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	Role	Signature / Date
Gerald Chan Weiheng	U2421492H	SC2002	FDAE	Co-Project Manager/Lead Developer	[illegible]
Goh Yue Jun Bradley	U2421395F	SC2002	FDAE	Co-Project Manager/ Documentation Lead	[illegible]
Swedha Prabakaran	U2420559B	SC2002	FDAE	UML Developer	[illegible]
Cherilynn Kong Qi Hui	U2422232G	SC2002	FDAE	Tester Lead	[illegible]
Vona Tanisha	U2422590D	SC2002	FDAE	Assistant Developer	[illegible]

Important notes:

1. Name must EXACTLY MATCH the one printed on your Matriculation Card.
2. Student Code of Academic Conduct includes the latest guidelines on usage of Generative AI and any other guidelines as released by NTU.

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

Chapter 1: Requirement Analysis & Feature Selection

1.1 Understanding the Problem and Requirements

We created a list of essential features based on reading and highlighting use cases and system requirements, identifying user roles and system entities. The BTO Management System intends to facilitate the handling of Build-To-Order housing applications in Singapore's public housing system. The **main problem** identified was the management of the BTO application process: project creation, application submission and management, application processing, flat booking, and enquiry system. The system needs to support three distinct user roles (Applicant, HDB Officer, and HDB Manager) with different permissions and capabilities within a single platform.

Explicit requirements given: User authentication with NRIC-based login (starting with S or T, followed by 7 digits and ending with a letter), Role-based access control (Applicant, HDB Officer, HDB Manager), Project management functionality (creating, viewing, toggling visibility), Eligibility-based application rules (age and marital status restrictions for flat types), Application status tracking (Pending, Successful, Unsuccessful, Booked), Flat booking process management, Enquiry submission and response system, Report generation capabilities, Command Line Interface (CLI) application

Implicit expectations inferred from context: File-based data persistence storage, Clear user interface for different user types, Process validation to ensure business rules are enforced, HDB Officer having dual role capabilities (can apply as an applicant)

Ambiguous or missing parts: Details on concurrent user access (we assumed single-user access at a time), Performance requirements for data retrieval (we assumed small data volumes), Specific error handling mechanism (eg. what error message to output).

1.2 Deciding on Features and Scope

We grouped features into three categories: core, optional, and excluded, to ensure core features demonstrated strong object-oriented principles without overcomplicating the system.

Prioritization criteria: Features were prioritized based on importance to the core functionality, alignment with object-oriented principles, implementation complexity, and available development time.

Full List of Potential Features Identified:

Core features (essential, must implement): ✅ User authentication with role-based access (Applicant, HDBOfficer, HDBManager) ✅Project creation and management by HDB Managers ✅Application submission by applicants with eligibility checks ✅HDB Officer registration for project assignment ✅Application approval/rejection by HDB Managers ✅Flat booking process with receipt generation ✅Enquiry submission and response system ✅ File-based data persistence

Optional or bonus features (nice to have, only if time permits): 🟡 Enhanced reporting capabilities with multiple filtering options 🟡 Password change functionality 🟡 Detailed error handling and validation

Excluded features, and why they were left out: ❌Payment processing - Beyond scope and would require integration with external systems ❌Concurrent user access - Would significantly increase complexity without adding educational value ❌Database persistence - Project specification explicitly required file-based storage

Chapter 2: System Architecture & Structural Planning

2.1 Planning the System Structure

We designed the overall system layout using the Boundary-Control-Entity (BCE) architectural pattern, providing a clear separation of concerns and aligning with object-oriented principles. This pattern was particularly appropriate for this system, allowing us to clearly distinguish between user interface (UI) components, business logic, and data representation.

Breaking down the system into logical components:

The system was organized into three main component types following the **BCE pattern:**

1. Boundary Classes: These classes handle user interaction, input validation, and display formatting. They separate UI logic from business logic. Each interface corresponds to a specific user role, presenting only the relevant options.

