SC2002 2024/2025 SEMESTER 2 ASSIGNMENT REPORT
Build-To-Order Management System (BTOMS)

GitHub: https://github.com/PuttTim/sc2002-bee-tee-ohh

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
Timmanee Yannaputt U2423542K	SC2002	FCSA Group 6	
Tan Wei Xin U2422174F			
Ma Shiqi U2421382B			
Koh Kai Jie U2421270C			
Ong Jee Shen U2423210K			

1. Design Considerations

1.1 Approach Taken

To develop the Build-To-Order Management System (BTOMS), we began by analysing the project requirements. We identified key classes by examining the nouns in the specification, while the verbs helped us understand each class's roles and responsibilities. This guided the design of appropriate attributes and methods for each class. To organise our ideas, we first created an Entity-Relationship (ER) diagram, which we later translated into a Unified Modelling Language (UML) Class Diagram that served as the structural blueprint for our system.

The four stages of computational thinking were also applied throughout this process:

- **Decomposition:** We broke down the use cases into individual methods.
- **Pattern Recognition:** We identified repeated methods across different use cases.
- **Abstraction**: We simplified overlapping methods by generalising shared behaviours.
- **Algorithm Design:** We developed algorithms to perform tasks efficiently.

We prioritised the most critical features to ensure that the main functions of the system were robust before extending them with additional features.

BTOMS was designed using a modular approach within a monolithic architecture, where all components are integrated into a single, unified application. Our modular design was guided by object-oriented programming (OOP) concepts and SOLID design principles to ensure reusability, extensibility, and maintainability. To support this, we focused on achieving high cohesion and low coupling. We began by separating the application into distinct packages, each with a clearly defined responsibility. Within each package, classes were designed to handle specific tasks while adhering to OOP principles, ensuring that their attributes and methods worked together to fulfil a single purpose. To minimise coupling, we linked classes only when necessary and carefully managed their relationships in accordance with SOLID principles.

Based on this structure, we adopted the Model-View-Controller (MVC) architecture for BTOMS, along with additional packages such as Services, Repositories, Interfaces, and Enums. This ensured a clear separation of concerns, making the system easier for new developers to understand, test, and maintain. It also simplified the implementation of changes by limiting the number of affected classes and reducing the ripple effect.

In addition, we employed other design patterns, such as a simplified repository pattern, to address specific challenges encountered during development.

1.2 Assumptions Made

a. The different stakeholders ('Applicant', 'Officer', and 'Manager') are assumed to be familiar with using BTOMS based on their respective roles. As such, the information displayed to them is concise and role-specific.

b. All data stored in the comma-separated values (CSV) files is assumed to be properly formatted and does not require data cleaning prior to import.

1.3 OOP Concepts

1.3.1 Abstraction

We applied abstraction to simplify the complexity of BTOMS by exposing only essential details to the user while hiding non-essential ones. For example, interfaces were used to abstract the internal logic of the Services and Views, as shown in Figure 1.3.1a.

```
14   /**
15    * Service class for managing property enquiry operations.
16    * Provides functionality for creating, retrieving, editing, and responding to property enquiries.
17    * Enforces business rules regarding enquiry ownership and status transitions.
18    */
19   public class EnquiryService implements IEnquiryService {
20       private static EnquiryService instance;
21
22       private EnquiryService() {}
23
24       public static EnquiryService getInstance() {
25           if (instance == null) {
26               instance = new EnquiryService();
27           }
28           return instance;
29       }
30
31       /**
32        * Retrieves all enquiries for a specific project.
33        *
34        * @param project the project to filter enquiries by
35        * @return list of enquiries associated with the given project
36        */
37       @Override
38       public List<Enquiry> getProjectEnquiries(Project project) {
39           return EnquiryRepository.getEnquiriesByProject(project.getProjectID());
40       }
```

Figure 1.3.1a: SomethingService implements ISomething

3

1.3.2 Encapsulation

We used encapsulation to protect the internal state of objects. All attributes were declared as private, and we provided public methods, such as getters and setters, as the sole means of access. This prevents unauthorised access or modification of data by external classes.

