**A logo of a university

AI-generated content may be incorrect.**

**SC/CE/CZ2002: Object-Oriented Design & Programming**

**AY 2024-2025 Semester 2**

**Group Assignment**

**Project Title:** Build-To-Order (BTO) Management System

**Group Members:**

| **NAME** | **STUDENT ID** |
| --- | --- |
| ALWINDERJIT SINGH BASANT | U2420597A |
| HONGCHAO LIN | U2322957G |
| LIM RAY’EN | U2422466C |
| JUSTIN WOON | U2422641B |
| DARRIUS YU DI QUAN | U2421474E |

# Chapter 1: Requirement Analysis & Feature Selection

## 1.1 Understanding the Problem and Requirements

We carefully reviewed the Week 8 BTO document line by line, highlighting explicit features and roles. From that, we mapped out the following:

**Explicit Requirements Identified:**

* Support for user login and password authentication.
* Role-based dashboards for Applicant, Officer, and Manager.
* Project creation and flat application workflows.
* Persistent storage of all records.

**Implicit Requirements:**

* Input validation and feedback.
* Enquiry & feedback functionality.
* Prevention of double booking, withdrawal flow.

**Ambiguous or Missing parts:**

* The CSV structure was incomplete as it did not meet our expectations. **To resolve this,** our system checks for missing columns at startup and auto-adds them.
* It wasn’t clear whether officers were a separate role or a subclass of applicants. **To resolve this,** we found similarities and modelled Officer as a subclass of Applicant.

## 1.2 Deciding on Features and Scope

We prioritised features based on **system necessity**, **feasibility within a CLI + CSV system**, and **our project timeline**:

### Core Features (Essential – Must Implement):

* **User Role Authentication**: NRIC & Password based
* **Applicant Features**:
  + User registration and login
  + Project viewing (with eligibility logic)
  + Flat application and withdrawal
  + Enquiry submission
* **Officer Features**:
  + Registration for the project assignment
  + Booking of flats for approved applicants
  + Project amenity and location editing
  + Enquiry viewing and reply handling
  + Receipt generation for booked applicants
* **Manager Features**:
  + Create, edit, and delete BTO projects
  + Toggle project visibility
  + Approve/reject applicant applications and withdrawal requests
  + Approve/reject officer registration
  + Generate reports with booking and payment details

### Optional or Bonus Features (Implemented if Time Permitted):

* **Feedback System** with resolver name, resolution tracking, and analytics
* **Officer–Applicant Role Switching** for officers with pending registration
* **Project Filtering for Managers** by visibility or manager-created projects
* **Amenity Management**: Officers can manage nearby MRTs, malls, schools

### Excluded Features (Not Implemented):

* **GUI interface. Not implemented because** it requires significant time and structural changes (e.g., JavaFX or Swing integration).
* **Multiple users at once. Not implemented because** CSV files are not concurrency-capable, a database backend is required, which is beyond the scope.
* **Geolocation API Integration**. **Not implemented because** it introduces monetary cost (API limits, tokens). This feature had limited value to the project

# Chapter 2: System Architecture & Structural Planning

## 2.1 Planning the System Structure

We began by sketching user journeys as flowcharts to visualise how each role would interact with the system, our **visual use cases as such:**

* **Applicant:** Register → View Projects → Apply → Withdraw/Success/Rejected
* **Officer:** Register to Project → Approve → Book → Generate Receipt
* **Manager:** Create Project → Approve Officers

**[Please refer to 3 Flowchart diagrams in the documentation folder]**

### Breaking the System into Logical Components

From the above, we broke the system into clear logical layers following the Model-View-Controller (MVC) architecture for parallel development and better code readability.

* **Model:** All entity representations (User, Application, Project, Receipt)
* **View:** CLI menus (ApplicantMenu, OfficerMenu)
* **Controller/Service:** Business logic (Booking, Application Validation)
* **Util:** File I/O, validation, CSV parsing

**[Please refer to MVC diagram image in the documentation folder]**

## 2.2 Reflection on Design Trade-offs

During early planning, our team discussed different design strategies and weighed the trade-offs

between simplicity, extensibility, and time feasibility.

