SC/CE/CZ2002: Object-Oriented Design & Programming

2024/2025 SEMESTER 2

Build-To-Order (BTO) Management System

GitHub Link

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 honoured 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	Matriculation Number	Lab Group	Group	Signature	Date
SHADID ZARRAF RAHMAN	U2420740B	FDAF	4		23 Apr. 25
AMEEN HUSSAIN SHAMSHAD		FDAF	4		23 Apr. 25
SIDDANTH MANOJ	U2411331F	FDAF	4		23 Apr. 25
PANDIARAJAN SREENITHI	U2323784D	FDAF	4		23 Apr. 25

Chapter 1: Design Considerations

1.1 Design Approach: Overall Architecture and Navigation

Our BTO Management System was meticulously crafted using the foundational principles of Object-Oriented Programming (OOP)—**Encapsulation**, **Inheritance**, **Polymorphism**, and **Abstraction**—to build a modular and extensible system. These principles ensured that our codebase remained manageable, adaptable, and logically structured.

From the outset, we drafted UML class diagrams to define system behaviour and identify responsibilities across components. Iterative refinement of these diagrams allowed us to minimize redundancy, improve cohesion, and enforce logical abstraction boundaries. This upfront planning was essential for preventing design drift and ensuring long-term maintainability.

To implement role-based workflows and enforce separation of concerns, we adopted the **Boundary-Control-Entity (BCE)** architecture:

- **Boundary Layer**:
  This layer encompasses the BTOConsole and various role-specific menu interfaces. Post-authentication, users are directed to their designated menus—Applicant, HDB Officer, or HDB Manager—where they access only the functionalities relevant to their roles. This approach reduces complexity and prevents unauthorized access to certain operations.
- **Control Layer**:
  Controllers such as ApplicationServiceImpl, ProjectManagerImpl, and EnquiryServiceImpl act as intermediaries between the boundary layer and the underlying entities. These services encapsulate business rules and coordinate workflows such as application submission, project creation, and enquiry handling.
- **Entity Layer**:
  This layer includes core data structures such as User, Applicant, BTOProject, Application, and FlatBooking. These classes encapsulate domain logic and persist critical state across sessions via CSV integration. The design ensures clear ownership of data and business behavior.

By employing the BCE architecture, we enforced **loose coupling** between modules and achieved **high cohesion** within them. This design not only supported effective team collaboration but also made the system scalable and easy to debug or extend.

1.2 Applied Design Principles: Extensibility and Maintainability

To ensure the longevity and robustness of our codebase, we rigorously applied the **SOLID principles**. These principles enabled us to write clean, testable, and scalable code while promoting clarity of purpose within each module.

1.2.1 Single Responsibility Principle (SRP)

Each class in our system is dedicated to a single responsibility, ensuring that it has only one reason to change. For example, the ApplicationServiceImpl class is responsible solely for displaying the application status of a logged-in applicant. It is not concerned with how applications are created, validated, or stored.

This strict adherence to SRP offered several benefits:

- **Maintainability**: Isolating concerns allowed us to modify individual functionalities without risking unintended side effects elsewhere in the code.
- **Reusability**: Classes focused on a single task could be reused in different parts of the system (e.g., displaying flat availability for both applicants and officers).
- **Readability**: Logical grouping of responsibilities made the code easier to navigate and understand, especially during debugging or onboarding of new contributors.

```
public class ApplicationServiceImpl implements ApplicationService {
    private List<Application> applications = new ArrayList<>();
    private CSVHandler csvHandler;
    private AuthenticationService authService;
    private ProjectManager projectManager;

    public ApplicationServiceImpl(CSVHandler csvHandler, AuthenticationService authService, ProjectManager projectManager)
        this.csvHandler = csvHandler;
        this.authService = authService;
        this.projectManager = projectManager;
        loadApplications();
    }
```

Figure 1: The ApplicationServiceImpl Class displaying SRP

1.2.2 Open/Closed Principle (OCP)

The **Open/Closed Principle** emphasizes that software components should be *open for extension* but *closed for modification*. This means developers can introduce new functionality without altering existing, stable code - ensuring system integrity while promoting scalability.

