Shapes

Contents

Description

8. Choose a point on the plane, and fill a collection with several regular shapes (circle, regular
triangle, square, regular hexagon). Which shapes lies the closest to the point? In case of the circle, the distance is measured from the line of the circle, if the point lies outside
of the circle. If a point lies inside of a circle, their distance is considered as zero.
Each shape can be represented by its center and side length (or radius), if we assume that one
side of the polygons are parallel with x axis, and its nodes lies on or above this side.
Load and create the shapes from a text file. The first line of the file contains the number of the
shapes, and each following line contain a shape. The first character will identify the type of the

shape, which is followed by the center coordinate and the side length or radius. Manage the shapes uniformly, so derive them from the same super class.

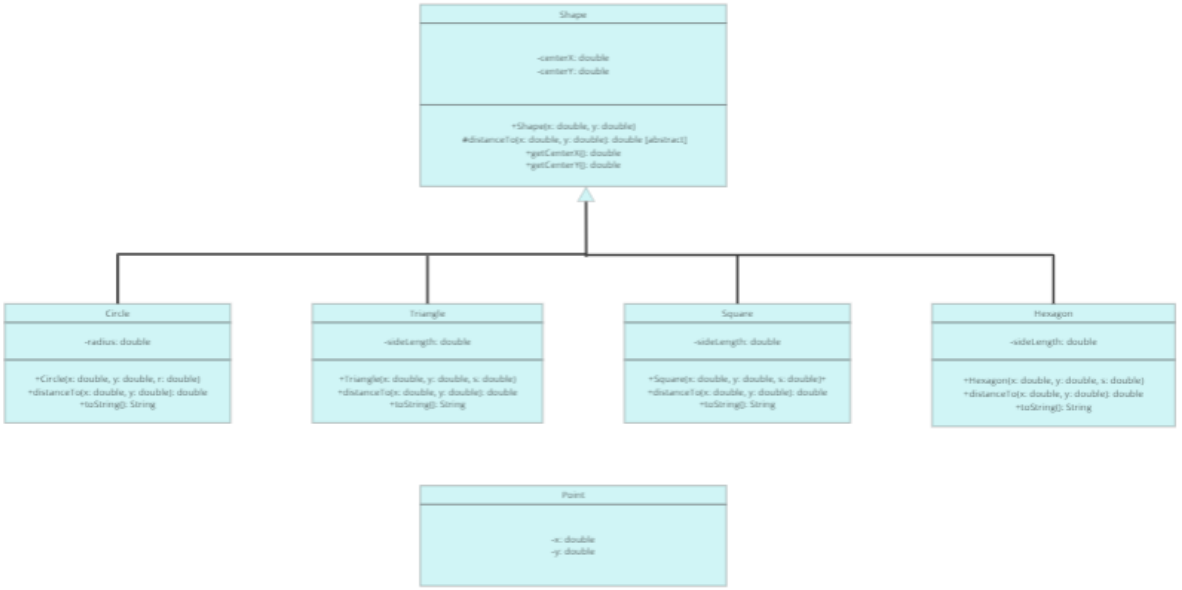
Input File Structure

Files should adhere to the following structure:

- The first line contains an integer, representing the number of shapes in the file.
- Each subsequent line should represent a shape. It starts with a character identifying the shape type (C for circle, T for triangle, etc.), followed by the X and Y coordinates of the center, and then the side length or radius. Data should be space-separated.
- Example:

```
3
C 5.0 5.0 2.5
T 3.0 3.0 4.0
S 7.0 7.0 3.0
```

Class Diagram



Description of each method

Directory Structure:

```
Shapes/
│
├── src/                                # Source code directory
│   │
│   ├── shapes/                         # Contains different sh
│   │   ├── Shape.java                  # Abstract base class f
│   │   ├── Circle.java                 # Class for circle shap
│   │   ├── Triangle.java               # Class for triangle sh
│   │   ├── Square.java                 # Class for square shap
│   │   └── Hexagon.java                # Class for hexagon sha
│   │
│   ├── handlers/                       # Contains utility hand
│   │   ├── ShapeLoader.java            # Class to load shapes
│   │   ├── ShapeHandler.java           # Class to calculate &
│   │   └── UserInputHandler.java       # (If used) Class to ha
│   │
│   ├── exceptions/                     # Contains custom excep
│   │   ├── InsufficientShapeParametersException.java    # Ex
│   │   ├── NegativeOrZeroValueException.java            # Ex
│   │   ├── UnknownShapeTypeException.java               # Ex
│   │   └── InvalidShapeFormatException.java             # Ex
│   │
│   └── app/                            # Contains the main app
│       └── Main.java                   # Entry point for the a
│
├── testfiles/                          # Directory for test in
│   ├── normal.txt                      # File for normal proce
│   ├── edge_cases.txt                  # File for edge case te
│   ├── extreme_values.txt              # File for testing extr
│   ├── missing_data.txt                # File for testing miss
│   └── invalid_type.txt                # File for testing inva
│
```

```
└── documentation/                    # (Optional) Directory
    └── Documentation.pdf             # The documentation fil
```

