



SC2002 OBJECT ORIENTED DESIGN & PROGRAMMING

Report of Project Structure Design & Functionality





Build-To-Order (BTO) Management System

Github: https://github.com/Xermal/BTO_Application/tree/main

Javadoc: https://Xermal.github.io/BTO_Application/javadoc/

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
Jasper Chang Yang Kai (U2420549D)	SC2002 (CS)	FCS6	
Prakriti Muralidharan (U2422005C)	SC2002 (CS)	FCS6	
Daniel Loh Eng Yeok (U2422150B)	SC2002 (CS)	FCS6	
Phua Wei Jie (U2422084G)	SC2002 (CS)	FCS6	

Chapter 1: Requirement Analysis & Feature Selection

1.1 Understanding the Problem and Requirements

We began by reviewing the document and extracting all the nouns and verbs from the explicit requirements given to us. From the nouns, such as *Applicant*, *Officer*, *Manager*, and *Project*, we identified potential entity classes. The verbs, including phrases like *apply for project*, *submit enquiries*, and *view projects*, helped us determine possible methods for those classes.

Subsequently, we determined the relationships between the classes through UML diagrams, while taking SOLID design principles into consideration. It was noted that there were some ambiguities in the requirements, such as whether users were able to submit enquiries about any project or just their project, which were resolved after seeking clarification from our professor.

1.1 Deciding on Features and Scope

We consolidated all the features we have identified and grouped them based on three categories, core, optional and excluded.

CORE		
Log in	Change password	View projects
Apply for projects	Create/Delete project listings	Request booking/booking flat
Register for officer role	Approve/Reject applications	Withdrawal from application
View details and status of application	Submit/View/Edit/Delete enquiries	Approve/Reject registrations
OPTIONAL		
Toggle visibility of projects	Viewing projects based on a given filter	View details of handling projects
Generate report of applicants with their booking details	View details and status of registration	Projects that are no longer active will not be visible

EXCLUDED		
Sort/filter past application/registration history	Favourite projects	

We prioritised all the core features essential to the BTO booking process. Without these, the application process cannot function. By identifying the core features, we can ensure that a skeletal implementation of the application can be done on time before working on the optional features. Two features were excluded as they are purely quality-of-life features that do not really provide any significant improvement to the application.

Chapter 2: System Architecture & Structural Planning

2.1 Planning the System Structure

We drew on the four cornerstones of computational thinking to help guide the planning phase of the system layout: Decomposition, Abstraction, Pattern Recognition, and Algorithmic thinking.

Decomposition

At the start, we grouped related features and assigned them to individual entity classes. We mapped these out using a UML class diagram, which helped us physically see and document what entity classes we needed to implement and what methods each would require. This approach gave us a concrete starting point for building out the system structure in code.

Abstraction

We focused on defining the core classes in our UML diagram first without worrying about their relationships. By isolating the functionality first, we ensured that no key features were missed. Once all essential components were identified, we then established the relationships between them—effectively "connecting the dots" to form a complete and cohesive system model.

Pattern Recognition

Thereafter, we noticed repeating structures or behaviors across different features. For example, applying for a project, registering as an officer, and submitting enquiries all require input

handling, data validation, and error reporting. Recognizing these patterns allowed us to identify reusable utility methods and base classes that are not explicitly stated in the project description. This is how we identified the boundary and control classes, reducing code duplication and improving consistency and readability across the system.

Algorithmic Thinking

Once the entire system structure was more or less defined, we went ahead to implement the code logic behind each feature. Using pseudocode, we mapped out step-by-step instructions for each method, ensuring that each method was logically sound before implementation. Translating these flows into code became more straightforward, as team members can simply refer to the pseudocode documentation to guide them while coding.

2.1 Reflection on Design Trade-offs

While identifying control and boundary classes, we realised that the process can become confusing very quickly. This is because defining a single control or boundary class often requires establishing multiple relationships with other classes, which can make the class diagram overly dense and difficult to interpret. This is further amplified if we want to adhere to SOLID design principles in every class and relationship. To manage this complexity, we chose to prioritize simplicity over extensibility in some areas. Rather than creating separate control or boundary classes for every feature, we introduced them only when necessary and grouped similar methods where it made sense. This made it simpler for us to manage the flow of logic without losing track of the system structure and significantly limiting extensibility.

3. Object-Oriented Design

3.1 Class Diagram (Refer to BTO_Application/docs/UML/class diagram.jpg)

As mentioned earlier, the entity classes were identified by identifying the nouns in the project requirements, such as *User*, *Project* and *Applicant*. These classes will act as data containers that will be used in data flow. The associated verbs help define the methods for each class. Additionally, we identified control and boundary classes based on control type functionalities explicitly stated or implied, such as *view project*, *register*, *change password* and *display UI*. To determine how the classes should relate to one another, we looked at existing implementations of booking or registration systems to guide our thinking.

