Project Title: BTO MANAGEMENT SYSTEM

Group Members:

Name	**Student ID**	**Lab Group and Signature**
Bhumi Gupta	U2422742A	FSCE, Bhumi/16 April 2025
Sridhar Abhinav	U2420140J	FSCE Abhinav/16 April 2025
Nikunj Vaghela	U2420717L	FSCE Nikunj/16 April 2025
Veda Ho Yong Qian	U2321709B	FSCE Veda/16 April 2025

Section 1: Requirement Analysis & Feature Selection

1.1 Understanding the Problem and Requirements

To identify the main problem domain, we closely reviewed the assignment brief and noticed frequent mentions of BTO applications, user roles (Applicants, Officers, Managers), and their distinct functions. We designed a role-based housing management system, where access and actions depend on user identity and application period. The explicit requirements were clearly stated: login via NRIC/password, CSV data persistence, application status tracking, officer registration/approval, and project visibility filtering. The use of CLI and the exclusion of databases and formats like JSON/XML were also emphasized. Rules like one application per user and eligibility based on age and marital status shaped our control logic.

We also inferred several implicit expectations, such as strong input validation, persistence across runs, and a strict status flow (e.g., Pending → Successful → Booked). We assumed that actions like booking would trigger updates like unit count changes and receipt generation, even if not always explicitly stated. Ambiguities arose around post-visibility behaviour and application withdrawals. We addressed these by prioritizing clarity and aligning with the FAQ—e.g., allowing users to view past applications even if visibility is off.

1.2 Deciding on Features and Scope

We grouped features into three categories: core, optional, and excluded. Our goal was to ensure that the core features demonstrated strong object-oriented principles without making the system overly complex, especially considering our team's familiarity with Java after only 10 weeks of learning.

Feature	Category
User role differentiation View available BTO projects (with filters) Project creation, editing, deletion Application submission and withdrawal Officer registration and approval Application status updates Flat booking via officer Enquiry submission and response Receipt generation (with filter) CSV data persistence Password change functionality Project filtering (e.g., by room type/location) Real-time visibility toggle for projects Auto-update of available flat units after booking	Core
GUI interface (not allowed per assignment spec) Real-time chat for enquiries (violates “no conversation” FAQ rule) Integration with actual Singpass authentication Multi-language support (not specified, hard to implement under time constraints)	Excluded

Section 2: System Architecture & Structural Planning

2.1 Planning the System Structure

Before we began implementing the BTO Management System, we focused on designing a clear and modular system architecture based on **object-oriented design principles**. Our first

step was to break down the system into **logical components**, using the Model-View-Controller (MVC) pattern:

- **Model**: All Entity classes (e.g., Applicant, Project, Application) represented the core business objects.
- **View**: Boundary classes (e.g., ApplicantMenu, HDBOfficerMenu) handled user input/output in the CLI.
- **Controller**: Control classes orchestrated the flow between boundary and service logic.
- **Service & Repository Layers**: Services encapsulated business logic, while repositories handled CSV data persistence.

To map user actions to the system, we listed key **use cases** for each role. For example, we created separate flows for how an **Applicant** would register, apply for a project, and request withdrawal, and how a **HDB Officer** would register, approve bookings, and generate receipts. These flows were then modelled in **early flowcharts** to visualize step-by-step processes before finalizing the UML class and sequence diagrams. This helped us ensure that each use case aligned with our class responsibilities, reinforcing **encapsulation**, **single responsibility**, and **clear role separation**.

2.2 Reflection on Design Trade-offs

We considered combining the controller and logic layer to simplify the codebase, but ultimately separated them to promote maintainability and allow for better testing and reuse. Although this added some complexity at first, it gave us the flexibility to make changes to the logic layer without affecting user interactions. Another trade-off was deciding whether to use one generic User class or have subclasses like Applicant, HDBOfficer, and HDBManager. We debated the pros and cons and eventually chose inheritance to better reflect role-specific behavior and apply **polymorphism** when checking role permissions.We also initially planned a more complex CSV structure (e.g., multiple files for sub-features) but simplified it to reduce file handling bugs, focusing instead on functional completeness.

