SC/CE/CZ2002 Object-Oriented Design & Programming

**Declaration of Original Work for SC2002 Assignment**

We hereby declare that the attached group assignment has been researched,

undertaken, completed, and submitted as a collective effort by the group members

listed below.

We have honored the principles of academic integrity and have upheld Student Code

of Academic Conduct in the completion of this work.

We understand that if plagiarism is found in the assignment, then lower marks or no

marks will be awarded for the assessed work. In addition, disciplinary actions may be

taken.

| **Name** | **Course** | **Lab Group** | **Signature / Date** |
| --- | --- | --- | --- |
| **Xing Fanglin** | **SC2002** | **4** |  |
| **Nicholas Chow** | **SC2002** | **4** | **Nicholas** |
| **Charlene Ngiam** | **SC2002** | **4** |  |
| **Gan Wan Xuan** | **SC2002** | **4** |  |

**Reflection**

**1.1 Understanding the Problem and Requirements**

The Build-to-Order (BTO) Management System is designed to handle the entire lifecycle of a public housing application process in Singapore. The problem statement revolves around managing various aspects of BTO projects, applications, officer registrations, and providing access to relevant users such as HDB Managers, HDB Officers, and Applicants.

The key functional and non-functional requirements were derived from the problem statement in the project documentation. The major functional requirements include:

* There are three main stakeholders in this project: Applicant, HDB Officer and HDB Manager
* Project Management: HDB Managers should be able to create, edit, and delete projects.
* Visibility and Access Control: Projects must have a visibility setting, where only eligible applicants can see certain projects.
* Application and Approval Process: Applicants must be able to apply for projects based on eligibility criteria, and HDB Officers and Managers must be able to approve or reject these applications based on the available flats.
* Enquiry Handling: All users should be able to view and respond to enquiries related to the project.

#### **Explicit Requirements:**

* Role Management: Users must be classified into three roles: HDB Manager, HDB Officer, and Applicant.
* Project Operations: HDB Managers should be able to create, update, and delete projects. These projects must include details like name, neighborhood, number of available flats, opening/closing dates, and officer slots.
* Flat Booking and Application: Applicants can only apply to projects they are eligible for based on their age and marital status, and once approved, they should be able to book flats from the project.
* Enquiry System: All users should be able to send and respond to project-related enquiries.

#### Implicit Requirements:

* User Authentication: A login system is implied to ensure that only authorized users can access the system.
* Data Persistence: The system should be able to load and save data, such as projects and user details, using text files (e.g. users.txt and projects.txt).
* Usability: The system should provide a simple Command Line Interface (CLI) that is easy to navigate for all user roles.

Ambiguity:

One of the key ambiguities in the requirements was the statement that any applicant could apply to become an HDB Officer, which didn’t make sense, as applicants cannot become HDB Officers in real life. To resolve this, we interpreted that applicants cannot be HDB Officers at all, and we implemented a restriction to prevent applicants from taking on this role within the system. This ensures there is a clear separation between the two roles and avoids any conflicts of interest.

**1.2 Deciding on Features and Scope**

Core features:

1. **User Authentication and Role Management**: Different roles (HDB Manager, Officer, Applicant) must be able to log in and access the correct functionalities.
2. **Project Creation, Editing, and Deletion (CRUD Operations)**: HDB Managers can create, update, and delete BTO projects.
3. **Project Visibility Toggle**: HDB Managers can toggle the visibility of projects for applicants.
4. **Eligibility Check for Applicants**: Applicants must meet certain criteria (age, marital status) to apply for projects.
5. **Application and Flat Booking**: Applicants apply for projects, and if approved, they book flats.
6. **Officer Registration**: HDB Officers register for specific projects, with a maximum number of officer slots available per project.
7. **Enquiry System**: Users (Managers, Officers, Applicants) can send and reply to enquiries related to BTO projects.
8. **Reporting and Filtering**: HDB Managers can generate reports based on applicant details like marital status and flat preferences.

**Optional Features**:

1. **Advanced Search and Filtering for Projects**: Allows users to search and filter projects based on various criteria (location, flat type, available slots, etc.).
2. **Data Persistence (Text File-Based)**: The system saves data such as users and projects into text files, enabling the system to remember state across sessions.
3. **Customizable Reports**: The ability for the manager to create more detailed reports (e.g., summary of applicants by marital status, number of flats booked, etc.).
4. **Multiple Project Views for Managers**: Allows HDB Managers to view projects they have not created (but have access to).

