TUNIS BUSINESS SCHOOL
UNIVERSITY OF TUNIS

جـــامعــة تــــونـــس
1960

Data Analysis Management System Using Object-Oriented Programming: Melbourne Housing Market

Project Report - IT 320 Object-Oriented Programming

BY
Asma Boubaker
Kenza Bacha
Seif Ben Soltana
Majd Hamdi

Professor
Dr. Ameni Azzouz

Academic Year
2024 - 2025

Contents

1 Introduction

1.1 Project Overview

This project focuses on developing a comprehensive data analysis management system tailored for the real estate domain, mainly utilizing Java and object-oriented programming (OOP) principles. The system is designed to handle and analyze housing-related data to assist agencies in gaining valuable insights into market trends and buyer behaviors in Melbourne, Australia.

The system allows users to seamlessly upload and store detailed property data, including information about properties, their locations, agencies, and buyers. It enables advanced data exploration, including descriptive statistics, visualizations, and correlation analyses, to uncover patterns and relationships in the data. Machine learning models, such as predictive analytics and classification, are integrated to forecast prices, identify purchasing trends, and segment the market effectively.

Additionally, the system offers robust reporting and dashboarding functionalities, presenting the analysis results in an easy and interactive manner. This empowers real estate agencies to make data-driven decisions, facilitate analysis, and gain a competitive edge in the market. The combination of advanced analytics and user-friendly interface ensures the system serves as a valuable tool for real estate market analysis.

1.2 Problem Statement

Real estate agencies often struggle to analyze large volumes of property data to understand market trends, predict prices, and make data-driven decisions. This project addresses the need for an efficient, user-friendly system to upload, manage, analyze, and visualize housing data, enabling agencies to uncover insights, forecast trends, and improve decision-making in a competitive market.

1.3 Project Scope

This project aims to create a customizable data analysis management system tailored for housing-related data. The system will include the following features:

Data Ingestion: Enable seamless import of property data from various sources, such as CSV and Excel files. Provide robust data cleaning and preprocessing tools to handle missing values, outliers, and data types. Store data in an efficient relational database structure for easy access and management.

Data Exploration and Analysis: Offer comprehensive exploration tools, including descriptive statistics (e.g., mean, median, mode), property-specific visualizations (e.g., sales trends, buyer demographics), and correlation analysis to identify relationships between variables (e.g., location and price).

Machine Learning Integration: Allow integration of predictive modeling algorithms to classify property types and forecast property prices and sales patterns, regression analysis for price prediction, and time series analysis for market trends aiding in decision-making for agencies.

Reporting and Dashboards: Provide dynamic and interactive dashboards for agencies to monitor purchasing trends, analyze market behavior, and generate reports with graphs and key metrics to provide data-driven insights.

The system will handle diverse data types (e.g., numerical, categorical, dates) and will be designed with flexibility to adapt to various property data characteristics and analysis needs.

2 Requirements Analysis

2.1 Functional Requirements

User Interface Requirements The interface will allow users to:

- **Upload Data:** Import property data from CSV or Excel files using libraries such as Apache POI (Excel handling) and OpenCSV (CSV parsing).
- **Data Exploration:** Perform descriptive analysis (mean, median) using Apache Commons Math, visualize trends with JFreeChart or JavaFX Charts, and compute correlations with JAMA (Matrix Library).
- **Machine Learning Analysis:** Integrate predictive models like regression and classification using Weka or Apache Spark MLlib.
- **Reporting and Dashboarding:** Generate interactive reports and dashboards with JavaFX or Jasperreports for reporting, combined with visualization libraries like JFreeChart.

The interface will be developed using Java OOP principles and incorporate libraries tailored for file handling, data analysis, visualization, and machine learning.

Data Storage Requirements The system will support data storage in CSV and XLS file formats, utilizing Apache POI and OpenCSV for file handling. Data will be uploaded, cleaned, and stored in a MySQL database using JDBC for seamless database integration. This ensures efficient handling of structured data for analysis and reporting.

Data Analysis Requirements The system will perform statistical analysis, including descriptive statistics (mean, median, standard deviation) and correlation analysis, to uncover relationships between property attributes. It will support data mining techniques, such as regression analysis for price prediction and clustering for market segmentation, tailored to the real estate domain. Libraries like Apache Commons Math and Weka will facilitate these analyses.

Visualization Requirements The system will generate interactive visualizations, including histograms, pie charts, bar charts, scatter plots, and line charts, based on user-selected data columns and filters. Users can customize visualizations to focus on specific attributes, enabling deeper insights into property sales trends. These visualizations will be implemented using libraries like JFreeChart or JavaFX Charts, and will integrate seamlessly with the reporting and dashboard modules to support data-driven decision-making.

2.2 Non-Functional Requirements

Performance Requirements

- **Response Time:**
  - The system will process data ingestion, including file uploads and integration with existing data, within 5 seconds.
  - Generating visualizations, such as charts and graphs, should take no more than 3 seconds.
  - Performing descriptive statistics will be completed within 2 seconds.
  - Machine learning predictions (e.g., regression or classification) will execute within 5 seconds.
- **Scalability:** The system will support additional data being added seamlessly while maintaining smooth operation. All functionalities will remain responsive regardless of data size increases.
- **Efficiency:**
  - Data cleaning tasks, such as removing missing values, duplicates, and outliers, will complete in under 5 seconds.
  - Exporting data to external formats (TXT or SQL) must be complete within 5 seconds.

Security Requirements Appropriate mechanisms will be implemented to safeguard data integrity and confidentiality.

Usability Requirements

- **User-Friendly Interface:** A clean and intuitive graphical interface will enable users to perform tasks such as uploading data, running analysis, and generating reports without extensive technical knowledge.
- **Intuitive Navigation:** All functionalities (e.g., data ingestion, exploration, visualization, reporting) will be accessible via a well-structured dashboard.
- **Interactive Visualizations:** Customizable charts and graphs will allow users to tailor outputs for specific analysis needs.
- **Error Handling:** Clear error messages will inform users of invalid file formats, failed operations, or missing inputs.
- **Customizable Reports:** The system will allow users to generate dynamic reports and dashboards, including visualizations and summary statistics, to meet their specific business or analytical needs.

3 System Design

3.1 Overall System Architecture

The system is designed with an architecture consisting of five key layers that interact with each other to achieve the goal of analyzing and managing property sales data. Below is a high-level breakdown of the layers and their responsibilities.

3.1.1 Business Logic Layer

This layer is responsible for handling all the core business logic of the system. It contains classes representing the system's data entities and their relationships, as well as the rules governing their behavior.

Key Classes

- **Property**: A parent class for all property types.
- **NonResidentialProperty, ResidentialProperty**: Specialized subclasses of **Property**.
- **House, Unit, Townhouse, DevelopmentSite**: Specific types of residential or non-residential properties.
- **Transaction, Buyer, Agent, Location**: Supporting entities representing transaction details, participants, and property location.

Responsibilities The business logic layer ensures that the data entities are properly structured and interact according to business rules (e.g., handling property details, managing transactions).

OOP Relationships

- **Inheritance:** The `Property` class serves as the parent class, with `ResidentialProperty` and `NonResidentialProperty` inheriting from it.
- **Composition:** Each `Property` must have an associated `Location`.
- **Association:** Objects such as `Property`, `Buyer`, and `Agent` are connected but can exist independently.
- **Encapsulation:** Each object hides its internal state and restricts direct access to its attributes.

```java
public abstract class Property {
    private int propertyId;
    private int landSize;
    private Location location;

    public Property(int propertyId, int landSize, Location location)
        {
        this.propertyId = propertyId;
        this.landSize = landSize;
        this.location = location;
    }

    public void displayBasicDetails() {
        System.out.println("Property ID: " + propertyId);
        System.out.println("Land Size: " + landSize + " sqm");
        System.out.println("Location: " + location.getRegionName());
    }

    public abstract String getPropertyType();
}
```

Listing 1: Example of Property Class in Java

3.1.2 Data Ingestion and Handling Layer

This layer is responsible for managing the import, cleaning, and export of data. It ensures data is properly handled, transformed, and stored for use in other layers.

