SC2002

SCE2-GRP 5

Ng Zhi Wei U2420053E

Tan Jun Hao U2322364F

Timothy Louis Barus U2320344D

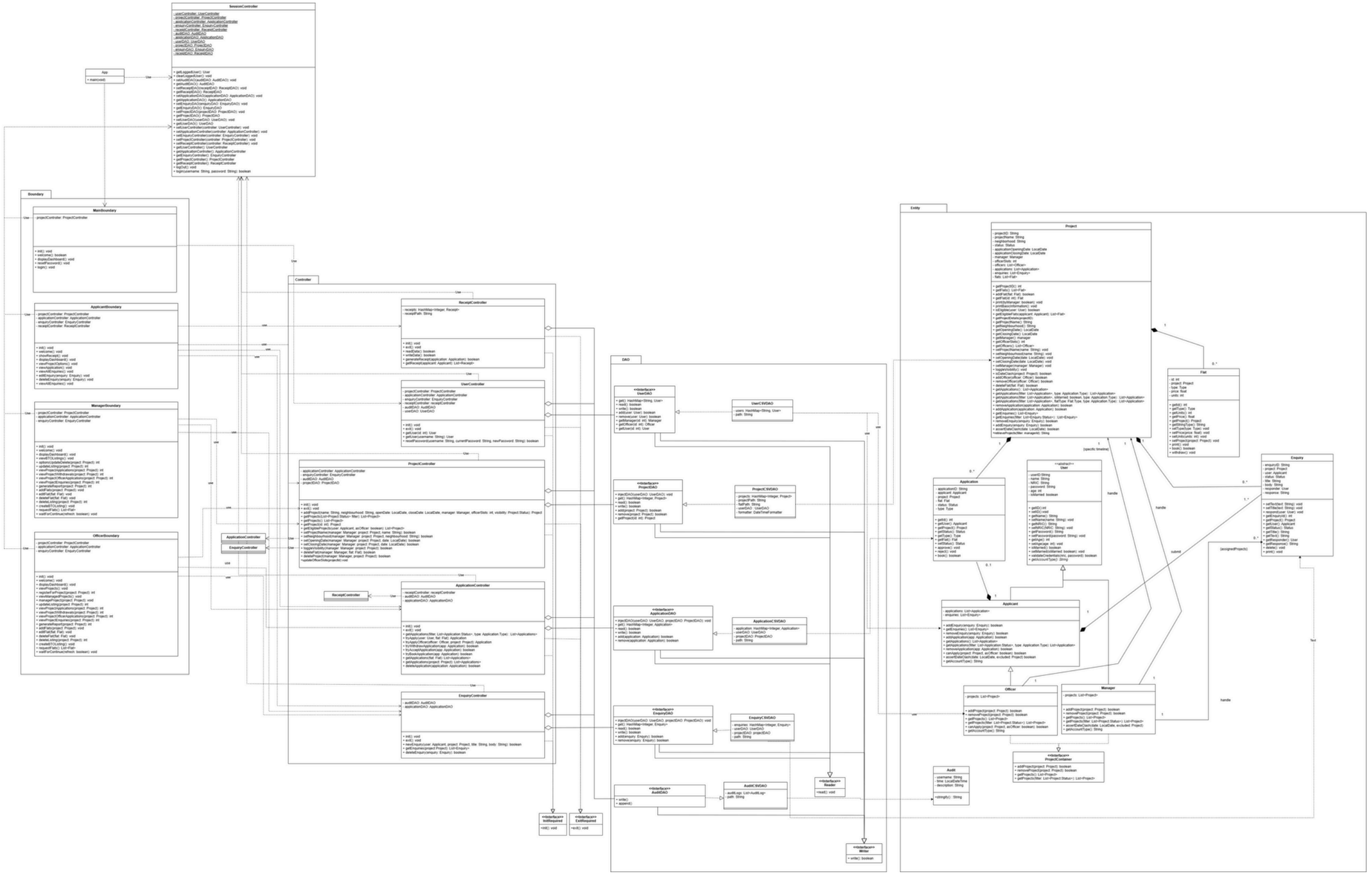
Sunkad Surabhi Sridhar U2423726E

www.reallygreatsite.com

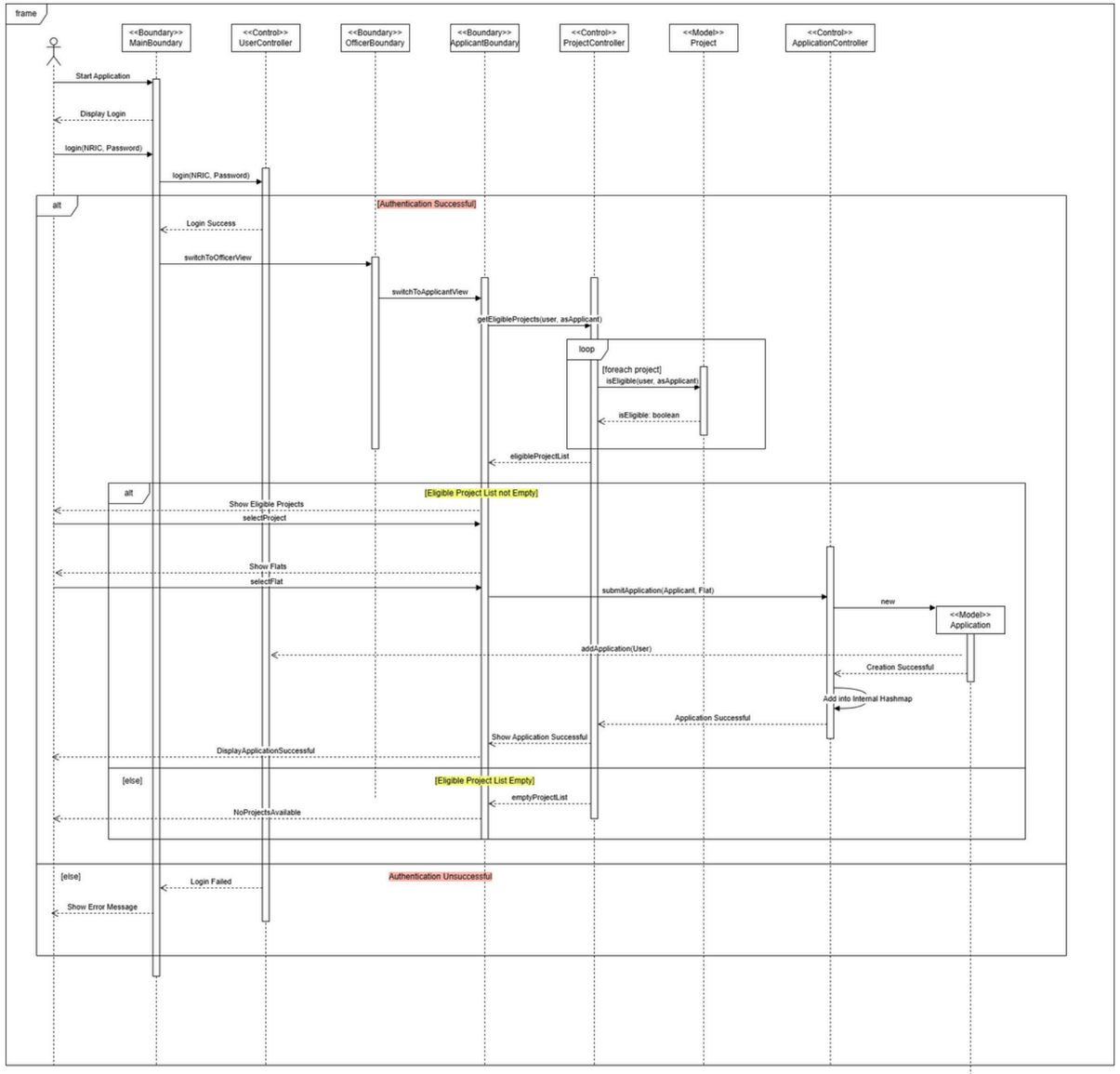
Assumptions

- All users are pre-registered in CSV files.
- Default password for all users is "password".
- Data in CSV files is well-formed and consistent.
- The system runs using Java 17.
- No use of databases, JSON, or XML as per assignment constraints.

