

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

**GROUP ASSIGNMENT**

**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 the 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** |
| --- | --- | --- | --- |
| **THUM MUN KUAN** | **SC2002** | **FCS2** | **24/4/2024** |
| **VO TRUONG THIEN KY** | **SC2002** | **FCS2** | **24/4/2024** |
| **WONG SHI HUI CHERYL** | **SC2002** | **FCS2** | **24/4/2024** |
| **ZHOU YINGLI** | **SC2002** | **FCS2** | **24/4/2024** |

**Chapter 1: Analysis & Feature Selection**

**1.1 Understanding Problem and Requirements**

The core problem domain centers around managing BTO public housing projects in Singapore. After reading through the requirements document, our group identified the several explicit requirements to implement, together with user roles and system entities as shown in the next part. We analysed that the BTO system needed to serve multiple stakeholders with different needs.

**1.2 Deciding on Features and Scope**

**Core Features (Essential):**

1. User Authentication and Management: user login/logout and role based access control to establish the multi-user system with different permissions.
2. Project Management: project creation with details, visibility control and applicant eligibility criteria, essential to implement core concepts and demonstrating aggregation relationships.
3. Application Processing: submitting application, checking status, submitting withdrawal, approval/rejection by manager. This represents the primary business process of the system and demonstrates state management.
4. Officer Management: registration, approval/rejection by manager for the complete administrative workflow and demonstrates inheritance hierarchy.
5. Enquiry System: submission and editing of enquiry, response function serving as an important communication channel between applicants and administrators.

**Optional Features (if time permits):** statistical summaries of applications/registrations/reports, manual assignment of flat projects to officers

**Excluded Features (out of scope, insufficient time):** automated balloting system, payment management, email/SMS notifications

**1.3 Assumptions Made**

In Application Workflow Details, the exact steps in the application approval process weren’t fully specified so we implemented a simple workflow with states (PENDING, APPROVED, REJECTED, WITHDRAWN) and appropriate transitions.

In the Flat Allocation Process, the detailed process for allocating specific flats to approved applicants wasn’t specified so we implemented a basic booking system where officers can assign flats to approved applicants.

**Chapter 2: System Structure Planning**

**2.1 Planning the System Structure**

To start, we used a top-down decomposition approach with the system requirements, user flows and functional responsibilities, and organized components around feature sets/domains like Applicant, HDB Officer and HDB Manager. In addition, our group decided to implement the MVCS model as well. We were able to determine the different overarching responsibilities of each class. That is,

1. **View Layer**: Contains all user interfaces for different types of users (Applicants, HDB Officers, and HDB Managers).

* Dashboard views for each user type (ApplicantDashboard, OfficerDashboard, HDBManagerDashboard)
* Specialized views for operations like registration, application review, and enquiry handling
* Login view for authentication

2. **Controller Layer**: Acts as an intermediary between views and services, handling input validation and orchestrating workflows.

* User-specific controllers (ApplicantProjectController, HDBManagerEnquiryController)
* Cross-cutting controllers (AuthenticationController)
* Operation-focused controllers (OfficerFlatBookingController, HDBManagerApplicationController)

3. **Service Layer**: Implements core business logic and enforces business rules.

* Services for each domain area (HDBManagerEnquiryService, ApplicantProjectService)
* Contains complex operations and validation logic

