NANYANG TECHNOLOGICAL UNIVERSITY

SC2002: OBJECT ORIENTED DESIGN AND PROGRAMMING

BUILD-TO-ORDER MANAGEMENT SYSTEM

Group Project Report

AY 24/25 Sem 2 | FCS4, Group 5

Name	Student ID	Role
Nguyen Tran Thanh Lam	U2420323B	UML Sequence Diagram Designer
Keerthana Sunil	U2411180L	UML Designer/Documentation Lead
Nguyen Tran Chien	U2420243A	Testing and coding
Nguyen Le Tam	U2420673E	Testing and coding

DECLARATION OF ORIGINAL WORK

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
Nguyen Tran Thanh Lam	SC2002	FCS4_G5	*Lam*
Keerthana Sunil	SC2002	FCS4_G5	*(signature)*

Nguyen Tran Chien	SC2002	FCS4_G5	Chiến
Nguyen Le Tam	SC2002	FCS4_G5	Tam

Chapter 1: Requirement Analysis & Feature Selection

1.1 Understanding the Problem and Requirements

The system is built to manage BTO (Build-To-Order) projects, focusing on both applicants and administrative roles (HDB Officers and HDB Managers). We identified a list of essential features which included support for user login across different roles, application submission and withdrawal for applicants, and administrative functions like project creation, enquiry handling, and report generation for managers and officers. By mapping these requirements, it became clear that the overall domain is a multi-role management system for BTO projects—a system where accuracy of data (e.g., timing of applications, eligibility rules) and role-based operations are of utmost importance. Although some areas, such as the precise flow for enquiry edits or the specific format for managerial reports were not fully detailed, we interpreted these ambiguities by aligning them with standard industry practices for similar housing application management systems, ensuring that the final design was both practical and comprehensive.

1.2 Deciding on Features and Scope

We grouped features into three categories: core, optional, and excluded, aiming to demonstrate strong object-oriented design while keeping the system manageable. Core features covered all essential functions for Applicants, HDB Officers, and HDB Managers—such as role-based login, password management, project filtering, application submission and withdrawal, enquiry handling, project assignment, and report generation. These were prioritised based on their alignment with explicit system requirements and end-user needs.

Optional features such as enquiry timestamps, clearer navigational prompts, and smoother role transitions (e.g., switching between options in the menus without repeatedly exiting or backtracking) were identified as enhancements to improve user experience. While beneficial, these features were not essential to the system's core functionality and were planned for implementation only if time permitted.

Chapter 2: System Architecture & Structural Planning

2.1 Planning the System Structure

We laid out a layered architecture inspired by the Model-View-Controller pattern. Although tailored for a CLI environment; the data layer comprises CSV-backed repositories, implementing generic interfaces (`IRepository` and `ICRUDRepository`), which are responsible for loading and persisting `Applicant`, `HDBOfficer`, `HDBManager`, `BTOProject`, `Application`, and `Enquiry` entities. Above this, controllers encapsulate business logic: `UserController` handles authentication and password changes, `ProjectController` and `ApplicationController` manage project applications and booking requests. `EnquiryController` oversees enquiry operations and the UI layer consists of boundary and dashboard classes for each role, along with service classes extending an abstract menu framework (`AbstractMenu` and `AbstractViewProjectsMenu`). This menu presents options, prompts for input, and invokes controller methods. We also introduced an additional feature, a **Notification system** to provide in-app alerts whenever key events occur. In the data layer we added a new `Notification` entity and a `NotificationRepository` (implementing `ICRUDRepository<Notification>`) which persists notifications in its own CSV file alongside our other repositories. Further, in the controller layer we induced a `NotificationController` that depends only on the `ICRUDRepository` abstraction. It showcases methods that send a notification by NRIC and retrieves all notifications for a user, and marks them as read. We then wired this controller into our existing services so that, for example, when the `ApplicationController` changes an application's status, it calls send `(applicant.getNric(), message)`.

2.2 Reflection on Design Trade-offs

Simplicity vs. Extensibility: Throughout our architectural planning, the team aimed to strike a careful balance between simplicity and extensibility. Initially, we considered merging controllers and repositories into single classes to speed up development. But we quickly recognized that this shortcut would compromise testability and make future changes difficult, as tightly integrated code often becomes fragile when requirements shift. Therefore, we chose a practical path: repositories focus strictly on raw data handling, controllers encapsulate business logic, and UI

components are responsible solely for presentation. Each layer interacts clearly through defined interfaces and dependency injection, ensuring a clean separation of concerns without unnecessary complexity. This approach also makes incremental enhancements straightforward. For instance, adding a new user type or changing the storage method requires only the creation of a new subclass or repository—no major rewrites needed.