Key Classes

- **XLSIngestion**: Imports data from XLS files.
- **CSVIngestion**: Similar to XLSIngestion but handles CSV files.
- **DataCleaning (Interface)**: Declares methods for cleaning data.
- **PropertyDataCleaner**: Implements the **DataCleaning** interface.
- **ExportData**: Exports cleaned data to external files or databases.

```java
public interface DataCleaning {
    void removeMissingValues(List<List<Object>> data);
    void removeDuplicates(List<List<Object>> data);
    void removeOutliers(List<List<Object>> data);
    void fixYear(List<List<Object>> data);
}
```

Listing 2: DataCleaning Interface Example

```java
public class PropertyDataCleaner implements DataCleaning {
    @Override
    public void removeMissingValues(List<List<Object>> data) {
        // Implementation
    }

    @Override
    public void removeDuplicates(List<List<Object>> data) {
        // Implementation
    }

    @Override
    public void removeOutliers(List<List<Object>> data) {
        // Implementation
    }

    @Override
    public void fixYear(List<List<Object>> data) {
        // Implementation
    }
}
```

Listing 3: PropertyDataCleaner Implementation

3.1.3 Data Exploration and Analysis Layer

This layer focuses on retrieving, analyzing, and visualizing data to uncover insights and relationships.

Key Classes

- **DataExploration (Abstract Class)**: Holds common attributes for data analysis.
- **DescriptiveAnalysis**: Performs statistical analysis (e.g., mean, median).
- **DataVisualization**: Generates visual representations of data.
- **CorrelationAnalysis**: Conducts statistical tests such as Pearson, Chi-Square, and Spearman.

```java
public abstract class DataExploration {
    public static List<Object> fetchColumn(String tableName, String
        columnName) {
        // Fetch data from a database column
    }
}
```

Listing 4: DataExploration Abstract Class

```java
public class DescriptiveAnalysis extends DataExploration {
    public String summary(List<?> column) {
        // Return summary statistics
    }
}
```

Listing 5: Descriptive Analysis Implementation

3.1.4 Machine Learning Layer

This layer provides machine learning capabilities for predictive modeling and classification.

Key Interfaces and Classes

- **MachineLearning (Interface)**: Contains methods for fitting, predicting, and evaluating models.
- **PredictiveModeling (Interface)**: For regression and time-series analysis.
  - Linear Regression (class)
  - TSA Analysis (class)
- **ClassificationModeling (Interface)**: For classification tasks.
  - Kmeans Clustering (class)
  - KNN Clustering (class)

```java
public interface MachineLearning {
    void fit(List<?> data);
    List<?> predict(List<?> data);
    double evaluate(List<?> testData);
}
```

Listing 6: MachineLearning Interface Example

3.1.5 Reporting and Dashboarding System

This module provides an interactive dashboard for real-time data visualization, enabling users to explore property transaction insights dynamically. It supports multiple chart types, including bar, line, pie, scatter, and donut charts, offering an intuitive graphical representation of market trends and buyer demographics. The dashboard is handled separately from the core system, ensuring a flexible and modular approach.

Key Interfaces and Classes

- `ChartAdminPanel` - The main class responsible for initializing and managing the dashboard interface.
- `ChartPanel` - A component from the JFreeChart library used to embed and display charts.
- `JPanel` - Various panels (e.g., `jPanel6`, `jPanel7`) serve as containers for different chart visualizations.
- `JTable` - Used to display structured tabular property data.
- Database connectivity classes utilizing JDBC for fetching and processing real estate data.

Responsibilities This module is responsible for:

- Retrieving real estate data from a MySQL database.
- Creating and displaying interactive charts, including:
  - **Scatter Plot**: Analyzes price vs. distance relationships.
  - **Pie Chart**: Shows property distribution by the number of rooms.
  - **Bar Chart**: Visualizes price variations based on transaction methods.
  - **Line Chart**: Represents property counts over different construction year ranges.
  - **Donut Chart**: Illustrates buyer gender distribution.
- Presenting property details in a structured table format.
- Ensuring modular and separate handling of the dashboard from the core system.

OOP Relationships

- `ChartAdminPanel` acts as the controller, orchestrating chart creation and data retrieval.
- Utilizes Java Swing components (`JPanel`, `JTable`) for UI rendering.
- Integrates with JFreeChart for visualization.
- Establishes JDBC connections for database queries and data extraction.

```java
public class ChartAdminPanel extends javax.swing.JFrame {

    public ChartAdminPanel() {
        initComponents();
        createScatterPlot();
        createPieChart();
        generateBarChart();
        generateLineChartWithYearRanges();
        createDonutChart();
        displayPropertyTable();
    }
}
```

Listing 7: ChartAdminPanel Class

```
private void createScatterPlot() {
    String jdbcURL = "jdbc:mysql://localhost:3306/java_db";
    String username = "root";
    String password = "projetIT2025*";

    String query = """
        SELECT t.price, COALESCE(h.distance, u.distance, th.distance
           , d.distance) AS distance
        FROM transactions t
        LEFT JOIN houses h ON t.propertyId = h.propertyId
        LEFT JOIN units u ON t.propertyId = u.propertyId
        LEFT JOIN townhouses th ON t.propertyId = th.propertyId
        LEFT JOIN developmentSite d ON t.propertyId = d.propertyId
        WHERE t.price IS NOT NULL AND
              (h.distance IS NOT NULL OR u.distance IS NOT NULL OR
                 th.distance IS NOT NULL OR d.distance IS NOT NULL);
    """;

    XYSeries series = new XYSeries("Price vs Distance");

    try (Connection connection = DriverManager.getConnection(jdbcURL
        , username, password);
         Statement statement = connection.createStatement();
         ResultSet resultSet = statement.executeQuery(query)) {

        while (resultSet.next()) {
            double price = resultSet.getDouble("price");
            double distance = resultSet.getDouble("distance");
            series.add(distance, price);
        }
    } catch (SQLException e) {
        System.out.println("Database error: " + e.getMessage());
        return;
    }

    XYSeriesCollection dataset = new XYSeriesCollection(series);
    JFreeChart scatterPlot = ChartFactory.createScatterPlot(
            "Scatter Plot: Price vs Distance",
            "Distance (km)",
            "Price ($)",
            dataset,
            PlotOrientation.VERTICAL,
            true, true, false
    );

    ChartPanel chartPanel = new ChartPanel(scatterPlot);
    jPanel7.setLayout(new java.awt.BorderLayout());
    jPanel7.add(chartPanel, java.awt.BorderLayout.CENTER);
    jPanel7.validate();
}
```
Listing 8: Example of Chart Creation Method

