SC2002: Object-Oriented Design & Programming

Build-To-Order (BTO) Management System
2024/2025 SEMESTER 2
GROUP 4

COLLEGE OF COMPUTING AND DATA SCIENCE (CCDS)
NANYANG TECHNOLOGICAL UNIVERSITY

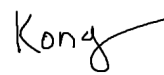| Name	Matriculation Number
Chew Jun Yang	U2323006A
Kong Kai Wang	U2323506H
Jayaraj Kishore Kumar	U2423796B
Jordan Chia Zhi Heng	U2421931C
Patrick Elliot Subagio	U2420459E

Declaration of Original Work for SC2002 Assignment

We hereby declare that the attached group assignment has been researched, undertaken, completed, and submitted as a collective effort by the group members listed below.

We have honored the principles of academic integrity and have upheld Student Code of Academic Conduct in the completion of this work.

We understand that if plagiarism is found in the assignment, then lower marks or no marks will be awarded for the assessed work. In addition, disciplinary actions may be taken.

Name Course Lab Group Signature / Date

Name	Course	Lab Group	Signature / Date
Chew Jun Yang	SC2002	FCSF	*chew* 23 April 2025
Kong Kai Wang	SC2002	FCSF	*Kong* 23 April 2025
Jayaraj Kishore Kumar	SC2002	FCSF	23 April 2025
Jordan Chia Zhi Heng	SC2002	FCSF	23 April 2025
Patrick Elliot Subagio	SC2002	FCSF	23 April 2025

1.   Understanding requirements

We began by reading through the BTO document line-by-line, highlighting all use cases and system requirements. Based on this, we created a list of essential features and identified user roles and system entities.

- Main Problem: Create console-based system to manage the lifecycle of a BTO system, including user managements, project creation and management, applicant applications, officer registrations, flat booking and enquiry handling.

- User Roles:

    - Applicant: Can view projects, apply, manage their application

    - HDBOfficer: Inherits Applicant capability, but also can register to handle projects, manage enquiries and process flat bookings.

    - HDBManager: Have control of projects, visibility, CRUD, manage applications, withdrawal, enquiries and generate reports

- Explicit: User login, role-based controls, project creation/deletion/editing, application to BTO, withdrawal request, officer registration/approval, flat booking, enquiry, password change

- Implicit: Persist data?, NRIC format (We assumed letter + 7 numbers + letter), date logic for overlapping, Using CSV files for data storage

- Ambiguous:

    - Eligibility: Based on the requirements, assuming single >= 35 can apply 2-room, married >= 21 2/3-room, does that mean single < 35 and married <21 cannot apply at all? we assume yes

    - Can officers see projects to register for even if not visible? We assumed yes

2.   Design Considerations

We designed our BTO management system using an object-oriented approach as a Command Line Interface (CLI) application. The Architecture pattern is designed to resemble a Model-View-Controller (MVC) design pattern, adapted for consoles with a Service Layer for data persistence management.

3.   Use of Object-Oriented Concepts

3.1.   Encapsulation

Encapsulation is the practice of bundling data (fields) and the methods that operate on the data into a single unit, while restricting direct access to some of the object's components. Notice how the fields registrationId, officerNric, projectName, status, and registrationDate are all declared as private. This is a key aspect of encapsulation.

- By making the fields private, we ensure that these values are hidden from direct access outside of the class. This protects the internal state of the object and helps avoid unintentional modifications.

- The constructor ensures that the class is initialized correctly. For instance, the registrationId is automatically generated based on the officerNric and projectName. We also validate that none of the required fields (officerNric, projectName, or registrationDate) are null, ensuring the object is always in a valid state when created.

```java
import Enums.OfficerRegistrationStatus;
import java.util.Date;

public class OfficerRegistration {
    private final String registrationId;
    private final String officerNric;
    private final String projectName;
    private OfficerRegistrationStatus status;
    private final Date registrationDate;

    public OfficerRegistration(String officerNric, String projectName, Date registrationDate) {
        if (officerNric == null || projectName == null || registrationDate == null) {
            throw new IllegalArgumentException("OfficerRegistration fields cannot be null");
        }
        this.registrationId = officerNric + "_REG_" + projectName;
        this.officerNric = officerNric;
        this.projectName = projectName;
        this.status = OfficerRegistrationStatus.PENDING;
        this.registrationDate = registrationDate;
```

3.2.   Abstraction

Abstraction allows the user to interact with a simplified interface while keeping the underlying complexity hidden. The BaseView class is an abstract class as it provides a simplified interface with essential methods like displayMenu() and getMenuChoice(), but it leaves the implementation details to subclasses that will extend BaseView.

