**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** | **Student ID** | **Course** | **Lab Group** | **Signature / Date** |
| Chin Wei Hao | U2322704F | SC2002 | FCSB | 19-4-25 |
| Georgina Chew Yee Chin | U2321902D | SC2002 | FCSB | 20-4-25 |
| Gu Jia Wei | U2422607L | SC2002 | FCSB | 20/4/2025 |
| Joel Ng Wei Heng | U2422193L | SC2002 | FCSB | 20-4-25 |
| Koh Qie Min | U2140345L | SC2002 | FCSB | 20-4-25 |

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

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

We began by reading the project document, highlighting all use cases, test cases and system requirements. Recurring keywords such as “apply”, “register”, “book”, “view” and “manage” helped us determine that the system’s core function revolves around BTO project applications by “Applicant” and user role-based access control by “Officer” and “Manager”. Based on this information, we created a list of essential features and identified the user roles, controllers, databases, and interfaces we might need when coding the project.

In addition, we inferred some implicit expectations. We assume the system should provide data validation, handle error scenarios properly, and ensure security through role-based access control. We also inferred that the system should follow a logical navigation structure for CLI interactions between the user and the system.

Some missing parts in the project brief included, for example, whether every applicant can submit an enquiry about any projects, whether an officer could unregister for a project after registering and manager access when they have an active project. These uncertainties were resolved by referring to the FAQ provided and through team discussions. We reached a consensus on how we should interpret and approach these various problems, ensuring that the features we implement will align with the overall system goals and ensure consistency across different components when splitting the work within the team.

* 1. **Deciding on Features and Scope**

Following that we prioritized features that were considered core to the system based on the potential impact it has on the user experience and their contribution to the system’s overall functionality. The potential features that we identified were

* Core
  + User Authentication with role-based menus
  + Create, edit and delete BTO projects
  + View Projects
  + Apply, withdrawal from Project
  + View Application status
  + Project Submission
  + Project browsing and search filters
  + Submit, edit and delete enquiry
  + Databases to store user data, login data, application data, project data
  + Input validation e.g. date format
  + Various controllers to check for the various conditions
  + Password Changing (When logged in)
* Optional
  + User Registration for various roles
  + Password Hashing
  + Password Changing (If Using Default Password)
  + User profile editing
  + Advanced search filters
* Excluded features
  + Password Encryption was excluded due to time constraints. We were unable to complete the implementation of encryption within our project timeline.
  + Password masking was not implemented as we initially used scanner for input which does not support masked input. We tried switching to console based input, but it doesn’t function properly as our application is not running in a proper terminal which is required for System.console() to work.

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

**2.1 System Structure**

In early planning phase, we adopted a monolithic architecture, where all components of the system reside within a single deployable unit, which is suited for the command-line interface (CLI) nature of the BTO system. Before implementation, we decided on using the Model-View-Controller (MVC) design to structure the BTO system. We analysed the requirements and split them up into Model, View and Controller components. The Model layer includes all entity classes such as User, Applicant, Officer, Manager, Project, etc. It corresponds to all the data-related logic the user works with. The Controller Layer includes all controller classes such as ProjectController, EnquiryController, AuthenticationController , etc. It enables interconnection between the view and model by processing all the business logic and incoming requests, manipulate data using the Model component, and interact with the Viewto render the final output. The View layer is implemented through Handler classes which interact directly with user via CLI. It passes input to controllers for processing and determines which options are presented based on user roles. By using MVC principles, we created a clean separation of concerns. This made our system easier to debug, extend and even redesign interfaces without having to rewrite core business logic.

**2.2 Design Trade-offs**

While creating the BTO system, we often traded off simplicity for extensibility and maintainability. For example, merging controller and logic layer could make codebase simpler, but we separated them to ensure maintainability and easier testing as well as debugging. We also chose to store lightweight references such as NRICs or names rather than the full User and Project Objects in associated classes. Even though the latter would have allowed us to access associated information directly, using identifiers would help to reduce coupling between classes, which made the codebase more flexible. We initially discussed using design patterns like Singleton, but we chose to avoid overengineering and focused on clean code structure for a monolithic CLI application.