In our BTO system, we implemented this principle primarily using abstract classes and interface-based designs. The User class serves as a foundation and is declared abstract. It encapsulates common properties and methods

shared by all user types, such as authentication and NRIC verification. Subclasses like Applicant, HDBOfficer, and HDBManager extend this base class to introduce role-specific features and behavior.

```
public class HDBOfficer extends HDBStaff {
    private BTOProject assignedProject;
    private Application currentApplication;
    private FlatBooking flatBooking;
    private List<Enquiry> enquiries;

    public HDBOfficer(String name, String nric, int age, String maritalStatus, String password) {
        super(name, nric, age, maritalStatus, password);
        this.enquiries = new ArrayList<>();
    }
```

Figure 2.1: HDBOfficer under the user/ directory

For example, HDBOfficer adds attributes related to assigned projects and pending flat bookings, while HDBManager introduces project approval and officer oversight functionalities. None of these extensions required changes to the core User class, showcasing adherence to OCP. This structure makes it seamless to introduce future roles (e.g., an "Admin" or "Contractor") by simply extending User and overriding methods as needed—without compromising existing functionality.

```
public class HDBManager extends HDBStaff {
    private List<BTOProject> managedProjects;

    public HDBManager(String name, String nric, int age, String maritalStatus, String password) {
        super(name, nric, age, maritalStatus, password);
        this.managedProjects = new ArrayList<>();
    }
```

Figure 2.2: HDBManager under the user/ directory

1.2.3 Liskov Substitution Principle (LSP)

The **Liskov Substitution Principle** asserts that objects of a superclass should be replaceable with objects of its subclasses without altering the correctness of the program.

Our system reflects this in components like AuthenticationServiceImpl, which handles login and password validation logic. When performing user actions like password updates or viewing role-based dashboards, the service operates on references of type User. However, at runtime, the concrete object could be an Applicant, HDBOfficer, or HDBManager.

```java
public void changePassword(String newPassword) {
    this.password = PasswordManager.hashPassword(newPassword);
}
```

Figure 3: changePassword() method under the User class

For instance, the changePassword(User user) method works uniformly across all subclasses, as each subclass retains the expected behaviors of its parent. This enables polymorphism and ensures that shared services (like login or profile display) are reusable and easily maintained, without requiring special logic for each role—fully complying with LSP.

1.2.4 Interface Segregation Principle (ISP)

The **Interface Segregation Principle** encourages the use of multiple, role-specific interfaces rather than a single, large, general-purpose one. This ensures that implementing classes only need to depend on the methods they actually use.

In our system, we segregated functionality through precise interfaces such as ApplicationService, EnquiryService, and WithdrawalService. This prevents classes from being forced to implement unused methods.

For example, ApplicationServiceImpl implements IApplicantApplicationService, which includes methods like applyForFlat() and viewApplicationStatus(). Similarly, EnquiryServiceImpl implements only the IApplicantEnquiryService, which includes submitEnquiry() and viewEnquiries(). This clean division prevents bloated classes and allows features to evolve independently, reducing accidental coupling and simplifying unit testing.

1.2.5 Dependency Inversion Principle (DIP)

The **Dependency Inversion Principle** states that high-level modules should not depend on low-level modules directly; both should depend on abstractions (interfaces). Additionally, abstractions should not depend on details—details should depend on abstractions.