4 Class Diagrams

4.1 Property Management System

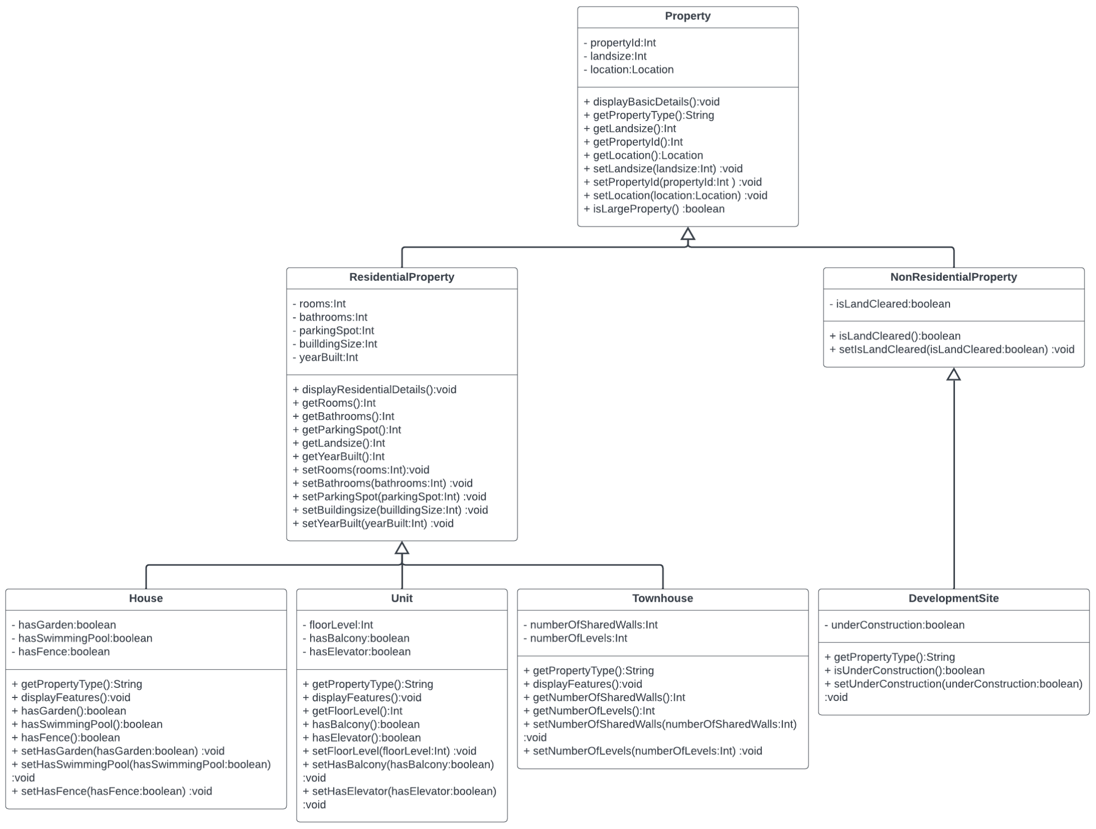

Property
- propertyId:Int
- landsize:Int
- location:Location
+ displayBasicDetails():void
+ getPropertyType():String
+ getLandsize():Int
+ getPropertyId():Int
+ getLocation():Location
+ setLandsize(landsize:Int) :void
+ setPropertyId(propertyId:Int) :void
+ setLocation(location:Location) :void
+ isLargeProperty() :boolean
ResidentialProperty
- rooms:Int
- bathrooms:Int
- parkingSpot:Int
- builldingSize:Int
- yearBuilt:Int
+ displayResidentialDetails():void
+ getRooms():Int
+ getBathrooms():Int
+ getParkingSpot():Int
+ getLandsize():Int
+ getYearBuilt():Int
+ setRooms(rooms:Int):void
+ setBathrooms(bathrooms:Int) :void
+ setParkingSpot(parkingSpot:Int) :void
+ setBuildingsize(builldingSize:Int) :void
+ setYearBuilt(yearBuilt:Int) :void
NonResidentialProperty
- isLandCleared:boolean
+ isLandCleared():boolean
+ setIsLandCleared(isLandCleared:boolean) :void
House
- hasGarden:boolean
- hasSwimmingPool:boolean
- hasFence:boolean
+ getPropertyType():String
+ displayFeatures():void
+ hasGarden():boolean
+ hasSwimmingPool():boolean
+ hasFence():boolean
+ setHasGarden(hasGarden:boolean) :void
+ setHasSwimmingPool(hasSwimmingPool:boolean) :void
+ setHasFence(hasFence:boolean) :void
Unit
- floorLevel:Int
- hasBalcony:boolean
- hasElevator:boolean
+ getPropertyType():String
+ displayFeatures():void
+ getFloorLevel():Int
+ hasBalcony():boolean
+ hasElevator():boolean
+ setFloorLevel(floorLevel:Int) :void
+ setHasBalcony(hasBalcony:boolean) :void
+ setHasElevator(hasElevator:boolean) :void
Townhouse
- numberOfSharedWalls:Int
- numberOfLevels:Int
+ getPropertyType():String
+ displayFeatures():void
+ getNumberOfSharedWalls():Int
+ getNumberOfLevels():Int
+ setNumberOfSharedWalls(numberOfSharedWalls:Int) :void
+ setNumberOfLevels(numberOfLevels:Int) :void
DevelopmentSite
- underConstruction:boolean
+ getPropertyType():String
+ isUnderConstruction():boolean
+ setUnderConstruction(underConstruction:boolean) :void

Figure 1: Class Diagram: Property Management System

Description: The diagram represents a hierarchical structure for managing property types, starting with a base *Property* class that encapsulates common attributes like `propertyId`, `landSize`, and `location`. It branches into abstract subclasses (*ResidentialProperty* and *NonResidentialProperty*), which further specialize into concrete classes such as *House*, *Unit*, *Townhouse*, and *DevelopmentSite*, each with their unique attributes and behaviors.

Object-Oriented Principles

1. **Inheritance:**
   - The diagram demonstrates a classic hierarchical inheritance structure:
     - *Property* is the parent class.
     - *ResidentialProperty* and *NonResidentialProperty* are abstract subclasses that inherit shared attributes and methods.

- Concrete classes (*House*, *Unit*, *Townhouse*, *DevelopmentSite*) inherit functionality from their respective parents while adding their unique properties and methods.

- **Example:** The `displayBasicDetails()` method is inherited by all subclasses, ensuring reusability of shared functionality.

2. **Polymorphism:**

   - **Method Overriding:** Each subclass implements its version of `getPropertyType()`, returning a specific property type like `"House"` or `"Unit"`.
   - This allows dynamic behavior where the `getPropertyType` method can be called on any `Property` object, regardless of its specific type.
   - **Example:** A `Property` object could point to a `House` or `DevelopmentSite` instance, and calling `getPropertyType()` will produce type-specific behavior.

3. **Abstraction:**

   - The `Property` and `ResidentialProperty` classes encapsulate shared functionality while hiding implementation details from higher levels.
   - The use of abstract methods like `getPropertyType` enforces that every subclass provides its specific implementation.
   - **Example:** The `isLargeProperty()` method in `Property` is defined for all subclasses, abstracting the logic for determining size across all property types.

4. **Encapsulation:**

   - All attributes are private, ensuring controlled access via public getter and setter methods.
   - **Example:** The `landSize` attribute in `Property` can only be accessed or modified using `getLandSize()` and `setLandSize(double)` methods, ensuring proper validation and data integrity.

