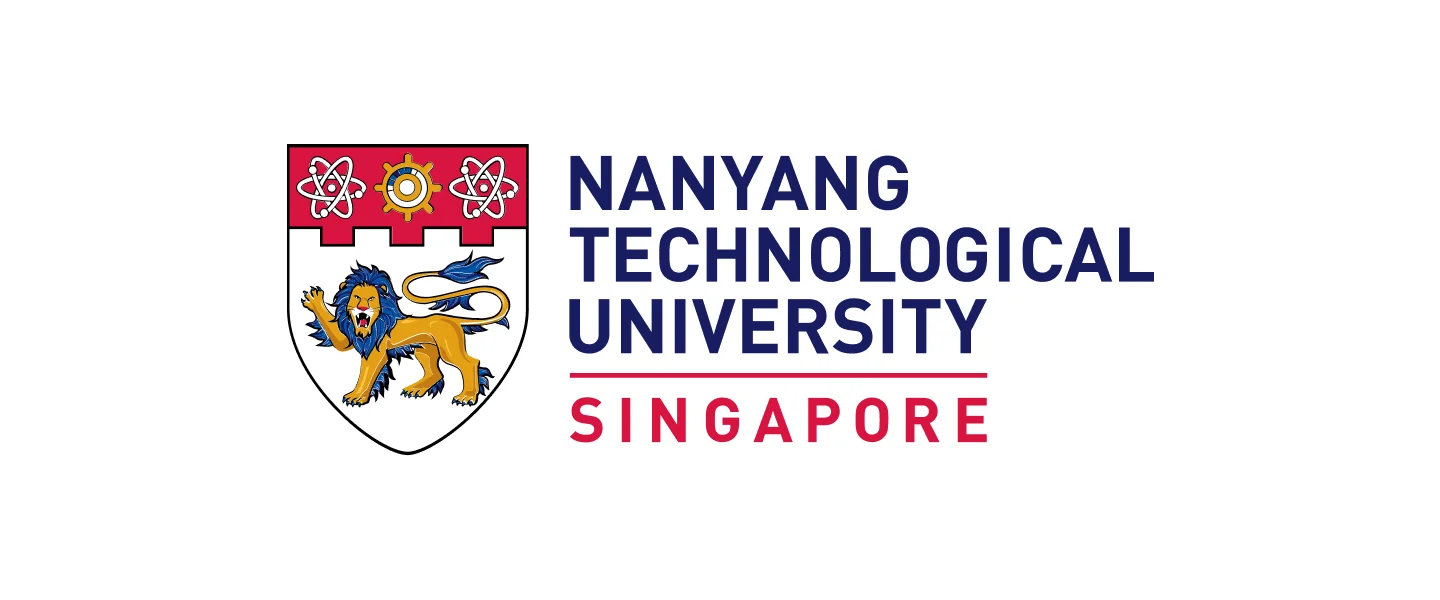
****

**Object Oriented Design and Programming**

**SC2002**

**24/25 Semester 2**

**Group Assignment**

| **Project Title: BTO Management System** | |
| --- | --- |
| **Name** | **Matric Number** |
| **Goh Ee Sheng** | **U2420928E** |
| **Goh Zhuan Boon** | **U2422902C** |
| **Chandra Praket** | **U2423524G** |
| **Zac Low** | **U2423890J** |

**Introduction**

This report outlines the design and implementation of the BTO Management System, a comprehensive application for managing Built-To-Order housing projects in Singapore. The system facilitates interactions between applicants, HDB officers, and HDB managers through a well-structured object-oriented architecture that addresses the complex requirements of the HDB application process

**Understanding the Problem and Requirements**

To start this project off, we read through the assignment document and we highlighted all the nouns and labeled them as potential classes. Based on this we discussed and went through a process of elimination to determine which of these were core entities, which were attributes, and which could be grouped into utility or service components. This process helped us define our initial class structure and identify the key entities.

The main problem domain is how we manage the BTO applications, bookings and how the applicant and HDB staff communicate with each other. The system’s entire purpose is to serve as a centralized platform for managing the BTO project, application processing as well as user interactions.

