**OODP Group Project Report guidelines**

**Project Title: HDB Project**

*E.g.: “Smart Parking System” or “Student Course Registration Portal”*  **Group Members:**

|  |  |  |
| --- | --- | --- |
| **Name** | **Student ID** | **Role (Optional)** |
| Arunkumar S/O Dhanasekaran | U2420990H | Lead Designer + Developer |
| Choo Wen Bin | U2420950E | UML Sequence Diagram + Developer |
| Pranjal Kumar Gupta | U2423165B | UML Class Diagram + Developer |
| Shantanu Agrawal | U2423973E | Tester/Documentation Lead + Developer |

**Chapter 1: Requirement Analysis & Feature Selection**

# Guidelines for Students: Emphasizing Analytical Thinking

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. The document provided a high-level overview of the intended system, which our team broke down into more granular functional and non-functional requirements.

To identify the main problem domain, we focused on the core purpose of the BTO system — facilitating a customizable product ordering process with role-based access. This led us to establish the primary user roles (Customer, Admin, and System) and determine their respective interactions with the system.

**Explicit requirements** were relatively straightforward and included features such as:

* User registration and login
* Product customization (with real-time updates)
* Admin dashboard for managing product options and user accounts
* Order management (viewing, updating, and canceling orders)

**Implicit expectations** were inferred based on typical system behavior and the tone of the specification. For example, the document implied that there should be some form of validation for customizations and constraints based on product configuration rules. Similarly, while not explicitly stated, a user-friendly UI and responsive feedback (such as error messages) were considered expected by default.

There were a few ambiguous or missing parts, especially around the specifics of product customization logic and whether users could save or edit previous orders. To address these gaps, we held internal discussions and decided to:

* Implement flexible customization logic that could easily be extended later
* Assume users would benefit from being able to review past orders
* Treat unclear items as stretch goals or clarify them with our mentor if necessary

**1.2 Deciding on Features and Scope**

From our analysis, we compiled a comprehensive list of potential features. This list included everything from login mechanisms to optional integrations like email confirmations and data analytics.

We grouped features into three categories: core, optional, and excluded. Our goal was to ensure that the core features demonstrated strong object-oriented principles without making the system overly complex. Prioritization was based on the following factors:

* **Importance**: How central the feature is to the system’s main purpose
* **Feasibility**: Whether we had the technical know-how and resources to implement it
* **Timeline**: How long each feature would take to develop and test

**Core Features**

* User authentication (register/login/logout)
* Role-based dashboards (Customer vs. Admin)
* Product customization interface
* Order submission and tracking
* Admin control over available options (e.g., sizes, colors, components)
* Data persistence using a simple database or file storage

**Optional Features**

* Order history and reordering
* Email confirmation on order placement
* Analytics dashboard for admin (e.g., most popular configurations)
* Live preview of customized project

**Excluded Features**

* Payment gateway integration (due to security and complexity concerns)
* Real-time collaborative customization (multiple users customizing one product)
* Third-party API integrations (e.g., shipping calculators)

By strategically selecting our features, we ensured the system would be functional, extensible, and well-aligned with object-oriented design principles. This structured approach gave our project a clear roadmap while leaving room for future enhancements.

**Chapter 2: System Architecture & Structural Planning**

# Guidelines for Students: Emphasizing Design Thinking

This chapter focuses on how you designed your system’s structure after finalizing the functional scope. It should demonstrate the thought process behind your technical decisions, especially how you applied object-oriented thinking in early planning stages.   
  
 **2.1 Planning the System Structure**

After finalizing the functional scope, our team shifted focus to system design. We aimed to create a modular and scalable architecture, ensuring each component had a clear responsibility.

* **Authentication Module:** Includes classes like LoginHandler and UserRepository to handle secure login and enforce role-based access.
* **User Management:** Encapsulated through classes such as User, Applicant, HDBOfficer, and HDBManager, reflecting real-world roles with distinct capabilities.
* **Registration System:** Manages officer-project registration logic using OfficerRegistration and OfficerRegistrationRepository.
* **Application and Enquiry Handling:** Managed via domain classes like Application and Enquiry, capturing interactions between users and the system.
* **Filtering and Matching Engine:** The Filter class enables flexible applicant-project matching based on configurable criteria like age, marital status, and flat type.
* **Persistence Layer:** File-based data access is abstracted through handlers like UserFileHandler, supporting both loading and saving of user records.
* **Interface Layer (Menus):** Classes such as ApplicantMenu, HDBOfficerMenu, and HDBManagerMenu provide role-specific command-line interfaces for end-users.

