# Property portfolio management system

**1. Introduction**

I'll go over the planning, execution, and evaluation of a property management program in this report. Three classes are offered in the program: Property, MarketProperty, and PropertyManagementCompany. I'll concentrate on the OOP characteristics and ideas used in the design, like encapsulation and the single responsibility concept.

2**. Encapsulation**

A key component of OOP, encapsulation fosters data abstraction and offers a means for data security and controlled access. It entails combining data and the methods that manipulate that data into a class, essentially shielding the class' internals from intervention.

Encapsulation is efficiently used in our property management program to maintain the required behavior of the classes and guarantee data integrity. Let's examine the implementation of encapsulation in each class:

**2.1 Property Class:** The Property class encapsulates the id, type, and size properties of a general property. These properties are available exclusively within the class itself because they are defined as private instance variables. There are getter and setter methods available to interact with these variables:

public class Property {

private int id; // Unique identifier for the property

private String type; // Type or category of the property

private double size; // Size of the property in square meters

/\*\*

\* Constructs a Property object with the specified id, type, and size.

\*

\* **@param** id The unique identifier of the property.

\* **@param** type The type or category of the property.

\* **@param** size The size of the property in square meters.

\*/

public Property(int id, String type, double size) {

this.id = id;

this.type = type;

this.size = size;

}

Direct access to the id, type, and size variables from outside the class is avoided because to their encapsulation. Instead, use the public methods that are offered to communicate with the Property class. As a result, it is possible to have more control over how the properties are accessed and changed. To ensure data consistency and adherence to business standards, we can also incorporate extra logic, like as validation or constraints, within the getter and setter methods.

**2.2 MarketProperty Class:** The MarketProperty class extends the Property class and encapsulates additional attributes specific to market properties, such as **initialPrice** and **currentValuation**. It inherits the encapsulated attributes (**id**, **type**, and **size**) from the Property class and provides its own getter and setter methods for the new attributes:

public class MarketProperty extends Property {

private double initialPrice; // Initial pricing of the property

private double currentValuation; // Current valuation of the property

/\*\*

\* Constructs a MarketProperty object with the specified id, type, size, and initial price.

\*

\* **@param** id The unique identifier of the property.

\* **@param** type The type or category of the property.

\* **@param** size The size of the property in square meters.

\* **@param** initialPrice The initial pricing of the property.

\*/

public MarketProperty(int id, String type, double size, double initialPrice) {

super(id, type, size);

this.initialPrice = initialPrice;

this.currentValuation = initialPrice;

}

By encapsulating the **initialPrice** and **currentValuation** attributes within the MarketProperty class, these values are encapsulated and hidden from direct access. The class provides public methods to get and set these attributes, allowing controlled access while maintaining data integrity.

**2.3 PropertyManagementCompany Class:** The management of the property portfolio is encapsulated by the PropertyManagementCompany class. It contains enclosed private instance variables included within the class, such as name, liquidity, and portfolio. Getter methods are used to limit access to these variables:

public class PropertyManagementCompany {

private String name; // Name of the property management company

private double liquidity; // Amount of money currently held by the company

private List<MarketProperty> portfolio; // Portfolio of MarketProperty objects

/\*\*

\* Constructs a PropertyManagementCompany object with the specified name and an empty portfolio.

\*

\* **@param** name The name of the property management company.

\*/

public PropertyManagementCompany(String name) {

this.name = name;

this.liquidity = 0.0;

this.portfolio = new ArrayList<>();

The name, liquidity, and portfolio variables are all contained, preventing direct alteration or access from outside the class. In order to ensure regulated read-only access, the class offers getter methods to access these data.

**3. Single Responsibility Principle**

The Single Responsibility Principle (SRP) states that a class should have a single responsibility and only one reason to change. It promotes modular and maintainable code by ensuring that each class is focused on performing a specific task.

