that it had its own logic, like tracking the status of an enquiry and its timestamp etc. Since we realised that it had its own behaviour with its own attributes, we decided to elevate it to a class.

The responsibilities of each class has been given below:

1. Boundary
    a. Console
        i. *Class* AppScanner: for input handling, *Class* ConsoleCommands: simply gives feel of “clearing page”, *Class* DateTimeFormat: just formatting the date time
    b. EnquiryIO
        i. *Class* EnquiryPrinter: displaying enquiry in console, with it able to print all messages in a single enquiry, print enquiry list, can view enquiry list, *Class* EnquriySelector: safely letting user select an enquiry it has access to
    c. HousingApplyIO
        i. *Class* HousingReqPrinter: print house application of user
    d. Menu
        i. *Class* EnquiryMenu: applicants, officers and managers can manage enquiries , *Class* MainMenu: menu interface for user, *Class* OfficerApplyMenu: interface only for officers who want to manage their applications to become HDB project officers, *Class* ProjectApplicationMenu: lets users view BTO projects, apply to projects, review their application history and request to withdraw an application, *Class* ProjectManageMenu: builder for manage your projects section, available to officers and managers

e. menuTemplate
   i. *interface* MenuAction: how menu items should be behaving in console, *Class* MenuGroup: represents a group of menu items, *Class* MenuItem: represents single option in a menu system, *Class* MenuNavigator: lets you navigate into further submenus or back into bigger menus, *Class* SelectionMenu: can show list of options as dynamic options
f. OfficerAssignIO
   i. *Class* OfficerAssignPrinter: prints assignment request information, *Class* OfficerAssignSelector: for managers to view assignment requests and accept or reject
g. ProjectIO
   i. *Class* ProjectPrinter: displays project details to user, *Class* ProjectSelect: selecting the project and displaying to user, *Class* ProjectSelector: actual logic to let different groups of users select projects and view projects based on diff criteria, *Class* RoomTypeSelector: let user select room type, *Class* SetUpProject: getting details for new project and creating a new project based on details, *Class* UserPrefSorting: let users sort by neighbourhood, number of rooms etc
h. Security
   i. *Class* LoginHandler: how to ask user to log in etc, *Class* PasswordResetHandler: reset hashed password for user, *Class* PasswordVerifier: verifies whether a given input password matches the stored password hash for a user, *Class* UserValidator: verify that the NRIC is correct

2. Control
   a. Enquiry
      i. *Class* Enquiry: models an enquiry as a collection of message objects related to a user and a project, *Class* EnquiryList: separate enquiry list for each of the user, project and main classes (manages creation, retrieval, selection and deletion of enquiry objects bw users and projects in the system), *Class* Message: user can edit message, manager can get timestamp, content of message and who sent it
   b. housingApply
      i. *Class* HousingReq: let users create application for house, *Class* HousingReqList: part of every user's attribute, part of every project's attribute, in the main function which contains every request
   c. officerApply
      i. *Class* AssignReq: represents officer's request to join a specific project, *Class* AssignReqList: controller/helper class that wraps the list and controls how assignment requests are made or removed
   d. Security
      i. *Class* HashingUtils: hashes an input string (password) and generating a salt and peppered password *Class* Password: stores hashed password with salt and pepper, *Class* Pepper: defining what the pepper will be, *Class* UserFetcher: fetch user based on NRIC and verifies provided password
   e. *Class* Main: entry point
   f. *Class* TimeCompare: to ensure that manager does not assign 2 conflicting projects (happening at the same time) to an officer

3. Entity
   a. Caching

        i. *Class* DataInitialiser: loading data into program from CSV files, *Class* RecordSaver: saving the updated data back into CSV files

    b. Project

        i. *Class* Project: encapsulates all the data related to a housing project, including unit details, officers, and the housing request process, *Class* ProjectList: retrieving a project from this list by its name

    c. Users

        i. *Class* Applicant: extends the User class and acts as a specialized subclass for users who are applicants, *Class* Manager: extends the officer class and adds functionality specific to managers within the system, *Class* Officer: extends the applicant class and adds functionality specific to officers working on projects, *Class* User: holds personal and account information about a user, including their name, age, marital status, password, as well as a list of housing requests (reqList) and enquiries (enquiryList), *Class* UserList: a specialized list of User objects that extends ArrayList<User>. It provides additional methods to search for users by their UserId or Name, and customizes the way users are represented as a string.

