

**SC2002 Object-Oriented Design & Programming**

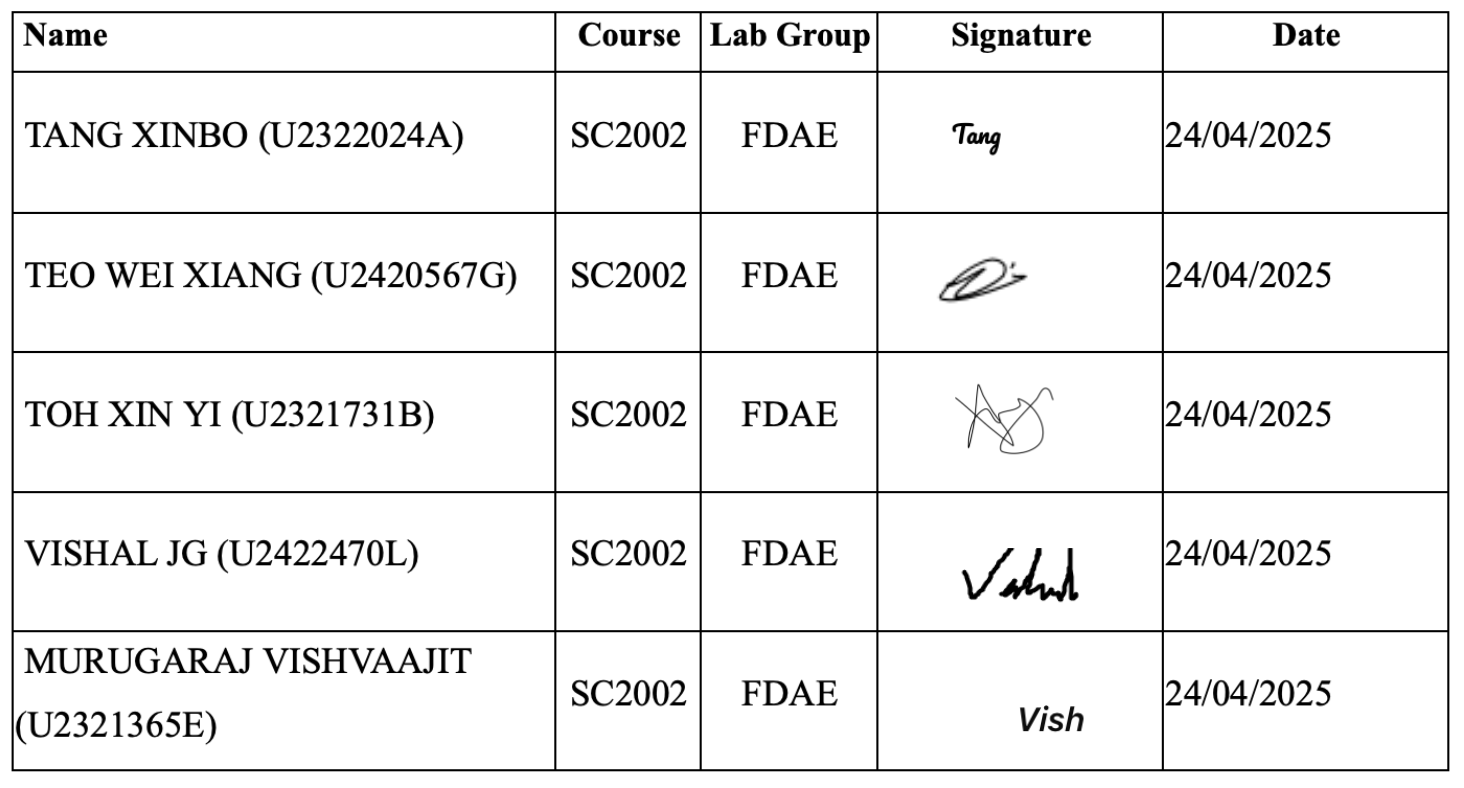
**ASSIGNMENT: Building an OO Application**

**FDAE Group 1**

**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.



**1.0 Design Considerations**

The Built-To-Order Management System (BTOMS) is a Java-based command-line application developed to simulate the application and management process of Singapore’s public housing system. Designed with clarity, modularity, and maintainability in mind, BTOMS allows Applicants, HDB Officers, and HDB Managers to apply for flats, review applications, book units, and manage BTO project listings. This system prioritises reusability, extensibility, and separation of concerns across a multi-layered architecture to support future upgrades and clean maintenance.