4.2 Relationships and Multiplicities

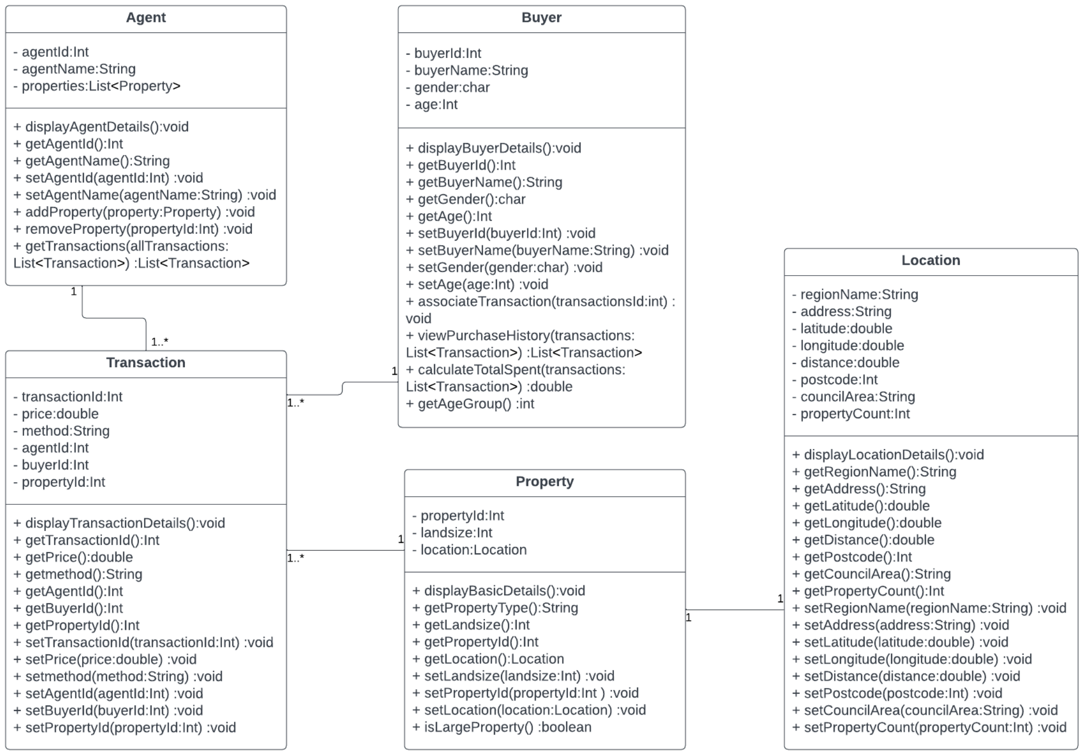

Agent
- agentId:Int
- agentName:String
- properties:List<Property>
+ displayAgentDetails():void
+ getAgentId():Int
+ getAgentName():String
+ setAgentId(agentId:Int) :void
+ setAgentName(agentName:String) :void
+ addProperty(property:Property) :void
+ removeProperty(propertyId:Int) :void
+ getTransactions(allTransactions: List<Transaction>) :List<Transaction>
Buyer
- buyerId:Int
- buyerName:String
- gender:char
- age:Int
+ displayBuyerDetails():void
+ getBuyerId():Int
+ getBuyerName():String
+ getGender():char
+ getAge():Int
+ setBuyerId(buyerId:Int) :void
+ setBuyerName(buyerName:String) :void
+ setGender(gender:char) :void
+ setAge(age:Int) :void
+ associateTransaction(transactionsId:int) : void
+ viewPurchaseHistory(transactions: List<Transaction>) :List<Transaction>
+ calculateTotalSpent(transactions: List<Transaction>) :double
+ getAgeGroup() :int
Location
- regionName:String
- address:String
- latitude:double
- longitude:double
- distance:double
- postcode:Int
- councilArea:String
- propertyCount:Int
+ displayLocationDetails():void
+ getRegionName():String
+ getAddress():String
+ getLatitude():double
+ getLongitude():double
+ getDistance():double
+ getPostcode():Int
+ getCouncilArea():String
+ getPropertyCount():Int
+ setRegionName(regionName:String) :void
+ setAddress(address:String) :void
+ setLatitude(latitude:double) :void
+ setLongitude(longitude:double) :void
+ setDistance(distance:double) :void
+ setPostcode(postcode:Int) :void
+ setCouncilArea(councilArea:String) :void
+ setPropertyCount(propertyCount:Int) :void
1
1..*
Transaction
- transactionId:Int
- price:double
- method:String
- agentId:Int
- buyerId:Int
- propertyId:Int
+ displayTransactionDetails():void
+ getTransactionId():Int
+ getPrice():double
+ getmethod():String
+ getAgentId():Int
+ getBuyerId():Int
+ getPropertyId():Int
+ setTransactionId(transactionId:Int) :void
+ setPrice(price:double) :void
+ setmethod(method:String) :void
+ setAgentId(agentId:Int) :void
+ setBuyerId(buyerId:Int) :void
+ setPropertyId(propertyId:Int) :void
1
1..*
Property
- propertyId:Int
- landsize:Int
- location:Location
+ displayBasicDetails():void
+ getPropertyType():String
+ getLandsize():Int
+ getPropertyId():Int
+ getLocation():Location
+ setLandsize(landsize:Int) :void
+ setPropertyId(propertyId:Int) :void
+ setLocation(location:Location) :void
+ isLargeProperty() :boolean
1
1..*
1
1

Figure 2: Class Diagram: Relationships and Multiplicities

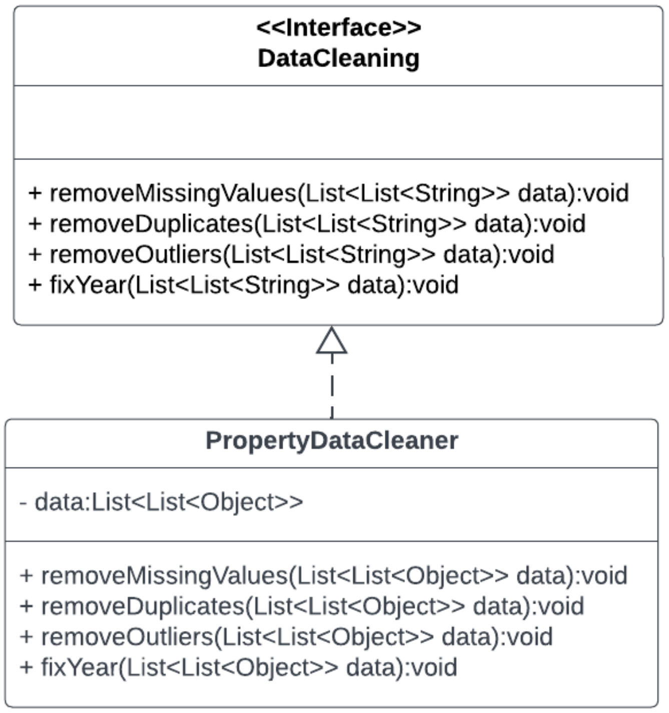
Description: The diagram illustrates key relationships and multiplicities in the real estate management system, emphasizing how different entities (Agent, Buyer, Transaction, Property, and Location) are interconnected.

Relationships and Multiplicities

1. **Agent to Transaction:**
   - **Multiplicity:** 1 Agent is associated with 1..* Transactions.
   - **Description:** An agent handles multiple transactions, but each transaction is associated with a single agent. This is modeled using the `getTransactions` method in the `Agent` class.
2. **Transaction to Buyer:**
   - **Multiplicity:** 1..* Transactions are associated with 1 Buyer.
   - **Description:** A buyer can participate in multiple transactions, but each transaction involves a single buyer. This relationship is captured through the `buyerId` attribute in the `Transaction` class.

3. **Transaction to Property:**
   - **Multiplicity:** 1..* Transactions are associated with 1 Property.
   - **Description:** A property can be part of multiple transactions, but each transaction involves a single property. This is reflected in the `propertyId` attribute of the `Transaction` class.
4. **Property to Location:**
   - **Multiplicity:** 1 Property is associated with 1 Location.
   - **Description:** Each property is tied to a specific geographical location. This relationship is modeled using the `location` attribute in the `Property` class.

4.3 Data Cleaning



<<Interface>>
DataCleaning
+ removeMissingValues(List<List<String>> data):void
+ removeDuplicates(List<List<String>> data):void
+ removeOutliers(List<List<String>> data):void
+ fixYear(List<List<String>> data):void
PropertyDataCleaner
- data:List<List<Object>>
+ removeMissingValues(List<List<Object>> data):void
+ removeDuplicates(List<List<Object>> data):void
+ removeOutliers(List<List<Object>> data):void
+ fixYear(List<List<Object>> data):void

Figure 3: Class Diagram: Data Cleaning

Overview:

- **Interface: DataCleaning:**
  - Defines four abstract methods:
    * `removeMissingValues`

    * `removeDuplicates`
    * `removeOutliers`
    * `fixYear`
  - These methods specify the expected behavior for data cleaning but do not provide implementations.

- **Class: PropertyDataCleaner:**
  - Implements the `DataCleaning` interface.
  - Contains the `data` attribute (a list of lists of objects) to store data to be cleaned.
  - Provides concrete implementations for the abstract methods defined in `DataCleaning`.

Object-Oriented Programming Concepts:

- **Inheritance:**
  - The `PropertyDataCleaner` class implements the `DataCleaning` interface, inheriting its method signatures.
- **Polymorphism:**
  - The `PropertyDataCleaner` class can be referred to as a `DataCleaning` type, allowing flexibility in handling objects using the interface.
- **Abstraction:**
  - The `DataCleaning` interface abstracts the concept of data cleaning, providing a blueprint for all classes that aim to implement these functionalities.
- **Encapsulation:**
  - The `data` attribute in `PropertyDataCleaner` is private, ensuring that access to the raw data is controlled through the cleaning methods.

4.4 Chart Administration Panel

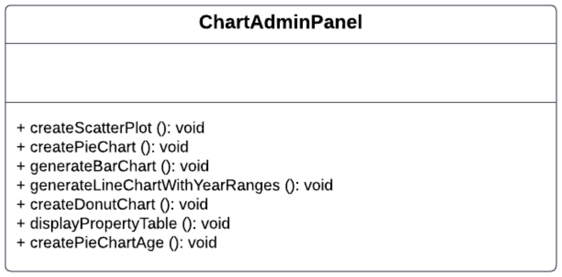

ChartAdminPanel
+ createScatterPlot (): void
+ createPieChart (): void
+ generateBarChart (): void
+ generateLineChartWithYearRanges (): void
+ createDonutChart (): void
+ displayPropertyTable (): void
+ createPieChartAge (): void

Figure 4: Class Diagram: Chart Administration Panel

Overview: The `ChartAdminPanel` class provides methods for generating and displaying various types of charts and data tables.

Methods:

- **Chart Creation:**
  - `createScatterPlot()`
  - `createPieChart()`
  - `generateBarChart()`
  - `generateLineChartWithYearRanges()`
  - `createDonutChart()`
  - `createPieChartAge()`
- **Data Display:**
  - `displayPropertyTable()`

These methods enable visualization and analysis of data through different chart formats and tables for administrative purposes.