In our program, each class adheres to the SRP:

* The Property class is responsible for representing a property with its attributes, such as ID, type, and size. It encapsulates the data and provides getter and setter methods for accessing and modifying the properties.
* The MarketProperty class extends the Property class and adds additional properties specific to market properties, such as initialPrice and currentValuation. It also includes methods for updating valuation and calculating profits. By extending the Property class, it inherits its properties and behavior, which promotes code reuse.
* The PropertyManagementCompany class manages a property portfolio, including buying, selling, and updating valuations. It maintains a list of MarketProperty objects and provides methods for managing the portfolio and calculating profit. The class has separate methods for each responsibility, adhering to the SRP.

**2.2 Testing Description**

Testing is a crucial aspect of software development as it ensures the correctness, reliability, and robustness of the program. In our property management program, we performed comprehensive testing to verify the functionality of the different parts of the program. We focused on testing the Property, MarketProperty, and PropertyManagementCompany classes, covering various functionalities and considering potential corner cases.

**Testing the Property Class:**

1. Testing Getter and Setter Methods: We tested the getter and setter methods of the Property class to ensure they correctly retrieve and modify the property attributes. We created a Property object, set different values using the setter methods, and then retrieved the values using the corresponding getters. We verified that the retrieved values matched the set values.
2. // Create properties
3. Property property1 = new Property(1, "Studio", 15);
4. Property property2 = new Property(2, "Apartment", 100);
5. Property property3 = new Property(3, "House", 70);
6. property3.setId(2);
7. property3.setType("Apartment");
8. property3.setSize(80.0);
9. int id = property3.getId();
10. String type = property3.getType();
11. double size = property3.getSize();
12. System.***out***.println("Id: " + id);
13. System.***out***.println("Type: " + type);
14. System.***out***.println("Size: " + size);
15. Testing Object Initialization: We tested the object initialization of the Property class to ensure that the provided values are correctly assigned to the attributes. We created Property objects with different constructor arguments and validated that the attribute values were set accordingly.
16. // Create market properties
17. MarketProperty marketProperty1 = new MarketProperty(4, "Apartment", 80, 200000);
18. MarketProperty marketProperty2 = new MarketProperty(5, "House", 150, 500000);

**Testing the MarketProperty Class:**

1. Testing Inheritance: We tested the inheritance relationship between the Property and MarketProperty classes. We created a MarketProperty object and verified that it inherited the attributes and methods from the Property class. We accessed and modified the inherited attributes using the getter and setter methods.
2. // Create market properties
3. MarketProperty marketProperty1 = new MarketProperty(4, "Apartment", 80, 200000);
4. MarketProperty marketProperty2 = new MarketProperty(5, "House", 150, 500000);
6. int idMarket = marketProperty2.getId();
7. String typeMarket = marketProperty2.getType();
8. double sizeMarket = marketProperty2.getSize();
9. double initialPrice = marketProperty2.getInitialPrice();
10. Testing Valuation Update: We tested the updateValuation method of the MarketProperty class, which calculates the current valuation based on inflation rate and volatility coefficient. We created a MarketProperty object, called the updateValuation method with different input values, and verified that the current valuation was updated correctly.
11. // Create market properties
12. MarketProperty marketProperty1 = new MarketProperty(4, "Apartment", 80, 200000);
13. MarketProperty marketProperty2 = new MarketProperty(5, "House", 150, 500000);
15. int idMarket = marketProperty2.getId();
16. String typeMarket = marketProperty2.getType();
17. double sizeMarket = marketProperty2.getSize();
18. double initialPrice = marketProperty2.getInitialPrice();
20. marketProperty2.updateValuation(0.05, 0.1);

