






Project Title: BOTTOMS (BTOMS) Build-To-Order 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.

Name	Student ID	Lab Group	Date/Signature
Perrin-Owen Heng	U2421956J	FCSA (Group 2)	 20/04/25
Tay Ghim Long Javier (Zheng Jinlong)	U2420787H	FCSA (Group 2)	 20/04/25
Seng Qi Ming	U2420037F	FCSA (Group 2)	 20/04/25
Jessica Joyson	U2420658H	FCSA (Group 2)	 20/04/25
Quan Hui Shan Audrey	U2422588L	FCSA (Group 2)	 20/04/25

Chapter 1: Requirement Analysis & Feature Selection

1.1 Understanding the Problem and Requirements

Our team first had a meeting to discuss the functionalities and nuances of the main BTO PDF. It is imperative that everyone agrees with the interpretation of the PDF so we can work more efficiently. We recognised that the system was mirroring the Real World BTO application process in Singapore and as such, used that as our overhead. After analysing, we concluded that the core domain objects are: **Applicant, Application, Project, Enquiry, User, HDB Officer, HDB Manager.**

Explicit requirements:

User - Login with NRIC and password, able to change password. All users need to have marital status and age.

Applicants - View Eligible projects, ability to apply or withdraw for projects, view application status, submit/edit enquiries.

HDB Officer - All applicant's capabilities, register for project, view and reply enquiries.

HDB Manager - Create/Edit/Delete projects, toggle visibility, approve/reject all applications /registration/withdrawals, view and reply enquiries, generate receipt/report.

Developed as a Command Line Interface (CLI) application.

No database application allowed.

Implicit Expectations:

Any Singles <35 cannot apply for any projects. Booked flats cannot be rebooked. Flats upon successful booking must decrease availability. Officers can only book when an application is successful. Withdrawal handling after booking was vague. Officer handling of multiple projects was ambiguous in the original doc. User experience was not defined.

To combat these, we referred to the FAQ and separated parts into critical or cosmetic purposes. For every unclear issue, we would ask the whole group and resolve or align our interpretation. Some of the difficulties encountered were ambiguous parts and different interpretations of the requirements. The original document was very intimidating so we divided tasks equally and ensured that each person had a small part to focus on. Converting requirements to code was a big hurdle as most of us had never done that before and were only solving coding problems.

We learned how to apply not just coding but real life Object Oriented Design Principles into our project. We learnt how to structure our code in such a way that makes it reusable and understood the importance of UML class and sequence diagrams.

1.2 Deciding on Features and Scope

We broke down the main analysis into 3 sections and prioritized them accordingly to importance, feasibility and timeline. We defined the 3 criterias as followed:

Importance: How integral it is to the system process.

Feasibility: How easy it is to create and reuse. How much time and storage does it need?

Timeline: A prediction of how long each feature may take to develop.

Core Features (Must Have)	Bonus Features (Can Have)	Excluded Features (Shouldn't Have)
User Login with NRIC and default password	Filter by location, room type	GUI interface (Only CLI allowed)
Change password	Login attempt limits	JSON/XML storage (Prohibited)
View Eligible Projects	Customisation of management system	Undo system (High implementation requirement but low value: not worth)
Project visibility	Audit/action logger	Role creation/deletion during runtime (Roles already fixed based on Excel)
Apply for Flats/BTO	CLI enhancements	
Application Process (Pending -> Successful\Unsuccessful -> Booked\Not Booked)	Backup/Autosave system	
CRUD functionalities for both enquiries and project		
Withdrawal approval/rejection		
Officer registration and approval		
Officer booking of Flats		
Receipt generation		

Chapter 2: System Architecture & Structural Planning

2.1 Planning the System Structure

We adopted the Model-View-Controller (MVC) pattern to ensure modularity, testability, and maintainability. MVC aligns with the Single Responsibility Principle (SOLID) by clearly separating concerns into three distinct layers: Models (business data), Controllers (logic and validation), and Views (CLI user interaction). We first analyzed functional requirements, then structured key roles and entities (Users, Projects, Applications) accordingly.

Breakdown of Logical Components

Following the MVC pattern, our system is organized into three core layers:

- Model Classes (Entities)
 - **User** (abstract class) and its subclasses: **Applicant**, **HDBOfficer**, **HDBManager**
 - **Project**, **Application**, **Enquiry**, **FlatType**
- View Classes (Boundaries)
 - **LoginCLI**, **ApplicantCLI**, **ApplicationCLI**, **EnquiryCLI**, **ManagerCLI**, **OfficerCLI**
- Controller Classes (Controls)
 - **AuthController**: Handles login, password changes, and user lookup
 - **ApplicationController**: Handles all actions that affects the application
 - **EnquiryController**: Manage applicant enquiries
 - **ManagerController**: Manages all project CRUD functionalities & officer approval
 - **OfficerController**: Handle officer requests and flat booking

Mapping User Flows to System Structure

We began by analyzing the assignment requirements and translating them into clear user stories, such as:

- An applicant can apply for only one project at a time
- An officer cannot register for a project if they have an ongoing application
- A manager can approve or reject applications, respecting flat type limits

End-to-end user flows for each role:

- Applicant Flow:
 - Login → View eligible BTO projects → Apply (once only) → Check status → Book flat (if successful) → Submit/manage enquiries

- This flow is managed by **LoginCLI**, **ApplicantCLI**, **ApplicationController**, and **Application** model.
- HDB Officer Flow:
 - Login → Register for a project → View applicants → Approve applications → Book flats → Generate receipts → Respond to enquiries
 - This flow involves **OfficerCLI**, **OfficerController**, **ApplicationController**, **Project**, and **Application**.
- HDB Manager Flow:
 - Login → Create/edit/delete projects → Toggle visibility → View officer registrations → Approve/reject officers and applicants → Generate reports
 - Managed through **ManagerCLI**, **ManagerController**, and the **Project** and **Application** model classes.

These flows ensured clear enforcement of business rules (e.g., single application limits, booking constraints) and well-defined responsibilities across MVC layers, improving maintainability and reducing duplication.

Early Visual Models (e.g., Flowcharts)

To support our planning, we created flowcharts for:

- Login & Role routing Flow

Start -> inputNRIC -> Input Password -> RoleRouting

- Application Submission Logic

RoleRouting-> ApplicantMenu -> ViewProject-> SelectProject -> SubmitApplication -> ApplicationPending

- Flat Booking & Receipt Generation

RoleRouting-> OfficeMenu-> ViewApplications -> SelectApplicant -> BookFlat -> GenerateReceipt

- Project Creation & Officer Approval

RoleRouting-> ManagerMenu -> CreateProject -> SaveProject -> ViewOfficerRegistrations -> ApproveOfficer

These visualizations were essential in clarifying responsibilities and identifying potential issues early, aligning with the rubric's emphasis on robust planning and design thinking.

2.2 Reflection on Design Trade-offs

During the planning phase, we encountered several trade-offs and had to make critical design decisions.

Design	Option A	Option B	Decision
Controller Structure	Separate logic & role-based controllers	Combine logic with controllers	Option B
User Class Hierarchy	Use inheritance (Applicant , Officer , etc.)	Use one User class with flags	Option A
Status Tracking	Use Strings for statuses	Use enum for type safety	Option B
View Design	One unified CLI	Role-specific CLI classes	Option B

Controller Structure:

- Separating controller logic into distinct classes (**AuthController**, **OfficerController**, etc.) facilitated clearer boundaries and easier testing.

User Class Hierarchy:

- Using inheritance allowed specific user behaviors (e.g., officer-specific project registrations) to be clearly defined without cluttering the base **User** class.

Status Tracking:

- Enum usage (**Application.Status**) ensured accurate application state management, preventing logic errors from arbitrary string values.

View Design:

- Separate CLI views per role enhanced readability, reduced conditional complexity, and simplified debugging.

Our team also debated how to handle project and application constraints. For instance, we discussed combining flat booking and application logic but ultimately decided to separate them to maintain controller focus and clarity. These design trade-offs were carefully considered to balance simplicity with extensibility and ensure long-term maintainability of the system.

Chapter 3: Object-Oriented Design

3.1 Class Diagram

In developing the UML Class Diagram, we extracted key nouns and entities from the assignment requirements to identify the main classes. This included main user roles such as **Applicant**,

HDBOfficer and **HDBManager**, which are inherited from the **User** class. We identified this BTO system's core functionalities and modeled them as **Project**, **Application** and **Enquiry** classes.

User allows for authentication and stores the profile information of all users. **Applicant** allows applicants to view and apply for projects, and submit enquiries. **HDBOfficer** allows officers to register for projects, reply to enquiries, along with all of the applicant's capabilities. **HDBManager** allows managers to create and manage project listings, approve officer applications and generate reports. **Project** contains all the relevant information about a BTO project listing. **Application** ensures that an applicant only applies for one project at a time, and shows the status of this application. **Enquiry** allows applicants to enquire about specific projects and receive replies from officers and managers.

Inheritance was used when a generalisation and specialisation hierarchy exists. For example, **Applicant**, **HDBOfficer** and **HDBManager** classes inherit from the **User** class, showing an "is-a" relationship. Composition was used to imply a whole-part relationship. For example, the relationship between **HDBManager** and **Project** suggests that a manager creates a project, and a project cannot exist without a manager, showing a "has-a" relationship.

As for trade-offs, we prioritised simplicity by using concrete controller classes instead of introducing additional interfaces or abstract classes for every role. We also prioritised abstraction by ensuring that overly deep inheritance chains that could potentially hinder performance were not used.

