NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

SC2002 – OBJECT ORIENTED DESIGN AND PROGRAMMING

Project Title: Build-To-Order (BTO) Management System

Lab group: SCE2 Group 5

Declaration of Original Work for CE/CZ2002 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	Student ID
Ng Zhi Wei	U2420053E
Tan Jun Hao	U2322364F
Timothy Louis Barus	U2320344D
Sunkad Surabhi Sridhar	U2423726E

1. Design Considerations

1.1 Overview of our approach

In our (Built-To-Order) management system, we adopted a modular approach based on SOLID design concepts and Object-Oriented Programming (OOP) principles. The Boundary-Controller-Entity (BCE) model is used to build the system architecture, guaranteeing a distinct division of responsibilities between business logic, fundamental data structures, and user interfaces. To simplify user roles and actions, we used fundamental OOP ideas like abstraction, encapsulation, inheritance, and polymorphism. To encourage reuse and cut down on repetition, the 'User' class, for instance, acts as a foundation for 'Applicant', 'Officer', and 'Manager'. The Open-Closed Principle promotes extensibility, while each class abides by the Single Responsibility Principle. Our design makes the system scalable, robust, and maintainable by ensuring that adding new user roles or system features can be done with little disturbance.

1.2 Assumption made

We assumed all imported CSV data files into the system are complete and well constructed, such as proper field arrangement, correct enumerations (e.g., user type, flat type, status), and no missing required values like IDs or dates. Our system does not heavily sanitize the imported data, specifically at the initial process of importation. Corruption or inconsistencies that happen afterward are assumed to be manually fixed before the data is added again.

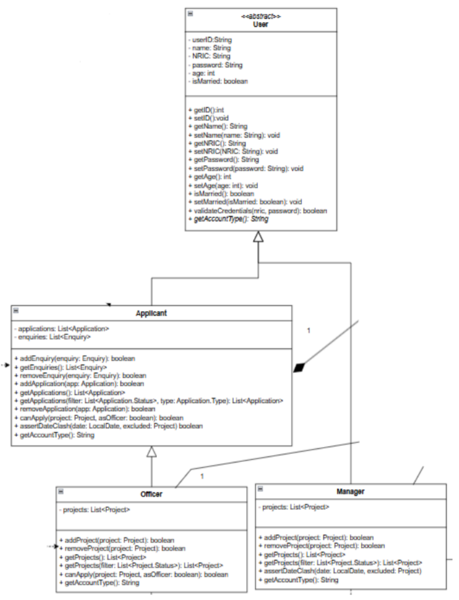
Secondly, we assumed that BTO CLI users possess basic knowledge of the domain of the system and the functionality, depending on the user type. i.e., Applicants know that there can be one project at the time submitted, Managers know that there can be listings managed, and inquiries can be replied to. The command lines were thus concise, and not much instructional or validation information was presented except where necessary.

Thirdly, we've assumed that the platform that the application is running on has all runtime dependencies, such that it has access to Java version 17 and necessary CSV files at the appropriate expected directory locations.

1.3 USE OF OBJECT-ORIENTED CONCEPTS

1.3.1 Abstraction

The system employs abstraction through the use of the *User* class becoming the overall frame for all the user categories.
It defines common attributes such as ID, name, NRIC, age, marital status, and password. To the extent that the user has all these attributes, the actions of each of the user roles (i.e., *Applicants*, *Officers*, and *Managers*) are abstracted into the subclasses.
This way, the complex interactions become easier since the software can handle all the users the same at the higher-level, the particular details being handled independently to each of the subclasses.