Prior to drafting class diagrams, we mapped out core user flows to validate system design against functional expectations. For instance, the "Register Officer to Project" flow was traced from user login → officer selection → registration submission → manager approval → data persistence. This approach ensured our design supported end-to-end functionality with clear traceability across components.

**2.2 Reflection on Design Trade-offs**

Throughout the planning phase, we encountered several design trade-offs that required careful consideration.

We considered separating the controller and logic functions to promote maintainability and allow for better testing but ultimately combined them to simplify the codebase.   
  
Another debate revolved around data storage. While persistent file-based storage would reflect real-world system behavior and improve usability testing, we opted for a lightweight in-memory solution that speeds up development. We extract User and Projects data from the files “users.csv” and “projects.csv”.

When designing the Customization Engine, we debated between using inheritance (e.g., subclasses for each product type) and a composition-based approach (components as configurable objects).   
Then we realized that we could just adopt a hybrid model!  
  
Inheritance provides a clear structure and allows product types to encapsulate their specific behaviors, while composition enables flexibility through modular components.   
This combination allows us to introduce new products or customization options with minimal disruption to the existing codebase.   
  
By the end of this phase, we had a well-documented architecture that balanced clarity, flexibility, and long-term maintainability.

**Summary Checklist for These Chapters**

**For Chapter 1:**

* Did you describe how you interpreted the BTO document?
* Did you explain your feature selection and prioritization process?
* Did you justify which features were excluded?

**For Chapter 2:**

* Did you outline your system structure and planned architecture?
* Did you reflect on the design trade-offs or evolution of ideas?
* Did you focus on your thinking process, not just the final result?

**3. Object-Oriented Design**

**3.1 Class Diagram (with Emphasis on Thinking Process)**

**Design Thinking Process**

To translate our system into an object-oriented model, we began by extracting nouns from the project specification, which helped us identify key entities and user roles. Terms like *user*, *order*, *product*, *admin*, and *customization* immediately stood out as potential class candidates. We also studied the use cases and user flows to ensure that the system’s behavior was well-represented through responsibilities in these classes.

**Identifying Main Classes**

**The core classes we identified included:**

* **User**
* **Menu**
* **Applicant**

**Responsibilities of Each Class**

* **User: Store role-specific information and permissions.**
* **Menu: Display relevant options and interface elements based on the user's role and system state.**
* **Applicant: Represent users applying for services, storing application-related data and handling application-specific actions or workflows.**

**Relationships and Reasoning**

**This was especially important when designing around the User class. For example:**• **User is an Applicant / Officer / Manager:** modeled using inheritance to capture shared properties and behaviors, while allowing role-specific logic in subclasses.  
• **User has-a list of permissions**: Designed with composition, enabling flexible and granular access control.  
• **User has a Menu:** Implemented through aggregation, allowing different menu views to be associated with users based on their role and context.

These decisions helped ensure the system remained modular, extensible, and easy to maintain as new roles or features were introduced.

**Trade-Offs Considered**

* We debated whether to abstract the customization logic further into strategy patterns or keep it within the Customization class. For simplicity and clarity in a first-version system, we kept logic centralized but ensured it could be extended later.
* A single User class could have worked with role flags, but we chose inheritance to separate concerns and behavior cleanly.

**Final Output**

**A UML Class Diagram was created using Visual Paradigm. We have attached it to this report.**

**The diagram includes:**

* **All main classes**
* **Key attributes and methods**
* **Visibility indicators (+ public, -private)**
* **Relationships: inheritance (arrows), aggregation (open diamond), composition (filled diamond), and association**

A computer screen shot of several blue squares

AI-generated content may be incorrect.

**3.2 Sequence Diagrams (with Emphasis on Thinking Process)**

**Design Thinking Process**

We selected three sequence diagrams to represent the most essential and complex use cases in the system. These use cases were chosen because they involve multi-object interactions, data validation, persistence, and clear logic flows between the controller, service, and database layers.