### What trade-offs did we consider?

* **Inheritance:** Inheriting from Applicant reduced duplication but blurred role boundaries, which may cause issues in larger systems.
* **Tightly Coupled Menu Logic:** Menus directly invoke services, reducing modularity. More controller classes would improve extensibility and support OCP.

### Debated Design Approaches

We evaluated several design approaches before settling on MVC:

* **Monolithic Design:** Too coupled, hard to test or scale — rejected early.
* **Microservices:** Overkill for CLI with file-based storage — unnecessary complexity.
* **MVC:** Balanced structure, easy to test, scalable for future GUI. **(We chose this)**

# Chapter 3: Object-Oriented Design

## 3.1 UML Class Diagram + Thinking Process

We identified classes based on nouns, roles and entities involved from the requirement brief.

* Users: Applicant, Officer, Manager all inherit from User
* Core entities: Project, Application, Feedback, Enquiry, Invoice
* Services: Handle logic without cluttering models (e.g., ReportService, OfficerService)

**Responsibilities of each class**

Each class was designed with the Single Responsibility Principle in mind.

| **Class Type** | **Responsibility** |
| --- | --- |
| Model classes | Represent system data (e.g., Application tracks applicant status, Invoice holds payment details) |
| User classes | Inherit from User, carry authentication and role-specific properties |
| Service classes | Handle role-specific workflows, file persistence, validations, and updates |
| Utility classes | Shared helpers like InputValidator, CsvUtil for parsing and type checking |

## Determining Relationships

***We asked ourselves two questions repeatedly to determine relationships:***

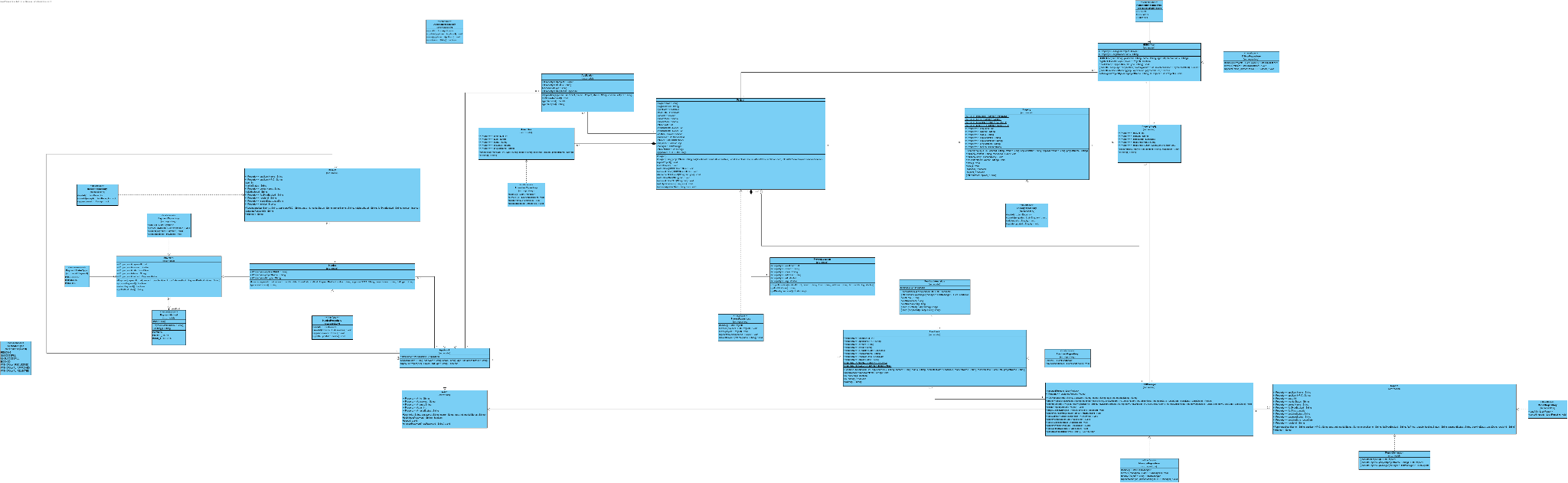
1. “Is this concept complex enough to deserve its own class?”
2. “Is this relationship based on identity or ownership?”

