**Project Title: HDB Management System**

|  |  |
| --- | --- |
| Name | Matriculation number |
| Yeo Wen Hong | U2421146B |
| Peng Zixiao | U2421551L |
| Wong Xiao Yao | U2421644E |
| Zou Ning | U2421398A |

**Chapter 1: Requirement Analysis & Feature Selection**

* 1. **Understanding the Problem and Requirements**

We began by reading through the BTO assignment specification line by line, systematically highlighting all system requirements and use cases. This analysis allowed us to identify the core problem domain: building a role-based BTO (Build-To-Order) Management System that allows Applicants, Officers, and Managers to interact with BTO projects through a centralised digital hub.

**Main Problem Domain**

The system simulates Singapore’s HDB BTO process in a digital environment. It must allow different types of users to: 1. Log in via NRIC-based credentials. 2. Manage or apply for BTO projects based on their roles. 3. Interact through role-specific functions (e.g., application, registration, booking, approval). We identified the system as one involving multiple user roles, with clearly defined capabilities and restrictions, interacting with shared entities such as Projects, Applications, Enquiries, and Registrations.

**Explicit Requirements**

**Login and User Management**

NRIC-based login with a default password (*password*), password change functionality for all users, data can be taken directly from files provided.

**Applicant Capabilities**

View eligible and visible BTO projects. Submit, withdraw, and view the status of applications. Submit, edit, and delete project-related enquiries. View flats booked.

**Officer Capabilities**

All applicant features. Register to handle a project. View registration status. View and reply to enquiries responsible project. Manage bookings and flat availability. Generate receipts and reply to enquiries.

**Manager Capabilities**

Create, edit, delete BTO projects, toggle visibility of projects, approve/reject officer registrations and applicant applications/withdrawals. Generate filtered booking reports. View and respond to project enquiries.

**Implicit Expectations**

Applications must be rejected regardless of Officer’s approval when the flat is unavailable. Registrations must be rejected when officer slots are full regardless of Manager’s approval. Pending registrations must be rejected regardless of Manager’s approval when one with overlapping dates has been approved.

**Interpretation of Ambiguities**

Once an application is successful, we were unsure of whether the applicant has to manually request to book a flat. For simplicity, an applicant just has to wait for the officer to process the successful application and the application status will be changed to “Booked” automatically.

By carefully dissecting the assignment document, we ensured that our design adhered to both the explicit functional requirements and the implicit business rules of the BTO system. This foundation guided the creation of our user roles, domain entities, access controls, and system architecture, allowing us to design a cohesive and well-structured application.

**1.2 Deciding on Features and Scope**

We grouped features into three categories: core, optional, and excluded. Our goal was to ensure that they demonstrated strong OO principles without making the system overly complex within the given timeline. We considered the explicit system capabilities required for each user role as outlined in the assignment brief alongside feasibility and alignment with OO best practices.

**Core Features (Essential, must implement)**

**Authentication & User Management**

NRIC-based login with default and changeable password**.** Role-based access to menu and functionalities**.** User attributes: age, marital status, and role.

**Applicant Functionalities**

View visible projects by marital status and age restrictions. Submit and withdraw BTO applications. Book flat after successful application (by the Officer). View current application status, even after project visibility is off. Request withdrawal of application at any status. Submit, edit, and delete enquiries. Filtering system for project viewing.

**Officer Functionalities**

All applicant capabilities inherited. View involving projects, registration status (as a Officer) and manage flat availability. Register to handle a project (depends on whether project dates clashes). Retrieve, update and approve applicant booking status. Generate flat booking receipt. Filtering system for project viewing.

**Manager Functionalities**

Create, edit, and delete BTO projects**.** Toggle project visibility Filter and view projects created by themselves or others**.** Approve/reject Officer registration and Applicant applications**.** Approve/reject withdrawal requests**.** Generate filtered reports on flat bookings**.** View and respond to enquiries for their own projects.

**Data Management and Persistence**

Centralised repositories for each entityfor seamless runtime access to in-memory data.

**CLI Interface**

Menu-driven UI per user roleforeasy navigation and text-based interaction.