Some explicit requirements that were given are:

* User Authentication: (using NRIC and default password)
* User Roles: (Applicant, HDBOfficer, HDBManager)
* Applicant Capabilities: (View projects, Apply projects, Withdraw, Book flats (through officer), Manage enquiries):
* HDBOfficer Capabilities: (Inherits all Applicant features and can manage flat bookings)
* Application Status Tracking: (Pending, Successful, Unsuccessful, Booked)

From there we can infer the following implicit requirements:

* Data Validation: (NRIC format validation, application constraints)
* State Persistence: (User sessions, application progress)
* Role-Based Access Control: (Different permissions based on user roles)
* File Upload Functionality: (Initialize user lists)

Some ambiguities and how we choose to resolve them are:

* Enquiry Management: (We implemented text-based enquiries with tracking)
* Receipt Generation: (PDF format generation following Single Responsibility Principle)

**Deciding on Features and Scope**

With the analysis we did, we then grouped features into three categories, core, optional and excluded. With this it helps us clearly outline our goal and makes the work more streamline.

We decided that the following features will be implemented in the BTO management system

| General Features (All Users) | - Login system with NRIC and password authentication  - Password change functionality  - View available projects  - Submit enquiries about projects  - Logout functionality |
| --- | --- |
| Applicant Features | - View available BTO projects  - Apply for projects with eligibility checking based on marital status and age  (TWO-ROOM for singles)  (THREE-ROOM for married couples)  - View application status  - Request application withdrawal  - Submit and track enquiries  - View personal application details |
| HDB Officer Features | - View and reply to enquiries  - View project details  - Able to access all applicant functionalities  - Register for projects  - Generate booking receipts  - Change password |
| HDB Manager Features | - Project Management such as View, Create, Edit, Delete projects (and toggle visibility)  - Officer Registration such as View, Approve, Reject officer registration  - Application Management such as approving or rejecting applications or withdrawal requests  - Enquiry Management (viewing and replying)  - Report Generation |
| Project Management Features | - Multiple flat types (Two or Three room)  - Project visibility controls  - Neighbourhood information  - Track flat availability  - Track application status |
| System Features | - Test case functionality  - Excel integration  - User session management |

**Planning the System Structure**

Before implementation, we carefully designed the overall system layout by breaking it down into logical components based on the identified entities, responsibilities, and user interactions

Before constructing our class diagram we modeled our user flow to get a better overview of how the system works. This step is crucial for us to have a better insight into the relationships between different components and classes. It has helped us to differentiate the "is-a" and "has-a" relationships. For example this step helped us come to the decision that we will make the HDB Officer inherit from Applicant because it possesses all applicant capabilities.

**Architecture Overview**

The system follows a traditional object-oriented layered architecture:

1. *Domain Layer* - Core entities (Project, Application, Enquiry)
2. *Service Layer* - Business logic for different user roles
3. *Data Access Layer* - Persistence and data management
4. *Presentation Layer* - User interfaces for different operations

**Reflection on Design Trade-offs**

There were some portions of the code that we initially wanted to combine such as the excelReader and excelWriter to make things simpler but after learning that we have to showcase design principles in our code we decided to separate them. For the majority of the project we decided that we would like to make things as simplistic as possible while maintaining the features. Our design goal was to ensure that it is simple to follow and understand while recognising the need for potential extensibility.

**Class Diagram**

As mentioned in the beginning of the report, as a team we read through the assignment document and we highlighted all the nouns and labeled them as potential classes. We then slowly eliminate repeated or similar words to determine which of these were core entities, which were attributes, and which could be grouped into utility or service components.