| **Type** | **Decision Example** | **Reasoning** |
| --- | --- | --- |
| Inheritance (is-a) | Officer extends Applicant | Code reuse (shared fields & methods) |
| Association (has-a) | Project ↔ Application | One-to-many applicant–project relationship |
| Composition (part-of) | Project has a Manager in-charge | Projects cannot exist without a manager in-charge |
| Aggregation | User (E.g. Applicant) objects can exist independently of the Project | A project has many applicants, but applicants still exist independently |

## 3.2 Trade-offs Considered

| **Trade-off** | **Explanation** |
| --- | --- |
| Simplicity vs. Abstraction | We kept Application as a separate class rather than nesting everything into User to promote reuse and clarity |
| Flexibility vs. Code Reuse | Using inheritance allowed for reuse, but added complexity when differentiating behaviors at runtime. |
| Data Integrity vs. Performance | Instead of caching related objects, we store ID to simplify CSV linking and avoid circular references. |

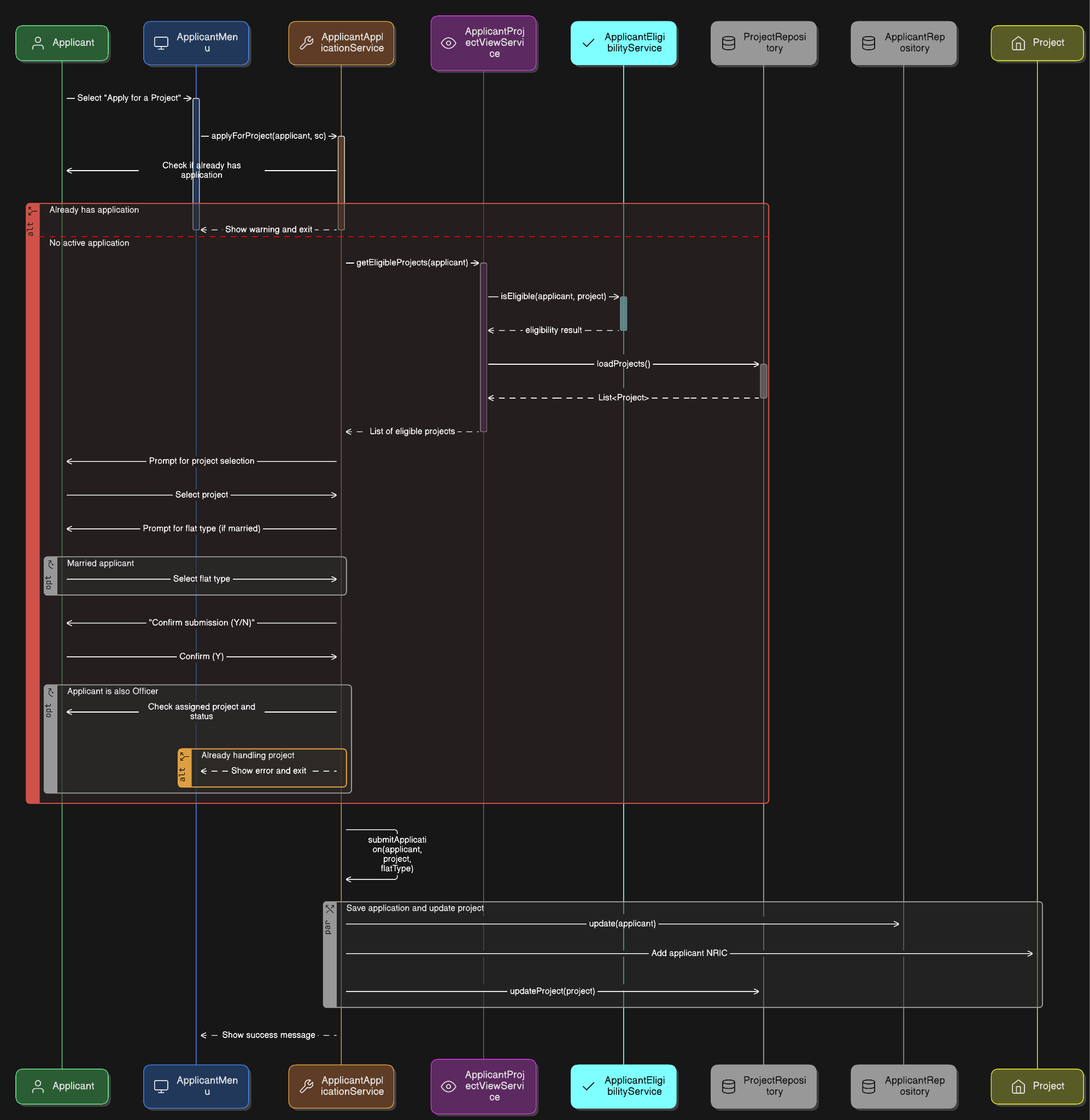
### Class Diagram (Please refer to class diagram image in the project folder as its too big)