**Chapter 3: Object-Oriented Design**

**3.1 Class Diagram (Design Thinking)**

We started by identifying the nouns found in the problem, which usually represents classes. Applicant, Officer, and Manager were found to be the primary User roles while Project, Enquiry, Application were entities to be mapped to classes. Verbs like view, edit, delete, submit, toggle provided potential methods within classes. The logic and control flow defined in the assignment led to identifying controller and handler classes, such as Project Controller, Enquiry Controller, and various Action Handler implementations.

The responsibilities of each class were:

* User & Subclasses (Applicant, Officer, Manager): Represent system users with shared attributes and subclasses extending this with role-specific responsibilities.
* Project: Maintains information on BTO projects
* ProjectApplication & OfficerApplication: Store application-related data for applicants and officers respectively.
* Enquiry: Represents questions submitted by applicants, and responses from officers/managers.
* Controllers (e.g., ProjectController): Handle business logic, such as project creation, assignment, eligibility checks, and data updates.
* Handlers (e.g., ApplicantActionHandler): Act as boundary classes for handling user inputs and actions during runtime
* Database (e.g., ProjectDatabase): Abstracted file-based persistence classes that read/write entity data.
* Utility Interfaces: Handle date operations (DataTimeController), input parsing (GetInput) and filters actions for users (ActionHandler)

For the relationship between classes, we determined it by asking whether classes have a “has-a” , “is-a” or “uses-a” relationship. We use inheritance for role hierarchies, such as Applicant extending User, which have a “is-a” relationship. We distinguish between association, aggregation, and composition which have a “has-a” relationship based on ownership and lifecycle, whereas dependency has a “uses-a” relationship.

During planning, we made trade-offs for simplicity vs. flexibility. We chose to store attributes instead of embedding user objects (e.g., officerNRIC instead of Officer) in places like Project to reduce coupling. However, additional lookups in the database needed when full information was required.

**3.2 Sequence Diagrams**

As per the assignment syllabus, the interactions of the officer role for project registration and application for a BTO will be demonstrated using sequence diagrams. As each process has multiple elements that it interacts with, for clarity and readability the functions have been split into smaller subprocesses: Account Registration, Login, Viewing of Projects, Project Registration, Project Applications, and Application for BTO. All the sequence diagrams will be provided as an attachment at the appendix as well as including the image files in the submission.

To start off, the account for the officer must be available for the user, to do so it involves the registration process. As seen in figure A1 the officer will have to interact with the system once it initializes to select registration. After which the user will be locked into an input loop where a particular is prompted for, this can be NRIC, Name, Marital Status, password, admin password. At each junction if the data supplied fails the validator tests, the user will be prompted again, if it happens 3 times on a particular input, the registration process terminates early and returns the user to the main menu.

Afterwards, it's the login session in figure A2. The account must be verified against the information stored in the database and return the correct user and login. Since the information is stored in the database as hashed passwords, every password attempt given by the officer is also hashed and goes through a similar input loop as in the previous sequence diagram, albeit abstracted away for readability. On figure A3, the user is displayed a menu that is attached to his user object and passed to views. The menu controller invokes a handler for its subsequent actions and is what allows for the other methods available. The first shown being the view of the available projects filtered by the role asking the database.

The officer can then interact again with the menu and handler to apply for the project. In figure A4, the application is started off by asking for the project name, which was learned through the prior function. The handler then invokes officer and the project controller to work together to ensure that the officer has no applications beforehand and is available, thus creating the application and submitting it to the corresponding database. Like view projects, the officer receives the appropriate view from the filtered database by id in figure A5, to view his project registrations.

Finally, as a parallel in figure A6, the application process for the officer is similar, but the user must modify their role first. The officerActionHandler will instead do some self-checks on the eligibility for application based on the user’s data fetched from officer, as well as project information fetched from the controller and database. The officer applying can then choose his desired project and flat type subject to availability, then creating an application in the appropriate database and signalling success to the user if all validation checks pass.

**3.3 Application of SOLID Principles**

1. **Single Responsibility Principle (SRP)**

The main idea of SRP is that a class should have only one reason to change meaning that it should only exist for a single reason. This can be seen from our implementation of AuthenticatorController which exists just to handle user authentication logic, such as verifying login credentials and validating inputs during account creation. Applying SRP made the codebase more organized and easier to test but increased the number of classes we had.

