



UE21CS352B - Object Oriented Analysis & Design using Java

Mini Project Report

Real Estate Retail Interface

Submitted by:

Shashank Prashanth Patali	PES2UG22CS521
Shaun DCosta	PES2UG22CS524
Shrijit S Hunsikatti	PES2UG22CS543
Shripad Amol Parchure	PES2UG22CS545

6th Semester I Section

Prof. Ruby Dinakar

January - May 2024

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
FACULTY OF ENGINEERING
PES UNIVERSITY**

(Established under Karnataka Act No. 16 of 2013)
100ft Ring Road, Bengaluru – 560 085, Karnataka, India

Problem Statement: Real Estate Retail Interface

In today's fast-paced digital world, the real estate industry faces significant challenges in managing property listings, streamlining user interactions, and handling appointments efficiently. Traditional property management systems often lack user-friendly interfaces, real-time appointment scheduling, and secure access control for different user roles. This project aims to address these issues by developing a comprehensive web-based Real Estate Retail Interface that enables seamless property browsing, user registration and authentication, appointment booking for site visits, and centralized property management.

Built using Vaadin for the frontend, Spring Boot for backend logic, and MySQL for data persistence, the system handles both administrators and customers by offering an intuitive, secure, and responsive platform for property transactions and engagement.

Key Features:

1. **User Authentication:**
 - Login and Signup views allow users to authenticate and register with roles like "Buyer" or "Admin".
2. **Property Management:**
 - Admins can manage properties, including adding, editing, and deleting properties.
3. **Property Listings:**
 - Separate views for Residential, Commercial, and Industrial properties, each displaying property details and allowing users to interact.
4. **Appointment Booking:**
 - Users can book appointments for properties, specifying date and time.
5. **Registration for Properties:**
 - Users can register their interest in properties, with a transaction process to confirm registration.
6. **Admin Dashboard:**
 - Admins can view and manage all appointments, users, and property registrations.

7. Email Notifications:

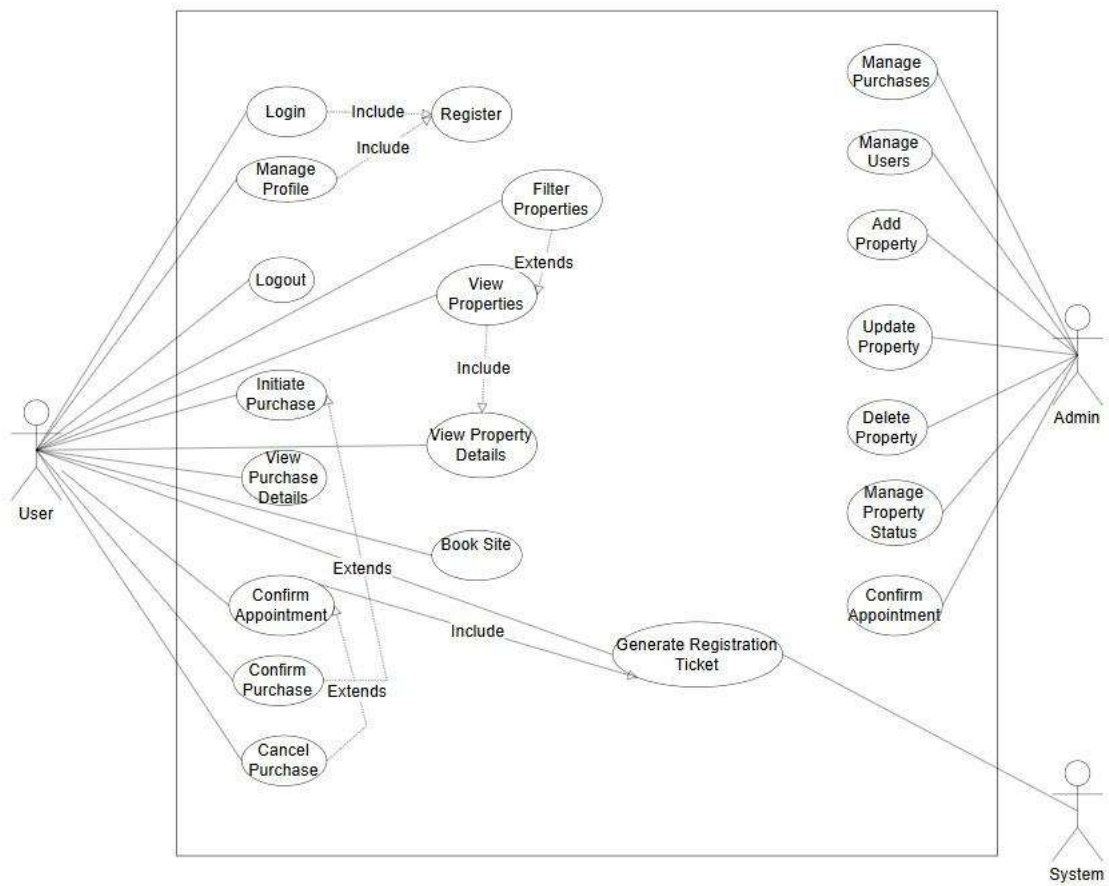
- Automated emails for appointment confirmations, registration success, and rejections.