Section 3: Object-Oriented Design

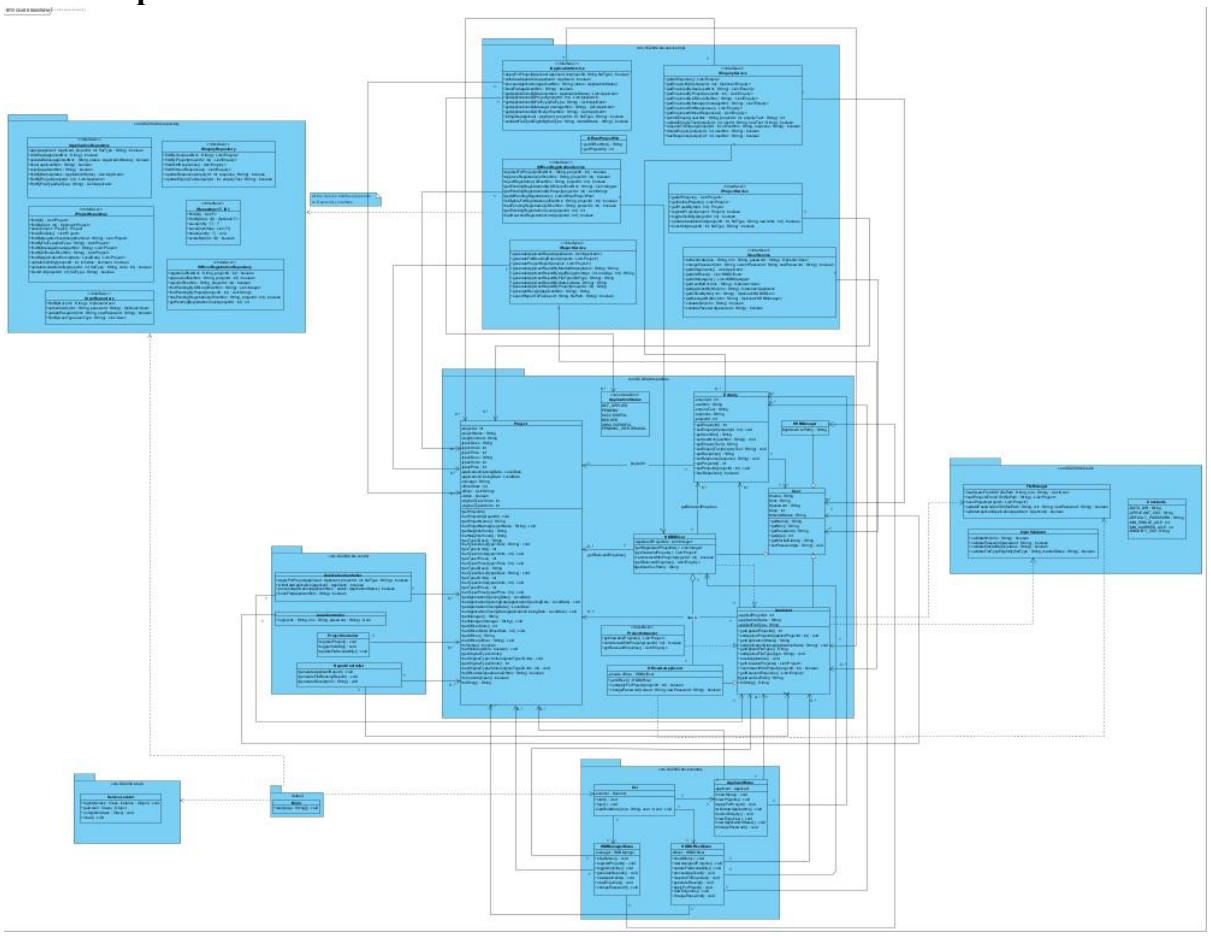
3.1 Class Diagram (with Emphasis on Thinking Process):

We identified the main classes through a close analysis of user roles, use cases, and requirements. Key entities like Applicant, HDBOfficer, HDBManager, Project, and Enquiry were directly mapped from the problem context into entity classes.We adopted a **Model-View-Controller (MVC)** architecture to keep the system modular. Boundary

classes handle CLI interactions, control classes manage application flow, services encapsulate business rules, and repositories abstract data access. Utility classes like FileManager and InputValidator support operations across layers.Responsibilities were clearly assigned to ensure **high cohesion** eg(Entity classes store persistent data.)