App

Main.java

- **Method** `public static void main(String[] args)`:
  - **Purpose**: The entry point of the application. It reads a list of specified test files, loads shapes from each of those files, prompts the user for a point, calculates the distances of all the loaded shapes from the provided point, and then displays the closest shape(s).
  - **Process**:
    - Instantiates a `Scanner` object to capture user input.
    - Iterates over a predefined list of test files.
    - For each file:
      - Initializes a new `ShapeLoader`.
      - Tries to load shapes from the current file.
      - Captures a point (X and Y coordinates) from the user.
      - Initializes a `ShapeHandler` with the loaded shapes and calculates distances from the point to all the shapes.
      - Fetches and displays the closest shape(s) to the user-provided point.
      - If any errors are encountered during file loading or shape processing, they're printed to the console.
    - Finally, after processing each file, a separator line is printed.
- **Inner Components**:
  - `ShapeHandler`: Handles shapes and provides functionalities like calculating distances of shapes from a given point and identifying the closest shape(s).
  - `ShapeLoader`: Responsible for loading shapes from a specified file and converting file data to a collection of `Shape` objects.

- `Scanner`: Helps in capturing user input from the console.
- `testFiles` **Array**: Contains names of the test files to be processed.

Handlers

ShapeHandler.java

1. **Constructor** `ShapeHandler(Collection<Shape> shapes)`:
   - **Purpose**: Initializes the ShapeHandler object with a given collection of shapes, setting up an empty map to store the distances from a reference point to each shape.
   - **Parameters**:
     - `shapes`: A collection containing shapes.
   - **Returns**: A new ShapeHandler instance with the provided shapes and an initialized distance map.
2. **Method** `calculateDistances(double pointX, double pointY)`:
   - **Purpose**: Calculates the distance from each shape in the collection to the specified point and stores these distances in the `distanceMap`.
   - **Parameters**:
     - `pointX`: X-coordinate of the reference point.
     - `pointY`: Y-coordinate of the reference point.
   - **Returns**: None. However, distances are calculated and stored in the internal `distanceMap`.
3. **Method** `findClosestShapes()`:
   - **Purpose**: Identifies the shape(s) with the shortest distance to the reference point.
   - **Parameters**: None.
   - **Returns**: A list containing the closest shape(s) to the reference point. If multiple shapes have the same shortest distance, they are all included in the returned list.

ShapeLoader.java

1. **Constructor** `ShapeLoader()`:

- **Purpose**: Initializes the ShapeLoader object, setting up an empty collection to store shapes.
- **Parameters**: None.
- **Returns**: A new ShapeLoader instance.

2. **Method** `loadShapesFromFile(String filename)`:
   - **Purpose**: Loads shapes from a specified file into the collection.
   - **Parameters**:
     - `filename`: The path to the file containing the shape data.
   - **Returns**: None. However, shapes are loaded into the internal `shapes` collection.
   - **Throws**: IOException if there's an issue reading the file, if the file is empty, or if there's missing shape data.
3. **Method** `createShapeFromLine(String line)`:
   - **Purpose**: Parses a line from the input file to create and add a shape object to the collection.
   - **Parameters**:
     - `line`: A string representing shape data.
   - **Returns**: None. A new shape is added to the internal `shapes` collection based on the parsed data.
   - **Throws**: IOException if the shape data is invalid or if an unknown shape type is encountered.
4. **Method** `getShapes()`:
   - **Purpose**: Provides access to the collection of shapes.
   - **Parameters**: None.
   - **Returns**: A collection of shapes that have been loaded.

Shapes

Circle.java

1. **Constructor** `Circle(double centerX, double centerY, double radius)`:

- **Purpose**: Initializes a Circle instance with specified center coordinates and radius.
- **Parameters**:
  - `centerX`: X-coordinate of circle's center.
  - `centerY`: Y-coordinate of circle's center.
  - `radius`: Radius of the circle.
- **Returns**: A Circle instance.

2. **Method** `distanceTo(double pointX, double pointY)`:
   - **Purpose**: Calculates the shortest distance from a given point to the boundary of the circle.
   - **Parameters**:
     - `pointX`: X-coordinate of the reference point.
     - `pointY`: Y-coordinate of the reference point.
   - **Returns**: The shortest distance to the circle.