4.5 Data Ingestion and Handling Layer

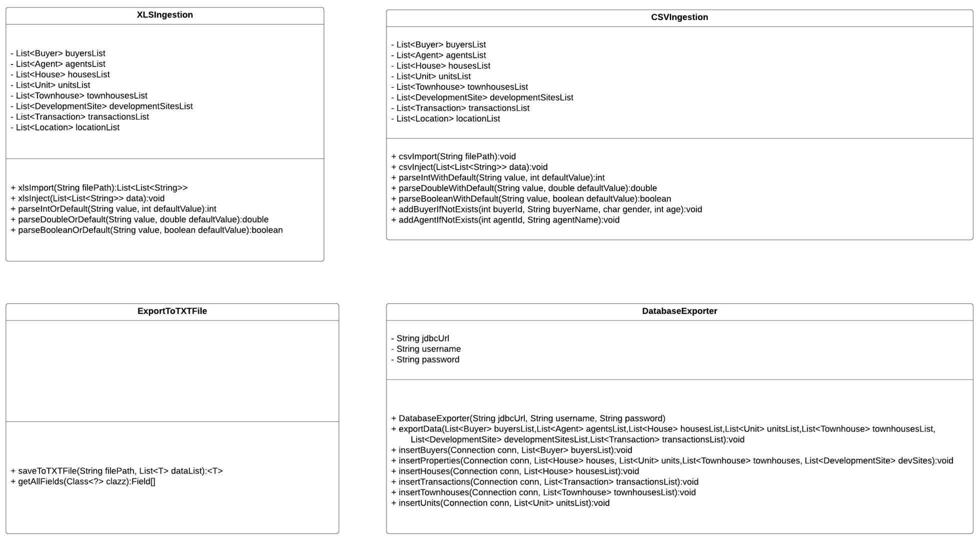

XLSIngestion
- List<Buyer> buyersList
- List<Agent> agentsList
- List<House> housesList
- List<Unit> unitsList
- List<Townhouse> townhousesList
- List<DevelopmentSite> developmentSitesList
- List<Transaction> transactionsList
- List<Location> locationList
+ xlsImport(String filePath):List<List<String>>
+ xlsInject(List<List<String>> data):void
+ parseIntOrDefault(String value, int defaultValue):int
+ parseDoubleOrDefault(String value, double defaultValue):double
+ parseBooleanOrDefault(String value, boolean defaultValue):boolean
CSVIngestion
- List<Buyer> buyersList
- List<Agent> agentsList
- List<House> housesList
- List<Unit> unitsList
- List<Townhouse> townhousesList
- List<DevelopmentSite> developmentSitesList
- List<Transaction> transactionsList
- List<Location> locationList
+ csvImport(String filePath):void
+ csvInject(List<List<String>> data):void
+ parseIntWithDefault(String value, int defaultValue):int
+ parseDoubleWithDefault(String value, double defaultValue):double
+ parseBooleanWithDefault(String value, boolean defaultValue):boolean
+ addBuyerIfNotExists(int buyerId, String buyerName, char gender, int age):void
+ addAgentIfNotExists(int agentId, String agentName):void
ExportToTXTFile
+ saveToTXTFile(String filePath, List<T> dataList):<T>
+ getAllFields(Class<?> clazz):Field[]
DatabaseExporter
- String jdbcUrl
- String username
- String password
+ DatabaseExporter(String jdbcUrl, String username, String password)
+ exportData(List<Buyer> buyersList,List<Agent> agentsList,List<House> housesList,List<Unit> unitsList,List<Townhouse> townhousesList, List<DevelopmentSite> developmentSitesList,List<Transaction> transactionsList):void
+ insertBuyers(Connection conn, List<Buyer> buyersList):void
+ insertProperties(Connection conn, List<House> houses, List<Unit> units,List<Townhouse> townhouses, List<DevelopmentSite> devSites):void
+ insertHouses(Connection conn, List<House> housesList):void
+ insertTransactions(Connection conn, List<Transaction> transactionsList):void
+ insertTownhouses(Connection conn, List<Townhouse> townhousesList):void
+ insertUnits(Connection conn, List<Unit> unitsList):void

Figure 5: Class Diagram: Data Ingestion and Handling Layer

Overview: The Data Ingestion and Handling Layer consists of:

- **XLSIngestion:** Handles importing and injecting data from XLS files into respective lists (e.g., buyers, agents, houses, etc.).

- **CSVIngestion:** Manages importing and injecting data from CSV files, with additional parsing methods to handle default values and avoid duplication.
- **ExportToTXTFile:** Provides functionality to save data to a text file and retrieve all fields of a class for serialization.
- **DatabaseExporter:** Handles database connections and exports data (buyers, properties, transactions, etc.) into a relational database using JDBC.

This layer ensures data is imported, parsed, and stored efficiently for further processing.

4.6 Data Exploration and Analysis Layer

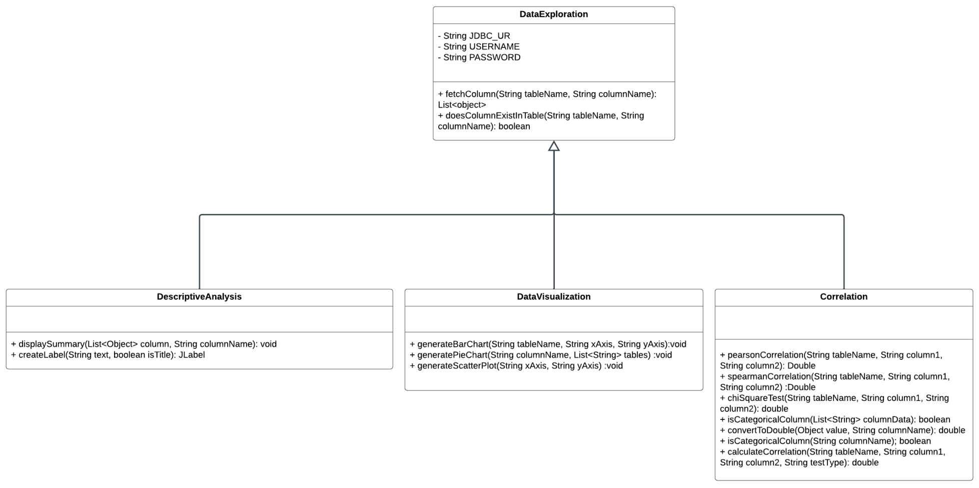

DataExploration
- String JDBC_UR
- String USERNAME
- String PASSWORD
+ fetchColumn(String tableName, String columnName): List<object>
+ doesColumnExistInTable(String tableName, String columnName): boolean
DescriptiveAnalysis
+ displaySummary(List<Object> column, String columnName): void
+ createLabel(String text, boolean isTitle): JLabel
DataVisualization
+ generateBarChart(String tableName, String xAxis, String yAxis):void
+ generatePieChart(String columnName, List<String> tables) :void
+ generateScatterPlot(String xAxis, String yAxis) :void
Correlation
+ pearsonCorrelation(String tableName, String column1, String column2): Double
+ spearmanCorrelation(String tableName, String column1, String column2) :Double
+ chiSquareTest(String tableName, String column1, String column2): double
+ isCategoricalColumn(List<String> columnData): boolean
+ convertToDouble(Object value, String columnName): double
+ isCategoricalColumn(String columnName); boolean
+ calculateCorrelation(String tableName, String column1, String column2, String testType): double

Figure 6: Class Diagram: Data Exploration and Analysis Layer

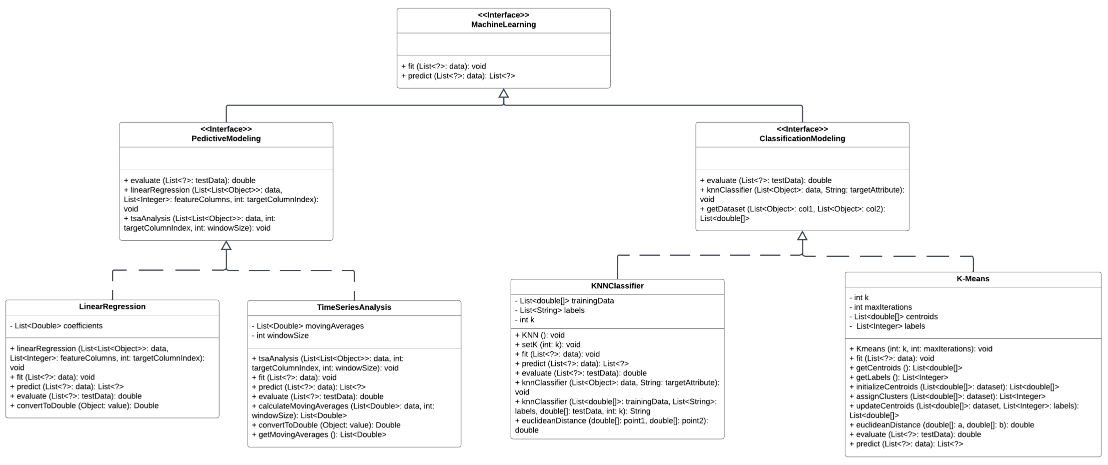
Class Diagram Explanation:

- **Parent Class: DataExploration**
  - Contains common attributes (`JDBC_URL`, `USERNAME`, `PASSWORD`) for database connection.
  - Provides shared methods:
    * `fetchColumn`: Retrieves data from a specific column.
    * `doesColumnExistInTable`: Checks if a column exists in a table.
- **Child Classes:**
  - **DescriptiveAnalysis:** Handles summary statistics and creates labels.
  - **DataVisualization:** Generates various charts (bar, pie, scatter).
  - **Correlation:** Performs statistical correlation tests (Pearson, Spearman, Chi-square).