1.3.3 Inheritance

To promote reusability, we used inheritance to establish a parent-child relationship between classes. A generic 'User' class was created to serve as the base, providing common attributes such as 'userNRIC', 'name', 'password', 'age', 'maritalStatus', and 'role', along with methods like 'getUserNRIC()' and 'setPassword()'. Subclasses such as 'Applicant', 'Officer', and 'Manager' inherit these properties and methods while extending them with role-specific functionality. For example, the 'Officer' class inherits from the 'Applicant' class and adds methods relevant to its responsibilities. This hierarchical structure is illustrated in the UML Class Diagram in Section 3.

1.3.4 Polymorphism

We implemented polymorphism, allowing objects of different subclasses to be treated as objects of a common superclass ('User') while still exhibiting their specific behaviours. This principle is applied, for example, in how different user roles are handled after authentication. Although the 'dispatchToController' method shown in Figure 1.3.4a uses a switch statement based on 'user.getRole()' to determine the specific actions and views (such as 'showApplicantMainMenu' or 'showOfficerMainMenu'), the underlying concept is to provide role-specific functionality derived from the common 'User' type.

Figure 1.3.4a: dispatchToController()

1.4 Principles Used

1.4.1 Single Responsibility Principle (SRP)

The SRP was demonstrated through the clear separation of responsibilities using well-structured packages (e.g., Controllers, Services, Repositories, Views, Models). Each class within these packages focuses on specific, well-defined responsibilities, supporting maintainability and ease of modification.

a. Models

Each user role, such as 'Manager', 'Officer', 'Applicant', has its own class, focusing on actions relevant to its own role. Other classes handle operations such as application, registration, and enquiries.

b. Controllers

Controller classes manage user interactions and handle the application flow. They receive user input and invoke the appropriate Service and View classes. For example,

'OfficerController' handles all operations related to 'Officer' actions, where, upon receiving a user request to view project details, 'OfficerController' might invoke 'ProjectService' to retrieve the relevant data.

c. Services

Service classes implement business logic. For instance, 'ApplicationService' contains logic for managing housing application operations.

d. Views

These classes are responsible for displaying information to the user, allowing users to view specific information about particular objects. For example, 'ApplicantApplicationView' enables applicants to view projects and application statuses.

e. Repositories

Repository classes encapsulate data access logic, separating file or database operations. For example, 'ProjectRepository' manages project data.

f. Interfaces

Interface classes, such as 'IProjectService', define important operations in the system without specifying their exact implementation.

1.4.2 Open-Closed Principle (OCP)

The OCP principle was implemented for easy extensibility without modifying existing code. This is done through interfaces to promote loose coupling, allowing seamless integration of new features. For example, adding a new user role is as simple as implementing the IUser service without existing functionality.

Services and views implement specific interfaces, ensuring flexibility and easy replacement. For example, the 'ProjectService' implements the 'IProjectService' interface, allowing us to add new handling project logic without impacting the rest of the system.

1.4.3 Liskov Substitution Principle (LSP)

The LSP was implemented by ensuring that subclasses do not have preconditions stronger than those of their superclass methods, and do not have postconditions weaker than those of the superclass methods. In this way, the subclasses can replace their parent classes without altering the expected behaviour.

Classes such as 'Applicant' and 'Manager' inherit common functionalities from the base class 'User'. Each subclass then includes the additional functionalities it requires.

1.4.4 Dependency Inversion Principle (DIP)

Our high-level classes, such as 'AuthService', depend on multiple repositories, all of which are injected through constructors. This adheres to DIP and improves maintainability and loose coupling as the high-level classes interact with low-level classes through abstractions.