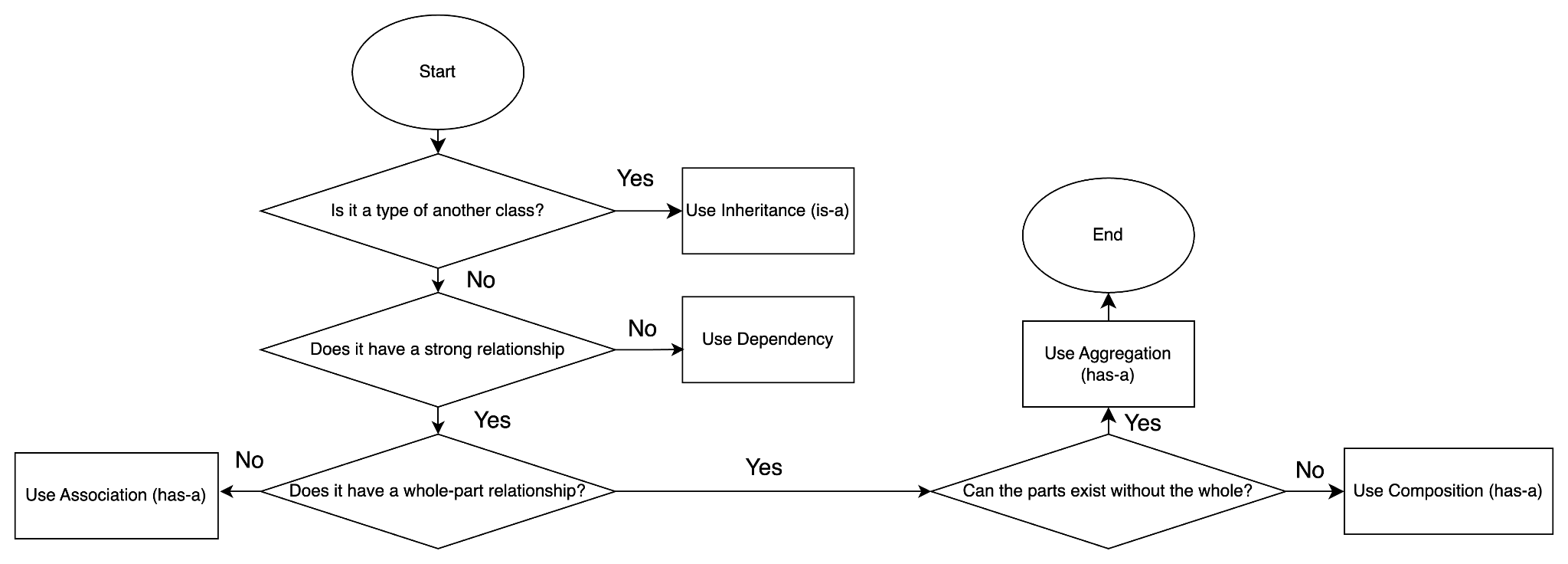
4. **Repository Layer**: Manages data persistence and retrieval operations.

* Entity-specific repositories (ApplicantRepository, ProjectRepository)
* Encapsulates data access concerns

5. **Model Layer**: Defines the domain entities and their relationships.

* Core domain objects (Applicant, Project, Enquiry, Application)
* User hierarchy (User – parent; Applicant, HDBOfficer, HDBManager – subclasses)

To support early design decisions, we created visual guides like flowcharts to help determine appropriate relationships between classes. One key diagram we used was a relationship decision flowchart(Fig 1). It helped us decide whether to use inheritance (is-a), dependency, or association types such as aggregation and composition based on the strength and nature of the relationships. This ensured we consistently applied object-oriented principles and chose the right modeling technique for each interaction. For instance, we used composition where components could not exist independently (e.g., an Application within a Project), and inheritance to define role-based behavior for different user types (e.g. an Applicant and a HDBOfficer)

Fig 1: Relationship Decision Table

To bridge functionality with structure, we first analyse user flows and common actions for each user role. For example, for an Applicant, the journey typically involved viewing available projects, filtering by preferences, and submitting an application(Fig 2). These user flows were used to draft flowcharts that helped visualize the sequential logic and conditional branches in each process. These early visual models helped us validate that we covered all necessary steps and accounted for exceptions like invalid input or project capacity limits.

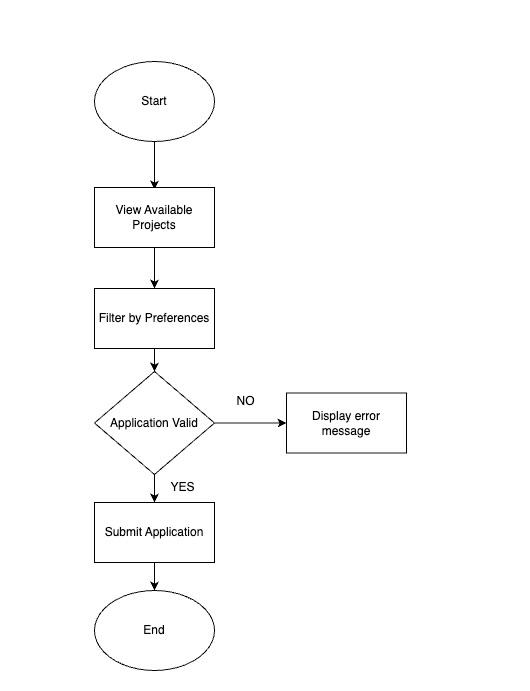


Fig 2: Sequential Logic Flowchart

Therefore by combining use-cases and our MVCS models, we started finding functional relationships between the classes and created several main application flows. For example

BTO Project Application Flow

ApplicantProjectView → ApplicantProjectController → ApplicantProjectService → ProjectRepository/ApplicationRepository → Project/Application

HDB Officer Registration for Project Flow

OfficerRegistrationView → OfficerRegistrationController → OfficerRegistrationService → OfficerRepository/ProjectRepository → HDBOfficer/Project

HDBManagerRegistrationView → HDBManagerRegistrationController → HDBManagerRegistrationService → OfficerRepository → HDBOfficer

Application Approval Flow

HDBManagerApplicationView → HDBManagerApplicationController → HDBManagerApplicationService → ApplicationRepository → Application

This step helped us to visualise and figure out how each process should flow in our program and helped us with the design and implementation of code.