3.2 Sequence Diagrams

The sequence diagrams we selected - Officer Assigns Flat to Applicants, Login, and Officer Applying for BTO - represent critical workflows involving the HDB Officer and were chosen for their complexity and relevance to the system's goals. These scenarios involve multiple object interactions, validation logic, and conditional flows, making them ideal for demonstrating how the system operates under realistic conditions. They follow a controller-service-repository

architecture, showcasing how user actions move through the CLI interface, are processed by various controllers, and ultimately interact with models and registries. These use cases reflect core BTO operations such as application submission, officer registration, and flat assignment, which are central to the system. The diagrams also demonstrate strong object collaboration across registries, controllers, and domain models, helping us confirm proper assignment of responsibilities and communication. Notable design patterns include conditional logic (e.g., verifying if an officer has already applied), state updates (such as marking an application as BOOKED), and report generation (e.g., creating a receipt after flat assignment), all of which reinforce the system's functional integrity.

Sequence Diagram 1: Login Flow (Refer to Appendix A Fig 1)

This diagram captures how the login process incorporates real-time collaboration between the CLI and the AuthController, emphasizing limited login attempts and secure NRIC-password verification. It showcases the use of conditional logic (e.g., invalid user or incorrect password) and recursive handling (e.g., re-prompting on failure), which ensures the login logic is robust, reusable, and easy to maintain. Including this diagram helps illustrate how essential user input is processed alongside backend validation in a controlled, layered architecture.

Sequence Diagram 2: Officer applying for a BTO Project (Refer to Appendix A Fig 2)

This diagram models an administrative action where an HDB Officer submits an application for a BTO project. The scenario includes validating project visibility, retrieving project and flat type information, and checking for flat availability before updating the system. It helps verify our system's support for role-based operations - officers acting as applicants - and highlights how registration data is updated conditionally. The interaction between the CLI, ApplicationController, ProjectRegistry, and ApplicationRegistry illustrates how state changes and validations flow through the controller-service-repository structure.

Sequence Diagram 3: Officer Assigning Flat (Refer to Appendix A Fig 3)

This sequence captures a key administrative workflow where an officer assigns a flat to a valid applicant. The process includes verifying the application, updating its status to BOOKED, retrieving the relevant project and flat type, and booking the unit in the system. Finally, a receipt

is generated to confirm the transaction. This diagram demonstrates important logic such as conditional flows (e.g., whether the application exists), permission checks (e.g., ensuring the officer is authorized to assign a flat), and system updates (e.g., booking the flat and updating statuses). It reinforces the coordination between the CLI, controllers, registries, and models, ensuring proper enforcement of business rules and reflecting the system's real-world operational needs.

3.3 Application of OOD Principles (SOLID)

Single Responsibility Principle

The Single Responsibility Principle states that a class should have only one responsibility and one reason to change. We applied this by separating functionalities across controller and model classes. For example, the **Application** class only stores the relevant applicant's attributes.

The **ApplicationController** class handles the business logic and implements the required functions. Thus, changes to data affect only **Application**, and changes to logic affect only **ApplicationController**.

Open-Closed Principle

The Open-Closed Principle states that entities should be open for extension but closed for modification. In our BTO Management System, the **User** abstract class defines shared functions, while specific roles like **Applicant**, **HDBOfficer** and **HDBManager** extend their own functions. This allows the system to be extended by creating subclasses instead of modifying the **User** class itself.

Liskov Substitution Principle

The Liskov Substitution Principle states that objects of a superclass should be replaceable with objects of its subclasses without breaking the program. For instance, **LoginView** handles the login authentication for the **User** superclass. Upon login, it returns a User object which points to any instance of the three subclasses: **Applicant**, **HDBOfficer** or **HDBManager**.

Polymorphism allows these role specific objects to be used interchangeably without causing error to the program.

Interface Segregation Principle

The Interface Segregation Principle specifies that specific interfaces are better than one general purpose interface, meaning that larger interfaces should be split into smaller ones. For instance, the `ISearchable` interface defines a single method that can be inherited by any class that supports filtering. Classes like Application, Project and Enquiry can implement `ISearchable` if needed.

Dependency Injection Principle

The Dependency Injection Principle states that high-level modules should not depend on low-level modules, but both should depend on abstractions. In our system `ManagerView` (high-level) depends on `ManagerController` to perform functions like project creation rather than low level modules like `Project` and `HDBManager`. This separation maintains clear abstraction layers.

Chapter 4. Implementation (Java)

4.1 Tools Used:

Java 17/23, IDE: IntelliJ / Visual Studios Code, Version control: GitHub

4.2 Sample Code Snippets:

- Encapsulation

```
You, 2 weeks ago | 2 authors (Javier Tay and one other)
public class HDBOfficer extends Applicant{
    private String assignedProjectName = null;
    private final Map<String, RegistrationStatus> registrationStatus;
```

```
public String getAssignedProject() {
    return assignedProjectName;
}

public void assignToProject(String projectName) {
    this.assignedProjectName = projectName;
}
```

We declare `assignedProjectName` as a private variable and are only able to access it with the getter and setters. This protects the internal state of the attribute

