**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** |
| --- | --- | --- | --- |
| JOHANNAS BIN JUMARI | Computer Science | FCSI | 24/04/2025 |
| JOSHUA TAN YING JIE | Computer Science | FCSI | 24/04/2025 |
| SILVIANA | Computer Science | FCSI | 24/04/2025 |
| WANG HAO STEFANO | Computer Science | FCSI | 24/04/25 |

Important notes:

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

# TABLE OF CONTENTS

[**TABLE OF CONTENTS 2**](#_o3j92zq8bc39)

[**1. Requirement Analysis & Feature Selection 3**](#_exz3gpzd89ei)

[1.1. Understanding the Problem and Requirements 3](#_yd2p1yboffsb)

[1.2. Features and Scope 4](#_qdr8y3e4fcn)

[**2. System Architecture & Structural Planning 4**](#_kfwbtlbd8og)

[2.1. System Structure Planning 4](#_4x62f4mcz7my)

[2.2. Reflection on Design Trade-offs 5](#_4igsyusr4b3p)

[**3. Object Oriented Design 6**](#_zbwkkwj2og2z)

[3.1. Class Diagram 6](#_fmpv0qzeqxgo)

[3.2. Sequence Diagrams 7](#_azcrmc3rl129)

[“Why did we choose this scenario to represent? What does it help us validate or communicate about our design?” 7](#_otfoyy4jw2hf)

[3.3. Application of SOLID Principles 7](#_wq7t16v2qhj3)

[3.3.1. Single Responsibility Principle 7](#_ag95kcno48xy)

[3.3.2. Open-Closed Principle 8](#_a6vkzgpz3d6j)

[3.3.3. Liskov Substitution Principle 8](#_glmyiz3mjtpz)

[3.3.4. Interface Segregation Principle 8](#_biasj5niulmo)

[3.3.5. Dependency Injection Principle 8](#_a61oxoaxw9cc)

[**4. Implementation 9**](#_vrzvs7w09vr3)

[4.1. Tools Used 9](#_6j1wfkhwl1bg)

[4.2. Sample Code Snippets 9](#_e5fomrsg7hmh)

[Encapsulation 9](#_v23h1euxfcke)

[Inheritance 9](#_7uxot1ddmmqv)

[Polymorphism 9](#_ua42agg0sv1g)

[- Interface use 10](#_acbq9iwb66r2)

[- Error handling 10](#_n3fwt1k7lei6)

[**5. Testing 11**](#_bsenk3q7ysj7)

[5.1. Test Strategy 11](#_89c30u7vjbjw)

[5.2. Test Case Table 11](#_2iay3vmqvfwz)

[**6. Reflection & Challenges 12**](#_mshflowhdw10)

[**7. Appendix 13**](#_k2emvnlsjd2h)

[**7.1. Test cases 13**](#_rknbuqr7jamt)

[**7.2. References 13**](#_l4nknpvl9160)

# Requirement Analysis & Feature Selection

# Understanding the Problem and Requirements

**Main Problem Domain**

To start development, our group closely read the assignment brief and the system description to identify the main problem domain. We extracted the key user roles and also their interaction with the system. From this, we are able to extract the main problem with their implicit requirements and some additional requirements we inferred to aid our project development.

**Explicit Requirements**

The assignment brief documents the explicit requirements expected of the system. More will be detailed in section 1.2, Features and Scope.

**Implicit Requirements**

Besides the clearly stated explicit requirements, our group also inferred some important requirements which we felt could aid us in our development.

1. Role-based Requirements: Each user group should have access to separate, well-defined role-appropriate interfaces
2. Data Security and Validation: Proper validation of NRIC, restricted access to project function based on roles and secure handling of password are expected.
3. Persistent State: Filter settings should persist when navigating through menus and data should be saved across sessions.
4. Reliability and Simplicity: As a command-line-interface(CLI) application, the interface should be intuitive and user-friendly, providing clear prompts, error and message display.

**Ambiguities and Missing Parts**

While reading we noticed a few ambiguities that needed further thinking/analysis.

