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

ASSIGNMENT

Building an OO Application - BTO Management System

SC2002 - FDAD - GRP 3

2024/2025 SEMESTER 2

COLLEGE OF COMPUTING AND DATA SCIENCE (CCDS)

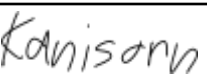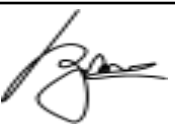**NANYANG TECHNOLOGICAL UNIVERSITY**

APPENDIX B:

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
Daga Shreyansh Kumar	CBC/Y2	FDAD	24/04/2025
Lee Zhi Xuan, Janelle	DSAI/Y1	FDAD	24/04/2025
Sawangsawai Kanisorn	DSAI/Y1	FDAD	24/04/2025
Yee Zhe Kai, Matthew	DSAI/Y1	FDAD	24/04/2025

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.

1. Requirement Analysis and Feature Selection

1.1 Understanding the Problem and Requirements

Our team began by reading the Week 8 document line by line to understand the core objectives of the BTO Management System. We broke down the document in sections, identifying key Actors (Applicant, Officer and Manager) and mapping out their responsibilities. Being a multi-user role system, we identified the main problem to be the interaction with a shared set of BTO projects through the application, registration and management process.

1.2 Deciding on Features and Scope

Our team compiled a list of features based on both explicit requirements and implicit user expectations, and then prioritized them according to feasibility and time constraints. We also decided to improve certain features to enhance the user experience. For example, instead of requiring officers to manually update the number of flats remaining, we implemented an automatic update mechanism triggered whenever an application is approved.

In addition, we made decisions to simplify certain features while staying within scope. For example, while advanced filtering options were considered, they were deprioritized due to their complexity and our time constraints. Our main goal was to ensure that all essential features were implemented effectively and reliably, while leaving room for potential future enhancements.

2. System Architecture & Structural Planning

2.1 Planning the System structure

As part of our early system planning, we created a hand-drawn blueprint (See Figures below) outlining the overall framework for Applicant, Officer and Manager. This helped us to visualise the overall flow of the system better, showing how the different actions would be connected. This serves as a foundation for translating requirements into system design. These visual aids also revealed areas where transitions between states or interfaces were unclear, allowing us to make adjustments that improved the system's overall flow.
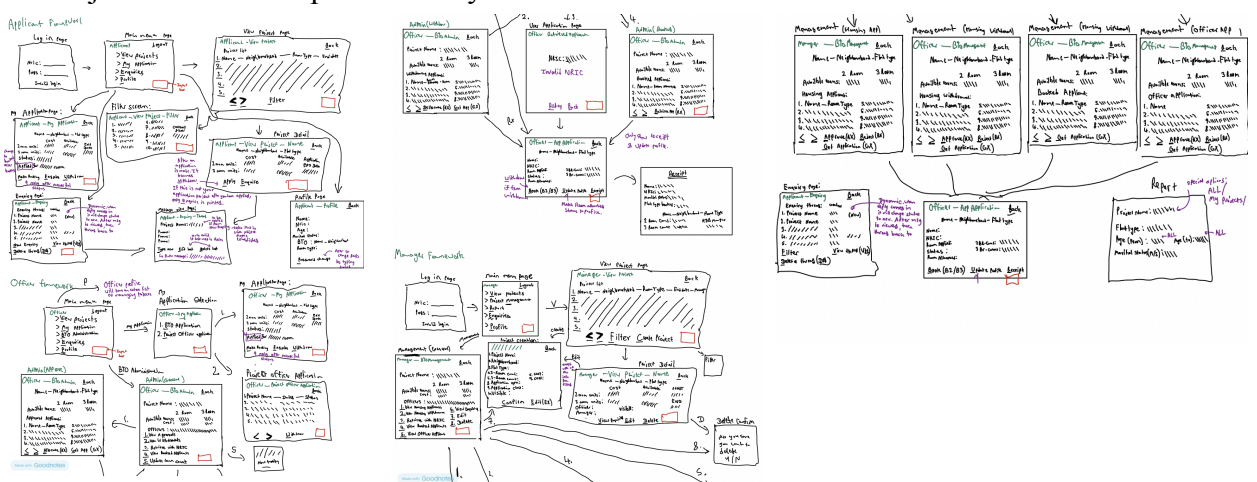
We then identified and broke down the system into logical components based on user roles and functionalities. Examples include:

- UserManager<T> interface and its role specific implementations to manage logic for each user type (Applicant, Officer, Manager)
- Project Manager to oversee project related operations
- UI classes (ApplicantUI, HDBOfficerUI, HDBManagerUI) to handle interactions
- Specialised managers to handle more complex workflows and system validation

2.2 Reflecting on Design Trade-offs

The main trade offs was the implementation of the boundary classes – ApplicantUI, HDBOfficerUI and HDBManagerUI. While they centralised user interaction logic effectively, they became increasingly large and complex as we integrated more functionalities into a single class. This made it difficult to fully follow the SOLID principle. We considered breaking down the UI into smaller subcomponents, but this would have required additional refactoring and interface coordination, which we deprioritized due to time constraints. As a result, while the classes served as a convenient hub for user actions, its **size eventually became a trade-off between simplicity and maintainability**. This made us realize the importance of keeping our code modular and concise. If we have to improve or scale the system in the future, we plan to refactor these large classes into smaller, more manageable components to make it easier to test, maintain, and extend.

2.3 Approach Taken

We employed a layered-architecture in the following:

- Presentation Layer: boundary Package
- Control Layer: control Package
- Entity Layer: entity Package
- Enums Layer: enums Package
- Data Access layer: data folder + utils Package
- Authentication Layer: auth package

This helped us to separate concerns, improve maintainability, reusability of shared components and make it easier for debugging and testing. Additionally, it also allowed for better team collaboration as we were able to work on different layers in parallel, reducing the risk of conflict when pushing our updated code into Github

3. Object-Oriented Design

3.1 Class Diagram

Please view our class diagram from the Github Link here: **ClassDiagram**

3.2 Sequence Diagrams + additional ones showing important interactions

Please view our sequence diagram that shows the flow of the HDB Officer's role in applying for a BTO and registering to handle project here: **SequenceDiagram**

3.2.1 BTO Project Application Process

The application process is the main interaction point for Applicants, Officers, and Managers. The sequence diagram below emphasizes the system's ability to facilitate this process seamlessly, which begins with applying for the project, followed by Manager approval, and finally Officer booking of the flat.
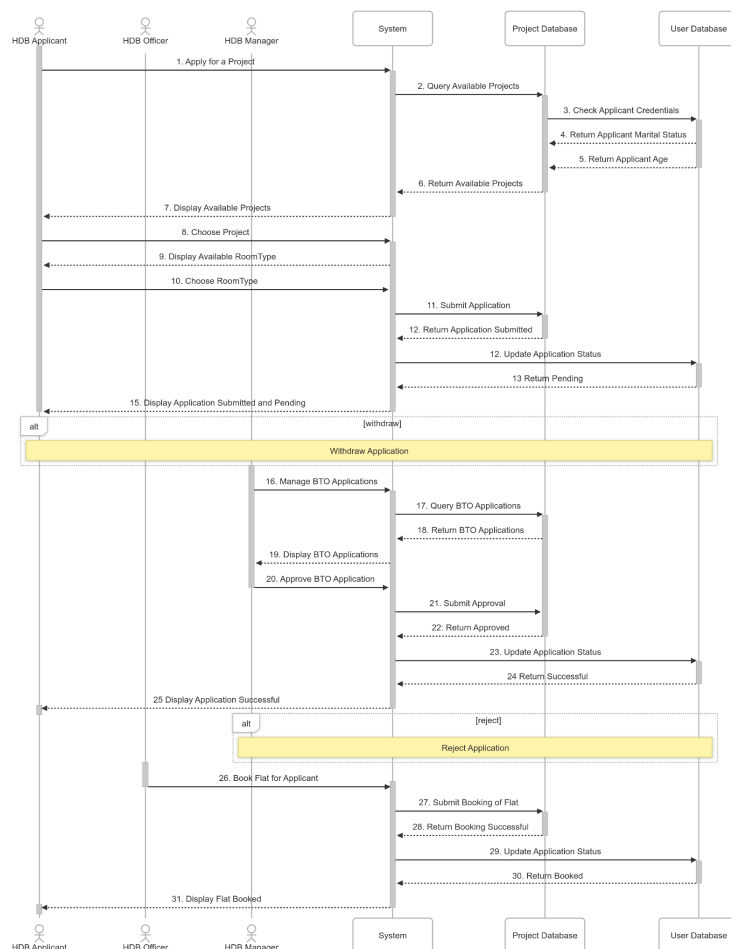
3.2.2 Enquiry Process