- Inheritance

```
10, 5 weeks ago | 1 author (Javier Tay)
public class Applicant extends User {
    public Applicant(String name, String nric, String password, int age, String maritalStatus) {
        super(name, nric, password, age, maritalStatus);
    }
}
```

The code snippet shown above is an example of inheritance where we extended from an abstract user class, reducing code duplication

- Polymorphism

In our user class, we defined a `getRole()` method for identifying the user types and in each of the 3 classes that inherits from it, we override the method to return their respective roles instead.

```
public abstract String getRole();
```

```
@Override  
public String getRole() {  
    return "Applicant";  
}
```

```
@Override  
public String getRole() {  
    return "HDBManager";  
}
```

- Interface use

```
public class Project implements Searchable{  
    private String name;
```

```
public interface Searchable {  
    boolean matches(Filter filter);  
}
```

We implemented a searchable interface to aid with our filtering of projects/applications. This also showed the concept of abstraction.

- Error handling

With the help of a utility class due to the number of menus we have, we are able to handle any non-integer inputs without repeating the same code for the different views.

```
while (true) {  
    try {  
        System.out.print(s:"Select an option: ");  
        return Integer.parseInt(scanner.nextLine().trim());  
    } catch (NumberFormatException e) {  
        System.out.println(x:"Invalid input. Please enter a number.");  
    }  
}
```

Chapter 5: Testing

5.1 Test Strategy

Our team adopted a unit testing strategy to test the system using JUNIT5. We tested the main classes through rigorous testing of individual processes to ensure the workflow is correct. We used multiple prints to debug and fix our code. Manual Testing was conducted for complex operations.

5.2 Test Case Table

(Refer to appendix B)

Chapter 6. Documentation

6.1 Javadoc

Java docs are included in both the submission folder and within the project folder as well.

All public and private classes/methods documented with Javadoc

HTML Javadoc files generated and included

6.2 Developer Guide

Step 1: Install JDK: Download and install JDK 17 or newer.

Step 2: Use an IDE (e.g., IntelliJ, Eclipse) for easier navigation. Open the IDE and select the project's root folder.

Step 3: In your IDE, paste the GitHub repository URL, choose a local folder, and click "Clone"

Step 4: Explore Project Structure:

- MainApp.java: Entry point to run the system
- controller: Manages application logic
- data: Handles loading/saving of user and project data.
- model: Defines core entities.
- view: Manages displays and menus.
- util: Provides shared helper tools.

Step 5: Run the System: Open MainApp.java under the src folder and run it. Follow the login prompts to start using the system.

Chapter 7. Reflection & Challenges

• What went well

From the start, we divided responsibilities based on individual strengths - some focused on UI/UX design, others on backend logic, and some on testing and documentation. This clear role assignment helped us work efficiently in parallel and reduced redundant efforts. Communication through group chats and meetings ensured everyone was aligned. We also successfully maintained clean, modular code using object-oriented programming principles and followed SOLID design guidelines. This made our codebase easier to maintain, extend, and debug. It also ensured different modules (like application management, project filtering, and enquiry handling) integrated smoothly.

• What could be improved

Most of our time was spent on implementing core features, which left less time for stress testing and polishing edge cases. Allocating more buffer time for testing and feedback iterations would improve overall stability and user experience.

• Individual contributions

Javier - Code implementation

Jessica & Audrey - UML Class and Sequence Diagrams

Perrin & Qi Ming- Unit tests + Testing

- Lessons learned about OODP

By implementing features like role-based menus or enquiry handling showed us the strength of polymorphism and overriding methods like `getRole()` or using interfaces for controllers, we can write flexible code that treats different user types uniformly where possible, while still allowing custom behaviors. It has also taught us to think ahead. By designing interfaces like `Searchable` and `Filterable`, we made it easy to add new features (like project and application filtering) without rewriting existing logic. This highlighted the value of designing for scalability.

Chapter 8. Appendix

- GitHub link (if any) <https://github.com/javiertay/SC2002>
- Link to UML Class Diagram [here](#)
- References (if any external libraries/tools used)
<https://poi.apache.org/download.html#POI-5.4.0> as we decided to keep the excel format so we can access multiple sheets in 1 file, we had to use an external library like apache poi to help read & write from the excel file
- Appendix A: (UML Sequence Diagram)