1. Project visibility - What should happen after the project is closed, it was not said whether applicants can view the project or not. We resolved this by turning off the visibility for new applicants but it will still be available for those applicants who already applied for it.
2. Enquiry Specifications - There were no requirements in what enquiry should consist of. It was also unclear whether the enquiry can be edited or deleted after it was replied by the Officer/Manager. We decided to deny applicants from editing/deleting the enquiry after it was replied to, so that it is available for other users to view.

# Features and Scope

We grouped features into three categories: core, optional, and excluded. To ensure core features are implemented and to be able to map out our plans in implementing the BTO system. We also added priority to ensure we know what features need to be focused on first based on the importance of the feature in the main project and how feasible it is.

| Features | Feat. Categories | Priority (High/Low) |
| --- | --- | --- |
| Login | Core features | High |
| Change Password | Core features | High |
| Password Hashing | Optional or bonus features | Low |
| View and Filter Projects | Core features | High |
| Project Applications | Core features | High |
| Booking Flats | Core features | High |
| Enquiries | Core features | High |
| Project Management | Core features | High |
| Applicant Reports | Core features | High |
| Project Registrations | Core features | High |
| Sorting Project | Optional or bonus features | Low |

# System Architecture & Structural Planning

# System Structure Planning

We broke down the system by using the Model-View-Controller-Service design pattern. We felt that this design pattern was the best in terms of function flexibility and modularity. The model is responsible for defining data structures and minor logic for entities such as users, projects, applications bookings etc.

The view represents the user interface, which in our case, is a command-line-interface(CLI). Views are tailored to each different user role, providing prompts, formatted outputs along with success and error messages when needed. They also collect user input.

The controller is the “middle man” between the service and view layers. It receives user input from the view and processes it through the appropriate services. Controllers are also specific to the user roles and enforce role-specific access rules.

The service is responsible for handling the business logic of the application. Services perform operations such as booking a flat, getting all the unreplied enquiries for manager to reply to and also generating receipts. This separation helps keep controllers lightweight and helps ensure that the business logic is centralised and consistent.

We then did a brief class diagram to make sure all of us were on the same page with how the system was going to flow. By doing this, we were able to align our understanding of class responsibilities and interactions with the Model-View-Controller-Service design pattern. This also helped each member as they were able to more confidently apply and develop their assigned modules.

# Reflection on Design Trade-offs

Our group considered 2 options:

1. Simplicity by implementing the View into the Controller, overall reducing the total amount of classes required and making it easier to understand and maintain. Also reducing the time to build the project and move to testing.
2. Extensibility by separating every function to different classes, this would help us by ensuring Open Closed Principle (OCP), to be able to implement more additional features without the need to change the initial code as it will not affect it.

Ultimately, We chose option 2. It allows us to improve team collaboration as we can work independently across different parts of the system without any code conflicts. Isolated classes makes it easier for debugging as the separation allows us to conduct tests for logic-heavy services. Enhancing of specific functionalities can also be done without breaking the overall flow.

# Object Oriented Design

# Class Diagram

To identify the main classes, we took notes of the nouns being used in the assignment brief (eg: User, Project, Application, Enquiry, Booking etc). We also took notes of user roles which implied specific permission-based flows and interfaces. As stated earlier our group opted for the model-view-controller-service design pattern as we felt it was the best in providing us flexibility and modularity, especially for this project which consisted of various complex business logic and multiple user roles.

Below are some key responsibilities of each class.

Some important responsibilities of key classes are as shown below:

1. ApplicantController : responsible for handling input/output with the Applicant, routes actions (eg : apply for project/add new enquiry) to the service layer, displays the appropriate view and more.
2. OfficerController : responsible for facilitating an officer’s registration to handle a specific project, allows officer to view and respond to enquiries, calls booking services to process successful applications and more.
3. ManagerController : Allows managers to create/edit/delete projects, approves/rejects applications and officer registrations, manages visibility toggles and more.

To identify relationships, we often asked ourselves key design questions such as “should this be an attribute or its own class” and “Is this an is-a or a has-a relationship”.

The first trade-off we considered included simplicity vs flexibility. We could have combined various views into the controller to simplify the codebase but we opted to divide them up into role-specific controllers and views to preserve the clear responsibilities and support the Open/Closed principle.