OOP Concepts:

- **Inheritance:**
  - All child classes inherit `DataExploration` attributes and methods for database interaction, ensuring code reuse.
- **Abstraction:**
  - Common operations (`fetchColumn`) are abstracted in the parent class, hiding database details from child classes.
- **Polymorphism:**
  - Each child class overrides functionality to implement specific tasks (e.g., chart generation or correlation tests).
- **Encapsulation:**
  - Attributes like `JDBC_URL`, `USERNAME`, and `PASSWORD` are private, accessed only through parent class methods, ensuring secure database interaction.

4.7 Machine Learning Layer



<<Interface>>
MachineLearning
+ fit (List<?>: data): void
+ predict (List<?>: data): List<?>
<<Interface>>
PedictiveModeling
+ evaluate (List<?>: testData): double
+ linearRegression (List<List<Object>>: data, List<Integer>: featureColumns, int: targetColumnIndex): void
+ tsaAnalysis (List<List<Object>>: data, int: targetColumnIndex, int: windowSize): void
<<Interface>>
ClassificationModeling
+ evaluate (List<?>: testData): double
+ knnClassifier (List<Object>: data, String: targetAttribute): void
+ getDataset (List<Object>: col1, List<Object>: col2): List<double[]>
LinearRegression
- List<Double> coefficients
+ linearRegression (List<List<Object>>: data, List<Integer>: featureColumns, int: targetColumnIndex): void
+ fit (List<?>: data): void
+ predict (List<?>: data): List<?>
+ evaluate (List<?>: testData): double
+ convertToDouble (Object: value): Double
TimeSeriesAnalysis
- List<Double> movingAverages
- int windowSize
+ tsaAnalysis (List<List<Object>>: data, int: targetColumnIndex, int: windowSize): void
+ fit (List<?>: data): void
+ predict (List<?>: data): List<?>
+ evaluate (List<?>: testData): double
+ calculateMovingAverages (List<Double>: data, int: windowSize): List<Double>
+ convertToDouble (Object: value): Double
+ getMovingAverages (): List<Double>
KNNClassifier
- List<double[]> trainingData
- List<String> labels
- int k
+ KNN (): void
+ setK (int: k): void
+ fit (List<?>: data): void
+ predict (List<?>: data): List<?>
+ evaluate (List<?>: testData): double
+ knnClassifier (List<Object>: data, String: targetAttribute): void
+ knnClassifier (List<double[]>: trainingData, List<String>: labels, double[]: testData, int: k): String
+ euclideanDistance (double[]: point1, double[]: point2): double
K-Means
- int k
- int maxIterations
- List<double[]> centroids
- List<Integer> labels
+ Kmeans (int: k, int: maxIterations): void
+ fit (List<?>: data): void
+ getCentroids (): List<double[]>
+ getLabels (): List<Integer>
+ initializeCentroids (List<double[]>: dataset): List<double[]>
+ assignClusters (List<double[]>: dataset): List<Integer>
+ updateCentroids (List<double[]>: dataset, List<Integer>: labels): List<double[]>
+ euclideanDistance (double[]: a, double[]: b): double
+ evaluate (List<?>: testData): double
+ predict (List<?>: data): List<?>

Figure 7: Class Diagram: Machine Learning Layer

Class Diagram Explanation:

- **Parent Interface: MachineLearning**
  - Defines common methods for all machine learning models:
    - `fit`: Train the model.
    - `predict`: Make predictions on data.

- **Child Interfaces:**
  - **PredictiveModeling:** Adds methods for regression and time-series analysis (`linearRegression`, `tsaAnalysis`).
  - **ClassificationModeling:** Adds classification-specific methods (`knnClassifier`, `getDataset`).
- **Implementing Classes:**
  - **LinearRegression:** Implements predictive modeling for linear regression.
  - **TimeSeriesAnalysis:** Handles time-series analysis.
  - **KNNClassifier:** Implements classification using k-Nearest Neighbors.
  - **K-Means:** Performs clustering using the K-Means algorithm.

OOP Concepts:

- **Inheritance:**
  - Child interfaces inherit `fit` and `predict` methods from `MachineLearning`, ensuring consistency across implementations.
- **Abstraction:**
  - Common operations like training and prediction are abstracted in the interfaces, leaving specifics to implementations.
- **Polymorphism:**
  - Methods like `evaluate` and `predict` are overridden in each class for their specific algorithms.
- **Encapsulation:**
  - Internal data structures (e.g., `coefficients`, `centroids`, `labels`) are private and accessed via class methods.

5 Models Output Interpretation

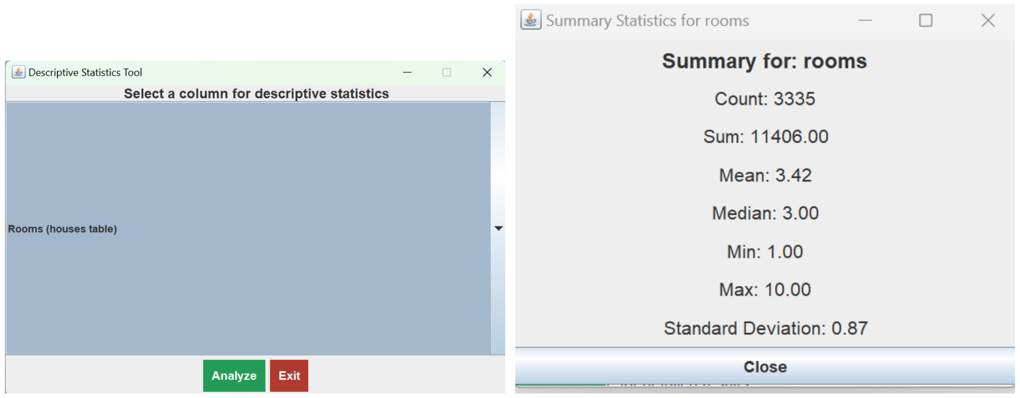
5.1 Data Exploration

5.1.1 Descriptive Statistics

The `DescriptiveAnalysis` class extends `DataExploration` to provide summary statistics for numerical data. It calculates key statistical metrics such as mean, median, minimum, maximum, sum, standard deviation, and count. The output is displayed in a graphical user interface (GUI) using Java Swing.

Functionality

- Converts a list of objects into numeric values for analysis.
- Computes summary statistics using Java's `DoubleSummaryStatistics`.
- Displays results in a structured GUI window.
- Provides an interactive interface with a close button for user convenience.


Descriptive Statistics Tool
Select a column for descriptive statistics
Rooms (houses table)
Analyze
Exit
Summary Statistics for rooms
Close

```
Summary for: rooms
Count: 3335
Sum: 11406.00
Mean: 3.42
Median: 3.00
Min: 1.00
Max: 10.00
Standard Deviation: 0.87
```

Figure 8: Summary statistics output in GUI.

5.1.2 Data Visualization

The `DataVisualization` class is responsible for generating and displaying graphical representations of data using Python's Matplotlib library. It provides interactive visualizations to help analyze numerical datasets effectively.

Functionality

- Reads and processes numerical data from a given source.
- Generates various types of plots, including bar charts, scatter plots, and pie charts.
- Uses Matplotlib for visualization with customizable labels and titles.
- Displays the generated plots in an interactive GUI window.

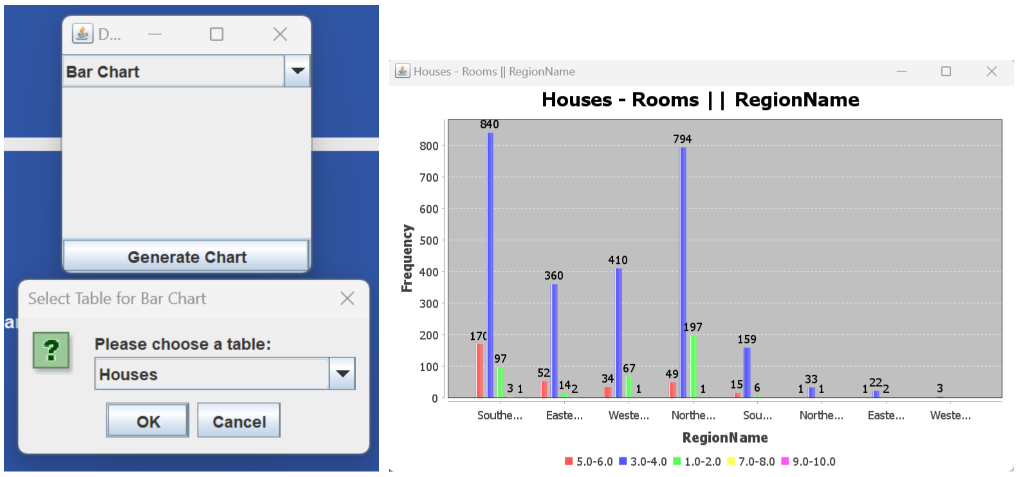


Figure 9: Sample visualizations generated by the DataVisualization module.