1. **Open-Closed Principle (OCP)**

We have an abstract super class User, which allow each role to implement specific behaviours without modifying the base class. So, if the developer adds more different types of users into the project, they can just extend the super class to create new subclasses with the role specific features needed. By applying OCP allows the system to be open for extension and closed for modification. This allows it to grow in functionality while minimizing the risk of breaking existing features. However, this approach takes more time in the planning phase to ensure that the system is properly designed for extensibility.

1. **Liskov Substitution Principle (LSP)**

User is an abstract superclass, extended by Manager, Applicant and Officer (which is a further extension of Applicant). Every User sub class implements the abstract methods defined in User (e.g. displayMenu) and only adds its own subclass-specific methods thereafter. There is no restriction nor removal of the initial implementation of the User class in its subclasses. By adhering to LSP, the subclasses can be safely used wherever a User is expected without breaking the program. This allowed our code to be more predictable, with its loosely coupled code.

1. **Interface Segregation Principle (ISP)**

The ApplicantActionHandler is a class that implements the ActionHandler interface. It makes full use of all functions defined by ActionHandler and inherits its default method. Having a default method follows ISP, as it provides an optional, opt-in function that not all implementing classes may use, preventing unnecessary or forced empty implementations. Implementing this added time to the designing and planning phase as we had to consider methods needed by every class, and how we could abstract them. However, it helped prevent bulky interfaces with forced implementations to appease the compiler, and improved flexibility of the code.

1. **Dependency Injection Principle (DIP)**

By using interfaces such as ActionHandler, PasswordReset, and Filter, it decouples high-level logic from specific implementation. Adhering to DIP has helped improve the code flexibility and allows different user roles to interact with the system through shared contracts, rather than concrete classes. Overall, DIP has helped identify a good balance between abstraction and practicality.

**💻 Chapter 4: Implementation (Java)**

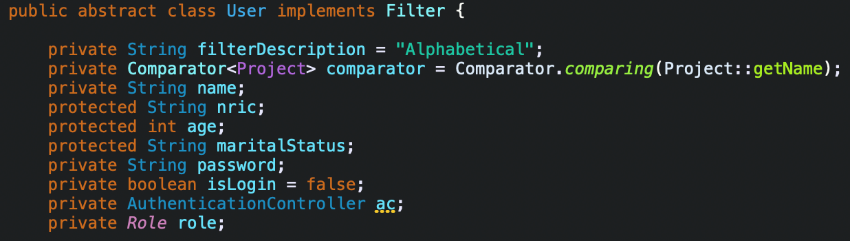
**4.1 Tools Used:**

Java 17 | IDEs: Eclipse, VS Code | Version Control: GitHub

**4.2 Sample Code Snippets:**

**• Encapsulation**

Superclass variables are protected if subclasses require access, otherwise they are kept private.

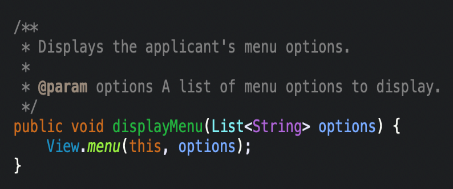


**• Inheritance**

Applicant subclass extending User and inheriting its state and behaviour, implementing its abstract methods, and adding its own methods. A screenshot of a computer screen

AI-generated content may be incorrect.A screen shot of a computer

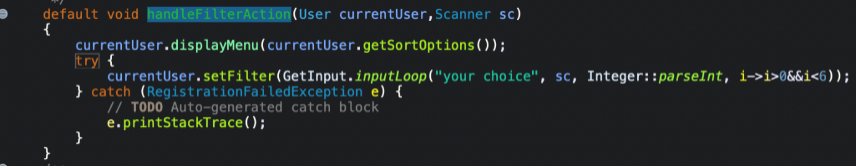
AI-generated content may be incorrect.