Relationships between the classes were determined based on the nature of their interactions. There are five kinds of relationships we have worked with here; association, dependency, inheritance, aggregation and composition. **(Refer to Picture1)**

Composition: Enquiry composed of User, String, Project, Message **(Refer to Picture2)**

Aggregation: Project is aggregation of LocalDate **(Refer to Picture3)**

Association: DateTimeFormat associated with DateTimeFormatter **(Refer to Picture4)**

Dependency: EnquiryPrinter depends on enquiry. **(Refer to Picture5)**

Amongst of the trade-offs we considered was abstraction vs performance. We introduced helper classes such as EnquiryPrinter and OfficerAssignSelector which adds in more layers of abstraction, but does impact performance, compared to using more direct logic in fewer classes. We also considered data integrity vs the simplicity of our code, where we opted to store request histories such as HousingReq, to improve data traceability, and a more transparent BTO management system. However, this did end up adding complexity in our classes. We also considered the trade off for security vs the ease of implementation, where for login, we chose to implement our password hashing with salt and pepper, increasing the security and prioritizing data protection over implementation simplicity. Lastly, we also tried to aim for modularity by having one class have a single responsibility, such as ProjectPrinter, printing out the details of the project to the user.

3.2 Sequence Diagram (with Emphasis on Thinking Process)

The entire sequence of a HDB officer applying for a BTO, registering to handle a BTO project and handling a successful booking can be broken down into 4 separate sequence diagrams.

3.2.1 Sequence Diagram 1: User logging into the BTO System as an Officer

<u>Process Flow</u>:

1. The user first inputs their NRIC, which is validated by the UserValidator class. The user must provide a valid NRIC before proceeding.
2. The LoginHandler then prompts the user for a password
3. Both the input NRIC and password are then passed to the UserFetcher class to retrieve the associated user from the system

4. The UserFetcher class calls the PasswordVerifier class to compare the entered password with the stored one. To enhance security, the system defines a HashingUtils class which hashes passwords with salt and pepper before comparison, ensuring sensitive information is never stored in plaintext.
5. If the user credentials are valid, the LoginHandler notifies the Main class that the login is complete. If the credentials are incorrect, an error message is displayed, prompting the user to try again (with a maximum of 5 failed login attempts)

3.2.2 Sequence Diagram 2: Officer registering to handle a BTO Project

Process Flow:

1. The officer interacts with the MainMenu, which triggers the MenuNavigator to refresh and display available options through the Console
2. The officer selects the "Manage your HDB Officer applications" option, which leads to the OfficerApplyMenu being displayed.
3. The officer then selects the "Join project as HDB Officer" option, which is handled by MenuNavigator and passed to MenuItem for execution
4. The officer then selects a project to apply for with the ProjectSelector class. First, UserPrefSorting class filters projects based on available slots and the officer's constraints. This list is then passed back to ProjectSelector, which prints the project details in a clear, readable format using ProjectPrinter on the Console for the officer to choose from
5. After the officer selects a project from the list, the selected project is added to the officer's list of assigned projects in Main

3.3 Application of OOD Principles (SOLID) — With Emphasis on Thinking Process

S - Single Responsibility Principle (SRP)

From the outset we partitioned the codebase into three layers—Boundary, Control, and Entity—so that each folder has one overlying theme, allowing SRP to be applied easier on the classes.. Boundary classes such as EnquiryMenu deal only with user interaction. It gathers input through AppScanner and formats output. Entity classes such as DataInitializer load existing enquiries from CSV files and store new states. Sitting between them, Control classes like Enquiry orchestrate workflows, validating input from Boundary and mutating Entity objects.