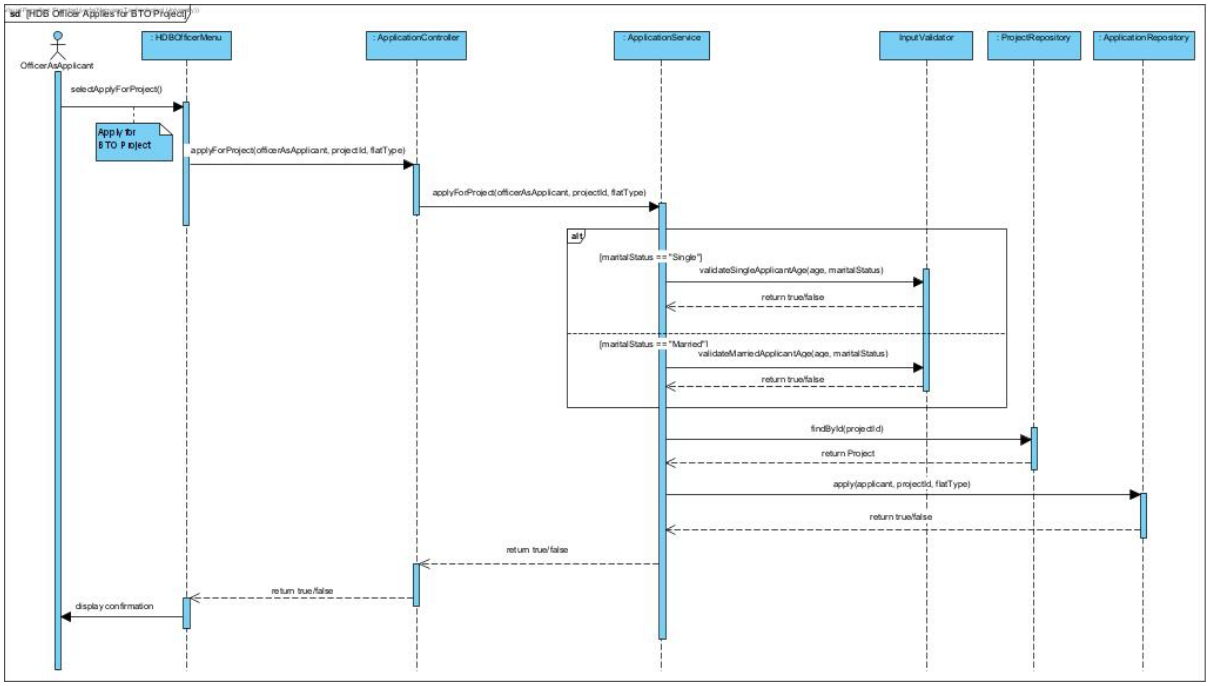
We used **inheritance** for shared behavior — for example, Applicant, HDBOfficer, and HDBManager inherit from User. Interfaces like ProjectInteractor define expected behaviors implemented by relevant classes. **Associations** and **composition** were added where needed, such as Project containing a list of officer NRICs and Enquiry referencing both the user and project.To balance **simplicity and flexibility**, we avoided data duplication by storing identifiers (e.g., NRIC) instead of full objects, and used service interfaces to allow easy testing and future extensions. Dependency inversion was supported via a ServiceLocator, reducing tight coupling across the system.