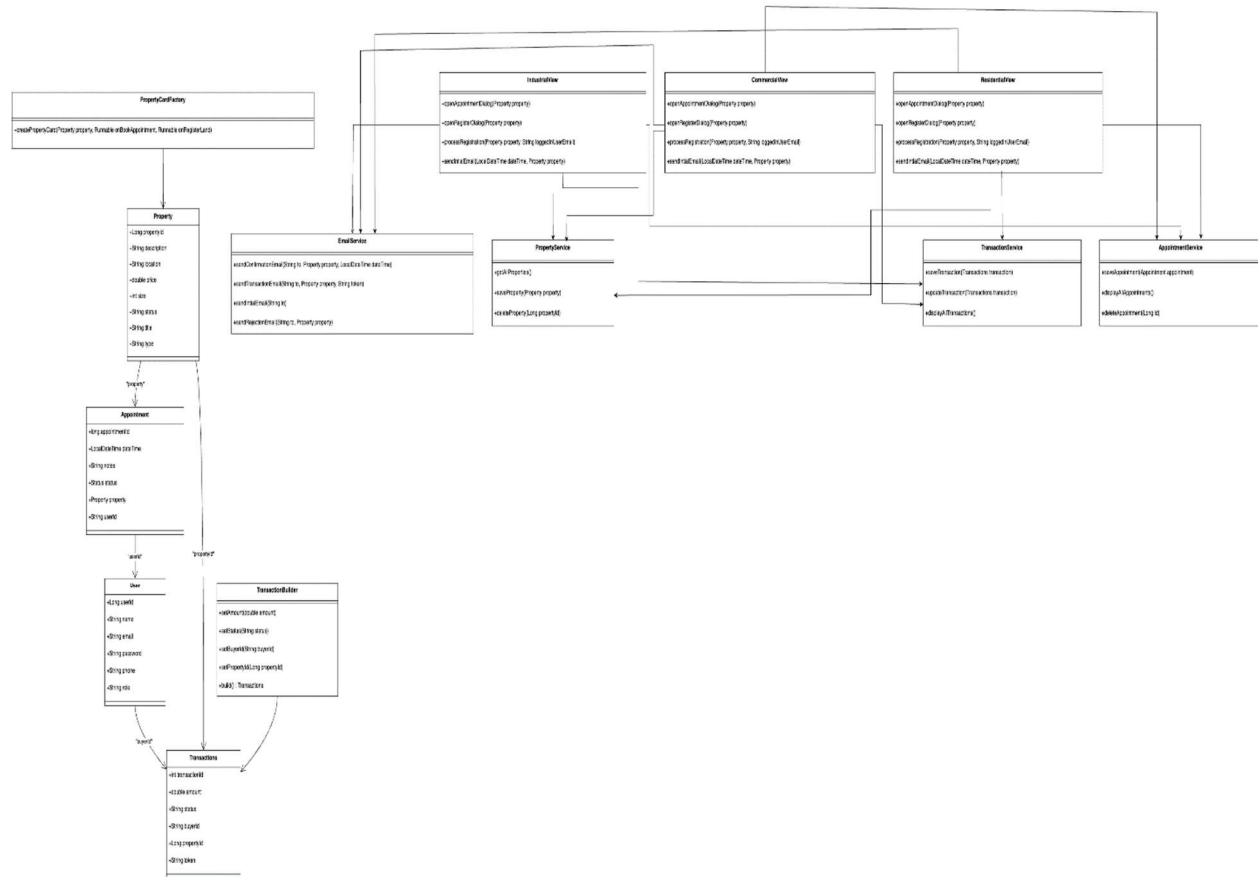
8. Transaction Management:

- Handles property registration transactions, including saving, updating, and completing transactions.

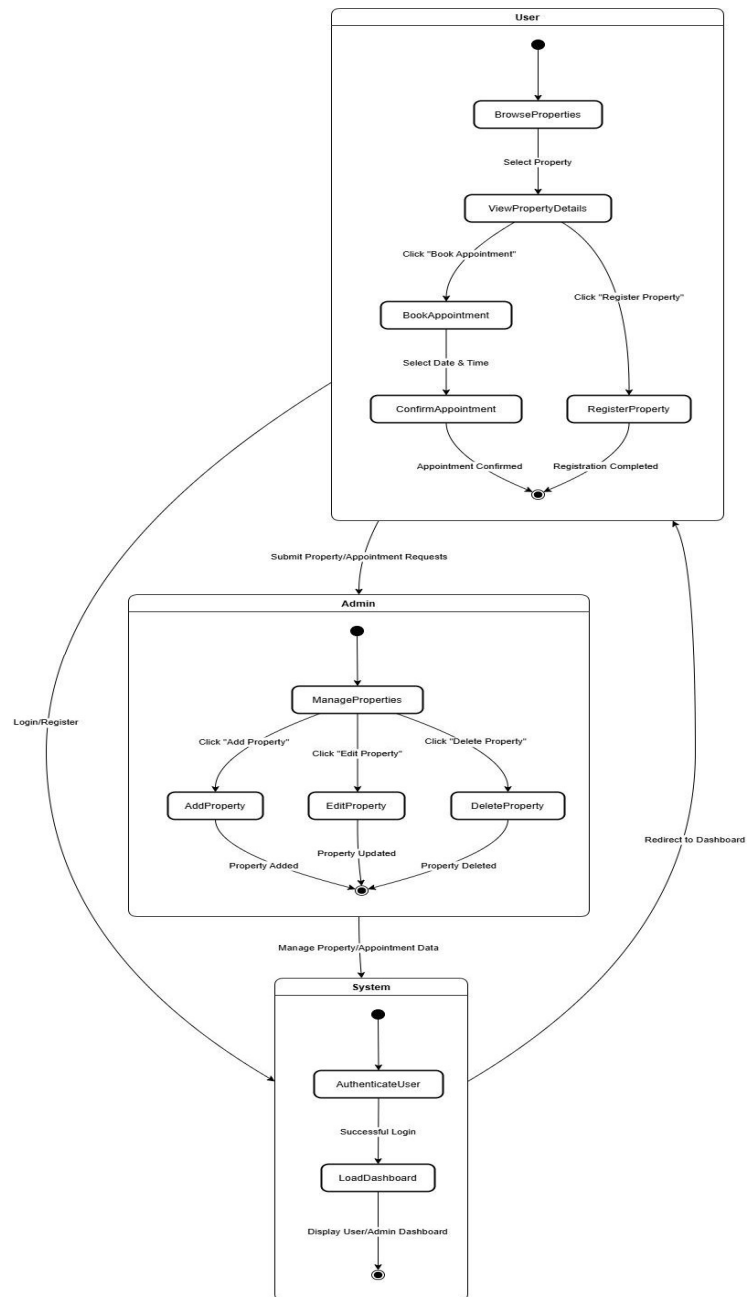
Models:

Use Case Diagram:

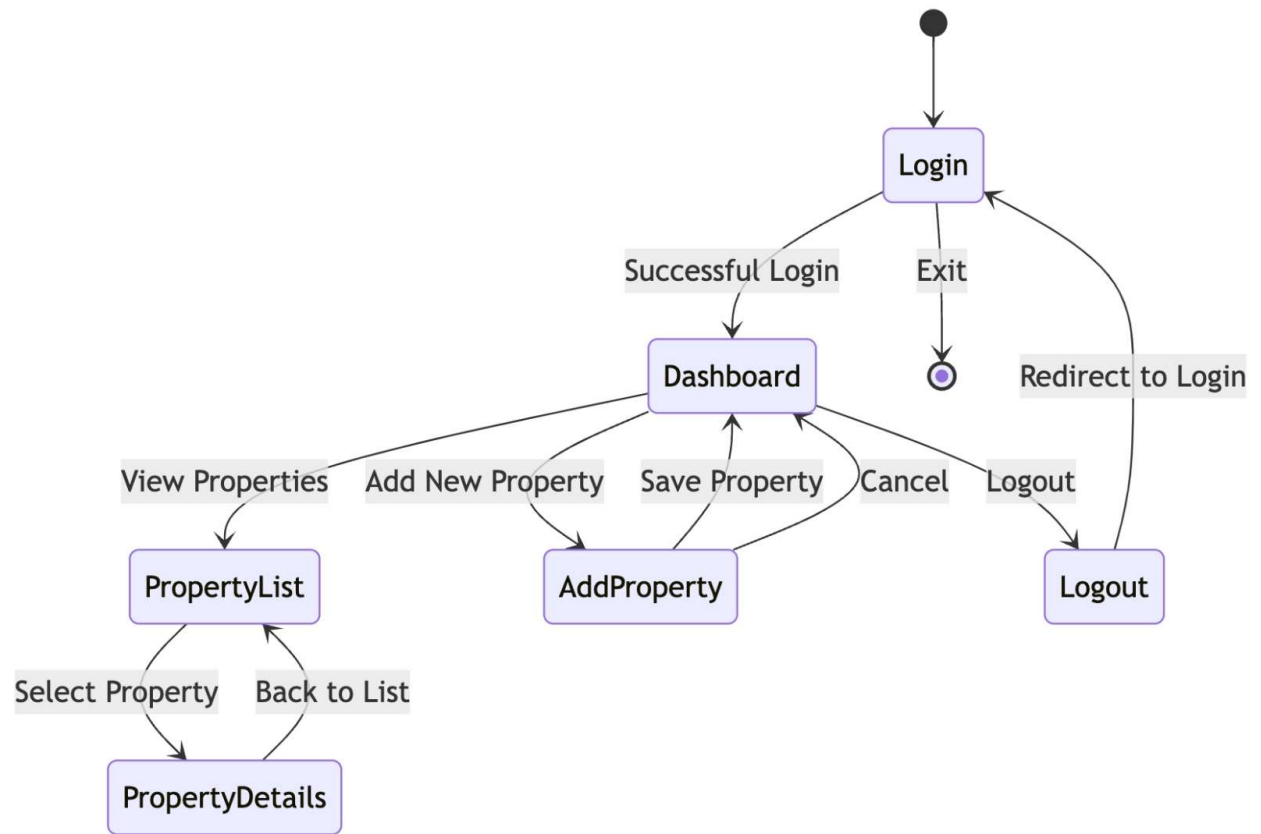




Activity Diagram



State Diagram



Architecture Patterns, Design Principles, and Design Patterns:

Architecture Patterns

1. Singleton Pattern.

The Singleton Pattern ensures a class has only one instance and provides a global point of access to it.

Usage – *EmailService* Class

- EmailService is implemented as a singleton pattern to manage all the email related services.
- Many other classes such as *ResidentialView*, *IndustrialView* and *CommercialView* are using *EmailService* to instantiate and send emails. Therefore, each of these classes can invoke its own instance of EmailService and this leads to unnecessary resource usage.
- If Singleton Pattern is used, *EmailService* is instantiated only once throughout the session and makes sure that every other class uses the same single instance created.
- This is done using the *getInstance()* method in the EmailService

Benefits –

- Reduces memory usage by avoiding multiple instances.
- Ensures consistent behavior across the application.
- Simplifies access to shared resources

```
@Service
public class EmailService {
    private static EmailService instance;
    private final JavaMailSender mailSender;

    private EmailService(JavaMailSender mailSender) {
        this.mailSender = mailSender;
    }

    public static EmailService getInstance(JavaMailSender mailSender) {
        if (instance == null) {
            synchronized (EmailService.class) {
                if (instance == null) {
                    instance = new EmailService(mailSender);
                }
            }
        }
        return instance;
    }
}
```


2. Factory Pattern

The Factory Pattern defines an interface for creating objects but allows subclasses to alter the type of objects that will be created.

