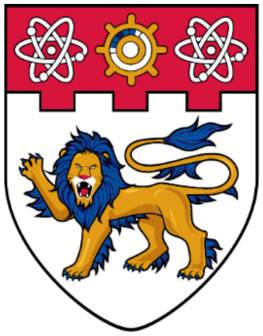


NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

SC2002 OBJECT ORIENTED DESIGN AND PROGRAMMING

BUILD-TO-ORDER [BTO] MANAGEMENT SYSTEM

REPORT

AY 24/25 SEM 2 | FDAD GROUP 5

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	Signature/Date
Bazil Boh Zhuang Kai	DSAI	FDAD	Bazil 24-04-25
Saw Yong Xuen	DSAI	FDAD	SYX 24-04-25
Tan Qi En	CS	FDAD	QiEn 24-04-25
Zhou Runhe	CE	FDAD	Runhe 24-04-25

1.0 Design Overview

1.1 Basic Features

User Specific Role Based Access

Once logged in, users will be granted access to the distinct features of the system according to the role they were assigned with, these namely being:

1. Applicants
    - Mainly only allowed to view the list of available housing projects that they are eligible for, apply or withdraw from said projects, and lodge related enquiries
2. HDB Officers
    - Have the same access as applicants as long as eligible for making applications.
    - HDB officers are given administrative privileges to handle flat booking, register to handle for projects and handle applicant enquiries
3. HDB Managers
    - Able to create,edit and delete BTO project listings
    - Process applications of project handled
    - Approve officer's registration to own project

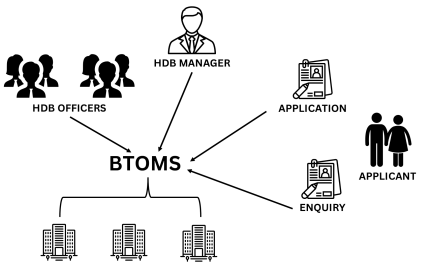
Flow of Application and Management

Applicants have an overview detailed description of housing projects and its eligibility criteria for application. Once applicants have applied for the listings, HDB managers are then able to review the applications through this system and decide whether to approve or deny the request.If it is approved, HDB officers can proceed with flat booking. Along with the application, this system will also handle the withdrawals which will be handled by HDB Managers.

Enquiry Feature

Allows applicants to submit queries regarding housing projects to the administrative team. The latter, which are both the HDB officers and managers, can then view and respond.

1.2 Design Pattern



To enforce the unique responsibilities of each user group, we adopted a protection proxy design pattern along with role based access control where BTOMS acts as the centralised terminal for users to access features that are relevant to them based on their assigned roles.

1.3 System Architecture

To ensure the modularity, extensibility and reusability of our code, we applied the system architecture of Model View Controller (MVC).

1. Model: Encapsulates core data such as user information, project details and applications records while defining attributes, relationships and domain-specific validation rules
2. View: Displaying intuitive interfaces for HDB Officer, HDB Manager and applicants
3. Controller: Handles functional logic such as processing applications, managing projects and user operations

By doing so, we are able to ensure a looser coupling between modules while maintaining high cohesion.

2.0 Design Considerations

2.1 OO Concepts Applied

2.1.1 Abstraction

Abstraction here lets the rest of the system work at a high level without having to know how that actually happens

```java
public boolean createApplication(Applicant applicant, BTOProject project, FlatType flatType) {
    BTOApplication currentApplication = applicant.getCurrentApplication();

    // If applicant has a current application...
    if (currentApplication != null) {
        // Allow creating a new application if the current one is UNSUCCESSFUL or WITHDRAWN
        if (currentApplication.getStatus() == ApplicationStatus.UNSUCCESSFUL ||
            currentApplication.getStatus() == ApplicationStatus.WITHDRAWN) {
            // Remove the old application before creating a new one
            applications.remove(currentApplication);
            currentApplication.getProject().removeApplication(currentApplication);
            // Continue with creating a new application
        } else {
            // For other statuses (PENDING, SUCCESSFUL, BOOKED), don't allow new application
            return false;
        }
    }

    BTOApplication application = new BTOApplication(applicant, project, flatType);
    applications.add(application);
    applicant.setCurrentApplication(application);
    project.addApplication(application);
    saveApplications();
    return true;
}
```

Abstraction is implemented through several ways:

1. Common Base Classes: We group shared fields and methods into abstract parents like User (NRIC, name, password handling) and ApplicantMenu (scanner setup, common menu options) so that each specific role (Manager, Officer, Applicant) only needs to fill in its own details.