3. Object-Oriented Design

The detailed UML diagrams are attached in a separate file as they were unable to fit into the report.

3.1 Class Diagram:

We identified core classes such as BTOProject, Application, and User by analysing key domain concepts; repository classes (e.g., `ApplicationRepository`, `ProjectRepository`) manage core data and behavior; boundary classes (e.g., `LoginBoundary`, `ManagerDashboard`) handle user interaction; controller classes (e.g., `UserController`, `EnquiryController`) coordinate logic across layers. Inheritance is used meaningfully: the abstract `User` class is extended by concrete types, and `HDBOfficer` further extends `Applicant`, allowing officers to apply for flats just like applicants and respecting the Liskov Substitution Principle, since `HDBOfficer` can be used wherever an Applicant is expected without breaking system behavior.

Association is used where appropriate—e.g., `Enquiry` with `EnquiryRepository` and `Application` with `ApplicationRepository`; `Report` aggregates `Receipt`; `Applicant` and `HDBOfficer` aggregate `Application`; and `HDBManager` aggregates `BTOProject`. Controllers depend on interface repositories for data access (e.g., `EnquiryController`→ `ICRUDRepository<Enquiry>`), thus promoting maintainability or loose coupling: you can change how data is stored (e.g., from CSV to a database later) by modifying only the repository, not the controller.

3.2 Sequence Diagrams
3.2.1 Use Case - Manager generating report

When a manager selects option 9 on the HDBManagerDashboard, GenerateReportService retrieves and displays their managed projects, then prompts 'Enter project ID to generate report' (or 'b' to go back):– exiting on "b," printing "Invalid input." on parse errors, and validating the ID against the project list (printing "Invalid project ID." if not found). It then calls ApplicationController.getAllApplications(), gathers all BOOKED applications for that project, and, if none exist, prints a "No booked applications…" message. Otherwise it constructs a new Report(bookedApps).toString(), prints it, and prompts Would you like to apply a filter? (Y/N):– on "Y" asking for a filter option (1=marital status, 2=flat type, 3=both), applying removeIf lambdas to refine bookedApps (or printing an invalid-option warning), reprinting a filtered or empty report, on "N" printing "No filter applied." Finally it prompts Type 'b' to go back: and exits on "b." This flow showcases encapsulation (only Application exposes status and type), separation of concerns (the service handles I/O and filtering; Report handles formatting), and the controller–service–repository pattern.

3.2.2 Use Case - Manager editing or deleting BTO Project

When a manager selects option 2 on the HDBManagerDashboard, EditDeleteProjectService calls manager.getManagedProjects(): if the list is empty it prints "You have no projects to manage." and exits; otherwise it displays the projects and prompts Enter project ID or 'b' to go back: – returning on "b," printing "Invalid project ID." on parse failures or missing IDs. For a valid ID it prompts 'e' to edit, 'd' to delete, or 'b' to return: – on "e" showing a numbered field menu (name; neighborhood; 2-room/3-room units and prices; open/close dates; officer slots), reading a choice via Prompt.promptInt(), invoking the matching BTOProject.setXxx(newValue), then calling projectController.updateProject() and printing "Project updated." On "d" it removes the project from the manager's list, iterates each HDBOfficer to call unassignProject(P) where needed, invokes projectController.removeProject(P), and prints "Project deleted." Finally it prompts Type 'b' to go back: and exits on "b." This flow demonstrates encapsulation (state changes via setters only), separation of concerns (service handles prompts and branching; controller handles persistence), and modular collaboration across UI, service, controller, domain, and repository layers.

3.3 Application of OOD Principles (SOLID)

Single Responsibility Principle (SRP): Each dashboard—for Applicants, HDBManagers, and HDBOfficers—focuses on its unique set of functions, ensuring that user interaction, data retrieval, and business logic are handled by separate, well-defined components. For example, by having each dashboard extend a common `SimpleMenu` and utilise service classes for specific tasks (like viewing projects or managing enquiries), each class has one clear responsibility, which aligns with SRP.

Open/Closed Principle (OCP): We anticipated that new features—such as additional ways to view or filter projects—would emerge. To avoid "modifying" existing menus each time, we built an abstract base `AbstractViewProjectsMenu` in our UI layer. Concretely, the `ViewAvailableProjectsService` and `ViewAllProjectsService` each extend this base and simply override a single method (`getProjectString`). This lets us add new listing behaviours (for officers, managers, or future roles) without touching the parent class. By programming to the abstraction of "a view‑projects menu" rather than concrete implementations, we kept the framework closed for modification but open for extension.