We had identified the features into three categories: key, additional and excluded. The goal was to make sure the key features were implemented first, then followed by the additional features while making sure to adhere to the object-oriented principles while implementing the features.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ✅  Key Features | Login Authentication with input validation, routing to correct user menu | Application, project management, officer registration, bookings, receipt and report generations, and the respective input validations | | Filtering of projects and generating reports |
| 🟡  Additional Features | Password mask and toggle, password strength validation, password hash | | Pending officer registration count, preferred flat type for married applicants, remaining flat units | |
| ❌  Excluded Features | Changing password at first registration. The default password upon registration is “password”. Ultimately, we decided that the user could manually change their password through their user interfaces after login. | | | |

**1.1 Design Approach**

Our design follows the Model-View-Controller (MVC) architectural pattern, extended with a service-repository structure for better separation of concerns. The final components include: *Entities, Views, Controllers, Services, Repository, Utilities, Exceptions.*

When users (i.e. *Entity*) log in, they are routed to a role-specific command line interface (i.e. *Views*). These interfaces guide users through available functions by prompting for inputs and validating basic data. Once the input is captured, the *view* forwards it to a corresponding *controller*, which interprets the user's intent and invokes the appropriate *service* method. *Services* handle the core business logic, such as checking application eligibility or performing bookings, and communicate with *repositories* to access or persist data accordingly. Any errors along the process are handled via *exceptions* classes, while *utility* classes support common functions like password hashing and ID generation.

This structured flow reduces redundancy and ensures each part of the system operates within a well-defined scope, improving maintainability and testability. The clear layers defined allows us to simplify testing, debugging, and future extensions. Our approach establishes individual components that evolve independently, thereby ensuring predictable data flow, loose coupling and high cohesion.

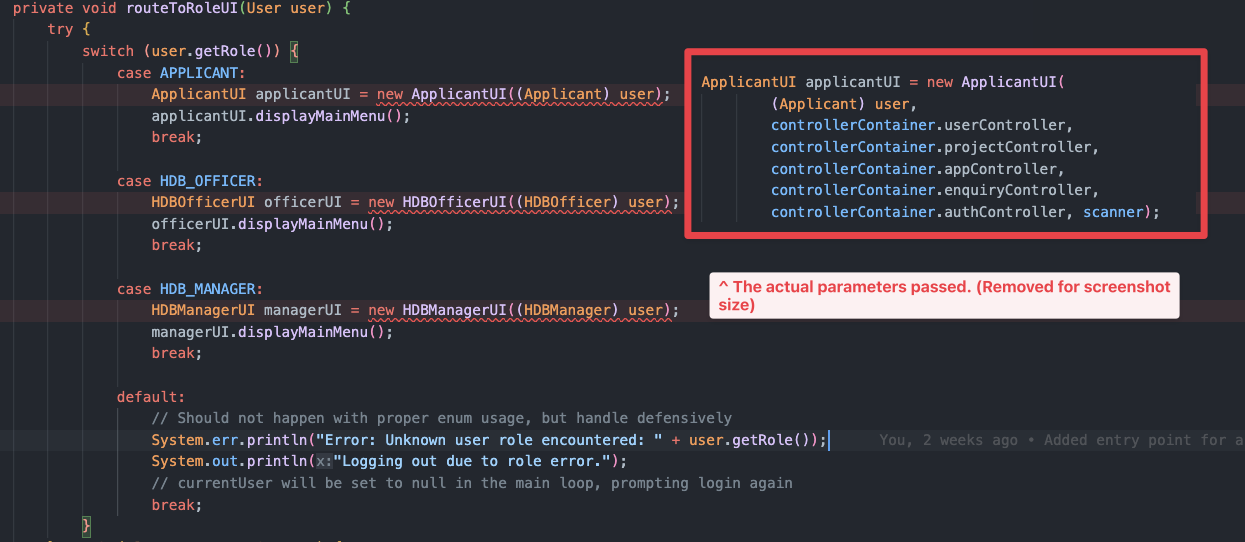
**1.2 Design Principles**

Our system design closely follows the SOLID principles of object-oriented programming to ensure maintainability, testability, and extensibility. These principles were not only used as theoretical guidelines but were applied concretely across the structure and logic of BTOMS, as reflected in our UML diagram and source code.

