

RIPHAH INTERNATIONAL UNIVERSITY, GG CAMPUS



Software Construction & Development

"Smart Real Estate Management System (SREMS)"

Project Team

Name	Sap ID	Program	Email Address
Nagarash Fateh	44815	BSSE	44815@students.riphah.edu.pk
Manahil Habib	47876	BSSE	47876@students.riphah.edu.pk
Esha Asif	45786	BSSE	45786@students.riphah.edu.pk
Rahat Qadeer	47234	BSSE	47234@students.riphah.edu.pk

Date of Submission

24/11/2024

Table of Contents

Project Proposal	3
Use Case Diagram.....	5
Fully Dressed Use Cases.....	7
Domain Modeling	12
Class Diagram.....	16
Activity Diagram	21
Sequence Diagram	24
State Transition Diagram	27
MVC Framework.....	31
GRASP Patterns.....	38

Artifact 01

Project Proposal

Project Title: Smart Real Estate Management System

Description:

The Smart Real Estate Management System (SREMS) is a comprehensive platform designed to manage real estate listings, schedule property viewings, handle client inquiries, and facilitate transactions. It supports different user roles—Admin, Agent, and Client—each with specific functionalities to streamline the property management and sales process. Following the MVC framework and GRASP principles, the system ensures organized code and seamless interactions between objects, providing secure authentication and role-based access.

Literature Survey:

Features Description:

- F-1 User Authentication and Role-Based Access Control
- F-2 Role-specific functionalities
- F-3 Admin manages users, oversees property listings, and monitors transactions
- F-4 Agent lists properties, manages client inquiries, schedules viewings, and handles offer approvals
- F-5 Client browses property listings, schedules viewings, submits offers, and tracks offer status
- F-6 Agents can create and manage property listings with details such as price, location, photos, and status
- F-7 Search and Filter properties by criteria like price, property type, location, and rooms
- F-8 Virtual Tours and Media for remote property viewing
- F-9 Scheduling Viewings
- F-10 Clients can request viewings based on available time slots
- F-11 Agents can approve, reschedule, or decline viewing requests
- F-12 Notification System for reminders on viewings and offer updates
- F-13 Offer Submission
- F-14 Clients can submit purchase offers
- F-15 Agents can accept, counter, or reject offers
- F-16 Transaction Records of accepted offers, including price, terms, and client-agent details
- F-17 Status Updates for clients on offer status (e.g., Pending, Accepted, Rejected)
- F-18 Property Performance reports with metrics on views, inquiries, and offers
- F-19 Agent Activity analytics on listings created, viewings scheduled, and successful sales
- F-20 Client Insights on client interest, viewing trends, and offer conversion rates

Artifact 02

Use Case Diagram



Figure 1 Use Case Diagram of Smart Real Estate Management System (SREMS)

Artifact 03

Fully Dressed Use Cases

Use Cases:

Critical Functionalities

- UC-1:** User Login: Secure login with role-based access.
- UC-2:** Role Management: Admin's ability to manage user roles and permissions.
- UC-3:** Create Property Listing: Agents adding and managing property listings.
- UC-4:** Search Properties: Clients searching and filtering properties.
- UC-5:** Request Property Viewing: Clients requesting viewings of properties.
- UC-6:** Review Offers: Agents reviewing, accepting, countering, or rejecting offers.
- UC-7:** Log Transaction Details: Recording transaction details upon offer acceptance.
- UC-8:** Submit Purchase Offer: Clients submitting purchase offers on properties.
- UC-9:** Offer Status Updates: Notifying clients of updates on their offers (Pending, Accepted, Rejected).

Medium Functionalities

- UC-10:** Update Property Listing: Agents updating property details or status.
- UC-11:** Manage Viewing Requests: Agents approving, rescheduling, or declining viewing requests.
- UC-12:** Automated Notifications: Sending reminders to agents and clients for viewings and offer updates.
- UC-13:** Property Performance Reports: Admin viewing reports on property views, inquiries, and offers.
- UC-14:** Agent Activity Reports: Admin reviewing agent activities such as listings and sales.
- UC-15:** Client Insights: Agents accessing analytics on client interest, viewing trends, and offer conversion.

Low Functionalities

- UC-16:** View Property Details: Clients viewing detailed property information, including media (photos, virtual tours).
- UC-17:** Role-Based Functionality: Ensuring each role has specific functionality (though this is part of the login and role management, it's a support feature).

UC-01 Login

Section	Content
Designation	Login
Name	User Authentication and Role-Based Access Control
Authors	Manahil Habib
Priority	High
Criticality	Critical
Source	System Requirements
Responsible	System Developer
Description	Secure login with role-based access that assigns specific functionalities to Admins, Agents, and Clients.
Trigger event	User navigates to the login page and enters credentials.
Actors	Admin, Agent, Client
Precondition	User has registered and has valid login credentials
Postcondition	User is logged in with access to role-specific functionalities
Result	User gains appropriate access based on their role.
Main Scenario	<ol style="list-style-type: none">1. User enters login credentials.2. System verifies the credentials and identifies the user's role (Admin, Agent, or Client).3. Based on the role, the system grants access to specific functionalities:<ul style="list-style-type: none">• Admin: Access to user management, property listings, and transaction monitoring.• Agent: Access to property listing management, client inquiries, viewings, and offer handling.• Client: Access to property browsing, scheduling viewings, and submitting offers.
Alternative Scenario	If credentials are invalid, the system prompts the user to retry or reset their password.

UC-02 Create Property Listing

Section	Content
Designation	Create Property Listing
Name	Property Listing and Management
Authors	Rahat Qadeer
Priority	High
Criticality	Critical
Source	System Requirements
Responsible	System Developer
Description	Allows agents to create, update, and manage property listings.
Trigger event	Agent wants to add or manage property listings.
Actors	Agent
Precondition	Agent is logged in with the necessary permissions.
Postcondition	Property listing is saved and visible to clients in search results.
Result	New or updated property listings are available for client searches.
Main Scenario	<ol style="list-style-type: none">1. Agent accesses the property management module.2. Agent creates a new property listing or updates an existing listing, including details like price, location, photos, and status (Available, Under Contract, or Sold).3. System saves the listing and makes it searchable for clients.
Alternative Scenario	If the listing details are incomplete, the system prompts the agent to fill in the required information.

UC-03 Submit Purchase Offer

Section	Content
Designation	Submit Purchase Offer
Name	Offer Submission and Transaction Management
Authors	Esha Asif
Priority	High
Criticality	Critical
Source	System Requirements
Responsible	System Developer
Description	Enables clients to submit purchase offers and agents to manage offer responses and transactions.
Trigger event	Client submits a purchase offer on a property.
Actors	Client, Agent
Precondition	Client is logged in, has selected a property, and is ready to make an offer.
Postcondition	Offer status is updated, and transaction details are logged if the offer is accepted.
Result	Transaction information is stored, and clients are updated on the offer status.
Main Scenario	<ol style="list-style-type: none">1. Client navigates to a property and submits a purchase offer, including terms and price.2. System notifies the agent associated with the property.3. Agent reviews the offer and responds by accepting, countering, or rejecting it.4. If the offer is accepted, the system logs transaction details (price, terms, client and agent information).5. Client receives a status update on the offer (Pending, Accepted, or Rejected).
Alternative Scenario	If the agent rejects or counters the offer, the client receives the updated status and can choose to respond or withdraw the offer.

Artifact 04

Domain Modeling

Domain Model

A domain model is a conceptual representation of the main entities, their attributes, and relationships within a specific problem domain. It is often used during the early stages of software development to help developers and stakeholders understand the structure and behavior of the system being designed. A domain model serves as a blueprint for how data and operations will be handled, ensuring clarity and alignment before detailed coding begins.