We extended the SRP further downstream to utility modules. In the Boundary layer, AppScanner manages a global scanner instance to avoid multiple instantiations, ConsoleCommands handles console clearing, and DateTimeFormat manages all formatting of date and time strings across the application. Each of these classes exists for a single purpose, which makes them easier to test, maintain, and replace. As an illustration, replacing the method inside ConsoleCommands.clearConsole() with an ANSI escape sequence affects only that class, while the rest of the system remains untouched.

O - Open/Closed Principle (OCP)

OCP guided the design of our menu system, particularly within the program.boundary.menuTemplate package. We built a flexible hierarchy of menu components: MenuItem, MenuGroup, SelectionMenu which are centered around the *MenuAction* functional interface, which defines a single abstract execute() method. This abstraction allows different menu behaviours to be implemented independently without changing the core menu logic. Each MenuItem is simply a description linked to a *MenuAction*, while MenuGroup aggregates multiple *MenuAction* together.

This design enables extensibility through composition and inheritance. For example, creating a new user-facing menu involves creating a new class that extends MenuGroup and adding in the appropriate MenuItem with their respective *MenuAction*. The underlying logic for navigation, input validation, and stack-based history is handled centrally by the MenuNavigator class. As a result, new menus can be introduced or updated without modifying the existing infrastructure, keeping it closed to modification but open to extension. This helps keep our codebase modular and reduces the likelihood of introducing bugs in unrelated parts of the system. The trade-off is the upfront effort to establish this structure, but it pays off in scalability and code reuse.

<u>L - Liskov Substitution Principle (LSP)</u>
LSP was applied throughout the user inheritance structure. When a user first enters the mainMenu, they will be greeted and this greeting depends on the class of user (Applicant, Officer, Manager). Within the entity layer, Applicant extends User, Officer extends Applicant and Manager extends Officer. Each subclass overrides the getGreeting() method, which is used during login to display a role-specific welcome message. Despite these differences, they can all be treated as a User in broader logic, such as list management and login handling, without breaking functionality. This promotes flexibility and ensures that subclassed types can be substituted for their base type safely.

However, one tradeoff was the need to ensure overridden methods in subclasses did not violate the expectations set by the base class. In the case of getGreeting(), its return type is a String but there was a chance of the code throwing an error when the Officer was not currently handling a project. Throwing an error which the base class would not have done would require special exception handling and violates LSP. As such careful design was necessary to ensure all overridden methods would adhere to behavioural expectations set by the base class as seen below where an if statement is utilized to prevent thrown exceptions.

<u>I - Interface Segregation Principle (ISP)</u>
ISP is clearly demonstrated in our menu system through the design of the *MenuAction* interface. ISP advocates that interfaces should be minimal and specific, so that implementing classes are not forced to depend on methods they do not use. Our *MenuAction* interface, located in program.boundary.menuTemplate.MenuAction, contains only a single abstract method execute(), allowing it to serve as a focused contract for any class that defines a menu behavior.

This interface is used throughout the MenuItem, MenuGroup, and SelectionMenu classes. Each menu item only requires knowledge of this one method to trigger its associated behavior. For instance, in MenuItem.java, a *MenuAction* is stored as a field and invoked when the item is selected. The tradeoff of implementing ISP is that each behaviour must be a separate class which can lead to highly complex systems with multiple similar interfaces with only slight differences. However this is outweighed by the benefit of easier maintainability in the future.

<u>D - Dependency Inversion Principle (DIP)</u>
DIP is demonstrated in our project through the use of abstractions in the menu system through the use of the *MenuAction* functional interface. DIP suggests that high-level modules should not depend on low-level modules

directly and instead should both depend on abstractions. Instead of having high-level modules like MenuNavigator and MenuItem depend directly on specific action implementations, they depend on the abstraction *MenuAction*, which defines a single method execute(). This design decouples the logic for user interaction from the logic of what each menu item actually does. As shown above in the OCP portion, DIP and OCP work hand-in-hand to achieve a more flexible, maintainable and scalable system. DIP decouples the modules and OCP ensures we can extend the system with new features without modifying existing classes.