**Single Responsibility Principle (SRP)**: Every class is designed to handle one clear responsibility. For example, ApplicationService is solely in charge of concrete implementations for application processes (i.e. submission, withdrawals), while ApplicantUI handles the displaying of application information. ApplicationRepository handles the reading and writing of application data across CSV files. This separation minimises the ripple effect of changes and highlights cohesion within classes.

**Open/Closed Principle (OCP)**: We use interface-driven development to promote extensibility without modifying existing code. For instance, having a IReportService interface allows us to add new types of booking reports in the future without altering core implementations in ReportService. Similarly, IApplicationService and IProjectService abstract key workflows such that alternate implementations and updates (e.g. integrating a database-backed service in place of CSV-based repositories) can be injected without disrupting the system. Such abstractions ensure our system’s flexibility.

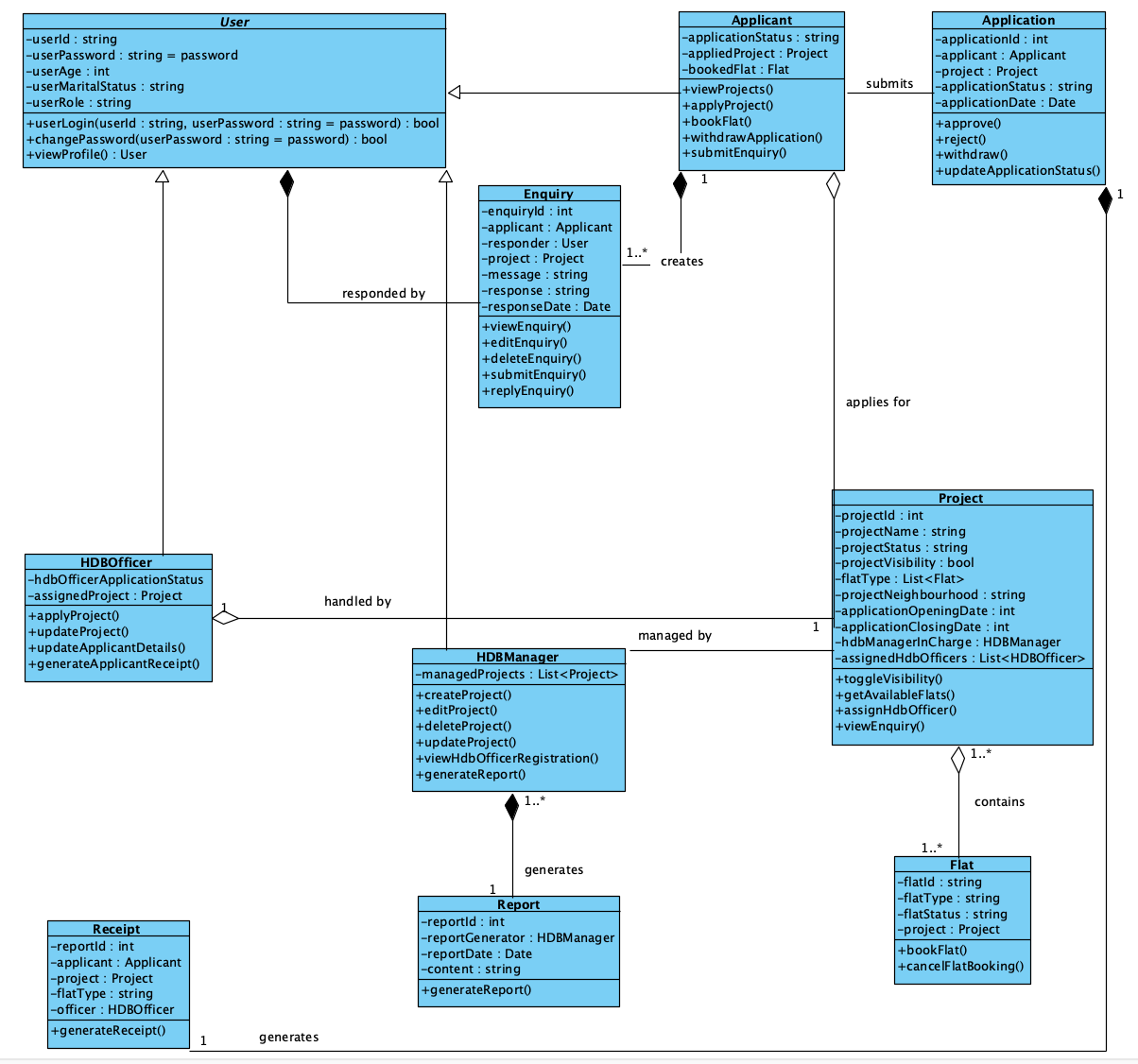
**Liskov Substitution Principle (LSP)**: Our User abstract class is extended by Applicant, HDBOfficer, and HDBManager, each overriding the getRole() method. These subclasses can be used interchangeably of the base class without causing disruptions to our program. For instance, in app.routeToRoleUI(User user), polymorphism ensures that the correct UI class is instantiated based on the runtime type of the user.