The enquiry process is crucial for Applicants to interact with Officers and Managers. Applicants are able to submit enquiries as many times as they want, with either the Registered Officer or Manager of the project replying to them. Officers are also able to submit enquiries if they take on the role of the Applicant.
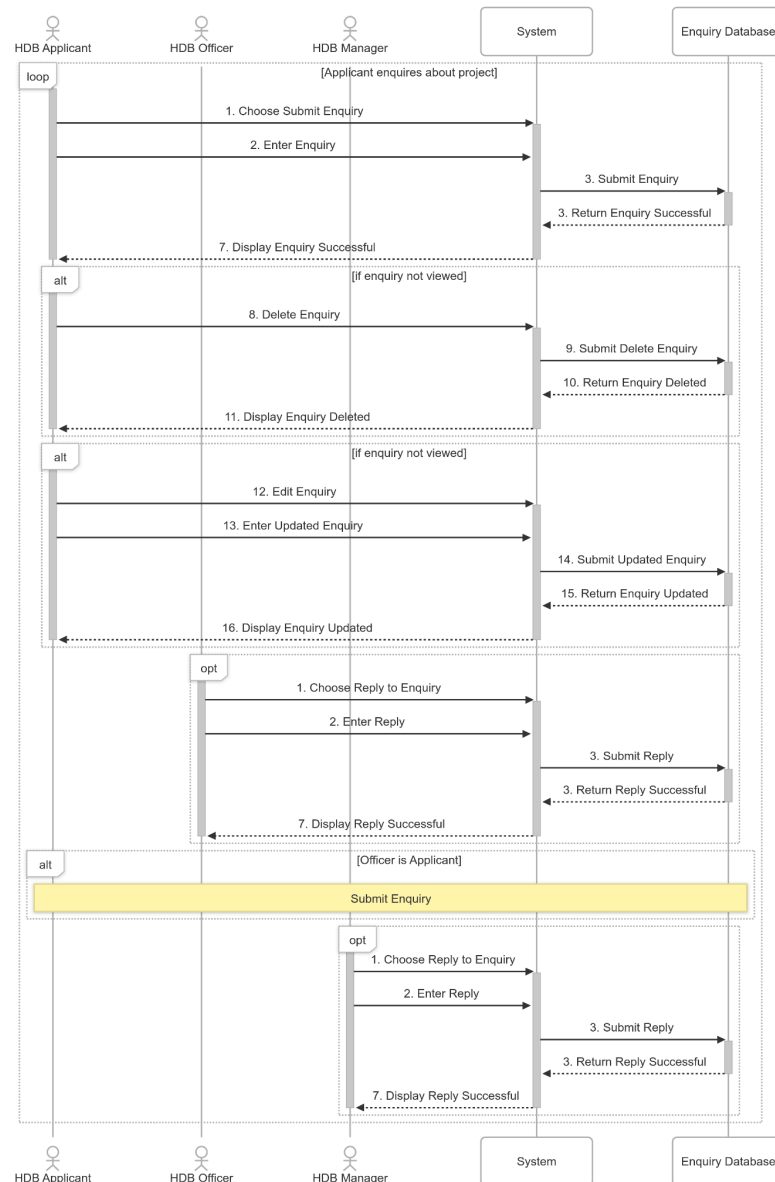
3.3 Application of Design Principles(SOLID)