Final Output:



3.2 Sequence Diagrams

[HDB Officer Applies for BTO Project]

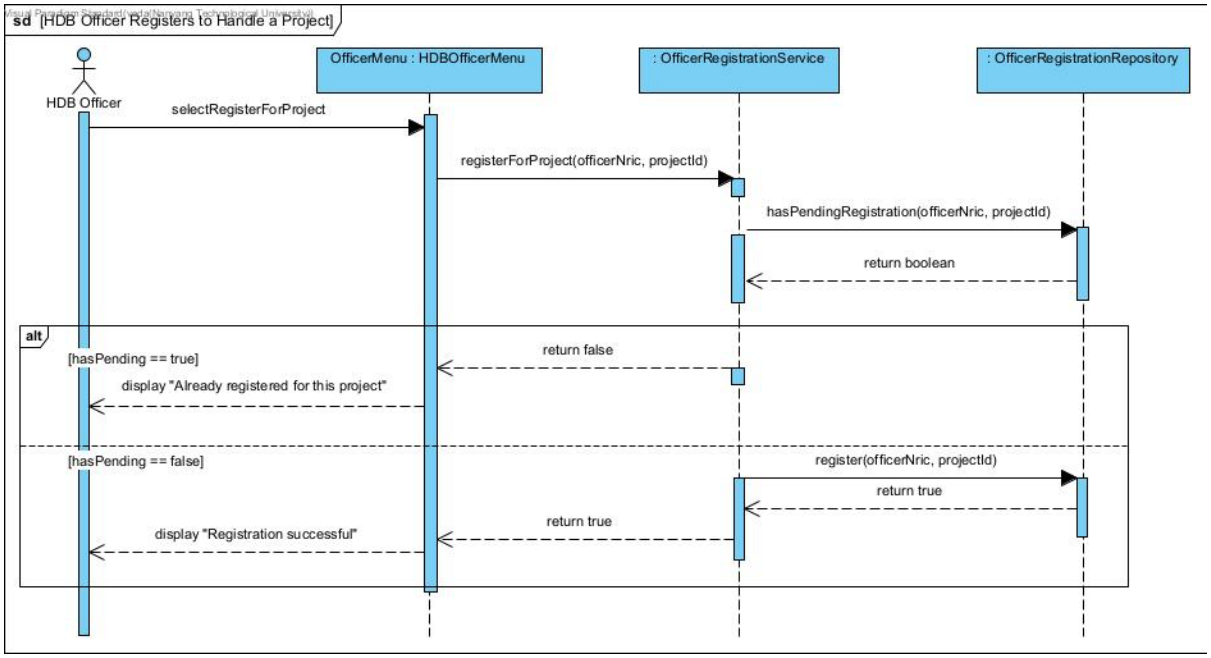


The "HDB Officer Applies for BTO Project" use case was selected because it represents one of the most critical and dynamic user interactions in the system. This scenario exercises multiple architectural layers, including the boundary (HDBOfficerMenu), control (ApplicationController), service (ApplicationService), utility (InputValidator), and repository (ProjectRepository, ApplicationRepository) components. By following the officer's journey as an applicant, the diagram illustrates how the system validates application eligibility, processes application data, and persists updates through the appropriate layers.

