GP 2	Generative Programming	ST 17, Exercise 3

Deadline: 05.05.2017, 13:50

☐ **Gr. 1,** J. Karder, MSc.
☐ **Gr. 2,** P. Fleck, MSc.

Name _____ **Effort in h** _____

Points _____ **Lecturer** _____

1. One, to generate them all (6 Points)

This time it is your task to implement a mechanism for generating source code for classes. This time, the specification of these classes is given in XML files; for example, the definition of the classes *Person* and *Student* could be given in the following way:

```
<?xml version="1.0" encoding="utf-8" ?>
<model>
  <class name="Person" namespace="GenerativeProgramming">
    <property name="Name" type="string"/>
    <property name="Age" type="int"/>
  </class>
  <class name="Student" namespace="GenerativeProgramming" base="Person">
    <property name="Id" type="string"/>
  </class>
</model>
```

Implement a **T4 template** for generating C# code using the given XML files. The definition of base classes should be optional; e.g., in this example the base class for the class *Student* is *Person*.

In addition to the defined properties the generated classes should have a *ToString()* function. For example, for an instance of the class *Student* the *ToString()* function should return:

```
Name: Ben
Age: 23
Id: 2045402
```

2. Clone 'Em All! (6 Points)

In many frameworks, for example also in HeuristicLab, cloning mechanisms are used for creating deep copies of arbitrary objects in an automated way. Your task is now to implement a **T4 template** which generates a function *IDeepCloneable Clone(Cloner cloner)* as well as a copy constructor for given classes.

Of course, fields of objects that are to be cloned are also cloned:

- Elements that implement the interface *IDeepCloneable* are also cloned using the *Cloner.Clone* function,
- Elements that have a *Clone()* function are copied using this function, and
- all other elements are simply assigned, as in this case no functionality for creating clones is available.

The here used interface *IDeepCloneable* is defined as follows:

```
namespace CloningGenerator {
  public interface IDeepCloneable : ICloneable {
    IDeepCloneable Clone(Cloner cloner);
  }
}
```

For example, let the classes *A* and *B* be defined as follows:

```
public partial class A : DeepCloneable {
  public int i;
  public B b;
  public A() { }
}
public partial class B : DeepCloneable {
  public string s;
  public int[] arr;
  public B() { }
}
```

For these classes the following functionality is generated using the T4 template that is to be implemented:

```
partial class A {
  protected A(A original, Cloner cloner)
    : base(original, cloner) {
    this.i = original.i;
    this.b = cloner.Clone(original.b);
  }
  public override IDeepCloneable Clone(Cloner cloner) {
    return new A(this, cloner);
  }
}
```

For finding all types that implement *IDeepCloneable* we analyze the assembly that is generated by the compiler.

3. SVG Generator (8 + 4 Points)

Scalable Vector Graphics (SVG) is a standard for vector graphics in XML syntax[1], defined by the W3C. All geometric primitives that are to be drawn are defined as tags in the XML document; using this XML document, a renderer is used for drawing the figure. For example, the following XML source text

```
<?xml version="1.0" encoding="UTF-8"?>
  <!DOCTYPE svg>
  <svg xmlns="http://www.w3.org/2000/svg" width="1000" height="600" viewBox="0 0 5 5">
    <rect id="black" fill="#000" width="5" height="3"/>
    <rect id="gray_i" fill="#444" width="5" height="2" y="1"/>
    <rect id="gray_ii" fill="#888" width="5" height="1" y="2"/>
    <rect id="gray_iii" fill="#ccc" width="5" height="1" y="3"/>
    <rect id="white" fill="#fff" width="5" height="1" y="4"/>
  </svg>
```

defines the following graphic:

Since all tags of geometric primitives have the same structure, template based approaches can be used for generating code for generating SVG graphics. Your task is to use *FreeMarker* for developing a framework for the generation of SVG graphics that consist of arbitrary shapes.

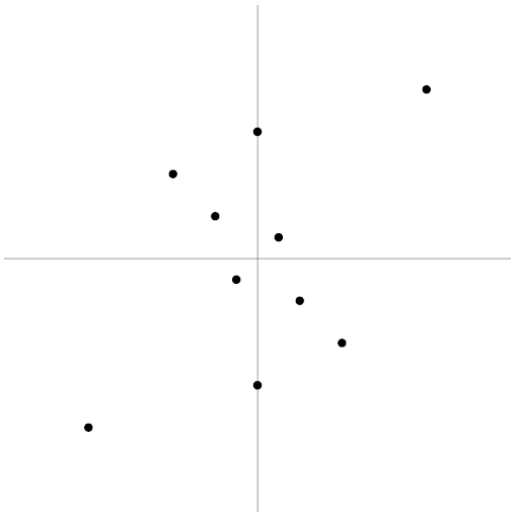
Implement a class *Diagram* that defines the following essential properties of a diagram:

- Width and height of the graphic in pixel (int) (= screen coordinates)
- Minimum and maximum x and y values of the coordinate system for defining the part of the figure that shall be displayed (= user coordinates)
- Standard size of a shape (double)
- Display of the axes (boolean)
- List of shapes that shall be displayed

Furthermore, your task is to implement a FreeMarker-template d*iagram.ftl* which is the basic frame of the SVG graphics that are to be generated. All data that is necessary for the generation of the diagram has to be passed to a class *DiagramGenerator* that uses the diagram template for code generation.

