Object Oriented Programming (CSC241)

Assignment 03

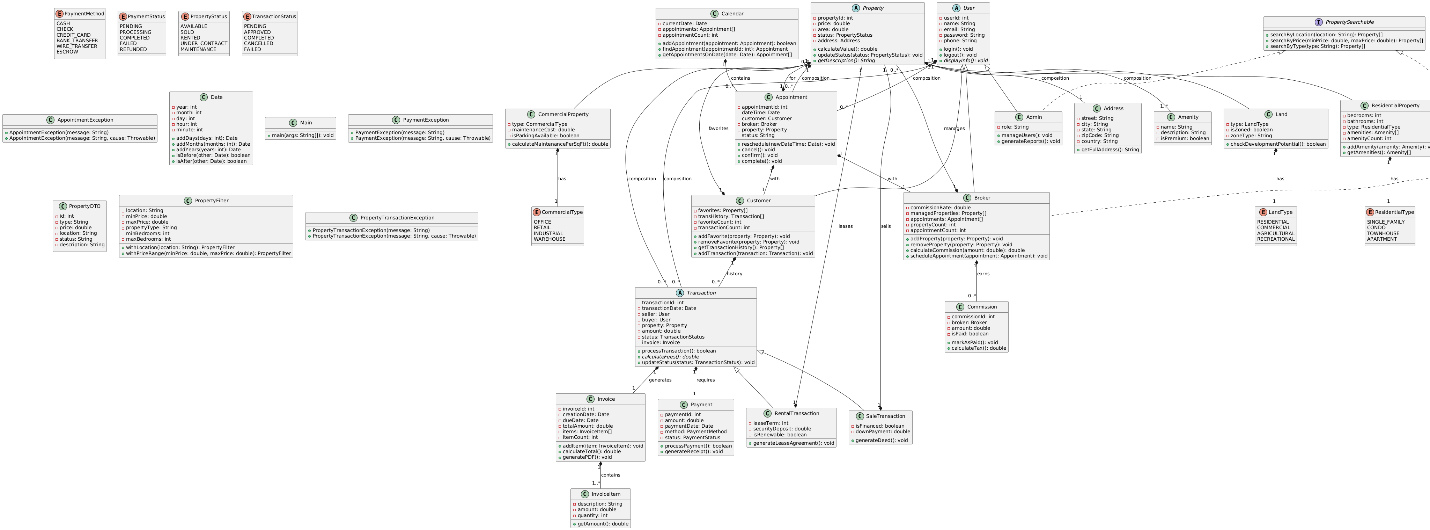
|  |  |
| --- | --- |
| Submitted to: | Mr. Muhammad Shahid Bhatti |
| Submitted by: | Muhammad Luqman (CIIT/FA24-BCS-074/LHR)  Ahmed Mohsin Nazir (CIIT/FA24-BCS-143/LHR) |
| Date: | May 15th, 2025 |

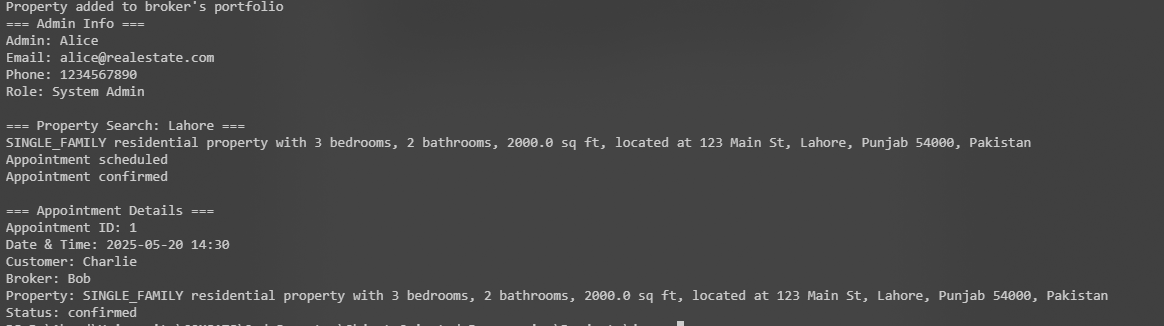
Project Title: Real Estate Management System

**Abstract:**

Our semester project is a Real Estate and Asset Management System designed to streamline the process of buying, selling, and renting properties. The system will enable users/customers to register, post property listings, and search for properties based on a variety of filters. The system will also include roles for admins, brokers, and employees. Brokers and employees will have access to manage listings with filters such as location, price, and type. Admins will oversee the platform, ensuring proper listing approvals and user management. The system will include a variety of property-related classes (e.g. residential, commercial, land) to demonstrate the use of object oriented principles such as inheritance, encapsulation, and polymorphism. It will also feature financial functionalities such as payment tracking, commission calculations for brokers, and cost estimations. Overall, the system is designed to be modular, scalable, and efficient, offering a complete real estate management solution.