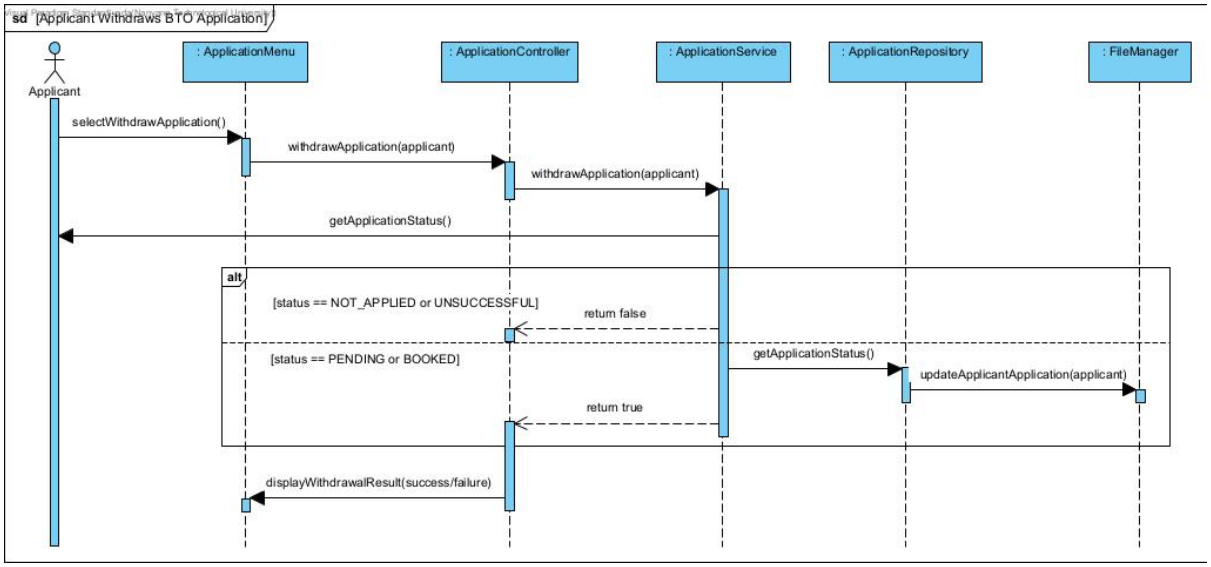
This use case is particularly useful in validating our design because it involves important system behavior such as role-based access, business rule validation, and real-time feedback to the user. It also features a conditional decision based on marital status, modeled using an alt fragment in the diagram. This highlights the system’s ability to apply logic dynamically depending on the applicant’s profile. Furthermore, by using OfficerAsApplicant, the diagram reflects our object-oriented approach to role reuse and inheritance, showing how officers can apply for projects without duplicating logic intended for standard applicants.

Overall, this sequence diagram was chosen because it demonstrates a high level of collaboration between system components, ensuring that our control-service-repository design is sound and extensible. It confirms that each class fulfills its single responsibility and that communication between objects is clear and purpose-driven. As one of the most frequently used features by end-users, it serves as an excellent example of how our system supports real-world user interactions through a well-structured and layered design.

[HDB Officer Registers to Handle a Project]



[Applicant Withdraws BTO Application]



This sequence diagram was chosen because it illustrates a core user-facing feature involving state validation and conditional logic, which are essential aspects of robust application processing. The scenario demonstrates how the system handles a withdrawal request by an applicant, enforcing business rules based on the applicant's current application status.

It effectively showcases the layered architecture of the system—starting from the ApplicationMenu (boundary), passing through the ApplicationController (control), invoking business logic in the ApplicationService, and finally modifying persistent data via the ApplicationRepository and FileManager. The use of the alt fragment explicitly captures the two possible branches: when withdrawal is not allowed (e.g., status is NOT_APPLIED or UNSUCCESSFUL) versus when the withdrawal is accepted and recorded (e.g., status is PENDING or BOOKED). This scenario helps validate our system's correctness in enforcing

application rules and highlights the separation of concerns between UI logic, business logic, and persistence handling.

Overall, it is representative of the system's ability to handle conditional behavior, data integrity, and user feedback, all of which are critical for any real-world BTO management system.

3.3 Application of OOD Principles (SOLID)

Design Principle	**Example**	**Reflection**
Single Responsibility Principle	FileManager.java handles only reading/writing CSV files.	Initially, file handling logic was mixed inside the service classes. We refactored this into a utility class to isolate the file I/O logic. This improved clarity and reduced duplication. However, the trade-off was managing slightly more dependencies between classes (e.g., passing the FileManager into repositories).
Open/Closed Principle	The use of interfaces like IApplicationRepository and the RepositoryFactory pattern.	We wanted to allow future extensions (e.g., swapping CSV with TXT or Excel files) without modifying the service layer. It made our system more extensible and testable, but we had to invest more time upfront to design and implement the interface-factory structure.
Liskov Substitution Principle	Applicant, HDBOfficer, and HDBManager all inherit from User.	This allowed us to use polymorphism and access shared methods (e.g., getNRIC()) across different user roles. It simplified shared behaviors and login handling. However, it required careful role-checking when dealing with role-specific features (e.g., officer-only menus)
Interface Segregation Principle	Split interfaces like IProjectService, IReportService, and IUserService	We didn't want one giant IService interface. Each interface is role- or functionality-specific. It helped keep implementations modular. The downside is managing many interfaces, which can be overwhelming for new team members.
Dependency Inversion Principle	Service classes (e.g., ApplicationService) depend on interfaces, not concrete CSV classes.	This decouples our business logic from data sources and allows mocking in future tests. The use of ServiceLocator helped manage this. While it increased abstraction, it also made debugging more indirect at times.