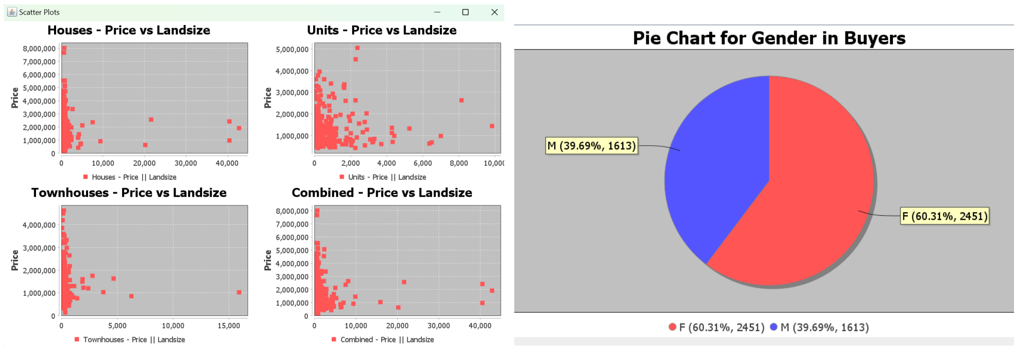


Figure 10: Additional plots showcasing dataset trends.

- Reads and processes numerical data from a given source.
- Generates various types of plots, including bar charts, scatter plots, and pie charts.
- Uses Matplotlib for visualization with customizable labels and titles.
- Displays the generated plots in an interactive GUI window.

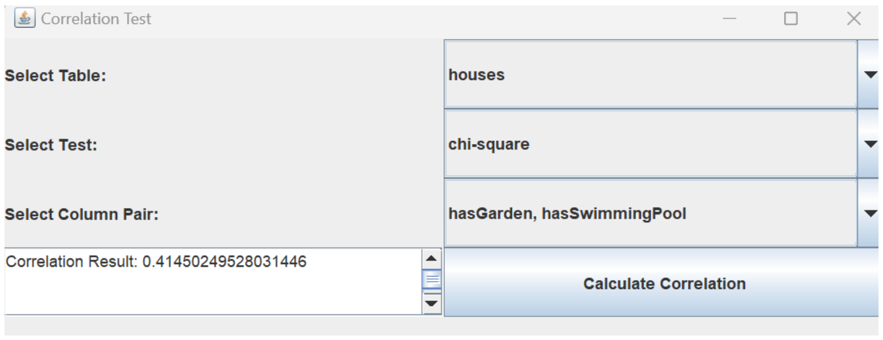
5.1.3 Correlation Analysis

The `Correlation` class extends `DataExploration` to compute statistical correlations between dataset columns. It supports Pearson, Spearman, and Chi-Square tests to analyze numeric and categorical relationships.

Functionality

- Extracts column data from a database table for correlation analysis.
- Computes Pearson correlation for numeric data to measure linear relationships.
- Computes Spearman correlation to assess rank-based dependencies.

- Performs Chi-Square tests to evaluate associations between categorical variables.
- Dynamically selects the appropriate correlation test based on column data types.
- Converts categorical and numeric values into a suitable format for calculations.



```
Correlation Result: 0.41450249528031446
```

Figure 11: Correlation analysis output in GUI.

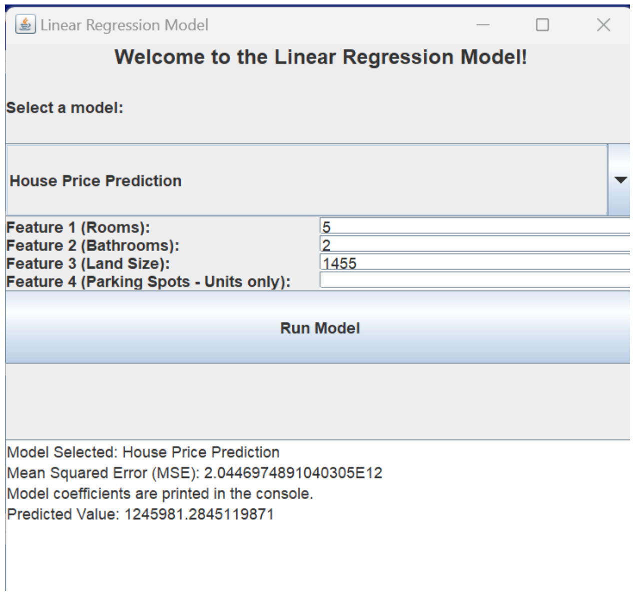
5.2 Machine Learning

5.2.1 Predictive Modeling

Linear Regression The `LinearRegression` class implements the `PredictiveModeling` interface to provide a predictive model based on linear regression. It fits a model using a set of feature variables and predicts target values.

Functionality

- Extracts numerical feature and target values from the dataset.
- Computes the slope and intercept using the least squares method.
- Fits the model to training data and stores computed coefficients.
- Predicts new target values based on the learned model.
- Evaluates model performance using Mean Squared Error (MSE).



Feature 1 (Rooms): 5
Feature 2 (Bathrooms): 2
Feature 3 (Land Size): 1455
Feature 4 (Parking Spots - Units only):

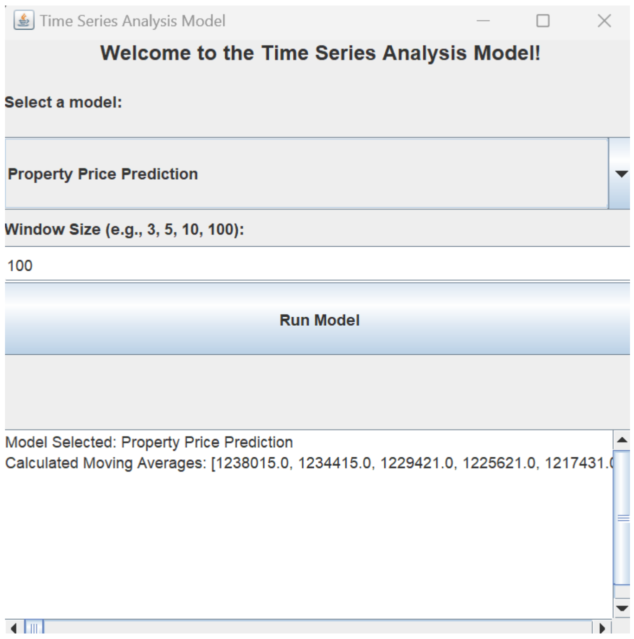
```
Model Selected: House Price Prediction
Mean Squared Error (MSE): 2.0446974891040305E12
Model coefficients are printed in the console.
Predicted Value: 1245981.2845119871
```

Figure 12: Linear regression model output in GUI.

Time Series Analysis The `TimeSeriesAnalysis` class implements the `PredictiveModeling` interface to analyze temporal data trends using moving averages. It extracts target values from time-dependent datasets and applies smoothing techniques.

Functionality

- Extracts target values from the dataset for time series analysis.
- Computes moving averages using a defined window size.
- Fits the model by processing sequential data points.
- Predicts future values based on the last computed moving average.
- Provides a structured approach for analyzing time-dependent data trends.


Time Series Analysis Model
Welcome to the Time Series Analysis Model!
Select a model:
Property Price Prediction
Window Size (e.g., 3, 5, 10, 100):
100
Run Model

```
Model Selected: Property Price Prediction
Calculated Moving Averages: [1238015.0, 1234415.0, 1229421.0, 1225621.0, 1217431.(
```

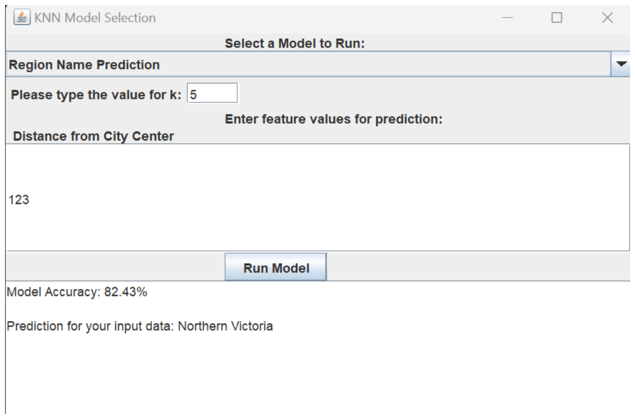
Figure 13: Time series analysis output in GUI.