Liskov Substitution Principle (LSP): We needed a uniform way to handle any kind of user—applicant, officer, or manager—through the same login boundary and dashboard switch. By ensuring that each subclass of `User` (`Applicant`, `HDBOfficer`, `HDBManager`) adhered to the same public contract (`getRole`, `getName`, `getNric`), we could treat them interchangeably in the main loop. For example, `UserController.login(...)` returns a `User`, and we immediately call `user.getRole()` to decide which dashboard to show. Since `HDBOfficer` extends `Applicant` without breaking any of the `Applicant` behaviors—officers can still apply for a flat, view personal enquiries, etc.—we honored Liskov: any service that expects an `Applicant` can safely handle an `HDBOfficer`. That made our login‑and‑dispatch logic straightforward, though it did require careful design to avoid accidentally overriding or weakening inherited preconditions in the officer subtype.

Interface Segregation Principle: At first we considered a single "everything" repository interface, but realized that not all clients need every method. For example, the `ApplicantRepository` needs only load, getAll, and update—it never needs to delete users in our flows. Meanwhile, `ApplicationController` needs add/remove/getById, but doesn't

need to list managers. By splitting our persistence abstraction into `IRepository<T>` (basic load/getAll/update) and `ICRUDRepository<T>` that extends `IRepository<T>` (adds create/remove/getById), each controller and service depends only on the interface that provides exactly the operations it uses.

Dependency Inversion Principle (DIP): High-level modules (App, controllers, services) depend on abstractions (`ISystem`, `IRepository`, `ICRUDRepository`), not on concrete classes. In `App.initialize()`, we bind interfaces to implementations `ApplicantRepository`, `ProjectRepository`, etc, before wiring controllers and App. This design allows easy substitution or mocking of dependencies for testing.

Trade-offs:

SRP led to clearer responsibilities but increased the number of classes and initial setup. OCP made extending functionality easier without modifying existing code, though it added complexity in managing subclasses. LSP ensured subclass compatibility but required careful abstraction. DIP enhanced testability and decoupling through dependency injection, though it introduced more boilerplate and setup effort. Overall, the benefits in maintainability and future scalability outweighed the costs.

4. Implementation (Java)

4.1 Tools Used:

Tools Used:

Java 17: The project is implemented in Java 17, which brings modern language features (such as improved switch expressions and records in later versions) and enhanced performance.

IDEs (IntelliJ / Eclipse): These IDEs are used for code management, debugging, and building the application. They provide powerful refactoring tools and integrated support for version control.

Version Control (GitHub): GitHub is employed for code hosting, collaboration, and maintaining a history of changes. This supports team collaboration and code review.

4.2 Sample Code Snippets:

<u>INHERITANCE</u>

```
// Inheritance: Applicant extends the abstract User class
public class Applicant extends User {
    // Constructor demonstrating inheritance by calling the parent constructor
```

```java
    public Applicant(String name, String nric, String password, int age, String maritalStatus) {

        super(name, nric, password, age, maritalStatus); // Calling the constructor of User

    }

    // Overriding the abstract method from User

    @Override

    public String getRole() {

        return "Applicant";

    }

}
```

POLYMORPHISM

```java
// indicates that CreateProjectService is a subclass of AbstractMenu.

public class CreateProjectService extends AbstractMenu {

    // ...

// override methods display polymorphism since when these methods are called on an object

// referenced as an AbstractMenu, Java's dynamic dispatch will invoke the implementations in

// CreateProjectService

    @Override

    public void display() {

        System.out.println("\n=== Create a New BTO Project ===");

    }

    @Override

    public void handleInput() {

        // ... (implementation details)

    }

}
```

ENCAPSULATION

```java
// from CreateProjectService class. By declaring fields as private and exposing only necessary
// operations (through public methods or constructors), the class ensures that its internal state is
// protected and cannot be modified arbitrarily from outside the class.
private HDBManager manager;

private ProjectRepository projectRepository;

private SimpleDateFormat dateFormat = new SimpleDateFormat("dd/MM/yyyy");


// The class exposes its behavior through public methods (such as display() and handleInput()).
// These methods operate on the encapsulated data, allowing controlled interaction with the object's
// state.


@Override

public void display() {

    System.out.println("\n=== Create a New BTO Project ===");

}
```

7

```
@Override
public void handleInput() {
    // Implementation details that process user input, modify state, etc.
}
```

<u>INTERFACE</u>

```
package ui;


public interface InputHandler {
    void handle(String input);
}
```

And

```
package ui;


public interface IUIElement {
    void display();
    void handleInput();
}
```