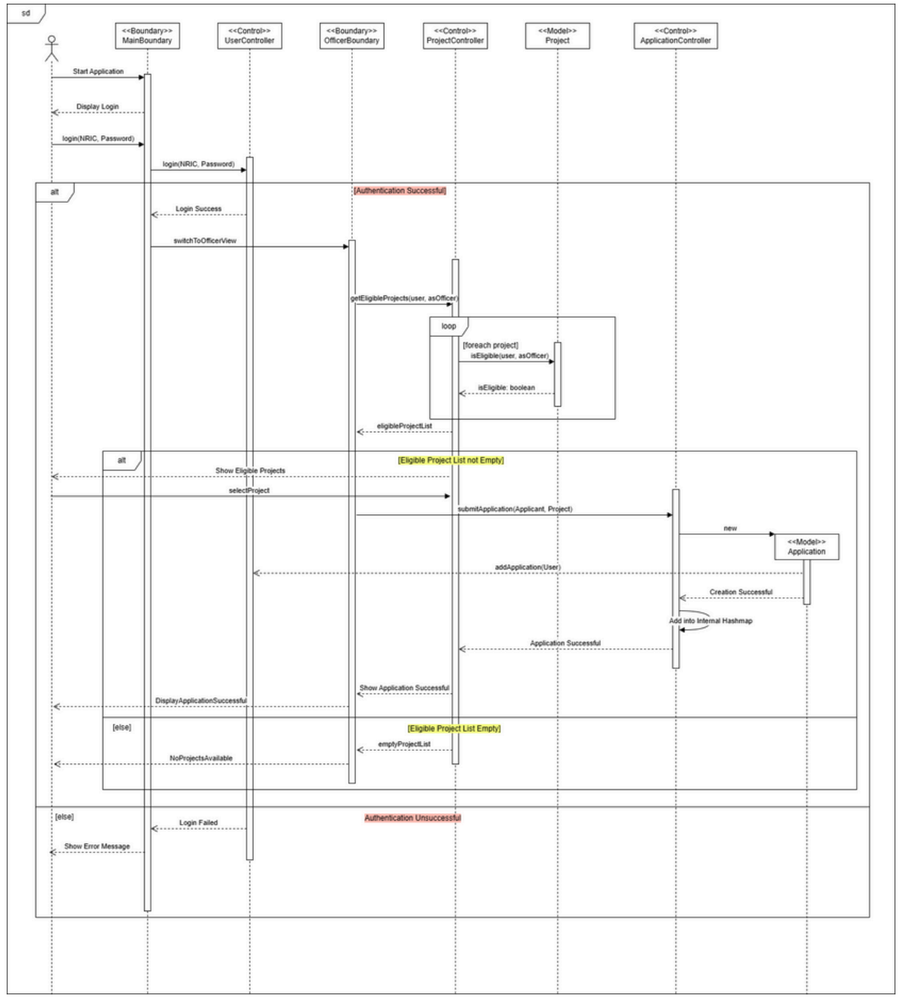
UML Diagram


SessionController
App
Boundary
MainBoundary
ApplicantBoundary
ManagerBoundary
OfficerBoundary
Controller
ReceptController
UserController
ProjectController
ApplicationController
EnquiryController
DAO
UserDAO
UserCSVDAO
ProjectDAO
ProjectCSVDAO
ApplicationDAO
ApplicationCSVDAO
EnquiryDAO
EnquiryCSVDAO
AuditDAO
AuditCSVDAO
Reader
Writer
Entity
Project
Flat
Enquiry
Application
User
Applicant
Officer
Manager
Audit
ProjectContainer