Single Responsibility Principle (SRP)

The **FilterUI** class demonstrates the **Single Responsibility Principle** as it is solely responsible for handling user input related to filter options for viewing. Its only task is to present a filter configuration menu to the user and update the Filter object based on the user's selections. It does not handle any data storage, project filtering logic, or filter application. Those responsibilities are delegated to other classes such as FilterManager and ViewProjectFilter. This clear separation improves code readability, maintainability, making each component of the Filter function easier to test and modify independently.

```
public class FilterUI {

    public static void promptFilterSettings(Scanner scanner, Filter filter) {
        while (true) {
            System.out.println("\n========== Project Filter Settings =========");
            System.out.println("Current Neighbourhood Filter: " + (filter.getNeighbourhood() != null ?
                                filter.getNeighbourhood() : "None"));
            System.out.println("Current Room Type Filter: " + (filter.getRoomType() != null ? filter.getRoomType() : "None"));
            System.out.println("============================================");
            System.out.println("1. Set Neighbourhood Filter");
            System.out.println("2. Set Room Type Filter");
            System.out.println("3. Reset All Filters");
            System.out.println("0. Back to Main Menu");
            System.out.println();
            System.out.print("Choose an option: ");

            int choice = -1;
            try { choice = scanner.nextInt(); } catch (InputMismatchException e) {}
            finally { scanner.nextLine(); }

            switch (choice) {
                case 1:
                    System.out.print("Enter neighbourhood: ");
                    String n = scanner.nextLine().trim();
                    filter.setNeighbourhood(n.isEmpty() ? null : n);
                    break;
                case 2:
                    RoomType[] values = RoomType.values();
                    for (int i = 0; i < values.length; i++)
                        System.out.println((i + 1) + ". " + values[i]);
                    int sel = scanner.nextInt(); scanner.nextLine();
                    if (sel > 0 && sel <= values.length)
                        filter.setRoomType(values[sel - 1]);
                    break;
                case 3:
                    filter.reset();
                    System.out.println("All filters reset.");
                    break;
                case 0:
                    return;
                default:
                    System.out.println("Invalid option.");
            }
        }
    }
}
```

Open-Close Principle (OCP)

The **UserManager<T extends User>** interface demonstrates the Open-Closed Principle by:

1) Being **open for extension**: The generic type parameter allows any User subtype to be managed through this interface without requiring interface modification.

2) Being **closed for modification**: Existing code implementing this interface does not require changes when a new user (e.g Admin) is introduced or when specialized implementations (e.g AdminManager) are created.

New user management functionality can be added by creating new implementations rather than modifying existing code. This allows the system to evolve safely without risking introducing bugs in already functioning modules.

```
package control;

import entities.User;

public interface UserManager<T extends User> {

    void loadUsers();

    void saveUsers();

    List<T> getUsers();

    T findByNRIC(String nric);

    boolean changePassword(String nric, String newPassword);
}
```

Liskov Substitution Principle (LSP)

The LSP is observed in **User, Applicant, and Officer** hierarchy ensuring that subclasses can generally be used where their superclass is expected. An **Applicant** fulfills the **User** contract, inheriting common properties and behaviors like password verification without altering the expected outcome. Similarly, an **Officer** fulfills both the **User** and **Applicant** contracts for inherited functionalities, such as accessing basic user data and checking application status.