**Excluded Features**:

1. **Database Integration**: Instead of using a full database (e.g., MySQL), our team decided to use simple text files to store data. This exclusion was made to keep the project simple and within the project scope, focusing on core OOP principles.
2. **Web Interface**: A full web interface was excluded in favor of a simple command-line interface (CLI). We decided that a CLI would be sufficient for demonstrating core functionality and principles without adding unnecessary complexity.
3. **Real-Time Notifications**: Features like email or push notifications were considered but excluded due to time constraints and the fact that the focus of the project was on implementing core business logic and object-oriented design.

By categorizing the features into core, optional, and excluded, our team ensured that we focused on implementing the most critical functionalities first. The core features aligned directly with the project requirements, while the optional features provided additional value without overwhelming the system with complexity. The excluded features were consciously omitted due to time and complexity constraints, but could be added in future versions of the system.

**2.1 Planning the System Structure**

We began by breaking the BTO Management System into logical components, focusing on key functional areas: user management, project management, application management, and enquiry management. Each module was assigned specific responsibilities and mapped to corresponding classes, such as Applicant, HDBOfficer, Project, Application, and Enquiry. This helped clarify the boundaries of each module and how they interacted with each other.

Next, we modeled the user flows by identifying key use cases like submit application, view project details, and make enquiry. I mapped these use cases to the system’s components, ensuring that each class interacted in a way that supported the corresponding user action. For example, the submit application use case linked the Applicant class with the Application class, while enquiry actions tied the Enquiry class to the Project class.

Finally, we created early visual models, including flowcharts, to outline the step-by-step process for critical operations like application submission and enquiry handling. These flowcharts visually represented the sequence of actions, decision points, and class interactions, ensuring the design was well-structured before finalizing the class diagram and moving to implementation.

**2.2 Reflection on Design Trade-offs**

**1. Simplicity vs Extensibility:**

Initially, we considered combining the controller and logic layer to simplify the codebase. This would have made the system quicker to implement and reduced the number of files to manage. However, we ultimately decided to separate them into distinct layers (controller, service, and model) to promote better maintainability and flexibility.

By keeping the controller layer separate from the business logic and data management layers, we were able to:

* Improve testability: This separation allowed us to write unit tests for the business logic without worrying about UI or input/output concerns.
* Achieve better scalability: It would be easier to add new functionality (e.g., integrating a database) without affecting the existing codebase.

#### 

#### **2. Time Constraints vs. Feature Completeness**

Given the limited project timeline, we had to decide which features were absolutely essential for the initial system, and which could be added later if time allowed. We prioritized core functionalities like project management, application handling, and user authentication, and deferred optional features like advanced search and customizable reports.

While we acknowledged that database integration or a web interface could significantly enhance the system’s functionality, these were excluded to maintain focus on the core features and deliver a working system within the timeline.

#### **3. Performance vs. Maintainability**

We decided to use text files for data storage (users.txt and projects.txt) instead of a database. This decision was based on:

* Feasibility: Text files were easier and faster to implement given the project’s constraints.
* Maintainability: While databases offer scalability, they were overkill for this application, and working with files allowed us to keep the design simple and focused on core object-oriented principles.

We also chose not to include features like real-time notifications, which would have been nice for improving user experience. However, this decision was made to avoid introducing unnecessary complexity into the system, especially when our focus was on demonstrating object-oriented concepts.

**3.1 Class diagram (Refer to github)**

In this class diagram, we translated the Build-To-Order (BTO) Management System into a logical object-oriented model by identifying key entities and their relationships. The main entities—such as User, Project, Application, Report, Registration, and Enquiry—were mapped to represent real-world concepts like users interacting with housing projects and submitting applications. By defining the responsibilities of each class, such as HDBManager handling project management and application approval, and ProjectServiceImpl managing project-related operations, we ensured clear separation of concerns. The use of interfaces (e.g., AuthService and Authenticatable) facilitates flexibility for future extensions, like adding new authentication methods or user roles. We applied key object-oriented design principles such as encapsulation, single responsibility, and dependency injection to ensure maintainability and scalability. This model is designed to allow easy integration of new features and future adaptability, with each class focusing solely on its designated role within the system.

**3.2 Sequence Diagram (Refer to github)**

### **1. Officer Registration to Handle Project**

**Justification**:  
 This sequence diagram was selected to represent the process of an HDBOfficer registering for a project in the system. It illustrates how the HDBOfficer interacts with the RegistrationService and ApplicationService to load projects and display them. The process is validated by checking project names and applicant data. The diagram helps to confirm that the registration process is functioning correctly and that HDBOfficers can only register valid projects while ensuring proper validation steps such as NRIC matching and status checks. This is crucial for the backend project management system and ensures the integrity of the registration process.