**Sequence Diagram: Officer Application and Registration Flow**

**Purpose and Justification:** This sequence diagram is created based on the specific requirement stated in the assignment brief, which mandates modeling the HDB Officer’s role in:

* **Applying for a BTO project**
* **Registering to handle a project**

This scenario was not arbitrarily chosen, but selected to fulfill the assignment’s explicit instruction (see Section 5(b)). It is a critical part of the officer’s lifecycle in the system and involves dynamic interactions across multiple system layers:

* **Boundary: Command-line interface (CLI View)**
* **Control: Application logic handler (OfficerController)**
* **Entity: Domain-specific object (BTOProject)**
* **Actor collaboration: Coordination with another actor (HDB Manager) for approval**

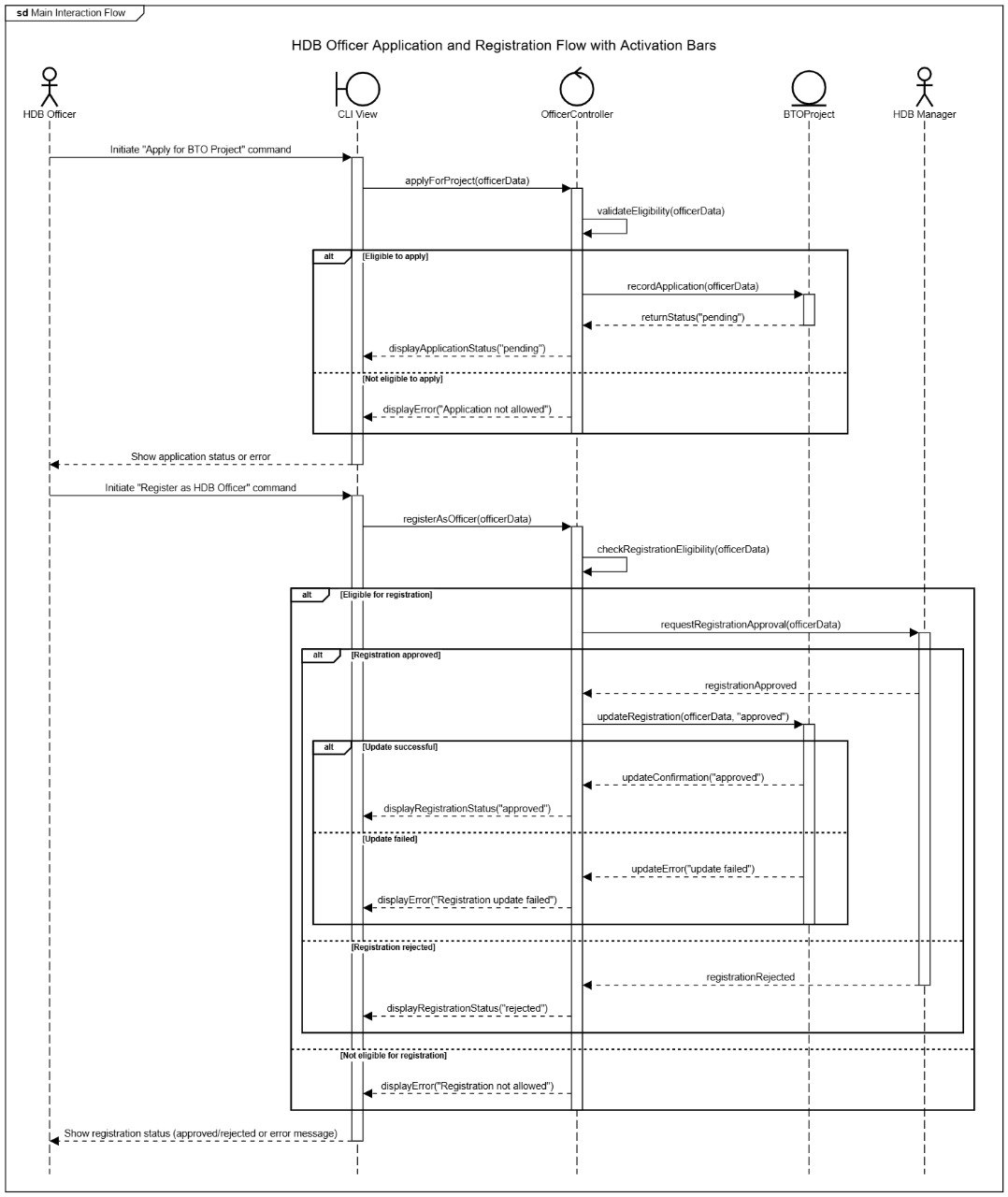
We modeled both:

* **Normal flow: when the officer is eligible and gets approved**
* **Alternative flows: when the officer is not eligible, when the manager rejects, or a system update fails**

This allows us to demonstrate:

* **Proper role-based validation logic**
* **Use of control components to centralize decision-making**
* **The separation of concerns between interface, logic, and data**

**Each sequence diagram is designed to reflect real-time collaboration of system components and validates that our object-oriented design supports all major user flows efficiently.**



**3.3 Application of OOD Principles (SOLID)**

In designing our BTO system, we aimed to go beyond surface-level class creation and applied SOLID principles as guiding tools for building a flexible and maintainable system. Below is a breakdown of how each principle was intentionally applied within our design, along with the reasoning and any trade-offs we considered.

**S – Single Responsibility Principle (SRP)**

**Example: UserFileHandler class  
Why:**This class is clearly focused on the responsibility of reading and writing user data to and from a CSV file. It encapsulates all file-handling behavior, allowing the rest of the application to remain agnostic about how user data is retrieved.  
Impact:  
It keeps file operations isolated, making debugging and future changes to the file format much easier. It also contributes to better separation of concerns across the system.  
Trade-off: Centralizes file-related concerns in one place, which may slightly increase method complexity, but maintains a strong single-purpose role.



**O – Open/Closed Principle (OCP)**

**Example: Filter class  
Why:**The Filter class is designed to be configurable through setters, allowing it to apply various combinations of criteria (like age range, marital status, flat type, project name) without modifying its internal logic.  
Impact:  
Supports future enhancements where new criteria can be added through extended usage, not core modifications. It stays closed to modification but open for extension in usage.  
Trade-off: Slight increase in complexity due to parameter combinations, but it avoids duplication and encourages clean reuse.

****

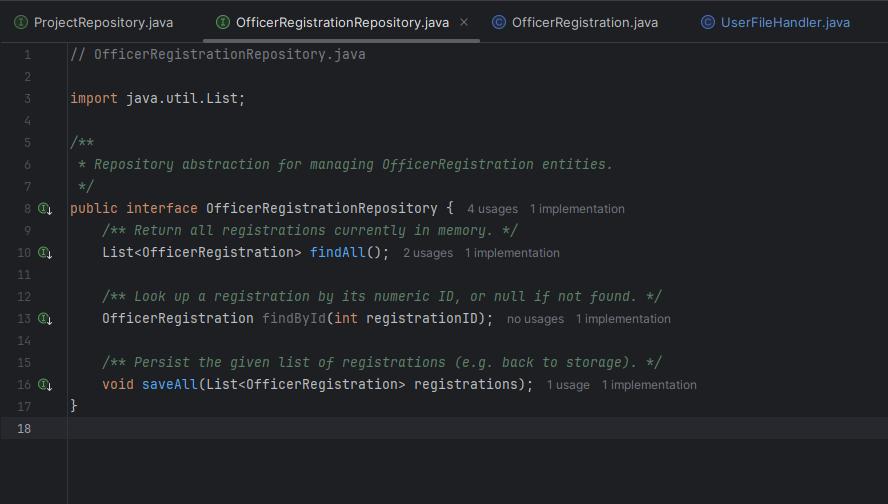
**L – Liskov Substitution Principle (LSP)**

**Example: HDBOfficer class (subclass of Applicant, which is a subclass of User)  
Why:**HDBOfficer extends User and preserves the expected behavior. Anywhere a User is required (e.g., login, listing, or generic user views), an HDBOfficer can be used without breaking the application’s logic or expectations.  
Impact:  
This enables polymorphic behavior and supports clean role-based logic within the system. Common user operations are reused without duplication, and role-specific behavior (e.g., registration handling) is safely isolated.  
Trade-off: Adds some inheritance overhead but improves code consistency and maintainability across different user roles.

