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
Goh Zhen Rong	BCG	FCSI	Goh Zhen Rong 19/4/25
Zhan Yi Yun	CSC	FSCI	Zhan Yi Yun 19/4/25

GitHub Link: https://github.com/masamune-prog/SC2002_bto

1. Requirement Analysis & Feature Selection

1.1 Understanding the Problem and Requirements

BTO Management System is a system that allows for Applicants to apply for BTOs and allows for HDB Managers and Officers to perform actions to offer the BTO to applicants, and process the BTO applications

By reading the document given and identifying nouns and verbs, we identify key classes like Applicant, Manager and Officer

1.2 Deciding features and scope

We identified features to be implemented in our BTO Management System which are listed below, and then categorised them into core, bonus and excluded features. Our prioritization strategy was guided by the need to deliver a **fully functional system that fulfils all essential user requirements first**, ensuring that the foundation of our system would be robust and

reliable. We also left room in our development plan for optional features – enhancements that, while not critical – could significantly **improve user experience** if time and resources permit.

Core:

1. Login (ALL)
2. Reset Password (ALL)
3. View BTO Projects based on eligibility (APPLICANT)
4. Apply for BTO Project (APPLICANT)
5. Request to book flat (APPLICANT)
6. Create enquiry (APPLICANT)
7. Register for BTO Project Handling (OFFICER)
8. View and Reply enquiries for Projects-In-Charge (OFFICER AND MANAGER)
9. View enquiries for all Projects (MANAGER)
10. Update Applicant's profile upon successful application (OFFICER)
11. Generate receipt of applicant's flat booking details (OFFICER)
12. Manage BTO Projects (MANAGER)
13. Manage HDB Officers (MANAGER)
14. Manage Applicants (MANAGER)
15. Generate Report of the list of applicants' flat booking details for a project (MANAGER)

Definitions

1. We defined the Enquiry system to be similar to an email system, where there is maximum 1 question and 1 answer at a time.
2. We define the report as the ability to filter Applicants that have Booked their flats. The report filters these users by their demographic.

Bonus:

1. Reflection-Based Model Handling
2. Generic Repository Class
3. Secure Password Encryption with SHA-3
4. Batch Import from CSV
5. Factory Design Pattern
6. Automatic Data Synchronization

2. System Architecture & Structural Planning

In our project, we separated our components into Model, View and Controller parts. This ensures that we would be forced to follow SOLID principles.

Model

1. We define the Model as the data structures defined to implement the Objects in real life, namely Applicant, Officer and Manager. We treat each role as a single user in the system.
2. These Objects are then stored in a universal Repository Class, which implements a mapping function which converts the data structures into strings to be stored in the txt files.

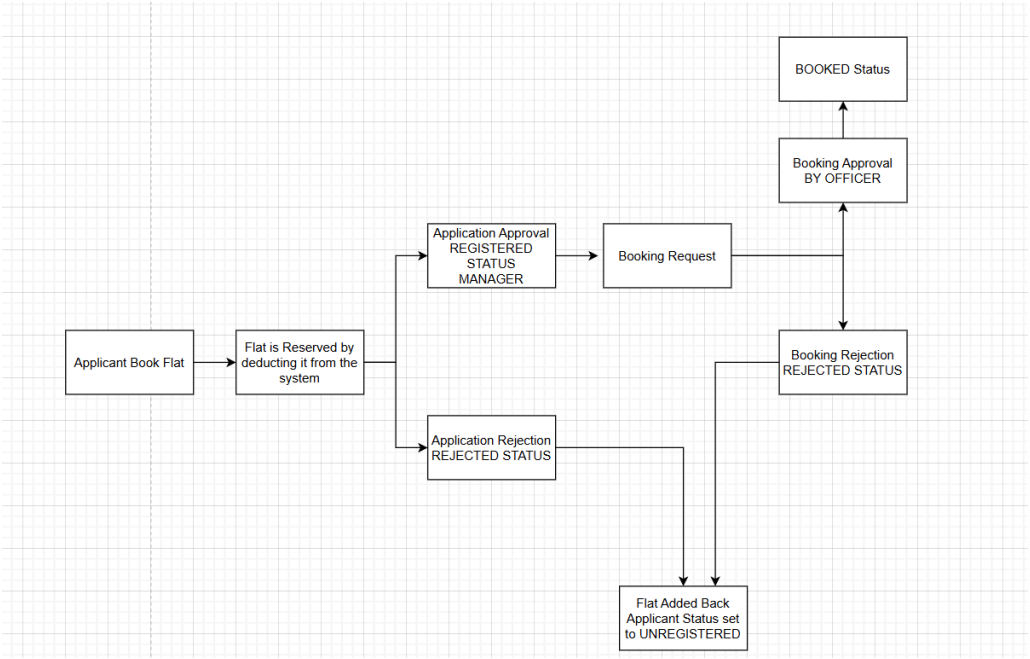
Controller

1. The controller classes handle the business logic of the Objects

View/Boundary

1. The boundary classes handle the display of the various interactions of the different Model Classes

In this project, each BTO project consists of 1 Manager, Maximum 10 Officers and a Limited amount of Applicants. To model the booking process, we made a flow chart



2.2 Reflection on Design Trade-Offs

In this project, we traded off simplicity for extensibility. Examples of such design choices included using Enums for different Status.

Initially, we explored using object composition to directly access applicant details from their associated objects.

However, this was difficult to properly Map and Store. We then relied on the NRIC of the Users as a foreign key to connect the different object repositories