We then decide that the following are the responsibilities of each class.

| **Core Domain Entities (Project, Application, Enquiry, FlatType)** | | |
| --- | --- | --- |
| **Class** | **Relationships** | **Responsibility** |
| **Project** | **Project → Application (One-to-Many)**  A project can have multiple applications  Applications are tied to specific projects  Project manages application lifecycle  --------------------------------------------------------  **Project → Enquiry (One-to-Many)**  A project can have multiple enquiries  Enquiries are associated with specific projects  Project tracks all related enquiries  --------------------------------------------------------  **Project → FlatType (Uses)** | Core entity that represents the BTO project managing:  - Project details (name, location, description)  - Application period validation  - Flat type availability and counts  - Application tracking  - Officer assignments  - Visibility control |
| **Application** | **Application → Project (Many-to-One)**  Applications are linked to specific projects  Project manages application validation  Application status affects project state | Represents a formal application for a flat in a project  Manages:  - Application lifecycle (creation, approval, withdrawal)  - Flat type selection  - Status tracking  - Withdrawal requests |
| **Enquiry** | **Enquiry → Project (Many-to-One)**  Enquiries are linked to specific projects  Project manages enquiry visibility  Enquiry status affects project communication | Represents user enquiries about BTO projects with complete tracking.  Handles:  - Enquiry creation and tracking  - Communication between users and HDBOfficer/Manager  - Reply management  - Timestamp tracking |
| **Receipt** | **Receipt → Application (One-to-One)**  One Application can have at most one Receipt  One Receipt is associated with exactly one Application  **Receipt → HDBOfficer (Many-to-One)**  One HDBOfficer can generate many Receipts  Each Receipt is generated by exactly one HDBOfficer | Record Booking Details:  Stores the applicant's personal information:   * Name * NRIC * Age * Marital Status   Records the booking details:   * Type of flat booked * Project information * Booking date   Has a generatePDF() method that creates a formal PDF document |

**-------------------------------------------------------------------------------------------------------------------------------**

| **Value Objects (FlatType, ApplicationStatus, Enquiry, FlatType)** | |
| --- | --- |
| **Enum** | **Responsibility** |
| **FlatType** | Defines the standard flat types available in BTO projects  - TWO\_ROOM  - THREE\_ROOM  - Enforces type safety in flat selection  - Used to track availability and manage quotas  - Essential for eligibility rules (e.g., singles can only apply for 2-room) |
| **ApplicationStatus** | Represents the complete lifecycle of a BTO application  - PENDING  - SUCCESSFUL  - UNSUCCESSFUL  - BOOKED  - Controls what actions can be performed (e.g., only SUCCESSFUL applications can book flats)  - Used to enforce business rules and workflows  - Essential for tracking application progress | |
| **RegistrationStatus** | Manages the registration process for HDB officers  - PENDING  - APPROVED  - REJECTED  - Controls officer permissions and capabilities  - Used to enforce officer eligibility rules  - Essential for project management workflows |
| **Withdrawal Status** | Manages the withdrawal status of a BTO application  - NIL  - PENDING  - APPROVED  - REJECTED  - Controls whether a withdrawal request is under review, accepted, or denied  - Used to prevent further actions on withdrawn applications  - Essential for auditing and refund processing workflows |

**-------------------------------------------------------------------------------------------------------------------------------------**

| **Presentation/UI (FlatType, ApplicationStatus, Enquiry, FlatType)** | |
| --- | --- |
| **Class** | **Responsibility** |
| **ApplicantMenu** | Primary interface for BTO applicants  - Handles application submission and tracking  - Manages flat booking process  - Allows viewing application status and project details  - Provides access to applicant-specific features |
| **ApplicationManagerMenu** | Manages application processing workflow  - Handles application approval/rejection  - Processes withdrawal requests |
| **EnquiryMenu** | Manages enquiry submission and tracking  - View all enquiries  - Reply to an enquiry |
| **HDBManagerMenu** | Main interface for HDB managers  - Provides access to all manager functions  - Handles user management  - Controls system settings  - Manages overall system operations |
| **HDBManagerProjectMenu** | Specialized interface for project management  - Handles project creation and editing  - Manages project visibility |
| **HDBOfficerMenu** | Main interface for HDB officers  - Provides access to officer functions  - Handles project registration  - Manages enquiries and responses  - Processes flat bookings |
| **OfficerRegistrationMenu** | Handles officer registration process  - Manages registration requests  - Tracks registration status  - Processes registration approvals  - Maintains officer record |
| **ProjectMenu** | General project interface  - Displays project details |
| **ProjectRegisterMenu** | Specialized interface for project registration  - Manages registration details  - Tracks registration status |
| **UserMenu** | Handles user account management  - Manages user profiles  - Controls authentication  - Handles password changes  - Manages user preferences |
| **LoginInterface** | Handles user authentication and directing users to their appropriate menus based on their user roles  - User authentication  - Role based menu navigation  - Error handling (NRIC Validation) |
| **Applicants:** ApplicantMenu  **Officers:** HDBOfficerMenu, ProjectMenu, ProjectRegisterMenu, EnquiryMenu  **Managers:** HDBManagerMenu, HDBManagerProjectMenu, OfficerRegistrationMenu, EnquiryMenu, ApplicationManagerMenu | |