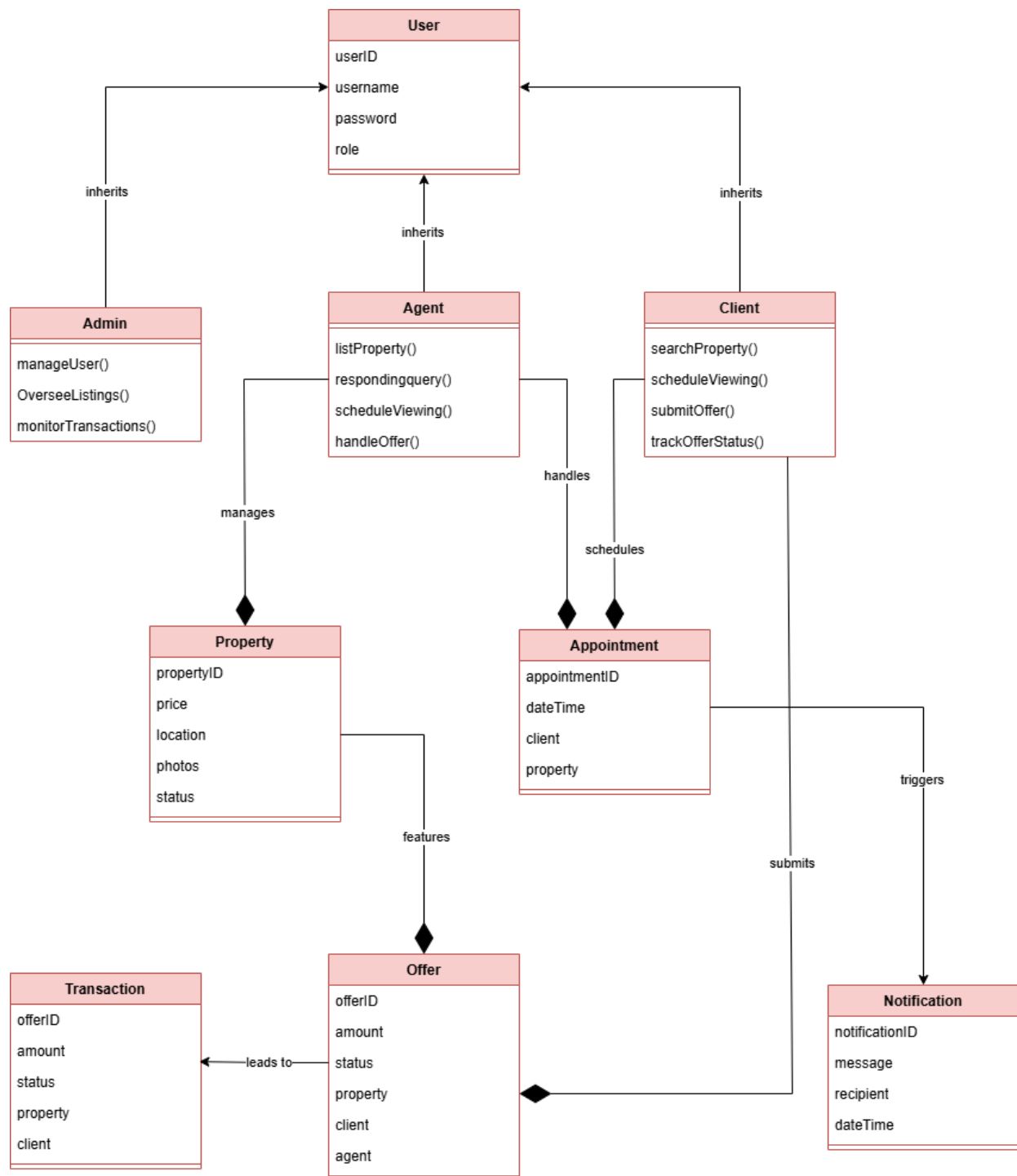


Figure 2 Domain Model of Smart Real Estate Management System (SREMS)

Key Classes and Their Roles:

1. **User:**
 - The User class is the base class representing any user in the system.
 - It includes common attributes such as userID, username, password, and role.
2. **Admin:**
 - Inherits from the User class.
 - Has capabilities such as manageUsers(), overseeListings(), and monitorTransactions() for overall system management.
3. **Agent:**
 - Also inherits from User.
 - Responsible for creating and managing Property instances, responding to client inquiries, scheduling viewings, and handling offers.
 - Contains methods like listProperty(), respondInquiry(), and scheduleViewing().
4. **Client:**
 - Inherits from User.
 - Represents a user interested in searching for properties, scheduling viewings, submitting offers, and tracking offer status.
 - Methods include searchProperty(), scheduleViewing(), submitOffer(), etc.
5. **Property:**
 - Represents real estate listings with attributes such as propertyID, price, location, photos, and status (e.g., Available, Under Contract, Sold).
 - Linked to Agent as agents create and manage properties.
6. **Appointment:**
 - Represents a scheduled viewing with attributes like appointmentID, dateTime, and references to the Client and Property involved.
 - Shows the relationship between clients scheduling appointments and properties.
7. **Offer:**
 - Represents a purchase offer, containing details such as offerID, amount, status (e.g., Pending, Accepted, Rejected), and references to Property, Client, and Agent.
 - Demonstrates interactions between clients submitting offers and agents handling them.
8. **Notification:**
 - A utility class that handles sending notifications to User instances with attributes like notificationID, message, and dateTime.

- Supports the functionality for notifying users about upcoming viewings or updates on offers.

9. Transaction:

- Records successful transactions after an offer is accepted.
- Contains attributes like transactionID, price, terms, and references to Client, Agent, and Property.
- Shows the outcome of offers leading to completed sales.

Relationships Explained:

- **Inheritance:**

- Admin, Agent, and Client inherit from User, representing different roles with their respective capabilities.

- **Associations:**

- Agent is associated with Property to show that agents manage property listings.
- Client is associated with Appointment to represent the clients schedule viewings.
- Property is linked to Offer, showing that properties are associated with purchase offers.
- Offer is connected to Transaction, indicating that accepted offers lead to recorded transactions.
- Notification is linked to User, as notifications are sent to users (e.g., clients, agents).

Artifact 05

Class Diagram

Class Diagram

A class diagram is a type of UML (Unified Modeling Language) diagram that visually represents the structure of a system by showing its classes, attributes, operations (methods), and the relationships between them. It is commonly used in object-oriented design to model the static structure of a system and is crucial during the planning and design phase of software development.

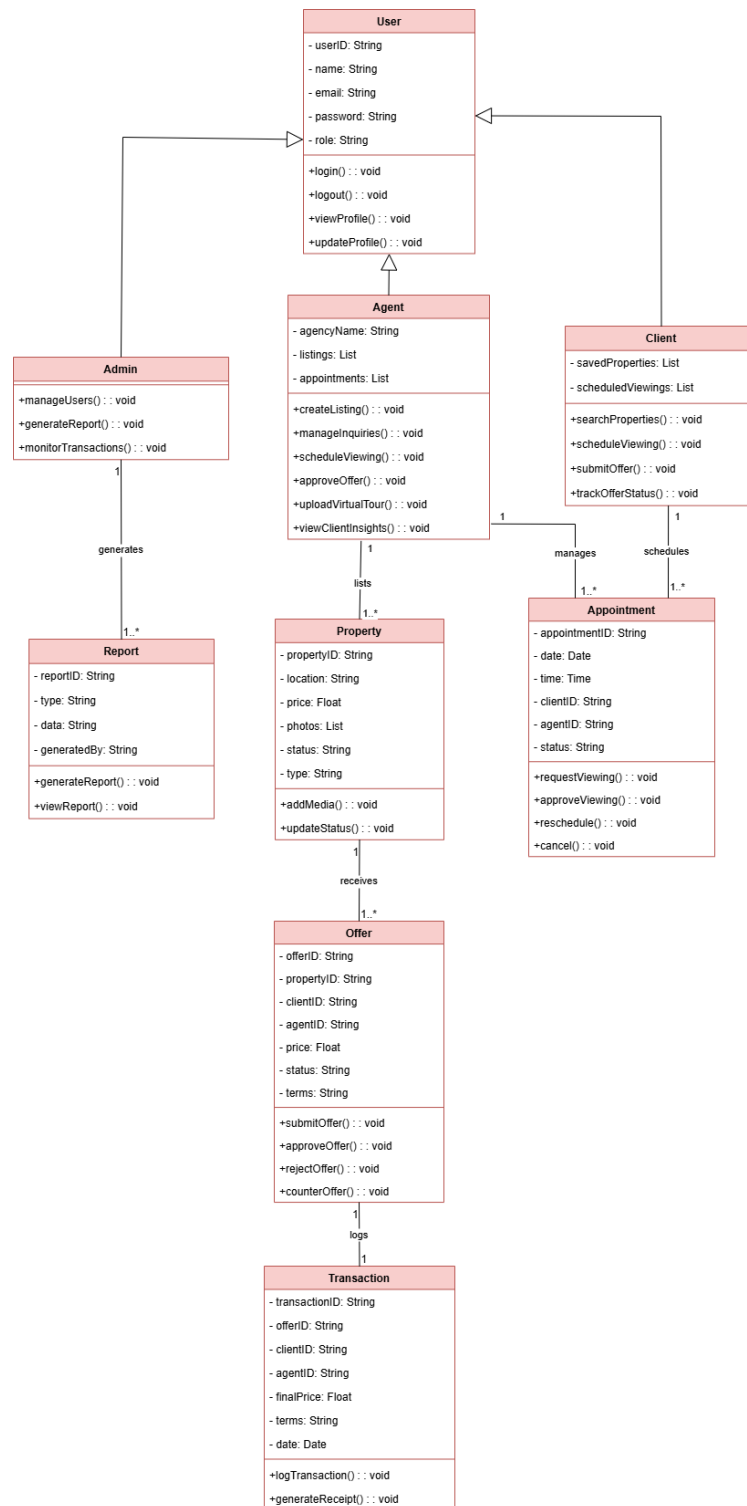


Figure 3 Class Diagram of Smart Real Estate Management System (SREMS)