● It defines a common structure but cannot be instantiated directly.
● displayMenu() is an abstract method: It must be implemented by subclasses, allowing specific functionality to be defined while hiding the details in the BaseView class.

```java
public abstract class BaseView {
    protected final Scanner scanner;
    protected final User currentUser;
    protected final BaseController controller;
    protected final AuthController authController;

    public BaseView(Scanner scanner, User currentUser, BaseController controller, AuthController authController) {
        this.scanner = scanner;
        this.currentUser = currentUser;
        this.controller = controller;
        this.authController = authController;
    }

    public abstract void displayMenu();
```

3.3.   Inheritance

Inheritance is an object-oriented principle where a class (subclass) inherits attributes and methods from another class (superclass), enabling code reuse and logical class hierarchies.
● Here ApplicantView and OfficerView extend BaseView as they inherit common methods like getMenuChoice() and changePassword() without rewriting them.
● Promotes code reuse and structure by sharing functionality that is written once in BaseView and reused in both views, keeping the code clean and consistent.

```java
public class OfficerView extends BaseView {
    private final OfficerController officerController;

    public OfficerView(Scanner scanner, User currentUser, OfficerController controller, AuthController authController) {
        super(scanner, currentUser, controller, authController);
        this.officerController = controller;
    }
```

```
public class ApplicantView extends BaseView {
    private final ApplicantController applicantController;

    public ApplicantView(Scanner scanner, User currentUser, ApplicantController controller, AuthController authController) {
        super(scanner, currentUser, controller, authController);
        this.applicantController = controller;
    }
```

3.4.   Polymorphism

Polymorphism allows different classes to provide specific implementations of the same method, making the system more flexible and dynamic.

- ApplicantView and OfficerView override displayMenu() to provide their own implementations, showing polymorphism.
- Methods in service classes can handle different user types (Officer, Applicant, Admin) using the same interface, enabling flexible behavior.

```
@Override
public void displayMenu() {
    boolean logout = false;
    while (!logout) {
        System.out.println("\n=========== Applicant Menu ===========");
        System.out.println("Welcome, " + currentUser.getName() + "!");
        System.out.println("1. View Available BTO Projects");
        System.out.println("2. Apply for BTO Project");
        System.out.println("3. View My Application Status");
        System.out.println("4. Request Application Withdrawal");
        System.out.println("5. Submit Enquiry");
        System.out.println("6. View My Enquiries");
        System.out.println("7. Edit My Enquiry");
        System.out.println("8. Delete My Enquiry");
        System.out.println("9. Apply/Clear Project Filters");
        System.out.println("10. Change Password");
        System.out.println("0. Logout");
        System.out.println("=====================================");
```

4.   Design Principles

4.1.   Single Responsibility Principle (SRP)

The Single Responsibility Principle states that each class should only have 1 responsibility. BTO Manager is split into multiple classes in multiple packages, each with their own purpose. Each class has different functionalities but aligned goals in their package. This provides clear organization of code and enables easy addition of new features in the future. The packages are elaborated below:

a. **Models**
    This is the core structure of our program. Classes such as *Applicant*, *HDBManager* and *HBDOfficer* and their abstract base class *User* is here.

b. **Views**

This package contains the classes where they manage what each user will see. Different end users are managed by different classes, such as *ApplicantView*, *ManagerView* and *OfficerView*, inherited by their abstract base class *BaseView*.

c. **Services**

Services give logic to models by interacting with each of them to provide a useful result, such as booking flats and applying for projects through *ApplicationService*, and manage the sending and receiving of enquiries between users through *EnquiryService*.

d. **Controllers**

Controllers handle the logic and coordination between the Services and View classes. *ManagerController* and *OfficerController* through their base abstract class *BaseController*

e. **Enums**

Enums provide classes that provide the options of certain attributes such as *MartialStatus* with either **Single** or **Married**.

f. **Parsers**

Contains *LSparse*, a List parser and *Dparse*, a Date parser.

g. **Utils**

Contains other useful functions, such as *DateUtils* and *NricValidator*.

h. **Data**

This is where our data is read and stored in their own respective CSV files. *ApplicantList*.csv, *Enquiries*.csv and *ManagerList* are some examples.