Interface Segregation Principle (ISP)

The ISP is also observed in our use of the **UserManager<T extends User>** interface. Rather than designing a fat interface with methods for every user function, we kept the interface focused and minimal, containing only the common user-related operations such as loading, saving, and changing password. Specialized behavior for each user type, such as officer registration or applicant booking, is handled in their respective classes without forcing other classes to implement methods they would not use. This ensures that each UserManager class remains focused on its own intended responsibilities.

Dependency Injection Principle (DIP)

While our system uses CSV files for data persistence, our **high-level modules such as LoginManager depend on abstractions which are UserManager<T> interface, rather than directly depending on low-level data handling details**. This separation allows core logic to remain independent of storage details, supporting maintainability and potential future changes (e.g. switching from CSV to a database) with minimal impact on high-level modules.

```java
public LoginManager(ApplicantManager applicantManager,
                    UserManager<Applicant> applicantUserManager,
                    UserManager<Officer> officerUserManager,
                    UserManager<Manager> managerUserManager,
                    EnquiryManager enquiryManager,
                    ApplicationManager applicationManager,
                    ProjectManager projectManager,
                    OfficerRegistrationManager officerRegistrationManager,
                    BookingManager bookingManager,
                    ReportManager reportManager,
                    FilterManager filterManager
                    ) {
```

4. Implementation

4.1 Tools used

Java 17, IDE: Eclipse/VisualStudioCode, Version control: GitHub

4.2 Sample Code Snippets - showing the 4 OO Concepts

4.2.1 Abstraction

Abstraction is shown by leveraging the UserManager interface to manage various types of users. Using generics (<T extends User>) allows the interface to work with other classes that extend the User class such as Officer. The OfficerUserManager class is one such example that provides customized implementations for all methods defined in the UserManager interface, tailored to handle officers. This allows the system to hide specific implementations of OfficerUserManager and expose only essential operations through UserManager.

```java
public interface UserManager<T extends User> {

    void loadUsers();

    void saveUsers();

    List<T> getUsers();

    T findByNRIC(String nric);

    boolean changePassword(String nric, String newPassword);
}
```

```java
public class OfficerUserManager implements UserManager<Officer> {

    @Override
    public List<Officer> getUsers() {
        return officers;
    }

    @Override
    public Officer findByNRIC(String nric) {
        for (Officer officer : officers) {
            if (officer.getNRIC().equalsIgnoreCase(nric)) {
                return officer;
            }
        }
        return null;
    }

    @Override
    public boolean changePassword(String nric, String newPassword) {
        Officer user = findByNRIC(nric);
        if (user != null) {
            if (user.changePass(newPassword)) {
                saveUsers();
                return true;
            } else {
                return false;
            }
        }
        System.err.println("Officer not found for password change: " + nric);
        return false;
    }
}
```

4.2.2 Encapsulation

Encapsulation is demonstrated in our **ReportManager** class, where the control classes ApplicantUserManager and OfficerUserManager are declared as private final, restricting external access to the data structures and only through specific methods. The FilterCriteria() method in ReportManager also encapsulates filtering options for reports and is declared as static, ensuring that it cannot be directly altered by other classes.

```java
public class ReportManager {

    private final UserManager<Applicant> applicantUserManager;
    private final UserManager<Officer> officerUserManager;

    public ReportManager(UserManager<Applicant> applicantUserManager, UserManager<Officer> officerUserManager) {
        if (applicantUserManager == null || officerUserManager == null) {
            throw new IllegalArgumentException(s:"UserManagers cannot be null.");
        }
        this.applicantUserManager = applicantUserManager;
        this.officerUserManager = officerUserManager;
    }

    public static class FilterCriteria {
        private Boolean maritalStatusFilter = null; // true=Married, false=Single, null=All
        private RoomType roomTypeFilter = null; // Specific RoomType or null=All

        public Boolean getMaritalStatusFilter() {
            return maritalStatusFilter;
        }
        public void setMaritalStatusFilter(Boolean maritalStatusFilter) {
            this.maritalStatusFilter = maritalStatusFilter;
        }
        public RoomType getRoomTypeFilter() {
            return roomTypeFilter;
        }
        public void setRoomTypeFilter(RoomType roomTypeFilter) {
            this.roomTypeFilter = roomTypeFilter;
```