Key Classes and Their Roles:

1. **User:**
 - **Attributes:**
 - userID, name, email, password, role: Basic user information and role designation (like admin, agent, or client).
 - **Methods:**
 - login(), logout(), viewProfile(), updateProfile(): Standard user operations to manage their account.
2. **Admin** (inherits from User):
 - **Methods:**
 - manageUsers(): Allows admin to manage user accounts.
 - generateReport(): Generates reports for the system.
 - monitorTransactions(): Tracks and monitors transactions in the system.
3. **Agent** (inherits from User):
 - **Attributes:**
 - agencyName, listings, appointments: Information related to the agent's agency and their property listings and appointments.
 - **Methods:**
 - createListing(), manageInquiries(), scheduleViewing(), approveOffer(), uploadVirtualTour(), viewClientInsights(): Functions to manage property listings, handle client inquiries, schedule viewings, approve offers, and track insights about clients.
4. **Client** (inherits from User):
 - **Attributes:**
 - savedProperties, scheduledViewings: Lists of saved properties and scheduled viewings for the client.
 - **Methods:**
 - searchProperties(), scheduleViewing(), submitOffer(), trackOfferStatus(): Functions for clients to search properties, schedule viewings, submit offers, and track their offers' statuses.
5. **Property:**
 - **Attributes:**
 - propertyID, location, price, status, type, etc.: Basic information about a property.
 - **Methods:**

- addMedia(), updateStatus(): Adds media (like photos) to the property listing and updates the property's status.

6. **Appointment:**

- **Attributes:**
 - appointmentID, date, time, clientID, agentID, status: Information about a scheduled appointment for viewing properties.
- **Methods:**
 - requestViewing(), approveViewing(), reschedule(), cancel(): Functions for managing appointment requests, approvals, rescheduling, and cancellation.

7. **Offer:**

- **Attributes:**
 - offerID, propertyID, clientID, agentID, price, status, terms: Details of an offer made on a property.
- **Methods:**
 - submitOffer(), approveOffer(), rejectOffer(), counterOffer(): Functions to submit, approve, reject, or counter an offer.

8. **Transaction:**

- **Attributes:**
 - transactionID, propertyID, clientID, agentID, finalPrice, terms: Information about a finalized transaction.
- **Methods:**
 - logTransaction(), generateReceipt(): Logs the transaction details and generates a receipt for the client.

9. **Report:**

- **Attributes:**
 - reportID, type, data, generatedBy: Report details.
- **Methods:**
 - generateReport(), viewReport(): Functions to generate and view reports.

Relationships Explained:

- **Inheritance:**
 - Admin, Agent, and Client classes inherit from User.
- **Associations:**
 - The Admin generates Report.

- Agent manages Property listings.
- Client schedules Appointment and submits Offer.
- Appointment involves a relationship between Client and Agent.
- Offer is linked to Transaction upon approval.
- Property is listed by Agent and can receive Offers from Client.

Artifact 06

Activity Diagram

Activity Diagram

An activity diagram is a type of UML (Unified Modeling Language) diagram that represents the flow of activities or actions within a system. It focuses on the sequence of actions, decision points, and parallel processes, making it useful for modeling workflows, business processes, or complex algorithms.

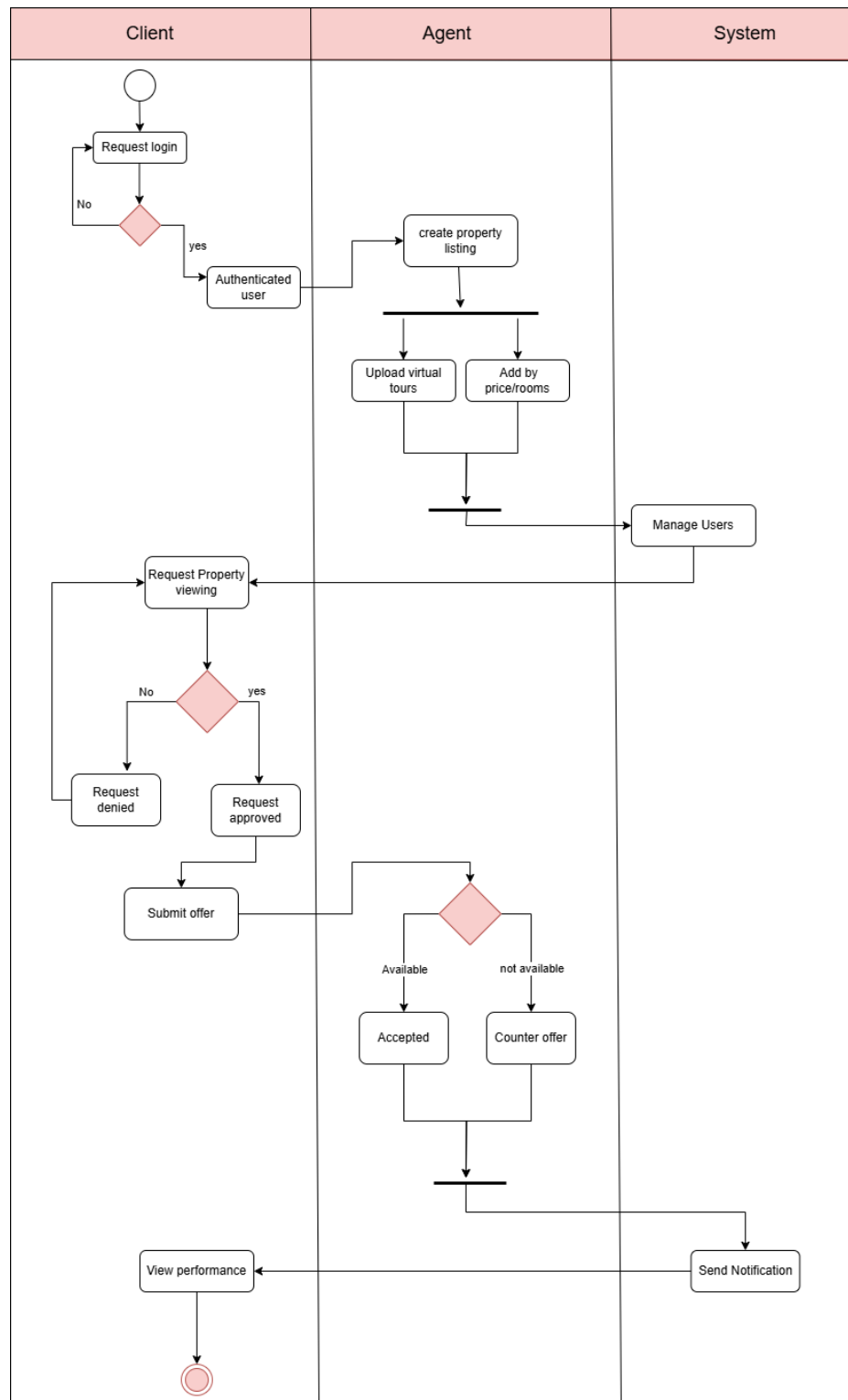


Figure 4 Activity Diagram of Smart Real Estate Management System (SREMS)

1. Login Process

- **Actors:** Client, System, Authentication Service
- **Flow:**
 - The **Client** initiates a login request to the **System**.
 - The **System** sends the login information to the **Authentication Service**.
 - An alt block checks the password:
 - If the password is correct, the **System** grants access, and the **Authentication Service** authenticates the user.
 - If the password is incorrect, access is denied, and authentication fails.

2. Property Listing Creation

- **Actors:** System, Authentication Service, Database
- **Flow:**
 - The **Agent** creates a new property listing in the **System**.
 - Property details are saved in the **Database**.