**-------------------------------------------------------------------------------------------------------------------------------------**

| **Utility (FlatType, ApplicationStatus, Enquiry, FlatType)** | |
| --- | --- |
| **Class** | **Responsibility** |
| **BTODataStore** | Centralized repository for users, projects, applications, enquiries, and registrations  - Provides methods for adding finding and removing entities  - Loads and saves data with PersistenceUtils |
| **excelReader** | Reads Excel with Apache POI  - Opens and goes through the excel sheet row by row and extracts the data. |
| **excelWriter** | Writes in Excel with Apache POI  - Exports data to Excel files  - Creates Workbook |
| **loginHandler** | Handles the user login process  - Extends Userfilehandler to reuse file reading logic  - Checks through the files for valid valid credentials - Ensures NRIC format is valid before verifying password |
| **PersistenceUtils** | Handles all reading/writing to CSV files  - Saves lists of users, projects, applications etc. to CSV  - Loads those lists back as well |

Our class relationships were determined by our own logic and interpretation of our code. For example the table below shows our thought process on how the relationship would be like for the User Hierarchy.

| **User Hierarchy (User, Applicant, HDBOfficer, HDBManager)** | |
| --- | --- |
| **Class** | **Justifications** |
| **User (Base Class)** | Defines shared attributes (NRIC, password, age, marital status) and behaviors (login, change password, view/filter projects, manage enquiries) for all users. Serves as the base class for all user roles, promoting encapsulation and reuse. |
| **Applicant (extends User)** | Represents users who can apply for BTO projects and view application status. Inherits common user features from User |
| **HDBOfficer (extends Applicant)** | HDB Officer should inherit from Applicant because it possesses all applicant capabilities.  Since Applicant already inherits from User, HDB Officer will indirectly inherit from User as well, avoiding redundant inheritance.  Adds on officer-specific functions such as registering projects, handling enquiries, confirming bookings, and generating receipts  This design ensures:  ✔ Code reuse (HDB Officer gets all Applicant features)  ✔ Avoids redundancy (No need to inherit from both User and Applicant) |
| **HDBManager (extends User)** | HDB Managers do not apply for BTOs. Therefore it directly extends User, excluding applicant functions. Responsible for project CRUD (Create, Read, Update, Delete), toggling visibility, approving officer registrations, applications, withdrawals, and generating reports. |

There were certain trade offs that we have to consider when working on this project. For example one of the requirements is to show the design principles. If we were to code the entire project adhering to the design principles it would be massive and would require more time. Thus we went for a simplistic approach.

**The UML diagram will be separated from this document to save space.**

**Sequence Diagrams**

For the Booking process Sequence Diagram, we chose to demonstrate the report generation with file output. This process was selected because it encapsulates some of the essential administrative functions of the system, such as project creation, applicant and officer approval, and final report output. Less relevant processes were omitted, such as replying to applicant enquiries, approving withdrawals, and password changes to maintain clarity and highlight the flow of this specific task.

For the Manager Sequence Diagram, we chose to demonstrate the flat application and booking process. Similarly, less relevant processes were omitted. This process was selected as it involves coordination across all three key user roles: the applicant initiates the booking, the HDB Manager approving the application, and only then can the HDB Officer proceed with booking on behalf of the applicant. The sequence diagram was used to clearly map out this multi-user interaction and the dependencies involved.