**Optional or Bonus Features**

Once an application is accepted, all other pending applications are automatically rejected when there are no more available units. Once the officer slots for a project are full, registrations are rejected automatically. Once a registration is approved, pending registrations for other projects with an overlapping application window by the same officer will be rejected automatically.

**Excluded Features**

File reading and utilised repository classes to store data. Multi-language support or accessibility options.

**Chapter 2: System Architecture & Structural Planning**

**2.1 Planning the System Structure**

Before implementation began, we spent time conceptualising the overall system layout to ensure a clean, modular, and extensible architecture. Our planning followed the principles of object-oriented thinking, which helped us break down the system into meaningful components and interactions.

**Breaking Down the System into Logical Components**

We divided the system into three main layers to reflect the MVC (Model-View-Controller) pattern:

* **Entity Layer (model)**: This includes domain classes such as *Applicant, Officer, Manager, Project, Application, Enquiry, Registration* etc. Each class models real-world attributes and encapsulates relevant data.
* **Controller Layer (controller)**: Controllers like *ApplicantController, OfficerController*, and *ManagerController* handle role-specific logic and mediate between the UI and data repositories. ProjectController, EnquiryController and RegistrationController were also implemented to manage communication between different users.
* **Boundary Layer (menu)**: Separate menu classes such as *ApplicantMenu, ManagerMenu*, and *OfficerMenu* handle user interaction. Each menu displays role-specific menus and captures user input to trigger the appropriate controller logic.
* **Repository Layer**: We also included *Repository* classes for each entity type class to store necessary information updated while the program is running

**Modeling User Flows and Mapping Use Cases**

We started by analysing the specification and identifying the main user flows and use cases for each role:

* Applicants: logging in → viewing eligible projects → submit/edit enquiries → submitting an application → checking status → booking → withdrawal/book flats.
* Officers: registering for a project → replying to enquiries → managing bookings → generating receipts.
* Managers: creating/editing/deleting a project → handling officer registration → approving applications and withdrawals → generating reports.

These flows guided how we distributed responsibilities among the controllers and reinforced our separation of concerns.

**Creating Early Visual Models**

While we did not use detailed UML activity or flow diagrams in the initial stages, we sketched rough flowcharts on paper and shared them within the team to ensure everyone understood the process logic. This included step-by-step outlines of login, application submission, booking logic. Moreover, we put much consideration into corner cases such as how the application should behave when flat availability reaches zero etc. The early planning allowed us to:

* Identify controller boundaries
* Prevent redundant responsibilities across classes
* Ensure role-based behaviours were properly isolated

**2.2 Reflection on Design Trade-offs**

During the planning phase, we had several discussions around how best to structure the system while balancing simplicity, modularity, and extensibility. Our goal was to create a maintainable system that adhered to object-oriented principles without overcomplicating the architecture.

**Trade-offs Considered**

* **Controller vs. Logic Layer Separation**: We initially considered merging the controller logic directly into the Menu or entity classes to simplify the architecture. However, we chose to keep a clear separation between the Menu, Controller, and Entity layers. This creates additional code but greatly improved maintainability, testing, and modular growth.
* **Role-Specific Menu vs. Shared Interface**: We debated whether to create a single unified Menu class with conditional logic based on user roles. In the end, we opted to implement separate Menu classes for each role (ApplicantMenu, OfficerMenu, ManagerMenu). This decision increased duplication but resulted in cleaner role-specific menus and logic separation.
* **Flat Booking Logic Encapsulation**: Initially, we thought of placing booking logic inside the Officer class. We later moved it into the OfficerController to better reflect real-world authority and improve system coherence.

**Team Discussions and Decisions**

As a team, we frequently debated between code simplicity and future-proof design. In most cases, we prioritised clarity, modularity, and separation of concerns, especially when different roles had distinct responsibilities and restrictions. In the end, these design choices helped us produce a system that not only fulfilled all requirements but could also be adapted more easily for additional features or future enhancements.