**• Polymorphism**

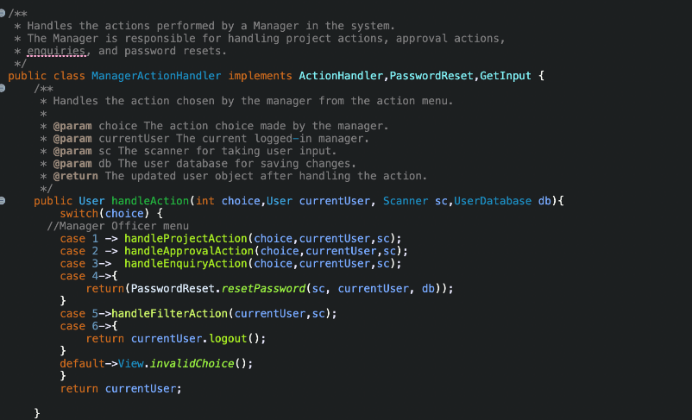
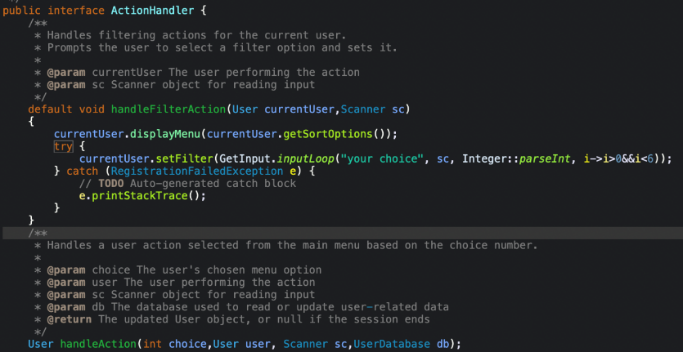
User implements the Filter interface, allowing a User instance to use setFilter, a method implemented only in Filter class.





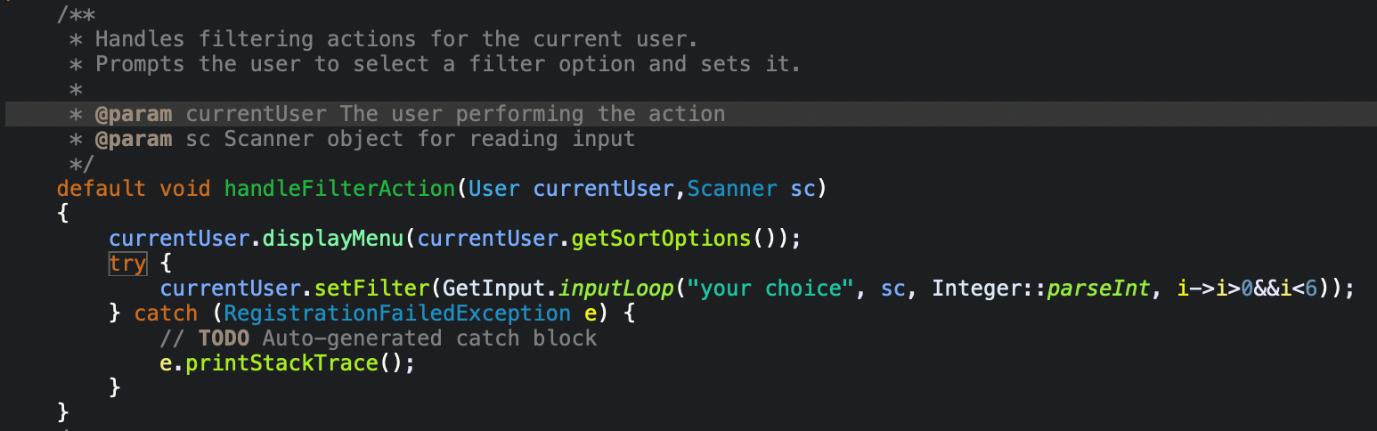
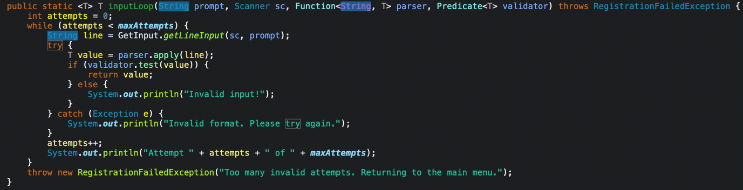
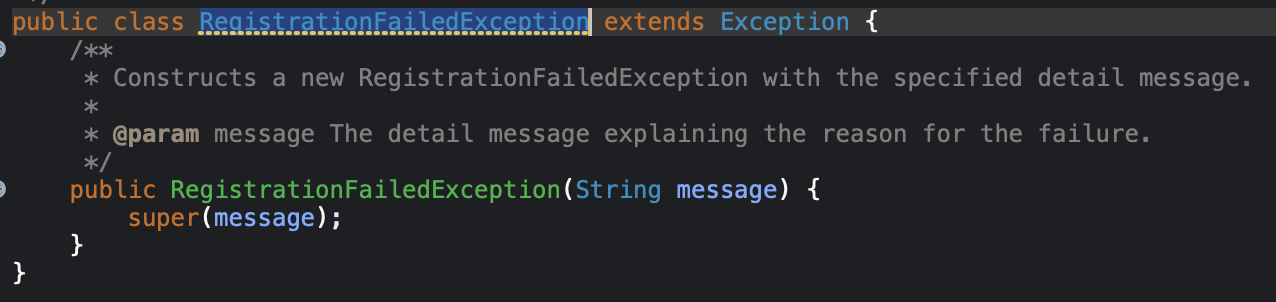
**• Interface use**