Usage – *PropertyCardFactory* class

- This is used to create property cards dynamically.
- This helps in centralising the creation logic of property cards and ensures consistency in their structure and styling.
- The logic of creating property cards remain same across various views such as *ResidentialView*, *CommercialView* and *IndustrialView*. Therefore, inconsistency and code duplication can take place if each of the views had their own method of implementing the creation of property cards.
- Any small change done to the design of property card would need changes across different places.
- Hence, the logic for creating property cards is centralized in the *PropertyCardFactory* class.
- The factory takes a Property object and callbacks for actions like booking appointments and registering land, making it reusable and modular.
- With this pattern, even the other views such as *ResidentialView*, *CommercialView* and *IndustrialView* will have lesser and cleaner code.

Benefits –

- Promotes reusability and reduces code duplication.
- Simplifies maintenance by isolating changes to the card creation process.
- Ensures uniformity in the UI components.

```
src > main > java > com > example > application > views > . PropertyCardFactory.java > PropertyCardFactory > createPropertyCard(Property, Runnable, Runnable)
1 package com.example.application.views;
2
3 import com.vaadin.flow.component.button.Button;
4 import com.vaadin.flow.component.dependency.CssImport;
5 import com.vaadin.flow.component.html.*;
6 import com.vaadin.flow.component.orderedlayout.*;
7 import com.example.application.model.Property;
8
9 @CssImport("./styles/commercial-view.css")
10 public class PropertyCardFactory {
11     public static Div createPropertyCard(Property property, Runnable onBookAppointment, Runnable onRegisterLand) {
12         Div card = new Div();
13         card.addClassName(className("property-card"));
14
15         // Title
16         H2 title = new H2(property.getTitle());
17         title.addClassName(className("property-title"));
18
19         // Location and Size
20         Div infoDiv = new Div();
21         infoDiv.addClassName(className("property-info"));
22         Span location = new Span("Location: " + property.getLocation());
23         Span size = new Span("Size: " + property.getSize() + " sq.ft");
24         infoDiv.add(location, size);
25
26         // Description
27         Paragraph description = new Paragraph(property.getDescription());
28         description.addClassName(className("property-description"));
29
30         // Price
31         H4 price = new H4("$" + property.getPrice());
32         price.addClassName(className("property-price"));
33
34         // Status
35         Span status = new Span(property.getStatus());
36         status.addClassName(className("property-status"));
37         if ("AVAILABLE".equalsIgnoreCase(property.getStatus())) {
38             status.addClassName(className("status-available"));
39         } else if ("SOLD".equalsIgnoreCase(property.getStatus())) {
40             status.addClassName(className("status-sold"));
41         } else if ("BOOKED".equalsIgnoreCase(property.getStatus())) {
42             status.addClassName(className("status-booked"));
43         }
44
45         // Buttons
46         HorizontalLayout buttonLayout = new HorizontalLayout();
47         buttonLayout.addClassName(className("button-container"));
48     }
49 }
```

3. Builder Pattern

The Builder Pattern separates the construction of a complex object from its representation, allowing the same construction process to create different representations.

Usage - *TransactionBuilder*

- *TransactionBuilder* class has been used to construct or build *Transactions* objects.
- This pattern simplifies the creation of complex *Transactions* objects with multiple optional attributes.
- If this pattern is not used, *Transactions* will have to be built using a constructor with multiple parameters. This constructor call will make the code more cluttered, less readable and very difficult to understand.
- Adding or modifying parameters would require changes to be made at multiple changes.
- The *TransactionsBuilder* class now provides a fluent interface for constructing *Transactions* object. It allows to chain methods to set properties which makes the code cleaner and easier to understand.

Benefits

- Improves code readability and reduces errors.
- Supports immutability by ensuring objects are fully constructed before use.
- Makes object construction flexible and intuitive

```
src > main > java > com > example > application > model > TransactionBuilder.java > com.example.application.model
1 package com.example.application.model;
2
3 public class TransactionBuilder {
4     private double amount;
5     private String status;
6     private String buyerId;
7     private Long propertyId;
8
9     public TransactionBuilder setAmount(double amount) {
10         this.amount = amount;
11         return this;
12     }
13
14     public TransactionBuilder setStatus(String status) {
15         this.status = status;
16         return this;
17     }
18
19     public TransactionBuilder setBuyerId(String buyerId) {
20         this.buyerId = buyerId;
21         return this;
22     }
23
24     public TransactionBuilder setPropertyId(Long propertyId) {
25         this.propertyId = propertyId;
26         return this;
27     }
28
29     public Transactions build() {
30         Transactions transaction = new Transactions();
31         transaction.setAmount(amount);
32         transaction.setStatus(status);
33         transaction.setBuyerId(buyerId);
34         transaction.setPropertyId(propertyId);
35         return transaction;
36     }
37 }
```

4. Iterator Pattern

The Iterator Pattern provides a way to access elements of a collection sequentially without exposing its underlying representation.