For instance, in the case of registering for orientations, each participant typically fills out a form, and at the end, they can type out any questions that they have for the organizers, which would be saved to a database. We drew a parallel from this to our system, where an applicant submits an enquiry for a project. This helped us establish a composition relationship between the *applicant* and *enquiry* classes, where an enquiry is created by an applicant and cannot exist independently. Drawing connections to the real-world made it easier to define relationships between our classes that make natural sense.

In the coding phase, we discovered new functionalities, which required further refinements to the class relationships to better reflect the system's behaviour. For example, we originally intended for the `ProjectManager.fetchAll()` to return `List<String[]>` to represent the details of all projects, but we realised that accessing project details from this return type is a hassle, when we can simply return a `List<Project>` and call the getter functions. As such, we draw a new association relationship between *ProjectManager* control class and *Project* entity class.

3.1 Sequence Diagrams

Key cases:

- HDB Officer Project Handling: Checks role conflicts (can't apply/manage same period).
- Flat Booking: Validates application, updates flat count, generates receipt.
- Eligibility Validation: Uses age/marital status to check if user can apply.

Involved objects: User, BTOProject, BTOApplication, ProjectRegistry

Important:

- Covers core logic, status transitions, real-time updates
- Tests role-based access, authentication, business rules

User roles and system flow:

- Applicant: applies and views status
- HDB Officer: Manages projects, books flats for successful applicants.
- HDB Manager: Oversees project creation
- Controller classes: (LoginSystem, ProjectRegistry) handle inputs and coordinate logic.
- Entity classes: (BTOApplication, BTOProject) enforce rules and updates.
- Repository: stores in-memory data

Collaboration:

- Officer registration: HDB Officer, ProjectRegistry, and BTOProject
- Flat booking: HDB Officer, BTOApplication, and ReceiptGenerator

- Eligibility check: User attributes (age/marital status) + validation logic.

Patterns + logic to demonstrate:

1. Authentication flow: validateCredentials() to enforce secure login access.
2. Role-based access control: The officer registration use case checks for conflicts based on user role and existing commitments.
3. Report generation with file output: Booking includes receipt output.
4. Conditional logic: Validates all actions (e.g., age-based eligibility).

Justification for sequence diagrams:

1. HDB Officer Registration- critical process involving multiple validation checks
2. Flat Booking Flow- demonstrates complex object coordination and output.
3. Eligibility Validation- Enforces rules based on user profile.

Improvements:

1. Add Boundary Classes: Include UI/boundary classes in your diagrams to better represent the Model-View-Controller pattern
2. Add Database/File Interactions: Illustrate file/database interactions for better system completeness
3. Include Validation Messages: Show more specific error and success messages in the return flows (e.g. "ERROR-BTO-001: Singles under 35 can only apply for 2-Room flats,)

```

alt Age < 35 and SINGLE applying for 3-Room
  AppC->>UI: ValidationError("ERROR-BTO-001: Singles under 35 can only apply for
  2-Room flats")
  UI-->>User: showErrorMessage("ERROR-BTO-001: Singles under 35 can only apply for
  2-Room flats")
else No available slots for flat type
  AppC->>UI: ValidationError("ERROR-BTO-002: No available slots for this flat type")
  UI-->>User: showErrorMessage("ERROR-BTO-002: No available slots for this flat type")
else Valid application
  AppC->>+App: new Application(selectedProject, PENDING)
  ...
  AppC->>UI: ApplicationSuccess("SUCCESS-BTO-001: Application #12345 submitted.
  You will be notified once processed.")
  UI-->>User: showSuccessMessage("SUCCESS-BTO-001: Application #12345 submitted.
  You will be notified once processed.")
end
  
```

A more detailed sequence diagram to show how HDB Manager Creates and Manages a Project:

<https://www.mermaidchart.com/app/projects/f93f89ec-d79f-472f-9161-e66c8f8c4109/diagrams/f8e4429e-f247-4310-9577-99afd25051d2/version/v0.1/edit>

In a real-world application, a separate diagram could depict the flat booking process for successful applicants.

4. Application of OOD Principles (SOLID)

The reference figures mentioned below are located in “**BTO_Application/docs/SOLID**”

Single Responsibility Principle (Refer to figure 1)

The Single Responsibility Principle (SRP) ensures that a class should only have one well-defined responsibility, where all methods defined in it should be closely related to one another. An example in our code where this is implemented is in our data managers. Each data manager has their own defined responsibility. DataManager handles utility readCSV and writeCSV methods, ApplicationManager deals with all applications for BTO methods, BookingManager deals with bookings, EnquiryManager handles enquiry-related logic, TimeManager checks project availability, ProjectManger handles project-related methods and UserManager handles fetching user data and changing password. This separation of responsibilities makes our codebase more organised, easier to debug, and more maintainable, where each class can be updated or tested independently without affecting unrelated parts of the system.