**Interface Segregation Principle (ISP)**: Rather than one massive general service or views interface, we divide contracts by specific responsibilities. For example, IOfficerRegistrationService is only used for officer registration workflows, while IBookingService manages only the booking logic. This ensures that classes such as OfficerRegistrationController and BookingController associated with the interfaces are only exposed to necessary methods, reducing unnecessary coupling and improving cohesion.

**Dependency Inversion Principle (DIP)**: Our implementation ensures dependence on abstractions rather than concrete implementations. For instance, high-level modules like ProjectController or ApplicantUI do not depend on any low-level modules like ProjectService, and rather on the IProjectService interface which is passed in via its constructor. Similarly, low-level modules like ProjectService implements IProjectService, depending on its abstractions. This fulfills DIP, where both high and low-level modules depend on abstractions rather than each other. Furthermore, the IProjectService interface contains only method signatures, and does not depend on the details of concrete method implementations. This ultimately ensures that high-level logic is decoupled to specific implementations and only depends on abstraction.

By applying these principles methodically throughout the project and even during test planning, we were able to create a system architecture that is not only robust and easy to understand but also resilient to change and ready for further expansion.

**1.3 Planning the System Structure and Design Trade-Offs**

We started by identifying the main actors: Applicant, HDBManager, HDBOfficer and supporting components like Project, Application, and Enquiry. These were grouped into logical classes based on responsibilities. Use cases such as applying for a project or booking a flat were mapped to classes methods like applyProject() and bookFlat() respectively.

Early process flows were drafted informally to visualise step-by-step logic. At this stage, classes like User and Project had multiple responsibilities. We used direct associations instead of interfaces to keep the structure simple, and there was minimal abstraction or enforcement of design principles. These were conscious trade-offs to focus on basic functionality. Later, we refactored the code to better follow SOLID principles and improve modularity and maintainability.

#### **1.4 Code Implementation and Sample Code Snippets**

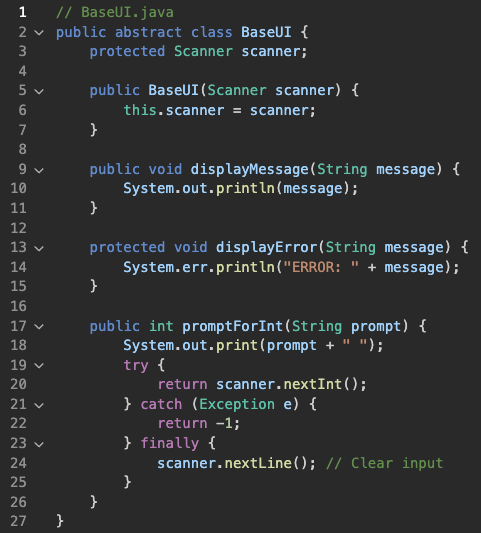
BTOMS was developed using *Java 17* but was made sure to be compatible with *Java 8*, following object-oriented design principles. Development was done in *Visual Studio Code*, with *GitHub* used for version control and collaboration.