**UML:**



**Output Screenshot:**

**Code:**

Address:

public class Address // Composition with Property

{

    private String street;

    private String city;

    private String state;

    private String zipCode;

    private String country;

    public Address(String street, String city, String state, String zipCode, String country)

    {

        setStreet(street);

        setCity(city);

        setState(state);

        setZipCode(zipCode);

        setCountry(country);

    }

    public String getFullAddress()

    {

        return street + ", " + city + ", " + state + " " + zipCode + ", " + country;

    }

    public String getStreet()

    {

        return street;

    }

    public void setStreet(String street)

    {

        this.street = street;

    }

    public String getCity()

    {

        return city;

    }

    public void setCity(String city)

    {

        this.city = city;

    }

    public String getState()

    {

        return state;

    }

    public void setState(String state)

    {

        this.state = state;

    }

    public String getZipCode()

    {

        return zipCode;

    }

    public void setZipCode(String zipCode)

    {

        this.zipCode = zipCode;

    }

    public String getCountry()

    {

        return country;

    }

    public void setCountry(String country)

    {

        this.country = country;

    }

}

Admin:

public class Admin extends User implements PropertySearchable

{

    private String role;

    public Admin(int userId, String name, String email, String password, String phone,

    String role)

    {

        super(userId, name, email, password, phone);

        this.role = role;

    }

    public void manageUsers() // User management logic

    {

        System.out.println("Managing users...");

    }

    public void generateReports()

    {

        System.out.println("Generating reports..."); // Report generation logic

    }

    @Override

    public void displayInfo() {

        System.out.println("Admin: " + getName());

        System.out.println("Email: " + getEmail());

        System.out.println("Phone: " + getPhone());

        System.out.println("Role: " + role);

    }

    // Implementation of PropertySearchable interface (global search across all properties)

    @Override

    public Property[] searchByLocation(String location)

    {

        // This would typically connect to a database

        // Simplified implementation for now

        System.out.println("Searching properties by location: " + location);

        return new Property[0];

    }

    @Override

    public Property[] searchByPrice(double minPrice, double maxPrice) {

        // This would typically connect to a database or repository

        System.out.println("Searching properties by price range: " + minPrice + " - " + maxPrice);

        return new Property[0];

    }

    @Override

    public Property[] searchByType(String type) {

        // This would typically connect to a database or repository

        System.out.println("Searching properties by type: " + type);

        return new Property[0];

    }

}

Amenity:

public class Amenity

{

    private String name;

    private String description;

    private boolean isPremium;

    public Amenity(String name, String description, boolean isPremium)

    {

        this.name = name;

        this.description = description;

        this.isPremium = isPremium;

    }

    public String getName()

    {

        return name;

    }

    public String getDescription()

    {

        return description;

    }

    public boolean isPremium()

    {

        return isPremium;

    }

}

Appointment:

public class Appointment

{

    private int appointmentId;

    private Date dateTime;

    private Customer customer;

    private Broker broker;

    private Property property;

    private String status;  // scheduled/confirmed/cancelled/completed

    public Appointment(int appointmentId, Date dateTime, Customer customer,

    Broker broker, Property property)

    {

        this.appointmentId = appointmentId;

        this.dateTime = dateTime;

        this.customer = customer;

        this.broker = broker;

        this.property = property;

        this.status = "scheduled";

    }

    public void reschedule(Date newDateTime)

    {

        if ("cancelled".equals(status))

        {

            throw new AppointmentException("Cannot reschedule a cancelled appointment");

        }

        else

        {

            this.dateTime = newDateTime;

            System.out.println("Appointment rescheduled to: " + newDateTime);

        }

    }

    public void cancel()

    {

        this.status = "cancelled";

        System.out.println("Appointment cancelled");

    }

    public void confirm()

    {

        if ("cancelled".equals(status)) {

            throw new AppointmentException("Cannot confirm a cancelled appointment");

        }

        this.status = "confirmed";

        System.out.println("Appointment confirmed");

    }

    public void complete()

    {

        if (!"confirmed".equals(status)) {

            throw new AppointmentException("Only confirmed appointments can be completed");

        }

        this.status = "completed";

        System.out.println("Appointment marked as completed");

    }

    public int getAppointmentId()

    {

        return appointmentId;

    }

    public Date getDateTime()

    {

        return dateTime;

    }

    public Customer getCustomer()

    {

        return customer;

    }

    public Broker getBroker()

    {

        return broker;

    }

    public Property getProperty()

    {

        return property;

    }

    public String getStatus()

    {

        return status;

    }

}

AppointmentException:

public class AppointmentException extends RuntimeException

{

    public AppointmentException(String message)

    {

        super(message);

    }

    public AppointmentException(String message, Throwable cause)

    {

        super(message, cause);

    }

}

Broker:

public class Broker extends User implements PropertySearchable

{

    private double commissionRate;

    private Property managedProperties[]; // Composition with Property

    private Appointment appointments[]; // Composition with Appointments

    private int propertyCount;

    private int appointmentCount;

    private static final int MAX\_PROPERTIES = 50;

    private static final int MAX\_APPOINTMENTS = 100;

    public Broker(int userId, String name, String email, String password, String phone,

    double commissionRate)

    {

        super(userId, name, email, password, phone);

        this.commissionRate = commissionRate;

        this.managedProperties = new Property[MAX\_PROPERTIES];

        this.appointments = new Appointment[MAX\_APPOINTMENTS];

        this.propertyCount = 0;

        this.appointmentCount = 0;