2. Managers as Facade: We hide the complex processing logic behind high-level calls. The call in the above picture checks the old application's status, removes it if needed, creates a new BTOApplication, ties it to both the Applicant and the BTOProject, and saves everything to disk, but none of the menu code knows about this.

3. Hiding Internal Logic: UI classes (menus) never see how data is stored, parsed, or validated—they just call methods like loadProjects() or updateApplicationStatus().

4. Data Abstraction: Complex structures (a map of flat sizes to counts inside BTOProject) are only exposed through simple methods like getRemainingUnits() or updateRemainingUnits(), so callers work with "units available" rather than juggling maps themselves.

This approach allows the system to work with high-level concepts without needing to know implementation details.

2.1.2 Encapsulation

Since the system involved a huge amount of confidential user data, encapsulation is applied across the whole BTO Management System to ensure data security.

```java
public abstract class User {
    private String nric;
    private String password;
    private int age;
    private MaritalStatus maritalStatus;
    private UserType userType;
    private String name;

    public User(String nric, String password, int age, MaritalStatus maritalStatus, UserType userType, String name) {
        this.nric = nric;
        this.password = password.isEmpty() ? "" : PasswordHasher.hashPassword(password);
        this.age = age;
        this.maritalStatus = maritalStatus;
        this.userType = userType;
        this.name = name;
    }

    public String getNric() {
        return nric;
    }

    public void setPassword(String password) {
        this.password = password;
    }
```

Encapsulation is implemented through several ways:

1. Private attributes: Sensitive data such as nric, password, age, marital status and name are declared as private variables within the user classes.
2. Controlled access via getters and setters: In order to access or update these private data, public setter and getter methods need to be called. To enforce the rule that NRIC of users cannot be changed, no setter is currently placed.
3. Information hiding: Core functional details for internal operations are hidden from the interface. For instance, the password hashing function which the users are not able to see how the hashing function works when changing password.
4. Role-based specific access: Users under different user types will be having different access to the system data. For instance, all HDB Officers and Applicants can only view but not edit the details of a project while HDB Manager can create, edit and delete the project.
5. Constructor validation: The constructor encapsulates initialization logic, ensuring all User objects are created in a valid state.

By isolating the data and limiting access, the system minimizes the risk of data leaks or accidental tampering.

2.1.3 Inheritance

To promote code reuse and logical hierarchy within the system, inheritance is applied throughout the BTO Management System to reduce redundancy and improve maintainability.

```java
public class Applicant extends User {
    private BTOApplication currentApplication;
```

```java
public class HDBOfficer extends Applicant {
    private BTOProject assignedProject;
```

```java
public class HDBOfficerMenu extends ApplicantMenu {
    private HDBOfficer officer;
```

Inheritance is implemented through several ways:

1. Structured Type Hierarchies: The hierarchy of inheritance is not just applied to user roles but also to different menus and functionalities.
2. Efficient Code Reuse: Shared attributes and behaviors are defined once in a base class and inherited by all user types, reducing duplication and streamlining development.
3. Consistent Interfaces with Customized Logic: Subclasses override base methods to provide role-specific functionality while maintaining a uniform interface across the system.
4. Uniform Handling with Flexibility: The system processes different user types in a consistent manner—such as during login—while still supporting unique behaviors through overridden methods such as showing different menus.
5. Role-Specific Enhancements: Individual user roles extend base functionality with specialized features. For instance, HDB Officer inherits from the Applicant to to get the base functionality as an applicant.