Secondly, modularity vs implementation time. Designing a modular system required more upfront time to plan and communicate between group members. However, it allowed different group members to work in parallel and independently across various different parts which saved a lot of time in the end.

The finalised class diagram will be in the zip file included with the source code due to the size

# Sequence Diagrams

## *“Why did we choose this scenario to represent? What does it help us validate or communicate about our design?”*

The scenario we chose to represent was where the officer attempts to apply for a BTO project or register to handle a project because it helps us validate our rule where officers cannot apply for a BTO project which they are assigned to manage and vice versa. This sequence is essential in demonstrating the robustness of our code and shows how our system enforces role integrity and prevents conflict of interest.

The finalised sequence diagram will be in the zip file included with the source code due to the size.

# Application of SOLID Principles

## Single Responsibility Principle

To adhere to the Single Responsibility Principle, the BTO Management System was organised into several different packages, each dedicated to a specific responsibility. The packages include Datastore, Initialiser, IO, Model, Parser, Service, Session, Util, and View.

1. Datastore

The datastore package includes the ***‘DataStore’*** class, which serves as the central in-memory data repository for the entire application. Used to hold a collection of all the core entities in the system, and allow for easy addition, retrieval and modification of data.

1. Service

Service package contains classes that handle the business logic of the application. These classes act as an intermediary between the controllers and the ***‘DataStore’***. For example, the ***‘RegistrationService’*** contains the logic used to manage officer registrations for the BTO projects, whereas the ***‘EnquiryService’*** class contains the logic for any enquiry related operations.

1. View

View package contains classes that are responsible for managing the presentation layer of the application. It handles all user interactions, including displaying information to the user and collecting input. For example, ‘**ManagerView’** provides the user interface for managers and ‘**ApplicantView’** provides the user interface for an applicant.

## 

## Open-Closed Principle

To ensure the extensibility of the BTO Management system, the Open-Closed principle was implemented in our view hierarchy. We built an abstract **‘*BaseView’*** class, which was extended by concrete, domain specific views such as ***‘ProjectView’***, ***‘EnquiryView’***, ***‘AccountView’***, etc. The core utilities in **‘*BaseView’***, such as **showMessage()**, **showError()** are stable and reused as-is. We then bundled the domain specific views into role specific view bundles such as ***‘ApplicantViews’*** and ***‘OfficerViews’***. This means that we can extend the role based views without touching the rest of the system. Our controllers receive a bundled object and access exactly what they need without knowing internal wiring.

## Liskov Substitution Principle

Our BTO Management System ensures that the subclasses can be passed in place of the base classes without breaking the code by defining a ***IBaseController***, which is extended by role-specific controllers such as ***ApplicantController***, ***OfficerController***, and ***ManagerController***. A clear example of this is exemplified by our implementation of ***MainMenuController.handleLogin()*** function where we retrieve the user role upon successful login, and dynamically generate a controller using ***controllerFactory.createControllerForUser(loggedIn)***. This factory returns an object that implements ***IBaseController***, and we invoke ***userController.start()*** without needing to know the actual class behind it. This demonstrates LSP: each specific controller class can be substituted in place of the ***IBaseController*** without breaking functionality or requiring special-case handling. As a result, our login and routing logic remains decoupled from role-specific implementations, promoting extensibility and stability.

## Interface Segregation Principle

To adhere to the Interface Segregation principle, we tried to avoid classes implementing interfaces that they do not use to ensure that the classes are not burdened with unnecessary methods. For example, the `***IApplicationPermission***` interface defines methods for managing application-related permissions. Different roles, such as `***Applicant***`, `***Manager***`, and `***Officer***`, have their own implementations of this interface, tailored to their specific needs.

## Dependency Injection Principle

Our group implemented Dependency Injection (DI) throughout the application to promote loose coupling between components. Instead of instantiating dependencies directly within controllers or views, we injected shared dependencies, such as ***‘Prompter’***, ***‘ProjectService’***, and ***‘ReportService’***, through constructors. For example, ***‘ProjectViewerController’*** receives its required ***‘SharedView’***, ***‘ProjectView’***, and ***‘ProjectService’*** via its constructor, allowing it to focus solely on coordinating logic rather than managing object creation. This design not only improves testability and modularity, but also aligns with the Dependency Inversion Principle, as high-level components depend on abstractions or externally provided instances rather than concrete implementations.