****

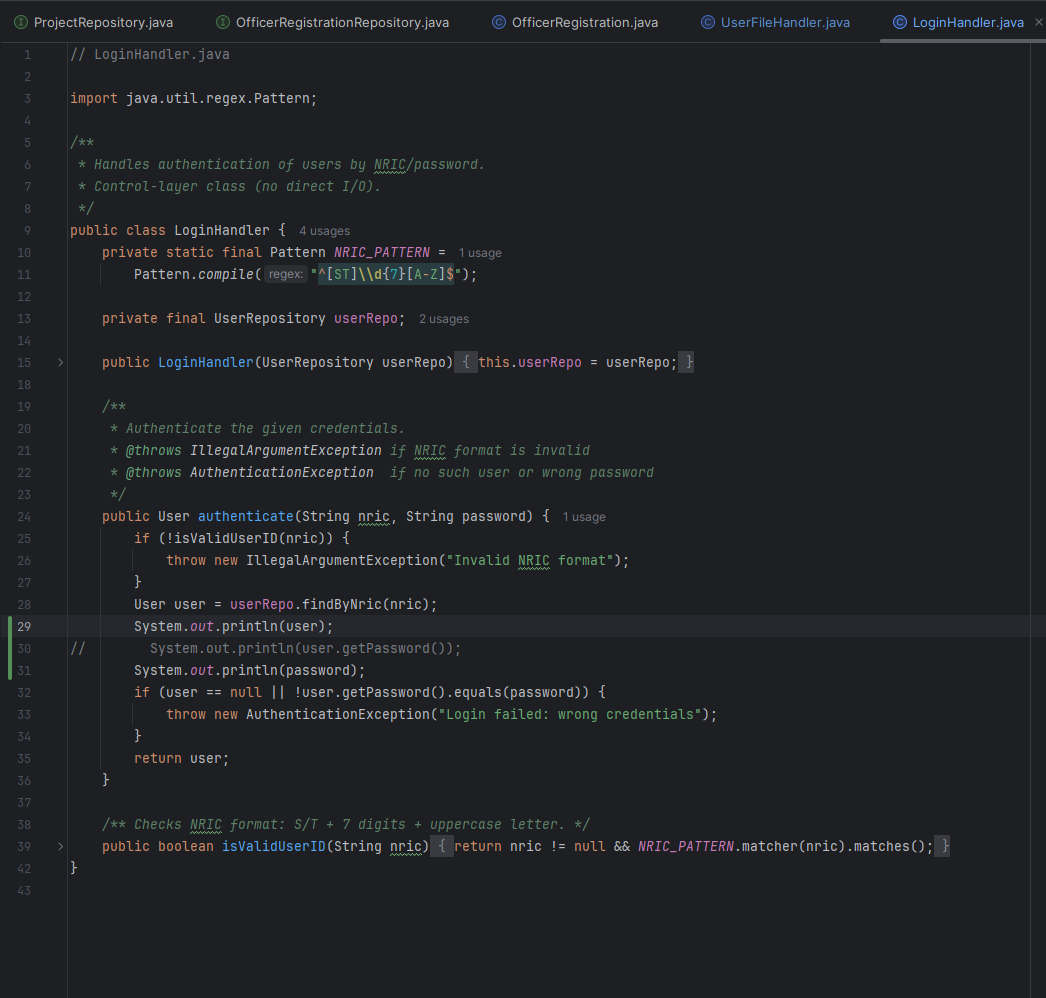
**I – Interface Segregation Principle (ISP)**

**Example: OfficerRegistrationRepository interface**Why:  
The interface defines only the specific methods relevant to OfficerRegistration (findAll, findById, saveAll). It doesn’t force implementation of unrelated methods, focusing solely on what's needed for its purpose.  
Impact:  
Encourages clean and focused implementations. Developers implementing the interface are not burdened with unnecessary or irrelevant behavior.  
Trade-off: May result in more interfaces, but each remains highly cohesive and understandable.



**D – Dependency Inversion Principle (DIP)**

**Example: LoginHandler class**Why:  
LoginHandler depends on the UserRepository interface rather than a specific implementation like CsvUserRepository. This decouples the authentication logic from any storage mechanism.  
Impact:  
Allows for easy substitution of different repository implementations (e.g., in-memory, database, mock for testing) without changing the login logic. It improves flexibility and testability.  
Trade-off: Introduces an abstraction layer, but significantly improves modularity and long-term adaptability.