[1] Further information about the SVG 1.1 specification can be found at http://www.w3.org/TR/SVG11/

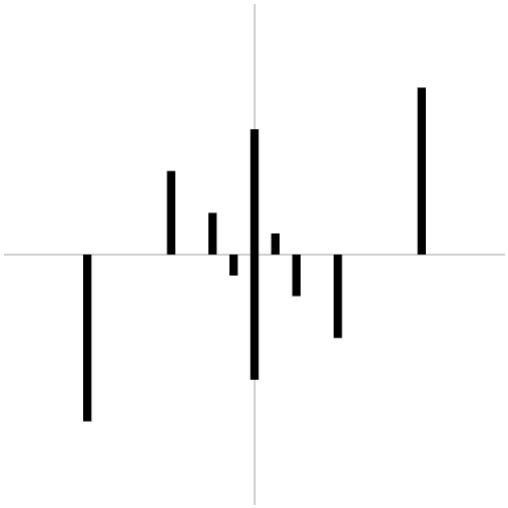
a) The first version of your generator should produce simple graphics representing sets of points. Additionally, the optional display of x and y axes shall be possible. The following example can be generated; the corresponding SVG code is given below:



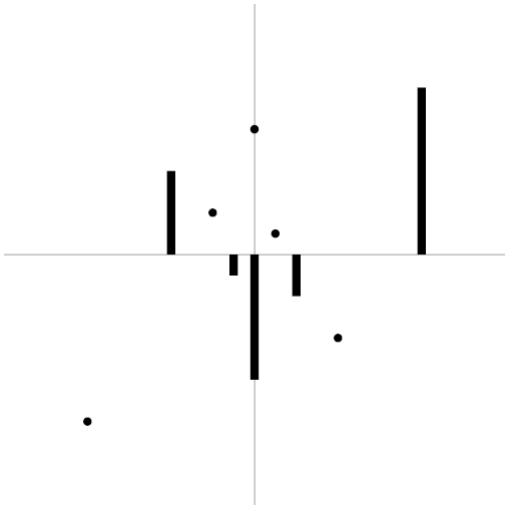
```
<?xml version="1.0" encoding="utf-8"?>
<svg xmlns=http://www.w3.org/2000/svg
     xmlns:xlink=http://www.w3.org/1999/xlink
     xmlns:ev=http://www.w3.org/2001/xml-events
     width="500"
     height="500"
     viewBox="-3.0 -3.0 6.0 6.0"
     preserveAspectRatio="none">

  <line x1="-3.0" y1="0" x2="3.0" y2="0" stroke="#555" stroke-width="0.01"/>
  <line x1="0" y1="-3.0" x2="0" y2="3.0" stroke="#555" stroke-width="0.01"/>

  <circle cx="-2.0" cy="2.0" r="0.05" fill="#000"/>
  <circle cx="-1.0" cy="-1.0" r="0.05" fill="#000"/>
  <circle cx="-0.5" cy="-0.5" r="0.05" fill="#000"/>
  <circle cx="-0.25" cy="0.25" r="0.05" fill="#000"/>
  <circle cx="0.0" cy="-1.5" r="0.05" fill="#000"/>
  <circle cx="0.0" cy="1.5" r="0.05" fill="#000"/>
  <circle cx="0.25" cy="-0.25" r="0.05" fill="#000"/>
  <circle cx="0.5" cy="0.5" r="0.05" fill="#000"/>
  <circle cx="1.0" cy="1.0" r="0.05" fill="#000"/>
  <circle cx="2.0" cy="-2.0" r="0.05" fill="#000"/>
</svg>
```

b) Improve your SVG generator by making it work in a modular way. For each shape a specific generator shall be implemented. One concrete variant of this generator (*PointGenerator*) has already been used implicitly in a); your task now is to implement at least two additional generators, e.g. a *RectangleGenerator* for generating bar charts or a *PolylineGenerator*. Please note: Implement the generators in analogy to the *DiagramGenerator*.

The result could look like the following example:



The combination of these generators within one graphic must also be possible:



Please note:
- Test your implementations at length (at least three test cases).
- Formulate reasonable documentations for your solutions.