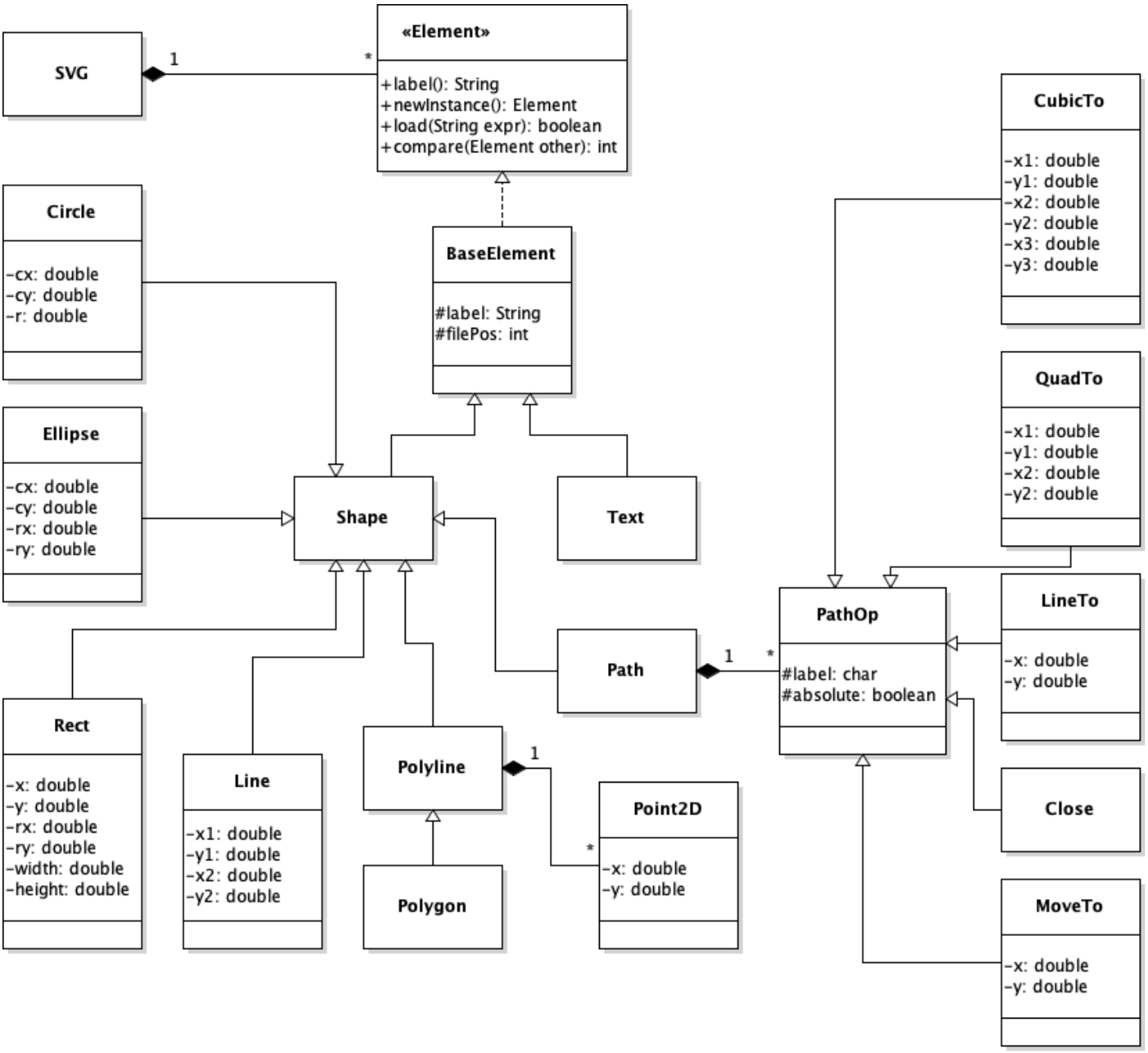
Maastricht University Department of Data Science and Knowledge Engineering

KEN1520 Software Engineering
Assignment 4 - Implementing a Class Structure

Instructor: Tom Pepels

Summary: This lab involves implementing a class structure from a UML diagram and related specification, applying design patterns where relevant.

The following UML class diagram shows one way to represent shape elements in the Standard Vector Graphics (SVG) file format:



Task

In this lab, you will complete a Java application to extract shape elements from an SVG file. The skeleton code provided loads an example SVG file to a String, and includes:

1. SVGLoader: The main app that loads the test file test-1.svg.
2. SVG: The SVG object that will hold the SVG file data that you will extract.
3. SVGParser: A class for parsing SVG files. You must complete the parse() method.

Some support classes are also provided to get you going. Element is the interface common to all SVG elements (including Shapes). BaseElement is the abstract implementation of this interface; it defines member variables including the element label and the element's position in the file, which is used for sorting elements.