This strategy enables the reuse of shared functionality and maintain simplicity of code

2.1.4 Polymorphism

The use of polymorphism in the BTO Management System is designed to allow different user roles and application processes to be handled efficiently while maintaining clear separation of concerns.

```java
private void showMainMenu() {
    User currentUser = userManager.getCurrentUser();

    if (currentUser instanceof HDBManager) {
        new HDBManagerMenu((HDBManager) currentUser).show();
    } else if (currentUser instanceof HDBOfficer) {
        new HDBOfficerMenu((HDBOfficer) currentUser).show();
    } else if (currentUser instanceof Applicant) {
        new ApplicantMenu((Applicant) currentUser).show();
    }
}
```

```java
// In ProjectManager.java
// Get project by name (String parameter)
public BTOProject getProject(String projectName) {
    return projects.stream()
            .filter(p -> p.getProjectName().equals(projectName))
            .findFirst()
            .orElse(null);
}

// Get project by ID (int parameter)
public BTOProject getProject(int projectId) {
    if (projectId >= 0 && projectId < projects.size()) {
        return projects.get(projectId);
    }
    return null;
}

// Get projects for neighborhood (String parameter with different method name)
public List<BTOProject> getProjectsByNeighborhood(String neighborhood) {
    return projects.stream()
            .filter(p -> p.getNeighborhood().equals(neighborhood))
            .collect(Collectors.toList());
}
```

Polymorphism implemented in several ways:

1. Dynamic Method Dispatch: At runtime, the system determines the actual object type and displays the appropriate menu.
2. Method Overloading: Multiple versions of methods are called according to different parameters. This is important when it comes to sorting functions to get projects with different parameters.
3. Method Overriding: Base methods are overridden according to different user roles. For instance, methods to get visible projects will be overridden under each different user class.

By doing so, common methods can be reused across different user roles, minimizing code duplication and promoting better software design practices.

2.2 OOD Principles (SOLID)

2.2.1 Single Responsibility Principle [SRP]

Each class in our system has one clear purpose, making the codebase easier to maintain, test, and extend. This is especially clear in our control classes that cater to different core functionalities.

```
public class ApplicationManager {
    private static ApplicationManager instance;
    private List<BTOApplication> applications;

    // Application-specific methods only
    public boolean createApplication(Applicant applicant, BTOProject project, FlatType flatType) {...}
    public boolean updateApplicationStatus(BTOApplication application, ApplicationStatus newStatus) {...}
    public boolean requestWithdrawal(BTOApplication application) {...}
    public boolean approveWithdrawal(BTOApplication application) {...}
    // Other application-related methods
}
```

ApplicationManager is the control class for application management. It only handles application-specific operations.

SRP on a wider scale:

1. Controller Layer : Each control class focus on handling single core processing logic (ApplicationManager, ProjectManager, UserManager)
2. Model Layer: Each entity classes focus solely on representing data and state (BTOProject, Application, Enquiry)

3. View Layer: Each menu class focuses on displaying relevant info to the user in different usage scenarios.

2.2.2 Open-Closed Principle [OCP]

Our system allows implementation to add new features down the line without altering the existing code.

```
public boolean updateApplicationStatus(BTOApplication application, ApplicationStatus newStatus) {
    application.setStatus(newStatus);
    saveApplications();
    return true;
}

public boolean requestWithdrawal(BTOApplication application) {
    application.requestWithdrawal();
    saveApplications();
    return true;
}
```

```
public enum ApplicationStatus {
    PENDING,
    SUCCESSFUL,
    UNSUCCESSFUL,
    BOOKED,
    WITHDRAWN
}
```

For the application status, we can simply extend by adding the enum instead of having to modify the existing code instead of having to modify code in Application Manager. For instance, we can just add in status like Project Deleted.

OCP implementation on a bigger scale:

1. Enum-based configuration: Entity attributes like UserType, maritalStatus for user, flatType for project and ApplicationStatus for application are enum-based, thus can be easily extended and configuring the enum classes.
2. Role-based Menu: New menu can be added instead of modifying the existing ones when new role is added in the future such as System Administrator
3. User class hierarchy: When a new type is added, it can inherit directly from the existing user class to ensure code reusability

We applied OCP to cater for future extensibility and scalability while keeping existing code intact.