3. Property Viewing Request

- **Actors:** Client, System, Agent
- **Flow:**
 - The **Client** requests a property viewing through the **System**.
 - The **System** sends a viewing request to the **Agent**.
 - An alt block checks agent availability:
 - If the **Agent** is available, the viewing is scheduled, and the **Client** is notified.
 - If the **Agent** is unavailable, the **Client** is notified that the request was declined.

4. Offer Submission Process

- **Actors:** Client, System, Agent, Database
- **Flow:**
 - The **Client** submits a purchase offer via the **System**.
 - The **System** notifies the **Agent** of the new offer.
 - An alt block checks if the offer is accepted:
 - If accepted, the **Agent** approves the offer, transaction details are logged, and the **Client** is notified.
 - If rejected, the **Client** is notified of the rejection.

5. Property Performance Report Request

- **Actors:** Agent, Admin, Database
- **Flow:**
 - The **Agent** or **Admin** requests a property performance report.
 - The **Database** generates the report data.
 - The performance report is then sent back to the requester.

Artifact 07

Sequence Diagram

Sequence diagram

A **sequence diagram** is a type of UML (Unified Modeling Language) diagram that shows how objects interact in a particular scenario of a system. It visually represents the sequence of messages or interactions between objects over time, making it useful for understanding the dynamic behavior of a system.

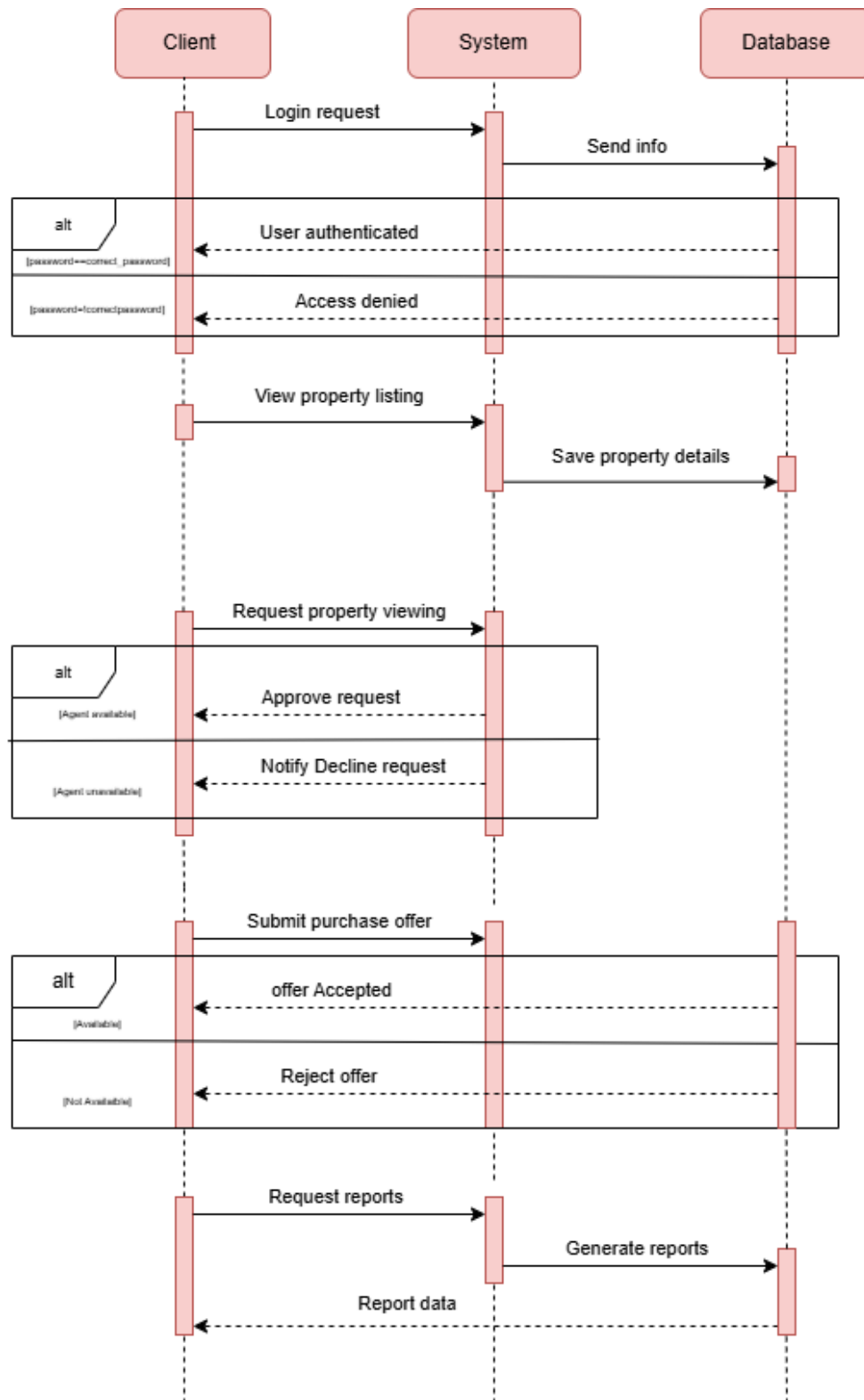


Figure 5 Sequence Diagram of Smart Real Estate Management System (SREMS)

1. User Login Process

- **Actors:** Client, System, Authentication Service
- **Flow:**
 - The **Client** sends a login request to the **System**.
 - The **System** forwards the login information to the **Authentication Service**.
 - An alt (alternative) block checks the login credentials:
 - If the password is correct, the **System** grants access, and the **Authentication Service** indicates that the user is authenticated.
 - If the password is incorrect, access is denied, and authentication fails.

2. Property Listing Creation

- **Actors:** System, Authentication Service, Database
- **Flow:**
 - The **Agent** initiates a property listing creation request in the **System**.
 - The **System** then saves the property details to the **Database**.

3. Property Viewing Request

- **Actors:** Client, System, Agent
- **Flow:**
 - The **Client** requests a property viewing through the **System**.
 - The **System** sends this request to the **Agent**.
 - An alt block checks the agent's availability:
 - If the **Agent** is available, the viewing is scheduled, and the **Client** is notified of the scheduled viewing.
 - If the **Agent** is unavailable, the **Client** is informed that the viewing request has been declined.

4. Offer Submission Process

- **Actors:** Client, System, Agent, Database
- **Flow:**
 - The **Client** submits a purchase offer via the **System**.
 - The **System** notifies the **Agent** about the new offer.
 - An alt block checks the offer status:
 - If the offer is accepted, the **Agent** approves it, the transaction details are logged in the **Database**, and the **Client** is informed that the offer was accepted.
 - If the offer is rejected, the **Client** is notified of the rejection.

5. Property Performance Report Request

- **Actors:** Agent, Admin, Database
- **Flow:**
 - The **Agent** or **Admin** requests a property performance report.
 - The **Database** processes this request and generates the report data.
 - The performance report is then sent back to the requester.

Artifact 08

State Transition Diagram

State Diagram:

A state diagram (also known as a state machine diagram or statechart) is a type of UML diagram that depicts the different states an object can be in throughout its lifecycle and the transitions between those states. It is particularly useful for modeling the behavior of objects that change state in response to external events.