## 3.3 Sequence Diagrams + Justifications

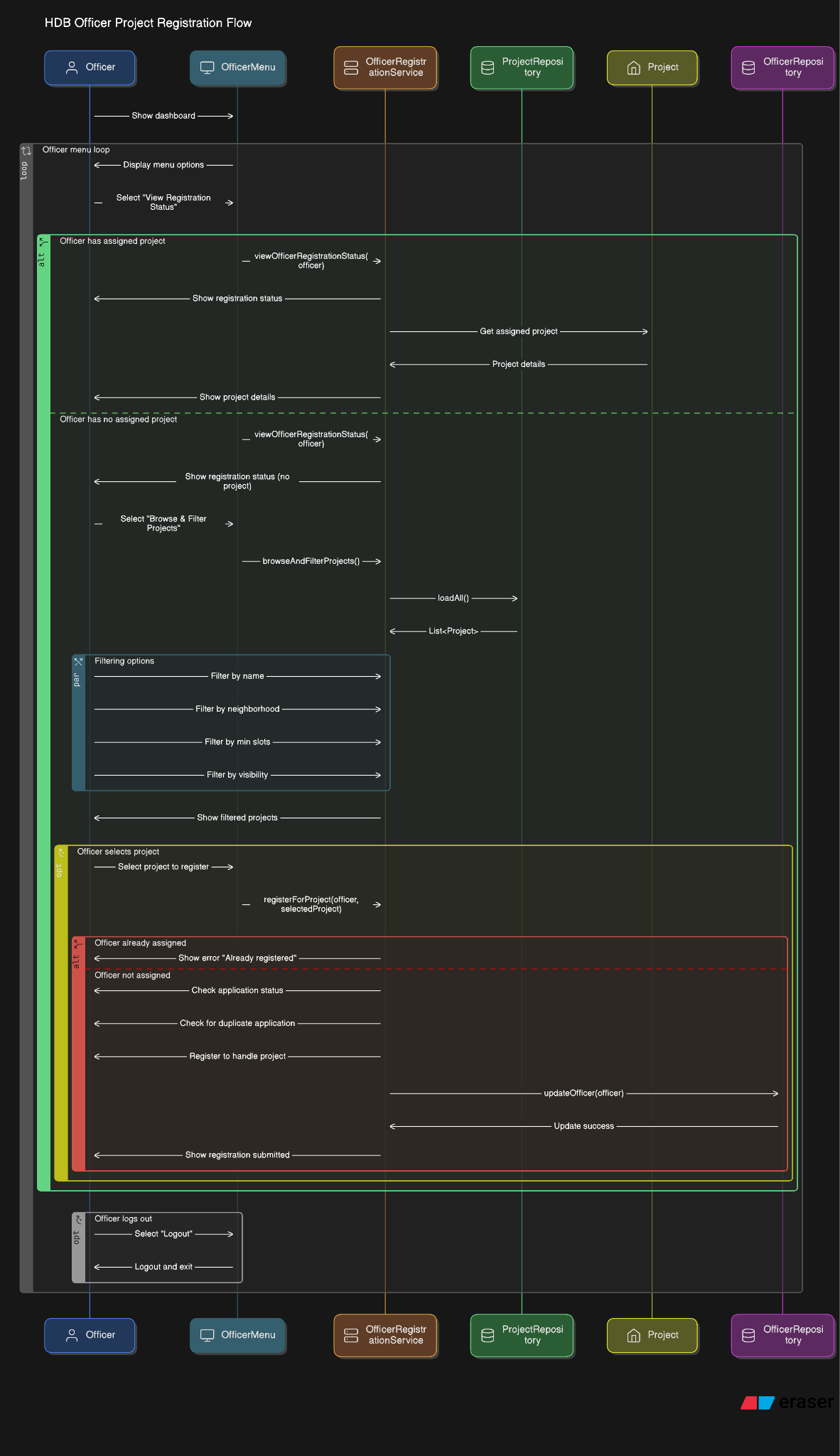
Diagrams below chosen because it **reflects MVC, complex messaging and conditional logic.**