- BTOManagementSystem: Main system class that initializes the application and provides the entry point
- UserInterface, ApplicantInterface, OfficerInterface, ManagerInterface

2. Control Classes: Each controller encapsulates a specific domain of business logic

- AuthController: Manages authentication and user session
- ProjectController: Handles project creation, visibility, and listing
- ApplicationController: Manages application submission, approval/rejection, and status updates
- EnquiryController: Handles enquiry submission and responses
- BookingController: Manages flat booking operations
- ReportController: Handles report generation
- ReceiptGenerator: Generates booking receipts
- RegistrationController: manages registration of HDBOfficers for projects
  WithdrawalController: Handles process of applicant's withdrawal request
- FileManager: Centralizes all file I/O operations for data persistence between .txt files and the programme itself

3. Interface Classes: Each Interface provides an abstraction about a specific domain of business logic mentioned above.

- IAuthController, IProjectController, IApplicationController, IBookingController, IEnquiryController, IRegistrationController, IWithdrawalController, IReportController, IReceiptGenerator

4. Entity Classes: These classes represent the core domain objects, data and maintain system state

- User: Base class managing authentication, profile, and common user attributes
  - Applicant: Handles application submission, viewing eligibility, and booking
  - HDBOfficer: Manages project registration, booking approval, and receipt generation while also having applicant capabilities

    - HDBManager: Oversees project creation/management, application approval, and report generation
- Project: Maintains project details, flat inventory, and related applications
- ProjectApplication: Tracks application status, flat type selection, and withdrawal status
- ProjectFlats: Manages individual flat units within a project, including booking and availability
- FlatType (enum): Represents available flat types (2-Room, 3-Room)
- FlatBooking: Represents the booking of a specific flat, tracking booking status
- Withdrawal: Manages application withdrawal process
- ApplicationStatus (enum): Represents status values (Pending, Successful, Unsuccessful, Booked)
- MaritalStatus (enum): Represents marital status (Single, Married)
- UserNeeds: Interfaces for Dependency Inversion and Interface Segregation
- ManagerOperations: Interfaces for Dependency Inversion and Interface Segregation
- Receipt: Generates and stores booking receipt information
- Enquiry: Manages enquiry content, submission, and responses
- OfficerRegistration: Tracks HDB Officer registrations for projects
- Report: Represents generated reports
- FilterCriteria: Used for filtering reports and projects

User flows and use case mapping: We modeled different user journeys through the system, mapping specific use cases to the appropriate boundary and control classes. For example, the application submission flow was mapped from ApplicantInterface through ApplicationController to the Project and Application entities.

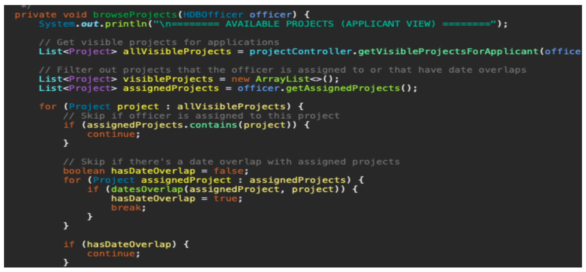
Early visual models: We created flowcharts to visualize key processes such as: User authentication process, Application submission workflow, Application approval process, Flat booking sequence.

2.2 Reflection on Design Trade-offs

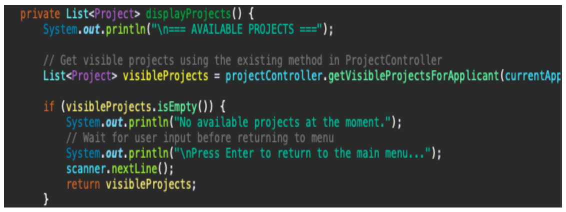
We considered combining the controller and logic layer to simplify the codebase, but ultimately separated them to promote maintainability and allow for better testing.