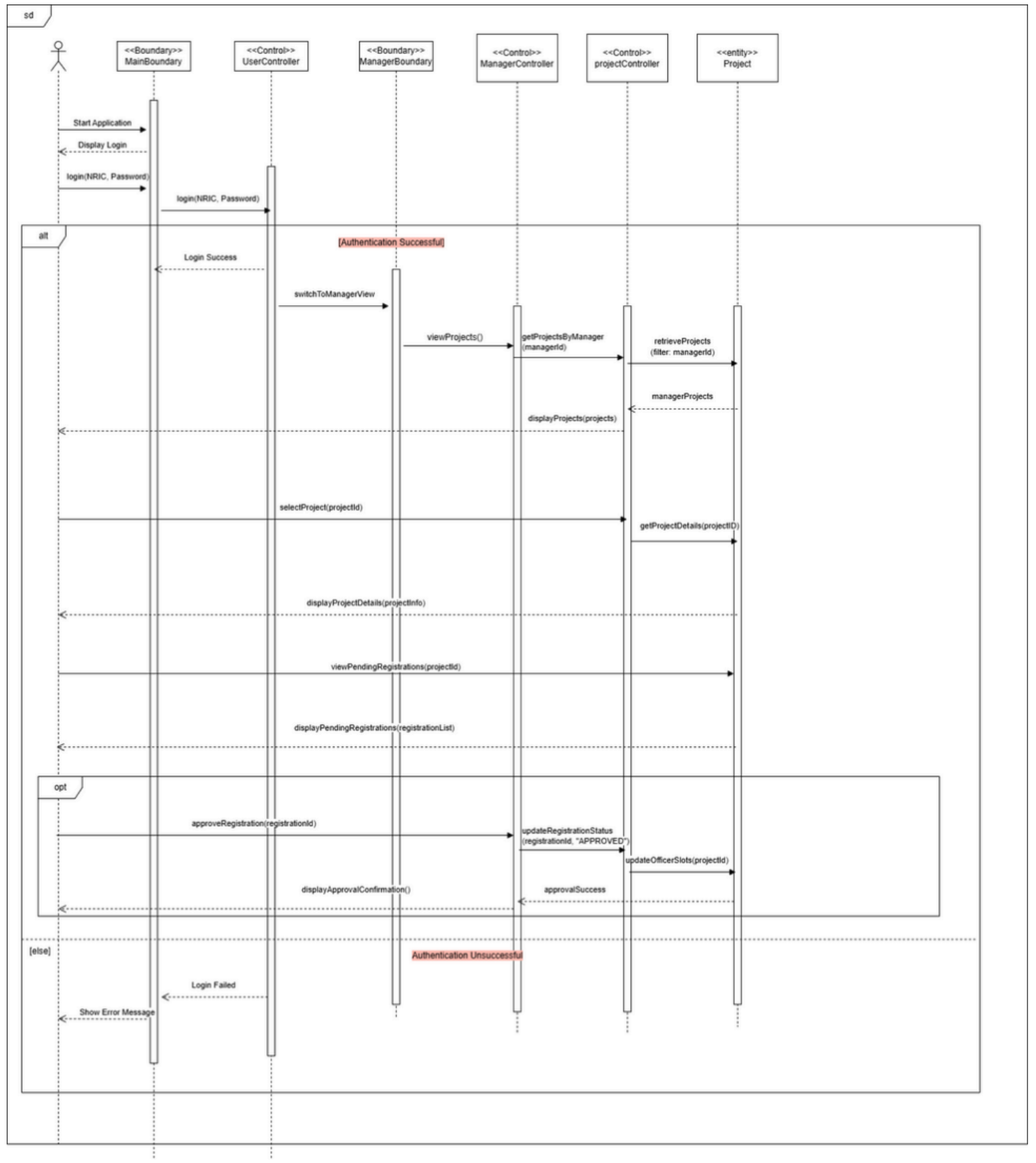
Sequence Diagram 1



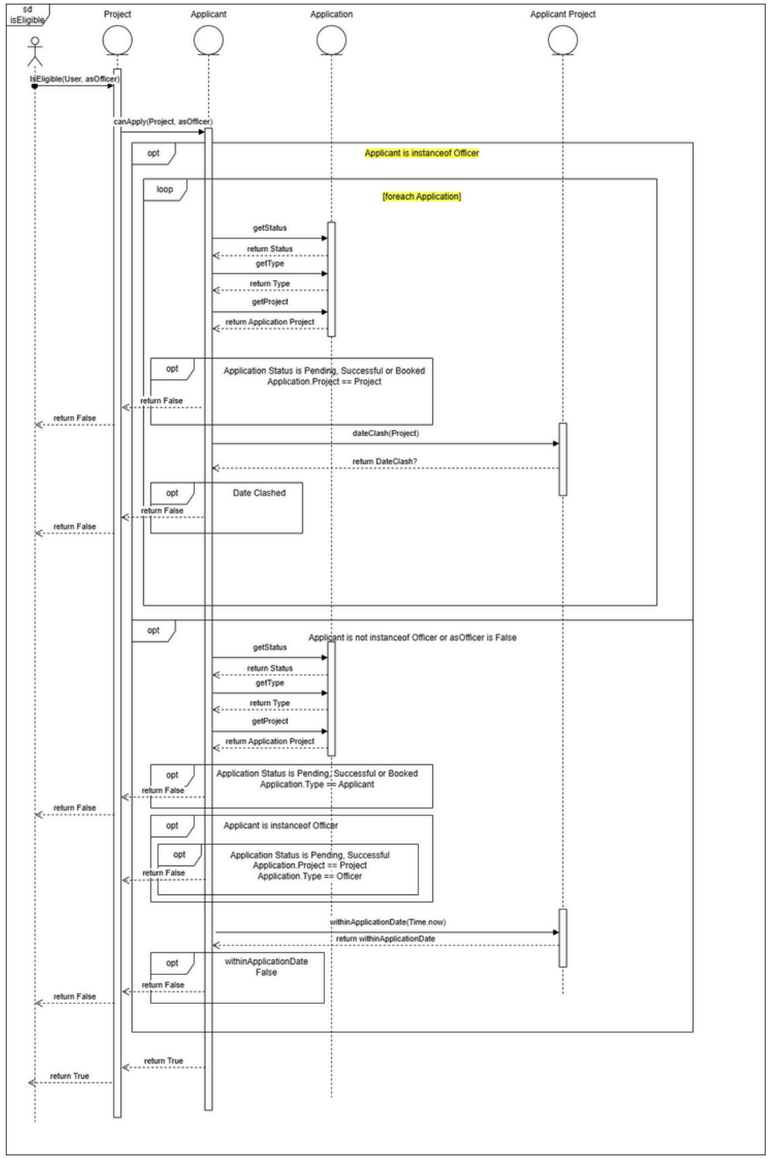
Sequence Diagram 2



Sequence Diagram 3



Sequence Diagram 4



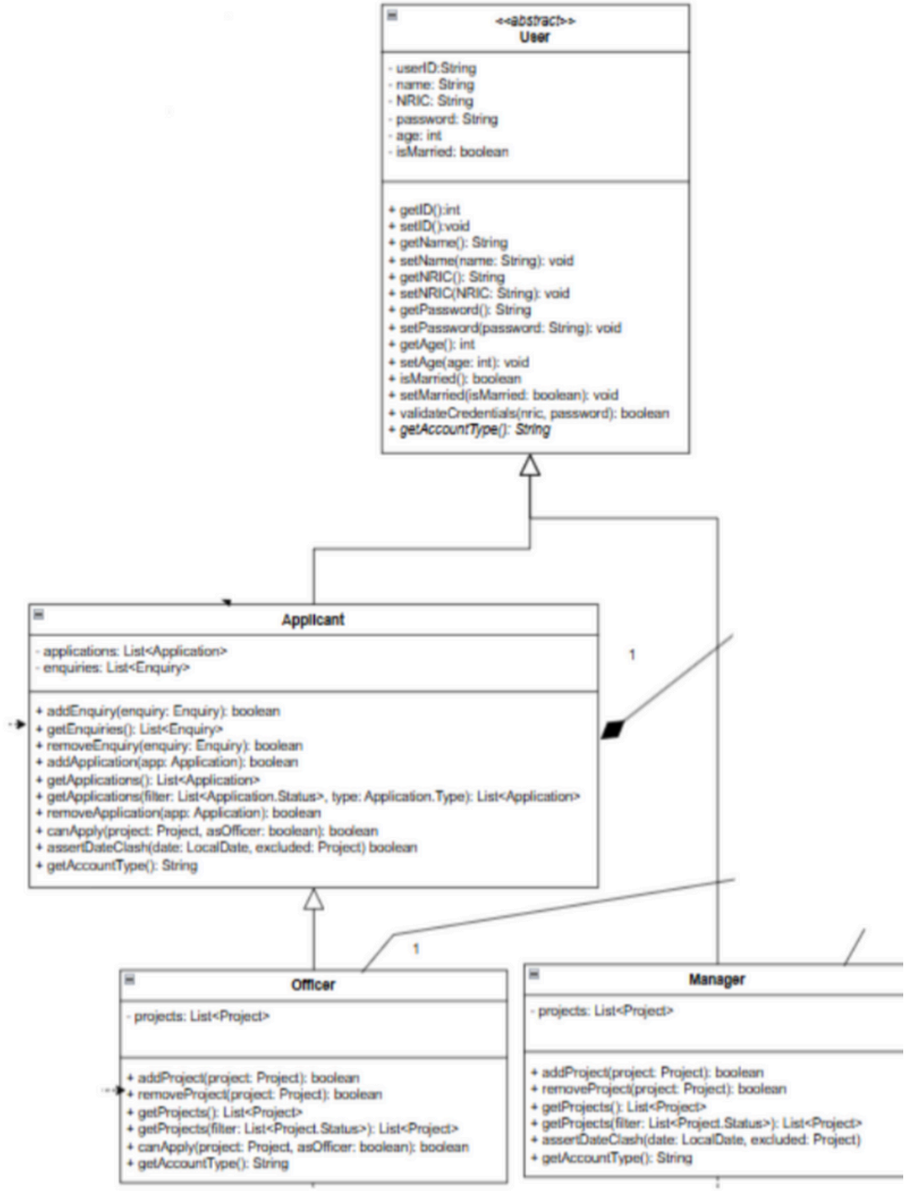
Use of OO Concepts

1. Abstraction

The system employs abstraction through the use of the User class becoming the overall frame for all the user categories.

It defines common attributes such as ID, name, NRIC, age, marital status, and password. To the extent that the user has all these attributes, the actions of each of the user roles (i.e., Applicants, Officers, and Managers) are abstracted into the subclasses.

This way, the complex interactions become easier since the software can handle all the users the same at the higher-level, the particular details being handled independently to each of the subclasses.