Section 4: Implementation (Java)

4.1 Tools Used

We used Eclipse IDE, Visual Paradigm and Github for development, Diagrams and version control respectively.

4.2 Sample Code Snippets

Encapsulation: Variables are private, and access is controlled via public methods.

```
public class Project {

   private String projectName;
   private int units2Room;

   public String getProjectName() { return projectName; }
   public int getUnits2Room() { return units2Room; }
   public void reduceUnitCount() { units2Room--; }
}
```

Inheritance: Officers extend User, reusing common properties like NRIC and password

```
public class HDBOfficer extends User {
   private String projectHandled;

   public String getProjectHandled() { return projectHandled; }
}
```

Polymorphism: Role-specific behavior is handled via typecasting and dynamic dispatch.

```
public void showMenu(User user) {
   if (user instanceof Applicant) {
      new ApplicantMenu().display((Applicant) user);
   } else if (user instanceof HDBOfficer) {
      new HDBOfficerMenu().display((HDBOfficer) user);
   }
}
```

Interface Use: Promotes abstraction and reduces coupling.

```
public interface IApplicationService {

   void apply(Project project, Applicant applicant);
   void withdraw(Applicant applicant);
}
```

Error Handling: All file operations are wrapped in try-catch to avoid crashing the CLI.

```
try {

   fileManager.loadProjects("ProjectList.csv");
} catch (IOException e) {
```

```
    System.out.println("Error loading project data: " + e.getMessage());
}
```

→ All file operations are wrapped in try-catch to avoid crashing the CLI.

Section 5: Testing

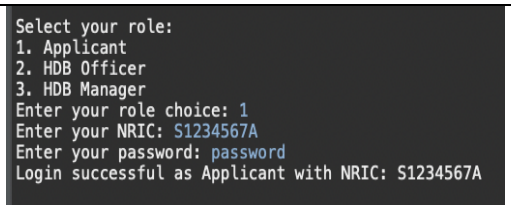
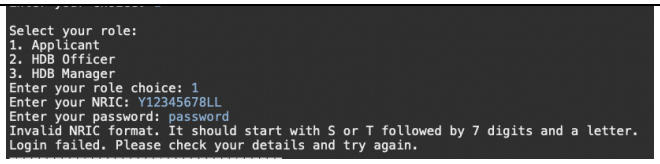
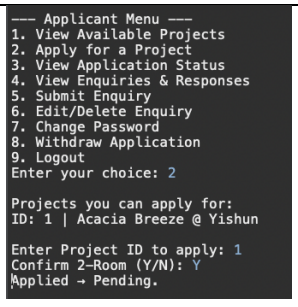
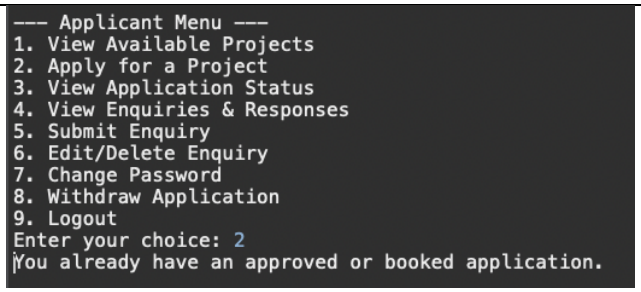
5.1 Test Strategy