Trade-offs considered:

Composition vs. Multiple Inheritance in HDBOfficer Implementation:
We decided to compose HDBOfficer with an Applicant role rather than using multiple inheritance. This approach avoided Java's inheritance limitations but introduced some code duplication (eg. visibleProjects code repeated in HDBOfficer and Applicant) since we are not able to override the viewing of available projects method (see image below):



```java
private void browseProjects(HDBOfficer officer) {
    System.out.println("\n======== AVAILABLE PROJECTS (APPLICANT VIEW) ========");

    // Get visible projects for applications
    List<Project> allVisibleProjects = projectController.getVisibleProjectsForApplicant(office

    // Filter out projects that the officer is assigned to or that have date overlaps
    List<Project> visibleProjects = new ArrayList<>();
    List<Project> assignedProjects = officer.getAssignedProjects();

    for (Project project : allVisibleProjects) {
        // Skip if officer is assigned to this project
        if (assignedProjects.contains(project)) {
            continue;
        }

        // Skip if there's a date overlap with assigned projects
        boolean hasDateOverlap = false;
        for (Project assignedProject : assignedProjects) {
            if (datesOverlap(assignedProject, project)) {
                hasDateOverlap = true;
                break;
            }
        }

        if (hasDateOverlap) {
            continue;
        }
```



```java
private List<Project> displayProjects() {
    System.out.println("\n=== AVAILABLE PROJECTS ===");

    // Get visible projects using the existing method in ProjectController
    List<Project> visibleProjects = projectController.getVisibleProjectsForApplicant(currentApp

    if (visibleProjects.isEmpty()) {
        System.out.println("No available projects at the moment.");
        // Wait for user input before returning to menu
        System.out.println("\nPress Enter to return to the main menu...");
        scanner.nextLine();
        return visibleProjects;
    }
```

Team debates:

We also discussed whether to separate ProjectFlats logic from Project class, eventually deciding to create a dedicated class to better encapsulate flat management. We debated whether to use test files or Excel. While Excel and CSV are commonly used for data storage and manipulation, with the nature of the data we were handling and the absence of complex structures, a text file allows for easy reading and writing operations with minimal overhead.

Chapter 3: Object-Oriented Design

3.1 Class Diagram (with Emphasis on Thinking Process)

Class Diagram: Download full resolution class diagram (use zip extractor for better view)

Design Thinking Process:

How the main classes were identified: We identified main classes by analyzing nouns in the problem description, user roles, and entities in use cases. The primary domain entities emerged from our understanding of the BTO application process:

- **User Hierarchy**: The system clearly needed different types of users with varying capabilities, leading to a parent User class with Applicant, HDBOfficer, and HDBManager subclasses.
- **Project Entities**: The central entity in the system is the Project, representing BTO housing projects. Related entities like ProjectApplication, ProjectFlats, and OfficerRegistration naturally emerged from the project-centric operations.
- **Process Entities**: From the application and booking processes, we identified the need for classes like FlatBooking, Receipt, Enquiry, and Withdrawal to represent distinct processes within the system.
- **Enum Types**: To represent fixed sets of values, we created enum types for ApplicationStatus, FlatType, and MaritalStatus.

Responsibilities of each class: (refer to section 2.1)
How relationships were determined: We carefully evaluated the nature of each relationship between classes:

- **Inheritance** was used for the user hierarchy (User → Applicant/HDBOfficer/HDBManager) since these represented "is-a" relationships with shared attributes and behaviors and specialized capabilities.
- **Composition** was used for HDBOfficer 'has-a' applicantRole, as this represented a whole-part relationship where the part cannot exist independently. This design decision was crucial as it allowed Officers to have Applicant capabilities without using multiple inheritance (which Java doesn't support), as well as a more defined segregation between the officer and applicant role.
- **Aggregation** was used for Project containing ProjectApplications, ProjectFlats, OfficerRegistrations, and Enquiries, as these are associated with projects but can conceptually exist independently.
- **Association** was used between controllers and entities they manage, and between different entities like FlatBooking and Receipt.
- **Polymorphism** was used for displayProfile() in UserInterface, which was overridden by each user type to implement specialised behaviour since HDB Officer, Manager and Applicant all have differing fields in their profile.