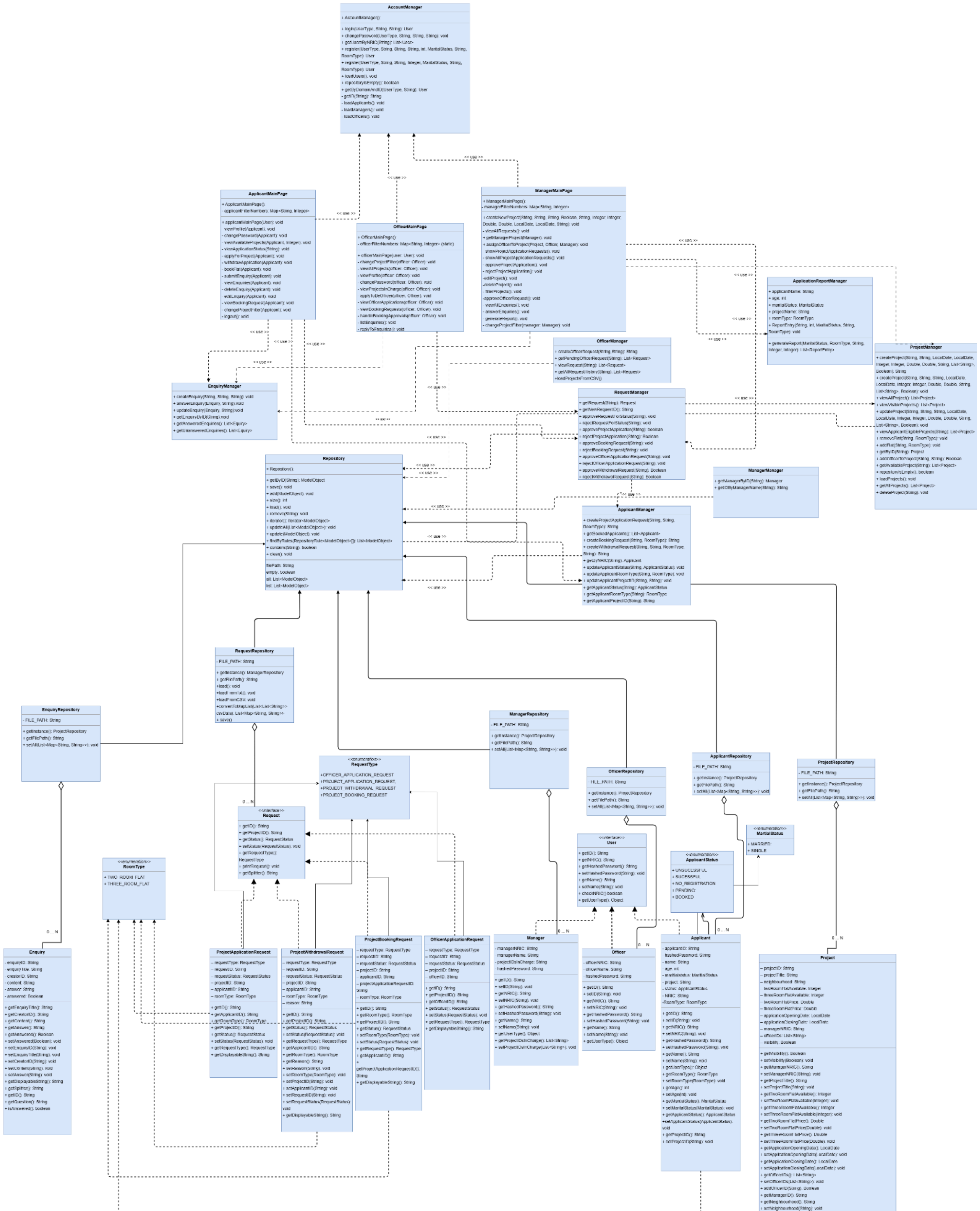
We also debated on using NRIC as the foreign key. This would not be ideal in production as NRIC numbers are sensitive. Alternatively, we could have generated UUID instead.

We believed that having a persistent system would save us time during testing. This was the main reason such a setup was used despite increasing the difficulty.

3. Class Diagrams

3.1 Class Diagrams(Links Provided At Appendix)

3.1.1 Main Class Diagram


AccountManager
ApplicantMainPage
OfficerMainPage
ManagerMainPage
ApplicationReportManager
ProjectManager
OfficerManager
EnquiryManager
RequestManager
ManagerManager
Repository
ApplicantManager
RequestRepository
EnquiryRepository
ManagerRepository
OfficerRepository
ApplicantRepository
ProjectRepository
RequestType
Request
MaritalStatus
User
ApplicantStatus
RoomType
Enquiry
ProjectApplicationRequest
ProjectWithdrawalRequest
ProjectBookingRequest
OfficerApplicationRequest
Manager
Officer
Applicant
Project

3.1.2 Design Thinking Process

We identified the main classes like Enquiry,Applicant,Officer,Project etc by looking at the nouns. We realised that a Project **has a** Applicant, Officer and Manager. This suggests association as the Users exist even when the Project is destroyed. We use the Users NRIC and the ProjectID as a form of foreign key to link the Users and the Project.

Each type of User(Applicant,Officer,Manager) **is a** User. This suggests that we can use an Interface to implement them. Similarly, this is the case for Requests as all different Requests is a type of Request.

To store the Users,Projects, Requests, Enquiries, we use a Repository. These have an object composition relationship as each Repository **is made of** 0 or many types of Objects. If the Repository is destroyed, all Objects are destroyed.

For Enquiry, it can only be created by a Applicant and answered by the Officer or Manager. We create a separate class and Repository for it. One Applicant can create as many Enquiry as they want.

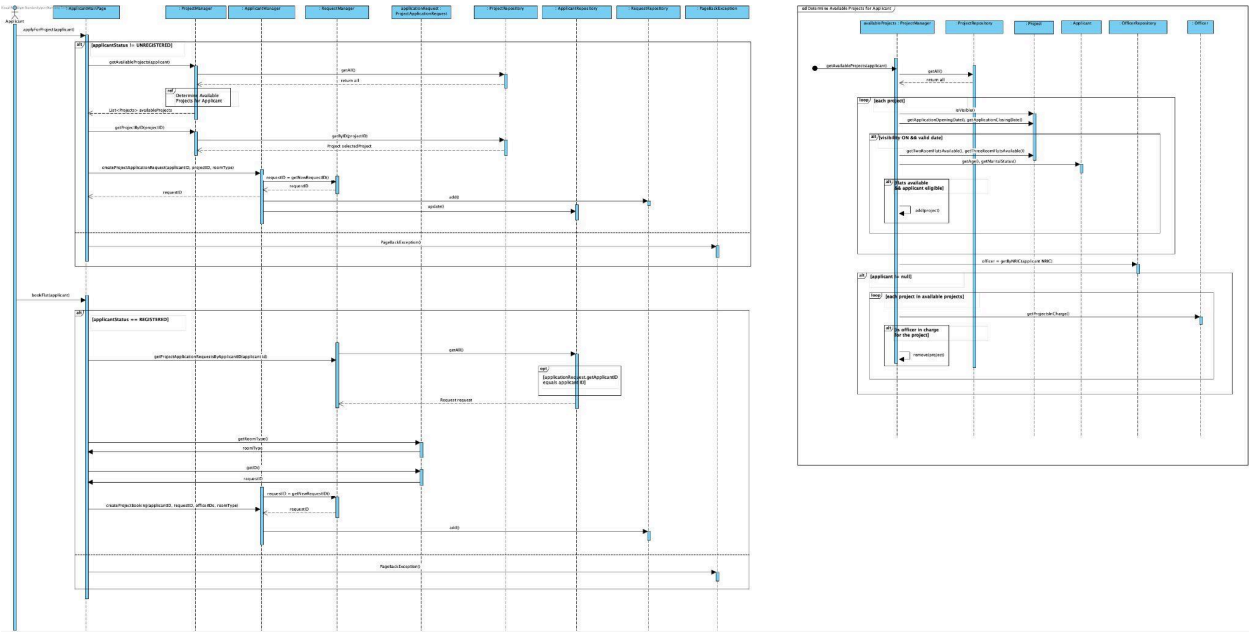
We define Receipt as a printed report after the booking.Similar in real life where people do not keep receipts, we can simply rely on the Users ProjectID and RoomType as these are only updated when the ApplicantBookingRequest is Approved.

We define Report as a printable report that allows the Manager to filter Projects based on his/her wishes.

We also see that there are **many kinds** of Statuses for Applicants and Request. We use Enums for these. Additionally, we use Enums for things like MaritalStatus and RoomType. Though the project only states that 2 kinds of each exist, in real life this is not true and using Enums makes expanding the project flexible as only an additional Type has to be added.