**Officer as a Applicant Sequence (View enlarged version in project folder as its too big)**

****

**Justification:** This use case highlights our system’s support for dual-role users; officers acting as applicants. This use case tests officers applying as applicants, showcasing role-based flow handling. It validates our architecture’s ability to manage dual roles with proper separation, validation, and user guidance.

**Officer Sequence (View enlarged version in project folder as its too big)**



**Justification:** This scenario demonstrates the officer’s project registration flow, highlighting role-specific access, assignment checks, and repository updates. It validates our modular use of services and ensures proper separation of logic across roles.

## 3.4 Application of SOLID Principles

Below are examples of how each **SOLID principle** was meaningfully applied in our project.

### **Single Responsibility Principle (SRP)** | Example: OfficerInvoiceService class

**Why & Reflection**: By isolating this logic into a dedicated class, we make the code easier to maintain and reuse. If the receipt issuance logic changes, we only need to update this service.

| public class OfficerInvoiceService {  public static List<Invoice> getInvoicesAwaitingReceipt(HDBOfficer officer) {  return InvoiceService.getAllInvoices().stream()  .filter(i -> "Awaiting Receipt".equalsIgnoreCase(i.getStatus()))  .filter(i -> ReceiptService.findByInvoiceId(i.getPaymentId()) == null)  .filter(i -> i.getProjectName().equalsIgnoreCase(officer.getAssignedProject().getProjectName()))  .toList();  } } |
| --- |

### Open–Closed Principle (OCP) | **Example:** ApplicantRepository interface

**Why & Reflection:** This approach makes our system more flexible and maintainable. If we decide to migrate to a database later, we can introduce DatabaseApplicantRepository without changing any controller or service logic. It also improves testability.

| public interface ApplicantRepository {  List<Applicant> loadAll();  void update(Applicant applicant);  void save(Applicant applicant);  boolean exists(String nric); }  public class CsvApplicantRepository implements ApplicantRepository {  *// Implementation using CSV files* } public class DatabaseApplicantRepository implements ApplicantRepository {  *// Future implementation using SQL database* } |
| --- |

### Liskov Substitution Principle (LSP) | Example: Overriding isOfficer() Why & Reflection: isOfficer() is defined in User and overridden in HDBOfficer, allowing officers to be treated as general users without breaking behavior. This enables polymorphic logic while respecting class contracts.

| if (applicant.isOfficer()) {  HDBOfficer officer = (HDBOfficer) applicant;  *// officer logic...* } |
| --- |

### **Interface Segregation Principle (ISP)** | **Example**: Use of individual interfaces

### Why & Reflection: By segregating the interfaces, each service class only needs to implement or depend on what's relevant. This makes it easier to test or replace specific components.