    }

    public void addProperty(Property property)

    {

        if (propertyCount < MAX\_PROPERTIES)

        {

            managedProperties[propertyCount] = property;

            propertyCount++;

            System.out.println("Property added to broker's portfolio");

        }

        else

        {

            System.out.println("Property portfolio is full");

        }

    }

    public void removeProperty(Property property)

    {

        boolean found = false;

        for (int i = 0; i < propertyCount; i++)

        {

            if (managedProperties[i].getPropertyId() == property.getPropertyId())

            {

                found = true;

                for (int j = i; j < propertyCount - 1; j++)

                {

                    managedProperties[j] = managedProperties[j + 1];

                }

                managedProperties[propertyCount - 1] = null;

                propertyCount--;

                System.out.println("Property removed from broker's portfolio");

                break;

            }

        }

        if (!found)

        {

            System.out.println("Property not found in broker's portfolio");

        }

    }

    public double calculateCommission(double amount)

    {

        return amount \* commissionRate;

    }

    public void scheduleAppointment(Appointment appointment)

    {

        if (appointmentCount < MAX\_APPOINTMENTS)

        {

            appointments[appointmentCount] = appointment;

            appointmentCount++;

            System.out.println("Appointment scheduled");

        }

        else

        {

            System.out.println("Appointment schedule is full");

        }

    }

    @Override

    public void displayInfo() {

        System.out.println("Broker: " + getName());

        System.out.println("Email: " + getEmail());

        System.out.println("Phone: " + getPhone());

        System.out.println("Commission Rate: " + (commissionRate \* 100) + "%");

        System.out.println("Managed Properties: " + propertyCount);

        System.out.println("Scheduled Appointments: " + appointmentCount);

    }

    @Override

    public Property[] searchByLocation(String location) // Implementation of PropertySearchable Interface

    {

        Property[] result = new Property[propertyCount];

        int count = 0;

        for (int i = 0; i < propertyCount; i++)

        {

            if (managedProperties[i].getAddress().getCity().equalsIgnoreCase(location) ||

                managedProperties[i].getAddress().getState().equalsIgnoreCase(location))

                {

                result[count] = managedProperties[i];

                count++;

            }

        }

        // Create a new array with the exact size of results

        Property[] finalResult = new Property[count];

        for (int i = 0; i < count; i++)

        {

            finalResult[i] = result[i];

        }

        return finalResult;

    }

    @Override

    public Property[] searchByPrice(double minPrice, double maxPrice)

    {

        Property[] result = new Property[propertyCount];

        int count = 0;

        for (int i = 0; i < propertyCount; i++)

        {

            double price = managedProperties[i].getPrice();

            if (price >= minPrice && price <= maxPrice)

            {

                result[count] = managedProperties[i];

                count++;

            }

        }

        // Create a new array with the exact size of results

        Property[] finalResult = new Property[count];

        for (int i = 0; i < count; i++)

        {

            finalResult[i] = result[i];

        }

        return finalResult;

    }

    @Override

    public Property[] searchByType(String type)

    {

        Property[] result = new Property[propertyCount];

        int count = 0;

        for (int i = 0; i < propertyCount; i++)

        {

            if ((type.equalsIgnoreCase("Residential") && managedProperties[i] instanceof ResidentialProperty) ||

                (type.equalsIgnoreCase("Commercial") && managedProperties[i] instanceof CommercialProperty) ||

                (type.equalsIgnoreCase("Land") && managedProperties[i] instanceof Land))

                {

                result[count] = managedProperties[i];

                count++;

            }

        }

        // Create a new array with the exact size of results

        Property[] finalResult = new Property[count];

        for (int i = 0; i < count; i++)

        {

            finalResult[i] = result[i];

        }

        return finalResult;

    }

}

CommercialProperty:

public class CommercialProperty extends Property

{

    private CommercialType type;

    private double maintenanceCost;

    private boolean isParkingAvailable;

    public CommercialProperty(int propertyId, double price, double area, PropertyStatus status,

    Address address, CommercialType type, double maintenanceCost, boolean isParkingAvailable)

    {

        super(propertyId, price, area, status, address);

        this.type = type;

        this.maintenanceCost = maintenanceCost;

        this.isParkingAvailable = isParkingAvailable;

    }

    @Override

    public String getDescription()

    {

        String parking = isParkingAvailable ? "with" : "without";

        return type + " commercial property, " + getArea() + " sq ft, " + parking +

        " parking, located at " + getAddress().getFullAddress();

    }

    public double calculateMaintenancePerSqFt()

    {

        return maintenanceCost / getArea();

    }

    public CommercialType getType()

    {

        return type;

    }

    public double getMaintenanceCost()

    {

        return maintenanceCost;

    }

    public boolean isParkingAvailable()

    {

        return isParkingAvailable;

    }

    @Override

    public double calculateValue()

    {

        double baseValue = super.calculateValue();

        // Adjust value based on property type

        double typeMultiplier = 1.0;

        switch (type)