1.5 Other Considerations

1.5.1 Foreign Key Association

Foreign keys were used to ensure that changes made to a table would propagate to the related tables to ensure data consistency. 'Applicants', 'Officers' and 'Managers' are stored as strings rather than objects in the private attributes of the ' Project' class. This way, the actual data of them is stored in the CSV files, and any changes made to them only require the CSV file to be updated, not every 'Project' that references them. This prevents any data inconsistency that would occur if they were stored as objects in 'Project'. Additionally, this increases the performance of the app by preventing stale data and fetching the data of 'Applicants', 'Officers' and 'Managers' through getter methods only when necessary.

1.5.2 Simplified Repository Pattern

Our repository interface 'IBaseRepository' contains methods that are in the simplest form, such as 'List<T> getAll();', 'void saveAll(List<T> items);' and 'void load();'. This implementation decouples business logic from data access and provides a centralised place to handle all data access logic. Additionally, it also allows us to simplify the testing of our code as we can create an isolated testing environment, such as a mock repository, for us to test individual components.

2. Additional Implementations

2.1 Password Hashing

All user passwords have been hashed using the SHA-256 algorithm (which has been chosen because of its strong cryptographic properties) before they are stored in the CSV file. This ensures that even if the database has been compromised, the user's passwords cannot be easily deduced.

2.3 Filtering

Filtering allows users to specify data based on specific criteria, like marital status or flat type. User experience can be improved through this, as users are able to find relevant information quickly. An example is 'generateApplicantReport()' in 'ManagerService', where applications are filtered by project ID and status.

2.4 Report Export to CSV

Reports can be exported to CSV files for easy sharing. This allows data to be stored externally and supports transfers to other systems for further processing if needed. For example, in 'ManagerService', 'exportReportToCsv' takes generated report data and exports it as a CSV file with a dynamically generated name.