When deciding between "attribute or class" and "is-a or has-a":
1. We made FlatType, ApplicationStatus, and MaritalStatus as enumerations rather than classes since they had fixed values with no behavior. **2.** We chose composition over multiple inheritance for HDBOfficer's applicant capabilities to avoid Java's inheritance limitations and better model the real-world relationship, where HDB officer could choose to have the role of an applicant, but the HDB officer is NOT an applicant. **3.** We separated ProjectFlats from Project to better encapsulate flat management logic. **4.** We created dedicated entities for processes like Withdrawal and Receipt rather than making them attributes of existing classes to maintain single responsibility

Trade-offs considered: Abstraction vs. performance: We opted for higher abstraction with the BCE pattern despite some performance overhead from the additional layers. **Cohesion vs. coupling**: We prioritized high cohesion within classes (like separating ProjectFlats from Project) even if it meant slightly more coupling between classes. **Inheritance vs. composition**: We used composition for HDBOfficer's applicant capabilities to avoid inheritance limitations and better model reality, despite increased code complexity

3.2 Sequence Diagrams (with Emphasis on Thinking Process)

Sequence Diagrams: Officer Sequence Diagrams

Design Thinking Process: Why These Diagrams?
We chose to develop sequence diagrams for the most critical and complex use cases in our system. These diagrams help illustrate the dynamic interactions between various components across the three layers of our architecture (Boundary, Control, and Entity) and are essential in validating the logical flow of our design decisions.
Most important or complex use cases:
- Officer registration to manage a BTO project
- Officer application for a BTO project

These scenarios were selected because they represent the core functionalities of our system and involve interactions across multiple architectural layers, ensuring inter-layer consistency. The officer registration and application flows highlight role transitions, particularly how an HDB

Officer can also act as an applicant, without compromising on separation of concerns. These sequences enforce critical business constraints such as conflict checks, slot availability, and multi-level approval logic. Furthermore, the inclusion of conditional paths and status management ensures the system behaves realistically under different edge cases. By modeling these flows, we were able to reason through the control logic, ensure responsibilities were appropriately distributed across modules, and validate our use of role composition over inheritance.

3.3 Application of OOD Principles (SOLID)

Single Responsibility Principle: Our controller classes manage a specific domain of functionality. This keeps classes cohesive and maintainable.
Example: The ReceiptGenerator class is solely responsible for generating receipts, making it easy to modify receipt generation without affecting other parts of the system, and File Manager class is solely used for saving and loading of data.

Open/Closed Principle: Using the interfaces created for each controller, we can implement newer versions of each controller based on new and different requirements in the future without changing anything in the main code, but just by creating a new class which implements the respective interface.
Example: If there needs to be an update on how withdrawals are managed, we can make a new withdrawalController2.java which implements the IWithdrawalController interface such that we would only need to change the constructor withdrawalController() to withdrawalController2() only once in the initialisation stage.

Liskov Substitution Principle: Our design ensures that derived classes can be substituted for their base class without altering system behavior.
Example: The system treats all User objects uniformly for authentication purposes, even though specific user types may have additional capabilities. The system also allows for any subclasses to perform actions in the super class such as changing of password and displaying of profile details.

Interface Segregation Principle: We created separate interfaces based on the needs of different User entity classes.

Example: The base User class implements UserNeeds (entity) and HDBManager Class implements ManagerOperations (entity) due to the different requirements of the entity classes. We did not use a one-size-fits-all implementation because some implementations might be redundant for different subclasses.