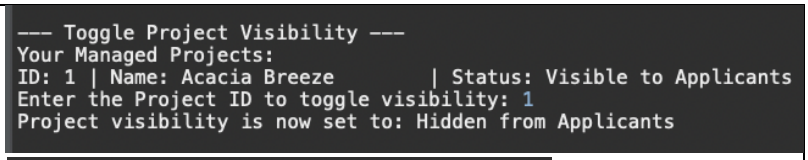
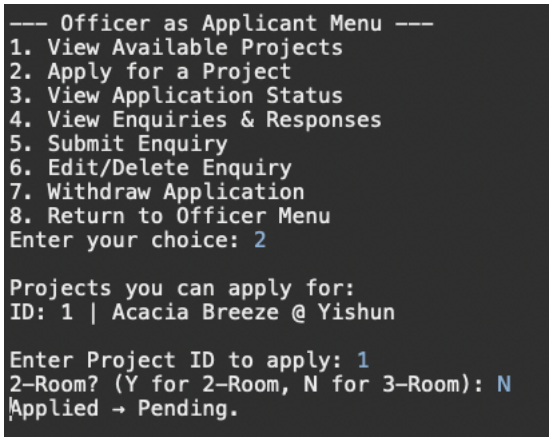
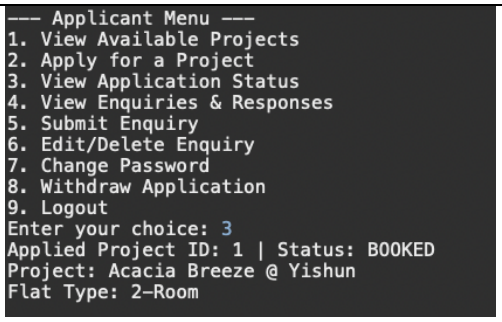
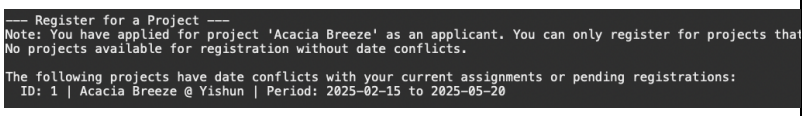
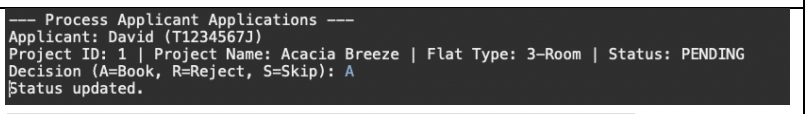
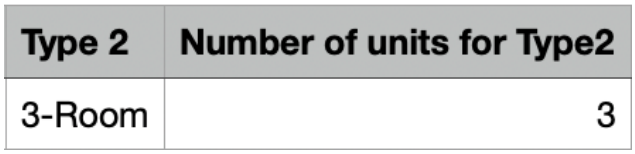
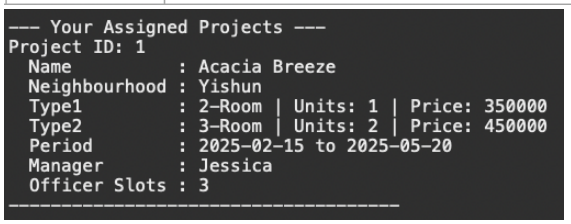
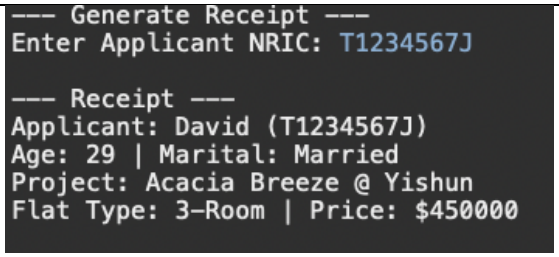
We used **manual functional testing** to validate core features across different user roles. Each team member tested specific flows (e.g., Applicant applications, Officer bookings, Manager approvals) using realistic inputs and scenarios based on the assignment requirements.