**2.2 Reflection on Design Trade-offs**

The group has 3 main design trade-offs that are the most impactful in the project.

**Centralized vs. Distributed Logic**: placing business logic in models makes them self-contained but can lead to bloated classes, hence we moved complex logic operations to service classes while retaining basic validation in the models.

**Repository Pattern**: The repository pattern adds a layer of indirection but it allowed database integration of our system and made testing easier.

**General vs. Role-Specific Controllers**: while reusable components reduce code duplication, we opted for tailored controllers for each user role to maintain clarity and isolate logic.

These choices were made with clarity, scalability, and adherence to OODP principles in mind.

**Chapter 3: Object Oriented Design**

**3.1 Class Diagrams**

After our analysis of the requirements and user flows outlined in Chapters 1 and 2, we focused on creating a class structure that related closely to both real-world responsibilities as well as align them with our MVCS model. [**Class diagram is in separate file attached**]

1. Identifying Key Classes

Our initial steps involved identifying the noun and deducing the roles. We also asked ourselves questions involving who, what, where, when and how, to ensure that our classes aligned with the requirements. Some prominent entities include:

* User: Base class for all users
* Applicant: Users applying for projects
* HDBOfficer: Act as an officer who can both register to handle project and apply as applicant
* HDBManager: Manages project creation and overall system settings
* Project: BTO Project
* Flat: Specific flat in a project
* Application: An application submitted by a user for a project
* Enquiry: A question raised by user

1. Class Responsibilities

Hence our group decided on implementing a **MVCS (Model-View-Controller-Service)** approach to further dissect the interactions between the classes. The system was divided into four main layers. For views, by having separate dashboards ApplicantDashboard, HDBOfficerDashboard, HDBManagerDashboard to show all the available features of each role and having a View for every function. For controllers, we have controllers for each function that the project requires such as ApplicationController, ReportController, HDBManagerProjectController, etc. this is to act as a bridge between the view and our model and handles business logic. For models, we have Project, Applicant, Application, ProjectApplication etc to store and load data.

1. Class Relationship

We utilised the relationship decision flowchart (see Fig 1) to determine the relationship between the different layers.

**3.2 Sequence Diagrams**

In accordance to the requirements provided, we determine 2 crucial use cases [**Sequence diagram is in separate file attached**], that is,

**1. Registration Process for HDB Officer**

1. Validates conditions such as existing BTO applications to ensure that it does not overlap with the registering project
2. Verify that the officer is not handling another project during the same application period
3. Involves the utilisation and collaboration across of multiple layers:
   1. HDBOfficerProjectView: Display the option to register as an HDB Officer of a project, receive and validate information required
   2. HDBOfficerRegisterController: Receive the registration request and performs the business logic (e.g. check for eligibility)
   3. HDBOfficerRepository/ProjectRepository: Check respective data in database
4. Involves approval by HDB Manager and status transitions, demonstrating how different actors interact in the system
5. Update the project’s available slots and the HDB Officer’s profile one approved

**2. Application Process**

1. Consist of user eligibility checks based on marital status and age
2. Checking of visibility and toggling of visibility at the control layer
3. Check for existing application, adhering to the application constraints
4. Demonstrates application status transitions
5. Final booking through HDB Officer
6. Update project flat counts, applicant profiles and generate receipts

**3.3 Application of OOD Principles (SOLID)**

1. **Single Responsibility Principle**

In our system, we ensured that each class had one distinct role, ensuring that they only had one reason to change. For example, we separated each class into distinct MVCS layers. Moreover, we separated enquiry-related logic into three distinct classes: ApplicantEnquiryController, HDBOfficerEnquiryController and HDBManagerEnquiryController. Applicants can submit, edit and delete their own enquiries, HDB Officers can view and reply to enquiries for the projects they handle and HDB Managers can view all enquiries and respond to those they are handling.