3.2 Sequence Diagrams(Links Provided)

Use case 1: Project Application to Booking Flow

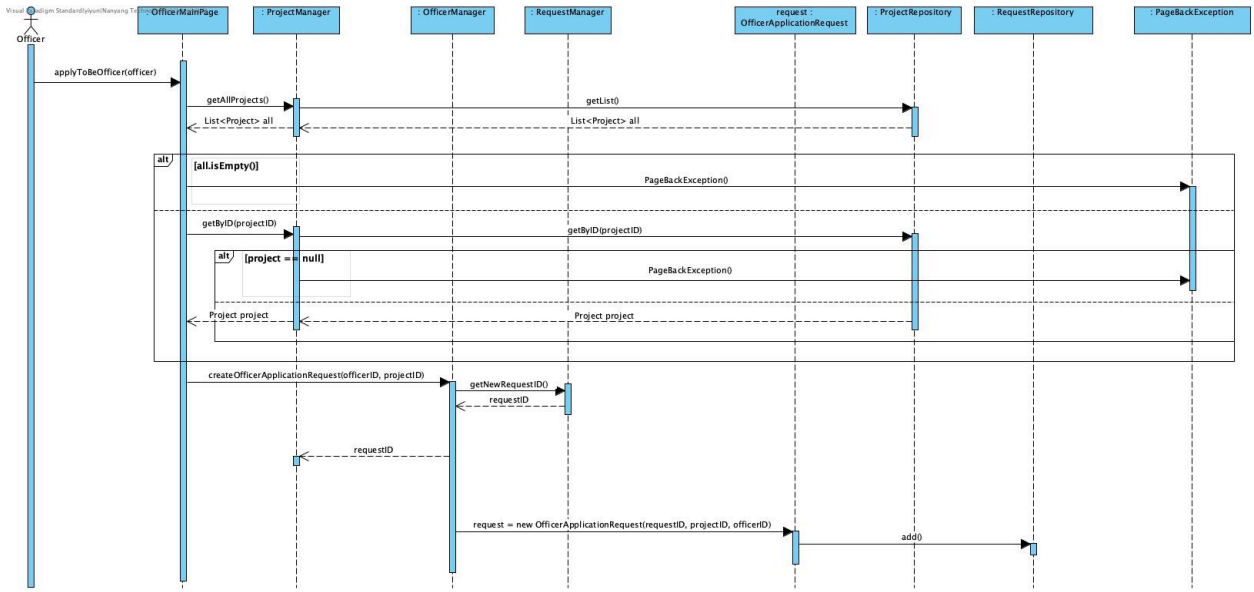


This use case illustrates the core functionality of our BTO system – which is to allow applicants to apply for and book a flat – and demonstrates our use of the Model-View-Controller (MVC) architecture.

For example, when booking a flat, the View layer (ApplicantMainPage) collects user input and calls the Controller layer (RequestManager) which retrieves applicant data from the Model layer (ApplicantRepository) to obtain the applicant's registered project. Once validated, the Controller layer (ApplicantManger) is able to generate the booking request, which is then stored via the Model layer (RequestRepository), ensuring data persistence. The flat number is deducted then to ensure no race conditions where too many applicants apply. If the application fails, the flat is then added back to the project.

If the application is approved, the Applicant Profile is updated with their ProjectID, RoomType and Status

Use case 2: Project Registration Flow



This use case demonstrates how our system handles role-specific access to shared data. In this use case, officers retrieve projects using ProjectManager.getAllProjects(), giving them full visibility of all available projects. In the previous use case, applicants access the same data using ProjectManager.getAvailableProjects(), which will filter visible projects based on applicant's eligibility and status. This validates that our design supports shared access to core resources like ProjectRepository, while maintaining distinct logic paths.

It also highlights our error-handling strategy, where inputs are first validated before core logic processing to ensure our system remains robust. For example, PageBackException is thrown if project repository is empty (i.e no projects exist) and if invalid projectID is entered by user during selection, which ensures proper handling or edge cases and prevents further processing of invalid inputs. When we show the available projects to the user, we already have filtered the projects based on their eligibility factor so they would not even be able to apply for projects.

3.3 Application of OOD Principles (SOLID) — With Emphasis on Thinking Process

3.3.1 Single Responsibility Principle(SRP)

The Single Responsibility principle recommends that each class should have a clear and singular responsibility, avoiding unrelated tasks. We ensure SRP by splitting the classes into MVC classes, where the model implements the data structures, the repository classes handle the saving and persistence, the controller classes execute logic to handle what has to be viewed along with updating User,Projects,Enquiry and Request and the boundary class handles only the printing of Objects and interactions.

3.3.2 Open/Closed Principle

The Open/Closed Principle (OCP) states that classes should be open for extension but closed for modification, allowing for the addition of new functionality without changing existing code. OCP can be implemented through abstraction, inheritance, and polymorphism.

In our project, OCP is applied by providing an abstract class, Repository<Model>, which serves as a base for creating specialized repositories like ProjectRepository and RequestRepository. Each subclass provides its own implementation of the getFilePath() method, making it easy to extend the repository system without modifying existing logic.

3.3.3 Liskov Substitution Principle(LSP)

LSP states that the subtypes can be a replacement for the base types.

In our system, this is used in the Request class. All subclasses of Request include the different types of requests(e.g. ProjectApplicationRequest,OfficerApplicationRequest etc). All of these subclasses are interchangeable with the Request interface, ensuring they can be used wherever a Request object is expected while maintaining correct behavior. When determining the type of a request, different request instances can invoke the getRequestType() method. Thanks to polymorphism, the method implementation specific to each subclass is called, returning the appropriate request type. This demonstrates effective use of polymorphism to achieve flexibility and maintainability in request handling.

3.3.4 Interface Segregation Principle(ISP)

The interface segregation principle states that we should use many smaller specific interfaces rather than 1 general interface. We should avoid creating fat interfaces that implement various

different methods. This ensures that the subclasses do not implement methods that they do not need.

In our project, we split the ‘Model’ into User and Request interfaces so that the different entities can implement the interfaces respectively. For example, Request has a requestType that is different from that of the UserType. By splitting the 2 interfaces, we define a well separated set of abilities and functions that different Models can perform, making it easy to extend and modify the project by simply creating a new interface rather than handling various case statements in 1 bloated interface.

3.3.5 Dependency Inversion Principle(DIP)

DIP states that both high and low level modules should depend on abstractions. In the case of our project, we implement this by providing a User Interface. Instead of depending on the concrete classes of Applicant,Manager or Officer, we can call User.getID() from the User Interface to get the userID/NRIC. This allows us to use our UserFactory to easily add new users in the future, making it trivial to implement functions like registering different types of new Users in the system

4. Implementation(Java)

4.1 Tools Used:

- Java 23
- Build Tool: Maven
- IDE: Intellij/Eclipse
- Version control: GitHub

4.2 Sample Code Snippets(Appendix)

5. Testing

5.1 Test Strategy

We used a mix of manual and unit testing. We used Junit5 to test the classes after implementation to ensure that it was creating and saving the correct information.

As the user interface cannot be tested automatically, we used manual testing especially to test the exception handling

5.2 Test case table

In appendix

6. Documentation

6.1 Javadoc

All Java Docs are in the javadocs Directory

6.2 Developer Guide.

See ReadME in github repository

7. Reflection and Challenges

7.1 Improvements

However we could have been more thoughtful when designing the controller classes. In our current design the ApplicantManager,OfficerManager and ManagerManager go directly to their respective Repository for data. However, we can achieve a more polished design by allowing only 1 Controller class to directly access the Repository, reducing the level of Cohesion. Furthermore, we could have used a Displayable Interface that implements a unified viewing experience for each data type that is to be displayed. This could also include sorting features that would sort the data type before viewing. We did not complete this due to a lack of time.