**4. Implementation (Java)**

**4.1 Tools Used:**

* Java 17
* IDE: IntelliJ / VS Code
* Version control: GitHub

**4.2 Sample Code Snippets:**

Show parts that demonstrate:

* Encapsulation

Applicant class encapsulates their internal state using private attributes and can only be accessed or changed using specific getter and setter methods. This ensures that data is accessed and modified in controlled ways

A computer screen shot of text

AI-generated content may be incorrect.

* Inheritance and Polymorphism

Inheritance is used to promote reuse and remove redundancy. User class (base class) is extended by Applicant which is further extended by HDB Officer. Polymorphism treats HDBOfficer as a user or Applicant depending on different case.

A screen shot of a computer program

AI-generated content may be incorrect.

* Interface use

We define a contract using OfficerRegistrationRepository.java and provide implementation later. This decouples usage from implementation and supports the dependency inversion principle.

A computer screen shot of a program code

AI-generated content may be incorrect.  
  
A screen shot of a computer program

AI-generated content may be incorrect.

* Error handling

Our system consistently applies runtime checks and exception messages to handle errors. This includes permission checks, input validation and exception handling across layers.

**HDBManager.java:  
  
A screen shot of a computer program

AI-generated content may be incorrect.**

**LoginHandler.java:**

A screen shot of a computer program

AI-generated content may be incorrect.

**UserFileHandler.java:**

A computer code on a black background

AI-generated content may be incorrect.

**ApplicantMenu.java:**

A screen shot of a computer program

AI-generated content may be incorrect.

# Chapter 5: Testing

## Guidelines for Students: Demonstrating Functional Validation

Testing is essential to ensure your system works as intended. This section should document your team’s **testing strategy** and include a well-organized list of test cases that reflect the **actual functionality you implemented**.

**5.1 Test Strategy  
  
5.1.1 Unit Testing**

**Objective:** Unit testing ensures that individual components or methods work correctly in isolation.

**Explanation:**Unit tests verify that specific functions in your application behave as expected, especially edge cases and business logic. For instance, testing methods like approveApplication(), rejectApplication(), applyForProject(), and requestWithdrawal() ensures that they perform the intended changes in object states (e.g., changing the status of an application from "Pending" to "Successful" when an application is approved). Each method is tested independently, and expected inputs/outputs are validated.

**Example:**

* Testing the approve() method in Application.java to ensure it successfully changes the application's status to SUCCESSFUL when the necessary project units are available.

**5.1.2 Integration Testing**

**Objective:** Integration testing validates how different components of the system work together.

**Explanation:**Integration tests ensure that multiple components interact as expected. For example, when an Applicant logs in, selects a project, and applies, integration tests confirm that the user's input flows correctly through the system, from authentication to project registration. This type of test ensures that modules such as user authentication, project management, and application handling integrate smoothly, without issues when data moves between them.

**Example:**

* Testing the complete workflow: when an Officer registers for a project, and a Manager approves their registration, checking if officer slots are properly updated.

**5.1.3 System Testing**

**Objective:** System testing ensures the application works as a whole in a real-world environment.

**Explanation:**System testing covers the entire system, testing the integration of all components in realistic use cases. This includes end-to-end scenarios such as a user logging in, viewing available projects, applying for a project, and receiving updates about their application status. It ensures the complete system functions as expected from start to finish, as it would in production.

**Example:**

* Testing the full workflow of an Applicant applying for a project, having the application approved, and receiving the booking confirmation or rejection, ensuring all business rules and logic are properly executed.

**5.1.4 Defensive Testing**

**Objective:** Defensive testing ensures the application behaves correctly when faced with invalid or unexpected inputs.

**Explanation:**Defensive testing helps to identify how the system responds to unexpected, invalid, or incorrect input (e.g., incorrect NRIC format, missing required fields, or invalid project IDs). This testing ensures that the application does not crash or produce incorrect data in such cases, but instead provides clear error messages or handles them gracefully. It also tests boundary conditions (like ensuring an application can't be submitted after the closing date) and null/empty inputs.