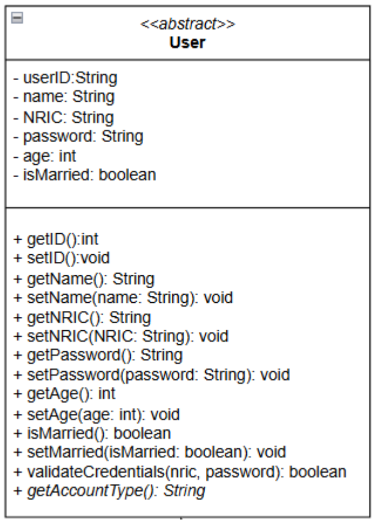


<<abstract>>
User
- userID:String
- name: String
- NRIC: String
- password: String
- age: int
- isMarried: boolean
+ getID():int
+ setID():void
+ getName(): String
+ setName(name: String): void
+ getNRIC(): String
+ setNRIC(NRIC: String): void
+ getPassword(): String
+ setPassword(password: String): void
+ getAge(): int
+ setAge(age: int): void
+ isMarried(): boolean
+ setMarried(isMarried: boolean): void
+ validateCredentials(nric, password): boolean
+ getAccountType(): String
Applicant
- applications: List<Application>
- enquiries: List<Enquiry>
+ addEnquiry(enquiry: Enquiry): boolean
+ getEnquiries(): List<Enquiry>
+ removeEnquiry(enquiry: Enquiry): boolean
+ addApplication(app: Application): boolean
+ getApplications(): List<Application>
+ getApplications(filter: List<Application.Status>, type: Application.Type): List<Application>
+ removeApplication(app: Application): boolean
+ canApply(project: Project, asOfficer: boolean): boolean
+ assertDateClash(date: LocalDate, excluded: Project) boolean
+ getAccountType(): String
1
1
Officer
- projects: List<Project>
+ addProject(project: Project): boolean
+ removeProject(project: Project): boolean
+ getProjects(): List<Project>
+ getProjects(filter: List<Project.Status>): List<Project>
+ canApply(project: Project, asOfficer: boolean): boolean
+ getAccountType(): String
Manager
- projects: List<Project>
+ addProject(project: Project): boolean
+ removeProject(project: Project): boolean
+ getProjects(): List<Project>
+ getProjects(filter: List<Project.Status>): List<Project>
+ assertDateClash(date: LocalDate, excluded: Project)
+ getAccountType(): String

1.3.2 Inheritance

Inheritance is applied consistently throughout the design of the system. User is the base abstract class to the classes *Applicant*, *Officer*, and *Manager*. It inherits necessary attributes like name, NRIC, and age, and behavior like password management, and extends new methods necessary for the specific jobs—*Officer* can handle projects whereas *Applicant* can book flats.
This application of inheritance prevents code duplication, promotes modular design, and properly reflects real world hierarchical relationships between the different user types of the BTO process.

1.3.3 Encapsulation

The encapsulation process is strictly maintained all the way through the system. Each class stores information in the private members and only provides necessary functionality using the public members. Sensitive information like passwords inside the User class is kept confidentially and accessed or modified using controlled functions like verifyPassword() or changePassword(). Similarly, internal members like unit numbers or status inside classes like Flat or Application never become accessible directly outside the class. This ensures data integrity along with controlled access.	

1.3.4 Polymorphism

Polymorphism is achieved primarily through the processing of the user by the abstract User reference. During the time of logging in, the system holds him/her as a simple User object but at runtime can be an Applicant, an Officer, or one of any Manager type.

This allows the system to access the users equally but keep each of their particular behaviors separate. For instance, at the moment of the log-in, role-specific dashboards are accessed according to the concrete class of the logged one. This kind of liberty makes the system scalable, where new kinds of users can be introduced without changing the fundamental log-in process.

One such implemented instance of polymorphism would be the canApply() function in both Applicant and Officer, where Applicants canApply() and the Officer's canApply() function are different, where the Officer's canApply() function does extra checks to check if the officer is already a officer of a project he is applying to.