2.5 Unit Testing

Our application uses JUnit to have a few unit tests set up to check authentication and hashing, automating testing for test cases such as using correct and incorrect login credentials, ensuring the password change functionality works, and various hashing-related tests.
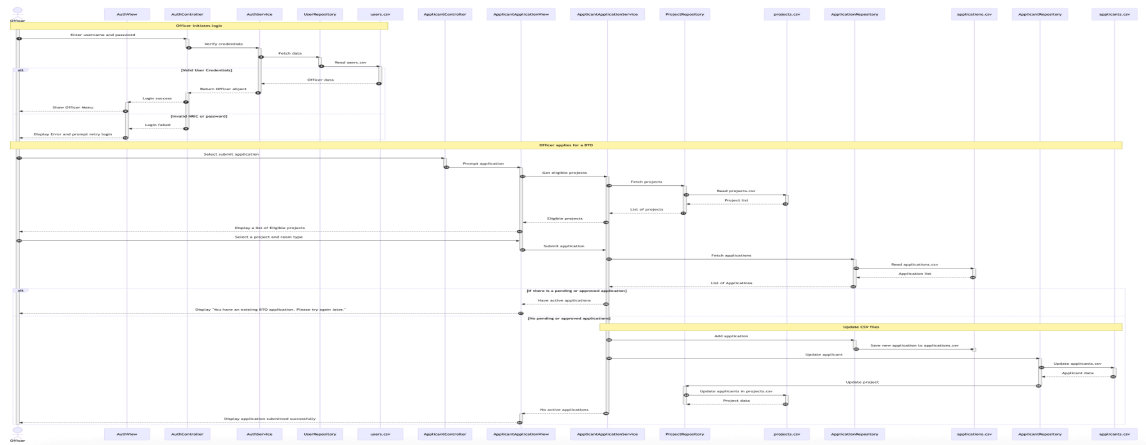
3. UML Class Diagram
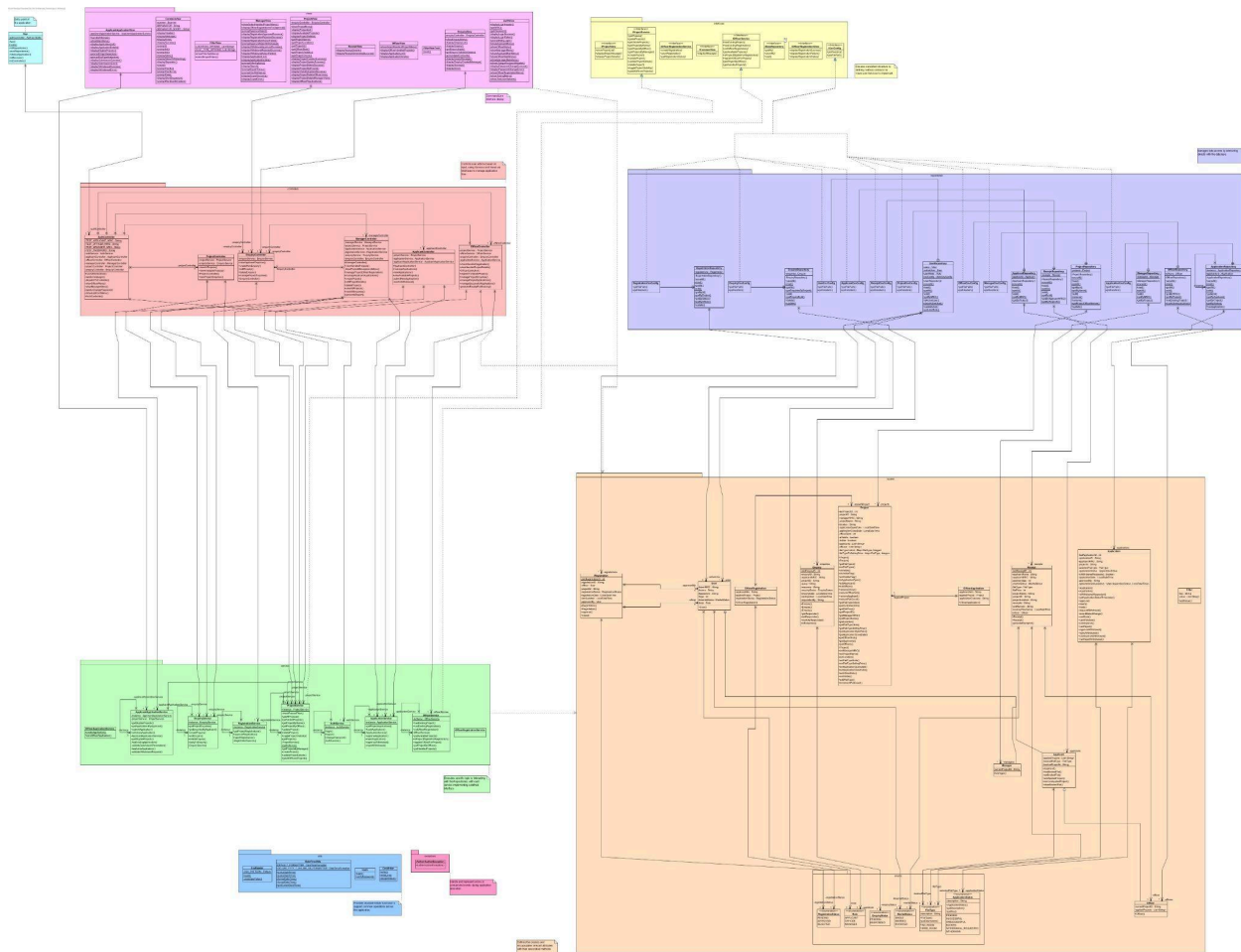
A higher resolution UML Class Diagram is in a separate file named "ClassDiagram.jpg"