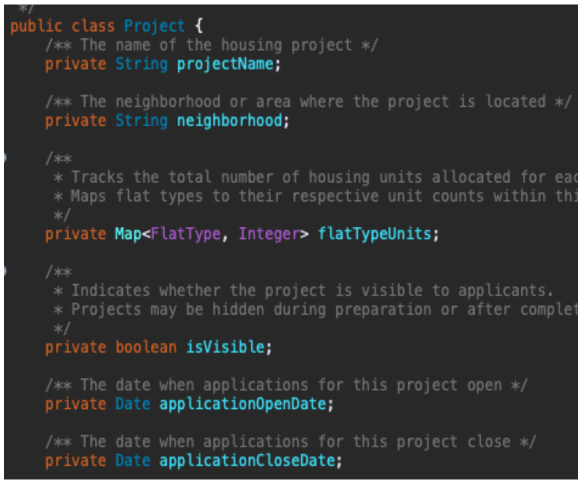
Dependency Inversion Principle: All of the high-level classes depend on the abstract interface classes instead of the concrete low-level classes in our system. This is to prevent tight coupling between the high and low-level concrete classes, and to improve modularity and extension of our system.
Example: All Boundary Classes such as ApplicantInterface, UserInterface, etc. (High Level Classes) depend on the abstract Interface classes like IAuthController, IProjectController, etc., instead of concrete (Low Level Classes) like AuthController, ProjectController, etc.

Chapter 4: Implementation (Java)

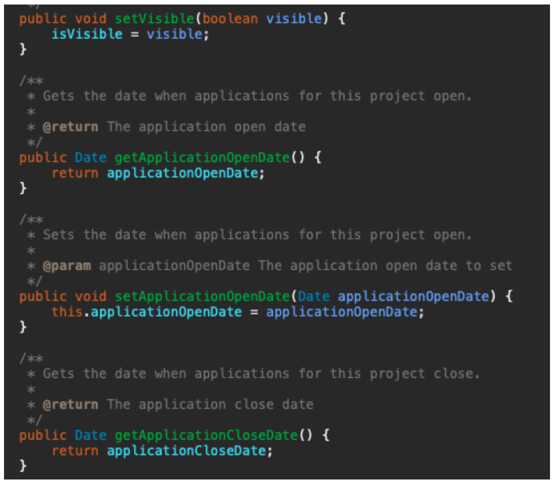
4.1 Tools Used: Java 17, IDE: IntelliJ / Eclipse, Version control: GitHub

4.2 Sample Code Snippets:

Encapsulation: Project class encapsulates private attributes and can only be accessed by getters

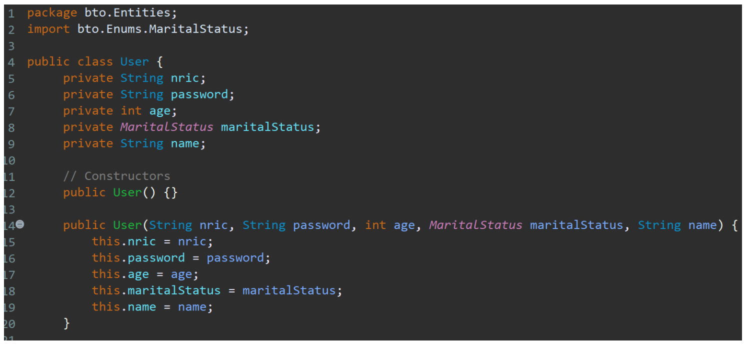


```java
 */
public class Project {
    /** The name of the housing project */
    private String projectName;

    /** The neighborhood or area where the project is located */
    private String neighborhood;

    /**
     * Tracks the total number of housing units allocated for eac
     * Maps flat types to their respective unit counts within thi
     */
    private Map<FlatType, Integer> flatTypeUnits;

    /**
     * Indicates whether the project is visible to applicants.
     * Projects may be hidden during preparation or after complet
     */
    private boolean isVisible;

    /** The date when applications for this project open */
    private Date applicationOpenDate;

    /** The date when applications for this project close */
    private Date applicationCloseDate;
```