4.2.3 Inheritance

The Officer class is a subclass of the Applicant class, which is a subclass of the parent class User. Below, we can see that Officer inherits the attributes and methods of Applicant with added methods and functionalities.

```java
public class Applicant extends User{
    private Project appliedProject;
    private RoomType chosenRoom;
    private ApplicationStatus status;

    public Applicant(String name, String NRIC, int age, boolean isMarried, String password) {
        super(name, NRIC, age, isMarried, password, Role.APPLICANT);
    }

    // Getters and Setters
    public Project getAppliedProject() {
        return appliedProject;
    }

    public void setAppliedProject(Project appliedProject) {
        this.appliedProject = appliedProject;
    }

    public RoomType getRoomChosen() {
        return chosenRoom;
    }

    public void setRoomChosen(RoomType flatTypeChosen) {
        this.chosenRoom = flatTypeChosen;
    }

    public ApplicationStatus getStatus() {
        return status;
    }
```

```
Login Successful!
Welcome John!
Signed in as Applicant.


========== Applicant Dashboard ==========
 User: John (S1234567A)
-----------------------------------------
 1. Change Password
 2. View Available BTO Projects
 3. Apply for a Project
 4. View My Application Status
 5. Withdraw My Application
 6. Submit an Enquiry
 7. View/Edit/Delete My Enquiries
 8. View My Profile
 9. Logout
=========================================
```

```java
public class Officer extends Applicant {

    private List<Project> registeredProjects;

    private Map<Project, OfficerRegistrationStatus> registrationStatuses;

    public Officer(String name, String NRIC, int age, boolean isMarried, String password) {
        super(name, NRIC, age, isMarried, password);
        this.registeredProjects = new ArrayList<>();
        this.registrationStatuses = new HashMap<>();
    }

    public void addRegisteredProject(Project project, OfficerRegistrationStatus status) {
        if (!registeredProjects.contains(project)) {
            registeredProjects.add(project);
            registrationStatuses.put(project, status);
        }
    }

    public List<Project> getRegisteredProjects() {
        return registeredProjects;
    }

    public OfficerRegistrationStatus getRegistrationStatusForProject(Project project) {
        return registrationStatuses.get(project);
    }

    public void updateRegistrationStatus(Project project, OfficerRegistrationStatus status) {
        if (registeredProjects.contains(project)) {
            registrationStatuses.put(project, status);
        }
    }
}
```

```
Login Successful!
Welcome Emily!
Signed in as HDB Officer.


========== HDB Officer Dashboard =========
 User: Emily (S6543210I)
-----------------------------------------
----- Applicant Actions (As Officer) -----
 1. Change Password
 2. View Projects
 3. Apply for a Project
 4. View My Application Status
 5. Withdraw My Application
 6. Submit an Enquiry
 7. View/Edit/Delete My Enquiries

------------ Officer Actions -------------
 8. Register to Handle a Project
 9. View My HDB Officer Registration Status
10. View Details of Project I Handle
11. View and Reply to Project Enquiries
12. Book Flat for Successful Applicant
13. Generate Booking Receipt for Applicant
14. View My Profile
15. Logout
=========================================
```