4.2. Open/Closed Principle (OCP)

Our Program follows the Open/Closed Principle as the classes are open to extension, but closed to modification. Our classes support extension through inheritance, as shown in 4.3. One example is our *User* abstract class, where it has been implemented by various other user roles such as *Applicant* and *HDBManager*.

4.3.   Liskov Substitution Principle (LSP)

Objects of a superclass should be replaceable with objects of its subclasses without breaking the application. This is achieved by implementing all the methods that the superclass has in the subclass by either by inheriting or overriding. This can be seen in the same *User* class example, where *Applicant* and *HDBManager* all implement the methods that *User* uses, hence supporting LSP.

4.4.   Interface Segregation Principle (ISP)

Interface Segregation Principle states that classes should not be forced to have methods that they do not use implemented by their interfaces. Therefore, we achieve ISP by splitting up each interface with their own respective use case. This can be seen by the multiple interfaces that we used, such as *IUserService*, *IProjectService* and *IEnquiryService*. Each interface only has methods specifically for their purposes, hence avoiding redundant implementations of methods.

4.5.   Dependency Inversion Principle (DIP)

Dependency Inversion Principle states that high-level modules should not depend on low level ones. We implemented these by having interfaces and segregating the low-level modules such as reading from CSV in *CsvRW* to get user's login details, from higher-level modules such as the login method in *AuthController*. This ensures that our high-level modules can be reusable, and will not be affected by changes in the low-level modules

5. Assumptions Made

- The app is only CLI

- Data is persisted everytime an action is taken

- Simplify NRIC to be just Letter + 7 Numbers + Letter

- No concurrency, designed for single user access

- Data Synchronisation assumes it won't encounter any inconsistencies when loading the CSV, though it does check (Application records make sure applicant exist in csv before loading)

- <35 Single and <21 Married got 0 projects can apply for as an applicant

- Assume only got 2-room and 3-room flats

- Data stored in CSVs

6. Extensibility & Maintainability

- Extensibility

  - New roles are relatively easy to create, simply extend the user and create corresponding views and controllers.

  - New Flat Types requiring modifying the respective enum, however also need to update eligibility logics like (*canApplyForFlatType*) as the age and potentially services.

  - New types of reports can be added to ReportManagerController, easy changes for simple reports.

  - Changing to a database would require modifying a large number of services as right now assume is CSV.

- Maintainability

  - Layered architecture, clear separations of where to find what mentioned in layers above

  - SRP, controllers and services and limited to their own scopes

  - DIP, tried to implement loose coupling with interfaces to reduce the ripple effects of changes

7.   Design Trade-offs

● Static DataService: Convenient for global access to sync or save operations but introduced tighter coupling for this task

● HDBOfficer extends Applicant promotes reuse of functionalities but requires instanceof checks to check if the user role is correct.

● BaseController Dependencies: We injected a lot of services into basecontroller, while convenient for subclasses, potentially violating ISP.

● Error Handling: Our code prioritized functional code over error recover

8.   Design Patterns Used

● Model-View-Controller (MVC), adapted for CLI, separates data (Model), presentation (View) and application logic/flow (Controller)

● Service Layer: An extra layer for data access

● Delegation controllers delegate specific tasks to specialized sub controllers.

● Interfaces used extensively for services to achieve DIP and OCP.

● Abstractions provide common structures and usable code, used for User, BaseView and BaseController.

● Alternatives Considered:

  ○ No Service layer, just combined with controllers, rejected as it would tightly couple the controllers to CSV handling, making the code less maintainable and extensible, rejected.

  ○ Using only 1 controller per role, we scratched that cause that would make each controller very large, rejected for better organization.