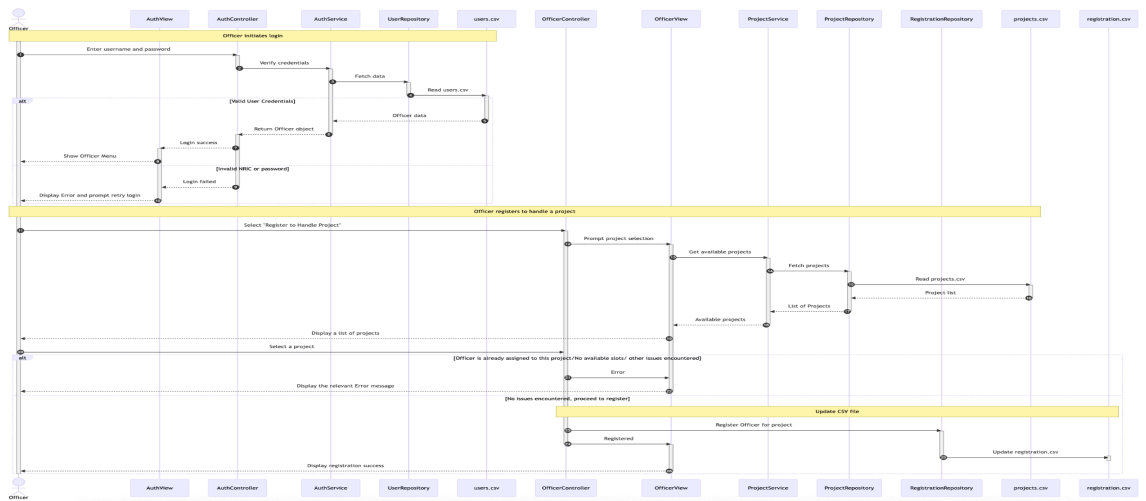# 4. UML Sequence Diagram

4.1 Officer applying for BTO

A higher resolution UML Sequence Diagram is in a separate file named "SequenceDiagram1.png"

4.2 Officer registering to handle a project

A higher resolution UML Sequence Diagram is in a separate file named "SequenceDiagram2.png"

5. Tests and Results

The Tests and Results are in a separate file named "TestsAndResults.pdf"

6. Reflection

6.1 Difficulties Encountered

At the beginning of the project, we faced difficulties in designing a robust class structure that accurately represented the relationships between classes. We struggled to determine which attributes and methods belonged to each class, and how these classes should interact with one another. For instance, it was challenging to decide whether certain methods should reside in a parent class or be delegated to child classes, as this decision would affect the maintainability and clarity of the codebase.

We overcame this challenge by creating an ER diagram to visualise how the classes were connected and dependent on one another. This helped us identify the key responsibilities of each class and ensure adherence to the principles of cohesion and coupling in object-oriented programming. We then applied inheritance and polymorphism, assigning shared behaviours to parent classes and unique behaviours to child classes.

Another difficulty we encountered was that changes made to one table in our database did not propagate correctly to related tables. To address this, we implemented foreign key associations, ensuring that updates made in one entity class would propagate to the related tables. This maintained data consistency across the application.

As our codebase expanded, we also faced challenges in testing. To resolve this, we applied the ISP from the SOLID design principles, which helped simplify and improve the testability of our code.

6.2 Knowledge Learnt

During the development of our application, we gained firsthand insight into the significance of object-oriented principles. While these principles can be challenging to grasp in theory, implementing them in practice enabled us to truly appreciate their value.

For example, by adopting the MVC architecture, we experienced several benefits: the codebase became easier to understand, as each component had a clear responsibility; testing was simplified, as we could test each component independently; and adapting to new features was more manageable, as changes could be made without rewriting the entire application.

6.3 Further Improvements

To further improve user security, we could implement Two-Factor Authentication in addition to the username and password. This would prevent malicious actors from brute-forcing their way into user accounts, as they would still require the authentication code even after obtaining a user's credentials.

To make our application more intuitive, we could allow users to select their preferred language at the start menu. This ensures that users can apply for flats even if they do not understand English.

Lastly, we could add an optional survey during the logout process to gather user feedback, such as any pain points encountered while using the application. These insights could then be used to further enhance the user experience.