ActionHandler interface, implemented by various subclass ActionHandlers. Here, it is used by ManagerActionHandler, inheriting its default method, and implementing the abstract handleAction function.



**• Error handling**

A custom exception is specified to be thrown when a number of invalid inputs (maxAttempts) are given.



**Chapter 5: Testing**

**5.1 Test Strategy**

Manual Functional Testing, Positive & Negative Testing, and Role-Based Access Testing were used. Manual Functional Testing is used to test the BTO system manually to ensure it behaves as expected based on requirements. Positive & Negative Testing is used to ensure the system works with valid input or how the system responds when there is invalid input. Role-Based Access Testing is used to check if users are shown role-specific dashboards after login.

**Tools Used:** Microsoft Excel for writing and tracking test cases.

**5.2 Test Case Table**

**A screenshot of a computer

AI-generated content may be incorrect.A screenshot of a computer

AI-generated content may be incorrect.**

A screenshot of a computer

AI-generated content may be incorrect.A screenshot of a computer

AI-generated content may be incorrect.A screenshot of a computer

AI-generated content may be incorrect.A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.A screenshot of a computer

AI-generated content may be incorrect.A screenshot of a computer

AI-generated content may be incorrect.A screenshot of a computer

AI-generated content may be incorrect.A screenshot of a computer

AI-generated content may be incorrect.A screenshot of a computer

AI-generated content may be incorrect.A screenshot of a computer

AI-generated content may be incorrect.A screenshot of a computer

AI-generated content may be incorrect.A screen shot of a computer

AI-generated content may be incorrect.

****

**Chapter 6: Documentation**

**6.1 Javadoc**

[**https://github.com/Jiawei-131/SC2002-BTO-System/tree/main/doc**](https://github.com/Jiawei-131/SC2002-BTO-System/tree/main/doc)

**6.2 Developer Guide**

* **Prerequisites:** JDK 17+, Java-compatible IDE, Git or ZIP download to access project.
* **Project Structure:** BTOSystem/src with packages: Controllers, Data, Entities, Handlers, Util, View, and BTOSystem (main).
* Refer to UML Class & Sequence diagrams and JavaDoc for logic, class relationships, and execution flow

**Chapter 7: Reflection & Challenges**

We separated the work quite well between us by assigning different responsibilities with some, working on the menu, user related functionalities and while another worked on code related to enquiry and project management. While the rest working on the UML class and sequence diagram. This division of work allowed us to work in parallel and avoid overlapping task which improved efficiency by reducing Git conflicts.

One of the toughest parts of this project would be integration. A lot of the classes were dependent on each other's design, such as the expected input and available outputs. It was a rough start trying to have everyone on the same page when it came to which class did what, but once we had a flow, it was easier from there.

By working on this project, we learn that OODP allows for a more coherent, flexible and predictable program, especially with the SOLID principles. This makes it easier to maintain, test, and expand on. We also realised that early planning of class interactions using UML diagrams can save time during integration and reduce miscommunication within the team.

**Chapter 8: Appendix**

[**https://github.com/Jiawei-131/SC2002-BTO-System**](https://github.com/Jiawei-131/SC2002-BTO-System)