**The Sequence diagram will be separated from this document to save space.**

**Application of OOD Principles (SOLID)**

While it may not be feasible for the BTO Management System to fully adhere to all SOLID design principles due to its complexity, we selectively applied each principle to specific parts of our BTO Management System to demonstrate our understanding

**Single Responsibility Principle (SRP)**

The **Single Responsibility Principle (SRP)** states that a system should have classes that are responsible for only one part of the functionality. This principle ensures ease of maintenance and clarity of the system structure by ensuring that each class has one distinct responsibility, making the system less prone to errors when changes are required. In our BTO Management System, we chose to separate the responsibilities of reading data from Excel files and writing data to Excel files into two different classes, ExcelReader and ExcelWriter.

The **ExcelReader class** is responsible for reading and extracting data from Excel files. This includes parsing the file, validating the contents, and converting the data into usable formats for further processing within the system. The sole responsibility of the ExcelReader class is to handle input from Excel files.

The **ExcelWriter class** is responsible for writing data to Excel files. It handles the functionality of formatting and exporting system data into an Excel file format suitable for reporting or archival purposes. The sole responsibility of the ExcelWriter class is to manage output to Excel files.

**Open-Closed Principle (OCP)**

The **Open-Closed Principle (OCP)** states that a system should be open for extension but closed for modification. This principle ensures that when new functionalities are added the core functionalities are not modified, thus promoting maintainability and scalability. In our BTO Management System, we chose the relationship between the user abstract class, Applicant, HDBOfficer and HDBManager to demonstrate the Open-Closed Principle.

To adhere to OCP, an abstract **User class** is introduced. The User class serves as the base class for all user types. It defines common attributes and behaviors such as authentication, user details (e.g., NRIC, age, marital status).

The system then extends User into two primary roles: **Applicant and HDBManager.** The HDBOfficer role would extend Applicant, inheriting its capabilities while adding specific functionalities related to project management and flat booking as the criteria stated that HDBOfficer possesses all applicant capabilities. Since Applicant already inherits from User, HDB Officer will indirectly inherit from User as well, avoiding redundant inheritance.

**Liskov Substitution Principle (LSP)**

The **Liskov Substitution Principle (LSP)** states that the object of a superclass should be able to be replaced by objects of a subclass without affecting the system. In our system, the abstract User class defines several abstract methods that establish a contract for all user types.

* public abstract Enquiry submitEnquiry(Project project, String enquiryText);
* public abstract boolean editEnquiry(String enquiryId, String newText);
* public abstract boolean deleteEnquiry(String enquiryId);

These abstract methods will define the expected behavior for the enquiry management across all the different user types (Applicant, HDBOfficer, HDBManager). All the subclasses must implement these methods to maintain the “contract” with the base class (User). By doing this, it benefits our system in such a way where the code is reusable as common functionality is defined once in the base class and reused across the subclasses, and it will be easier to maintain as changes to the base class will mean all the subclasses have to adapt accordingly.

**Interface Segregation Principle (ISP)**

The **Interface Segregation Principle (ISP)** states that a class should not be forced to implement methods that it does not need. Although we did not not explicitly implement Java interfaces, we followed ISP by designing cohesive classes with focused responsibilities, avoiding "fat" interfaces with methods that clients don't need and creating specific menu classes for different operations. This ensures that each user role only contains the methods that are relevant to its own responsibilities. This implementation will help reduce coupling and improve the maintenance of the code.

**Dependency Inversion Principle (DIP)**

The Dependency Inversion Principle (DIP) states that high-level modules should not depend on low-level modules. The core of these principles emphasizes decoupling and abstraction and both the high-level and low-level modules should depend on abstraction. In our BTO Management System we demonstrate DIP through the design of our data storage and project management components.

High-level modules such as the user classes depend on abstractions rather than concrete implementation and the BTODataStore is implemented as a singleton which provides a stable abstraction for data access. All in all, we rely on abstractions rather than concrete implementations where appropriate.