# Implementation

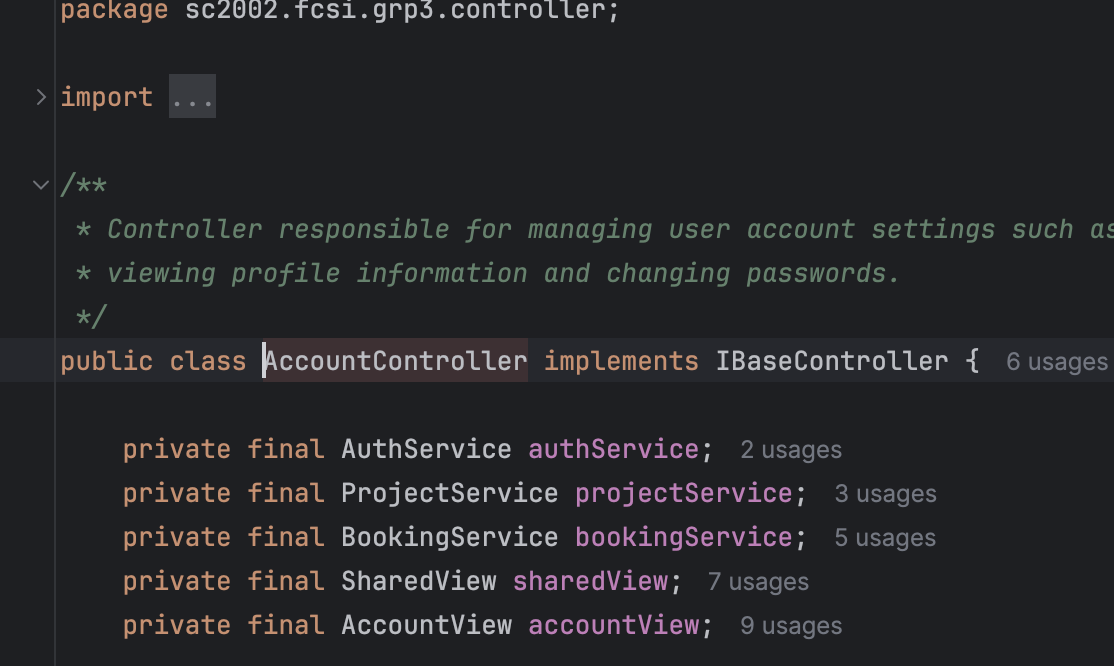
# Tools Used

* Java 23
* IDE: Intellij / Visual Studio Code
* Version Control: GitHub

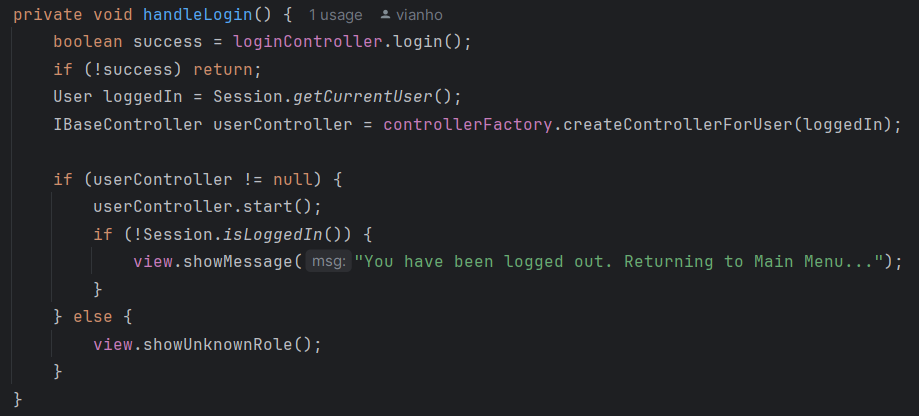
# Sample Code Snippets

## Encapsulation

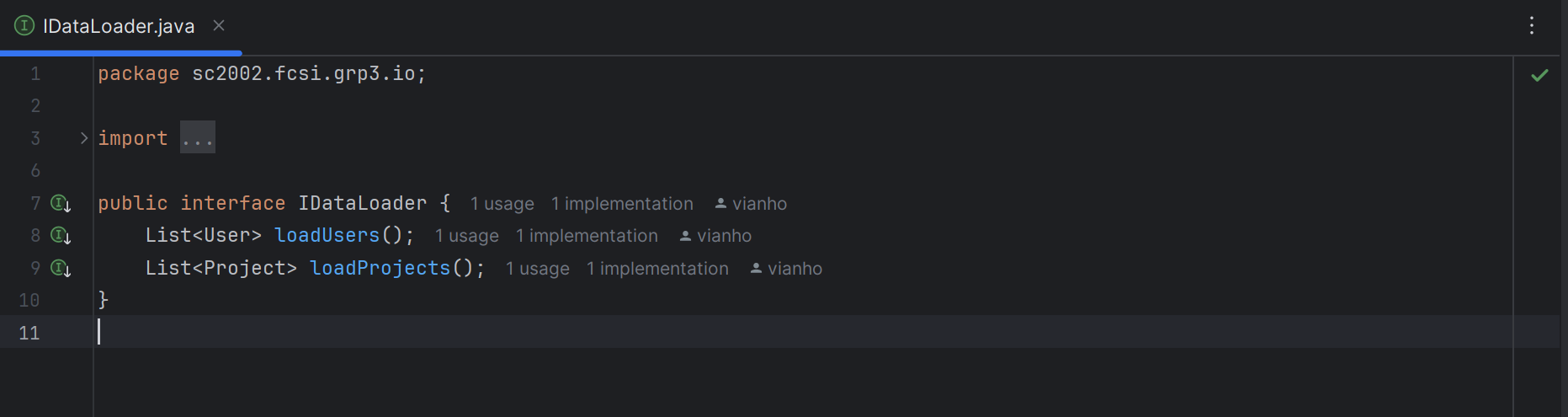
## Inheritance



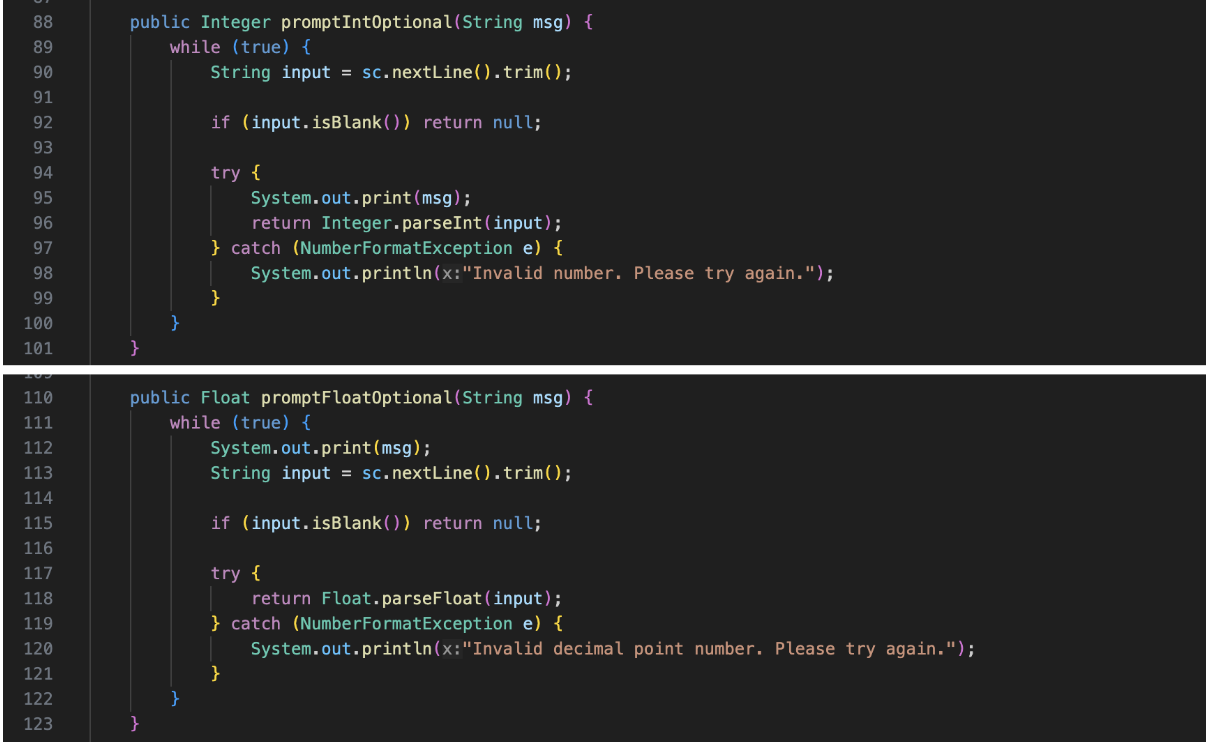
## Polymorphism



## Interface use



## Error handling



# Testing

# Test Strategy

Due to time constraints, we used manual functional testing, making sure we adhered to the test case table.

# Test Case Table

In the appendix.

# Reflection & Challenges

**What went well?**

Overall our team collaborated effectively by splitting responsibilities well based on each member's strengths. Implementing the MVCS design pattern helped us keep our code modular, flexible and scalable. Additionally, explicitly visualising our class relationships and sequence flows made it easier to validate our design decisions early on.