Hexagon.java

1. **Constructor** `Hexagon(double centerX, double centerY, double sideLength)`:
   - **Purpose**: Initializes a Hexagon instance with specified center coordinates and side length.
   - **Parameters**:
     - `centerX`: X-coordinate of hexagon's center.
     - `centerY`: Y-coordinate of hexagon's center.
     - `sideLength`: Side length of the hexagon.
   - **Returns**: A Hexagon instance.
2. **Method** `distanceTo(double pointX, double pointY)`:
   - **Purpose**: Calculates the shortest distance from a given point to the boundary of the hexagon.
   - **Parameters**:
     - `pointX`: X-coordinate of the reference point.

    - `pointY`: Y-coordinate of the reference point.
- **Returns**: The shortest distance to the hexagon.

Shape.java

1. **Constructor** `Shape(double centerX, double centerY)`:
    - **Purpose**: Initializes a generic Shape instance with specified center coordinates.
    - **Parameters**:
        - `centerX`: X-coordinate of shape's center.
        - `centerY`: Y-coordinate of shape's center.
    - **Returns**: A Shape instance.
2. **Method** `distanceTo(double pointX, double pointY)` (Abstract):
    - **Purpose**: To be overridden in subclasses to calculate the shortest distance from a given point to the shape.

Square.java

1. **Constructor** `Square(double centerX, double centerY, double sideLength)`:
    - **Purpose**: Initializes a Square instance with specified center coordinates and side length.
    - **Parameters**:
        - `centerX`: X-coordinate of square's center.
        - `centerY`: Y-coordinate of square's center.
        - `sideLength`: Side length of the square.
    - **Returns**: A Square instance.
2. **Method** `distanceTo(double pointX, double pointY)`:
    - **Purpose**: Calculates the shortest distance from a given point to the boundary of the square.
    - **Parameters**:
        - `pointX`: X-coordinate of the reference point.
        - `pointY`: Y-coordinate of the reference point.

- **Returns**: The shortest distance to the square.

Triangle.java

1. **Constructor** `Triangle(double centerX, double centerY, double sideLength)`:
   - **Purpose**: Initializes a Triangle instance with specified center coordinates and side length.
   - **Parameters**:
     - `centerX`: X-coordinate of triangle's center.
     - `centerY`: Y-coordinate of triangle's center.
     - `sideLength`: Side length of the triangle.
   - **Returns**: A Triangle instance.
2. **Method** `distanceTo(double pointX, double pointY)`:
   - **Purpose**: Calculates the shortest distance from a given point to the base of the triangle.
   - **Parameters**:
     - `pointX`: X-coordinate of the reference point.
     - `pointY`: Y-coordinate of the reference point.
   - **Returns**: The shortest distance to the triangle.

Testing

1. **Normal Processing:** `normal.txt`
   - **Purpose**: To ensure that the program correctly processes valid and standard input.
   - **Expected Result**: The program successfully loads each shape and calculates the closest shape(s) to the specified point without any errors.
2. **White Box Test Cases**:

   a. **Testing Edge Cases in Input Parsing:** `edge_cases.txt`
   - **Purpose**: To test the program's ability to handle irregular input formatting, specifically checking its resilience against formatting errors.

- **Example**: A line in `edge_cases.txt` might look like "C 5.0,5.0 2.5", where a comma disrupts the regular space-separated format.
- **Expected Result**: The program should throw an error indicating invalid shape data due to the comma in the circle's coordinates.

b. **Testing with Extreme Values:** `extreme_values.txt`

- **Purpose**: To ensure the program handles very large or very small numerical values without crashing or giving wrong results.
- **Expected Result**: The program should successfully load the shapes and calculate the distance to the specified point. Extreme values shouldn't cause arithmetic overflows or other issues.

3. **Black Box Test Cases**:

a. **Testing With Missing Data:** `missing_data.txt`

- **Purpose**: To test the program's response when not all required data for a shape is provided.
- **Example**: A line in `missing_data.txt` might look like "C 5.0 5.0", missing the radius value.
- **Expected Result**: The program should throw an error indicating insufficient parameters for shape data.

b. **Testing With Invalid Shape Type:** `invalid_type.txt`

- **Purpose**: To see how the program responds to an unrecognized shape type.
- **Example**: A line in `invalid_type.txt` might look like "Z 5.0 5.0 3.0" or "X 3.0 3.0 4.0", where 'Z' and 'X' are unrecognized shape identifiers.
- **Expected Result**: The program should throw an error indicating an unknown shape type for the shape represented by the letter 'Z' and 'X'.