        {

            case OFFICE:

                typeMultiplier = 1.3;

                break;

            case RETAIL:

                typeMultiplier = 1.2;

                break;

            case INDUSTRIAL:

                typeMultiplier = 1.1;

                break;

            case WAREHOUSE:

                typeMultiplier = 0.9;

                break;

        }

        // Parking adds value

        double parkingPremium = isParkingAvailable ? 50000 : 0;

        // Higher maintenance cost reduces value

        double maintenanceFactor = 1.0 - (calculateMaintenancePerSqFt() \* 0.1);

        return (baseValue \* typeMultiplier \* maintenanceFactor) + parkingPremium;

    }

}

CommercialType:

public enum CommercialType

{

    OFFICE,

    RETAIL,

    INDUSTRIAL,

    WAREHOUSE

}

Commission:

public class Commission

{

    private int commissionId;

    private Broker broker;

    private double amount;

    private boolean isPaid;

    public Commission(int commissionId, Broker broker, double amount)

    {

        this.commissionId = commissionId;

        this.broker = broker;

        this.amount = amount;

        this.isPaid = false;

    }

    public void markAsPaid()

    {

        isPaid = true;

        System.out.println("Commission marked as paid for broker: " + broker.getName());

    }

    public double calculateTax()

    {

        return amount \* 0.20;

    }

    public int getCommissionId()

    {

        return commissionId;

    }

    public Broker getBroker()

    {

        return broker;

    }

    public double getAmount()

    {

        return amount;

    }

    public boolean isPaid()

    {

        return isPaid;

    }

}

Customer:

public class Customer extends User

{

    private Property favorites[];  // Array of favorite properties (Composition)

    private Transaction transHistory[];  // Array of transaction history (Composition)

    private int favoriteCount;

    private int transactionCount;

    private static final int MAX\_FAVORITES = 20;

    private static final int MAX\_TRANSACTIONS = 50;

    public Customer(int userId, String name, String email, String password, String phone)

    {

        super(userId, name, email, password, phone);

        this.favorites = new Property[MAX\_FAVORITES];

        this.transHistory = new Transaction[MAX\_TRANSACTIONS];

        this.favoriteCount = 0;

        this.transactionCount = 0;

    }

    public void addFavorite(Property property)

    {

        if (favoriteCount < MAX\_FAVORITES)

        {

            favorites[favoriteCount] = property;

            favoriteCount++;

            System.out.println("Property added to favorites");

        }

        else

        {

            System.out.println("Favorites list is full");

        }

    }

    public void removeFavorite(Property property)

    {

        boolean found = false;

        for (int i = 0; i < favoriteCount; i++)

        {

            if (favorites[i].getPropertyId() == property.getPropertyId())

            {

                found = true;

                for (int j = i; j < favoriteCount - 1; j++)

                {

                    favorites[j] = favorites[j + 1];

                }

                favorites[favoriteCount - 1] = null;

                favoriteCount--;

                System.out.println("Property removed from favorites");

                break;

            }

        }

        if (!found)

        {

            System.out.println("Property not found in favorites");

        }

    }

    public Property[] getTransactionHistory()

    {

        Property history[] = new Property[transactionCount];

        for (int i = 0; i < transactionCount; i++)

        {

            history[i] = transHistory[i].getProperty();

        }

        return history;

    }

    public void addTransaction(Transaction transaction)

    {

        if (transactionCount < MAX\_TRANSACTIONS)

        {

            transHistory[transactionCount] = transaction;

            transactionCount++;

        }

        else

        {

            System.out.println("Transaction history is full");

        }

    }

    @Override

    public void displayInfo() {

        System.out.println("Customer: " + getName());

        System.out.println("Email: " + getEmail());

        System.out.println("Phone: " + getPhone());

        System.out.println("Favorites: " + favoriteCount);

        System.out.println("Transaction History: " + transactionCount);

    }

}

Invoice:

public class Invoice

{

    private int invoiceId;

    private Date creationDate;

    private Date dueDate;

    private double totalAmount;

    private InvoiceItem items[]; // Composition with InvoiceItem

    private int itemCount;

    private static final int MAX\_ITEMS = 20;

    public Invoice(int invoiceId, Date creationDate, Date dueDate, double totalAmount)

    {

        this.invoiceId = invoiceId;

        this.creationDate = creationDate;

        this.dueDate = dueDate;

        this.totalAmount = totalAmount;

        this.items = new InvoiceItem[MAX\_ITEMS];

        this.itemCount = 0;

    }

    public void addItem(InvoiceItem item)

    {

        if (itemCount < MAX\_ITEMS)

        {

            items[itemCount] = item;

            itemCount++;

            recalculateTotal();

        }

        else

        {

            System.out.println("Maximum number of invoice items reached");

        }

    }

    public double calculateTotal()

    {

        double total = 0.0;

        for (int i = 0; i < itemCount; i++)

        {

            total += items[i].getAmount();

        }

        return total;

    }

    private void recalculateTotal()

    {

        totalAmount = calculateTotal();

    }

    public void generatePDF()

    {

        try

        {

            System.out.println("Generating PDF invoice: " + invoiceId);

            // Logic to generate PDF invoice

        }

        catch (Exception e)