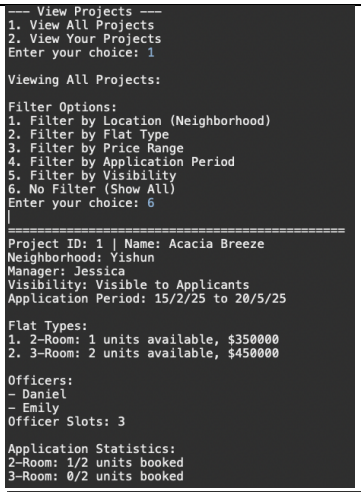
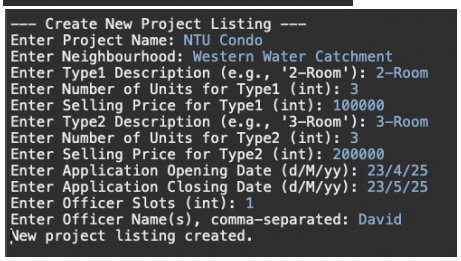
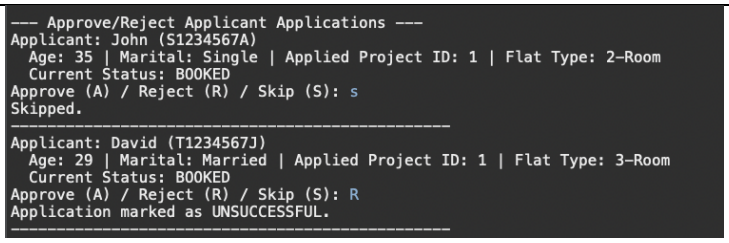
Although we didn't use formal unit testing tools due to the CLI format, we tested individual service methods directly during development to verify business logic. A shared spreadsheet was used to track test steps, expected outcomes, and actual results.

We focused on both standard flows and **edge cases** — such as enforcing age/marital status rules, preventing duplicate bookings, and testing project visibility — to ensure robustness. All tests were repeated until the system behaved as expected.

5.2 Test Case Table

Description	Test Steps	Expected Result	Actual Resul
Valid User Login	Enter valid NRIC and default password, click login	User is directed to dashboard corresponding to their role	 Pass
Invalid NRIC Format	Enter invalid NRIC (e.g., 12345678Z), click login	System blocks login and shows error message for NRIC format	 Pass
Project Application with Eligibility Check	Login as Single user aged 34, attempt to apply for 3-Room flat	System prevents application by only showing applicable projects	 Pass
Restrict Multiple Flat Bookings	Login as Applicant with a successful application, try booking multiple flats	Only one flat booking is allowed; subsequent attempts are blocked	 Pass

View Application Status After Visibility Toggle	Apply for project, then toggle project visibility OFF as Manager, relogin as Applicant	Applicant still sees their application details	  Pass
Officer Registration Eligibility	Login as Officer who has applied for the same project, attempt to register as officer	System blocks registration and gives eligibility error	 Pass 
Flat Booking by Officer	Officer logs in, retrieves successful applicant, updates booking	Applicant status updated to 'Status Updated', flat count decreases	   Pass
Receipt Generation for Booking	Officer books flat for applicant, triggers receipt generation	Receipt contains all necessary info: NRIC, name, flat type, project, etc.	 Pass

Manager Creates/Edits /Deletes BTO Projects	Manager logs in, creates a new project, edits details, deletes it	All actions are successful and reflected in the system	  Pass
Manager Approves/Rejects BTO Applications	Login as Manager, view pending applications, approve and reject	System reflects decisions correctly; updates application status	 Pass

6. Reflection & Challenges

Our team's early alignment on using MVC and SOLID principles was a key success, enabling clear separation of layers (boundary, control, service, repository) and smooth task delegation. GitHub helped us manage code effectively, and object-oriented techniques like inheritance and interfaces simplified user role logic. However, we faced delays due to challenges with Java file handling and CSV persistence, and some methods became repetitive before we introduced refactoring. Flow modeling for features like enquiries and flat booking also caused confusion early on. In hindsight, we would prioritize UML design and documentation earlier in the process. Through this project, we learned that Object-Oriented Design & Programming (OODP) is about structuring problems thoughtfully using abstraction, encapsulation, and single responsibility. A solid design foundation reduces bugs and enhances collaboration—skills valuable not just in coursework, but also in hackathons and real-world development.