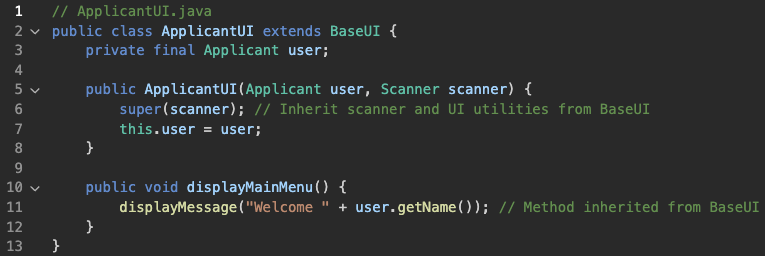
Our implementation demonstrates the application of core object-oriented concepts, including encapsulation, inheritance, polymorphism, interface-based design, and error handling.

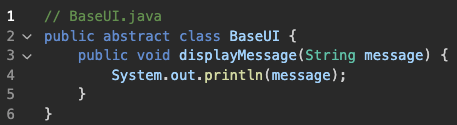
##### **1.4.1 Encapsulation**

Our classes, for instance, demonstrate **encapsulation** by keeping its fields private and exposing them only through public getters. This prevents direct access to internal data and ensures better control over how values are accessed or modified.

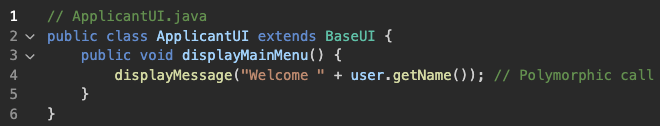
##### **1.4.2 Inheritance**





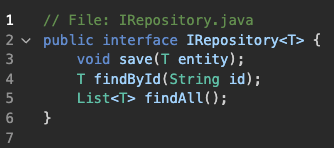
The class ApplicantUI inherits from BaseUI, reusing UI logic like displayMessage() and promptForInt(). This promotes reuse and cleaner code by centralising shared behavior.

##### **1.4.3 Polymorphism**



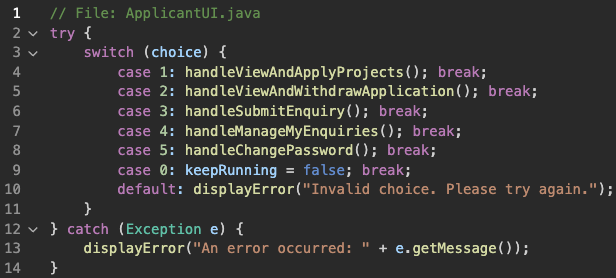
The method displayMessage() is inherited from BaseUI. Even though it is not overridden, calling it through a BaseUI reference to ApplicantUI enables runtime polymorphism, where the appropriate version is chosen at runtime based on the object type.

##### **1.4.4 Interface Use**



This demonstrates interface-based design. IRepository defines a contract, and BookingRepository provides a concrete implementation, enabling flexibility and dependency inversion.

##### **1.4.5 Error Handling**

This snippet shows error handling using a try-catch block to catch unexpected runtime exceptions during user interaction, ensuring graceful failure and feedback.

**2.0 Detailed UML Class Diagram: Design Thinking Process**

As described in Section 1.3, the main classes were identified via nouns in the project description, particularly under User’s Capabilities. Initially, main classes like User, Applicant, HDB Officer, HDB Manager, Project, Booking et cetera were detected, alongside attributes like Age, MaritalStatus, and ProjectId. The relationships between classes were ascertained via reflecting whether it is a ‘is-a’, ‘has-a’, or ‘uses-a’ relationship. For instance, Applicant, HDB Officer, and HDB Manager ‘is-a’ User, thereby highlighting that they should be subclasses of User, which should be a superclass. Our application of SOLID design principles introduced enhanced implementations of interfaces (i.e. IService, IRepository) and further separate classes based on respective responsibilities, culminating to five UML sub diagrams. The Entity UML sub diagram represents the fundamental models used for the system, while the Views UML sub diagram reflects user interface and presentation logic. The Controller UML sub diagram manages and directs user requests, where core business logic is detailed in the Service UML sub diagram. Data access and persistence is reflected in the Repository UML sub diagram.

Due to the large scale of the design, the UML diagrams into the five sub diagrams. Such as the SOLID design principles, if the UML diagram would be combined into a single UML diagram, view would depend on abstractions, and dependency inversion principles, services will depend on repository, controllers depend on services and finally views depend on the controllers.