<u>ERROR HANDLING</u>

```
// input validation from DeleteEnquiryService
try {
    int enquiryId = Integer.parseInt(input);
    enquiryController.deleteEnquiry(enquiryId);
    System.out.println("Enquiry deleted successfully.");
} catch (NumberFormatException e) {
    System.out.println("Invalid input. Please enter a valid enquiry ID or 'b' to go back.");
}
```

Chapter 5: Testing

5.1 Test Strategy

Our team approached testing the system primarily through black box testing. We developed and executed a comprehensive set of test cases that covered both normal operations and valid/invalid cases. The tests were designed to verify important features, ensuring that the system behaved as expected under a variety of conditions. By manually validating each test case, we were able to observe real-user interactions, identify inconsistencies, and confirm the reliability of critical functionalities across our application.

5.2 Test Case Table

No:	Test Case	Input(s)	Expected Result	Outcome
1.	Valid User Login	Login with valid/invalid NRIC and password S1234567A S123,G1234567B.	User should be able to access their dashboard based on their roles if the login is right. If not then it will re prompt for the password	Passed.
2.	Password Change Functionality	Testing password change for all 3 kind of users	System updates password, prompt re login and allows login with new credentials	Passed.
3.	Project Visibility Based on User Group & Toggle	Log in as Applicant, Manager and toggle on off Project visibility	Projects are visible to users based on their age, marital status and the visibility setting. (Only manager can toggle the visibility)	Passed.
4.	Project Application	Login as Applicant and Officer then register for Project	Users can only apply for projects relevant to their group or when visibility is off	Passed.
5.	Single Flat Booking per Application	Login as Applicant and Officer then book the second flat	System allows booking one flat and restricts further Bookings	Received.
6.	Able to register new project after last project was successful	Log in as Applicant and apply for second project after the first one was successful	Able to register new project after last project was successful	Passed.
7.	Applicant's enquiries management	Submit, edit, delete enquiry as applicant	Enquiries can be Successfully submitted, displayed, modified, and removed..	Passed.
8.	HDB Officer Registration Eligibility.	Register as officer and check the status	System allows Registration and check the status	Passed.

9.	Response to Enquiries	Manager/officer responds to enquiries	Officers & Managers can access and respond to enquiries Efficiently	Passed.
10.	Flat Selection & Booking Management.	Officer updates booking and flat availability	Officers retrieve the correct application, update flat availability accurately, and correctly log booking details in the applicant's profile.	Passed.
11.	Receipt Generation for Bookings	Generate receipt for a booking	Accurate and complete receipts are generated for each successful booking	Passed.
12	Create, Edit, Delete Project Listings	Manager creates, edits, deletes projects	Managers should be able to add new projects, modify existing project details, and remove projects from the system.	Passed.
13	Single Project per Application Period	Manager creates multiple projects in one period	System prevents assignment of more than one project to a manager within the same application dates.	Passed.
14	View All / Filtered Projects	Manager views all/filter projects	Managers should see all projects and be able to apply filters to (we apply the filter for all the user)	Passed.
15	Approve/Reject BTO Applications & Withdrawals	As manager, we can approve or reject application	Approvals and rejections are processed correctly, with system updates to reflect the decision	Passed.
16	Generate & Filter Reports	Generate reports and apply filters	Accurate report generation with options to filter by various categories.	Passed.
17	Notification on successful action	Act as an applicant and tried to apply for a project	The system should sent notifications after each success action.	Passed.

6. Reflection & Challenges

6.1.1 What Went Well

The use of encapsulation, inheritance, and polymorphism made it easier to reason about user roles and shared behaviors across different types of users.

- Separation of Concerns: Distinct layers for UI (boundary classes and AbstractMenu), controllers, models, and repositories helped in isolating functionality and improving maintainability.
- Error Handling: The project successfully anticipates various user input errors (e.g., invalid NRIC format, parse exceptions) and provides user-friendly messages.

6.1.2 Areas for Improvement

While abstraction was achieved through abstract classes, further decoupling could be achieved using more Java interfaces for persistence or service layers, which would enable easier swapping of implementations. Currently, many UI messages and prompts are hard-coded. A more configurable UI component could allow for internationalization or customization of user interactions.

6.1.3 Lessons Learned

Applying SOLID design principles helped structure the project in a way that achieves clear modularity. In particular, careful consideration was given to the Single Responsibility Principle (SRP) and the Open/Closed Principle, ensuring that adding new features, like additional service, would not require major changes to the existing codebase. This project offered insights into how design trade-offs such as simplicity versus extensibility manifest in a real-world project, and how these decisions have the ability to affect testing and maintenance.

7. Appendix

• GitHub link