Usage – Used in various classes like *ResidentialView*, *CommercialView*, *IndustrialView*

- All the above views need to iterate over all the properties to create a property card.
- If `List<property>` and a For loop was used, the views become highly coupled to the List implementation and making it harder to change the underlying data structure.
- Using the Iterator pattern, the views now use a *PropertyCollection* class to manage properties and a *PropertyIterator* to traverse them.
- The iteration logic is encapsulated in the *PropertyIterator*, making the views cleaner and more focused on UI concerns.

Benefits –

- Enhances maintainability and scalability.
- Simplifies traversal logic, making it easier to iterate over collections.
- Supports multiple types of collections without modifying the iteration logic.

Iterator pattern in use

```
CommercialPropertyIterator iterator = propertyCollection.createIterator();
if (!iterator.hasNext()) {
    add(header, new Paragraph(text:"No commercial properties available at the moment")
} else {
    while (iterator.hasNext()) {
        Property originalProperty = iterator.next();
```

5. Observer Pattern

The Observer Pattern defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

Usage – *NotificationUtils* Class

- Notification System which is the *NotificationUtils* is an observer which is used for displaying styled notifications.
- It is used to notify users of changes or actions (e.g., appointment confirmation, registration success).

Benefits –

- Decouples notification logic from the main application logic.
- Enhances user experience with real-time feedback.
- Simplifies the addition of new notification types.

Observer pattern in use

```
src > main > java > com > example > application > views > NotificationUtils.java > ...
1  package com.example.application.views;
2
3  import com.vaadin.flow.component.notification.Notification;
4  import com.vaadin.flow.component.html.Div;
5
6  public class NotificationUtils {
7
8      public static void showStyledNotification(String message, int durationMs) {
9          Notification notification = new Notification();
10         notification.setDuration(durationMs);
11         notification.setPosition(Notification.Position.TOP_CENTER);
12
13         Div content = new Div();
14         content.setText(message);
15         content.getStyle()
16             .set(name:"background-color", value:"#2C3E50")
17             .set(name:"color", value:"white")
18             .set(name:"padding", value:"10px")
19             .set(name:"border-radius", value:"8px")
20             .set(name:"font-weight", value:"bold");
21
22         notification.add(content);
23         notification.open();
24     }
25 }
26
```

6. Model – View – Controller Pattern (MVC)

The Model-View-Controller (MVC) architecture pattern is widely used in software development to separate the concerns of an application into three interconnected components: the Model, View, and Controller. Each component has a specific role and responsibility within the system.

Components of MVC:

1. Model –

The Model layer is responsible for representing the real-world entities and handling the core logic of the application. It contains the classes such as –

- User
- Property
- Appointment
- Transactions

The model also provides methods to manipulate and access these data structures, such as updating user profiles, creating new properties or setting up appointments. It serves as the foundation for storing, retrieving, and modifying the application's state.

2. View –

The View layer is implemented using Java Swing (or similar UI framework) and defines the graphical interface that users interact with. It contains classes such as –

- ResidentialView
- IndustrialView
- CommercialView
- LoginView
- SignupView

3. Controller –

The Controller layer acts as the coordinator between the view and the model. It handles user requests, processes them, and updates both the model and the view accordingly. Controllers decouple the user interface from the logic, allowing the application to scale or evolve independently in either direction.

Benefits of MVC:

- MVC separates the application into three interconnected components: Model, View, and Controller. This separation makes the application easier to manage, test, and scale by isolating business logic (Model), user interface (View), and user input handling (Controller).
- The Model and View components are independent of each other, meaning the same Model can be reused with different Views, and Views can be reused with different Controllers, reducing duplication and effort.
- MVC allows for modular development. Teams can work on different components (Model, View, Controller) simultaneously, making it easier to scale the application and development process.
- Changes in one component (e.g., View) do not affect others (e.g., Model). This modularity simplifies debugging, testing, and updating the application

Architecture Principles

1. Single Responsibility Principle (SRP)

Principle: **A class should have only one reason to change.**

Usage :

- Methods like `refreshProperties`, `showAllAppointments`, and `showAllUsers` in the *HomeView* class each have a single responsibility.
- Each method is responsible for a specific task, such as refreshing the property list or displaying appointments, ensuring the class remains modular and easier to maintain.

2. Open – Closed Principle (OCP)

Principle: **Software entities (such as classes, modules, and functions) should be open for extension but closed for modification.**

Usage:

- The *HomeView* services like *PropertyService*, *AppointmentService*, *TrasanctionsService* are classes that can be extended with new functionality without modifying the *HomeView* class
- The *HomeView* class is open for extension (adding new services or features) but closed for modification, as the logic for managing properties, appointments, and transactions is encapsulated in their respective services.

3. Liskov Substitution Principle (LSP)

Principle: Objects of a superclass should be replaceable with objects of a subclass without affecting the correctness of the program.

Usage:

- The Grid component in methods like *showAllAppointments* and *showAllRegistrations* is used with different data types (*Appointment*, *Transactions*).
- The Grid class adheres to LSP by allowing substitution with any compatible data type, ensuring that the behaviour of the Grid remains consistent regardless of the data type used.

4. Separation of Concerns

Principle - Separation of Concerns (SoC) is a design principle that promotes dividing a software system into distinct sections, where each section addresses a specific concern or responsibility

Usage –

- The *HomeView* class separates UI logic (Ex - creating property cards, dialogs) from business logic (Ex- saving properties, managing appointments).
- Each layer of the application has a distinct responsibility. For example, the *PropertyService* handles property-related operations, while the *HomeView* focuses on rendering the UI and interacting with the user.