In our architecture, controllers such as OfficerMenu and ManagerMenu (part of the boundary layer) rely only on service interfaces like ProjectManager or ApplicationService rather than concrete classes. This is made

possible through dependency injection via constructors during the initialization of the BTOConsole. For instance, ProjectManagerImpl implements the ProjectManager interface, and is injected into the console as a dependency. Thus, BTOConsole only requires a ProjectManager-type object and remains agnostic to the specific implementation details. This promotes testability (as mocks or stubs can be easily substituted) and enhances flexibility by allowing us to swap out implementations or migrate to new frameworks with minimal impact. This inversion of control aligns our system with DIP, resulting in a loosely coupled, highly adaptable structure.

```java
import main.java.sg.gov.hdb.bto.control.interfaces.*;
import main.java.sg.gov.hdb.bto.entity.user.*;
import main.java.sg.gov.hdb.bto.entity.project.BTOProject;
import main.java.sg.gov.hdb.bto.entity.application.*;
import main.java.sg.gov.hdb.bto.util.CSVHandler;
import main.java.sg.gov.hdb.bto.util.IdGenerator;
import main.java.sg.gov.hdb.bto.util.ReportGenerator;
import main.java.sg.gov.hdb.bto.util.AuditLogger;

public class ProjectManagerImpl implements ProjectManager {
    private List<BTOProject> projects = new ArrayList<>();
    private Map<BTOProject, List<OfficerRegistration>> registrations = new HashMap<>();
    private CSVHandler csvHandler;
    private AuthenticationService authService;

    public ProjectManagerImpl(CSVHandler csvHandler, AuthenticationService authService) {
        this.csvHandler = csvHandler;
        this.authService = authService;
        loadProjects();
        loadOfficerRegistrations();
    }
```

Figure 4: ProjectManagerImpl implementing the ProjectManager interface

1.3 Additional Features Implemented

In addition to meeting all core functional requirements, our team designed and implemented several advanced features that significantly elevate the user experience, data resilience, and overall usability of the BTO Management System. These enhancements were guided by user-centered design principles and best practices in software engineering.

1.3.1 Password Validation and Hashing

To ensure robust account security, we implemented a secure and verifiable password management flow:

- Passwords are hashed using **SHA-256** before storage. This ensures sensitive credentials are not stored in plaintext and reduces the risk of compromise if the CSV is accessed externally.

- During login, the system hashes the entered password and compares it with the stored hash using the User.authenticate() method.
- When creating or changing passwords, users must enter the password twice to confirm it matches. This prevents accidental typos from locking users out.

This functionality is encapsulated within the User class, and integrated during login, registration, and password change flows across the system.

1.3.2 Sort and Filter Project Listings

To enhance project discoverability and personalization, we implemented dynamic sorting and filtering capabilities for viewing projects. These capabilities are context-aware, adapting to the user's role:

- **Applicants** can filter projects based on available flat types (e.g., 2-Room) or neighbourhood, using options in the applicant dashboard.
- **HDB Officers** view only their assigned project but can filter flat availability within that project.
- **HDB Managers** view all projects they created and can sort by date, availability, or visibility.

This feature is implemented through methods in ProjectManagerImpl, utilizing Java's Stream API with filter() and Comparator.comparing() to sort and filter project data in memory before presentation.

1.3.3 Persistence Enhancement: CSV Export

To support audit trails and offline analysis, we implemented CSV export functionality for two key areas:

- **Flat Booking Reports**: Managers and officers can export a list of successful flat bookings by accessing the relevant command in their dashboard. This is implemented via the DataServiceImpl.exportBookingReport() method.
- **Enquiries Report**: Similarly, user-submitted enquiries can be exported into a well-structured CSV using exportEnquiriesReport().

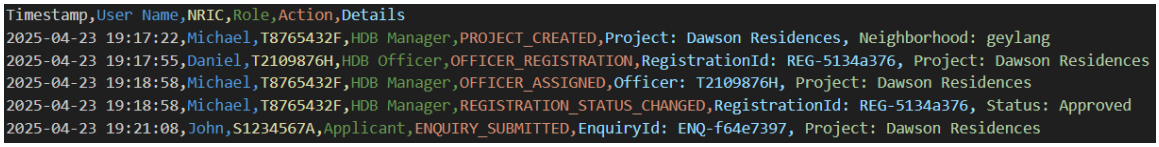
The CSVHandler.writeCSV() method is used to generate the exported file under the resources/exports directory. This allows public housing officers and administrators to keep historical logs, generate reports, or perform analytics externally in tools like Excel.