**Chapter 3: Object-Oriented Design**

**3.1 Class Diagram**

**How were the main classes identified?**

We started by identifying nouns and roles in the problem specification. From this, we derived key entity classes:

* Applicant, Officer, and Manager were extracted based on distinct user roles.
* Project, Application, Registration, Enquiry were derived from recurring terms in the user flows and system operations.
* UserType, RegistrationStatus, and ApplicationStatus were implemented using enums.

**What are the responsibilities of each class?**

* Applicant, Officer, Manager: Represent user accounts with identity and role-based capabilities.
* Project: Holds project-specific data such as flat types, visibility, and officer slots.
* Application: Records an applicant’s intent to book a flat and tracks its status.
* Registration: Manages officer applications to projects, including status tracking.
* Enquiry: Enable communication between users regarding projects.
* Repositories: Acts as databases for each entity class
* Controllers: Handle business logic based on user actions (e.g., approving applications, generating reports).
* Menu classes: Display role-specific menus and interact with the user.

**How were relationships determined?**

**Inheritance**:

*Officer* inherits from *Applicant* since they share base attributes (e.g., NRIC, password, marital status) but have additional privileges. Similarly, Manager, Officer and Applicant inherit from User.

**Association**:

*Applicant* is associated with one or more *Application* objects. *Manager* is linked to the *Project* they created. *Enquiry* is linked to both the *Applicant* and the *Project*.

**Composition**:

*Project* composites *UnitType* availability and is associated with *Application* and *Registration*.

**We frequently asked ourselves:**

* *"Should this concept be an attribute or a full class?"* Example: We made Application a separate class to encapsulate statuses and dates instead of keeping them as attributes of Applicant.
* *"Is this an is-a or has-a relationship?"* Example: Officers and Managers are modeled as special types of Applicants (is-a), while Applicants “have” Applications (has-a).

**What trade-offs were considered?**

Firstly, we considered Simplicity vs. Extensibility: We avoided combining roles into a single user class with role flags, choosing instead to use inheritance for clarity and flexibility. Secondly, we considered encapsulation vs. Duplication: Having separate Menu classes introduced some duplication, but it kept the responsibilities and user flows cleanly separated.

**Final Output:** The final output of the UML class diagram has been attached.

**3.2 Sequence Diagrams**

**Design Thinking Justification: Why These Two Scenarios?**

**Officer registers to handle a project**

This scenario demonstrates a role-restricted and condition-driven flow where an HDB officer attempts to register for a project. It reflects real-world constraints such as:

* Date validation: The project must be within its opening and closing window.
* Project eligibility check: Officers cannot register for overlapping projects.
* Dual-role restrictions: Officers cannot have applied as applicants to other projects.

It exercises controller-service-model interactions, enforcing multi-layer separation, and showcases how role-based access and time-sensitive constraints are tightly integrated into our system.

**Officer Applies for a Flat as an Applicant**

On top of checking for eligible projects that he can apply as an applicant (age and marital status), his intention to register for the same project was also checked to determine his eligibility.

The scenario highlights how business logic combines user profile data, project timing, and role assignments to govern application eligibility. It also demonstrates how loop conditions and optional guards handle real-world complexity gracefully.

**Why These Diagrams Matter**

It ensures the correct delegation of logic between UI, controller, and data layers and enforces policy constraints on both user intent and system state. This reinforces the importance of temporal and role-aware logic in a government-like system where user access is conditional. Together, they capture the most constraint-heavy interactions, helping us verify that our system correctly filters, processes, and enforces eligibility in high-stakes workflows.

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

**Single Responsibility Principle (SRP)**

Classes such as ApplicantController, ApplicantMenu, and Applicant each has a single responsibility. The menu classes act as the boundary enabling user interaction, the controller classes handle the logic behind user actions, and the entity classes represent the core data and state of the system.

**Open/Closed Principle (OCP)**

Classes such as OfficerController extends ApplicantController to enable officer-specific duties while retaining applicant functionalities, allowing new capabilities to be supported without modifying the base class, thus promoting code reusability and future extensibility.

**Liskov Substitution Principle (LSP)**

Classes such as OfficerController extends ApplicantController by inheriting common methods like project browsing and adding officer-specific functionalities such as managing applications or projects, thereby preserving expected behaviors and ensuring it can be used polymorphically wherever ApplicantController is expected, enhancing code correctness, reducing the need for type checks or casting, and improving overall maintainability.