1.4 DESIGN PRINCIPLES

1.4.1 Single Responsibility Principle (SRP)

When each class is designed, they are designed with only one responsibility in mind. This ensures that when a class is changed, it is only for the tasks related to that responsibility.
For example, ProjectController, ApplicationController, UserController is only responsible for the manipulation and business logic of their respective class, Project, Application, User. For instance, while *Applicant* class holds application and enquiry listings specific to the user, it doesn't get involved with application or withdrawal—it's taken care of as should-be in *ApplicationController*.

1.4.2 Open-Closed Principle (OCP)

This strategy is based on the principle that software components should be extensible without requiring changes to previous code. In the architecture, the *User* class is used as the base framework from which other user types can derive, such as *Applicant*, *Officer*, or *Manager*.This simplifies adding future enhancements; adding additional levels of authority or types of users is achievable through subclassing without having to alter base class. Adding a new type of user, such as a Housing Administrator, will have special behavior but still inherit from User without negating pre-established roles. It also makes it more maintainable as enhancements are added over time.

1.4.3 Liskov Substitution Principle (LSP)

The Liskov Substitution Principle demands that superclass objects are substitutable with objects of their subclasses without affecting system functionality. This is ensured by polymorphism in the user module. In our design, both *Officer* objects and *Applicant* objects can be treated as Applicant types in generic processes such as HDB Application or enquiry submission. The system will treat subclass objects in a similar manner by using references to their superclass type with promises of appropriate, predictable behavior regardless of which specific subclass is used.

1.4.4 Interface Segregation Principle (ISP)

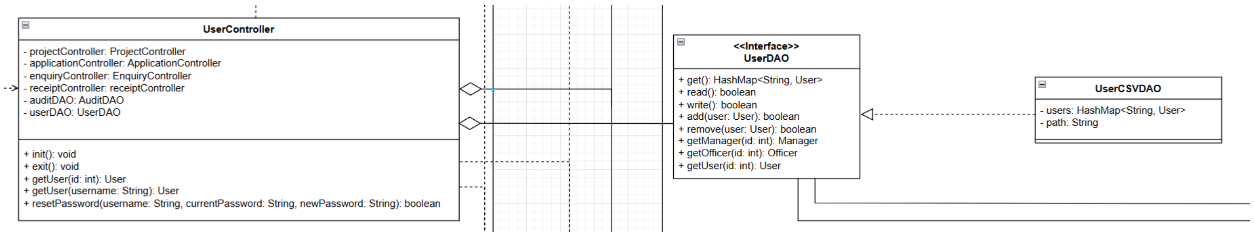
Interface Segregation Principle (ISP) prefers defining several specialist interfaces over one fat interface. The principle is successfully implemented in this system by using narrow single-purpose interfaces such as *Reader*, *Writer*, *InitRequired*, and *ExitRequired*. These are equivalent to different contracts pertaining to separate functional concerns of the system. For example, DAOs such as UserDAO, ApplicationDAO, and EnquiryDAO implement only *Reader* and *Writer*, and thus are never required to have responsibility for persistence behavior with unwanted concerns. Similarly, controller objects implement *InitRequired* and *ExitRequired* for clear specification of initialization and shutdown duties. This makes it more modular, less likely to implement unwanted methods in foreign classes, and to have high cohesion. Following ISP, it is quite easy to specify finer contracts, test, and evolve later.

1.4.5 Dependency Inversion Principle (DIP)

According to the Dependency Inversion Principle, modules of higher abstraction levels are not to depend upon those of lower abstraction levels; rather, they are to depend upon abstract modules. Even in the absence of formal interfaces that promote abstraction across layers, the system is in accordance with this principle by maintaining loose coupling among classes.

For instance, we implemented an abstraction layer between the reading, writing and storage of data in the system, allowing for greater flexibility in switching between different storage mechanisms (eg. CSV, database or cloud storage), without altering the core logic of the application.