1. **Open-Closed Principle**

We kept in mind that classes should be open for extension but closed for modification, allowing the functionality to be extended without changing the code. An example of this principle would be the use of the generic User abstract class. Applicant, HDBOfficer and HDBManager are subclasses of this parent class, inheriting common attributes and behaviours. Thus allows the system to be open for extension, should new user types be required to be added, but closed for modification as existing User logic remains unchanged. For example, if a new Assistant class is introduced, it can extend User without altering existing classes.

1. **Liskov Substitution Principle**

We noted that superclass should be replaceable with objects of a subclass without affecting the program. Wetook this into consideration when designing our FileHandler class, ensuring that the CSVFileHandler subclass is interchangeable for FileHandler. Keeping in mind that preconditions should not be stronger than the base method and post conditions are no weaker than the base method. This promotes polymorphism, allowing methods from FileHandler to function correctly regardless of the specific subclass.

1. **Interface Segregation Principle**

We kept in mind that classes should not be forced to implement methods it does not use. Interfaces should be specific and tailored to the needs of the class. For example, instead of having a general interface for all HDB Officer abilities, we segregated the interfaces into IHDBOfficerRegistrationController, IHDBOfficerFlatController and IHDBOfficerEnquiryController. These interfaces would then include its own methods as required. This avoids forcing unrelated responsibilities on classes.

1. **Dependency Injection Principle**

We made it a point to have high level modules depend on abstractions (interfaces) rather than directly on low level modules. For example, for ApplicantProjectView, it will depend on IApplicantProjectController rather than directly depending on ApplicantProjectController. THis can help to decouple the view layer from specific controller implementations to make the system more maintainable and testable. This made it possible to test the views in isolation by providing a mock controller and allowed us to swap controller implementations without changing the view.

**Extensibility and Maintainability of Our System**

Our BTO system is designed with future growth in mind, enabling future improvements with minimal cost and effort.

**Extensibility**

Firstly, following OCP, our system allow new features (eg. flat types, BTO projects) or user typesto be added without having to modify existing code. Secondly, following ISP, we created role-specific interfaces, we were able to keep the functionalities modular and easier to extend. Thirdly, following DIP, we allowed views to depend on interfaces rather than concrete controllers, making it easier to swap or add new implementations.

**Maintainability**

The BTO system is also easy to understand, and allows for quick debugging and updating. Abstraction is used throughout to organize logic into manageable components (eg. abstract base classes and clean interfaces), making our code easier to understand and modify.

Following SRP, the clear separation of UI, logic, and data ensures updates in one does not affect another. Unit testing and debugging can hence be done efficiently as features like filtering, applications, and user management are handled separately, making the system scalable while easy to maintain as requirements evolve.

**Chapter 4: Implementation**

**4.1 Tools Used:**

* Java 17
* IDE: Visual Studio Code
* Version Control: Github

**4.2 Sample Code snippet:**

**Abstraction**

To separate what each class does from how it does it, we used abstraction. It also helps to keep our system modular and easier to extend without changing existing code.

Key abstractions include

* Defining clear interfaces between layers: controllers interact with repositories through method calls without needing to know how data was stored or retrieved (e.g., whether it used in-memory structures or CSV files), making the system easier to manage and allowed changes to internal implementations without affecting other components.
* Meaningful method names: applyForProject(), filterProjects(), or approveRegistration() hides internal logic and makes the code easier to understand at a high level.

**Encapsulation**

We applied encapsulation to all Model classes to protect sensitive user data and control access across the system. For example the User class, personal attributes like NRIC and age are declared as private and are only accessible via getters/setters. This preserves data integrity by preventing direct modification, and protects user privacy through access control.

public abstract class User{

private String nric;

… (other attributes)

public String getNRIC() {

return nric;

}

… (other methods)

}

**Inheritance**

To handle the 3 distinct user types Applicant, HDBOfficer, and HDBManager, we use inheritance. In our code, Applicant and HDBManager extends from the User superclass, while HDBOfficer extends Applicant. Each subclass incrementally adds their own role-specific behaviour. This structure promotes code reusability and logical grouping of shared behaviour, while keeping unique responsibilities separated.