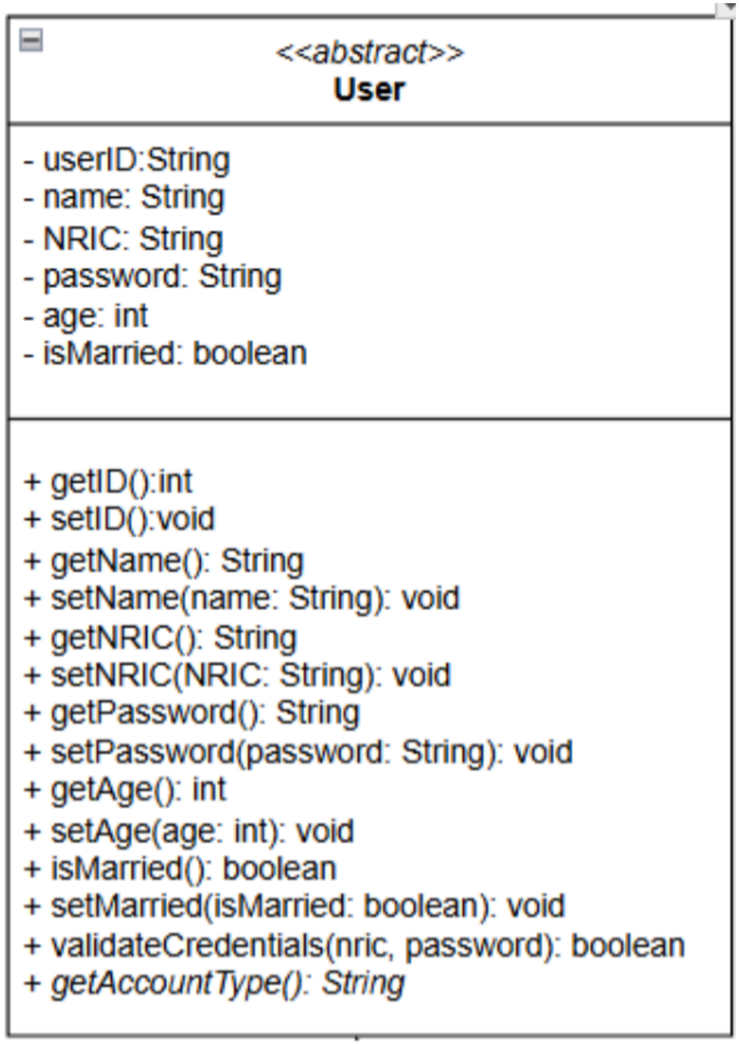
Use of OO Concepts

2. Encapsulation

The encapsulation process is strictly maintained all the way through the system. Each class stores information in the private members and only provides necessary functionality using the public members.

Sensitive information like passwords inside the User class is kept confidentially and accessed or modified using controlled functions like verifyPassword() or changePassword().

Similarly, internal members like unit numbers or status inside classes like Flat or Application never become accessible directly outside the class. This ensures data integrity along with controlled access.



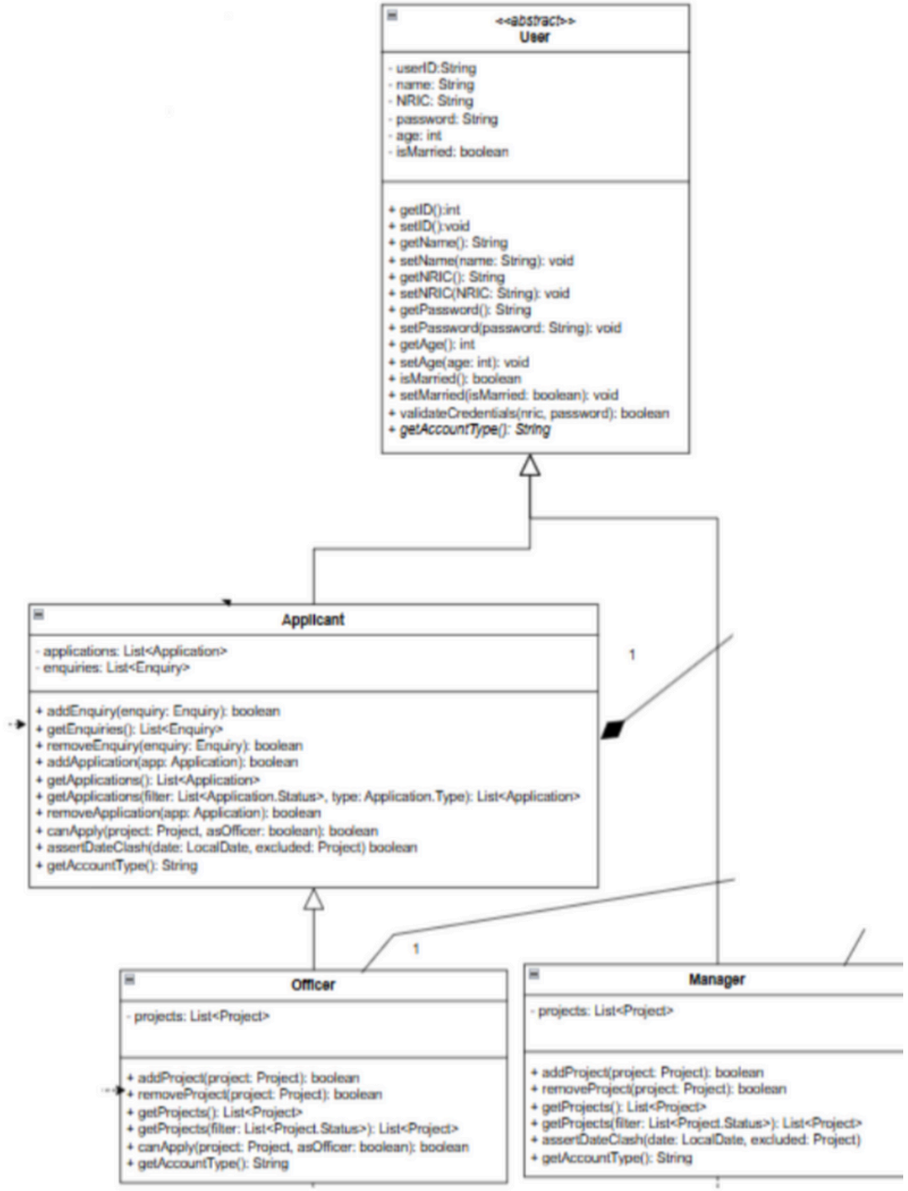
Use of OO Concepts

3. Inheritance

Inheritance is applied consistently throughout the design of the system. User is the base abstract class to the classes Applicant, Officer, and Manager.