### **3.0 Detailed UML Sequence Diagram**

For the UML Sequence Diagram, we thought of 3 specific scenarios that showcased the most critical parts of BTOMS. The first scenario reflects the authentication flow, where a user is logged into their specific role menu. The sequence diagram highlights processes like input validation, such as the handling of invalid inputs like incorrect NRIC formats or passwords.

The second scenario showcases project creation by HDB Managers. This is an important scenario as it involves many interactions between the controller-service-repository layers, such as calling the ProjectController to validate all the inputs, where it will then call ProjectService to handle all business logic, such as whether there exists a current project already. There are also various control layers to handle specific tasks such as generating the newly created project ID. Finally it will save the project to the file via the ProjectRepository.

The last scenario examines the entire flow of an HDB Officer applying for a project (as an applicant) and then registering to manage a project (as an officer).

### **4.0 Additional Features and Logical Assumptions**

Additional features such as main menu interfaces, user registration, (toggle) masked password entry, password strength validation were implemented to ensure our system’s completeness. When a user runs the program, they are first brought to the Main Menu page to login or register as an applicant. When entering their passwords, a default mask is applied to ensure enhanced privacy, defending against potential threats of password shoulder surfing. However, we acknowledge that it may lead to reduced visual feedback. Thus, we offer users the option to toggle this password mask. Moreover, when users change their password, they must comply with password strength criterias, thereby enhancing protection from unauthorised access.

Furthermore, our system requires married applicants’ preferred flat types, given their ability to apply for both two and three room flats. Subsequent flat approvals and bookings are completed based on this preferred flat type to cater to their needs and preferences, ensuring that they attain a suitable housing allocation. Additional details like how applicants would be unable to select unavailable flats (i.e. with zero units available), pending officer registration count for HDB Managers et cetera serves as functional safeguards to our system.

Additionally, logical assumptions were made to bridge the gaps between user capability and system implementation. For instance, applicants are only able to view project details with valid Application Opening and Closing dates (assumption of real-time current date). As such, while officer registrations can occur for upcoming projects, approved HDB Officers can only manage current active ones. Another example would be application withdrawal logic, where a subsequent applicant withdrawal for a BTO project would not increase the flat units available for the project. This reflects real-world BTO exercises, where balloting or subsequent selection processes tend to occur instead. Re-adding the unit immediately upon withdrawal could unfairly benefit applicants who apply later.

Last but not least, we determined the application approval logic. Our team determined that HDB Managers, who typically have higher authorities, should uphold the primary responsibility to screen and approve BTO applications. This implies that every applicant with a “SUCCESSFUL” status would eventually be “BOOKED” by the corresponding approved HDB Officer, lest project withdrawal. For simplicity, we assumed a First-Come-First-Serve approval process. However, we acknowledge that in the real world, other factors such as priority schemes (i.e. Married Child Priority Scheme), citizenship et cetera may contribute more towards initial screenings.

### **5.0 Test Cases and Results**

Our team performed manual testing rather than taking additional time to implement unit testing due to time constraints. Further testing frameworks like JUni could be implemented in the future should regular testing be required. The table below lists the test cases that have been tested.