//Base User class

public abstract class User{...}

public class HDBManager extends User{...}

public class Applicant extends User {...}

public class HDBOfficer extends Applicant {...}

**Polymorphism**

Our system uses both static and dynamic polymorphism to allow flexibility of the different user types which perform the same actions but in different ways. For example in the Dashboard classes, we will have a base class MainDashBoard

public class MainDashBoard {

Public void displayDashBoard(){};

}

public class ApplicantDashBoard extend{

@Override

Public void displayDashBoard(){...};

}

public class HDBOfficerDashBoard {

@Override

Public void displayDashBoard(){...};

}

public class HDBManagerDashBoard {

@Override

Public void displayDashBoard(){...};

}

**Interface**

Our system uses interfaces to define contracts and enable loose coupling. Here’s an example of interface usage between View and Controller layer:

public

**Chapter 5: Testing**

| **TC-** | **Test case Description** | **Expected Output** | **Passed (Y/N)** |
| --- | --- | --- | --- |
| 1 | Login with correct password | Redirected to Appropriate dashboard | Y |
| 2 | Login with wrong password format | Error message displayed | Y |
| 3 | Login with wrong password | Error message displayed | Y |
| 4 | Change Password | Re Login with new password | Y |
| 5 | Project visibility | Project are visible based on User Group and Visibility | Y |
| 6 | Project Creation | Project created successfully (Visibility Off by default) | Y |
| 7 | Project Edit | Project edited successfully | Y |
| 8 | Project Delete | Project deleted successfully | Y |
| 9 | Project Toggle Visibility | Project visibility toggle success | Y |
| 10 | Project Application (for Applicant) | Applicants can apply for relevant projects (is opening, is open…) | Y |
| 11 | Project Application (for Officer) | Officers can apply for projects that he’s not handling and relevant. | Y |
| 12 | Viewing Application Status (After Visibility Off) | Officers/Applicants can view application details regardless of project visibility | Y |
| 13 | Flat Booking (From Applicant) | Only Book when Status is Successfully, Only one Flat Booking | Y |
| 14 | Register to handle project (Officer), View Registration Status | Officer can only register for project when he is not handling other project in the same period time, Officer can not register for project he is applying for as an Applicant | Y |
| 15 | Project Details Access for Officer | Officer can access project details regardless of visibility | Y |
| 16 | Flat Booking Management (From Officer) | Officer can help Applicant book flat | Y |
| 17 | Manager Approve/Reject Application | System update Application status correctly | Y |
| 18 | Manager Approve/Reject Registration | System update Registration status correctly | Y |
| 19 | Manager can view/filter list of projects | Filter all projects | Y |
| 20 | Manager can generate report | Can generate report based on filter | Y |
| 21 | Applicant submit Enquiries | Submit successfully | Y |
| 22 | Applicant view Enquiries reply | Can view | Y |
| 23 | Officer view pending Enquiries | Can view | Y |
| 24 | Officer reply Enquiries | Can reply | Y |
| 25 | Manager view pending Enquiries | Can view | Y |
| 26 | Officer reply pending Enquiries | Can reply | Y |

**Chapter 6: Javadoc**

[submitted in separate file]

**Chapter 7: Reflection**

This project provided a valuable opportunity to apply OODP principles to a real-world problem. By implementing concepts learned, we were able to build a modular, maintainable system that mirrors actual processes in Singapore’s BTO flat booking.

We divided responsibilities among the team, with members taking charge of a different component in the MVCS model. However, we faced difficulties in keeping method signatures consistent, and to let our code function well when put together. Therefore, we prioritized finalizing a UML class diagram that defined shared attributes, method names, and class relationships. This allowed each team member to work independently while ensuring consistency when referencing each other’s methods, minimizing integration issues. We also relied on GitHub to collaboratively track changes and understand class dependencies.

Our final system successfully implemented all the core features. We also adopted several useful design patterns, such as the repository pattern for data access, MVC architecture, and polymorphism for dynamic dashboard rendering. The system is built with distinct role permissions and robust error handling, and we ensured that inputs and flows were validated at the controller and view levels.

While we achieved most of what we planned, there were areas for improvement. For instance, the current file-based data persistence system works for a prototype, but a database-backed solution would better support scalability and data consistency. Our test coverage focused on core functionality and edge cases such as age eligibility, but additional integration and stress testing could further enhance system reliability. Additionally, further breaking down some classes into smaller, more specialized components would help us better adhere to principles like SRP and ISP.

Through this project, we learned the importance of planning ahead. Investing time in thoughtful class design saved us effort later in debugging and restructuring. We also came to appreciate the balance between inheritance and composition, and how abstraction enabled greater flexibility. Encapsulation played a vital role in safeguarding user data and enforcing business rules, while polymorphism allowed us to support multiple user roles through shared interfaces. This project showed us that good design not only makes implementation smoother but also creates a strong foundation for future extensibility and maintenance.