1.3.4 Audit Log System

To enhance accountability and provide administrators with traceability, we developed an **audit log system** that records critical user actions:

- Logs are generated during flat application, booking confirmation, enquiry submission, and officer registration.
- Each log entry includes the user's NRIC, timestamp, and action performed (e.g., "S1234567A applied for 3-Room at Acacia Breeze").

This logging is handled centrally in DataServiceImpl.logUserAction() and written to a plaintext file audit_log.csv in the resources directory. It serves as a lightweight monitoring mechanism to review system usage and troubleshoot user-reported issues.



```
Timestamp,User Name,NRIC,Role,Action,Details
2025-04-23 19:17:22,Michael,T8765432F,HDB Manager,PROJECT_CREATED,Project: Dawson Residences, Neighborhood: geylang
2025-04-23 19:17:55,Daniel,T2109876H,HDB Officer,OFFICER_REGISTRATION,RegistrationId: REG-5134a376, Project: Dawson Residences
2025-04-23 19:18:58,Michael,T8765432F,HDB Manager,OFFICER_ASSIGNED,Officer: T2109876H, Project: Dawson Residences
2025-04-23 19:18:58,Michael,T8765432F,HDB Manager,REGISTRATION_STATUS_CHANGED,RegistrationId: REG-5134a376, Status: Approved
2025-04-23 19:21:08,John,S1234567A,Applicant,ENQUIRY_SUBMITTED,EnquiryId: ENQ-f64e7397, Project: Dawson Residences
```

Figure 5: Sample data in the audit_logger.csv file, showing recent action history.

Chapter 2: UML Class Diagram

(Complete diagram is also attached separately with project report)

Our class design for the BTO Management System strictly follows the principles of low coupling and high cohesion. To maintain a modular and scalable architecture, we utilized a combination of abstract classes and multiple role-specific interfaces. This approach allows us to reduce rigid class associations, promoting flexibility through interface-based dependencies rather than concrete class links.