**Testing the PropertyManagementCompany Class:**

1. Testing Buying Properties: We tested the buyProperty method of the PropertyManagementCompany class, which adds a MarketProperty to the company's portfolio. We created a PropertyManagementCompany object, called the buyProperty method with different inputs, and verified that the property was successfully added to the portfolio and the liquidity was updated accordingly.
2. // Create property management company
3. PropertyManagementCompany company = new PropertyManagementCompany("ABC Property Management");
4. // Add properties to the company's portfolio
5. company.buyProperty(marketProperty1, 200000);
6. company.buyProperty(marketProperty2, 500000);
7. // Print portfolio details
8. System.***out***.println("Company Name: " + company.getName());
9. System.***out***.println("Liquidity: $" + company.getLiquidity());
10. System.***out***.println("Portfolio Size: " + company.countProperties());
11. System.***out***.println("\nProperties in Portfolio:");
12. List<MarketProperty> portfolio = company.getPortfolio();
13. for (MarketProperty property : portfolio) {
14. System.***out***.println("Property ID: " + property.getId());
15. System.***out***.println("Type: " + property.getType());
16. System.***out***.println("Size: " + property.getSize() + " sqm");
17. System.***out***.println("Initial Price: $" + property.getInitialPrice());
18. System.***out***.println("Current Valuation: $" + property.getCurrentValuation());
19. System.***out***.println();
20. }
21. // Update valuations
22. company.updateAllValuations(0.05, 0.1);
23. System.***out***.println("Updated Portfolio Valuations:");
24. for (MarketProperty property : portfolio) {
25. System.***out***.println("Property ID: " + property.getId());
26. System.***out***.println("Current Valuation: $" + property.getCurrentValuation());
27. System.***out***.println();
28. }
29. Testing Selling Properties: We tested the sellProperty method of the PropertyManagementCompany class, which removes a MarketProperty from the company's portfolio. We created a PropertyManagementCompany object, added properties to the portfolio, called the sellProperty method with different properties, and verified that the property was successfully removed from the portfolio and the liquidity was updated accordingly.
30. // Sell a property and calculate potential profit
31. if (!portfolio.isEmpty()) {
32. MarketProperty propertyToSell = portfolio.get(0);
33. double potentialProfit = company.getPotentialProfit(propertyToSell);
34. System.***out***.println("Potential Profit from Selling Property ID " + propertyToSell.getId() + ": $" + potentialProfit);
35. // Sell the property
36. company.sellProperty(propertyToSell);
37. // Print updated portfolio details
38. System.***out***.println("\nUpdated Portfolio Size: " + company.countProperties());
39. System.***out***.println("Updated Liquidity: $" + company.getLiquidity());
40. } else {
41. System.***out***.println("Portfolio is empty. No properties to sell.");
42. }
43. }

**2.3 Improvements**

While the property management program we have developed provides a solid foundation for managing properties, there are several areas where improvements can be made to enhance its functionality and reflect a more accurate representation of a real-world system. Below are two possible improvements:

1. **Enhanced Financial Tracking and Reporting**: Currently, the program focuses primarily on property acquisition, valuation updates, and selling properties. However, financial tracking and reporting are crucial components of a real-world property management system. We could add functions like tracking rental revenue, spending, and producing financial reports to the system to make it better. Property managers would thus be able to get a complete picture of their portfolio's financial features, including cash flow, profitability, and return on investment. A more realistic picture of the financial impact of property ownership would be possible with the addition of capability for tracking property expenses such maintenance fees, property taxes, and insurance.
2. **Tenant Management**: Tenant management is important in a real property management system. We may add capabilities for handling tenant information, such as lease agreements, tenant screening, rent collection, and maintenance requests, to the program to improve it. This would make it possible for property managers to manage tenant-related duties quickly while maintaining smooth communication, rent collection, and property upkeep. Additionally, adding tools for managing lease renewals, tracking lease expirations, and creating tenant-related reports would give property managers useful information about occupancy rates, lease terms, and rental income.
3. **Justification:**
4. The addition of improved financial tracking and reporting features would offer real-time information to property managers about the financial health of their properties. Property managers may make informed judgments about buying properties, setting prices, and managing their entire portfolio by keeping track of rental income, spending, and producing financial reports. This improvement is in line with what is actually needed to manage properties profitably in the real world.
5. A crucial component of property management is tenant management. The solution would be more in line with actual property management procedures by including tenant management elements. Property managers would be able to increase communication, streamline tenant-related chores, and guarantee prompt rent collection and upkeep of the property. This upgrade improves the program's usefulness and usability for property managers and reflects the all-encompassing nature of property management.

By putting these changes into practice, a property management system that addresses important facets of budget tracking and tenant management will become more thorough and useful. With these improvements, property managers would be able to simplify their operations, have a more realistic representation of their actual property management obligations, and make choices based on solid financial and tenant-related data.