7.4 Difficulty faced

Initially, we faced issues with identifying a good design that would allow us to follow SOLID principles. We overcame this by implementing MVC architecture, with additional design patterns like Factory that would allow us to separate application concerns. In addition, the use of Interfaces allowed us to better achieve this. Good planning is important before implementation and I have much more to learn

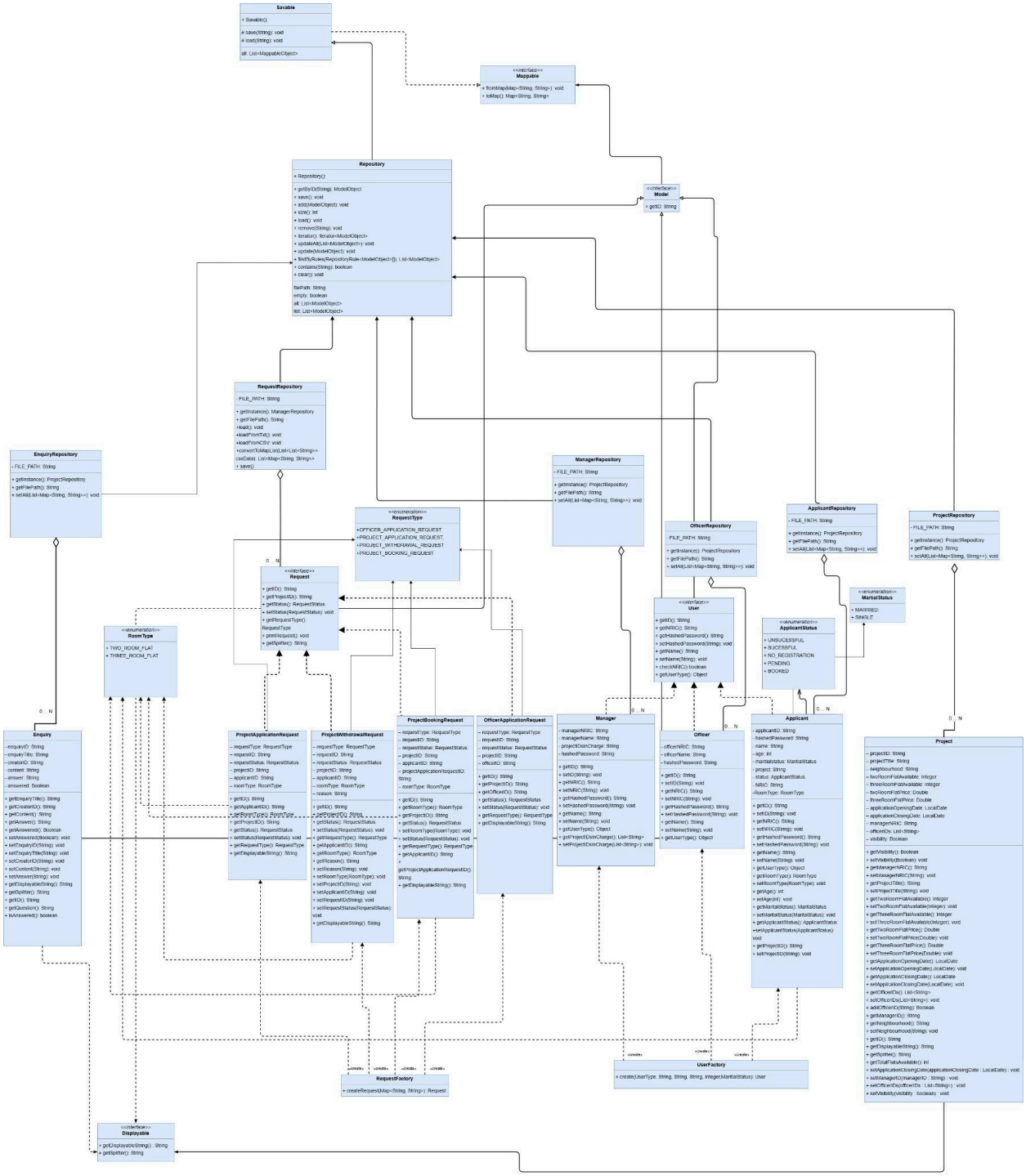
In addition, we also faced difficulty in ensuring data integrity. Initially, we realised that persistence would allow us to save time on manual testing. However, using the CSV files as a data store proved too much of a challenge and we used txt files instead. We then implemented the Repositories which proved successful in abstracting away data operations.

7.5 Knowledge Learnt

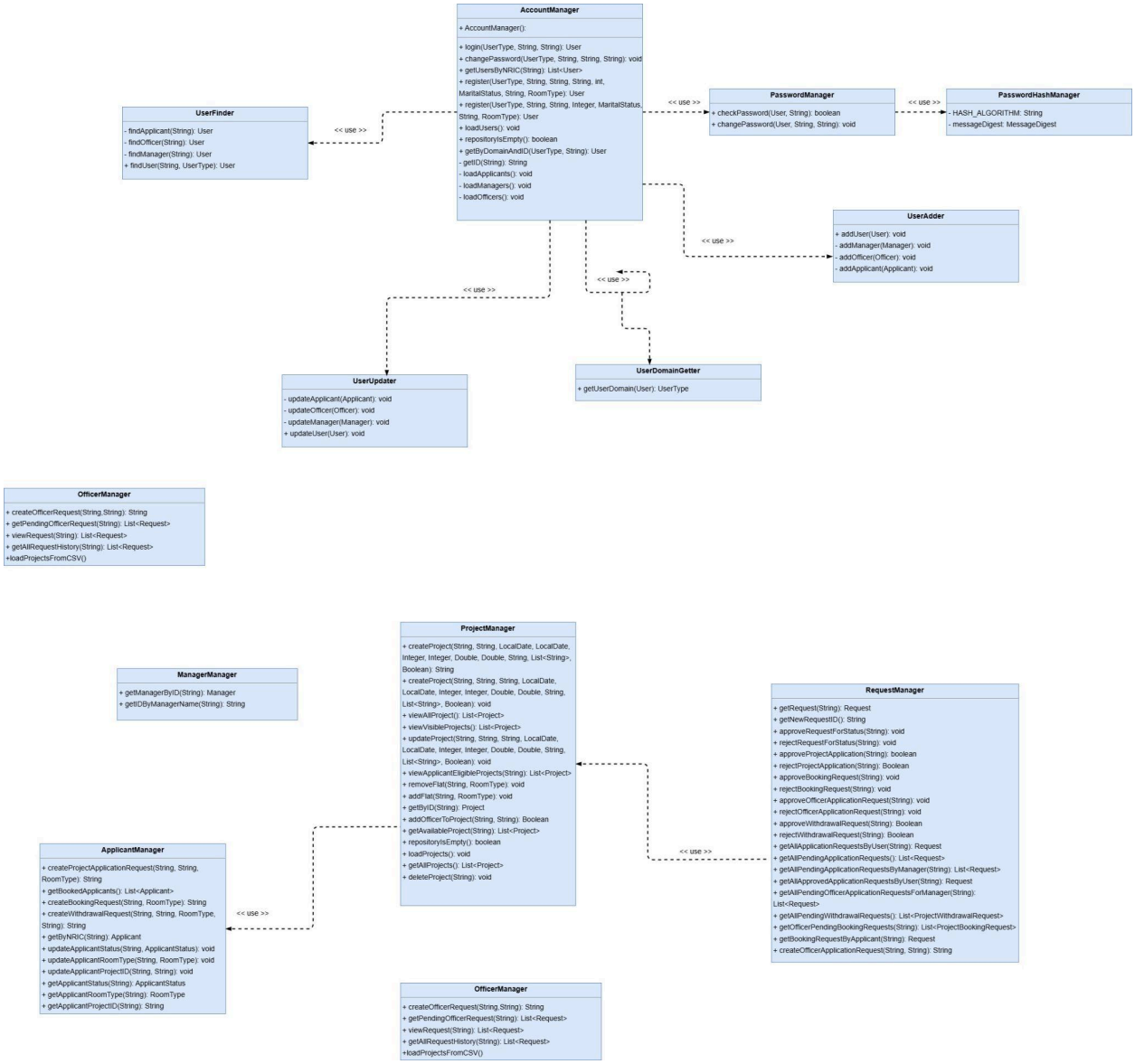
Through the development of this project, we gained a better understanding of OOP and SOLID principles along with a better appreciation of a system with high coupling and low cohesion as this made adding new features easy. A well structured architecture helps make developer experience better as it makes adding new features easy. Additionally, naming conventions are also important as it helps make things clear for developers. In the course of coding, I made many mistakes in calling similarly named functions, especially when I was tired. Using Domain-driven naming for classes like ProjectApplicantRequest and OfficerApplicationRequest helps improve clarity and ease of coding. Abstraction of the data layer helps to make common data manipulation tasks easy and allows the rest of the application to access and update files easily. This improves maintainability of the application.