### Dependency Inversion Principle (DIP) | **Example**: Constructor injection of interfaces

**Why & Reflection:** ManagerMenu depends on abstractions (IManagerReportService) instead of concrete classes. This improves modularity, testing, and flexibility for future changes like swapping implementations or mocking.

| public class ManagerReportService implements IManagerReportService { ... }  public ManagerMenu(IManagerProjectService projectService, IManagerReportService reportService, ...) { ... } |
| --- |

# Chapter 4: Implementation (Java)

### 4.1 Tools & Technologies Used

* **Java 17:** The language used to develop all components.
* **VSCode:** Our primary development environment for writing and testing code.
* **Git & GitHub:** Version control and collaboration.
* **CSV Files:** Data persistence

### 4.2 Code Snippets Demonstrating OOD & Implementation

**Encapsulation,** Internal data is hidden using private only accessible via getter/setters

| public class Project {  private String projectName;  private boolean isVisible;  private int availableFlats2Room;  private int availableFlats3Room;   public void decrementFlatCount(String flatType) {...}  public int getRemainingFlats(String flatType) {...} } |
| --- |

**Inheritance & Method Overriding,** HDBOfficer inherits from Applicant and overrides logic

| *// Parent class* public class Applicant extends User {  public boolean isOfficer() { return (this instanceof HDBOfficer) && this.application == null; } }  *// Subclass* public class HDBOfficer extends Applicant {  @Override  public boolean isOfficer() { return true; } } |
| --- |

**Polymorphism,** system resolves behavior at runtime based on actual object type.

| User user = AuthService.authenticate(nric, password);  if (user instanceof HDBManager manager) {  ManagerMenu.show(manager); } else if (user instanceof HDBOfficer officer) {  OfficerMenu.show(officer); } else if (user instanceof Applicant applicant) {  ApplicantMenu.show(applicant); } |
| --- |

**Interface Use,** interfaces decouple logic, allowing easy switching between implementations

| *// Interface* public interface ApplicantRepository {  List<Applicant> loadAll();  void save(Applicant applicant);  void update(Applicant applicant); }  *// Implementation using CSV* public class ApplicantCsvMapper implements ApplicantRepository {  @Override  public List<Applicant> loadAll() {*// Loads from CSV*}   @Override  public void save(Applicant applicant) {*// Appends to CSV*}   @Override  public void update(Applicant applicant) {*// Updates existing row*} } |
| --- |

**Example 5: Input Validation and Exception/Error Handling**

| public static String getNonEmptyString(String prompt) {  while (true) {  System.*out*.print(prompt);  String input = *sc*.nextLine().trim();  if (!input.isEmpty()) return input;  System.*out*.println("❌ Input cannot be empty.");  }  } |
| --- |

# Chapter 5: Testing

### 5.1 Test Strategy

We used a combination of manual and scripted testing to verify the system’s behavior.

**Manual Testing:** Core features were tested through the CLI using realistic user inputs. Invalid scenarios were intentionally tested to verify input handling.

**Scripted Testing:** We implemented ApplicantTest.java, ApplicantTestForSingle.java, and ManagerTest.java to simulate real user flows using input stream redirection to check logic and CSV persistence.