### **2. Officer Applies for BTO Project**

**Justification**:  
 This sequence diagram shows how an HDBOfficer attempts to apply for a BTO project, which is not allowed based on the system rules. The diagram demonstrates the role-checking mechanism through the RegistrationService, ensuring that the HDBOfficer cannot apply for the project, which is consistent with role-based access control in the system. By validating the officer's status and applying the correct restrictions, this diagram ensures that the isOfficer logic is correctly enforced, preventing officers from applying for a BTO project, thus maintaining the integrity of the application process.

### **3. HDB Officer Responsibilities within a BTO Project**

**Justification**:  
 The third sequence diagram focuses on the responsibilities of the HDBOfficer in selecting flats for applicants, handling enquiries, and managing the status of applications. It demonstrates the interaction between the HDBOfficer, RegistrationService, ApplicationService, and other entities such as Project and Receipt. The process includes validation of application status, flat allocation, and updating the project’s availability status. Additionally, it shows how HDBOfficers handle enquiries and generate receipts. This sequence is essential for ensuring that all steps of the flat allocation process are handled efficiently, from flat status verification to the creation of a Receipt for the applicant. It validates that the system correctly handles the integration of flat selection, status updates, and record creation, ensuring smooth operations in real-world project management.

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

**Single Responsibility Principle:**

In the BTO Management System, we ensured that each class had a single responsibility by separating functionality based on user roles. For example, the HDBManager class is dedicated to project management tasks, while the Applicant class handles only application-related actions. By following SRP, we made the system easier to understand, maintain, and extend, as each class focuses on a specific responsibility.

**Open/Closed Principle:**

The system was designed to be open for extension but closed for modification. A good example is the BTOProject class, which initially supports only 2-room and 3-room flats. However, it is easy to add new flat types in the future by extending the class without altering its core logic. This makes the system more flexible and scalable while preserving the integrity of existing features.

**Liskov Substitution Principle:**

We applied LSP by ensuring that subclasses of the User class, such as Applicant, HDBOfficer and HDBManager, can be used interchangeably wherever a User is expected. This guarantees that any subclass can replace its parent class without affecting system behavior. By following LSP, we improved **code consistency** and ensured **predictable behavior** when extending the system with new user roles.

**Interface Segregation Principle:**

To adhere to ISP, we defined smaller, role-specific interfaces like IUserService and IProjectService, ensuring that each class implements only the methods it needs. For instance, the IUserService interface focuses only on user-related functionality, avoiding unnecessary dependencies on unrelated features. This led to cleaner, more modular code and prevented classes from being burdened with irrelevant methods.

**Dependency Inversion Principle:**

DIP was implemented by ensuring that the SystemController depends on abstractions like IUserService and IProjectService, rather than concrete implementations. This approach decouples high-level logic from low-level details, making it easier to modify or replace underlying services without affecting the controller. It also enhanced testability and maintainability, as the system could easily be extended or modified without changing the controller's logic.

**4.1 Tools Used**

- Java 17: for backend implementation using object-oriented principles.

- Visual Studio Code (or IntelliJ IDEA): as the primary development environment.

- draw.io / Lucidchart: for creating UML diagrams like class diagrams and sequence diagrams.

- GitHub: for version control and team collaboration.

- Terminal & command line: for compiling and running Java applications.

**4.2 Sample Code Snippets**

Refer to github

**5.1 Test Strategy**

To ensure that the BTO Management System functions as intended, we adopted a combination of manual functional testing and integration testing. This approach allowed us to verify that both individual components and the entire system worked correctly. Testing was a critical part of our development process, as it helped us identify and address any issues early on, ensuring that the system was robust and user-friendly.

**Manual Function Testing:** We manually tested individual features and functionalities of the system to ensure they worked as intended from an end-user perspective. This involved simulating real-world user interactions, such as user registration, login, project creation, application submission, flat booking, and officer registration. Team members acted as different user types (Applicants, HDB Officers, and HDB Managers) to check whether they could access the right functionalities and perform their respective roles correctly.

Example: Testing the registration process for an Applicant by entering valid user data and ensuring the system allows the creation of a new account.

**Integration Testing:** We tested the interactions between the system’s components to ensure data flowed correctly across different modules (e.g., between User Management, Project Management, and Application Handling). After individually testing the core functionalities, we tested the integration of these functionalities. For example, after a HDB Manager created a new project, we checked if an Applicant could view that project and apply. We also tested if the HDB Officer could manage the application once submitted.