8. Appendix(Images also in the UML directory of Github repository) IMAGES ONLY

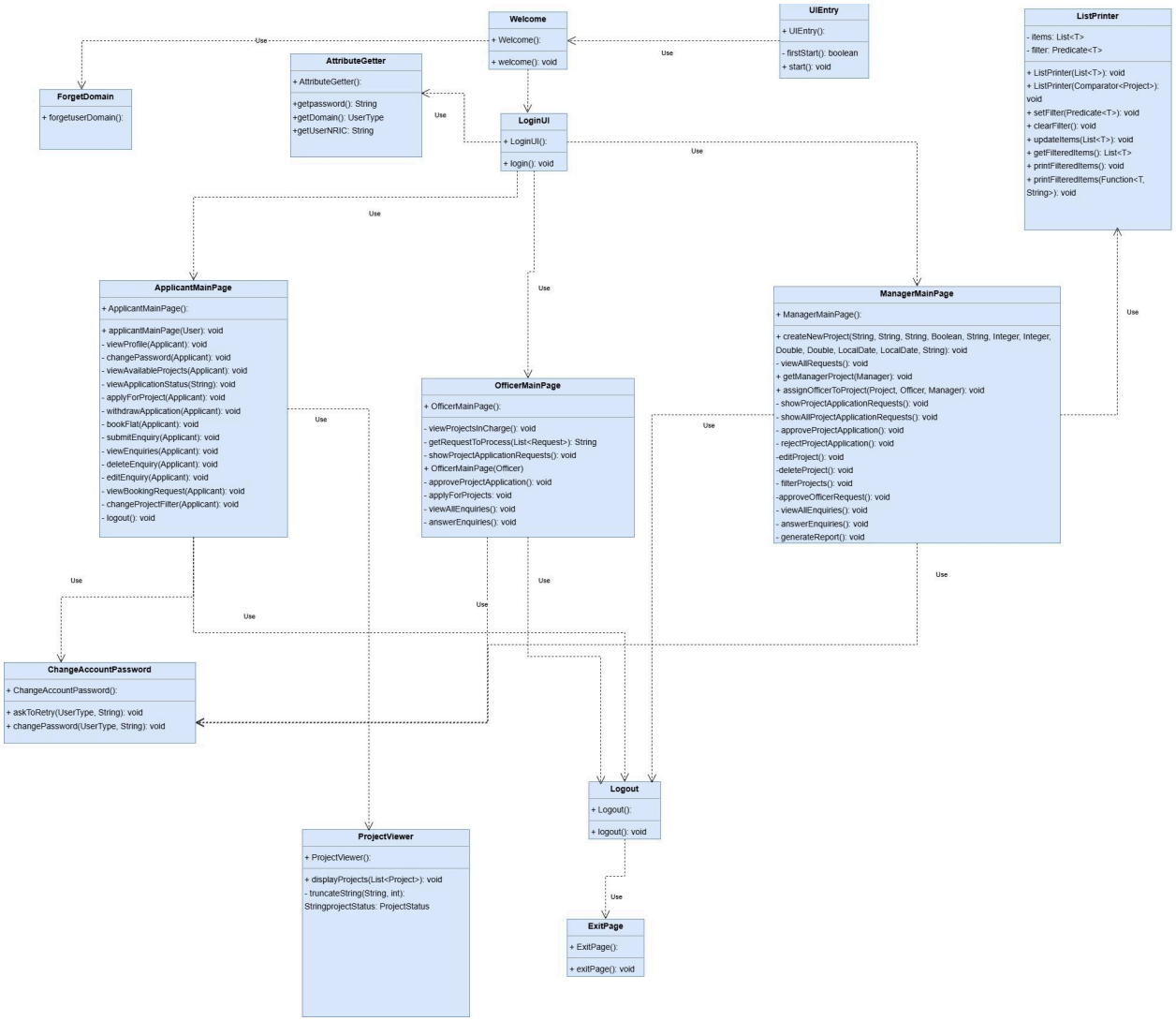
8.1 Entity Subdiagram


Savable
+ Savable()
save(String): void
load(String): void
all: List<MappableObject>
<<interface>>
Mappable
+ fromMap(Map<String, String>): void
+ toMap(): Map<String, String>
Repository
+ Repository()
+ getByID(String): ModelObject
+ save(): void
+ add(ModelObject): void
+ size(): int
+ load(): void
+ remove(String): void
+ iterator(): Iterator<ModelObject>
+ updateAll(List<ModelObject>): void
+ update(ModelObject): void
+ findBy Rules(Repository Rule<ModelObject>[]): List<ModelObject>
+ contains(String): boolean
+ clear(): void
filePath: String
empty: boolean
all: List<ModelObject>
list: List<ModelObject>
<<interface>>
Model
+ getID: String
RequestRepository
- FILE_PATH: String
+ getInstance(): ManagerRepository
+ getFilePath(): String
+load(): void
+loadFromFile(): void
+loadFromCSV: void
+convertToMap(List):List<List<String>>
csvData: List<Map<String, String>>
+ save()
EnquiryRepository
- FILE_PATH: String
+ getInstance(): ProjectRepository
+ getFilePath(): String
+ setAll(List<Map<String, String>>): void
ManagerRepository
- FILE_PATH: String
+ getInstance(): ProjectRepository
+ getFilePath(): String
+ setAll(List<Map<String, String>>): void
<<enumeration>>
RequestType
+OFFICER_APPLICATION_REQUEST
+PROJECT_APPLICATION_REQUEST
+PROJECT_WITHDRAWAL_REQUEST
+PROJECT_BOOKING_REQUEST
OfficerRepository
FILE_PATH: String
+ getInstance(): ProjectRepository
+ getFilePath(): String
+ setAll(List<Map<String, String>>): void
ApplicantRepository
- FILE_PATH: String
+ getInstance(): ProjectRepository
+ getFilePath(): String
+ setAll(List<Map<String, String>>): void
ProjectRepository
- FILE_PATH: String
+ getInstance(): ProjectRepository
+ getFilePath(): String
+ setAll(List<Map<String, String>>): void
<<enumeration>>
MaritalStatus
+ MARRIED
+ SINGLE
0...N
<<interface>>
Request
+ getID(): String
+ getProjectID(): String
+ getStatus(): RequestStatus
+ setStatus(RequestStatus): void
+ getRequestType()
RequestType
+ printRequest(): void
+ getSplitter(): String
<<enumeration>>
Room Type
+ TWO_ROOM_FLAT
+ THREE_ROOM_FLAT
<<interface>>
User
+ getID(): String
+ getNRIC(): String
+ getHashedPassword(): String
+ setHashedPassword(String): void
+ getName(): String
+ setName(String): void
+ checkNRIC(): boolean
+ getUserType(): Object
<<valuetype>>
ApplicantStatus
+ UNSUCESSFUL
+ SUCESSFUL
+ NO_REGISTRATION
+ PENDING
+ BOOKED
Enquiry
- enquiryID: String
- enquiryTitle: String
- creatorID: String
- content: String
- answer: String
- answered: Boolean
+ getEnquiryTitle(): String
+ getCreatorID(): String
+ getContent(): String
+ getAnswer(): String
+ getAnswered(): Boolean
+ setAnswered(Boolean): void
+ setEnquiryID(String): void
+ setEnquiryTitle(String): void
+ setCreatorID(String): void
+ setContent(String): void
+ setAnswer(String): void
+ getDisplayable String(): String
+ getSplitter(): String
+ getID(): String
+ getQuestion(): String
+ isAnswered(): boolean
ProjectApplicationRequest
– requestType: RequestType
– requestID: String
– requestStatus: RequestStatus
– projectID: String
– applicantID: String
– roomType: RoomType
+ getID(): String
+ getApplicantID(): String
+ getProjectID(): String
+ getRequestType(): RequestType
+ getStatus(): RequestStatus
+ setStatus(RequestStatus): void
+ getDisplayable String(): String
ProjectWithdrawalRequest
– requestType: RequestType
– requestID: String
– requestStatus: RequestStatus
– projectID: String
– applicantID: String
– roomType: RoomType
– reason: String
+ getID(): String
+ getRoomType(): RoomType
+ getReason(): String
+ setReason(String): void
+ setRoomType(RoomType): void
+ setProjectID(String): void
+ setApplicantID(String): void
+ setRequestID(String): void
+ setRequestStatus(RequestStatus): void
+ getDisplayable String(): String
ProjectBookingRequest
– requestType: RequestType
– requestID: String