5. High Cohesion

Principle - High Cohesion is a design principle that ensures a class, module, or component is focused on a single, well-defined purpose or responsibility.

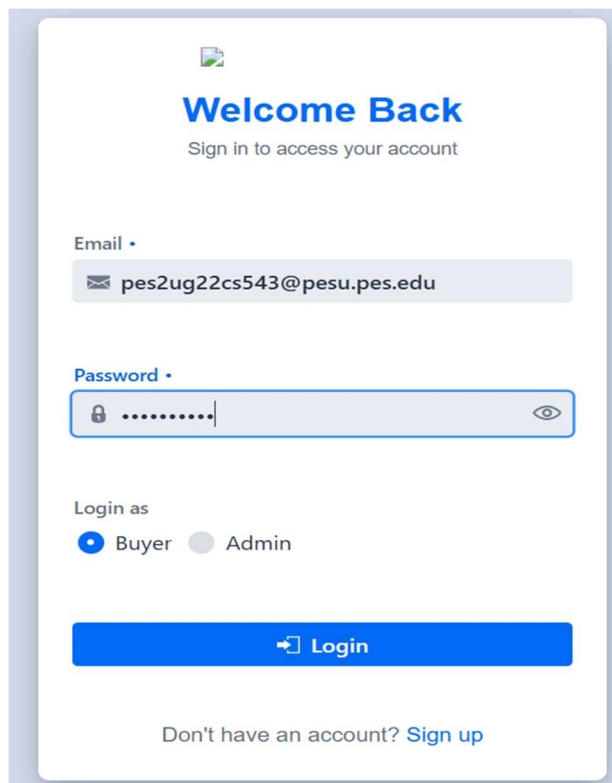
Usage –


- Classes like *PropertyCardFactory* demonstrates high cohesion.
- This is because it is solely responsible for creating property cards.
- All its methods and logic are focused on this single task.
- It avoids handling unrelated concerns like UI rendering or data storage, which are managed elsewhere.

Github link : https://github.com/shrijitt04/Real_Estate_Interface

UI SCREENSHOTS:

Login page

A screenshot of a login page. At the top, there is a small house icon, followed by the text "Welcome Back" in bold blue, and "Sign in to access your account" in a smaller font. Below this, there are two input fields: "Email" with a dropdown arrow and a value "pes2ug22cs543@pesu.pes.edu", and "Password" with a dropdown arrow, a masked password ".....", and an eye icon to toggle visibility. Under the password field, there is a "Login as" section with two radio buttons: "Buyer" (selected) and "Admin". At the bottom of the form is a blue "Login" button with a right-pointing arrow icon. Below the button, there is a link "Don't have an account? Sign up".



Welcome Back

Sign in to access your account

Email •


✉ pes2ug22cs543@pesu.pes.edu

Password •

🔒 👁

Login as

☒ Buyer ☐ Admin

 Login


Don't have an account? [Sign up](#)

SignUp page


Create Account

Sign up to start your real estate journey



Name •

 SHRIJIT S HUNSIKATTI


Email •

 shrijitsh@gmail.com

Password •


 

Phone Number •

 09108221968

Role

☒ Buyer ☐ Admin

 Sign Up

Admin View

[Add New Property](#) [Manage Appointments](#) [Manage Users](#) [Manage Registrations](#)

Property Management

Beachfront Resort Land

Location: Goa, India
Size: 50000
prime beachfront land suitable for a luxury resort.
\$2000000.0
Type: Commercial
[AVAILABLE](#)
[Edit Details](#)
[Delete Property](#)

Industrial Park Land

Location: Pune, Maharashtra
Size: 140000
Sprawling land perfect for an industrial park.
\$4500000.0
Type: INDUSTRIAL
[AVAILABLE](#)
[Edit Details](#)
[Delete Property](#)

Scenic Hilltop Land

Location: Shimla, Himachal Pradesh
Size: 30000
Beautiful hilly plot perfect for a private villa.
\$750000.0
Type: RESIDENTIAL
[AVAILABLE](#)
[Edit Details](#)
[Delete Property](#)

Luxury Hotel Site

Location: Udaipur, Rajasthan
Size: 75000
Well-connected land for setting up a high-end hotel.
\$3800000.0
Type: COMMERCIAL
[SOLD](#)
[Edit Details](#)
[Delete Property](#)

