**Real Estate Management System**

**GitHub Code path-** https://github.com/tussoftwaredesign/project-rooshmica

**1. Introduction**

The **Real Estate Management System** is a Java-based application designed to manage real estate properties efficiently, adhering to modern programming principles and features from the latest Java LTS (Java 21). The application enables users to perform CRUD operations on residential and commercial properties, search for properties based on specific criteria, and dynamically update property details.

The project showcases both fundamental and advanced Java features, ensuring a maintainable, modular, and robust implementation. It focuses on real-world problem-solving by effectively utilizing object-oriented design, advanced Java constructs, and functional programming techniques.

**2. User Stories Completed**

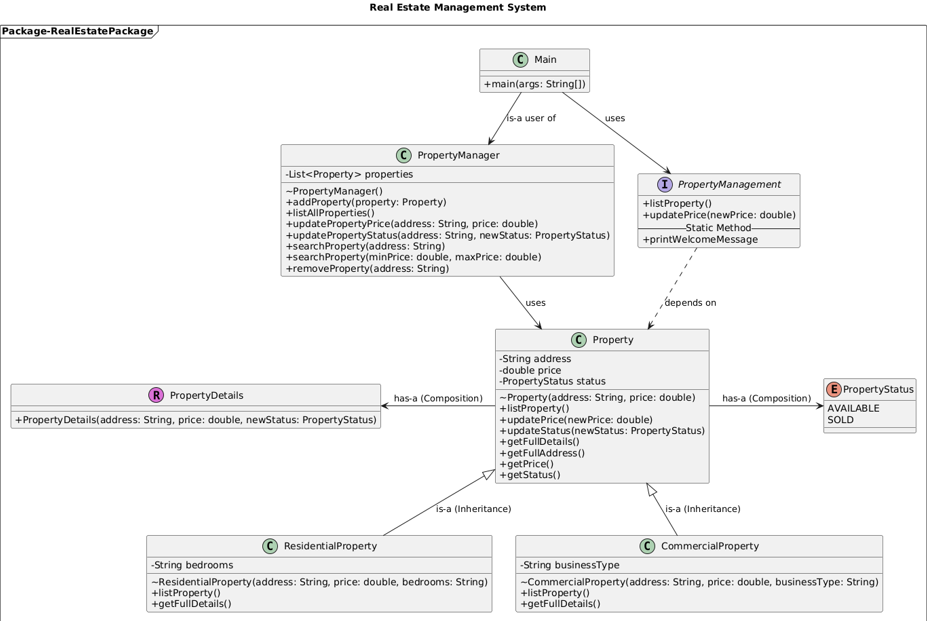
The following user stories were successfully implemented:

1. **Add a New Property:**  
   Users can add residential or commercial properties to the system.
2. **Search Properties:**  
   Search properties by address or price range using predicates for filtering.
3. **Update Property Details:**  
   Update property attributes like price and status.
4. **Remove Property:**  
   Delete properties from the system based on a unique identifier.
5. **List Properties:**  
   Display all properties in the system, categorized by type (residential/commercial).
6. **Validation and Error Handling:**  
   Handle invalid inputs, such as negative prices or invalid property IDs, through appropriate exceptions.
7. **Sold Properties:**

Users can update the property status and view all the properties

**3. UML**

* Property is the central class, serving as a superclass for ResidentialProperty and CommercialProperty (Inheritance relationship).
* PropertyDetails is a record, associated with Property through a composition relationship (has-a).
* PropertyManager manages Property objects, demonstrating aggregation (uses).
* PropertyManagement is an interface, implemented by Property to define common behavior like listProperty() and updatePrice().
* PropertyStatus is an Enum has a composite relationship with the property class

****

**4. Evaluation**

Evaluating the Java Concepts Used in Real Estate Management System

1. **Classes**

* A class is a blueprint for creating objects, encapsulating data and methods that operate on the data. All classes in the project (‘Property’, ‘ResidentialProperty’, ‘CommercialProperty’, ‘PropertyManager’, ‘Main’) are examples of class usage.

**Code Example:**

public sealed class Property implements PropertyManagement

permits ResidentialProperty, CommercialProperty {

// Constructor and methods

}

1. **Contrast this() and this.**

* this. refers to the current instance's member (field or method). We have referenced in the ‘updatePrice’ method in the ‘Property’ class.
* this() invokes another constructor of the same class. Used in the default constructor of the ‘Property’ class to chain constructors.