After parsing the test file provided, your program should output the fields of each element, in the order they occur in the file. For example, after parsing test-1.svg, your program should output something similar to the following:

```
src/test-1.svg has 8 elements:
rect: x=10.0, y=10.0, rx=0.0, ry=0.0, width=30.0, height=30.0
rect: x=60.0, y=10.0, rx=10.0, ry=10.0, width=30.0, height=30.0
circle: cx=25.0, cy=75.0, r=20.0
ellipse: cx=75.0, cy=75.0, rx=20.0, ry=5.0
line: x1=10.0, y1=110.0, x2=50.0, y2=150.0
polyline: (60.0,110.0) (65.0,120.0) (70.0,115.0) (75.0,130.0)
(80.0,125.0) (85.0,140.0) (90.0,135.0) (95.0,150.0) (100.0,145.0)
polygon: (50.0,160.0) (55.0,180.0) (70.0,180.0) (60.0,190.0)
(65.0,205.0) (50.0,195.0) (35.0,205.0) (40.0,190.0) (30.0,180.0)
(45.0,180.0)
path: [M: x=20.0, y=260.0]* [Q: x1=40.0, y1=205.0, x2=80.0, y2=260.0]*
[Z]*
```

Steps

1. Split into groups of eight. Anyone left over, just join a group.

2. Assign roles within your group. After an initial discussion to discuss an overall plan, you might split into pairs or sub-groups for relevant sub-tasks (interpreting UML diagram, studying data format from SVG specification, implementing class structure and factories, implementing parsing algorithm, etc.).

If you delegate sub-tasks within your group and work on them concurrently, make sure that all group members keep in regular communication throughout, so that you are all working towards the same goal.

3. Look over the provided skeleton code and get familiar with the available functionality.

4. Define an abstract Shape class that extends BaseElement. This will be the parent class for your shape classes.

5. Define each concrete shape class (Circle, Rect, Path, etc.) with the relevant member variables. Each shape should know its own label (e.g. Circle should be labelled “circle”).

6. Override the load(String expr) method for each concrete shape class. The expr parameter should be a string expression passed in by the parser that describes that element. You can extract named fields from the string as follows:

```
if (expr.contains(" cx="))
{
        final Double result = SVGParser.extractDouble(expr, " cx=");
        if (result != null)
                cx = result.doubleValue();
}
```

7. The concrete shape class Path requires additional handling as each path is composed of a number of path operations. You should define a hierarchy of PathOp sub-classes, as per the UML diagram.

8. Complete the implementation of SVGParser.parse(). The commented-out code searches the content of the SVG file for the occurrence of elements, and when each element is found a new instance is created, its data should is loaded, and it is added to the elements collection. Note that this code uses the Prototype and Factory Method design patterns.

The String.indexOf(str, fromIndex) method is useful for identifying substrings within a string, e.g. for locating element labels within the SVG file content.

Note: You should identify and apply relevant design patterns in your design. For example, you might use a Factory Method to create new instances of Elements as you locate them in the SVG content (as per the commented-out code). You might also use a separate Factory Method for PathOp objects as you locate them within path descriptions. These factories might use the Prototype pattern to store a list of pre-defined Shape and PathOp objects to use as templates. The Singleton pattern may apply if only one factory of each type is required, and so on.

Hint: Once you have decided on your class structure, try implementing a single test class first, e.g. Circle, to check that your mechanism is correct and working as expected. Then implement the other Shape types when you are satisfied with your design.

You should be able to extend your design to include additional shape types simply by defining a new class for each type derived from Shape (and adding a prototype in the relevant factory, if you take that route). If you need to modify SVGParser with the addition of each new shape, then that indicates something wrong with your design that violates the Open/Closed Principle.

Implementing your first shape may take some time, but implementing subsequent shapes should be much quicker, as you can reuse the same approach and much of the code.

Note: You do not need to implement the Text class (it is just there for completeness) or any of the painting style elements (stroke, fill, transform, etc.). Just the shapes.

Resources

Scalable Vector Graphics (SVG) is a plain text XML-based vector image format for 2D graphics, with an open specification (https://www.w3.org/TR/SVG11/).

The SVG Wikipedia page gives an overview of the SVG format:
https://en.wikipedia.org/wiki/Scalable_Vector_Graphics

Introduction to Scalable Vector Graphics gives more detail:
http://fivedots.coe.psu.ac.th/Software.coe/J2ME/SVG/x-svg-a4.pdf

Submission

Deadline for submission is **23:59 - 8 may 2020.**

Upload your work to the KEN1520 section on EleUM. Your submission should include:

1. Your source code as a single .zip file.
2. A PDF document containing your list of team members (names and student numbers) and any relevant notes, e.g. why you made certain design choices, which design patterns are applicable and why, etc.
3. Mail your group's solution **once** to tom.pepels@maastrichtuniversity.nl with subject:
   Group <n> assignment 3