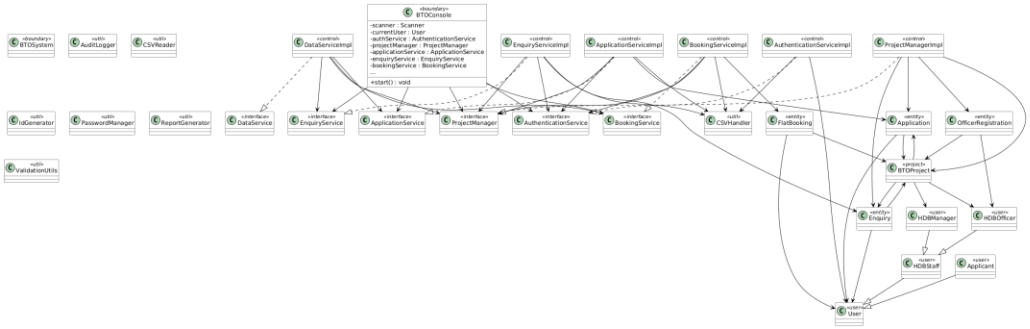


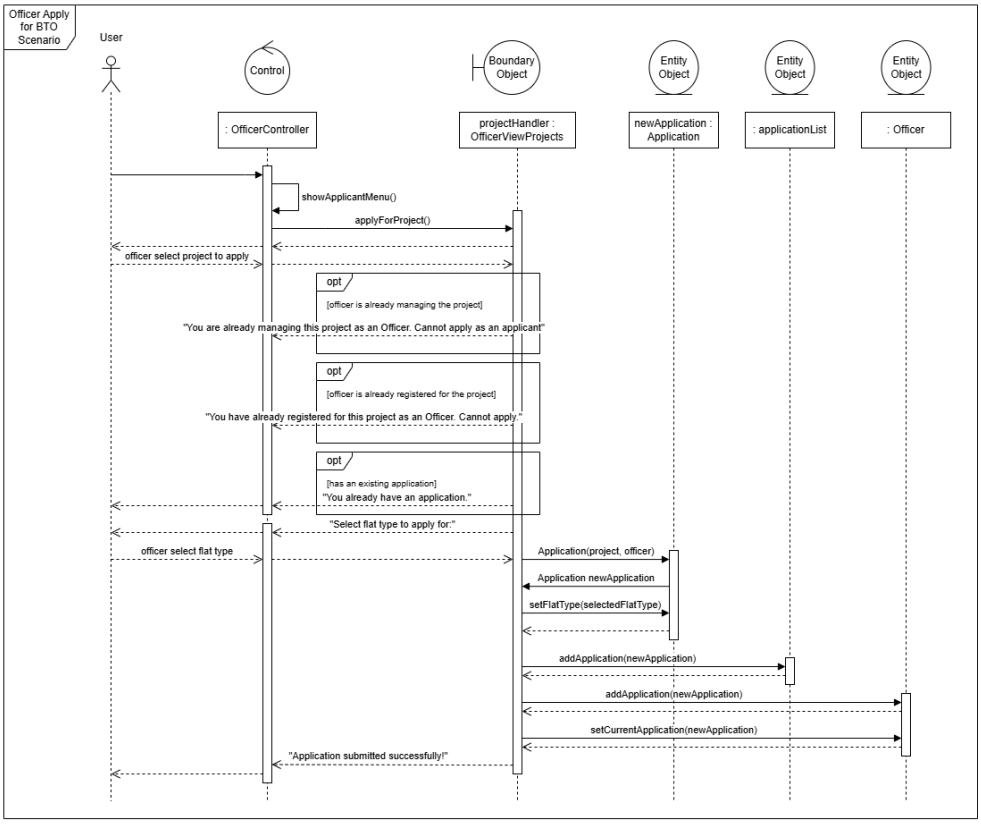
Figure 6: UML Class Diagram for the BTO System

Chapter 3: UML Sequence Diagram

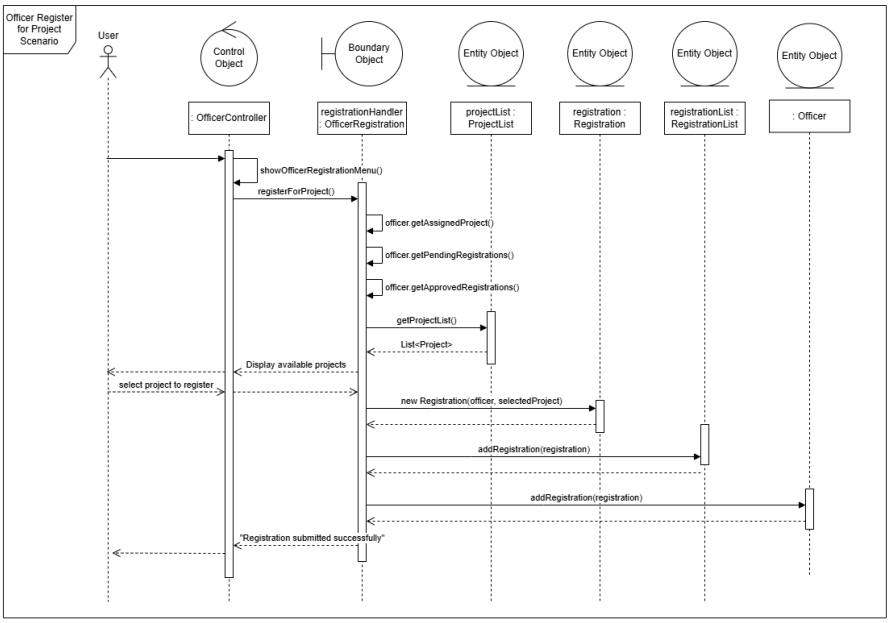
(Complete diagrams are also attached separately with project report)

To maintain clarity and focus, each sequence diagram begins at a meaningful entry point within the system, assuming any prior navigation (e.g., system launch or role selection) has already occurred. The diagrams illustrate the key interactions between the boundary, control, and entity objects involved in handling each use case.