**Implementation**

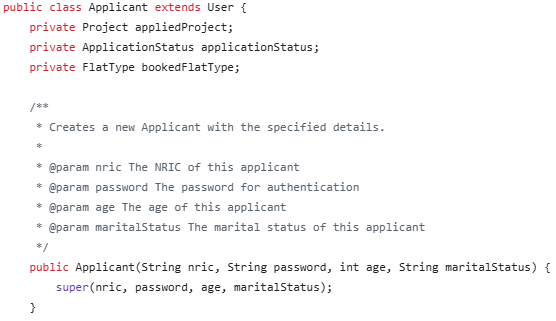
For our tools used, we used Java 17 and must be compiled with JDK 17, our preferred IDE was Visual Studio Code and the version control is GitHub. We also used Maven to manage the build process and documentation of Java.

**Sample Code Snippets**

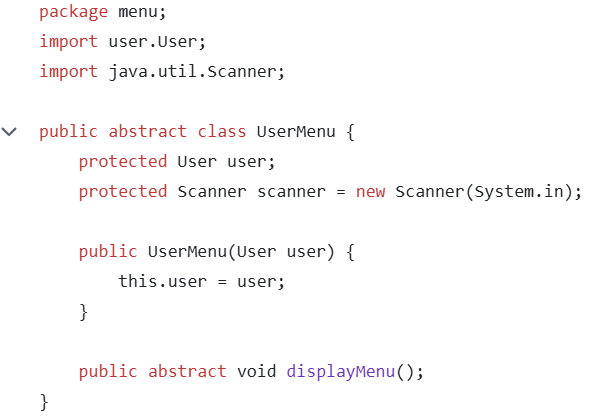
**Encapsulation** in Java involves hiding information about attributes and methods. The importance of it is to keep certain fields safe from unintended modification and to hide the implementation. The snippet of the code below shows the private fields of the HDBOfficer class. The private fields are hidden from other classes and only the HDBOfficer can access them directly. However the public fields are accessible to everyone.

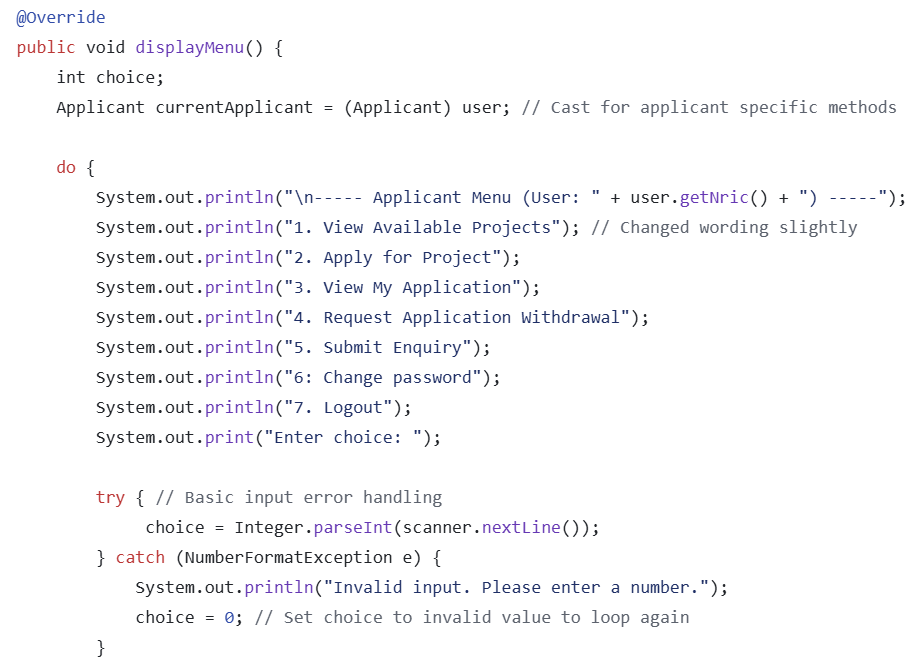
****

**Inheritance** in Java allows new classes to inherit properties and methods of an existing class. The benefit of it is that the code can be reusable and does not need to be rewritten. It promotes extensibility and other subclasses can further extend and it can cater if a more specialized version of the parent class were to be created in the future. The snippet of the code below shows that Applicant extends User where the user is the parent class. This means the Applicant class will inherit the properties of the User. The Applicant can later add on specific attributes and methods.