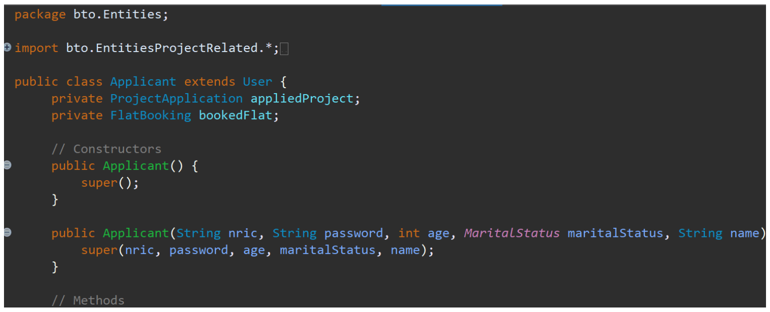


```java
public void setVisible(boolean visible) {
    isVisible = visible;
}

/**
 * Gets the date when applications for this project open.
 *
 * @return The application open date
 */
public Date getApplicationOpenDate() {
    return applicationOpenDate;
}

/**
 * Sets the date when applications for this project open.
 *
 * @param applicationOpenDate The application open date to set
 */
public void setApplicationOpenDate(Date applicationOpenDate) {
    this.applicationOpenDate = applicationOpenDate;
}

/**
 * Gets the date when applications for this project close.
 *
 * @return The application close date
 */
public Date getApplicationCloseDate() {
    return applicationCloseDate;
}
```

Inheritance: Each specialized user type inherits common attributes and behaviors from the User class (eg. Applicant extends User for attributes like nric, age, password, etc.)

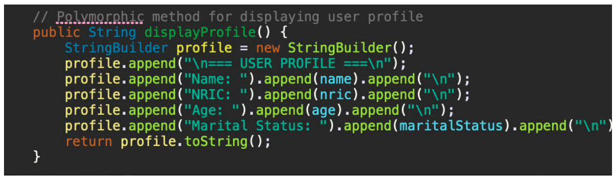


```java
package bto.Entities;
import bto.Enums.MaritalStatus;

public class User {
    private String nric;
    private String password;
    private int age;
    private MaritalStatus maritalStatus;
    private String name;

    // Constructors
    public User() {}

    public User(String nric, String password, int age, MaritalStatus maritalStatus, String name) {
        this.nric = nric;
        this.password = password;
        this.age = age;
        this.maritalStatus = maritalStatus;
        this.name = name;
    }
```

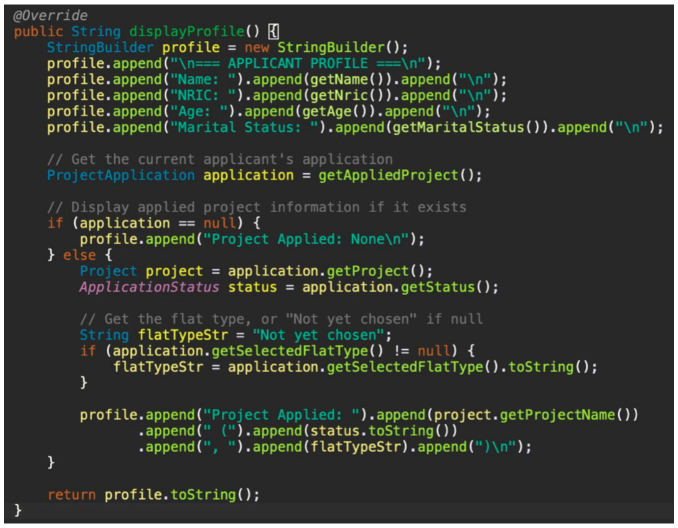


```java
package bto.Entities;

import bto.EntitiesProjectRelated.*;

public class Applicant extends User {
    private ProjectApplication appliedProject;
    private FlatBooking bookedFlat;

    // Constructors
    public Applicant() {
        super();
    }

    public Applicant(String nric, String password, int age, MaritalStatus maritalStatus, String name)
        super(nric, password, age, maritalStatus, name);
    }

    // Methods
```