3.1: Officer Apply for BTO Scenario



3.2: Officer Registering for Project

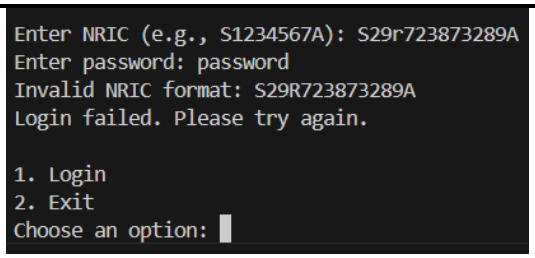
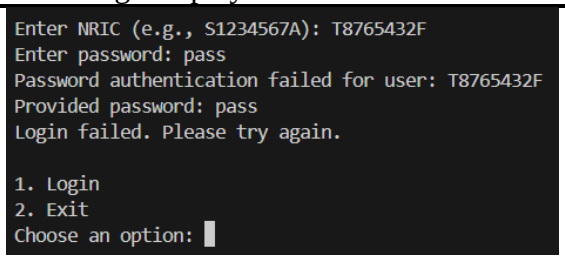
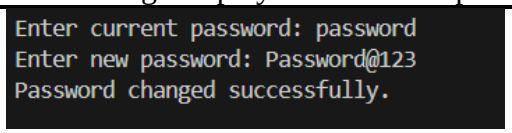


Starting from the BTOConsole, this diagram captures the interaction sequence between the user and the BookingServiceImpl, highlighting booking verification, flat availability checks, and the generation of booking confirmations.

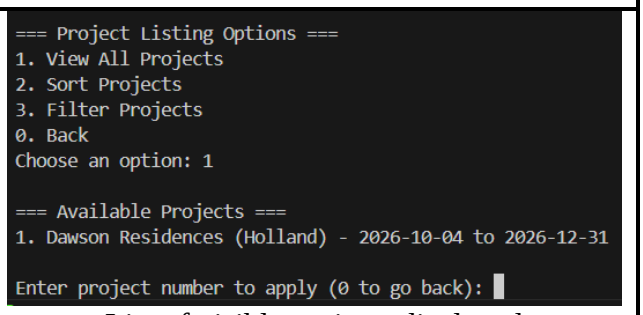
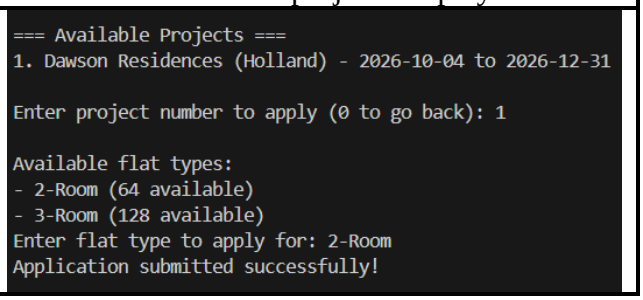
Each diagram showcases how control classes mediate between the UI and the underlying data structures to fulfil user actions within the Build-To-Order (BTO) system.