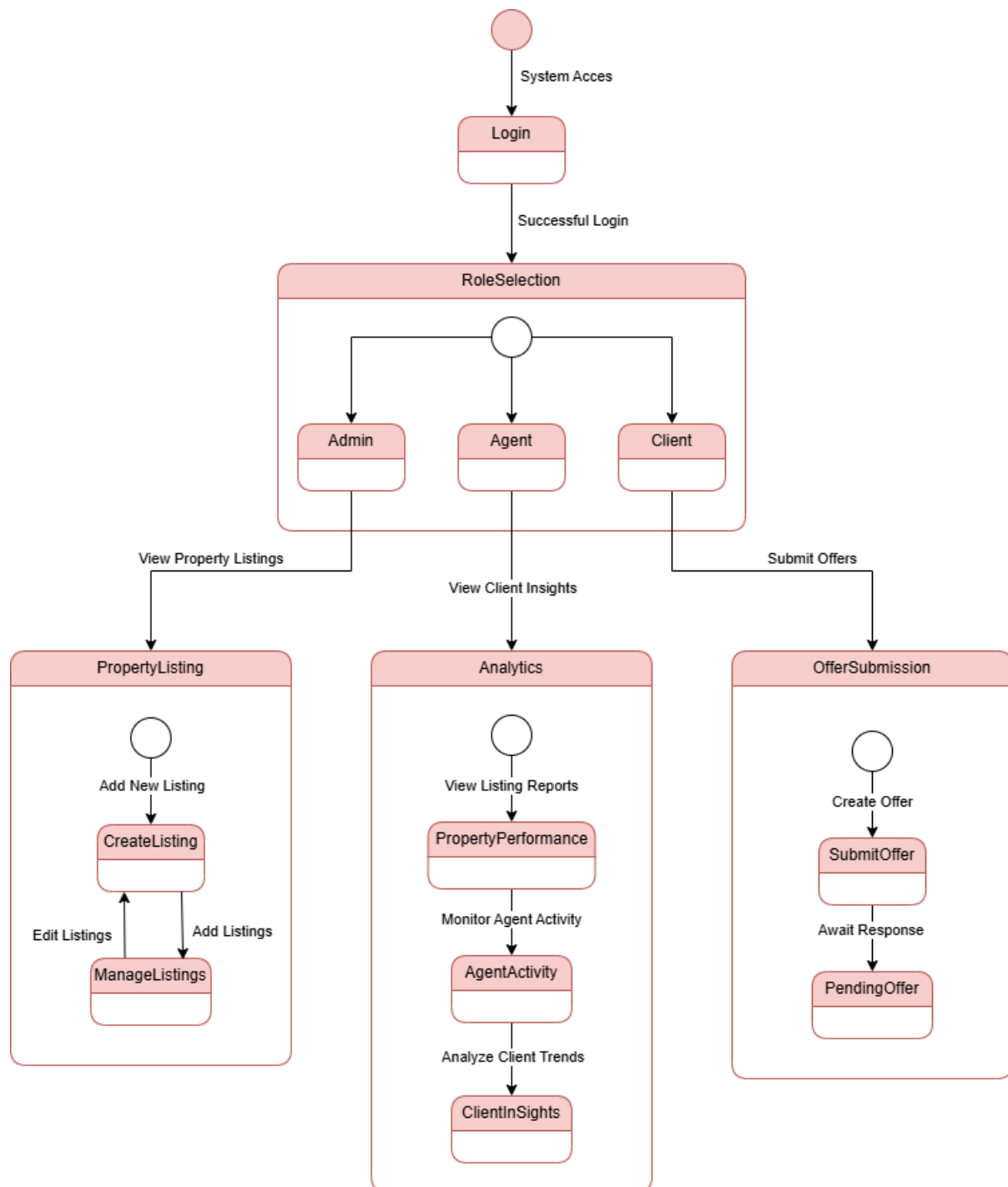


Figure 6 States Diagram of Smart Real Estate Management System (SREMS)

Key Classes and Their Roles:

1. Notification:

- **Attributes:** notificationID, recipientID, message, timestamp, status - details of notifications sent to users.
- **Methods:**
 - sendNotification(): Sends a notification.
 - viewNotifications(): Allows the user to view their notifications.

2. User:

- **Attributes:** userID, email, password, role - basic information common to all users.
- **Methods:**
 - login(), logout(), updateProfile(), changePassword(): Basic user operations for account management.

3. Admin (inherits from User):

- **Attributes:** adminID - identifier for the admin.
- **Methods:**
 - manageUsers(): Allows the admin to manage user accounts.
 - viewReports(), manageTransactions(), managePropertyListings(): Methods to manage reports, transactions, and property listings in the system.

4. Client (inherits from User):

- **Attributes:** clientID, preferredLocation - client-specific information.
- **Methods:**
 - searchProperties(), requestViewing(), submitOffer(), trackOfferStatus(): Methods for clients to search properties, request viewings, submit offers, and track their status.

5. Agent (inherits from User):

- **Attributes:** agentID, listProperty() - agent-specific details.
- **Methods:**
 - manageProperty(), scheduleViewing(), respondToInquiry(), approveOffer(), uploadVirtualTour(): Methods to manage property listings, schedule viewings, respond to client inquiries, approve offers, and upload virtual tours.

6. Property:

- **Attributes:** propertyID, address, price, location, rooms, status, media - information related to each property.
- **Methods:**

- `updateStatus()`, `addMedia()`, `filterProperties()`: Functions to update the property's status, add media, and filter properties based on certain criteria.

7. Viewing:

- **Attributes:** `viewingID`, `propertyID`, `clientID`, `agentID`, `viewingDate`, `status` - details of property viewings.
- **Methods:**
 - `scheduleViewing()`, `approveReschedule()`, `sendReminder()`: Functions for scheduling, rescheduling, and sending reminders for viewings.

8. Offer:

- **Attributes:** `offerID`, `propertyID`, `clientID`, `agentID`, `amount`, `offerDate`, `status` - details of offers made on properties.
- **Methods:**
 - `submitOffer()`, `updateStatus()`, `counterOffer()`: Functions to submit, update, or counter an offer.

9. Transaction:

- **Attributes:** `transactionID`, `offerID`, `clientID`, `agentID`, `transactionDate`, `amount`, `terms` - information about a transaction.
- **Methods:**
 - `recordTransaction()`, `generateReceipt()`: Records a transaction and generates a receipt.

Relationships Explained:

- **Inheritance:** Admin, Client, and Agent are all types of Users.
- **Associations:**
 - Notification is sent to User.
 - Admin manages Property.
 - Client searches for Property, requests Viewing, and submits Offer.
 - Agent schedules Viewing, submits Offer, and manages Property.
 - Viewing involves both Client and Agent.
 - Offer leads to Transaction upon acceptance.
 - Property can have multiple Viewings and Offers.

Artifact 09

MVC Framework

Model-View-Controller (MVC) Architecture

The **Model-View-Controller (MVC)** is a software architectural pattern that divides an application into three main components: **Model**, **View**, and **Controller**. This separation of concerns provides a modular structure, enabling each component to be developed, maintained, and tested independently, ultimately leading to cleaner and more maintainable code. MVC is especially useful for applications where data needs to be managed and displayed interactively, as it isolates business logic from the user interface.

Components of MVC

1. Model

The **Model** component is responsible for the application's data and business logic. It represents the core functionality of the application by managing data storage, retrieval, and any operations or rules associated with data processing. The Model communicates directly with the data source or database, performing operations like create, read, update, and delete (CRUD).

- **Responsibilities:**
 - Defines and manages application data and business logic.
 - Encapsulates all data-related operations, including CRUD operations.
 - Notifies the View of data changes when necessary to keep the user interface updated.
- **Model Components in This Application:**
 - **User:** Represents system users, containing information such as username, password, and role (Admin, Agent, or Client). The User model provides methods for checking user roles and managing user details.

Code:

```
1 package model;
2
3 public class User {
4     private String username;
5     private String password;
6     private String role;
7
8     // Constructor with default password
9     public User(String username, String role) {
10         this.username = username;
11         this.password = "defaultPassword"; // Set a default password
12         this.role = role;
13     }
14
15     // Constructor with username, password, and role
16     public User(String username, String password, String role) {
17         this.username = username;
18         this.password = password;
19         this.role = role;
20     }
21
22     // Getter methods
23     public String getUsername() {
24         return username;
25     }
26
27     public String getPassword() {
28         return password;
29     }
30
31     public String getRole() {
32         return role;
33     }
34
35     // Role-based access control
36     public boolean isAdmin() {
37         return role.equalsIgnoreCase("Admin");
38     }
39
40     public boolean isAgent() {
41         return role.equalsIgnoreCase("Agent");
42     }
43 }
```

- **Property:** Represents a real estate listing, containing fields like ID, title, location, price, and status (e.g., Available, Under Contract, Sold). This model manages all property-related data.

Code:


```

Property.java
src > model > Property.java
You, 12 hours ago | 1 author (You) | Codium Refactor | Explain
1 public class Property {
2     private String id;
3     private String title;
4     private String location;
5     private double price;
6     private String status; // Available, Under Contract, Sold
7
8     public Property(String id, String title, String location, double price) {
9         this.id = id;
10        this.title = title;
11        this.location = location;
12        this.price = price;
13        this.status = "Available"; // Default status when created
14    }
15
16    // Getters and Setters
17    Codium Refactor | Explain | X
18    public String getId() {
19        return id;
20    }
21
22    Codium Refactor | Explain | Generate Javadoc | X
23    public String getTitle() {
24        return title;
25    }
26
27    Codium Refactor | Explain | Generate Javadoc | X
28    public String getLocation() {
29        return location;
30    }
31
32    Codium Refactor | Explain | Generate Javadoc | X
33    public double getPrice() {
34        return price;
35    }
36
37    Codium Refactor | Explain | Generate Javadoc | X
38    public String getStatus() {
39        return status;
40    }
41
42    Codium Refactor | Explain | Generate Javadoc | X
43    public void setStatus(String status) {
44        this.status = status;
45    }
46
47 }

```

- **Offer:** Represents a client's offer on a property. It stores the client making the offer, the property being offered on, and the amount of the offer.