It inherits necessary attributes like name, NRIC, and age, and behavior like password management, and extends new methods necessary for the specific jobs—Officer can handle projects whereas Applicant can book flats.

This application of inheritance prevents code duplication, promotes modular design, and properly reflects real world hierarchical relationships between the different user types of the BTO process.



Use of OO Concepts

4. Polymorphism

Polymorphism is achieved primarily through the processing of the user by the abstract User reference. During the time of logging in, the system holds him/her as a simple User object but at runtime can be an Applicant, an Officer, or one of any Manager type.

One such implemented instance of polymorphism would be the canApply() function in both Applicant and Officer, where Applicants canApply() and the Officer's canApply() function are different, where the Officer's canApply() function does extra checks to check if the officer is already a officer of a project he is applying to.

Applicant side

```
public boolean canApply(Project project, boolean asOfficer)
{
    if(project == null)
        return false;

    if(!project.isEligible(this)) //check age here.
    {
        return false;
    }

    //check if today's date within application period.
    LocalDate today = LocalDate.now();
    if (today.isBefore(project.getOpeningDate()) || today.isAfter(project.getClosingDate())) {
        // Today is NOT within the project period
        return false;
    }
```

Officer Side

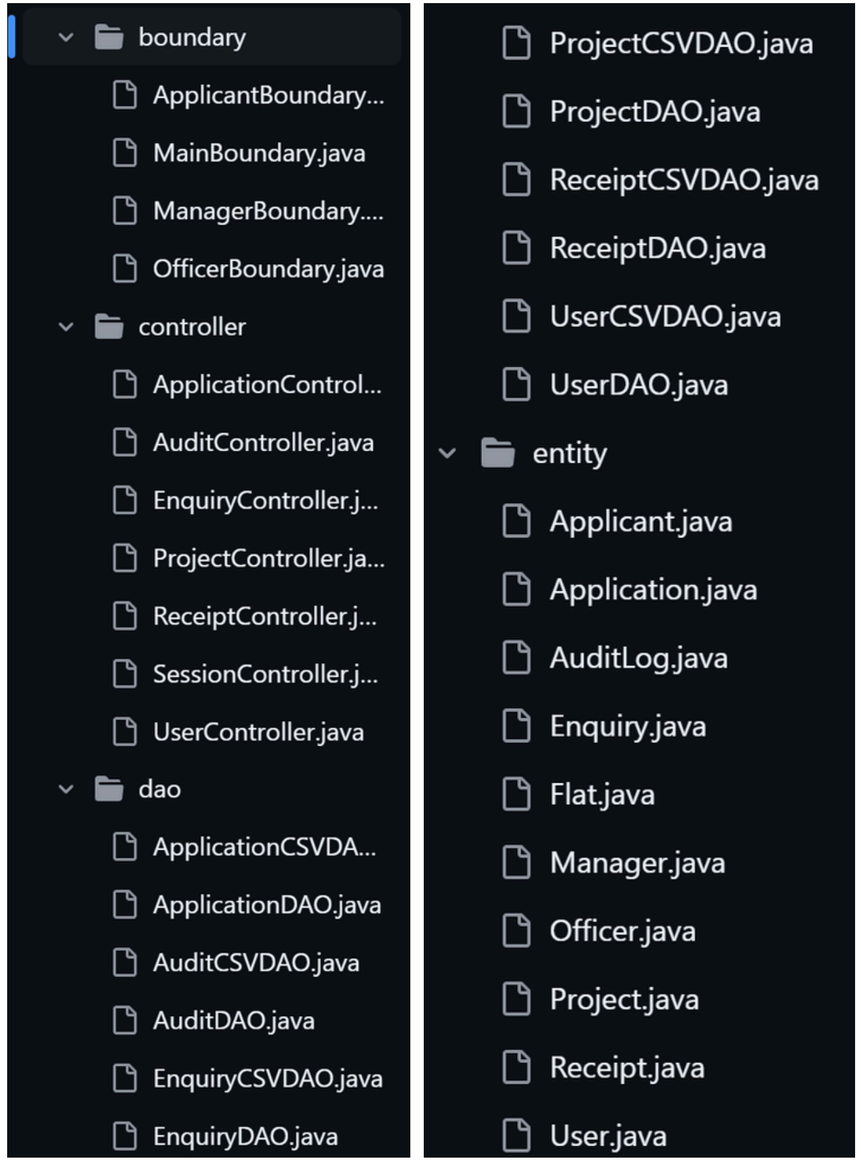
```
@Override
public boolean canApply(Project project, boolean asOfficer){
    //check against list of projects, then check against applicant side's applicants.

    //check if officer has already applied to this project as an applicant
    List<Application> filtered = getApplications(List.of(Application.Status.PENDING, Application.Status.SUCCESSFUL, Application.Sta
    for(Application application : filtered){ //this checks if officer has already applied for this as a project.
        if(application.getProject() == project){
            //System.out.println("CannotApplyProject AS APPLICANT" + project.getProjectName());
            return false;
        }
    }

    //check if officer applications date clash with any other applications as applicant
    for(Application application : filtered){
        if(!(application.getProject().assertDateClash(project.getOpeningDate()) && application.getProject().assertDateClash(project
            //System.out.println("CannotApplyProject DATE CLASH APPLICANT" + project.getProjectName());
            return false;
        }
```

SOLID design principles

Single Responsibility Principle (SRP)

When each class is designed, they are designed with only one responsibility in mind. This ensures that when a class is changed, it is only for the tasks related to that responsibility.