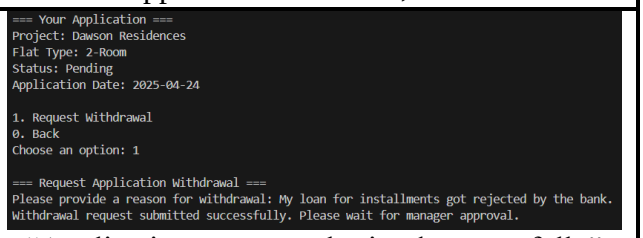
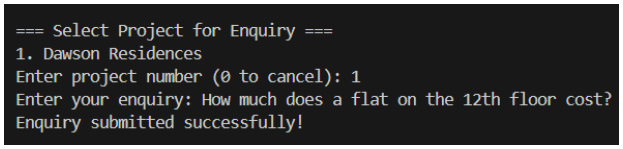
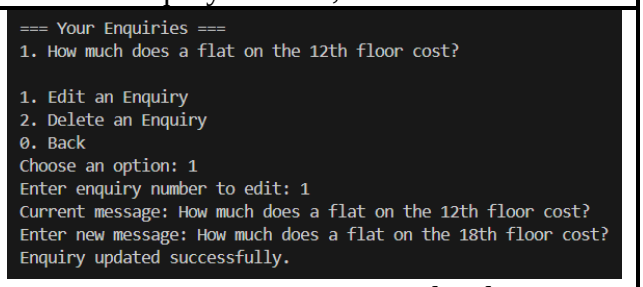
Chapter 4: Test Cases

4.1 User Login and Password Management

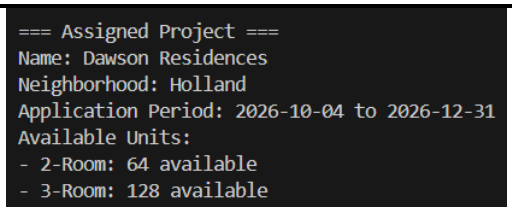
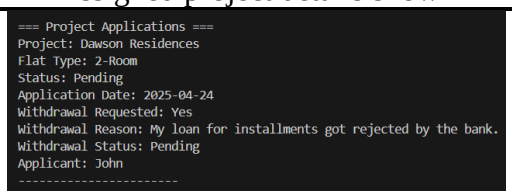
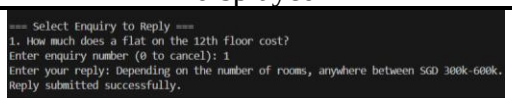
Test Case ID	Description	Inputs	Expected Outcome
TC1.2	Login with invalid NRIC	Invalid NRIC, any password	 Error message displayed: "Invalid NRIC format: "
TC1.3	Login with incorrect password	Valid NRIC, wrong password	 Error message displayed: "Incorrect password"
TC1.4	Change password with correct current password	Old password, new password	 Password is updated and confirmation message shown

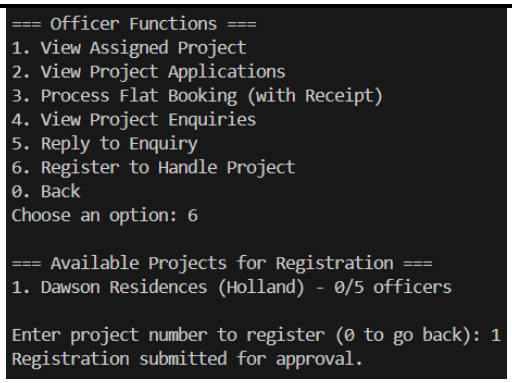
4.2 Applicant

Test Case ID	Description	Inputs	Expected Outcome
TC2.1	View available projects	Logged-in applicant	 List of visible projects displayed
TC2.2	Apply for a flat	Project name, flat type	

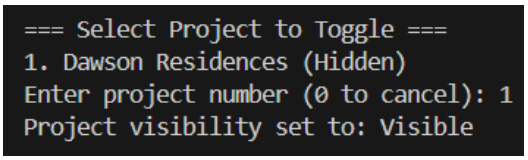
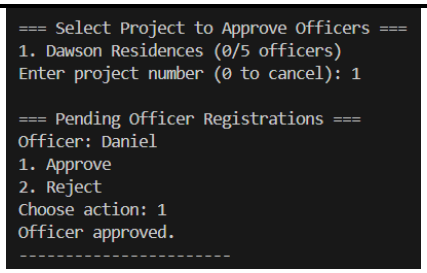
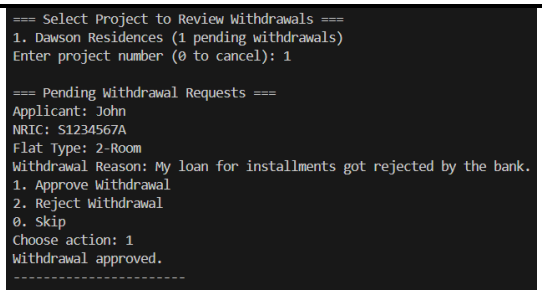
			Application is created, if valid
TC2.5	Request application withdrawal	Reason for withdrawal	 "Application request submitted successfully" message shown
TC2.6	Submit enquiry	Project, message content	 Enquiry is saved, user notified
TC2.7	Edit/Delete enquiry	New message content	 Enquiry message is updated