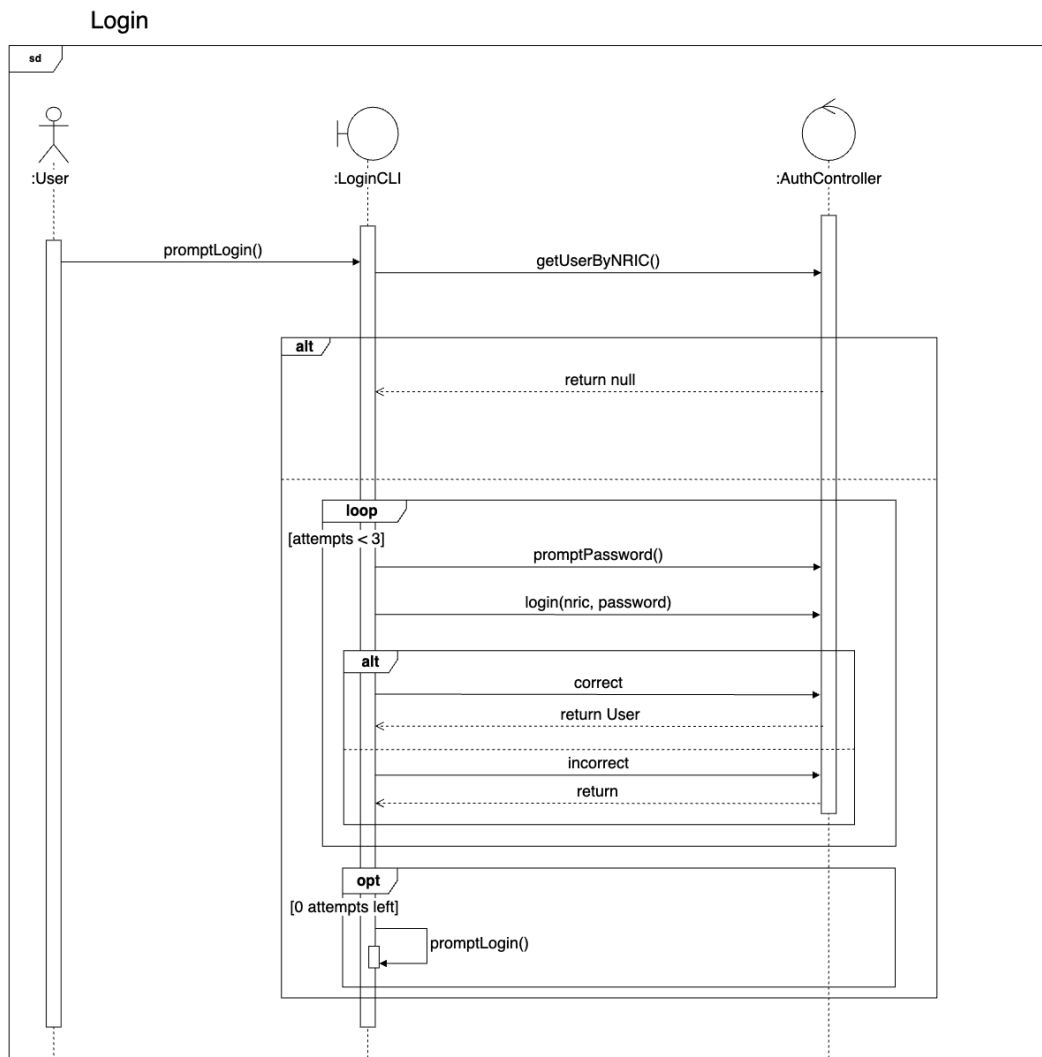


Figure 1

Officer applying for BTO

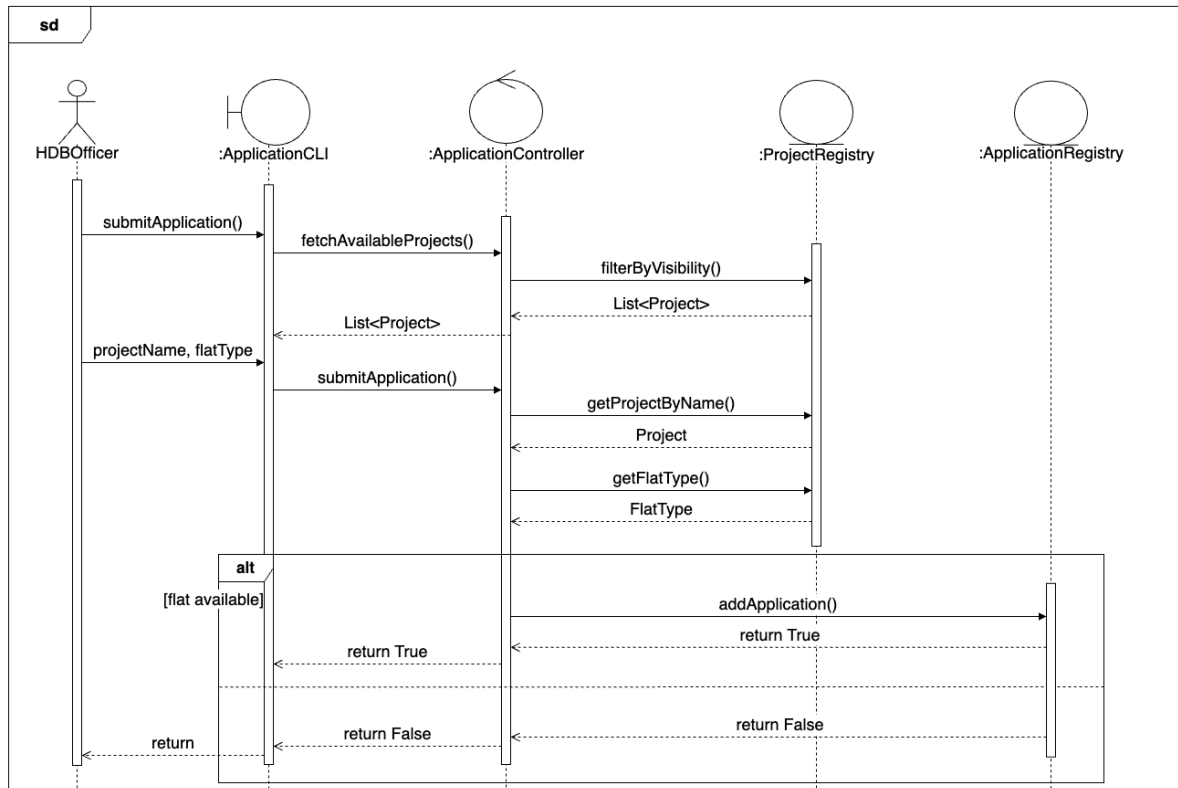


Figure 2

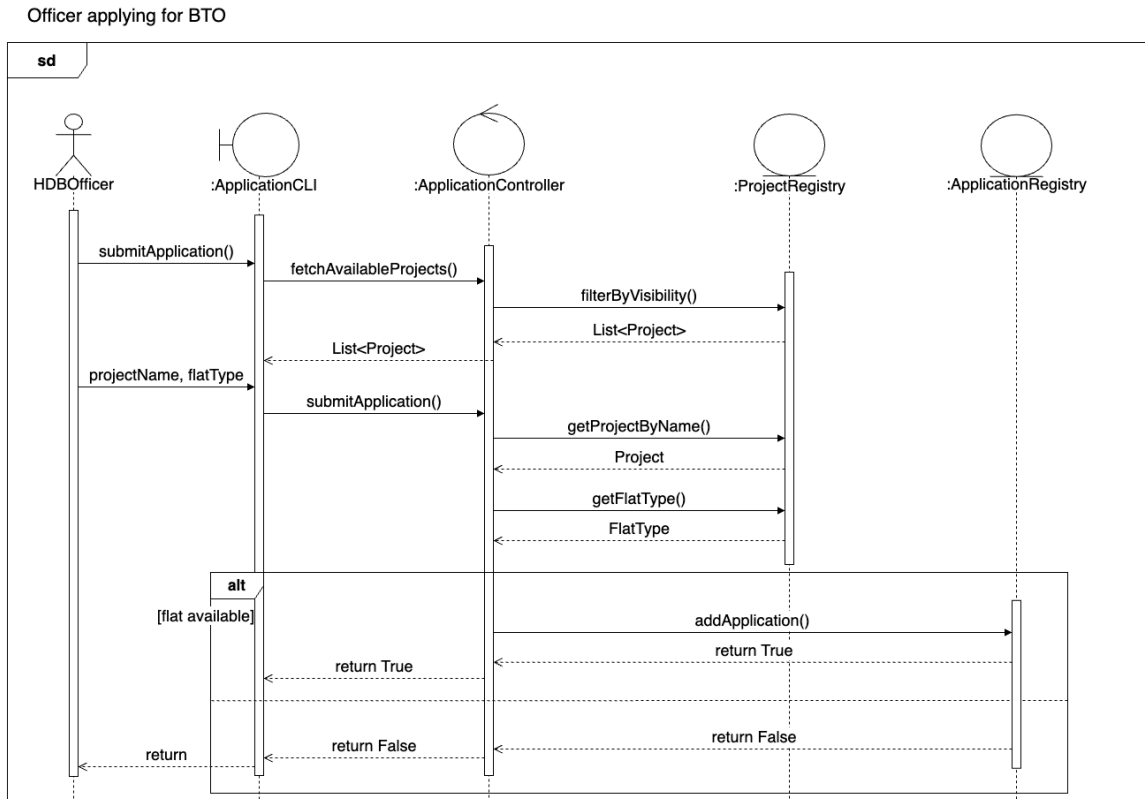


Figure 3

- Appendix B (Test Cases)

No	Test Cases	Expected Behaviour	Test Methods
1	Valid User Login	User should be able to access their dashboard based on their roles	login_validCredentials_return sUserAndPrintsSuccess()
2	Invalid NRIC Format	User receives a notification about incorrect NRIC format	login_invalidNricFormat_thr owsNullPointerException()
3	Incorrect Password	System should deny access and alert the user to incorrect password	login_wrongPassword_return sNullAndPrintsIncorrectPass word()
4	Password Change Functionality	System updates password, prompt re login and allows login with new credentials	changePassword_nullUser_re turnsFalseAndPrintsError(), changePassword_validUser_c hangesPasswordAndPrintsSu ccess(), prompt-flow tests in PromptPasswordChangeTests
5	Project Visibility Based on	Projects are visible to users based on	getAllAvailableProjects_filter