4. Implementation (Java)

4.1 Sample Code Snippets

Encapsulation

Encapsulation is demonstrated in the Password class where internal data: password hash and salt are declared as private fields. These variables are not directly accessible from outside the class. Instead, they are exposed via public getter methods. This design ensures that sensitive information remains protected and only accessed or modified in controlled ways. The constructor handles the generation and hashing of passwords using internal logic, reinforcing the idea that the internal state of the object is managed privately. Encapsulation like this improves security, promotes modularity, and reduces the risk of unintended interference with an object's state. **(Refer to Picture6)**

Inheritance

The user hierarchy applies inheritance where Applicant extends User, Officer extends Applicant and Manager extends Officer. This chain of inheritance allows the Manager to inherit all properties and methods of Officer and User, including fields like NRIC, name, and methods related to authentication or enquiries. This structure avoids code duplication and promotes reusability. Inheritance also allows specialized behavior to be added in subclasses when necessary, without altering the original base class, keeping the code extensible. **(Refer to Picture7)** As seen above in the Officer class where the greeting() method is overridden, instead of completely replacing the base method's behaviour, it calls super.greeting() to reuse the base implementation and extend it further with additional context.

Polymorphism

Polymorphism is present when handling different user roles under a common User type. In the UserFetcher class, a variable client is declared as type User, but at runtime it may reference an Applicant, Officer, or Manager object. This is known as upcasting, where a subclass object is treated as a superclass reference. Regardless of its actual subclass, the object is treated uniformly as a User when performing actions like verifying passwords or logging in. At runtime, dynamic binding ensures that the correct subclass implementation is used when calling overridden methods. For example, if the greet() method were overridden in each user type, then calling client.greet() would invoke the appropriate version for Applicant, Officer, or Manager depending on the actual object type, despite the client typed as User. **(Refer to Picture8)** In contrast, static binding applies to methods that are resolved at compile-time, such as overloaded methods. Method overloading is when multiple methods in the same class share the same name but differ in parameter types or count. The Applicant class is an example of overloaded constructors which accept various argument combinations. These constructors provide different initialization options, and the correct one is chosen by the compiler based on the arguments provided. **(Refer to Picture9)**

Interface Use

MenuAction is a functional interface which has a single abstract method, execute(), which is implemented by all menu actions in the application. For example, menu items that trigger logouts, open submenus, or perform tasks all implement MenuAction. This design allows the MenuNavigator and MenuItem classes to depend only on the abstract behavior defined by the interface, not on the specific details of what each action does. Interfaces like *MenuAction* provide a clean contract for behavior, support dependency inversion, and allow components to be swapped or extended with minimal coupling. **(Refer to Picture10)**

Error Handling

Error Handling is present throughout multiple classes, particularly in boundary classes like UserValidator which controls user I/O. When taking user input for NRIC, the application wraps the validation logic in a try-catch block. If the NRIC is invalid due to wrong formatting or incorrect length, an exception is thrown with a clear message, and the user is prompted to try again. This approach prevents the program from crashing on bad input and ensures a smoother user experience. By anticipating possible errors and handling them gracefully, the system becomes more reliable and user-friendly. **(Refer to Picture11)**

Chapter 5: Testing

5.1 Test Strategy

Satisfy all client (lecturer) requirements. In interest of brevity, the below test cases are the directions to conduct the tests manually. We based our saving mechanism on csv files in the interest of easier testing. We first considered using gradle scripts to execute junit tests, but we were worried about requiring the developer to install gradle support on their device first. Hence, we conducted the tests manually.