Polymorphism: Overriding of implemented methods in the superclass by all of its subclasses (eg. displayProfile() of User.java vs Applicant.java)

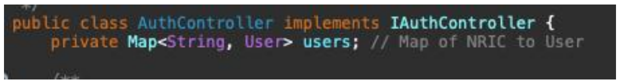


```java
    // Polymorphic method for displaying user profile
    public String displayProfile() {
        StringBuilder profile = new StringBuilder();
        profile.append("\n=== USER PROFILE ===\n");
        profile.append("Name: ").append(name).append("\n");
        profile.append("NRIC: ").append(nric).append("\n");
        profile.append("Age: ").append(age).append("\n");
        profile.append("Marital Status: ").append(maritalStatus).append("\n")
        return profile.toString();
    }
```

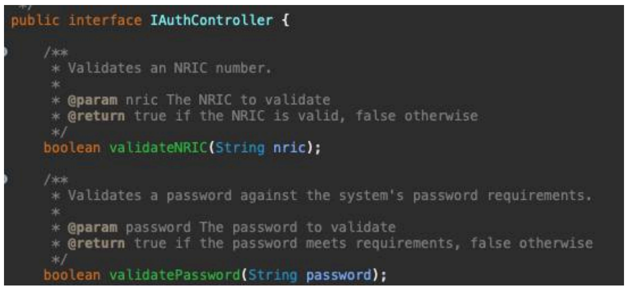


```java
@Override
public String displayProfile() {
    StringBuilder profile = new StringBuilder();
    profile.append("\n=== APPLICANT PROFILE ===\n");
    profile.append("Name: ").append(getName()).append("\n");
    profile.append("NRIC: ").append(getNric()).append("\n");
    profile.append("Age: ").append(getAge()).append("\n");
    profile.append("Marital Status: ").append(getMaritalStatus()).append("\n");

    // Get the current applicant's application
    ProjectApplication application = getAppliedProject();

    // Display applied project information if it exists
    if (application == null) {
        profile.append("Project Applied: None\n");
    } else {
        Project project = application.getProject();
        ApplicationStatus status = application.getStatus();

        // Get the flat type, or "Not yet chosen" if null
        String flatTypeStr = "Not yet chosen";
        if (application.getSelectedFlatType() != null) {
            flatTypeStr = application.getSelectedFlatType().toString();
        }

        profile.append("Project Applied: ").append(project.getProjectName())
                .append(" (").append(status.toString())
                .append(", ").append(flatTypeStr).append(")\n");
    }

    return profile.toString();
}
```

Interface use: All controllers depend on abstract interface classes (eg. AuthController implements IAuthController)