5.2.2 Classification Modeling

K-Nearest Neighbors (KNN) Class The `KNN` class implements the `ClassificationModeling` interface to classify data points based on the K-Nearest Neighbors algorithm. It uses labeled training data to predict the class of new data points by considering the majority class of the nearest neighbors.

Functionality

- Initializes the classifier with a default value of `k = 3`.
- Allows dynamic adjustment of the number of neighbors (`k`) using the `setK` method.
- Trains the model by storing features and labels from the training dataset through the `fit` method.
- Predicts labels for new data points based on their proximity to training data points using the `predict` method.
- Evaluates the model's accuracy by comparing predicted labels with true labels from the test dataset using the `evaluate` method.
- Implements the `knnClassifier` method, which calculates the Euclidean distance between points to identify the nearest neighbors and return the most frequent class.



Please type the value for k: 5
Enter feature values for prediction:
Distance from City Center
123

Model Accuracy: 82.43%

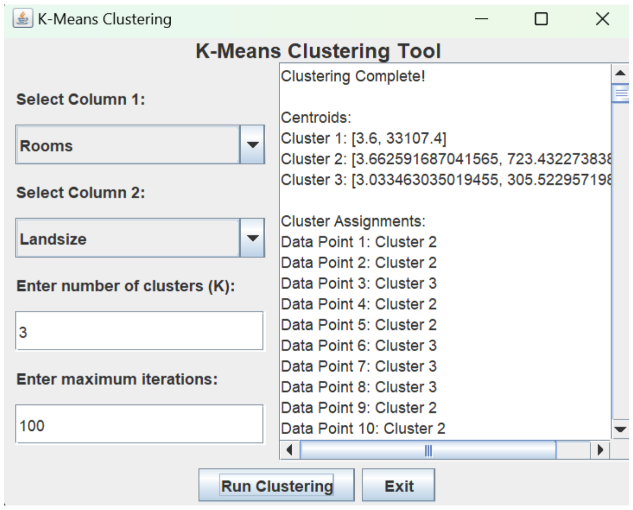
Prediction for your input data: Northern Victoria

Figure 14: KNN classification model overview.

K-Means Clustering Class The `Kmeans` class implements the `ClassificationModeling` interface to perform clustering using the K-Means algorithm. It groups data points into k clusters by iteratively adjusting centroids based on the assigned points.

Functionality

- Initializes the model with k clusters and a maximum number of iterations.
- The `fit` method iterates over the dataset, adjusting centroids until they stabilize or the maximum iterations are reached.
- The `initializeCentroids` method initializes centroids by selecting points from the dataset.
- The `assignClusters` method assigns each data point to the closest centroid.
- The `updateCentroids` method recalculates centroids based on the mean of points assigned to each cluster.
- Provides methods to retrieve the final centroids and cluster labels after fitting.
- Does not implement prediction or evaluation methods as K-Means is an unsupervised learning algorithm.



Select Column 1: Rooms

Select Column 2: Landsize

Enter number of clusters (K): 3

Enter maximum iterations: 100

```
Clustering Complete!

Centroids:
Cluster 1: [3.6, 33107.4]
Cluster 2: [3.662591687041565, 723.432273838
Cluster 3: [3.033463035019455, 305.522957198

Cluster Assignments:
Data Point 1: Cluster 2
Data Point 2: Cluster 2
Data Point 3: Cluster 3
Data Point 4: Cluster 2
Data Point 5: Cluster 2
Data Point 6: Cluster 3
Data Point 7: Cluster 3
Data Point 8: Cluster 3
Data Point 9: Cluster 2
Data Point 10: Cluster 2
```

Figure 15: K-Means clustering model output.

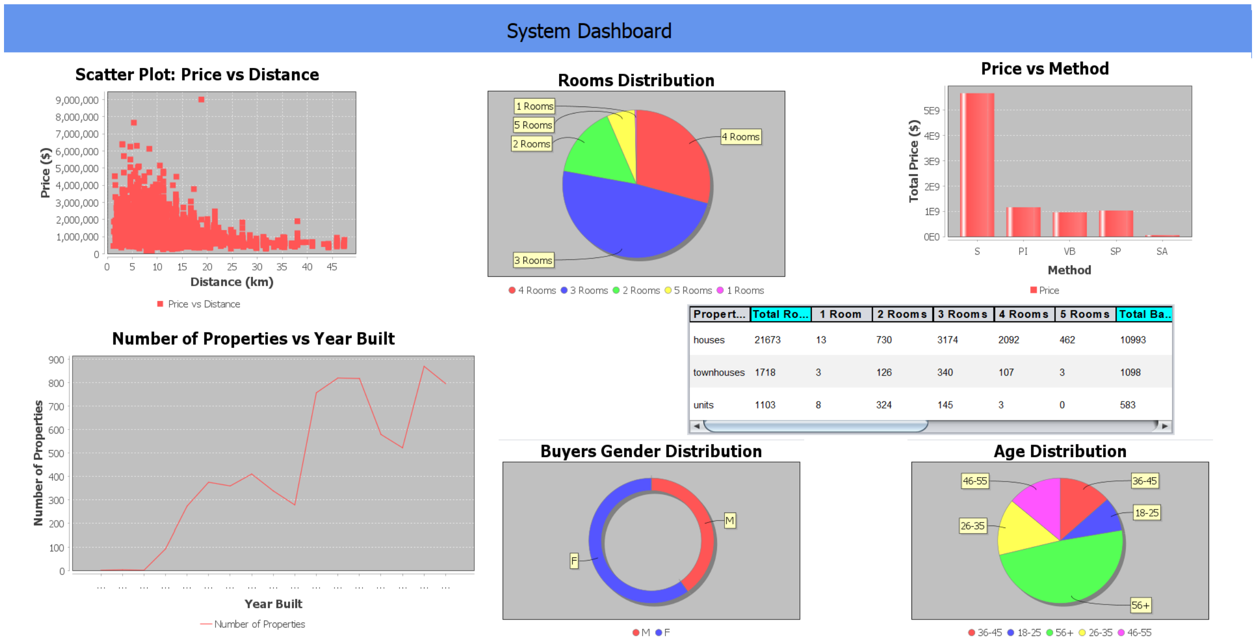
5.3 Dashboard

The dashboard provides a comprehensive visualization of property transaction data, enabling a detailed analysis of key metrics. Below are the interpretations of the different visualizations included:

- **Scatter Plot (Price vs Distance)**: This visualization highlights the relationship between property prices and their distances from key locations. A potential trend may indicate that properties closer to urban centers tend to be more expensive.
- **Pie Chart (Rooms Distribution)**: This chart categorizes properties based on the number of rooms, offering insights into the most common property configurations available in the market.
- **Bar Chart (Price vs Method)**: This visualization illustrates the total price distribution across different transaction methods, helping to identify which methods are more commonly used in high-value transactions.
- **Line Chart (Year Built vs Property Count)**: This chart displays the number of properties constructed over time, revealing trends in property development and the concentration of older or newer buildings.
- **Table Summary of Property Features**: The tabular data presents a breakdown of property types based on key attributes such as rooms, bathrooms, building sizes, and parking spots, allowing for quick comparisons.
- **Donut Chart (Buyers' Gender Distribution)**: This visualization presents the proportion of property buyers based on gender, providing demographic insights into the buyer population.

- **Pie Chart (Buyers' Age Distribution)**: The chart categorizes buyers into age groups, highlighting which demographic segments are most active in the property market.

The insights derived from these visualizations assist in understanding property market trends, buyer demographics, and the influence of various factors on property pricing.



Propert...	Total Ro...	1 Room	2 Rooms	3 Rooms	4 Rooms	5 Rooms	Total Ba..
houses	21673	13	730	3174	2092	462	10993
townhouses	1718	3	126	340	107	3	1098
units	1103	8	324	145	3	0	583

Figure 16: Dashboard Visualization

6 Conclusion

This project is a robust real estate management system designed to efficiently handle property data, transactions, and user interactions. It integrates advanced data processing capabilities, including property classification, data ingestion, and cleaning tools, ensuring high-quality and structured information. Additionally, the system incorporates machine learning algorithms such as linear regression, clustering, and KNN to enhance predictive analytics and classification tasks.

A key component of this system is the dashboard, which is managed independently to provide users with an intuitive and dynamic data visualization experience. Through interactive charts—including bar, line, scatter, and pie charts—the dashboard enables insightful analysis of market trends, buyer demographics, and transaction patterns. By decoupling the dashboard from the core system, the project ensures flexibility, scalability, and seamless data interpretation without interfering with backend operations.

7 Project Repository

The source code for this project is available on GitHub:
https://github.com/AsmaBoubaker22/IT320Project