|  |  |
| --- | --- |
|  |  |
|  | |

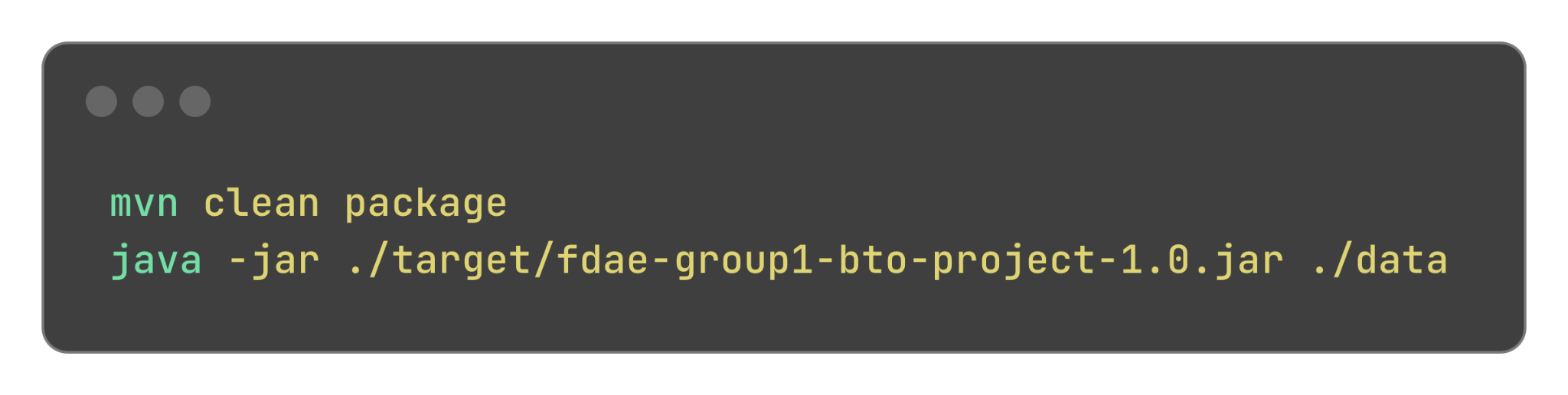
**6.0 Documentation**

**6.1 Javadoc**

* All public classes/methods documented with Javadoc
* HTML Javadoc files generated and included

**6.2 Developer Guide**

Ensure you have the Java Development Kit (JDK) and Apache Maven installed on your computer and follow the steps below:

1. Download the project from GitHub via the link at the Appendix.
2. Use Maven to build the project. Run:
3. The packaged jar build will be in the target folder, you can run the program with java -jar as shown in the image.

**7.0 Reflection and Conclusion**

#### **7.1 Our Successes**

Firstly, our team collaborated effectively using *GitHub* as it helped us avoid code conflicts, keep backups, and divide tasks by packages. We learned to create separate branches to commit changes and resolve merge issues constructively.

Secondly, by applying SOLID principles, especially SRP and DIP, we were able to keep each component focused and independent. This made debugging and iterative development easier, especially when handling complex flows like officer registration or flat booking. Moreover, separating user responsibilities into Applicant, HDBOfficer, and HDBManager through inheritance allowed for clear, enforced role logic across the system.

#### **7.2 Possible Improvements**

Initially, we only tested modules manually after implementing full flows. If we had set up automated testing or test-driven development (TDD) earlier, we could have caught edge cases sooner. Additionally, implementing data loading from .csv files introduced complexity particularly for when mapping nested structures like ProjectFlatInfo. Furthermore, certain features (i.e. officer slot validation) took longer than expected due to the additional implicit logic needed. An earlier visualisation of the sequential flows could mitigate this.

### **7.3 Individual Contributions**

**Tang Xinbo**  
Assisted in implementation and focused on comprehensive testing of program functionality across different user roles. Documented test cases and helped identify edge cases and logical bugs.

**Teo Wei Xiang**Acted as the team lead and contributed significantly to the implementation of core services such as Application, Officer Registration, and Report. Also assisted in integration across modules and resolving bugs during testing. Implemented sequence diagram to/from code and vice-versa.

**Toh Xin Yi**Implemented HDBManager responsibilities (relating to project, officer registration, report, application) and additional logical and system features. Documented testing procedures and made corresponding updates to code segments. Implemented UML diagrams to/from code.

**Vishal Jg**  
Supported development of service classes and focused on the construction and refinement of UML diagrams in Visual Paradigm. Ensured consistency between the codebase and visual design.

**Murugaraj Vishvaajit**Participated in service implementation, conducted comprehensive test case testing across user roles, and helped identify edge cases and logical bugs, while contributing to final report and presentation documentation.

#### **7.4 Lessons learnt & Final Thoughts**

One of our biggest takeaways was the importance of early planning and design. Designing the UML diagrams before writing any code was helpful as the visual reference provided a standardised guide and helped us avoid duplication. Whenever we violated OOD principles, we quickly saw the effect of cascading changes across multiple classes. Refactoring to follow better design practices helped minimise code dependencies, improve clarity and team collaboration. As we frequently altered code and diagram implementations given the trade-off between simplicity and extensibility, we learned to better balance theoretical knowledge and practicality.

**8. Appendix**

GitHub Repository Link: <https://github.com/VISHVAAJ001/SC2002-project>