– requestStatus: RequestStatus
– projectID: String
– applicantID: String
– projectApplicationRequestID: String
– roomType: RoomType
+ getID(): String
+ getRoomType(): RoomType
+ getProjectID(): String
+ getStatus(): RequestStatus
+ setRoomType(RoomType): void
+ setStatus(RequestStatus): void
+ getRequestType(): RequestType
+ getApplicantID(): String
+ getProjectApplicationRequestID(): String
+ getDisplayable String(): String
OfficerApplicationRequest
– requestType: RequestType
– requestID: String
– requestStatus: RequestStatus
– projectID: String
– officerID: String
+ getID(): String
+ getProjectID(): String
+ getOfficerID(): String
+ getStatus(): RequestStatus
+ setStatus(RequestStatus): void
+ getRequestType(): RequestType
+ getDisplayable String(): String
Manager
- managerNRIC: String
- managerName: String
- projectDateCharge: String
- hashedPassword: String
+ getID(): String
+ setID(String): void
+ getNRIC(): String
+ setNRIC(String): void
+ getHashedPassword(): String
+ setHashedPassword(String): void
+ getName(): String
+ setName(String): void
+ getUserType(): Object
+ getProjectDateCharge(): List<String>
+ setProjectDateCharge(List<String>): void
Officer
- officerNRIC: String
- officerName: String
- hashedPassword: String
+ getID(): String
+ setID(String): void
+ getNRIC(): String
+ setNRIC(String): void
+ getHashedPassword(): String
+ setHashedPassword(String): void
+ getName(): String
+ setName(String): void
+ getUserType(): Object
Applicant
- applicantID: String
- hashedPassword: String
- name: String
- age: int
- maritalstatus: MaritalStatus
- project: String
- status: ApplicantStatus
- NRIC: String
-Room Type: RoomType
+ getID(): String
+ setID(String): void
+ getNRIC(): String
+ setNRIC(String): void
+ getHashedPassword(): String
+ setHashedPassword(String): void
+ getName(): String
+ setName(String): void
+ getUserType(): Object
+ getRoomType(): RoomType
+ setRoomType(RoomType): void
+ getAge(): int
+ setAge(int): void
+ getMaritalStatus(): MaritalStatus
+ setMaritalStatus(MaritalStatus): void
+ getApplicantStatus(): ApplicantStatus
+setApplicantStatus(ApplicantStatus): void
+ getProject(): String
+ setProject(String): void
Project
– projectID: String
– projectTitle: String
– neighbourhood: String
– twoRoomFlatAvailable: Integer
– threeRoomFlatAvailable: Integer
– twoRoomFlatPrice: Double
– threeRoomFlatPrice: Double
– applicationOpeningDate: LocalDate
– applicationClosingDate: LocalDate
– managerNRIC: String
– officerIDs: List<String>
– visibility: Boolean
+ getVisibility(): Boolean
+ setVisibility(Boolean): void
+ getManagerNRIC(): String
+ setManagerNRIC(String): void
+ getProjectTitle(): String
+ setProjectTitle(String): void
+ getTwoRoomFlatAvailable(): Integer
+ setTwoRoomFlatAvailable(Integer): void
+ getThreeRoomFlatAvailable(): Integer
+ setThreeRoomFlatAvailable(Integer): void
+ getTwoRoomFlatPrice(): Double
+ setTwoRoomFlatPrice(Double): void
+ getThreeRoomFlatPrice(): Double
+ setThreeRoomFlatPrice(Double): void
+ getApplicationOpeningDate(): LocalDate
+ setApplicationOpeningDate(LocalDate): void
+ getApplicationClosingDate(): LocalDate
+ setApplicationClosingDate(LocalDate): void
+ getOfficerIDs(): List<String>
+ setOfficerIDs(List<String>): void
+ addOfficerID(String): Boolean
+ getManagerID(): String
+ getNeighbourhood(): String
+ setNeighbourhood(String): void
+ getID(): String
+ getDisplayable String(): String
+ getSplitter(): String
+ getTotalFlatsAvailable(): int
+ setApplicationClosingDate(applicationClosingDate : LocalDate) : void
+ setManagerID(managerID : String) : void
+ setOfficerIDs(officerIDs : List<String>) : void
+ setVisibility(visibility : boolean) : void
<<create>>
RequestFactory
+ createRequest(Map<String, String>): Request
<<create>>
UserFactory
+ create(UserType, String, String, String, Integer,MaritalStatus): User
<<interface>>
Displayable
+ getDisplayable String(): String
+ getSplitter(): String