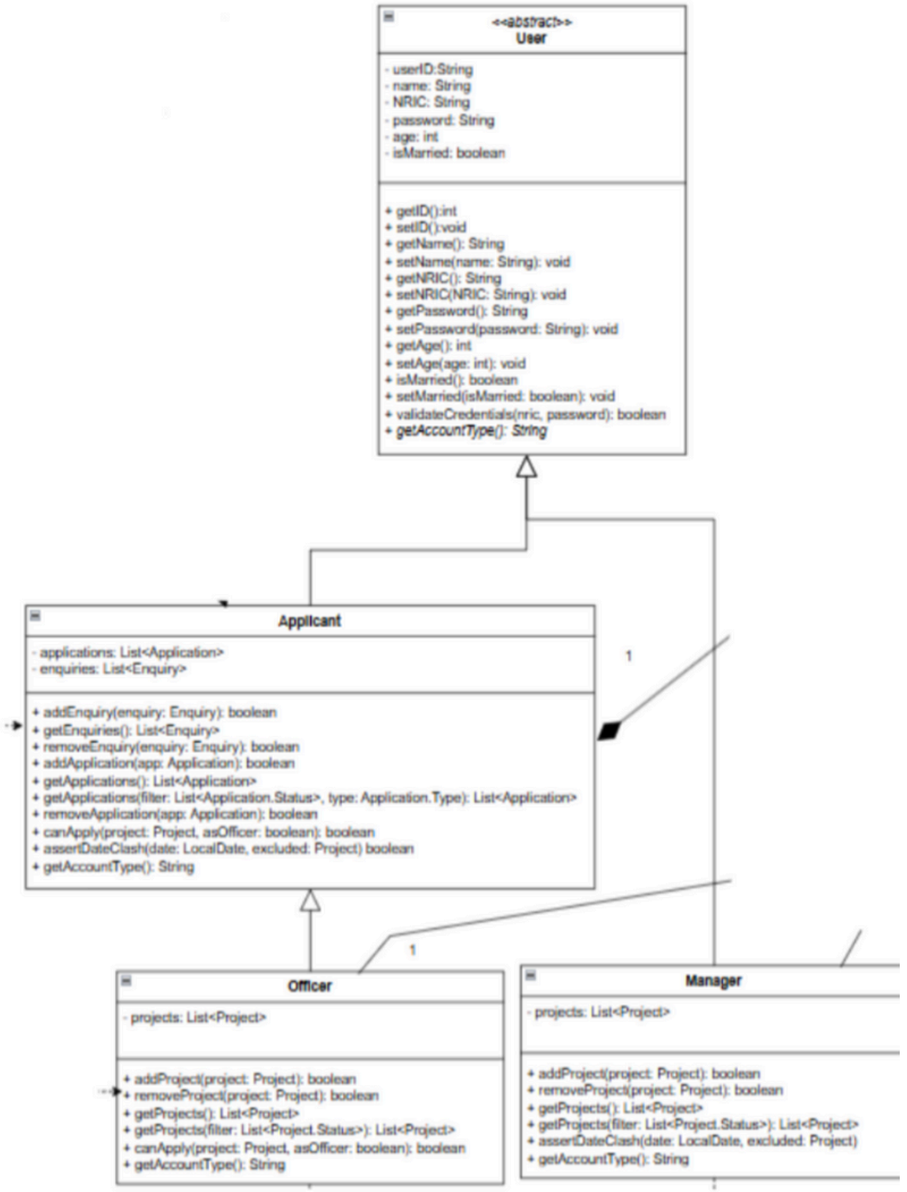
For example, ProjectController, ApplicationController, UserController is only responsible for the manipulation and business logic of their respective class, Project, Application, User. For instance, while Applicant class holds application and enquiry listings specific to the user, it doesn't get involved with application or withdrawal—it's taken care of as should-be in ApplicationController.



SOLID design principles

Open-Closed Principle (OCP)

This strategy is based on the principle that software components should be extensible without requiring changes to previous code. In the architecture, the User class is used as the base framework from which other user types can derive, such as Applicant, Officer, or Manager. This simplifies adding future enhancements; adding additional levels of authority or types of users is achievable through subclassing without having to alter base class. Adding a new type of user, such as a Housing Administrator, will have special behavior but still inherit from User without negating pre-established roles. It also makes it more maintainable as enhancements are added over time.



SOLID design principles

Liskov Substitution Principle (LSP)

The Liskov Substitution Principle demands that superclass objects are substitutable with objects of their subclasses without affecting system functionality. This is ensured by polymorphism in the user module. In our design, both Officer objects and Applicant objects can be treated as Applicant types in generic processes such as HDB Application or enquiry submission. The system will treat subclass objects in a similar manner by using references to their superclass type with promises of appropriate, predictable behavior regardless of which specific subclass is used.

Applicant side

```
public boolean canApply(Project project, boolean asOfficer)
{
    if(project == null)
        return false;

    if(!project.isEligible(this)) //check age here.
    {
        return false;
    }

    //check if today's date within application period.
    LocalDate today = LocalDate.now();
    if (today.isBefore(project.getOpeningDate()) || today.isAfter(project.getClosingDate())) {
        // Today is NOT within the project period
        return false;
    }
```