	User Group and Toggle	their age, marital status and the visibility setting	sCorrectly()
6	Project Application	Users can only apply for projects relevant to their group or when visibility is off	submitApplication_and_view ApplicationStatus_outputsTable()
7	Viewing Application Status after Visibility Toggle Off	Applicants continue to have access to their application details regardless of project visibility.	viewApplicationStatus_after VisibilityToggleOff_showsClosed()
8	Single Flat Booking per Successful Application	System allows booking one flat and restricts further bookings	testSingleFlatBooking_and_preventMultipleBookings()
9	Applicant's enquiries management	Enquiries can be successfully submitted, displayed, modified, and removed.	submitAndGetEnquiriesByUser(), updateAndDeleteEnquiry()
10	HDB Officer Registration Eligibility	System allows registration only under compliant conditions	testReqToHandleProject_Success(), testReqToHandleProject_AlreadyPending(), testReqToHandleProject_FailsWhenAppliedAsApplicant(), testReqToHandleProject_FailsWhenActiveRegistrationOnOtherProject()
11	HDB Officer Registration Status	Officers can view pending or approved status updates on their profiles.	testViewRegistrationStatus()
12	Project Detail Access for HDB Officer	Officers can always access full project details, even when visibility is turned off.	testViewAssignedProjectDetails(), testViewAssignedProjectDetails_WhenVisibilityOff()
13	Restriction on Editing Project Details	Edit functionality is disabled or absent for HDB Officers.	testOfficerCannotEditProjectDetails()
14	Response to Project Enquiries	Officers & Managers can access and respond to enquiries efficiently.	replyToEnquiry_and_invalidCases(), testManagerCanReplyToEnquiry_ForManagedProject(), testManagerCannotReplyIfNotManaging()

15	Flat Selection and Booking Management	Officers retrieve the correct application, update flat availability accurately, and correctly log booking details in the applicant's profile.	testSingleFlatBooking_and_preventMultipleBookings()
16	Receipt Generation for Flat Booking	Accurate and complete receipts are generated for each successful booking	testGenerateReceipt_forBookedApplicant_printsReceipt()
17	Create, Edit, and Delete BTO Project Listings	Managers should be able to add new projects, modify existing project details, and remove projects from the system	testCreateProject_Success(), testCreateProject_DuplicateName(), testEditProjectDetails_AllFields(), testDeleteProject_Success(), testDeleteProject_NotManaging()
18	Single Project Management per Application Period	System prevents assignment of more than one project to a manager within the same application dates.	testCreateProject_FailsWhenOverlapDates()
19	Toggle Project Visibility	Changes in visibility should be reflected accurately in the project list visible to applicants	covered by Scenario 7 (viewApplicationStatus_afterVisibilityToggleOff_showsClosed())
20	View All and Filtered Project Listings	Managers should see all projects and be able to apply filters to narrow down to their own projects.	covered by Scenario 5 (getAllAvailableProjects_filtersCorrectly())
21	Manage HDB Officer Registrations	Managers handle officer registrations effectively, with system updates reflecting changes accurately.	Unit Testing unavailable. Refer to the screenshot below.
22	Approve or Reject BTO Applications and Withdrawals	Approvals and rejections are processed correctly, with system updates to reflect the decision	testApproveRejectWithdrawal_SuccessAndFailure()
23	Generate and Filter Reports	Accurate report generation with options to filter by various categories.	Unit Testing unavailable. Refer to the screenshot below.

```

Page 1 of 1
| Name      | NRIC       | Project Name | Status    |
|-----|-----|-----|-----|
| Daniel    | T2109876H | Acacia Breeze | Pending   |
Enter Officer NRIC to process (or 'back' to return): T2109876H
Approve or Reject? (A/R): A
Officer Daniel application has been approved.

```

(Scenario 21 - Manager Screen - Officer Approved by Manager)

```

Welcome back Daniel!
- You have not applied for any BT0 projects yet.
- You have not made any enquiries for any BT0 projects.
- 2 Projects currently open. 4 more upcoming soon!

- Your Assigned Project: Acacia Breeze
- 0 Applications pending booking
- 1 Enquiries awaiting reply

=== HDB Officer Menu ===
1. Manage HDB Applications
2. Register for a Project
3. View Registration Status
4. View Assigned Project Details
5. Flat Selection (Assign Flat)
6. Manage Enquiries
7. Change Password
0. Logout
Select an option: 3

Your Registration Status:
- Project: Acacia Breeze | Status: APPROVED

```

(Scenario 21 - Officer Screen 1 - Officer Approved by Manager)

```
=== HDB Officer Menu ===
1. Manage HDB Applications
2. Register for a Project
3. View Registration Status
4. View Assigned Project Details
5. Flat Selection (Assign Flat)
6. Manage Enquiries
7. Change Password
0. Logout
Select an option: 4

=== Project Details ===
Project Name: Acacia Breeze
Neighborhood: Yishun
Application Period: 2025-03-24 to 2026-03-20
Visibility: Visible
Flat Types:
- 2-Room ($350000): 2 units available
- 3-Room ($450000): 3 units available
```