### 5.2 Test Case Table

| **Test Case ID** | **Scenario** | **Role** | **Input** | **Expected Outcome** |
| --- | --- | --- | --- | --- |
| TC001 | Register new applicant | Applicant | Valid NRIC, Name, Password | New applicant appears in ApplicantList.csv |
| TC002 | Apply for BTO flat | Applicant | Select project and flat type | Application is created and marked as PENDING |
| TC003 | Book flat for applicant | Officer | Select applicant with SUCCESSFUL status | Flat count reduced, status updated to BOOKED, invoice generated |
| TC004 | Approve officer registration | Manager | Select pending officer from list | Officer status updated to APPROVED, project’s officer slot decremented |
| TC005 | Generate invoice and receipt | Officer | Select applicant with successful application | Invoice and receipt saved in InvoiceList.csv and ReceiptList.csv |
| TC006 | Booking after unit quota filled | Officer | Attempt booking when no flats remain | Booking rejected with appropriate error message |
| TC007 | Withdraw application after approval | Applicant | Select active application | Application status updated to WITHDRAW\_REQUESTED |
| TC008 | Submit enquiry and receive reply | Applicant | Enter enquiry message | Enquiry saved and reply appears from officer/manager |
| TC009 | Attempt login with incorrect password | All Users | Valid NRIC, wrong password | Login denied with error message |
| TC010 | Approve non-existent officer | Manager | Select invalid NRIC | Error message shown, no changes made |
| TC011 | Submit feedback | Applicant | Enter feedback text and project | Feedback appears in FeedbackList.csv with status PENDING |
| TC012 | Resolve feedback | Officer/Manager | Select feedback and resolve it | Status set to RESOLVED, resolver name and resolved date recorded |
| TC013 | View eligible projects | Applicant | Age ≥ 35 (single) or ≥ 21 (married) | List of projects displayed that match criteria and are visible |
| TC014 | Enquiry auto-closes after reply | Officer/Manager | Reply to an enquiry | Enquiry status changes to CLOSED automatically |
| TC015 | View application status | Applicant | None | Current application status is displayed (e.g., PENDING, BOOKED, etc.) |
| TC016 | Reject officer registration | Manager | Select pending officer and reject | Officer removed from list, status set to REJECTED, project unlinked |
| TC017 | View all receipts | Applicant | None | List of receipts filtered by applicant NRIC shown |
| TC018 | View all invoices and make payment | Applicant | Select unpaid invoice, choose payment method | Invoice status updated, payment written to PaymentList.csv |
| TC019 | Attempt to submit feedback without application | Applicant | No application submitted | Feedback rejected, error message shown |
| TC020 | Generate project booking report | Manager | Trigger report generation | Report data written to ReportList.csv, summary stats displayed |

### 5.3 Reflection on Testing Process

Manual testing helped us trace edge cases and logic flaws early in development. Scripted flows later verified consistent behavior across different user paths. While GUI-based or JUnit testing was not applicable, our test approach ensured coverage and validation of core business logic and system integrity.

# Chapter 6: Documentation

### 6.1 JavaDoc Usage and Coverage (Please refer to the JavaDoc HTML attached in ZIP file)

All public classes and methods were documented using JavaDoc. Comments were kept concise and human-readable, avoiding over-documentation of trivial methods like getters/setters. JavaDoc was generated and included in the submission.

### 6.2 Developer Guide

To run the system locally:

**Requirements:** Java 17+, CSV editor, terminal access

**Project Setup:**

* Clone or download the repository from GitHub / NTULearn.
* Ensure CSVs in the /data folder
* Execute Main.java.
* Follow the CLI prompts to register/login and interact with the system.
* To build the project, compile using javac or a supported IDE.
* Output files will be in /out or /bin depending on setup.

# Chapter 7: Reflection and Challenges

### 7.1 What Went Well

* Early use of MVC kept logic modular and testable
* Services and utilities improved readability and reuse
* JavaDoc and structured packages helped team understanding
* Collaboration & version control through GitHub

### 7.2 What could be improved

* CLI menus became complex with nested options
* CSV parsing edge cases (e.g., missing columns) needed extra handling
* Officer inheriting Applicant led to role confusion
* Limited test automation due to CLI nature

### 7.3 Individual Contributions

* **Alwin:** Built base system, officer/payment flow, refactoring, documentation
* **Rayen:** Developed applicant flow, JavaDoc, system logic improvements
* **Hongchao:** Led testing, built manager/report modules, documentation
* **Justin:** Testing support, documentation polish and review
* **Daras:** Sequence diagrams, requirement verification, documentation

### 7.4 Lessons Learned About OODP

* MVC improved clarity, debugging, and scalability
* Interface use enhanced flexibility (e.g., ApplicantRepository)
* Code reuse vs clarity trade-offs became clearer during inheritance
* JavaDoc and utility abstraction supported maintainable code

# Chapter 8: Appendix

### 8.1 GitHub Repository

<https://github.com/AlwinSingh/HDBSystem/tree/RAYEN_branch>

### 8.2 References

* VisualParadigm
* Git + GitHub