Open/Closed Principle (Refer to figure 2)

The Open Closed Principle (OCP) states that a class should be open to extension but closed for modification, where new functionality should be implemented without modifying existing code. An example in our code where this is implemented is in our UI classes inheriting from the IUserGroupUI interface. The IUserGroupUI interface defines a single abstract method runMenu, where the ApplicantUI, OfficerUI and ManagerUI can then implement their own runMenu logic. If we are to add a new user type like “Agent” or “Admin”, we don’t have to modify any existing UI classes. Instead, we can provide an extension by implementing IUserGroupUI and defining a new runMenu logic.

Liskov Substitution Principle

The Liskov Substitution Principle (LSP) states that objects of a subclass should be substitutable for objects of the superclass without affecting the correctness of the program. This principle is reflected in our use of the User superclass and its subclasses: Applicant, Officer, and Manager. Since all three subclasses extend User without altering its expected behavior, they can be safely

used wherever a User object is required, such as method parameters. For example, the runMenu method accepts a User parameter and functions correctly regardless of whether the actual object is an Applicant, Officer, or Manager. This includes scenarios where upcasting and downcasting are used, where objects of the subclasses can be passed to methods expecting a User without breaking the program. This has the added bonus of complementing the Open Closed principle that was mentioned previously, where new User types can be implemented without modifying the runMenu method arguments. With the Liskov substitution principle, it ensures that the code is flexible and easily extensible to incorporate new user types.

Interface Segregation Principle (Refer to figure 3)

The Interface Segregation Principle (ISP) states that no class should be forced to depend on methods it does not use. This means that interfaces should be small and specific rather than large and general, so that classes pick which methods they need. This is best shown in the three interfaces: IFilterProjectsByUserGroup that define the getValidProjects method, ISortProjects that define the sortProjects method and IViewFilter that define the view method. Each interface has only a single abstract method, keeping them small and focused. This keeps the code easy to extend as it allows new functionalities to be added without redundant method definitions that are not needed.

Dependency Inversion Principle (Refer to figure 4)

The Dependency Inversion Principle (DIP) states that high-level modules should not directly depend on low-level modules. This means that both should depend on an abstraction, which acts as the middle layer that maintains the independence of high level modules. This can be seen in the interaction between ProjectSorter class and ISortProjects interface. Rather than directly calling the different concrete sorting implementations sortByLocation, sortByName, sortByThreeRoom etc, it depends on the abstraction ISortProjects. By providing the parameter sortType, the appropriate sorting implementation is fetched and automatically executed to return a sorted list of projects as desired. This keeps the code flexible and easy to extend. This allows for polymorphic behaviour and ensures that ProjectSorter does not need to know the underlying logic of each sort type.

APIE Design Principles:

Abstraction

Classes like ApplicationManager, EnquiryManager, and BookingManager offer abstracted operations like applyTo(), viewApplication() and requestWithdrawal(), letting other classes interact with the system without knowing how the internal data is managed

Polymorphism

Interfaces such as IViewFilter and IUsergroupUI allow classes to call different implementations of the same method name and parameter, promoting flexibility of code.

Inheritance

As Officer shares similar functionality with Applicant, Officer inherits from the Applicant class, promoting code reusability and maintainability.

Encapsulation and Information hiding

Each class has its own private attributes (e.g., Project class encapsulates projectName, openDate, visibility, etc.), preventing access. Sensitive data access methods like writePassword are kept private with a public access wrapper to prevent direct data access.

Chapter 5: Testing

5.1 Test Strategy

We mainly did manual testing to test our code.

5.2 Test Case Table

Login System	
Input	Input is properly validated to allow only Integers in choice selection, or String if needed
UserID	Not case sensitive for letters Eg. s1234567A
Password	Case sensitive, exact string matching
Access Level	Eg. Applicant is unable to login as Officer/Manager
Password Change	Must not be the same password. Both the new password and confirmation password must be the same before it is changed.

Viewing Projects (Shows only what is needed)	
Married Status	Singles can only view available 2-Room flats. Married individuals can view both 2 and 3-Room flats.
Flat Vacancy	If a project has no more available slots, they are not shown.
Open Date - Close Date	Current date is checked to see if it falls within the availability period. If the project has expired, it is not shown. If the project is upcoming, it is still shown.
Visibility	If set to not visible by a Manager, the project is not shown.
Sorting	Adding a variety of projects to ensure the sorting is valid. Name or Location (Alphabetical), Flat Availability or Price (Ascending)

Creating Application (Applying for BTO)	
No Existing Application	Checks if the applicant has an existing pending or successful application.
Project Availability	Checks if the project entered is available to the applicant, using logic as stated in "Viewing Projects", including if there are no projects available at all. Unable to apply for a project if it is not yet active.
Room Availability	Automatically applies singles for a 2-Room flat if there are vacancies, gives the option to choose between 2 or 3-Room flats for married individuals.
Officer as Applicant	If an officer applies as an applicant, checks if there is an existing registration for the project they are applying for. Also checks if the officer holds appointments for existing active projects.
Viewing Applications	
Existing Application	Displays application unless the user has not applied/ no application.