5.2 Test Case Table

1. User should be able to access their dashboard based on their roles **(Refer to Picture12)**	2. User receives a notification about incorrect NRIC format **(Refer to Picture13)**
3. System should deny access and alert the user to incorrect password **(Refer to Picture14)**	*Log In (S1234567A, password) → Reset Password → newPassword → Go Back* *Log In (S1234567A, newPassword) → Go Back* 4. System updates password, prompt re-login and allows login with new credentials
Log In (S1234567A, password) (John, 35, Single) → Manage Your BTO Application → *View Projects: Visibility False Breeze is not shown* 5. Projects are visible to users based on their age, marital status and the visibility setting	*Log In (T2345678D, password) → Manage Your BTO Application → View Projects:* *Visibility False Breeze not shown.* 6. Users can only apply for projects relevant to their group and when visibility is on
Log In (T2345678D, password) → Manage Your BTO Application → View Projects: *No Officer Slot Breeze shown due to previous application, despite visibility == false (check projectList.csv for proof).* 7. Applicants continue to have access to their	8. System allows booking one flat and restricts further bookings **(Refer to Picture15 & Picture16)**

application details regardless of project visibility	
Log In (S1234567A, password) → Manage Enquiries → Create Enquiry/ View Enquiry/ Edit Enquiry/ Delete Enquiry *Log In (S5678901G, password) → Manage Enquiries → Reply Enquiry* 9. Enquiries can be successfully submitted, displayed, modified, and removed	*Log In (S6543210I, password) → Manage your HDB Officer applications → Join project as HDB Officer → Only Not Yet Open Breeze available, others all intersect with her current working times* *If use (S7352210I, password) then can apply anywhere due to not having a current assignment.* 10. System allows registration only under compliant conditions
11. Officers can view pending or approved status updates on their profiles **(Refer to Picture17)**	*Log In (T1918537J, password) → Manage your BTO Application → View projects: Only VISIBILITY FALSE BREEZE is printed, due to Primary School Student working there, despite visibility == false, and all others are not visible due to his age* 12. Officers can always access full project details, even when visibility is turned off
Log In (S3645932G, password) (Manager) → Manage your Projects → Manage BTO Project Listings → Edit Project Listing *Log In (T2109876H, password) (Officer) → Manage your Projects → no Manage BTO Project Listings option.* 13. Edit functionality is disabled or absent for HDB Officers	*Log In (S5678901G, password) → Manage Enquiries → Reply Enquiry* 14. Officers & Managers can access and respond to enquiries efficiently
Log In(T9574576H, password) → Manage your Projects → Help applicant book → Choose application (Updates the flat availability and logs booking details). *Check Applicant bookings through* *Log In() → Manage your BTO Application → View your applications* 15. Officers retrieve the correct application, update flat availability accurately, and correctly log booking details in the applicant's profile	**(Refer to Picture18)** *Log In(T9574576H, password) → Manage your Projects → (Option appears if helped others booked any projects before) → Generate Receipt of Applicant Booking* 16. Accurate and complete receipts are generated for each successful booking

LogIn (S3645932G, password) → Manage your Projects → Manage BTO Project Listings → Create Project Listing/ Edit Project Listing/ Delete Project Listing 17. Managers should be able to add new projects, modify existing project details, and remove projects from the system	(Refer to Picture19) *LogIn (S3645932G, password) → Manage your Projects → Manage BTO Project Listings → Create Project Listing/ Edit Project Listing* 18. System prevents assignment of more than one project to a manager within the same application dates
LogIn (S3645932G, password) → Manage your Projects → Manage BTO Project Listings → Toggle Visibility of Project Listing 19. Changes in visibility should be reflected accurately in the project list visible to applicants	*LogIn(S3645932G, password) → Filter options* *LogIn(S3645932G, password) → Manage your Projects → View projects* 20. Managers should see all projects and be able to apply filters to narrow down to their own projects
LogIn(T5678201G, password) → Manage your Projects → View/Approve incoming Officer requests → Officer: Jobless | Status: applied 21. Managers handle officer registrations effectively, with system updates reflecting changes accurately	<After Test 21.> LogIn(S7352210I, password) → Login greeting includes: You are currently handling project: COMPLETELY NO ROOM BREEZE. Check project details through: Manage your BTO Application → View projects Check profile through: Manage your HDB Officer applications → Check your Profile 22. Approvals and rejections are processed correctly, with system updates to reflect the decision
LogIn(T8765432F, password) → Manage your projects → Generate report on Applicants → (Report shown) → Select Filter or go Back 23. Accurate report generation with options to filter by various categories	

6. Documentation

The documentation is predominantly done in HTML javadocs in the source code.