**What could be improved?**

We initially faced challenges with the role separation as we felt it was very confusing handling user types with shared responsibilities (Applicant and Officer both can apply for BTO / Officer and manager being able to reply to enquiries but viewing it differently). Also, communicating more about the SOLID principles would have improved consistency and reduced some refactoring later on.

**Teammate Contributions**

Johannas - EnquiryController, DataLoader, Parsers

Joshua - OfficerController, Booking, Registration

Silviana - ApplicantControllers, DataStore, ConfigLoader, Applicant-Subcontrollers, Authentication, Initializers

Stefano - ManagerController, ProjectService

# Appendix

Github Link : <https://github.com/vianho/sc2002_fcsi_group3>

<https://refactoring.guru/design-patterns/singleton>

<https://refactoring.guru/design-patterns/factory-method>

## Authentication

| **Test Description** | **Expected Result** | **Pass/Fail** |
| --- | --- | --- |
| **[SUCCESSFUL LOGIN]**  (a) User enters correct NRIC and password | When (a) occurs, user should be logged in and shown menu according to their role | PASS |
| **[FAIL TO LOGIN]**  (a) User enters the wrong/invalid NRIC  (b) Enter wrong password for correct NRIC | When (a) and (b) occur, an error message is displayed and it returns to the main page | PASS |
| **[CHANGE PASSWORD]**  (a) Upon successful login, the user changes their password by entering their current password, followed by their new password twice, ensuring it is of length 8, includes upper/lower case and a special character | When (a) occurs, a successful message prints. | PASS |