Creating an Enquiry	
Project Availability	Project has to be available/ visible to the user for an enquiry to be made.
Editing and Deleting Enquiries	
Officer Response	If an officer has answered the enquiry, the applicant can no longer change it.
Applicants	Applicants can edit/ delete only the enquiries that they make.
Replying (Officers)	Officers can only reply to enquiries without an answer, and only to enquiries from projects they are handling.
Viewing Enquiries	
Project Availability	Applicants can only view enquiries from projects available to them. No enquiries shown if no enquiries were made/ no projects are available.
Handling Projects (Officers)	Only shows enquiries from projects handled by the officer. No enquiries shown if no enquiries were made/ no projects are handled.

Handling officer registrations	
Accessing officer registrations	Flags out any invalid officer ID.
Approve officer registrations	If the number of officers exceeds the project limit stated, then no other officers can be approved for the project.
Editing officer registrations	Any input other than “approve” or “reject” will be automatically deemed invalid and the edit will be cancelled.

Altering projects	
Edit project	If the project name entered does not exist, the system returns “Project not found”.
Add project	Checks if the new project only has 2 or 3-room flats being created, any other type of

Altering projects	
	flats (e.g. 4-room) will be denied and the system will prompt the user to key in a valid number again (e.g. 2 or 3).

6. Documentation

6.1 Javadoc :

https://Xermal.github.io/BTO_Application/javadoc/

7. Reflection & Challenges

1. What went well

We clearly separated our control, boundary, and entity classes, which made everything more manageable as we progressed. It was especially helpful when we were debugging or writing new features. We also made a conscious decision to simulate repository behavior with in-memory lists instead of adding unnecessary complexity, and that kept our focus on logic rather than technical overhead. Designing our sequence diagrams upfront helped us map out the user flows clearly, which meant fewer surprises during implementation. Also, writing test cases for core functionalities like login, officer registration, and flat booking helped us catch bugs early and validate our logic throughout the build.

2. Improvements

Looking back, one of our biggest challenges was trying to make the system *too* realistic too early. We spent quite a bit of time handling edge cases that, while interesting, added more complexity than we initially expected. Some classes became a bit bulky with business logic. Eventually, we learned to split those concerns out into helper methods and keep our classes more focused. Also, since we were limited to a command-line interface, certain features like search, filters, and report viewing were harder to visualize and test. If we had the option for a Graphical User Interface, we think that would've made the system much more intuitive to interact with and demo.

We felt that there could have been some improvements to the design of our program.

Instead of embedding direct logic directly within each class, we opted to use dedicated Service or

Manager classes to handle the majority of the functionality. As some of the logic required multiple Classes, and required constant reading from CSV files, we decided to use the Manager Classes. This decision helped prevent individual classes from becoming overly bloated and difficult to read. However, in hindsight, it would have been beneficial to include minimal method definitions within the domain classes themselves. This would have allowed external users to quickly understand the capabilities of each class without needing to delve into the underlying implementation details.

One trade-off of this approach is that it can hinder extensibility and reduce the effective use of polymorphism. Additionally, consolidating too much functionality within a single Manager class risks violating the **Single Responsibility Principle (SRP)**, making the codebase harder to maintain. To address this, we should have further decomposed the Manager classes into smaller, more focused components. Such as ApplicationViewer or ApplicationEditor. This would have improved both extensibility and support for polymorphic behavior.

In addition, as this was our first time building such a program using OOP, we developed a greater understanding of OOP throughout the process, and came up with improvements that could not be easily implemented within the timeframe, as it required significant code to be rewritten. However, we can now take into consideration design choices and flaws gained through this experience for future such projects.

3. Lessons learned about OODP

- Encapsulation ensures that entity classes protect their internal data by exposing only necessary information through getters and setters, maintaining data integrity
- Polymorphism and abstraction allow different user roles (like applicants and administrators) to interact with the system through common interfaces, enabling flexible and scalable behavior.
- Low coupling and high cohesion keeps each class focused and independently testable and each class remains focused on a single responsibility and is easier to test, modify, and reuse
- Visual tools like sequence diagrams (for dynamic behavior) and class diagrams (for static structure) help in planning, understanding, and communicating the system design.

8. Appendix

Github: https://github.com/Xermal/BTO_Application/tree/main