4.2.4 Polymorphism

Polymorphism is demonstrated when our **HDBOfficerUI** class **overrides** methods from the **ApplicantUI** class to provide behaviours specific to officers, while retaining the exact method signatures. Below shows part of the displayProjectListWithRooms() method, where HDBOfficerUI includes a visibility status 'ON' or 'OFF' as Officers are able to see projects available even if the visibility status is 'OFF', while Applicants are not able to see it.

```java
public class ApplicantUI {
    protected void displayProjectListWithRooms(List<Project> projects) {

            // Format dates
            String openDate = p.getOpenDate() != null ? p.getOpenDate().format(dateFormatter) : "N/A";
            String closeDate = p.getCloseDate() != null ? p.getCloseDate().format(dateFormatter) : "N/A";

            System.out.printf(formatStr,
                    p.getName(),
                    p.getNeighbourhood(),
                    roomInfo.toString(),
                    openDate,
                    closeDate);
        }
    }
    System.out.println(x:"=================================================
    }
```

```
public class HDBOfficerUI extends ApplicantUI {
    protected void displayProjectListWithRooms(List<Project> projects) {
            roomInfo.append(str: "<NO room info>");
        }
        // Format dates
        String openDate = p.getOpenDate().format(dateFormatter);
        String closeDate = p.getCloseDate().format(dateFormatter);
        // Add visibility status
        String visibilityStatus = p.isVisibility() ? "ON" : "OFF";

        System.out.printf(format:" %-20s | %-15s | %-30s | %-12s | %-12s | %-10s%n",
            p.getName(),
            p.getNeighbourhood(),
            roomInfo.toString(),
            openDate,
            closeDate,
            visibilityStatus);
        }
    }
    System.out.println("=".repeat(count:115));  // Adjust based on total width
}
```

5. Testing

5.1 Test Strategy

We used manual functional testing, acting as a user to go through all the features in the system. We then listed all the noted bugs and improvements in a google document for easy reference.

5.2 Test Case Table

View our complete list of test cases from our link here: **testcases**

6. Reflection & Challenges

6.1 Reflection

Working on the BTO Management System strengthened our grasp of object-oriented design and programming (OODP), particularly the application of SOLID principles to build a more maintainable system. One of the things that went well was our use of a **layered architecture**, which helped us separate user interface logic from application control and data management. This made our system easier to scale and debug. We also successfully implemented features like role-based user flows, persistent CSV data storage, and user validation, all of which gave us valuable experience in handling real-world constraints such as eligibility logic and data consistency. Overall, the project improved our technical skills and team collaboration.

6.2 Challenges

1) **Passing on the code for editing**

   As the system expanded, navigating between interconnected classes became more

difficult, making code handoff challenging. To address this, we maintained consistent naming conventions, held regular code walkthroughs and clarified method responsibilities through comments in the code.

2) Consistent File I/O Across Roles

Managing persistent data using CSV files for multiple user types introduced complexity. We overcame this by encapsulating file login within role-specific manager classes, allowing them to handle their own CSV parsing while ensuring uniform structure.

3) Avoiding Feature Conflicts and Logical Overlaps

We encountered edge cases like officers trying to register to handle projects they are also applied to as applicants. We resolved this by implementing additional validation logic in OfficerRegistrationManager, preventing conflicts at the point of registration.

6.3 Further Improvements

1) Introduce proper interface-based abstractions

We could define interfaces for Manager classes (e.g ProjectManager, ApplicationManager etc), to fully follow the Dependency Inversion Principle.

2) Add automated unit testing

We could use an open source framework such as JUnit to ensure that individual units of code work as expected, helping with early bug detection and improving code quality.

3) Refactor large UI classes

Break the boundary classes (ApplicantUI, HDBOfficerUI, HDBManagerUI) into smaller components to improve readability and align with the Single Responsibility Principle.

4) Add more filters to improve user experience

Currently, the viewing filters only have 2 options-Neighbourhood and Type of room. We aim to expand the filter to include options like budget range or completion year.

5) Replace CSV files with a database

This enables better scalability in the future when there are more users of the system. This also reduces file I/O complexity and improves efficiency.

7. Appendix

Github Repository: https://github.com/jannxle/SC2002-FDAD-GRP3-.git

Please follow instructions in the **README.md** to run our project from terminal.