* + 1. HDB Manager Functionality

| **Test Description** | **Expected Result** | **Pass/Fail** |
| --- | --- | --- |
| **[CREATE BTO PROJECT]**  (a) Creates Project with valid details  (b) Creates Project with invalid details, specifically overlapping project opening and closing dates with other projects managed by manager | (a) Project Created successfully  (b) Project Created unsuccessfully | PASS |
| **[VIEW ALL PROJECTS]**  (a) There are no existing Projects  (b) There are existing Projects | (a) No project available displayed  (b) ALL projects are displayed in the correct format | PASS |
| **[VIEW MANAGED PROJECTS]**  (a) Manager does not have any managed projects  (b) Manager does have managed projects | (a) No project available displayed  (b) Projects managed by manager are displayed in the correct format | PASS |
| **[EDIT PROJECT]**  **(Includes toggling visibility)**  (a) Accesses their managed project and successfully enters all fields  (b) Accesses their managed project and edits the dates which causes project with overlapping dates  (c) Attempts to access project managed by other managers | (a) Successfully updates project and displays success message  (b) Fails to update project displays error message  (c) Fails to update project, displays error message | PASS |
| **[DELETE PROJECT]**  (a) Attempts to delete managed project  (b) Attempts to delete other manager’s project | (a) Deletes project successfully  (b) Cannot delete project because of unauthorised access, error message | PASS |
| **[VIEW PROJECT DETAILS]**  (a) Attempts to view existing project  (b) Views invalid project | (a) Successfully views | PASS |
| **[ENQUIRIES]**  (a)Attempts to view all enquiries of all projects  (b) Attempts to reply to specific enquiry of project they are handling | 1. Successfully views 2. Successfully able to reply to enquiry | PASS |
| **[APPROVE APPLICATION]**  (a) Attempts to approve an application they are handling  (b) Attempts to approve application for a project not assigned to them | (a) Application is marked approved and status updated  (b) Error or "Not authorized" message | PASS |
| **[APPROVE REGISTRATION]**  (a) Attempts to approve new user registration  (b) Attempts to approve an already approved or invalid registration | (a) User registration status is updated to "approved"  (b) Error or appropriate warning message shown | PASS |
| **[APPROVE WITHDRAWAL REQUEST]**  (a) Approves a valid withdrawal request  (b) Attempts to approve a non-existent or already processed request | (a) Application status updated to "withdrawn"  (b) Error or appropriate message shown | PASS |
| **[GENERATE FILTERABLE REPORT]**  (a) Applies valid filters and generates report  (b) Applies invalid or conflicting filters | (a) Report generated with filtered data  (b) Warning message or empty result with no crash | PASS |

* + 1. HDB Applicant Functionality

| **[View Available Projects]**  (a)View projects that the applicant can apply for | (a)Only show project based on   * Visibility On * Single or Married * Currently Open | PASS |
| --- | --- | --- |
| **[Create Filters and Sorting for Projects]**  (a) Access filter and sorting menu  (b) Create optional filters based on Project attributes such as Neighbourhood, Application opening and closing date, price, and flat types  (c) Move to another page and come back to project viewer page | (a) Can view all projects before filtering  (b) Can view filtered projects based on created filter  (c) Filter and sorting persists after changing pages | PASS |
| **[Apply Project]**  (a)Able to apply for a project under certain conditions | (a)Can only apply for one project   * Singles(35>) for only 2 rooms * Married(21>) for any flat types | PASS |
| **[View Application Status]**  (a)View the status of their application status | (a)View the status of the project they applied for   * Visibility does not affect | PASS |
| **[Request Withdrawal]**  (a)Able to withdraw their application | (a)Can withdraw before or after an officer has booked the flat | PASS |
| **[Enquiries]**  (a) Able to view all enquiries  (b) Able to reply to enquiries for the project they are handling | (a)Can submit an enquiry and view/edit/delete their own enquiry if it is unreplied | PASS |

* + 1. HDB Officer Functionality

| **[APPLICANT FUNCTIONALITY]**  (a) Possesses all applicant capabilities | (a)Able to do anything an applicant can do | PASS |
| --- | --- | --- |
| **[JOIN A PROJECT]**  (a) Join a project as an officer | (a)Cannot join if officer applies for project as an applicant  (b)Not an officer of another project | PASS |
| **[VIEW REGISTRATION STATUS]**  (a)View status of their registration to join a project | (a)Shows their registration “Pending”/”Approved” by the manager | PASS |
| **[VIEW HANDLED PROJECT]**  (a)View details of the officer’s handled project | (a)Only shows details of the project handled | PASS |
| **[VIEW/REPLY ENQUIRIES]**  (a)View and reply to enquiries from the applicants | (a)View all enquiries  (b)Reply to any enquiry, records the date posted | PASS |
| **[FLAT BOOKING]**  (a) Book a flat for applicant | (a)Update number of remaining flats per type  (b)Retrieve applicant’s application using NRIC  (c)Update status from “Successful” to “Booked”  (d)Update applicant’s profile with chosen flat type | PASS |
| **[GENERATE RECEIPTS]**  (a) Generate a receipt for applicants after the booking is successful | Generate a receipt with the following details   * Name * NRIC * Age * Marital status * Flat Type * Project Details | PASS |