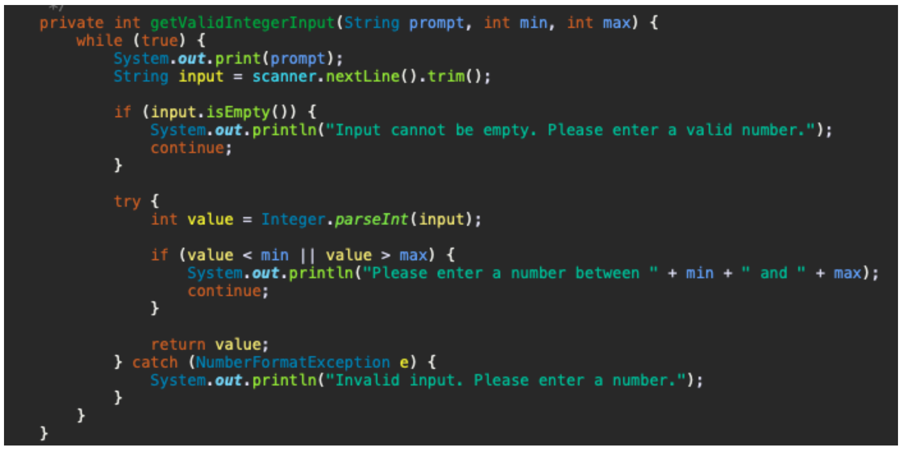


```
public class AuthController implements IAuthController {
    private Map<String, User> users; // Map of NRIC to User
```



```
public interface IAuthController {

    /**
     * Validates an NRIC number.
     *
     * @param nric The NRIC to validate
     * @return true if the NRIC is valid, false otherwise
     */
    boolean validateNRIC(String nric);

    /**
     * Validates a password against the system's password requirements.
     *
     * @param password The password to validate
     * @return true if the password meets requirements, false otherwise
     */
    boolean validatePassword(String password);
```

Error handling: Made a helper function for every single time we needed to parse an input into an integer for selection of choices.



```
private int getValidIntegerInput(String prompt, int min, int max) {
    while (true) {
        System.out.print(prompt);
        String input = scanner.nextLine().trim();

        if (input.isEmpty()) {
            System.out.println("Input cannot be empty. Please enter a valid number.");
            continue;
        }

        try {
            int value = Integer.parseInt(input);

            if (value < min || value > max) {
                System.out.println("Please enter a number between " + min + " and " + max);
                continue;
            }

            return value;
        } catch (NumberFormatException e) {
            System.out.println("Invalid input. Please enter a number.");
        }
    }
}
```

Chapter 5: Testing

Chapter 5.1: Testing Strategy

To ensure the quality of our system, we adopted a hybrid testing approach consisting of black-box testing, which evaluates the system without knowledge of internal implementation, and white-box testing, which examines internal code logic.

Chapter 5.2: Test Case Table

Black Box and White Box Testing: SC2002 BTO Black Box Testing.xlsx

Chapter 6: Documentation

6.1 Javadoc:

Project HTML Javadoc Documentation: https://ccheriii.github.io/SC2002_Group5_FDAE_BTO/

6.2 Developer Guide:

To set up the environment and build the project:

1. Ensure Java 17 JDK is installed
2. Clone the repository from GitHub (git clone https://github.com/gertherald/SC2002-FDAE-BTO-Submission/tree/main)
3. Open the project in IntelliJ/Eclipse
4. Build the project using Maven/Gradle
5. Change directory to src/bto/boundaries
5. Run the BTOManagementSystem class to start the application
6. Default credentials for testing:
    - Applicant: NRIC = "T2345678D", Password = "password"
    - Officer: NRIC = "T2109876H", Password = "password"
    - Manager: NRIC = "S5678901G", Password = “password”

Chapter 7: Reflection & Challenges

What went well:

The Boundary-Control-Entity pattern provided a clear separation of concerns, making the system more maintainable. Object-oriented principles were effectively applied to create a flexible, extensible system. The role-based design successfully enforced business rules about eligibility and access control

What could be improved:

Enhanced design planning with additional UML diagram types (Activity and Use Case diagrams) would provide better visualization of complex workflows. A more comprehensive approach to testing could have been implemented earlier in the development process.

Lessons learned about OODP:

Good design emerges from balancing theoretical ideals with practical constraints. The BCE pattern demonstrated its value in creating a maintainable architecture that localized changes. Design patterns should be applied deliberately based on specific needs, not rigidly.

Individual contributions: FDAE SC2002 BTO WBS.xlsx

Chapter 8: Appendix

GitHub link: https://github.com/gertherald/SC2002-FDAE-BTO-Submission/tree/main