**Interface Segregation Principle (ISP)**

We used lightweight interfaces like IApplicantServices, IOfficerServices, and IManagerServices, where each user role implements only the methods relevant to its domain. For example, ApplicantController implements IApplicantServices, while OfficerController implements IOfficerServices, which led to a cleaner, more maintainable codebase with clearly focused responsibilities, thereby simplifying future changes and enhancing reliability.

**Dependency Inversion Principle (DIP)**

For example, ApplicantMenu depends on the IApplicantServices interface rather than directly on ApplicantController, ensuring that high-level modules like UI menus depend on abstractions instead of concrete implementations. This keeps ApplicantMenu flexible and decoupled, enhances testability, allows easy substitution of different controllers in the future, and adheres to clean architecture.

**Chapter 4: Implementation (Java)**

* 1. **Tools Used:** Java 17. IDE: Eclipse/Vscode. Github
  2. **Sample Code Snippets:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Applicant Interface**  Figure 1: Encapsulation | A screen shot of a computer program  AI-generated content may be incorrect.  Figure 2: Applicant Interface | A screen shot of a computer program  AI-generated content may be incorrect.  Figure 3: Polymorphism & Inheritance | Figure 4: Error Handling |

**Chapter 5: Testing**

**5.1 Test Strategy**

To ensure our BTO Management System functioned as intended, we adopted a manual functional testing approach. Given that our system is a command-line application without automated test frameworks, we focused on validating realistic user flows across all roles and features.

**Types of Testing Used**

**Manual Functional Testing**: We executed test cases for each major feature, such as application submission, flat booking, officer registration, project management, based on role-specific menus. Each function was tested with both valid and invalid inputs to verify proper error handling and boundary enforcement. We followed step-by-step test flows mimicking actual usage, such as an Applicant booking flow from login to flat selection, Officer managing flat availability and replying to enquiries, Manager handling registrations and generating reports etc.

**5.2 Test case table**

|  |  |  |  |
| --- | --- | --- | --- |
| **A screenshot of a computer program  AI-generated content may be incorrect.**  Figure 5: : Login test case | **A white sheet with black text  AI-generated content may be incorrect.**  Figure 6: Officer Table | **A white sheet of paper with black text  AI-generated content may be incorrect.**  Figure 7: Applicant test case | **A white sheet of paper with black text  AI-generated content may be incorrect.**  Figure 8: Applicant test case |

**Chapter 6: Documentation**

* 1. **Javadoc:** A copy of the Javadoc is included in the zip file

**6.2 Developer Guide**

1. Clone the repository using the link [here](https://github.com/yeowh03/BTO-Project.git). 2. Open project in Visual studio code/Eclipse. 3. Run Main java

**Chapter 7: Reflection & Challenges**

**What Went Well**

Our team successfully implemented the required functionalities for each user role, and the system reflected realistic workflows aligned with the BTO process.The use of layered architecture (Entity, Controller, UI) improved modularity and made logic separation clearer.We were able to apply SOLID principles meaningfully to improve maintainability.

**What Could Be Improved**

Time constraints limited our ability to implement more advanced features like a user-friendly GUI.Some earlier controller designs became tightly coupled with UI decisions, which made refactoring a bit more difficult later on.

**Individual Contributions**

Wen Hong: Logic flow and Coding. Zixiao: Report and Debugging. Xiao Yao and Zou Ning: UML and Sequence Diagrams

**Lessons Learned About OODP**

Firstly, we gained hands-on experience translating real-world roles and constraints into object-oriented models, with meaningful relationships like inheritance and aggregation. Secondly, we saw how SOLID principles, while sometimes adding complexity up front, greatly improved scalability and testing ease.Designing controller-entity boundaries helped us understand the importance of separation of concerns in complex systems. Lastly, we also learned that early planning of data structures and system flow is key to reducing bugs and rework.

**Chapter 8: Appendix**

* [Github link](https://github.com/yeowh03/BTO-Project.git)
* No references were used