(Scenario 21 - Officer Screen 2 - Officer Approved by Manager)

```

Page 1 of 1
| Name      | NRIC      | Project Name | Status |
|-----|-----|-----|-----|
| Daniel | T2109876H | Acacia Breeze | Pending |
Enter Officer NRIC to process (or 'back' to return): T2109876H
Approve or Reject? (A/R): R
Officer Daniel application has been rejected.

Page 1 of 0
| Name | NRIC | Project Name | Status |
|-----|-----|-----|-----|

[N]ext, [P]rev, [Q]uit:

```

(Scenario 21 - Manager Screen - Officer Rejected by Manager)

Welcome back Daniel!

- You have not applied for any BTO projects yet.
- You have not made any enquiries for any BTO projects.
- 2 Projects currently open. 4 more upcoming soon!

- You are not currently assigned to any project.

=== HDB Officer Menu ===

1. Manage HDB Applications
2. Register for a Project
3. View Registration Status
4. View Assigned Project Details (Unavailable)
5. Flat Selection (Unavailable)
6. Manage Enquiries
7. Change Password
0. Logout

Select an option: 3

Your Registration Status:

- Project: Acacia Breeze | Status: REJECTED

(Scenario 21 - Officer Screen - Officer Rejected by Manager)

```

Current Application Filters:
- Marital Status: -
- Flat Type: -
- Project Name: -
- Age Range: - to -
- Status: -

No filters applied. Would you like to add filters? (Y/N): N

=== Filtered Application Report ===

Page 1 of 2
| Name | NRIC | Age | Marital Status | Flat Type | Project | Status |
|-----|-----|-----|-----|-----|-----|-----|
| Bob | S7894523E | 30 | Married | password | Twin Waterfall | BOOKED |
| Grace | S9876543C | 37 | Married | 3-Room | idk | PENDING |
| Grace | S9876543C | 37 | Married | 3-Room | idk | UNSUCCESSFUL |
| Grace | S9876543C | 37 | Married | 3-Room | Twin Waterfall | SUCCESSFUL |
| James | T2345678D | 30 | Married | 3-Room | hi | SUCCESSFUL |

[N]ext, [P]rev, [Q]uit: |

```

(Scenario 23 - Reported Generated without filter)

```

=== Filter Applications by: ===
Filter by Marital Status (e.g., Married, Single leave blank to skip): Single
Filter by Flat Type (e.g., 2-Room, 3-Room leave blank to skip):
Filter by Project Name (comma-separated, leave blank to skip):
Filter by Minimum Age (leave blank to skip): 21
Filter by Maximum Age (leave blank to skip): 90
Statuses: PENDING, SUCCESSFUL, BOOKED, UNSUCCESSFUL, WITHDRAWN
Filter by Application Status (leave blank to skip):
Application filters updated.

=== Filtered Application Report ===

Page 1 of 1
| Name | NRIC | Age | Marital Status | Flat Type | Project | Status |
|-----|-----|-----|-----|-----|-----|-----|
| John | S1234567A | 35 | Single | 2-Room | Twin Waterfall | PENDING |

```

(Scenario 23 - Reported Generated with filter)

Additional Test Cases

No	Description & Intended Behaviour	Test Method
1	Submit to non-existent project: should return FALSE and print “Project not found.”	ApplicationController.submitApplication_nonexistentProject_fails()
2	Submit with invalid flat type: should return FALSE and print “Flat type does not exist in this project.”	ApplicationController.submitApplication_invalidFlatType_fails()
3	Duplicate application submission: second submission should return FALSE and print “You already have an active application. Withdraw or wait for rejection to reapply.”	ApplicationController.submitApplication_duplicateApplication_fails()
4	Apply when project visibility is off: should return FALSE and print “This project is no longer visible to applicants.”	ApplicationController.submitApplication_visibilityOff_fails()
5	Ineligible applicant (under-age or wrong status): should return FALSE and print “You are not eligible to apply for this flat type.”	ApplicationController.submitApplication_ineligibleApplicant_fails()
6	Delete enquiry with wrong NRIC: should return FALSE and leave the enquiry in the registry (no deletion).	EnquiryController.deleteEnquiry_wrongSender_returnsFalse()
7	Get all enquiries for a project: should return a list containing only that project’s enquiries.	EnquiryController.getProjectEnquiries_returnsAllForProject()
8	Update flat units for missing flat type: should throw NullPointerException.	ManagerController.updateFlatUnits_nonexistentFlatType_throwsNPE()
9	Toggle visibility on non-existent project: should print “Project not found.”	ManagerController.toggleVisibility_nonexistentProject_printsNotFound()
10	Assign flat when officer not assigned/approved: should print “No successful application found for this applicant in your project.”	OfficerController.assignFlat_whenNotAssignedOrNotApproved_fails()