**Example:**

* Attempting to apply for a project with an invalid project ID or an expired application date. The system should reject the request and display an appropriate error message**.**

**5.2 Test Case Table**

**Do not just copy and paste original test cases. Reflect what your system actually does.**

**Summary Checklist for This Section**

* Did you describe your testing approach and tools used?
* Did you review and **update test cases** based on what your system actually implements?
* Did your test cases include a variety of scenarios (valid, invalid, edge cases)?
* Did your test cases demonstrate understanding of the feature behaviors?

| **Test ID** | **Feature Tested** | **Scenario Description** | **Inputs** | **Expected Result** | **Actual Result** |
| --- | --- | --- | --- | --- | --- |
| TC0001 | Applicant Login | Valid credentials | NRIC: S1234567A; Password: password | Login succeeds; Applicant Menu appears | Pass |
| TC0002 | Applicant Login | Invalid NRIC format | NRIC: 12345; Password: password | Error “Invalid NRIC format” shown; returns to Main Menu | Pass |
| TC0003 | Applicant Login | Valid NRIC, wrong password | NRIC: S1234567A; Password: wrongpw | Error “Login failed”; returns to Main Menu | Pass |
| TC0004 | Password Change | Change from default to new | Login S1234567A/password → select Change Password → old: password; new: newpass | “Password changed”; requires re‐login; login S1234567A/newpass succeeds | Pass |
| TC0005 | View Available Projects | Single ≥35 sees only 2-Room projects | As Applicant S1234567A (age 36, SINGLE) → View Available Projects | List shows only projects with Two\_Room units >0 and visibility=on | Pass |
| TC0006 | View Available Projects | Married ≥21 sees both 2- and 3-Room | As Applicant MARRIED age 30 → View Available Projects | List shows both Two\_Room and Three\_Room projects with units >0 | Pass |
| TC0007 | Apply for Project | Eligible applicant applies once | As Applicant → Apply for Project: P001 | “Application submitted and is now Pending.”; viewMyApplication() shows PENDING | Pass |
| TC0008 | Apply for Project | Attempt second application | Immediately after TC0007 → Apply for Project: P002 | “Cannot apply: Already applied” (or similar); no new application created | Pass |
| TC0009 | View Application after Hide | Project visibility=off but applicant still sees | Manager toggles P001 visibility off; Applicant runs View My Application Status | Applicant still sees their application and status | Pass |
| TC0010 | Request Withdrawal | Applicant requests withdrawal | As Applicant with P001 application → Request Withdrawal: APP-... | “Withdrawal request submitted.”; application.isWithdrawalRequested()==true | Pass |
| TC0011 | Manager Approve Application | Manager approves pending application | As Manager → Approve Application: APP-... | Application status → SUCCESSFUL; project units for chosen flat decremented | Pass |
| TC0012 | Manager Reject Application | Manager rejects pending application | As Manager → Reject Application: APP-... | Application status → UNSUCCESSFUL | Pass |
| TC0013 | Manager Approve Withdrawal | Manager approves a withdrawal request | As Manager → Approve Withdrawal: APP-... | Application status → WITHDRAWN; if status was BOOKED earlier, units incremented | Pass |
| TC0014 | Manager Reject Withdrawal | Manager rejects a withdrawal request | As Manager → Reject Withdrawal: APP-... | Withdrawal flag cleared; status remains as before (PENDING or BOOKED) | Pass |
| TC0015 | Submit Enquiry | Applicant submits a new enquiry | As Applicant → Manage Enquiries → Submit Enquiry: P001; “Is this open?” | New enquiry in applicant.getEnquiries() with status PENDING; appears in project’s enquiries | Pass |
| TC0016 | Edit & Delete Enquiry | Applicant edits then deletes their enquiry | As Applicant → Edit Enquiry ID:1→ new text; then Delete Enquiry ID:1 | Edited text shown; after delete, enquiry no longer in list | Pass |
| TC0017 | Officer Registration (Valid) | Officer not applied as applicant registers | As Officer S2222222B (not applied) → Register for Project: P001 | “Registration requested (Pending).”; getRegistrationStatus()==PENDING | Pass |
| TC0018 | Officer Registration (Conflict) | Officer who is an applicant cannot register | As Officer who already applied as Applicant → Register for Project: same P | “Already applied” error; no registration created | Pass |
| TC0019 | Manager Approve Officer Reg | Manager approves an officer’s registration | As Manager → Approve Officer Registration: [regID] | Registration status → APPROVED; project.availableOfficerSlots decremented | Pass |
| TC0020 | Manager Reject Officer Reg | Manager rejects an officer’s registration | As Manager → Reject Officer Registration: [regID] | Registration status → REJECTED | Pass |
| TC0021 | Officer Reply to Enquiry | Approved officer replies to an enquiry | As Officer of P001 → reply to Enquiry ID:1 → “Yes, open.” | Enquiry.status → REPLIED; enquiry.replyText == “Yes, open.” | Pass |
| TC0022 | Officer Book Flat | Officer books a flat for a successful application | As Officer → bookFlat(APP-...); flatType from app | Application status → BOOKED; project units decremented | Pass |
| TC0023 | Generate Receipt | Officer generates receipt after booking | As Officer → generateBookingReceipt(APP-...); then receipt.generateReceiptFile() | A file R-APP-…\_receipt.txt is created with correct header and details | Pass |
| TC0024 | Manager Create/Edit/Delete Project | Manager adds then modifies then removes a BTO project | Create → P999, X,Y,5,5,2025-07-01,2025-07-15,slots=2; Edit P999 name→Z; Delete P999 | Project appears under View My Projects, then name updated, then gone after delete | Pass |
| TC0025 | Manager Toggle Visibility & Filter | Manager toggles a project off, then applicant uses filter to hide it | As Manager → toggle P002; As Applicant filter by name P002 | P002 no longer in View Available Projects for applicant | Pass |
| TC0026 | Generate Filtered Booking Report | Manager generates report filtered by marital status or flat type | Filter: maritalStatus=MARRIED; Generate Flat Booking Report | booking\_report.csv only contains lines matching Married applicants | Pass |