        {

            System.out.println("Error generating invoice PDF: " + e.getMessage());

        }

    }

    public int getInvoiceId()

    {

        return invoiceId;

    }

    public Date getCreationDate()

    {

        return creationDate;

    }

    public Date getDueDate()

    {

        return dueDate;

    }

    public double getTotalAmount()

    {

        return totalAmount;

    }

    public InvoiceItem[] getItems()

    {

        InvoiceItem[] result = new InvoiceItem[itemCount];

        for (int i = 0; i < itemCount; i++)

        {

            result[i] = items[i];

        }

        return result;

    }

}

InvoiceItem:

public class InvoiceItem

{

    private String description;

    private double amount;

    private int quantity;

    public InvoiceItem(String description, double amount, int quantity)

    {

        this.description = description;

        this.amount = amount;

        this.quantity = quantity;

    }

    public double getAmount()

    {

        return amount \* quantity;

    }

    public String getDescription()

    {

        return description;

    }

    public int getQuantity()

    {

        return quantity;

    }

    public double getUnitAmount()

    {

        return amount;

    }

}

Land:

public class Land extends Property

{

    private LandType type;

    private boolean isZoned;

    private String zoneType;

    public Land(int propertyId, double price, double area, PropertyStatus status,

    Address address, LandType type, boolean isZoned, String zoneType)

    {

        super(propertyId, price, area, status, address);

        this.type = type;

        this.isZoned = isZoned;

        this.zoneType = zoneType;

    }

    @Override

    public String getDescription()

    {

        String zoning = isZoned ? "zoned for " + zoneType : "not zoned";

        return type + " land, " + getArea() + " sq ft, " +

        zoning + ", located at " + getAddress().getFullAddress();

    }

    public boolean checkDevelopmentPotential()

    {

        // Logic to determine development potential based on zoning and type

        if (!isZoned)

        {

            return false;

        }

        switch (type)

        {

            case RESIDENTIAL:

                return zoneType.equalsIgnoreCase("residential") ||

                       zoneType.equalsIgnoreCase("mixed");

            case COMMERCIAL:

                return zoneType.equalsIgnoreCase("commercial") ||

                       zoneType.equalsIgnoreCase("mixed");

            case AGRICULTURAL:

                return zoneType.equalsIgnoreCase("agricultural");

            case RECREATIONAL:

                return zoneType.equalsIgnoreCase("recreational") ||

                       zoneType.equalsIgnoreCase("mixed");

            default:

                return false;

        }

    }

    public LandType getType()

    {

        return type;

    }

    public boolean isZoned()

    {

        return isZoned;

    }

    public String getZoneType()

    {

        return zoneType;

    }

    @Override

    public double calculateValue()

    {

        // Land value calculation

        double baseValue = super.calculateValue();

        // Adjust value based on land type

        double typeMultiplier = 1.0;

        switch (type)

        {

            case RESIDENTIAL:

                typeMultiplier = 1.2;

                break;

            case COMMERCIAL:

                typeMultiplier = 1.5;

                break;

            case AGRICULTURAL:

                typeMultiplier = 0.8;

                break;

            case RECREATIONAL:

                typeMultiplier = 0.9;

                break;

        }

        // Zoning adds value

        double zoningMultiplier = isZoned ? 1.3 : 1.0;

        return baseValue \* typeMultiplier \* zoningMultiplier;

    }

}

LandType:

public enum LandType

{

    RESIDENTIAL,

    COMMERCIAL,

    AGRICULTURAL,

    RECREATIONAL

}

Main:

public class Main

{

    public static void main(String[] args)

    {

        Address address = new Address("123 Main St", "Lahore", "Punjab", "54000", "Pakistan");

        Admin admin = new Admin(1, "Alice", "alice@realestate.com", "admin123", "1234567890", "System Admin");

        Broker broker = new Broker(2, "Bob", "bob@realestate.com", "broker123", "0987654321", 0.05);

        Customer customer = new Customer(3, "Charlie", "charlie@gmail.com", "cust123", "1122334455");

        ResidentialProperty resProperty = new ResidentialProperty(101,5000.0,2000.0,

        PropertyStatus.AVAILABLE,address,3, 2,ResidentialType.SINGLE\_FAMILY

        );

        broker.addProperty(resProperty);

        System.out.println("=== Admin Info ===");

        admin.displayInfo();

        System.out.println("\n=== Property Search: Lahore ===");

        Property[] searchResults = broker.searchByLocation("Lahore");

        if (searchResults.length == 0) {

            System.out.println("No properties found in Lahore.");

        } else {

            for (Property prop : searchResults) {

                System.out.println(prop.getDescription());

            }

        }

        Date appointmentDate = new Date(2025, 5, 20, 14, 30); // Custom Date object

        Appointment appointment = new Appointment(1, appointmentDate, customer, broker, resProperty);

        broker.scheduleAppointment(appointment);

        appointment.confirm();

        System.out.println("\n=== Appointment Details ===");

        System.out.println("Appointment ID: " + appointment.getAppointmentId());

        System.out.println("Date & Time: " + appointment.getDateTime());

        System.out.println("Customer: " + appointment.getCustomer().getName());