2.2.3 Liskov Substitution Principle [LSP]

The Liskov Substitution Principle is upheld in our system by ensuring that subclasses like HDBOfficer and Applicant can be used interchangeably wherever their parent class User is expected, without affecting the correctness of the program.

viewAvailableProjects() from ApplicantMenu

```
protected void viewAvailableProjects() {
    List<BTOProject> projects = projectManager.getVisibleProjectsForUser(applicant);
    if (projects.isEmpty()) {
        System.out.println("No available projects found.");
        return;
    }
```

viewAvailableProjects() from HDBOfficerMenu

```
@Override
protected void viewAvailableProjects() {
    // Get ALL projects (including hidden ones)
    List<BTOProject> allProjects = projectManager.getAllProjects();

    if (allProjects.isEmpty()) {
        System.out.println("No projects found in the system.");
        return;
    }
```

This ensures the behaviour is consistent between subclasses and parent classes. The child class will also maintain the guarantee of parent classes.

2.2.4 Interface Segregation Principle [ISP]

This is done by implementing interfaces that are simple with specific functions to avoid unintended dependencies and also allow clearer debugging process:

```
public interface IEnquiryManager { 4 usages  1 implementation  reinhard
    List<Enquiry> getEnquiriesForProject(String projectName); 2 usages  1 imp
    List<Enquiry> getEnquiriesForUser(String nric); 1 usage  1 implementation
    Enquiry getEnquiry(String id); 3 usages  1 implementation  reinhard
    Enquiry createEnquiry(User creator, BTOProject project, String content);
    boolean updateEnquiry(String id, String content, User user); 1 usage  1 i
    boolean deleteEnquiry(String id, User user); 1 usage  1 implementation
```

```
public interface IProjectManager { 5 usages  1 implementatio
    List<BTOProject> getAllProjects(); 3 usages  1 implemen
    List<BTOProject> getVisibleProjects(); 3 usages  1 imple
    List<BTOProject> getVisibleProjectsForUser(User user);
    BTOProject getProject(String projectName); 9 usages  1 i
    void addProject(BTOProject project); 1 usage  1 implemer
    boolean deleteProject(String projectName); 1 usage  1 im
    boolean removeProject(BTOProject project); 1 usage  1 im
```

By splitting each responsibility into its own small interface, each interface is focused (a project-handling class only implements IProjectManager), and doesn't have to provide

irrelevant methods. Role based interfaces create separate interfaces for each user role while feature segregation allows each feature area to have its own set of interfaces.

2.2.5 Dependency Inversion Principle [DIP]

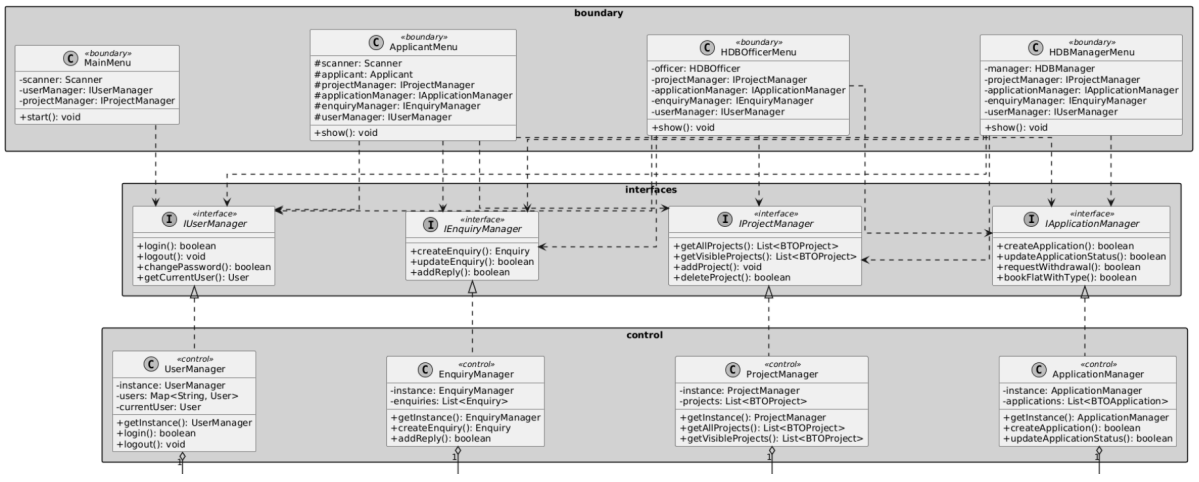
It states that high-level modules should not depend on low-level modules and both should instead depend on interfaces and abstract classes, this is to allow flexibility and extensibility when changes are to be made, without affecting the core function of our system

```
public interface IUserManager {  5 usages   1 implementation
    boolean login(String nric, String password);  3 usages   1 impleme
    void logout();  3 usages   1 implementation
    boolean changePassword(String oldPassword, String newPassword);
    User getCurrentUser();  3 usages   1 implementation
    User getUser(String nric);  4 usages   1 implementation
    void saveUsers();  1 usage   1 implementation
}
```