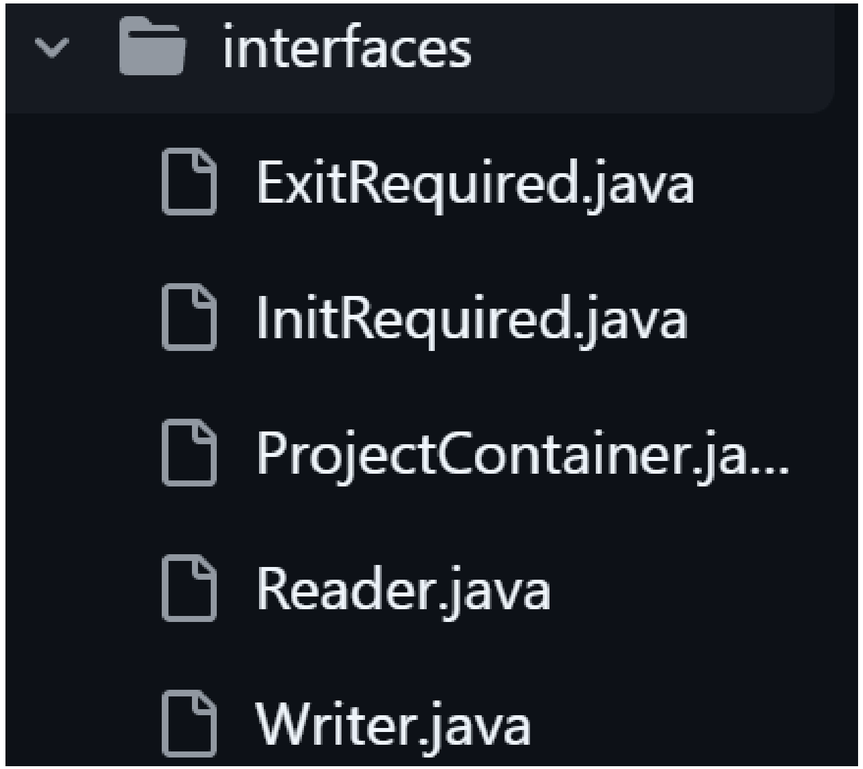
Officer Side (inherits from Applicant class)

```
@Override
public boolean canApply(Project project, boolean asOfficer){
    //check against list of projects, then check against applicant side's applicants.

    //check if officer has already applied to this project as an applicant
    List<Application> filtered = getApplications(List.of(Application.Status.PENDING, Application.Status.SUCCESSFUL, Application.Sta
    for(Application application : filtered){ //this checks if officer has already applied for this as a project.
        if(application.getProject() == project){
            //System.out.println("CannotApplyProject AS APPLICANT" + project.getProjectName());
            return false;
        }
    }

    //check if officer applications date clash with any other applications as applicant
    for(Application application : filtered){
        if(!(application.getProject().assertDateClash(project.getOpeningDate()) && application.getProject().assertDateClash(project
            //System.out.println("CannotApplyProject DATE CLASH APPLICANT" + project.getProjectName());
            return false;
        }
```

SOLID design principles

Interface Segregation Principle (ISP)

Interface Segregation Principle (ISP) prefers defining several specialist interfaces over one fat interface. The principle is successfully implemented in this system by using narrow single-purpose interfaces such as Reader, Writer, InitRequired, and ExitRequired. These are equivalent to different contracts pertaining to separate functional concerns of the system. For example, DAOs such as UserDAO, ApplicationDAO, and EnquiryDAO implement only Reader and Writer, and thus are never required to have responsibility for persistence behavior with unwanted concerns. Similarly, controller objects implement InitRequired and ExitRequired for clear specification of initialization and shutdown duties. This makes it more modular, less likely to implement unwanted methods in foreign classes, and to have high cohesion. Following ISP, it is quite easy to specify finer contracts, test, and evolve later.

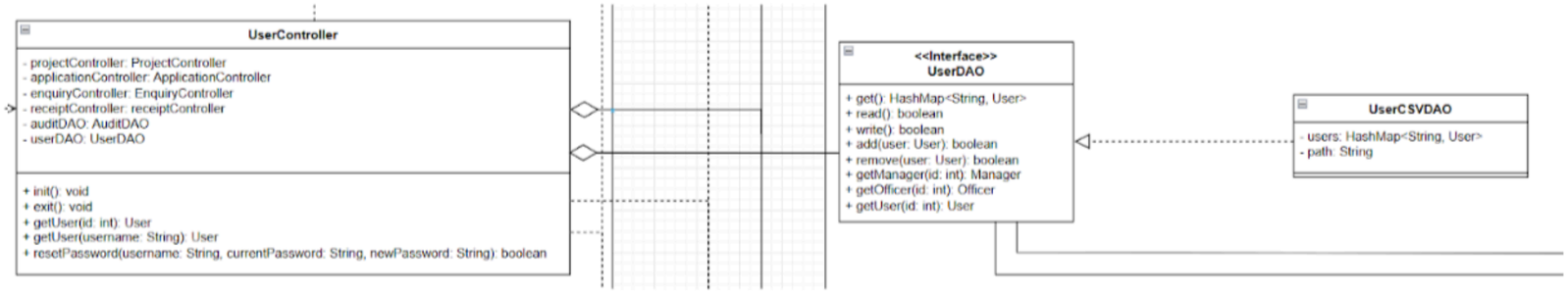


SOLID design principles

Dependency Inversion Principle (DIP)

According to the Dependency Inversion Principle, modules of higher abstraction levels are not to depend upon those of lower abstraction levels; rather, they are to depend upon abstract modules. Even in the absence of formal interfaces that promote abstraction across layers, the system is in accordance with this principle by maintaining loose coupling among classes.

For instance, we implemented an abstraction layer between the reading, writing and storage of data in the system, allowing for greater flexibility in switching between different storage mechanisms (eg. CSV, database or cloud storage), without altering the core logic of the application.



Additional features

DAO Layer Using CSV:
Abstracted data access via DAO and CSVDAO classes for all major entities. This design allows flexible data handling and future extensibility .

Audit Logging:
Logs all critical system events (reset passwords, bookings, approvals) in AuditLog.csv.

Password Change Functionality:
Users can change their passwords after login to secure their account.

CSV-Based Persistence:
Application state is saved and loaded from .csv files for user, project, flat, application, enquiry, and receipt data.


Search...
DEMO
TIME
www.reallygreatsite.com


LICERIA.CO
Home
About Us
Contact
Search...
Thank you
www.reallygreatsite.com