Example: A HDB Manager creates a project, and an Applicant applies for it, followed by the HDB Officer approving the application.

**Error Handling Testing**: We tested the system's ability to handle invalid inputs and error scenarios gracefully. Team members deliberately entered incorrect data or tried to access restricted actions (e.g., logging in with an incorrect password) to verify if the system displayed appropriate error messages. We also tested edge cases, such as attempting to create a project with missing details or booking a flat when no flats were available.

Example: Trying to log in with an invalid password and ensuring the system responds with "Invalid password" instead of allowing access or crashing.

**5.2 Test Case Table**

Refer to excelsheet

**6.1 Javadoc**

Refer to github

**6.2 Developer Guide**

Refer to github

**7. Reflections and Challenges**

What went well:

Throughout the development of the BTO Management System, the team made significant progress in applying Object-Oriented Design Principles (OODP) and successfully implementing the core features. The following aspects went well:

* Clear Role-Based Structure: One of the key strengths of the project was the clear distinction between the different user roles — Applicant, HDB Officer, and HDB Manager. By adhering to the Single Responsibility Principle (SRP), we were able to design a system where each class had a clear responsibility, making the code easier to understand and maintain.
* Separation of Concerns: The system architecture maintained a good separation between the controller, service, and model layers, adhering to the Dependency Inversion Principle (DIP). This made it easier to modify or extend the system without affecting other parts of the codebase. For example, changes made to the project management system did not affect user authentication logic.
* Role-Based Access Control: We implemented role-based access control effectively, ensuring that each user type could only access the relevant features. This helped keep the system secure and ensured that users had the right permissions for the actions they performed.
* Successful Use of Polymorphism: By leveraging polymorphism, we made the system more extensible. For example, the viewProfile() method was overridden in different subclasses (Applicant, HDBManager) to provide role-specific behavior, allowing us to treat different users uniformly while maintaining unique actions for each role.

What could be improved:

1. **Scalability and Performance:**As the system grows, performance could become an issue. Implementing more efficient data storage and retrieval mechanisms (e.g., using a database like MySQL or PostgreSQL instead of simple file-based storage) could improve scalability. Furthermore, optimizing code for performance, especially when managing large datasets, will be important for long-term use.
2. **Security Enhancements:**Since the system handles sensitive user data (e.g., NRIC, marital status), it would benefit from enhanced security measures such as encryption for sensitive fields, stronger authentication mechanisms (e.g., two-factor authentication), and secure data storage practices.

Individual contributions:

Fanglin: Worked on the class diagram, sequence diagram and the report

Charlene: Worked on the main part of the code, and helped to test the test cases

Wan Xuan: Worked on the class diagram, the code and helped with the test cases

Nicholas: Worked on the class diagram, the code and the report

Lessons learned about OODP:

Throughout the development of the BTO Management System, we gained valuable insights into the practical application of Object-Oriented Design Principles (OODP):

* Encapsulation: We learned how encapsulation allows for better control over the state of objects. By making fields private and providing getter and setter methods, we ensured that data could only be modified in a controlled manner, which helped prevent unwanted side effects.
* Inheritance and Polymorphism: These principles were key in making our system flexible and scalable. We saw firsthand how inheritance allowed us to create a common base class for users and extend it for different roles. Polymorphism allowed the same method to behave differently depending on the user type, making our system extensible without breaking existing code.
* Design for Change: We learned the importance of designing with future changes in mind. By adhering to the Open/Closed Principle (OCP), we made it easy to add new features, such as different flat types or additional user roles, without modifying existing code. This made the system easier to maintain and extend in the future.
* Separation of Concerns: Following the Single Responsibility Principle (SRP) and Dependency Inversion Principle (DIP) helped keep the code modular. We found that separating the controller, service, and model layers made it easier to maintain, debug, and test the system.

In conclusion, the development of the BTO Management System was a valuable learning experience in object-oriented design. The project’s role-based structure and modular design made it easy to manage and extend. While we successfully applied key OODP principles such as encapsulation, inheritance, polymorphism, and interface use, there were areas where we could improve, especially in terms of automated testing and error handling. The system is functional and well-structured, but there is always room for future enhancements, such as better data persistence and a more user-friendly interface.

**8. Appendix**

https://github.com/XFanglin/SC2002-Group-Project/commit/2fc554630646fb6a710b45df6299976e2dbfde2f