This is closely linked with ISP shown above as the IUserManager simply calls methods on abstractions which are the different interface types like (IProjectManager). This demonstrates the idea of DIP as the Applicant menu (High level module) and Project Manager (low-level module) for example depends on abstractions (interface)

High-level module: ApplicantMenu (the UI logic)
Abstractions: Interfaces IProjectManager, IApplicationManager, IEnquiryManager
Low-level modules: ProjectManager, ApplicationManager, EnquiryManager

The overall DIP structure is shown below, i.e. all classes depend on abstract interfaces:



2.3 Additional features implemented

- Password requirements (minimum of 6 characters, with 1 alphabet and 1 number)
- Password hash function (password is hidden to protect user privacy)
- After status is “booked”, applicants are able generate receipt with details in a txt file
- Applicants are able to carry out project filtering
- Officers can view unit availabilities (2-Room and 3-Room) before assigning rooms
- Improved formatting for a clearer UI
- Managers can see additional statistics such as distribution of flat type for application details
- Managers will be prompted to re enter their password for deletion of projects
- Manager is able to create a new project that opens in the future using an auto-publish function. This auto-publish can be toggled on and off after project is open and set to visible
- Utility function such as tableprinter and systemlogger are introduced

2.4 Assumptions made

1. Data storage is stored in a text file rather than a actual database system
2. Just NRIC and password is sufficient to login to the system, without requiring secondary verification layers like SingPass authentication
3. System is developed as a Command Line Interface (CLI) application and will not be extended

3.0 UML Diagram

3.1 Class Diagram

To view the full picture of the class diagram, click on Class Diagram

3.2 Sequence Diagram

To view the full sequence diagram and others, click on Sequence Diagram

4.0 Testing

To see our full test cases table, click on Test Cases. The **additional features** we have implemented are at the bottom of the test cases table.

5.0 Reflection on lessons learned and the challenges we faced along the way

Reflection

One lesson we took away from this project was the skill of balancing time management and creativity. Although good time management was important to ensure that we could meet the project deadline, we also had to account for a list of creative features that we wanted to include to enhance the functionality of the system. This taught us the importance of having a strong grasp of the relevant concepts taught in this module as key general concepts such as SOLID and OOP principles had a direct impact on the way we carry out the project. Technical knowledge such as UML class and sequence diagrams allowed us to effectively communicate our ideas and brush out any differences. It also highlighted areas of our comprehension that were still lacking, providing us an opportunity to then refine our understanding and address the gaps. Overall, this project enhanced our theoretical knowledge of Java and OOP.

Challenges and how we conquered it

Given the scale of this project, it reinforced our attention to details as small mistakes such as stray colons or mistyped variable names can often result in major functionality issues down the line. This made us improve on our documentation and communication practices to maintain consistency of work across the team so as to minimise redundant work and allow us to better spot mistakes. Another challenge was incorporating the SOLID design principles as we are so used to coding without any knowledge of coding principles. Therefore, this taught us to be more aware and stringent in our coding.

Appendix

Github Link: https://github.com/AlvinSaw/SC2002---BTOMS