Code:

```

Offer.java
src > model > Offer.java
You, 12 hours ago | 1 author (You) | Codium Refactor | Explain
1 package model;
2
3 public class Offer {
4     private User client; // Client making the offer
5     private Property property; // Property the offer is for
6     private double amount; // Offer amount
7
8     public Offer(User client, Property property, double amount) {
9         this.client = client;
10        this.property = property;
11        this.amount = amount;
12    }
13
14    // Getters and Setters
15    Codium Refactor | Explain | X
16    public User getClient() {
17        return client;
18    }
19
20    Codium Refactor | Explain | Generate Javadoc | X
21    public Property getProperty() {
22        return property;
23    }
24
25    Codium Refactor | Explain | Generate Javadoc | X
26    public double getAmount() {
27        return amount;
28    }
29
30    Codium Refactor | Explain | Generate Javadoc | X
31    public void setAmount(double amount) {
32        this.amount = amount;
33    }
34
35    Codium Refactor | Explain | Generate Javadoc | X
36    @Override
37    public String toString() {
38        return "Offer [Client=" + client.getUsername() + ", Property=" + property.getTitle() + ", Amount=" + amount + " ]";
39    }
40
41 }

```

2. View

The **View** component is responsible for presenting data to the user and capturing user input. It manages the user interface, displaying information provided by the Model and allowing the user to interact with the application. The View does not contain any business logic; instead, it purely reflects the data it receives and gathers input from the user.

- **Responsibilities:**
 - Renders data from the Model for the user to view.
 - Collects user inputs and relays them to the Controller.
 - Updates the user interface based on interactions managed by the Controller.
- **View Component in This Application:**
 - **View:** This class provides methods to display messages and prompt the user for input. Using the Scanner, it handles basic text-based interaction with the user. The View class

is focused solely on communication, leaving all business logic to the Controller and Model.

Code:

```
View.java X
src > view > View.java > ...
1 package view;
2
3 import java.util.Scanner;
4
5 You, 12 hours ago | 1 author (You) | Codeium: Refactor | Explain
6 public class View {
7     private Scanner scanner;
8
9     public View() {
10         scanner = new Scanner(System.in);
11     }
12
13     // Display a message to the user
14     public void displayMessage(String message) {
15         System.out.println(message);
16     }
17
18     // Get input from the user
19     public String getInput(String prompt) {
20         System.out.print(prompt + " ");
21         return scanner.nextLine();
22     }
23 }
```

3. Controller

The **Controller** component acts as an intermediary between the Model and the View. It listens for user inputs from the View, processes them, interacts with the Model as needed, and updates the View accordingly. Controllers contain application-specific logic and determine how the application responds to various user actions.

- **Responsibilities:**
 - Receives and processes user inputs from the View.
 - Interacts with the Model to execute data operations and handle business logic.
 - Updates the View to reflect any data changes or responses from the Model.
- **Controller Components in This Application:**
 - **Admin Controller:** Manages admin-specific tasks, including managing users and properties. This controller bridges interactions between the admin user, property data, and user management logic.

Code:

```
AdminController.java M X
src > controller > AdminController.java > AdminController > manageUsers()
1 package controller;
2 import model.User;
3 import model.Property;
4 import view.View;
5 import java.util.ArrayList;
6 import java.util.List;
7
8 You, 12 hours ago | 1 author (You) | Codeium: Refactor | Explain
9 public class AdminController extends UserController {
10     private List<User> users; // List to manage all users
11     private List<Property> properties; // List to store all properties
12
13     public AdminController(User user, View view) {
14         super(user, view);
15         this.users = new ArrayList<>();
16         this.properties = new ArrayList<>();
17     }
18
19     @Override
20     public void start() {
21         super.start();
22         view.displayMessage(message:"Admin Dashboard");
23         boolean running = true;
24         while (running) {
25             view.displayMessage(message:"1. Manage Users");
26             view.displayMessage(message:"2. View All Properties");
27             view.displayMessage(message:"3. Exit");
28             String choice = view.getInput(prompt:"Choose an action:");
29             switch (choice) {
30                 case "1":
31                     manageUsers();
32                     break;
33                 case "2":
34                     viewAllProperties();
35                     break;
36                 case "3":
37                     running = false;
38                     break;
39                 default:
40                     view.displayMessage(message:"Invalid choice. Please try again.");
41             }
42         }
43     }
44 }
```

Output:

```
Enter Username:
admin
Enter Password:
admin123
Welcome, admin!
Admin Dashboard
1. Manage Users
2. View All Properties
3. Exit
Choose an action: █
```

- **Agent Controller:** Allows agents to manage property listings, view client offers, and interact with property data. This controller facilitates agent-related property transactions.

Code:

```
AdminController.java M AgentController.java X
src > controller > AgentController.java > AgentController > start()
Now, 12 hours ago | 1 author (You) | Codeium: Refactor | Explain
1 package controller;
2
3 import model.User;
4 import model.Property;
5 import model.Offer;
6 import view.View;
7
8 import java.util.ArrayList;
9 import java.util.List;
10
11 Now, 12 hours ago | 1 author (You) | Codeium: Refactor | Explain
12 public class AgentController {
13     private User user;
14     private View view;
15     private List<Property> properties; // List of properties the agent is managing
16     private List<Offer> offers; // List of offers from clients
17
18     public AgentController(User user, View view) {
19         this.user = user;
20         this.view = view;
21         this.properties = new ArrayList<>();
22         this.offers = new ArrayList<>();
23     }
24
25     Codeium: Refactor | Explain | Generate Javadoc | X
26     public void start() {
27         view.displayMessage("Welcome Agent: " + user.getUsername());
28         boolean running = true;
29         while (running) {
30             view.displayMessage(message:"1. List New Property");
31             view.displayMessage(message:"2. View Properties");
32             view.displayMessage(message:"3. Manage Property Viewings");
33             view.displayMessage(message:"4. Respond to Offers");
34             view.displayMessage(message:"5. Exit");
35             String choice = view.getInput(prompt:"Choose an action:");
36             switch (choice) {
37                 case "1":
38                     listNewProperty();
39                     break;
40                 case "2":
41                     viewProperties();
42                     break;
43                 case "3":
44                     manageViewings();
45                     break;
46             }
47         }
48     }
49 }
```

Output:

```
Enter Username:
agent
Enter Password:
agent123
Welcome Agent: agent
1. List New Property
2. View Properties
3. Manage Property Viewings
4. Respond to Offers
5. Exit
Choose an action: █
```

- **Client Controller:** Manages client interactions, such as browsing available properties and making offers. It handles requests from the client side, updating the View with relevant property information.

Code:

```
ClientController.java
src > controller > ClientController.java > ClientController > start()
You, 12 hours ago | 1 author (You)
1 package controller;
2
3 import model.User;
4 import view.View;
5 import model.Property;
6 import model.Offer;
7
8 import java.util.ArrayList;
9 import java.util.List;
10
11 You, 12 hours ago | 1 author (You) | Codium: Refactor | Explain
12 public class ClientController extends UserController {
13     private List<Property> properties;
14     private List<Offer> offers;
15
16     public ClientController(User user, View view) {
17         super(user, view);
18         this.properties = new ArrayList<>();
19         this.offers = new ArrayList<>();
20         // mock properties for browsing
21         populateMockProperties();
22     }
23
24 Codium: Refactor | Explain | Generate Javadoc | X
25 @Override
26 public void start() {
27     super.start();
28     view.displayMessage(message: "Client Dashboard");
29     boolean running = true;
30     while (running) {
31         view.displayMessage(message: "1. Browse Properties");
32         view.displayMessage(message: "2. Schedule Property Viewing");
33         view.displayMessage(message: "3. Submit Offer");
34         view.displayMessage(message: "4. View Offers");
35         view.displayMessage(message: "5. Exit");
36         String choice = view.getInput(prompt: "Choose an action:");
37         switch (choice) {
38             case "1":
39                 browseProperties();
40                 break;
41             case "2":
42                 scheduleViewing();
43                 break;
44             case "3":
45                 submitOffer();
46                 break;
47         }
48     }
49 }
```

Output:

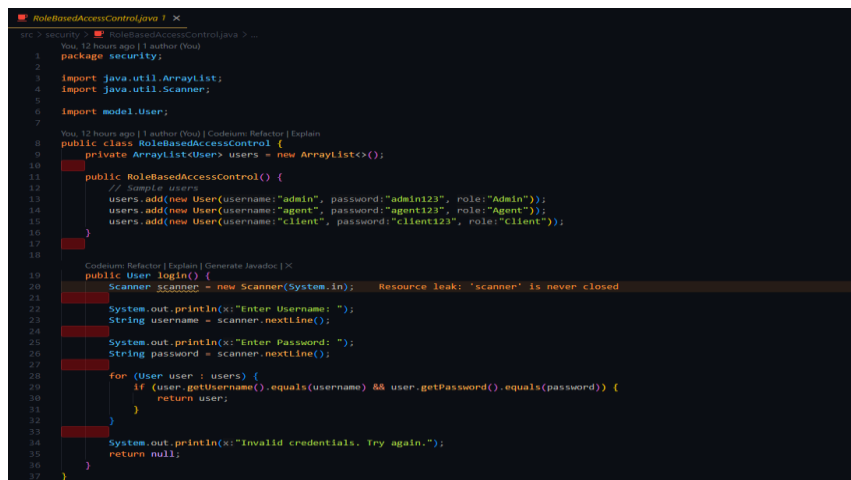
```
Enter Username:
client
Enter Password:
client123
Welcome, client!
Client Dashboard
1. Browse Properties
2. Schedule Property Viewing
3. Submit Offer
4. View Offers
5. Exit
Choose an action: |
```

How MVC Works Together

In the MVC architecture, each component plays a unique role, working together to provide a seamless user experience:

1. **User Interaction:** The user interacts with the View (e.g., entering login credentials).
2. **Controller Processing:** The View passes this input to the Controller (e.g., the RoleBasedAccessControl class manages login).

Code:



```
1 package security;
2
3 import java.util.ArrayList;
4 import java.util.Scanner;
5
6 import model.User;
7
8 public class RoleBasedAccessControl {
9     private ArrayList<User> users = new ArrayList<>();
10
11     public RoleBasedAccessControl() {
12         // Sample users
13         users.add(new User(username:"admin", password:"admin123", role:"Admin"));
14         users.add(new User(username:"agent", password:"agent123", role:"Agent"));
15         users.add(new User(username:"client", password:"client123", role:"Client"));
16     }
17
18     public User login() {
19         Scanner scanner = new Scanner(System.in); // Resource leak: 'scanner' is never closed
20
21         System.out.println("Enter Username: ");
22         String username = scanner.nextLine();
23
24         System.out.println("Enter Password: ");
25         String password = scanner.nextLine();
26
27         for (User user : users) {
28             if (user.getUsername().equals(username) && user.getPassword().equals(password)) {
29                 return user;
30             }
31         }
32
33         System.out.println("Invalid credentials. Try again.");
34         return null;
35     }
36 }
```

3. **Model Interaction:** The Controller processes the input, interacts with the Model to verify user credentials or retrieve data.
4. **View Update:** The Model responds with updated data, which is then displayed to the user by updating the View accordingly.

Application-Specific MVC Workflow

- **Role-Based Access Control (RBAC):** The RoleBasedAccessControl class manages user authentication, allowing the App class to connect users with their appropriate permissions and views.
- **App Class (Main Application Entry):**
 - Initializes RoleBasedAccessControl and View.
 - Manages the login process and loads the appropriate controller based on the user's role (Admin, Agent, or Client).
 - Each controller then communicates with the View and the Model to execute role-specific tasks, such as property management or offer handling.

Benefits of Using MVC in This Application

- **Separation of Concerns:** MVC separates the Model, View, and Controller, promoting organized code and allowing each component to manage a specific aspect of the application.
- **Modularity:** Each component can be independently modified, making it easy to add new features or change parts of the application without affecting other components.
- **Reusability:** The individual components are reusable in different contexts or applications, such as utilizing the AdminController logic for a different admin interface.

By implementing MVC, this application achieves a clear, modular structure that simplifies user management, property listings, and offer submissions, enabling an organized and maintainable codebase.

Artifact 10

GRASP Patterns

GRASP

GRASP (General Responsibility Assignment Software Patterns) is a set of guidelines for assigning responsibilities to classes and objects in object-oriented design. It helps designers create maintainable, understandable, and robust systems by promoting good object-oriented practices. These patterns were introduced by Craig Larman in his book "*Applying UML and Patterns*".

Core GRASP Patterns

1. Creator:

Assign the responsibility of creating instances of a class to the class that has the most knowledge about when and how to create them.

- **Example:** The Agent class will be responsible for creating property listings.

```
//.....CREATOR.....//
Codeium: Refactor | Explain
public class Agent {
    Codeium: Refactor | Explain | Generate Javadoc | X
    public Property createProperty(String price, String location, String description) {
        return new Property(price, location, description);
    }
}
```

2. Information Expert:

Assign a responsibility to the class that has the necessary information to fulfill it, promoting high cohesion and minimizing coupling.

- **Example:** The Property class will manage its own state, such as price, location, and status.

```
//.....INFORMATION EXPERT.....//
Codeium: Refactor | Explain
public class Property {
    private String price;
    private String location;
    private String status;

    public Property(String price, String location, String status) {
        this.price = price;
        this.location = location;
        this.status = status;
    }

    Codeium: Refactor | Explain | Generate Javadoc | X
    public String getPrice() {
        return price;
    }

    Codeium: Refactor | Explain | Generate Javadoc | X
    public void setPrice(String price) {
        this.price = price;
    }

    Codeium: Refactor | Explain | Generate Javadoc | X
    public String getLocation() {
        return location;
    }
}
```

```

Codeium: Refactor | Explain | Generate Javadoc | X
public void setLocation(String location) {
    this.location = location;
}

Codeium: Refactor | Explain | Generate Javadoc | X
public String getStatus() {
    return status;
}

Codeium: Refactor | Explain | Generate Javadoc | X
public void setStatus(String status) {
    this.status = status;
}

Codeium: Refactor | Explain | Generate Javadoc | X
public void updateStatus(String newStatus) {
    this.status = newStatus;
}

Codeium: Refactor | Explain | Generate Javadoc | X
public boolean isAvailable() {
    return this.status.equals("Available");
}

```

3. Low Coupling:

Aim for classes to have minimal dependencies on each other, facilitating easier maintenance and flexibility in the system.

- **Example:** Decouple the Offer and Transaction classes. The Transaction doesn't need to know the details of the Offer.

```

//.....LOW COUPLING.....//
Codeium: Refactor | Explain
public class Offer {
    private Property property;
    private Client client;
    private String offerPrice;

    public Offer(Property property, Client client, String offerPrice) {
        this.property = property;
        this.client = client;
        this.offerPrice = offerPrice;
    }

    Codeium: Refactor | Explain | Generate Javadoc | X
    public Property getProperty() {
        return property;
    }

    Codeium: Refactor | Explain | Generate Javadoc | X
    public void setProperty(Property property) {
        this.property = property;
    }

    Codeium: Refactor | Explain | Generate Javadoc | X
    public Client getClient() {
        return client;
    }

    Codeium: Refactor | Explain | Generate Javadoc | X
    public void setClient(Client client) {
        this.client = client;
    }

    Codeium: Refactor | Explain | Generate Javadoc | X
    public String getOfferPrice() {
        return offerPrice;
    }

    Codeium: Refactor | Explain | Generate Javadoc | X
    public void setOfferPrice(String offerPrice) {
        this.offerPrice = offerPrice;
    }
}

```



```

Codeium: Refactor | Explain
public class Transaction {
    private Offer acceptedOffer;
    private String transactionDetails;

    public Transaction(Offer acceptedOffer, String transactionDetails) {
        this.acceptedOffer = acceptedOffer;
        this.transactionDetails = transactionDetails;
    }

    Codeium: Refactor | Explain | Generate Javadoc | X
    public Offer getAcceptedOffer() {
        return acceptedOffer;
    }

    Codeium: Refactor | Explain | Generate Javadoc | X
    public void setAcceptedOffer(Offer acceptedOffer) {
        this.acceptedOffer = acceptedOffer;
    }

    Codeium: Refactor | Explain | Generate Javadoc | X
    public String getTransactionDetails() {
        return transactionDetails;
    }
}

Codeium: Refactor | Explain | Generate Javadoc | X
public void setTransactionDetails(String transactionDetails) {
    this.transactionDetails = transactionDetails;
}

Codeium: Refactor | Explain | Generate Javadoc | X
public void createTransaction(Offer offer) {
    this.acceptedOffer = offer;
    this.transactionDetails = "Transaction created for offer price: " + offer.getOfferPrice();
}
}

```

4. High Cohesion:

Ensure that the responsibilities within a class are closely related and focused, enhancing readability, maintainability, and reusability.