        System.out.println("Broker: " + appointment.getBroker().getName());

        System.out.println("Property: " + appointment.getProperty().getDescription());

        System.out.println("Status: " + appointment.getStatus());

    }

}

Payment:

public class Payment

{

    private int paymentId;

    private double amount;

    private Date paymentDate;

    private PaymentMethod method;

    private PaymentStatus status;

    public Payment(int paymentId, double amount, Date paymentDate, PaymentMethod method)

    {

        this.paymentId = paymentId;

        this.amount = amount;

        this.paymentDate = paymentDate;

        this.method = method;

        this.status = PaymentStatus.PENDING;

    }

    public boolean processPayment()

    {

        try

        {

            // Payment processing logic based on method

            System.out.println("Processing payment of $" + amount + " via " + method);

            boolean success = true;

            if (success)

            {

                status = PaymentStatus.COMPLETED;

                System.out.println("Payment completed successfully");

            }

            else

            {

                status = PaymentStatus.FAILED;

                System.out.println("Payment processing failed");

            }

            return success;

        }

        catch (Exception e)

        {

            status = PaymentStatus.FAILED;

            System.out.println("Payment processing error: " + e.getMessage());

            return false;

        }

    }

    public void generateReceipt()

    {

        if (status == PaymentStatus.COMPLETED)

        {

            System.out.println("Generating receipt for payment: " + paymentId);

        }

        else

        {

            throw new PaymentException("Cannot generate receipt for incomplete payment");

        }

    }

    public int getPaymentId()

    {

        return paymentId;

    }

    public double getAmount()

    {

        return amount;

    }

    public Date getPaymentDate()

    {

        return paymentDate;

    }

    public PaymentMethod getMethod()

    {

        return method;

    }

    public PaymentStatus getStatus()

    {

        return status;

    }

}

PaymentException:

public class PaymentException extends RuntimeException

{

    public PaymentException(String message)

    {

        super(message);

    }

    public PaymentException(String message, Throwable cause)

    {

        super(message, cause);

    }

}

PaymentMethod:

public enum PaymentMethod

{

    CASH,

    CHECK,

    CREDIT\_CARD,

    BANK\_TRANSFER,

    WIRE\_TRANSFER,

    ESCROW

}

PaymentStatus:

public enum PaymentStatus

{

    PENDING,

    PROCESSING,

    COMPLETED,

    FAILED,

    REFUNDED

}

Property:

public abstract class Property // Abstract Parent Class for All Properties

{

    private int propertyId;

    private double price;

    private double area;

    private PropertyStatus status;

    private Address address;  // Composition with Address

    public Property(int propertyId, double price, double area, PropertyStatus

    status, Address address)

    {

        this.propertyId = propertyId;

        this.price = price;

        this.area = area;

        this.status = status;

        this.address = address;

    }

    public double calculateValue()

    {

        return price \* area;

    }

    public void updateStatus(PropertyStatus status)

    {

        this.status = status;

        System.out.println("Property status updated to: " + status);

    }

    public abstract String getDescription(); // Abstract Method

    public int getPropertyId()

    {

        return propertyId;

    }

    public double getPrice()

    {

        return price;

    }

    public void setPrice(double price)

    {

        this.price = price;

    }

    public double getArea()

    {

        return area;

    }

    public PropertyStatus getStatus()

    {

        return status;

    }

    public Address getAddress()

    {

        return address;

    }

}

PropertyDTO:

// Data Transfer Object for API responses

public class PropertyDTO

{

    private int id;

    private String type;

    private double price;

    private String location;

    private String status;

    private String description;

    public PropertyDTO(Property property)

    {

        this.id = property.getPropertyId();

        if (property instanceof ResidentialProperty)

        {

            this.type = "Residential: " + ((ResidentialProperty) property).getType();

        }

        else if (property instanceof CommercialProperty)

        {

            this.type = "Commercial: " + ((CommercialProperty) property).getType();

        }

        else if (property instanceof Land)

        {

            this.type = "Land: " + ((Land) property).getType();

        }

        else

        {

            this.type = "Unknown";

        }

        this.price = property.getPrice();

        this.location = property.getAddress().getCity() + ", " + property.getAddress().getState();

        this.status = property.getStatus().toString();

        this.description = property.getDescription();

    }

    public int getId()

    {

        return id;

    }

    public String getType()

    {

        return type;

    }

    public double getPrice()

    {

        return price;

    }

    public String getLocation()

    {

        return location;

    }

    public String getStatus()

    {

        return status;

    }

    public String getDescription()

    {

        return description;

    }

}

PropertyFilter:

// Helper classes for GUI and API development

public class PropertyFilter

{

    private String location;

    private double minPrice;

    private double maxPrice;

    private String propertyType;

    private int minBedrooms;

    private int maxBedrooms;

    public PropertyFilter()

    {

        this.minPrice = 0;

        this.maxPrice = Double.MAX\_VALUE;

        this.minBedrooms = 0;

        this.maxBedrooms = Integer.MAX\_VALUE;

    }

    // Builder pattern for easy filtering

    public PropertyFilter withLocation(String location)

    {

        this.location = location;

        return this;