9.   UML Class Diagram

- Refer to Class.png for better view as it's quite large, it's included in the base directory of our java application alongside *BTOApp*.java
- To create this, we used https://www.drawio.com
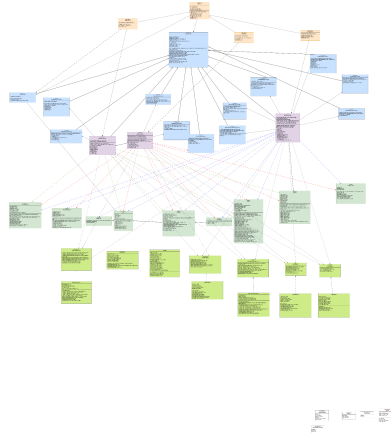
10.   UML Sequence Diagram

- Refer to SequenceApply.png and SequenceRegister.png for better view as it's quite large, it's included in the base directory of our java application alongside *BTOApp*.java
- To create these, we used https://sequencediagram.org/
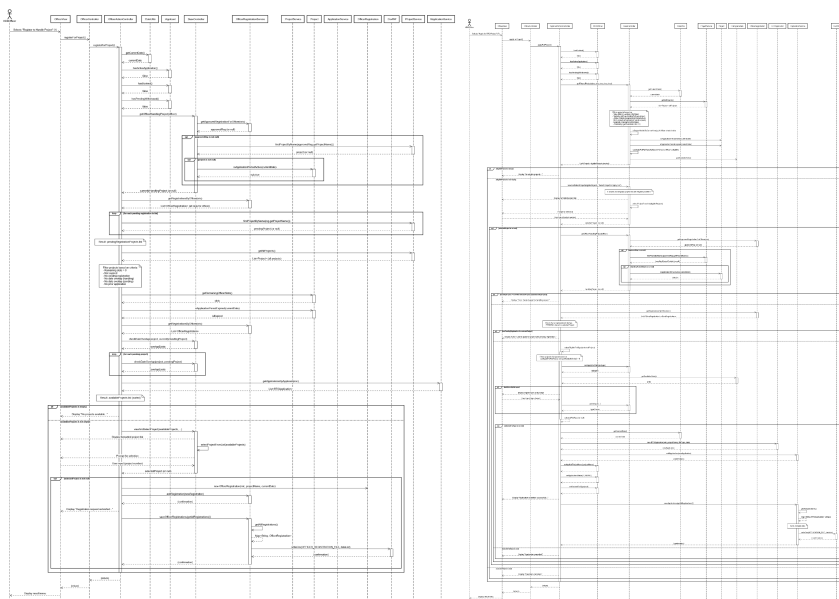
11.   Additional Features Implemented

- Flexible project filtering (*BaseController.getFiltereredProjects*), with multiple boolean checks to display only relevant projects based on eligibility, availability, application period status, user-defined location/flat type filters

- Data synchronization in DataService.synchronizeData, implements the logic to ensure consistency between different data sources after loading (*Users*, *BTOApplication*, *ProjectList*, etc.)

- Withdrawal Status, we added a PENDING_WITHDRAWL status to differentiate between successful and unsuccessful withdrawal

- Automatic ID Generations for enquiries, applications, registrations that are generated upon creation, simplifies the data management

- Manual CSV Parsing (*CsvRW*), we manually parsed each csv since data management libraries weren't allowed

12.   Testing

- Refer to Test Cases.pdf as it's also quite large, but we used the same test cases as the assignment, however we added an additional column called Copy which is lines of inputs for easier testing, just copy paste into the terminal to perform the operations.

13.   Test Cases and Results

- Refer to Tests directory, we ran each of the commands in the Copy column for each and saved the results in a folder called Tests in the project directory

14.   Reflection

- Through the development of this BTO Management System, we gained a lot of experience in applying object-oriented principles and software design best practices. By separating business logic from user interface logic, the modular design helped us manage development. We also recognised how crucial it is to arrange class interactions in advance. The focus on class design and planning class interactions early in the process was crucial. This helped us avoid tight coupling and ensure that the system was scalable.

Moreover, the use of inheritance, encapsulation, and polymorphism made the system more flexible, allowing for code reuse and more efficient maintenance.

- It was really hard to implement SOLID in the code, the designing process was complicated and required accounting for future needs, although we didn't actually implement it that well in the end, we tried to apply it to some cases (*Controller*, *Services*) to the best of our abilities

15. **Further Enhancements**

In future developments, we can use a database to store our users information. This can ensure data consistency and prevent concurrency issues between multiple applications running at the same time. Furthermore, we can introduce an analytics dashboard, where HDB Managers can view visual statistics such as the number of successful applications by flat type or the popularity of projects across neighborhoods. This would help managers make data-driven decisions for future housing launches and resource planning. We also plan to be able to better enforce SOLID principles in the code base.