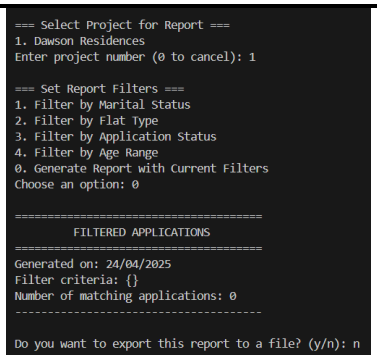
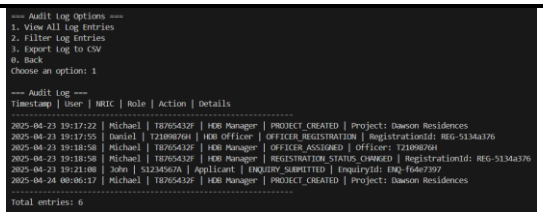
4.3 Officer

Test Case ID	Description	Inputs	Expected Outcome
TC3.1	View assigned project	Officer login	 Assigned project details shown
TC3.2	View applications for assigned project	Officer login	 List of applications for the project displayed
TC3.4	Reply to enquiry	Enquiry ID, reply content	 Enquiry reply saved and displayed to applicant

TC3.6	Apply for new project	Shows current available projects the officer can apply for	 List of projects shown, along with application portal

4.4 Manager

Test Case ID	Description	Inputs	Expected Outcome
TC4.1	Toggle project visibility	Project name	 Visibility status toggled and updated
TC4.2	Approve officer registration	Registration ID	 Officer status changed to approved
TC4.4	Review, Approve, Reject withdrawal request	Application ID	 Withdrawal approved and applicant notified

TC4.6	Generate filtered report	Date range, filters	 Report generated and exported
TC4.7	View audit log	Manager login	 Full audit log displayed

Chapter 5: Reflection

During the early stages of designing our UML Class Diagram for the BTO Management System, our team encountered several structural and organizational challenges. Initial drafts contained redundant classes and overlapping responsibilities that made the system design overly complex and difficult to scale. Through multiple iterations and collaborative reviews, we gradually refined the architecture by consolidating related functionalities into cohesive, reusable classes—enhancing modularity and reducing duplication.

We placed a strong emphasis on the principles of Object-Oriented Programming (OOP), particularly **loose coupling** and **high cohesion**, to ensure that classes remain flexible, focused, and independently testable. This approach greatly improved maintainability and allowed us to adapt the codebase as our understanding of the system evolved.

A key learning curve was mastering persistent data storage using CSV files. We had to self-learn how to properly read from and write to CSV files—such as for Applicants, Officers, Managers, Projects, Applications, Enquiries, and Bookings—while ensuring that user-specific data like passwords and application records are retained correctly across sessions.

5.1 Future Improvement Suggestions

With additional time and resources, there are several enhancements we would include:

- **Mandatory Password Reset on First Login**
  Users would be required to change their default password upon their first login. This helps enforce the creation of personalized and stronger credentials.
- **Multi-Factor Authentication (MFA)**
  Introducing MFA—such as one-time codes or email verification—would add a layer of security, ensuring that only authorized users can access their accounts.

These improvements, along with UI Integration, would significantly strengthen the security, user trust, and robustness of the system, especially if deployed in a real-world housing management context.