**Code Example:**

public Property() {

this("Unknown Address", 0.0); // Invokes another constructor

}

public void updatePrice(double newPrice) {

this.PropertyDetails = new PropertyDetails(PropertyDetails.address(), newPrice, PropertyDetails.status());

}

1. **Method Overloading**

* Methods in the same class with the same name but different parameter lists. In the ‘PropertyManager’ class, the ‘searchProperty’ method is overloaded.

**Code Example:**

public Property searchProperty(String address) {

// Implementation

}

public List<Property> searchProperty(double minPrice, double maxPrice) {

// Implementation

}

1. **Varargs**

A feature to pass a variable number of arguments to a method. Not explicitly used in the current code.

**Local Variable Type Inference (LVTI)**

Using var to infer the type of local variables. Used in the enhanced for loop in the ‘Main’ class.

**Code Example:**

for (var action : actions) {

// Implementation

}

1. **Encapsulation**

Restricting direct access to fields and providing controlled access through methods. Fields in the ‘Property’ class are private, and access is provided through getters and setters.

**Code Example:**

private PropertyDetails PropertyDetails;

public String getFullAddress() {

return PropertyDetails.address();

}

1. **Interfaces**

A reference type in Java, used to specify behaviours that classes must implement. The ‘PropertyManagement’ interface is implemented by the ‘Property’ class.

**Code Example:**

public interface PropertyManagement {

void listProperty();

void updatePrice(double newPrice);

}

1. **Inheritance**

Mechanism to create a new class using the properties and methods of an existing class. The ‘ResidentialProperty’ and ‘CommercialProperty’ classes extend the ‘Property’ class.

**Code Example:**

final class ResidentialProperty extends Property {

// Additional fields and methods

}

final class CommercialProperty extends Property {

// Additional fields and methods

}

1. **Overriding and Polymorphism**

Redefining a method in the subclass and achieving runtime behavior. The ‘listProperty’ method is overridden in the ‘ResidentialProperty’ and ‘CommercialProperty’ classes.

**Code Example:**

@Override

public void listProperty() {

System.out.println("Residential property at " + super.getFullAddress());

}

@Override

public void listProperty() {

System.out.println("Commercial property at " + super.getFullAddress());

}

1. **Contrast super() and super.**

* Super. refers to the superclass's methods or fields. Used in the ‘listProperty’ method in subclasses.
* Super() invokes the superclass constructor. Used in constructors of the ‘ResidentialProperty’ and ‘CommercialProperty’ classes.

**Code Example:**

ResidentialProperty(String address, double price, int bedrooms) {

super(address, price);

this.bedrooms = bedrooms;

}

@Override

public void listProperty() {

System.out.println("Residential property at " + super.getFullAddress());

}

Commercial Property(String address, double price, int bedrooms) {

super(address, price);

this.businessType = businessType;

}

@Override

public void listProperty() {

System.out.println("Commercial Property at " + super.getFullAddress());

}

1. **Exceptions (Checked and Unchecked)**

Mechanism to handle runtime errors.

* Checked Exception: IllegalArgumentException in ‘updatePrice’.
* Unchecked Exception: NullPointerException in ‘addProperty’.

**Code Example:**

if (newPrice < 0) {

throw new IllegalArgumentException("Price cannot be negative");

}

catch (NullPointerException e) {  
 System.*out*.println("Error at method listProperty(): " + e.getMessage());  
}

1. **Enums**

A special Java type for a fixed set of constants. The ‘PropertyStatus’ enum is used to define property statuses.

**Code Example:**

public enum PropertyStatus {

AVAILABLE, SOLD

}

1. **Arrays**

A container that holds a fixed number of values of a single type. Used in the ‘actions’ array in the ‘Main’ class.

**Code Example:**

String[] actions = {"ADD\_PROPERTY", "LIST\_PROPERTY", "SEARCH\_BY\_ADDRESS", "SEARCH\_BY\_PRICE", "UPDATE\_PROPERTY", "UPDATE\_STATUS", "REMOVE\_PROPERTY"};

1. **Use of Java Core API**

**String**: Used for storing and manipulating text, such as in property addresses.

**Date API**: Used to print the current date in the welcome message.

We have imported two Date API for printing

**Code Example:**

import java.time.LocalDate;  
import java.time.format.DateTimeFormatter;

LocalDate currentDate = LocalDate.*now*();  
DateTimeFormatter formatter = DateTimeFormatter.*ofPattern*("dd MMMM yyyy");  
System.*out*.println("Today's Date: " + currentDate.format(formatter));

1. **Records**

Immutable data classes introduced in Java. The ‘PropertyDetails’ record stores address, price, and status. PropertyDetails record is used in the Property constructor class for all the properties along with setting the default property status.