****

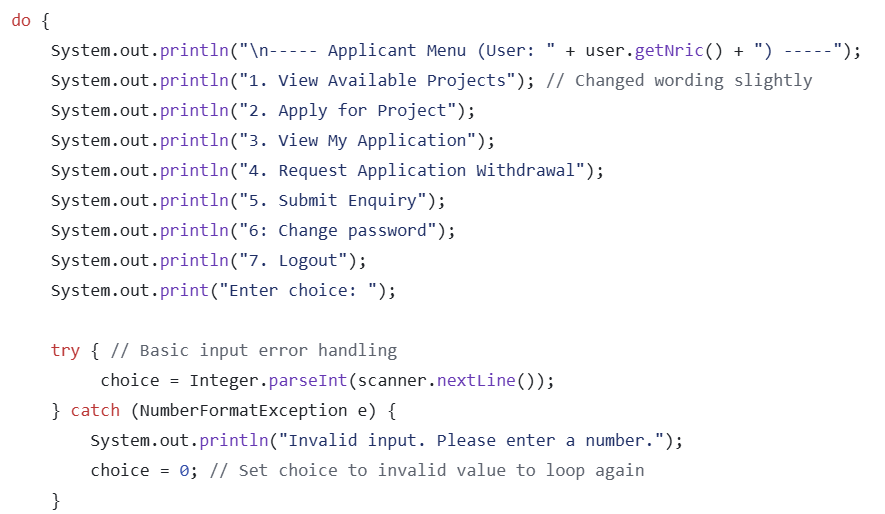
**Polymorphism** in Java is where a task performs a single action in different ways. One common example is method overriding which is when a subclass have its own implementation of a method which is defined in the superclass. The snippet of the code below shows how each menu extends the abstract user menu. The Applicant, HDBOfficer and HDBManager each have their own implementation of the displayMenu() method. This demonstrates how the same method name behaves differently depending on the object’s runtime type.





**We do not have any interfaces. Our system mainly makes use of inheritance and abstract classes like the UserMenu as shown above.**

**Error Handling** in Java is to ensure that our application can handle unexpected situations such as when a user input something invalid or missing files to prevent crashes for a better user experience. One such example is the try and catch blocks which allows the program to catch exceptions. The snippet of the code below shows an example of a basic input error handling. In our code, it tries to read the user’s input and if the user types something that's not an integer (number), it catches the error and shows a friendly message.

****

**Test Strategy**

For our test strategy, the majority of our test cases are unit testing with a few requiring manual functional testing. We went from a top down approach with the testing and the flow would be how the user login to the main functionalities of the Applicant such as booking an application and submitting and enquiry, to the main functionalities of the HDBOfficer such as checking their eligibility and responding to enquiries, to lastly the main functionalities of the HDBManager such as togginging project visibility and view all project listings.

For example, the login process includes 4 test cases. Valid User Login, Invalid NRIC Format, Incorrect Password and Password Change Functionality. A user with a wrong user login should not be able to access the system. In the event the user types in an invalid NRIC format, the system will prompt them and say that the format is invalid. In the event the user types in an invalid password the system will say that it is incorrect. The system also has a password change functionality where users can change their password. The rest would be as follows in the test case table.

**Test Case Table**

| **No.** | **Test Case** | **Test case Implementation (all Passed)** |
| --- | --- | --- |
| 1 | Valid User Login |  |
| 2 | Invalid NRIC Format |  |
| 3 | Incorrect Password |  |
| 4 | Password Change Functionality |  |
| 5 | Project Visibility Based on User Group and Toggle |  |
| 6 | Project Application |  |
| 7 | Viewing Application Status After Visibility Toggle Off |  |
| 8 | Single Flat Booking per Successful Application |  |
| 9 | Applicant’s enquiries management |  |
| 10 | HDB Officer Registration Eligibility |  |
| 11 | HDB Officer Registration Status |  |
| 12 | Project Detail Access for HDB Officer |  |
| 13 | Restriction on Editing Project Details |  |
| 14 | Response to Project Enquiries |  |
| 15 | Flat Selection and Booking Management |  |
| 16 | Receipt Generation for Flat Booking |  |
| 17 | Create, Edit, and Delete BTO Project Listings |  |
| 18 | Single Project Management per Application Period |  |
| 19 | Toggle Project Visibility |  |
| 20 | View All and Filtered Project Listings |  |
| 21 | Manage HDB Officer Registrations |  |
| 22 | Approve or Reject BTO Applications and Withdrawals |  |
| 23 | Generate and Filter Reports |  |