    }

    public PropertyFilter withPriceRange(double minPrice, double maxPrice)

    {

        this.minPrice = minPrice;

        this.maxPrice = maxPrice;

        return this;

    }

    public PropertyFilter withPropertyType(String propertyType)

    {

        this.propertyType = propertyType;

        return this;

    }

    public PropertyFilter withBedroomRange(int minBedrooms, int maxBedrooms)

    {

        this.minBedrooms = minBedrooms;

        this.maxBedrooms = maxBedrooms;

        return this;

    }

    public String getLocation()

    {

        return location;

    }

    public double getMinPrice()

    {

        return minPrice;

    }

    public double getMaxPrice()

    {

        return maxPrice;

    }

    public String getPropertyType()

    {

        return propertyType;

    }

    public int getMinBedrooms()

    {

        return minBedrooms;

    }

    public int getMaxBedrooms()

    {

        return maxBedrooms;

    }

}

PropertySearchable:

public interface PropertySearchable

{

    Property[] searchByLocation(String location);

    Property[] searchByPrice(double minPrice, double maxPrice);

    Property[] searchByType(String type);

}

PropertyStatus:

public enum PropertyStatus

{

    AVAILABLE,

    SOLD,

    RENTED,

    UNDER\_CONTRACT,

    MAINTENANCE

}

PropertyTransactionException:

public class PropertyTransactionException extends RuntimeException

{

    public PropertyTransactionException(String message)

    {

        super(message);

    }

    public PropertyTransactionException(String message, Throwable cause)

    {

        super(message, cause);

    }

}

RentalTransaction:

public class RentalTransaction extends Transaction

{

    private int leaseTerm;  // In months

    private double securityDeposit;

    private boolean isRenewable;

    public RentalTransaction(int transactionId, Date transactionDate, User owner, User tenant,

    Property property, double amount, int leaseTerm, double securityDeposit, boolean isRenewable)

    {

        super(transactionId, transactionDate, owner, tenant, property, amount);

        this.leaseTerm = leaseTerm;

        this.securityDeposit = securityDeposit;

        this.isRenewable = isRenewable;

    }

    @Override

    public double calculateFees()

    {

        double baseFee = getAmount() \* 0.10;  // One month's rent as fee (standard)

        // Adjustments based on lease term

        if (leaseTerm <= 6)

        {

            baseFee += getAmount() \* 0.05;  // Additional fee for short-term leases

        }

        else if (leaseTerm >= 24)

        {

            baseFee -= getAmount() \* 0.02;  // Discount for long-term leases

        }

        return baseFee;

    }

    public void generateLeaseAgreement()

    {

        try

        {

            System.out.println("Generating lease agreement for property: " + getProperty().getPropertyId());

            // Logic to generate a lease agreement document

        }

        catch (Exception e)

        {

            System.out.println("Error generating lease agreement: " + e.getMessage());

            throw new PropertyTransactionException("Failed to generate lease agreement: " + e.getMessage());

        }

    }

    public int getLeaseTerm()

    {

        return leaseTerm;

    }

    public double getSecurityDeposit()

    {

        return securityDeposit;

    }

    public boolean isRenewable()

    {

        return isRenewable;

    }

}

ResidentialProperty:

public class ResidentialProperty extends Property

{

    private int bedrooms;

    private int bathrooms;

    private ResidentialType type;

    private Amenity amenities[];  // Composition

    private int amenityCount;

    private static final int MAX\_AMENITIES = 10;

    public ResidentialProperty(int propertyId, double price, double area, PropertyStatus status,

    Address address, int bedrooms, int bathrooms, ResidentialType type)

    {

        super(propertyId, price, area, status, address);

        this.bedrooms = bedrooms;

        this.bathrooms = bathrooms;

        this.type = type;

        this.amenities = new Amenity[MAX\_AMENITIES];

        this.amenityCount = 0;

    }

    @Override

    public String getDescription()

    {

        return type + " residential property with " + bedrooms + " bedrooms, " + bathrooms +

        " bathrooms, " + getArea() + " sq ft, located at " + getAddress().getFullAddress();

    }

    public void addAmenity(Amenity amenity)

    {

        if (amenityCount < MAX\_AMENITIES)

        {

            amenities[amenityCount] = amenity;

            amenityCount++;

            System.out.println("Amenity added: " + amenity.getName());

        }

        else

        {

            System.out.println("Maximum number of amenities reached");

        }

    }

    public Amenity[] getAmenities()

    {

        Amenity result[] = new Amenity[amenityCount];

        for (int i = 0; i < amenityCount; i++)

        {

            result[i] = amenities[i];

        }

        return result;

    }

    public int getBedrooms()

    {

        return bedrooms;

    }

    public int getBathrooms()

    {

        return bathrooms;

    }

    public ResidentialType getType()

    {

        return type;

    }

    @Override

    public double calculateValue()

    {

        double baseValue = super.calculateValue();

        double amenityValue = 0;

        for (int i = 0; i < amenityCount; i++)

        {

            if (amenities[i].isPremium())

            {

                amenityValue += 10000;  // Premium amenities add more value

            }

            else

            {

                amenityValue += 5000;   // Standard amenities add less value

            }

        }

        // Adjust value based on property type and rooms

        double typeMultiplier = 1.0;

        switch (type)