**Code Example:**

public record PropertyDetails(String address, double price, PropertyStatus status) {}

public Property(String address, double price)  
{  
 this.PropertyDetails = new PropertyDetails(address, price, PropertyStatus.*AVAILABLE*);  
}

1. **Custom Immutable Type**

A class where all fields are final, and its state cannot be modified. The ‘ResidentialProperty’ and ‘CommercialProperty’ classes.

**Code Example:**

final class ResidentialProperty extends Property {

private final int bedrooms;

}

final class CommercialProperty extends Property  
 {  
 private final String businessType;

}

1. **Lambdas (Predicate)**

A concise way to represent an anonymous function. Used in ‘PropertyManager’ for filtering properties by price.

**Code Example:**

Predicate<Property> priceInRange = property -> property.getPrice() >= minPrice && property.getPrice() <= maxPrice;

1. **Switch Expressions**

A more concise way to handle conditional logic. In the ‘Main’ class to process actions.

**Code Example:**

switch (action) {  
 case "ADD\_PROPERTY" -> addProperties.run();  
 case "LIST\_PROPERTY" -> listProperties.run();  
 case "SEARCH\_BY\_ADDRESS" -> searchByAddress.run();  
 case "SEARCH\_BY\_PRICE" -> searchByPrice.run();  
 case "UPDATE\_PROPERTY" -> updateProperty.run();  
 case "UPDATE\_STATUS" -> updateStatus.run();  
 case "REMOVE\_PROPERTY" -> removeProperty.run();  
 default -> System.*out*.println("No action: " + action);  
}

1. **Call-by-Value and Defensive Copying**

**Call-by-Value**

Java uses call-by-value for method arguments. When primitive types (like double for price updates) or object references (like Property objects) are passed to methods, the method receives a copy of the value or reference.

public void updatePropertyPrice(String address, double newPrice) {

try {

if (address == null || address.isEmpty()) {

throw new IllegalArgumentException("Address cannot be null or empty.");

}

for (Property property : properties) {

if (property.getFullAddress().equalsIgnoreCase(address)) {

property.updatePrice(newPrice); // newPrice is passed as call-by-value

return;

}

}

throw new NullPointerException("Property not found: " + address);

} catch (IllegalArgumentException | NullPointerException e) {

System.out.println("Error at method updatePropertyPrice(): " + e.getMessage());

}

}

1. **Sealed Classes**

Restricts which classes can extend or implement a class/interface. The ‘Property’ class is sealed.

**Code Example:**

public sealed class Property permits ResidentialProperty, CommercialProperty {

// Implementation

}

1. **Private, Default, and Static Interface Methods**

The static method **printWelcomeMessage()** in the PropertyManagement interface demonstrates a static interface method.

public interface PropertyManagement {

static void printWelcomeMessage() {

LocalDate currentDate = LocalDate.now();

DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd MMMM yyyy");

System.out.println("\nWelcome to the Real Estate Management System!!!");

System.out.println("Today's Date: " + currentDate.format(formatter));

}

}

**Private Fields**

All fields in the Property class (PropertyDetails) and the PropertyManager class (properties) are private.

private PropertyDetails PropertyDetails;

private List<Property> properties;

**5. Challenges Encountered**

1. **Encapsulation Violation:**  
   Initially, direct access to parent class fields without using getters, setters, or super broke encapsulation. This was corrected by adhering to proper encapsulation practices.
2. **Code Complexity and Readability:**  
   Integrating all logic into a single class increased complexity, making it hard to maintain and understand. The solution was to modularize the code into distinct classes and interfaces, improving maintainability.

**6. Concepts that are not used with reason**

* **Defensive Copying**

**Reason for not being used -** Immutable PropertyDetails record ensures safe data handling, reducing the need for defensive copies.

* **Switch Expressions with Pattern Matching**

**Reason for not being used** - The design of the application already utilizes inheritance and polymorphism effectively. Explicit pattern matching is unnecessary as the polymorphic behaviour is sufficient for determining the type of property and invoking the correct implementation.

* **Varargs**

**Reason for not being used** - The current application does not have any method or functionality that inherently requires a variable number of arguments. All input data (e.g., properties, addresses) is handled as individual parameters or collections like List. The design ensures clarity by using well-defined inputs, avoiding scenarios where the number of arguments is dynamic.

**7. Future Development**

We can create a new website adding all type of properties with real time data and accuracy of ease to use for the users.

**8. Conclusion**

The Real Estate Management System effectively showcases the integration of both core and advanced Java concepts to address practical, real-world challenges. By overcoming initial development hurdles and adhering to modern programming best practices, the project achieves its goals and establishes a solid framework for future enhancements and scalability.