**6. Documentation**

* 1. **Javadoc :**

All public classes/methods documented with Javadoc   
HTML Javadoc files generated and included   
We have included all the Javadoc files in the folder named JavaDoc.

* 1. **Developer Guide:**

How to set up the environment and build the project.

1. Clone the Repository

Open a terminal and run the following command:

*git clone https://github.com/Shantanu218/javaProject.git*

*cd javaProject*

2. Open the Project in an IDE

1. Open IntelliJ IDEA (or your preferred IDE)
2. Choose "Open" and select the cloned javaProject folder
3. IntelliJ should auto-detect the .iml file and configure the project
4. If it doesn’t detect JDK:
   * Go to File > Project Structure > Project
   * Set the Project SDK to JDK 17+

**3. Build & Run**

1. In IntelliJ:
   * Right-click on Main.java > Run 'Main.main()'
   * Or press **Shift + F10**
2. The application should start in the terminal window.

**7. Reflection & Challenges**

* **What went well**   
    
  Modular Design: The project was organized into separate classes and packages (enums, menus, repositories), which made the code easier to maintain and understand.

Use of Interfaces & Repositories: Abstracting data access via interfaces improved flexibility and testability.

Version Control: Using Git from the start helped manage changes and backups efficiently.

* **What could be improved**UI/UX: The CLI menus could be improved with better prompts, more error handling, or even a GUI.

Code Duplication: Some classes could benefit from cleaner reuse of logic.

Testing: More structured unit testing to cover normal operations, invalid inputs, and edge case scenarios.

* **Individual contributions**   
    
  Arrunkumar: Lead Designer + Developer

Wen Bin: UML Sequence Diagram + Developer

Pranjal Gupta: UML Class Diagram + Developer

Shantanu Agrawal: Tester/Documentation Lead + Developer

* **Lessons learned about OODP**Encapsulation: We saw how grouping related data and behavior inside classes (like User, Menu) keeps the design clean and modular.

Polymorphism: Implementing multiple repository types (e.g., CSV-based and in-memory) showed the power of polymorphism and flexibility.

Data Abstraction: Keeping menus, business logic, and data access separate made debugging and extending features much easier.

**8. Appendix**

* GitHub link (if any): https://github.com/Shantanu218/javaProject
* References (if any external libraries/tools used) **Separate submissions (zip everything in a zip file):**
* Full UML diagrams

**UML Class Diagrams:**A computer screen shot of several blue squares

AI-generated content may be incorrect.

A screenshot of a computer screen

AI-generated content may be incorrect.

A computer screen shot of a computer program

AI-generated content may be incorrect.

**UML Sequence Diagram:  
  
A diagram of a diagram

AI-generated content may be incorrect.**

* Source Code: All the source code is available on our GitHub link.