        {

            case SINGLE\_FAMILY:

                typeMultiplier = 1.2;

                break;

            case CONDO:

                typeMultiplier = 1.0;

                break;

            case TOWNHOUSE:

                typeMultiplier = 1.1;

                break;

            case APARTMENT:

                typeMultiplier = 0.9;

                break;

        }

        return (baseValue \* typeMultiplier) + amenityValue;

    }

}

ResidentialType:

public enum ResidentialType

{

    SINGLE\_FAMILY,

    CONDO,

    TOWNHOUSE,

    APARTMENT

}

SaleTransaction:

public class SaleTransaction extends Transaction

{

    private boolean isFinanced;

    private double downPayment;

    public SaleTransaction(int transactionId, Date transactionDate, User seller, User buyer,

    Property property, double amount, boolean isFinanced, double downPayment)

    {

        super(transactionId, transactionDate, seller, buyer, property, amount);

        this.isFinanced = isFinanced;

        this.downPayment = downPayment;

    }

    @Override

    public double calculateFees()

    {

        double baseFee = getAmount() \* 0.06;  // 6% standard real estate commission

        if (isFinanced)

        {

            baseFee += getAmount() \* 0.01;  // 1% loan origination fee

        }

        return baseFee;

    }

    public void generateDeed()

    {

        try

        {

            System.out.println("Generating deed for property: " + getProperty().getPropertyId());

        }

        catch (Exception e)

        {

            System.out.println("Error generating deed: " + e.getMessage());

            throw new PropertyTransactionException("Failed to generate deed: " + e.getMessage());

        }

    }

    public boolean isFinanced()

    {

        return isFinanced;

    }

    public double getDownPayment()

    {

        return downPayment;

    }

}

Transaction:

public abstract class Transaction // Abstract Parent Class for All Transactions

{

    private int transactionId;

    private Date transactionDate;

    private User seller;

    private User buyer;

    private Property property;

    private double amount;

    private TransactionStatus status;

    private Invoice invoice;  // Composition

    public Transaction(int transactionId, Date transactionDate, User seller, User buyer,

    Property property, double amount)

    {

        this.transactionId = transactionId;

        this.transactionDate = transactionDate;

        this.seller = seller;

        this.buyer = buyer;

        this.property = property;

        this.amount = amount;

        this.status = TransactionStatus.PENDING;

    }

    public boolean processTransaction()

    {

        try

        {

            this.invoice = new Invoice(generateInvoiceId(), new Date(), calculateDueDate(), amount);

            updateStatus(TransactionStatus.APPROVED);

            return true;

        }

        catch (Exception e)

        {

            System.out.println("Transaction processing failed: " + e.getMessage());

            updateStatus(TransactionStatus.FAILED);

            return false;

        }

    }

    public abstract double calculateFees(); // Abstract Method

    public void updateStatus(TransactionStatus status)

    {

        this.status = status;

        System.out.println("Transaction status updated to: " + status);

    }

    private int generateInvoiceId()

    {

        return this.transactionId + 1000;

    }

    private Date calculateDueDate()

    {

        Calendar calendar = Calendar.getInstance();

        calendar.setTime(transactionDate);

        calendar.add(Calendar.DATE, 30);

        return calendar.getTime();

    }

    public int getTransactionId()

    {

        return transactionId;

    }

    public Date getTransactionDate()

    {

        return transactionDate;

    }

    public User getSeller()

    {

        return seller;

    }

    public User getBuyer()

    {

        return buyer;

    }

    public Property getProperty()

    {

        return property;

    }

    public double getAmount()

    {

        return amount;

    }

    public TransactionStatus getStatus()

    {

        return status;

    }

    public Invoice getInvoice()

    {

        return invoice;

    }

}

TransactionStatus:

public enum TransactionStatus

{

    PENDING,

    APPROVED,

    COMPLETED,

    CANCELLED,

    FAILED

}

User:

public abstract class User // Abstract Parent Class for All Users

{

    private int userId;

    private String name;

    private String email;

    private String password;

    private String phone;

    public User(int userId, String name, String email, String password, String phone)

    {

        this.userId = userId;

        this.name = name;

        this.email = email;

        this.password = password;

        this.phone = phone;

    }

    public void login() // Authentication logic

    {

        System.out.println("User " + name + " logged in");

    }

    public void logout() // Logout logic

    {

        System.out.println("User " + name + " logged out");

    }

    public abstract void displayInfo(); // Abstract method

    public int getUserId()

    {

        return userId;

    }

    public String getName()

    {

        return name;

    }

    public void setName(String name)

    {

        this.name = name;

    }

    public String getEmail()

    {

        return email;

    }

    public void setEmail(String email)

    {

        this.email = email;

    }

    public String getPhone()

    {

        return phone;

    }

    public void setPhone(String phone)

    {

        this.phone = phone;

    }

    public void setPassword(String password)

    {

        this.password = password;

    }

    public boolean verifyPassword(String inputPassword) // Method to check password

    {

        return this.password.equals(inputPassword);

    }

}