**Documentation**

**The Javadoc and the Developer Guide on how to set up the environment will be separated from this document.**

**Reflection & Challenges**

The duration of the project has many ups and downs. Some of the things that went well was the selection of which portion of our program we should pick to work on the sequence diagram. The group shared the same view on what we should pick thus making the decision process easier.

Working on the class diagram was harder than we expected because we had to make multiple drafts and changes. We kept shifting our views on what the linkage should be between the classes and how to display the multiplicity as different classes have different multiplicities with each other depending on which class we start our viewpoint from.

Communication can be improved during the project. As we are all quite new to collaborative programming it is hard to sync our work up and communicate. With other modules to work on and our differences in timetable we found it hard to get together to update each other on our individual progress. Having regular meetings to check-in can be something we work on and implement for our future projects.

The coding portion of the project is no easy feat. As java was a new language for all of us, we struggled initially as we are still trying to understand the concepts that we are being taught during the lectures. The second half content was content heavy in terms of being quite closely tied to our project. The introduction to UML Class Relationships, UML Model Sequence Diagram and Design Principles took us aback as it was a steep learning curve and we realised how challenging the project will be when we found out that we are required to demonstrate all the following knowledge in our project.

Some lessons we learned are the high importance of having a good structured class diagram as it can go a long way when we code our program. A simple yet detailed class diagram can allow multiple people to work on the same project and one look at the diagram we should know how the flow works. This entire project made us realize the importance of computational thinking. In a sense, creating a class diagram is a form of abstraction where we simplify the long requirement list by identifying essential attributes and behaviors, allowing us to focus on the core structure without getting lost in unnecessary details. Decomposition is also important as this is a relatively huge project with many deliverables and requirements,it is important that we split our work into smaller, manageable tasks such as designing classes, implementing methods, and testing individual components, making the overall problem easier to solve collaboratively.

**Appendix**

The github link is as follows: <https://github.com/goheesheng/sc2002_HD>

For the development of the UML diagram we used Visual Paradigm.

Our external libraries includes

* Apache POI (Version 5.2.3) - To Read and Write Excel Files
* poi: Core library for Microsoft Office file operations
* poi-ooxml: Support for Office Open XML format (XLSX)
* Maven: For dependency management and project build automation

Our supporting libraries include

* XMLBeans (Version 5.1.1) - XML processing support for Apache POI
* Apache Commons Compress (Version 1.2.1) - Handling compressed file formats

**References**

[1] Apache Maven Project. (2024). Maven - Welcome to Apache Maven. https://maven.apache.org/

[2] Visual Paradigm. (2024). Visual Paradigm - UML, SysML, BPMN, ArchiMate, ERD, DFD, and more. https://www.visual-paradigm.com/

[3] Apache POI. (2024). Apache POI - the Java API for Microsoft Documents. https://poi.apache.org/

[4] Apache XMLBeans. (2024). Apache XMLBeans. https://xmlbeans.apache.org/

[5] Apache Commons Compress. (2024). Apache Commons Compress. https://commons.apache.org/proper/commons-compress/

[6] SLF4J. (2024). SLF4J - Simple Logging Facade for Java. https://www.slf4j.org/

[7] Maven Build Helper Plugin. (2024). build-helper-maven-plugin. https://www.mojohaus.org/build-helper-maven-plugin/

[8] Maven Compiler Plugin. (2024). maven-compiler-plugin. https://maven.apache.org/plugins/maven-compiler-plugin/

[9] Maven Exec Plugin. (2024). exec-maven-plugin. https://www.mojohaus.org/exec-maven-plugin/