Implementation:


UserController
- projectController: ProjectController
- applicationController: ApplicationController
- enquiryController: EnquiryController
- receiptController: receiptController
- auditDAO: AuditDAO
- userDAO: UserDAO
+ init(): void
+ exit(): void
+ getUser(id: int): User
+ getUser(username: String): User
+ resetPassword(username: String, currentPassword: String, newPassword: String): boolean
<<Interface>>
UserDAO
+ get(): HashMap<String, User>
+ read(): boolean
+ write(): boolean
+ add(user: User): boolean
+ remove(user: User): boolean
+ getManager(id: int): Manager
+ getOfficer(id: int): Officer
+ getUser(id: int): User
UserCSVDAO
- users: HashMap<String, User>
- path: String

2. UML Diagram

2.1 UML Class Diagram

A clearer image is available on GitHub, named Class Diagram.draw io.png.

2.2 Sequence diagram

A clearer image of each sequence diagram is available on GitHub

2.2.1 Sequence Diagram 1: HDB Officer Applying for a BTO Project (as an Applicant)

This scenario demonstrates how role duality is handled in the system—where an officer can also act as an applicant. By representing an officer applying for a flat, we validate that role-based restrictions and data flows are correctly implemented even when users take on multiple roles.

2.2.2 Sequence Diagram 2: HDB Officer Registering to Handle a Project

This scenario was chosen to highlight the system's approval workflow and how officers gain access to project-related functionalities. It helps validate the role authorization design, where registration requests must pass through managerial approval before access is granted.

2.2.3 Sequence Diagram 3: Manager Approving Officer to Handle Project

This scenario was chosen to complete the Officer Approval Process in Sequence Diagram 2.

2.2.4 Sequence Diagram 4: isEligible method

This scenario was chosen to show how various objects interact with each other to perform checks to check if the user is allowed to apply for this project. It shows method overriding, as when canApply is called by an Officer, the first opt fragment is run, then calling the super canApply in the Applicant class.

3. Functional Tests and Results

We have conducted unit testing to ensure all the individual components fulfill all the necessary requirements before proceeding to integrated testing. Our unit testing focused on verify main functionality system login functionality, manager functionality,officer functionality ,applicant functionality and withdrawal functionality

3.1 System Login Functionality

Test Case	Test Description	Expected Result	Pass/Fail
1	[Valid Login] User enters correct NRIC and password	User logs in successfully and the appropriate role-based menu is displayed	Pass

2	[Invalid Login] User enters wrong NRIC or password	Error message is shown and access is denied	Pass
3	[Change Password] User changes password by entering old password followed by new password	Password is changed successfully, user is logged out and can log in with new password	Pass

3.2 Manager Functionality

Test Case	Test Description	Expected Result	Pass/Fail
1	[Create BTO Project] Manager creates new project with all required details	Project is created successfully and appears in the project list	Pass
2	[Edit Project] Manager edits project details including visibility toggle	Project details are updated successfully in the system	Pass
3	[View All Projects] Manager views all projects including those created by other managers	Complete list of all projects is displayed regardless of creator	Pass
4	[Filter Projects] Manager filters to see only projects they created	Only projects created by the logged-in manager are displayed	Pass
5	[Process Officer Registration] Manager approves/rejects officer registration requests	Officer registration status is updated and slots count is adjusted correctly	Pass
6	[Process Application] Manager approves/rejects applicant applications	Application status is updated correctly based on flat availability	Pass
7	[Process Withdrawal] Manager approves/rejects withdrawal requests	Withdrawal status is updated and applicant can apply for other projects if approved	Pass
8	[Generate Reports] Manager generates reports with various filters	Correctly filtered reports are generated showing relevant applicant data	Pass