- **Example:** The TransactionManager class is responsible for handling transaction-related activities only.

```

//.....HIGH COHESION.....//
Codeium: Refactor | Explain
public class TransactionManager {

    private PaymentProcessor paymentProcessor;
    private OfferManager offerManager;
    private NotificationService notificationService;

    public TransactionManager(PaymentProcessor paymentProcessor, OfferManager offerManager, NotificationService notificationService) {
        this.paymentProcessor = paymentProcessor;
        this.offerManager = offerManager;
        this.notificationService = notificationService;
    }

    Codeium: Refactor | Explain | Generate Javadoc | X
    public void processTransaction(Offer offer) {
        if (offer == null) {
            throw new IllegalArgumentException("Offer cannot be null");
        }

        if (!offerManager.isValidOffer(offer)) {
            System.out.println("Offer is not valid.");
            return;
        }

        boolean paymentSuccess = paymentProcessor.processPayment(offer.getBuyer(), offer.getPaymentDetails());

        if (!paymentSuccess) {
            System.out.println("Payment failed. Transaction aborted.");
            return;
        }

        boolean transactionSuccess = offerManager.completeTransaction(offer);

        if (!transactionSuccess) {
            System.out.println("Transaction failed during finalization.");
            return;
        }

        notificationService.notifyBuyer(offer.getBuyer());
        notificationService.notifySeller(offer.getSeller());

        System.out.println("Transaction completed successfully for offer: " + offer.getId());
    }
}

```

5. Controller:

Assign the responsibility of handling system events or coordinating activities to a controller class, promoting centralized control and avoiding cluttered classes.

- **Example:** A ViewingsController handles client viewing requests.

```
//.....CONTROLLER.....//
Codeium: Refactor | Explain
public class ViewingsController {
    private Agent agent;
    private ViewingScheduler viewingScheduler;
    private NotificationService notificationService;

    public ViewingsController(Agent agent, ViewingScheduler viewingScheduler, NotificationService notificationService) {
        this.agent = agent;
        this.viewingScheduler = viewingScheduler;
        this.notificationService = notificationService;
    }

    Codeium: Refactor | Explain | Generate Javadoc | X
    public void requestViewing(Client client, Property property, String time) {
        if (client == null || property == null || time == null) {
            throw new IllegalArgumentException("Client, property, and time must not be null");
        }

        if (!agent.isAvailable(time)) {
            System.out.println("Agent is not available at the requested time.");
            return;
        }

        ViewingRequest request = viewingScheduler.scheduleViewing(client, property, time);

        if (request == null) {
            System.out.println("Unable to schedule the viewing. Please try again.");
            return;
        }

        notificationService.notifyClient(client, request);
        notificationService.notifyAgent(agent, request);

        System.out.println("Viewing scheduled successfully.");
    }
}
```

```
Codeium: Refactor | Explain | Generate Javadoc | X
public void approveViewing(ViewingRequest request) {
    if (request == null) {
        throw new IllegalArgumentException("Viewing request cannot be null");
    }

    if (!viewingScheduler.isViewingScheduled(request)) {
        System.out.println("Viewing request is not valid.");
        return;
    }

    boolean isApproved = viewingScheduler.approveRequest(request);

    if (!isApproved) {
        System.out.println("Viewing request could not be approved.");
        return;
    }

    notificationService.notifyClient(request.getClient(), "Your viewing has been approved.");
    notificationService.notifyAgent(request.getAgent(), "Viewing request approved for property: " + request.getProperty().getId());

    System.out.println("Viewing request approved.");
}
}
```

6. Pure Fabrication:

Introduce new classes to fulfill responsibilities without violating cohesion and coupling principles, promoting cleaner and more maintainable designs.

- **Example:** NotificationService could be a pure fabrication to handle notifications for viewings and offers.

```
//.....PURE FABRICATION.....//
Codeium: Refactor | Explain
public class NotificationService {

    private EmailService emailService;

    public NotificationService(EmailService emailService) {
        this.emailService = emailService;
    }

    Codeium: Refactor | Explain | Generate Javadoc | X
    public void sendViewingReminder(Client client, Property property) {
        if (client == null || property == null) {
            throw new IllegalArgumentException("Client and property must not be null");
        }

        String subject = "Reminder: Property Viewing Scheduled";
        String body = "Dear " + client.getName() + ",\n\n"
            + "This is a reminder that you have a viewing scheduled for the property located at "
            + property.getAddress() + " on " + property.getViewingTime() + ".\n\n"
            + "We look forward to your visit.\n\n"
            + "Best regards,\nReal Estate Agency";

        emailService.sendEmail(client.getEmail(), subject, body);
        System.out.println("Viewing reminder sent to " + client.getName());
    }

    Codeium: Refactor | Explain | Generate Javadoc | X
    public void sendOfferUpdate(Client client, String status) {
        if (client == null || status == null) {
            throw new IllegalArgumentException("Client and status must not be null");
        }

        String subject = "Offer Status Update";
        String body = "Dear " + client.getName() + ",\n\n"
            + "We would like to inform you that your offer for the property has been " + status + ".\n\n"
            + "Thank you for your interest in our properties.\n\n"
            + "Best regards,\nReal Estate Agency";

        emailService.sendEmail(client.getEmail(), subject, body);
        System.out.println("Offer status update sent to " + client.getName());
    }
}
```

7. Indirection:

Use intermediaries or abstractions to decouple classes and promote flexibility in design.

- **Example:** Use an AppointmentScheduler to handle the scheduling process rather than having clients and agents communicate directly.

```
//.....INDIRECTION.....//
Codeium: Refactor | Explain
public class AppointmentScheduler {

    private Agent agent;
    private PropertyManager propertyManager;
    private NotificationService notificationService;

    public AppointmentScheduler(Agent agent, PropertyManager propertyManager, NotificationService notificationService) {
        this.agent = agent;
        this.propertyManager = propertyManager;
        this.notificationService = notificationService;
    }

    Codeium: Refactor | Explain | Generate Javadoc | X
    public void scheduleViewing(Client client, Property property, String time) {
        if (client == null || property == null || time == null) {
            throw new IllegalArgumentException("Client, property, and time must not be null");
        }

        if (!agent.isAvailable(time)) {
            System.out.println("Agent is not available at the requested time.");
            return;
        }

        if (!propertyManager.isAvailable(property, time)) {
            System.out.println("The property is not available for viewing at this time.");
            return;
        }

        ViewingRequest request = new ViewingRequest(client, property, time, agent);
        boolean isScheduled = propertyManager.scheduleViewing(request);

        if (!isScheduled) {
            System.out.println("Viewing could not be scheduled. Please try again.");
            return;
        }

        notificationService.sendViewingReminder(client, property);

        System.out.println("Viewing scheduled successfully for " + client.getName() + " at " + time);
    }
}
```

8. Polymorphism:

Utilize inheritance and interfaces to enable multiple implementations of behaviors, allowing for flexible and extensible systems.

- **Example:** Handle various types of notifications using polymorphism.

```
//.....POLYMORPHISM.....//
Codeium: Refactor | Explain
public abstract class Notification {
    protected String recipient;
    protected String message;

    public Notification(String recipient, String message) {
        this.recipient = recipient;
        this.message = message;
    }

    public abstract void send();
}

Codeium: Refactor | Explain
public class EmailNotification extends Notification {

    public EmailNotification(String recipient, String message) {
        super(recipient, message);
    }

    Codeium: Refactor | Explain | Generate Javadoc | X
    @Override
    public void send() {
        System.out.println("Sending email to " + recipient + " with message: " + message);
    }
}
```

```
Codeium: Refactor | Explain
public class SMSNotification extends Notification {

    public SMSNotification(String recipient, String message) {
        super(recipient, message);
    }

    Codeium: Refactor | Explain | Generate Javadoc | X
    @Override
    public void send() {
        System.out.println("Sending SMS to " + recipient + " with message: " + message);
    }
}

Codeium: Refactor | Explain
public class NotificationSender {
    Codeium: Refactor | Explain | Generate Javadoc | X
    public void sendNotification(Notification notification) {
        if (notification == null) {
            throw new IllegalArgumentException("Notification cannot be null");
        }
        notification.send();
    }
}
```