8.2 Controller subdiagram



8.3 Boundary Subdiagram



Draw.io Link for UML: [main.drawio]

8.2 Full Test Case Table

No.	Test Case	Expected Behavior	Failure Indicators	Pass
1	Valid User Login	User should be able to access their dashboard based on their roles	User cannot log in or receives incorrect error messages	Pass
2	Invalid NRIC Format	User receives a notification about incorrect NRIC format	User is allowed to log in with an invalid NRIC	Pass
3	Incorrect Password	System should deny access and alert the user to incorrect password	User logs in successfully with a wrong password	Pass
4	Password Change Functionality	System updates password, prompt re-login and allows login with new credentials	System does not update the password or denies access with the new password	Pass
5	Project Visibility Based on Group	Projects are visible to users based on their age, marital status and the visibility setting	Users see projects not relevant to their group or when visibility is off	Pass
6	Project Application	Users can only apply for projects relevant	Users can apply for projects not relevant to their group or when visibility is off	Pass

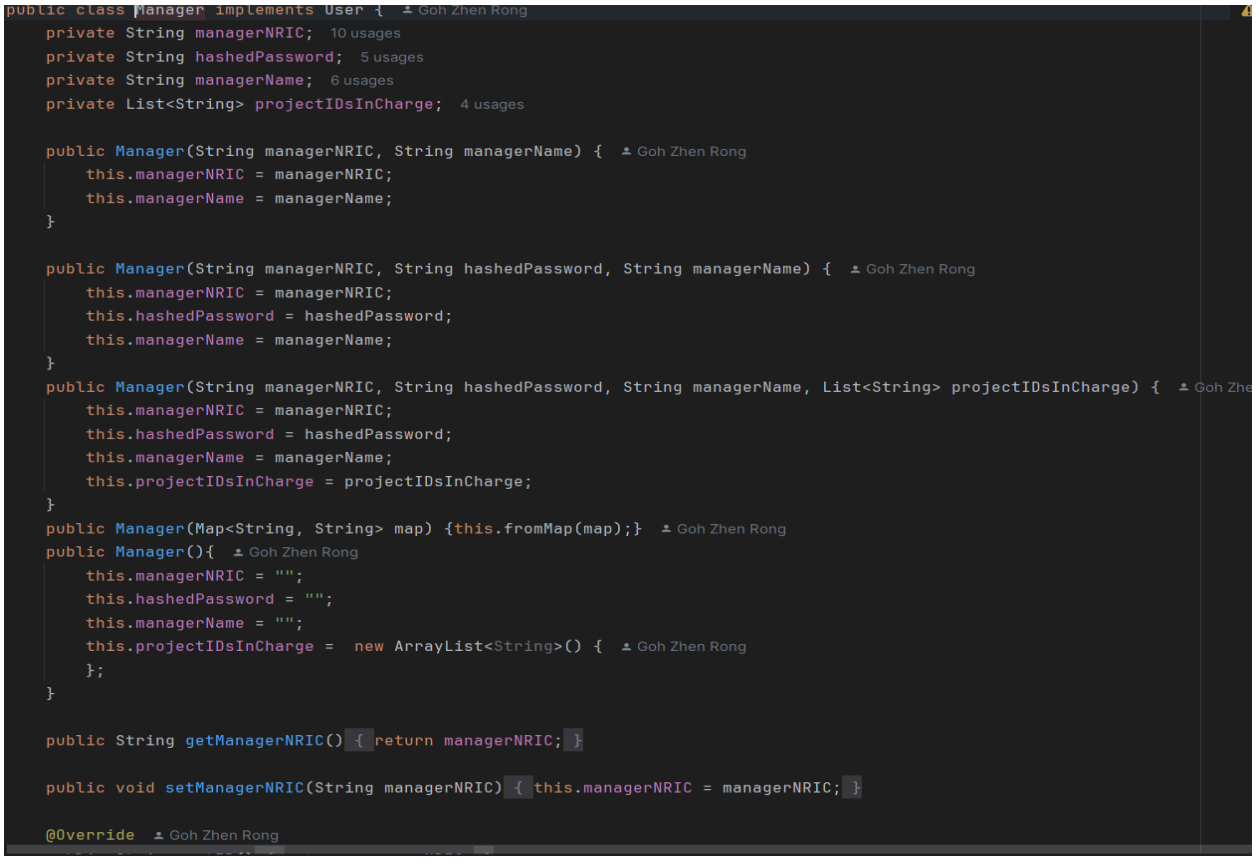
		to their group or when visibility is off		
7	View App Status (Visibility Off)	Applicants continue to have access to their application details regardless of project visibility.	Application details become inaccessible once visibility is off.	Pass
8	Single Flat Booking	System allows booking one flat and restricts further bookings	Applicant is able to book more than one flat	Pass
9	Applicant Enquiries Management	Enquiries can be successfully submitted, displayed, modified, and removed.	Enquiries cannot be submitted, edited, or deleted; or do not display correctly.	Pass
10	HDB Officer Registration Eligibility	System allows registration only under compliant conditions	System allows registration while the officer is an applicant or registered for another project...	Pass
11	HDB Officer Registration Status	Officers can view pending or approved status updates on their profiles.	Status updates are not visible or incorrect	Pass
12	Project Detail	Officers can always access full project	Project details are inaccessible when visibility is toggled off	Pass

	Access (HDB Officer)	details, even when visibility is turned off.		
13	Restriction on Editing (HDB Officer)	Edit functionality is disabled or absent for HDB Officers.	Officers are able to make changes to project details	Pass
14	Response to Project Enquiries	Officers & Managers can access and respond to enquiries efficiently.	Officers & Managers cannot see enquiries, or their responses are not recorded.	Pass
15	Flat Selection & Booking Mgmt	Officers retrieve the correct application, update flat availability accurately, and correctly log booking details...	Incorrect retrieval or updates, or failure to reflect booking details accurately.	Pass
16	Receipt Generation	Accurate and complete receipts are generated for each successful booking	Receipts are incomplete, inaccurate, or fail to generate	Pass
17	Create, Edit, Delete BTO Projects	Managers should be able to add new projects, modify existing project details, and remove	Inability to create, edit, or delete projects or errors during these operations.	Pass

		projects from the system		
18	Single Project Mgmt per Period	System prevents assignment of more than one project to a manager within the same application dates.	Manager is able to handle multiple projects simultaneously during the same period.	Pass
19	Toggle Project Visibility	Changes in visibility should be reflected accurately in the project list visible to applicants	Visibility settings do not update or do not affect the project listing as expected	Pass
20	View All & Filtered Projects	Managers should see all projects and be able to apply filters to narrow down to their own projects.	Inability to view all projects or incorrect filtering results	Pass
21	Manage HDB Officer Registratio ns	Managers handle officer registrations effectively, with system updates reflecting changes accurately.	Mismanagement of registrations or slot counts do not update properly.	Pass
22	Approve/R eject BTO Apps/With drawals	Approvals and rejections are processed correctly, with system updates	Incorrect or failed processing of applications or withdrawals	Pass

		to reflect the decision		
23	Generate and Filter Reports	Accurate report generation with options to filter by various categories.	Reports are inaccurate, incomplete, or filtering does not work as expected	Pass
24	Data Persistence	Data is persisted and saved after every run	Data Loss	Pass