3.3 Officer Functionality

Test Case	Test Description	Expected Result	Pass/Fail
1	[Register for Project] Officer registers to handle a project	Registration is submitted with PENDING status and awaits manager approval	Pass
2	[View Registration Status] Officer checks registration status	Current status (PENDING or APPROVED) is correctly displayed	Pass
3	[View Project Details] Officer views details of project they're handling	Project details are visible regardless of visibility setting	Pass
4	[Cannot Edit Project] Officer attempts to edit project details	System prevents editing and shows appropriate message	Pass
5	[Apply for Different Project] Officer applies for a project they're not handling	Application is accepted and processed normally	Pass
6	[Cannot Apply for Handled Project] Officer attempts to apply for project they're handling	System prevents application with appropriate error message	Pass
7	[View and Reply to Enquiries] Officer views and responds to enquiries	User can see the reply from the officer	Pass
8	[Process Flat Booking] Officer retrieves successful application and updates status	Flat availability is decreased, application status changed to BOOKED	Pass
9	[Generate Receipt] Officer generates receipt for booked flat	Receipt contains all required details of applicant and booking	Pass

3.4 Applicant Functionality - Married

Test Case	Test Description	Expected Result	Pass/Fail

1	[View Available Projects] Applicant views list of projects	Only VISIBLE projects applicable to married status are shown	Pass
2	[Apply for Project] Applicant applies for a project with 3-Room flat	Application is submitted successfully with PENDING status	Pass
3	[Cannot Apply for Multiple Projects] Applicant attempts to apply for second project	System prevents multiple applications with appropriate error message	Pass
4	[Submit Enquiry] Applicant submits enquiry about a project	Enquiry is submitted successfully and wait for the reply from officer or manager	Pass
5	[Edit Enquiry] Applicant edits previously submitted enquiry	Enquiry content is updated successfully	Pass
6	[Delete Enquiry] Applicant deletes previously submitted enquiry	Enquiry is removed from the system	Pass
7	[View Application Status] Applicant checks application status	Status is correctly displayed (PENDING/SUCCESSFUL/UNSUCCESSFUL/BOOKED)	Pass
8	[Request Withdrawal] Applicant requests to withdraw application	Request is submitted successfully	Pass

3.5 Applicant Functionality - Single

Test Case	Test Description	Expected Result	Pass/Fail
1	[View Available Projects] Single applicant views list of projects	Only VISIBLE projects with 2-Room flats are shown	Pass
2	[Cannot Apply for 3-Room] Single applicant attempts to apply for 3-Room flat	System prevents application with appropriate error message	Pass

3	[Apply for 2-Room] Single applicant applies for a 2-Room flat	Application is submitted successfully with PENDING status	Pass

3.6 After-Withdrawal Functionality

Test Case	Test Description	Expected Result	Pass/Fail
1	[View Withdrawal Status] Applicant with approved withdrawal checks status	Status shows as WITHDRAWN	Pass
2	[Apply After Withdrawal] Applicant applies for new project after withdrawal	New application is accepted and processed normally	Pass

4. Reflection

4.1 Difficulties Encountered

One of our main challenges was role-based navigation between Applicants, Officers, and Managers. All had some level of access to views and functionality. We were able to address this problem by clearly defining their boundaries and allowing the UserController to keep track of the logged in user such that role-based operations were performed.

Second, we had to keep application status transitions in sync throughout the system. Because the Application object would itself be in PENDING, BOOKED, or WITHDRAWN states, used by multiple controllers, we needed to propagate changes to associated objects such as Flat, Project, Applicant, etc. We tackled this problem by tightly coupling the status changes to operations such as book() and approve() within the class Application.

4.2 Knowledge Learnt

Through this project, we gained an in-depth understanding of how object-oriented principles translate into real-world software design. Concepts like abstraction, inheritance, and polymorphism became more intuitive as we applied them to practical scenarios. We also learned how crucial it is to separate concerns using design principles such as SRP and OCP to maintain scalability and code readability.

4.3 Further Improvements

One key area of improvement would be implementing formal interfaces more consistently across the system to reinforce loose coupling and improve testability. We could also enhance error-handling mechanisms for greater user feedback during invalid operations. Lastly, introducing persistent data storage using a database or file I/O would improve realism and data continuity, especially for multi-session interactions, making the system more robust and closer to a production-level application.

5. Appendix

Github link:https://github.com/ngzhiwei517/SC2002.git