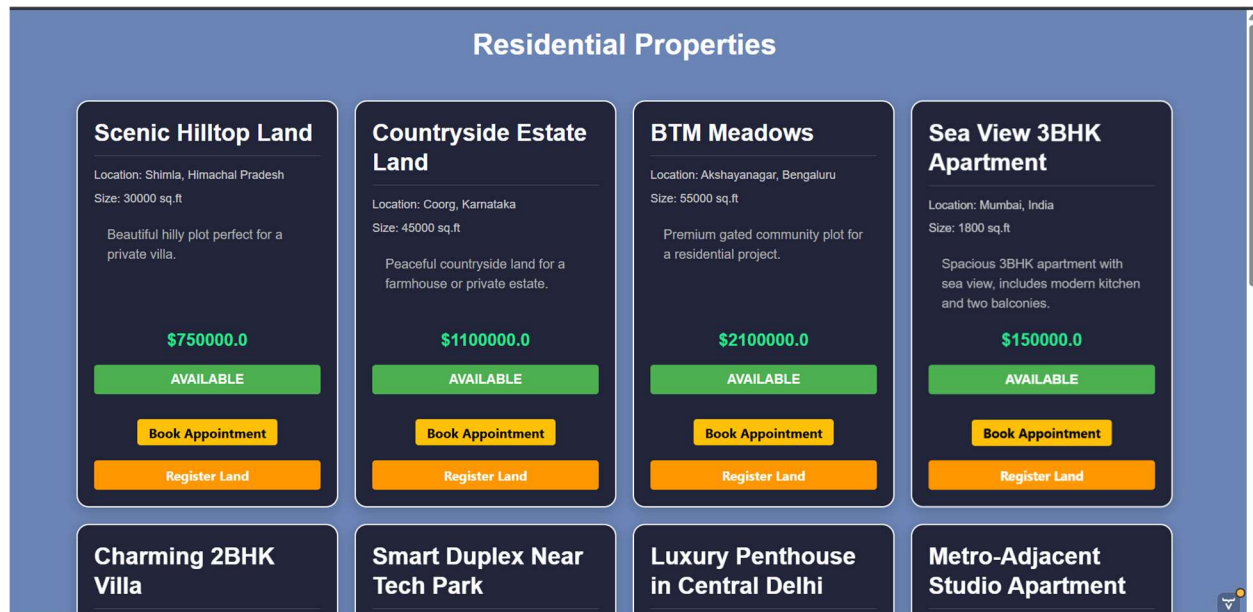
Manufacturing Hub Land

Countryside Estate Land

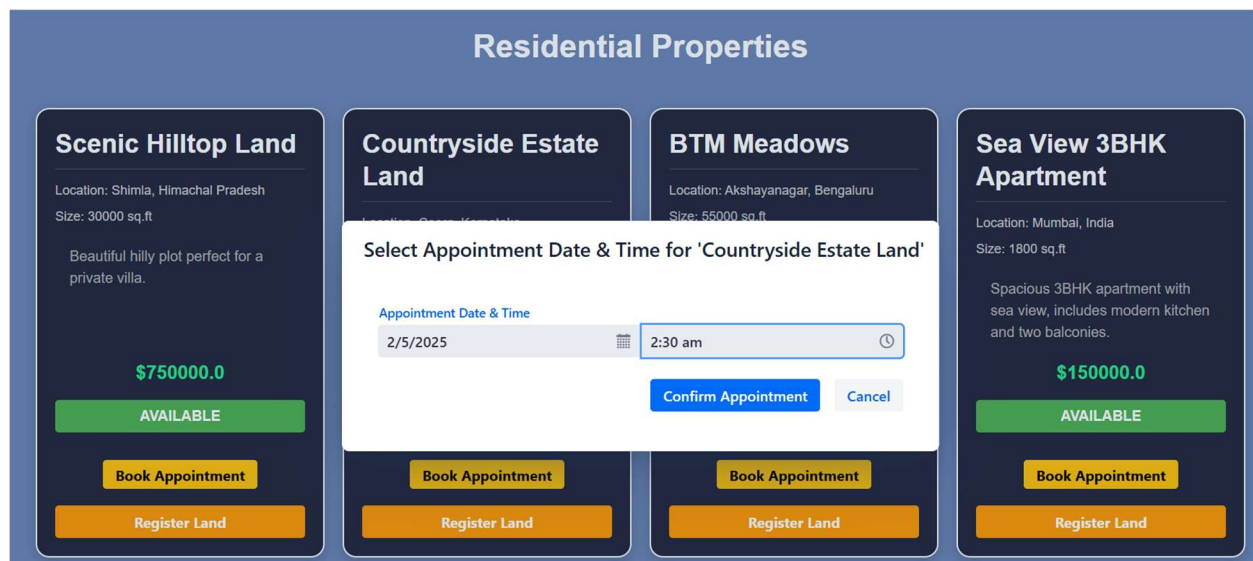
Budget Hotel Site

Logistics & Warehousing Land

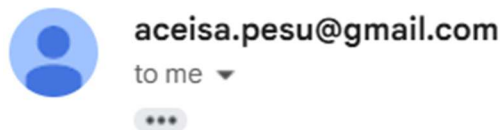
User Home



User can book appointment



Mail initiated after booking an appointment



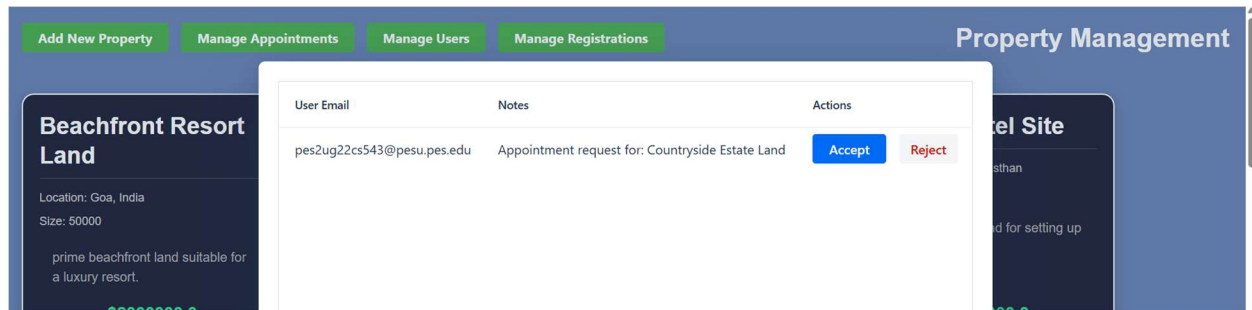
Hello!