1. Encapsulation



```
public class Manager implements User {  Goh Zhen Rong
    private String managerNRIC;  10 usages
    private String hashedPassword;  5 usages
    private String managerName;  6 usages
    private List<String> projectIDsInCharge;  4 usages

    public Manager(String managerNRIC, String managerName) {  Goh Zhen Rong
        this.managerNRIC = managerNRIC;
        this.managerName = managerName;
    }

    public Manager(String managerNRIC, String hashedPassword, String managerName) {  Goh Zhen Rong
        this.managerNRIC = managerNRIC;
        this.hashedPassword = hashedPassword;
        this.managerName = managerName;
    }
    public Manager(String managerNRIC, String hashedPassword, String managerName, List<String> projectIDsInCharge) {  Goh Zhe
        this.managerNRIC = managerNRIC;
        this.hashedPassword = hashedPassword;
        this.managerName = managerName;
        this.projectIDsInCharge = projectIDsInCharge;
    }
    public Manager(Map<String, String> map) {this.fromMap(map);}  Goh Zhen Rong
    public Manager(){  Goh Zhen Rong
        this.managerNRIC = "";
        this.hashedPassword = "";
        this.managerName = "";
        this.projectIDsInCharge =  new ArrayList<String>() {  Goh Zhen Rong
        };
    }

    public String getManagerNRIC() { return managerNRIC; }

    public void setManagerNRIC(String managerNRIC) { this.managerNRIC = managerNRIC; }

    @Override  Goh Zhen Rong
```

We set the Manager attributes as private and expose them using public getter and setter

2. Inheritance

```java
* The StudentRepository class is a repository for managing the persistence of student objects
* through file I/O operations.
* It extends the Repository class, which provides basic CRUD operations for the repository.
*/
public class ApplicantRepository extends Repository<Applicant> {  24 usages  Goh Zhen Rong

    /**
     * The path of the repository file.
     */
    private static final String FILE_PATH = "/data/user/ApplicantList.txt";  1 usage

    /**
     * Constructor for the StudentRepository class.
     */
    ApplicantRepository() {  1 usage  Goh Zhen Rong
        super();
        load();
    }

    /**
     * Gets a new instance of the StudentRepository class.
     *
     * @return a new instance of the StudentRepository class
     */
    public static ApplicantRepository getInstance() { return new ApplicantRepository(); }

    /**
     * Gets the path of the repository file.
     *
     * @return the path of the repository file
     */
    @Override  7 usages  Goh Zhen Rong
    public String getFilePath() { return RESOURCE_LOCATION + FILE_PATH; }
```

ApplicantRepository Inherits from Repository Superclass

3. Polymorphism

```
ublic class UserFactory {  1 usage   Goh Zhen Rong
    /**
     * Creates a new User object based on the given parameters.
     *
     * @param userType      The type of user to be created (applicant, officer, or manager).
     * @param userNRIC          The user's NRIC/ID.
     * @param password      The user's password in plaintext.
     * @param name          The user's name.
     * @param age           The user's age.
     * @param maritalStatus The user's marital status.
     * @return              A new User object of the specified type.
     */
    public static User create(UserType userType, String userNRIC, String password, String name,   Goh Zhen Rong
                              Integer age, MaritalStatus maritalStatus) {
        String hashedPassword = PasswordHashManager.hashPassword(password);
        return switch (userType) {
            case APPLICANT -> new Applicant(name, userNRIC, age, maritalStatus, hashedPassword);
            case OFFICER -> new Officer(userNRIC, name, hashedPassword);
            case MANAGER -> new Manager(userNRIC,hashedPassword,name);
        };
    }
```

In this User Factory Class, we Create Objects of User Type, where Applicant,Officer and Manager implement the same set of methods that all Users need like getNRIC and gethashedPassword. Another example is the how the Repositories callbuse the same add method, but the actual logic doing the adding is different

4. Interface

```
package model.user;

import controller.account.password.PasswordHashManager;
import model.project.RoomType;
import utils.iocontrol.Mappable;
import utils.parameters.EmptyID;
import utils.parameters.NotNull;

import java.util.LinkedHashMap;
import java.util.Map;

public class Applicant implements User {   Goh Zhen Rong
    private String applicantNRIC;  8 usages
    private String hashedPassword;  4 usages
    private String applicantName;  6 usages
    private Integer applicantAge;  6 usages
    private MaritalStatus maritalStatus; // Use enum instead of String  6 usages
    private String applicantProjectID;  4 usages
    private ApplicantStatus applicantStatus;  6 usages
    private RoomType roomType;  // booked flat type, null if none  6 usages

    //for the default password
    public Applicant(String name, String NRIC, Integer age, MaritalStatus maritalStatus) {   Goh Zhen Rong
        this.applicantName = name;
        this.applicantNRIC = NRIC;
        this.applicantAge = age;
        this.maritalStatus = maritalStatus;// Assuming NRIC is unique and used as ID
        this.applicantStatus = ApplicantStatus.NO_REGISTRATION; // Default status
        this.roomType = RoomType.NONE; // Default status
    }
```

Applicant implements methods defined in User Interface

5. Error Handling

```java
public class LoginUI {  2 usages  Goh Zhen Rong
    /**
     * Displays a login page.
     *
     * @throws PageBackException if the user chooses to go back to the previous page.
     */
    public static void login() throws PageBackException {  3 usages  Goh Zhen Rong
        ChangePage.changePage();
        UserType domain = AttributeGetter.getDomain();
        String userNRIC = AttributeGetter.getUserNRIC();
        if (userNRIC.equals("")) {
            try {
                ForgetDomain.forgotUserDomain();
            } catch (PageBackException e) {
                login();
            }
        }
        String password = AttributeGetter.getPassword();
//          System.err.println("Logging in...");
//          System.err.println("Domain: " + domain);
//          System.err.println("User ID: " + userID);
//          System.err.println("Password: " + password);
//          new Scanner(System.in).nextLine();
        try {
            User user = AccountManager.login(domain, userNRIC, password);
            switch (domain) {
                case APPLICANT -> ApplicantMainPage.applicantMainPage(user);
                case OFFICER -> OfficerMainPage.officerMainPage(user);
                case MANAGER -> ManagerMainPage.managerMainPage(user);
                default -> throw new IllegalStateException("Unexpected value: " + domain);
            }
            return;
        } catch (PasswordIncorrectException e) {
            System.out.println("Password incorrect.");
        } catch (ModelNotFoundException e) {
            System.out.println("User not found.");
        }
```

Error Handling used in the Login page to handle incorrect or invalid input during login