Your appointment request has been received and the process has been initiated.

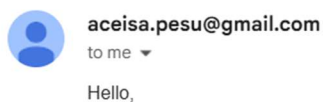
An email confirmation will be sent to you regarding the same.

Thank You!

Admin can accept or reject the appointment.



Confirmation Email sent after accepting the appointment



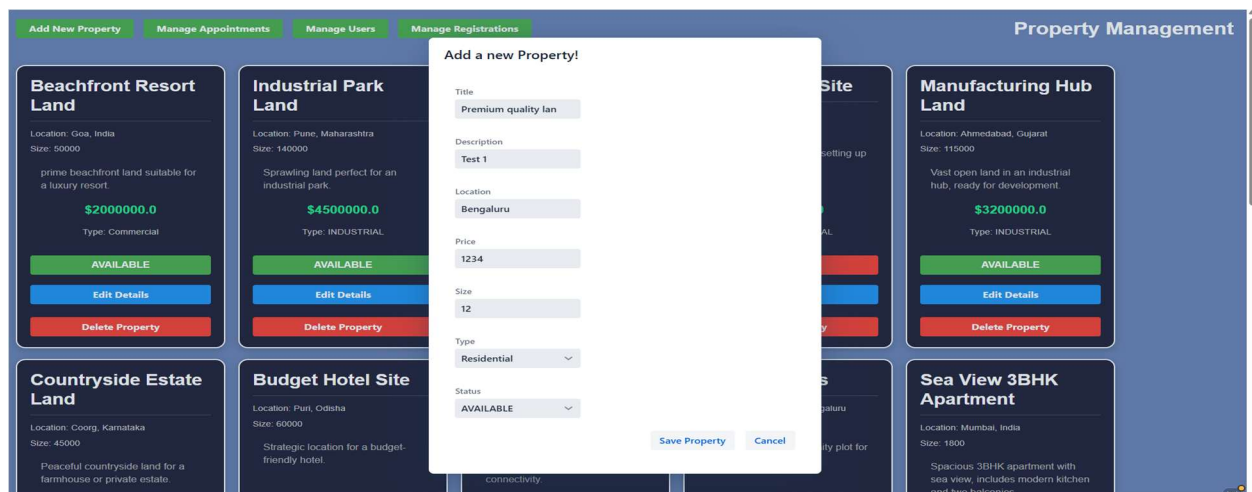
Hello,

Your appointment for the property titled "Countryside Estate Land" located at Coorg, Karnataka has been confirmed for 2025-05-02T02:30.

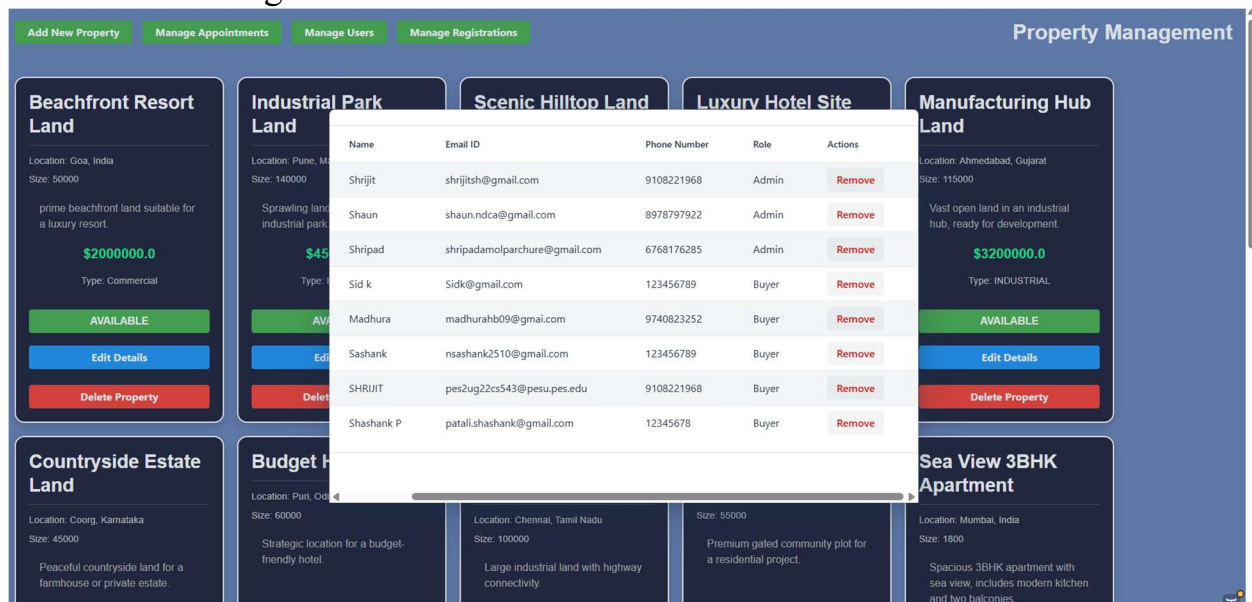
Thank you!



Admin can add a new property



Admin can manage users



Individual contributions of the team members:

Name	Module worked on
Shrijit S Hunsikatti	Property Management Module
Shashank Prashanth	User Management Module
Shaun Navanit Dcosta	Purchase and Registration Module
Shripad Amol Parchure	Appointment Module