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.

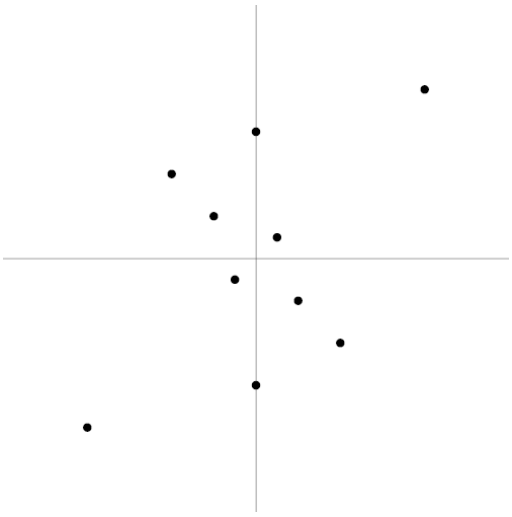
Implement a class *Diagram* that defines the following essential properties of a diagram:

- 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/

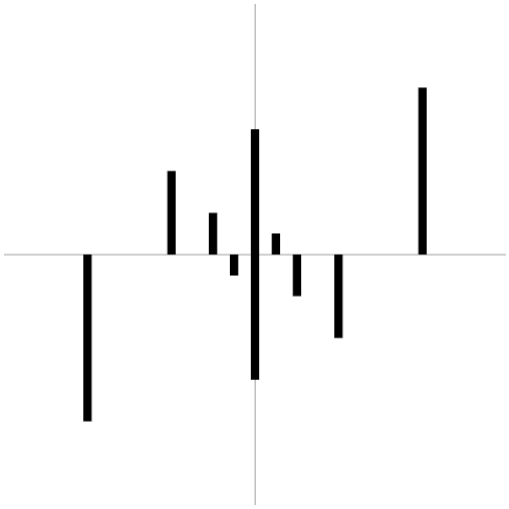
a) The first version of your generator should produce simple graphics representing sets of points. Additionally, the optional display of x and y axes shall be possible. The following example can be generated; the corresponding SVG code is given below:



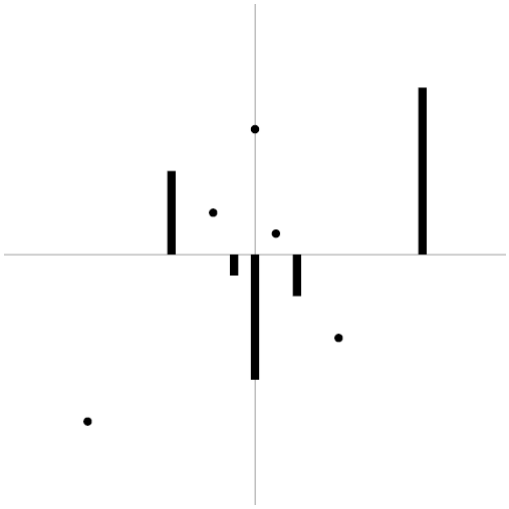
```
<?xml version="1.0" encoding="utf-8"?>
<svg xmlns=http://www.w3.org/2000/svg
     xmlns:xlink=http://www.w3.org/1999/xlink
     xmlns:ev=http://www.w3.org/2001/xml-events
     width="500"
     height="500"
     viewBox="-3.0 -3.0 6.0 6.0"
     preserveAspectRatio="none">

  <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.

1 One, to generate them all

1.1 Lösungsidee

Dieser Abschnitt beschäftigt sich mit der Dokumentation der Lösungsidee der Aufgabenstellung *One, to generate them all*. Um von den *XML-Tag*-Namen und dessen Attributen unabhängig zu sein sollen die *XML-Tag*-Namen und dessen Attribute in Konstanten ausgelagert werden und sollen bei der Navigation durch das *XML*-Dokument verwendet werden, um die *XML-Tags* sowie dessen Attribute zu adressieren. Die Unabhängigkeit von der Struktur des *XML*-Dokuments ist aber nicht möglich.

Da die im *XML*-Dokument definierten Klassen sich im selben Namensraum befinden können, sollen die definierten Klassen nach den Namensräumen gruppiert werden, damit die Namensräume nicht mehrfach in den generierten Klassen definiert werden müssen. Um das zu realisieren soll *Linq* verwendet werden. Um sich zu viele *If*-Blöcke im *T4-Template* zu vermeiden soll mit dem *?* Operator gearbeitet werden, um die resultierenden Zeichenketten des *C#*-Quelltextes zu generieren. Dadurch soll das *T4-Template* übersichtlicher werden, da meiner Meinung nach das Verwenden von zu vielen *If*-Anweisungen das *Template* unübersichtlich macht.

Zusätzlich zu den verlangten Attributen der *XML-Tags* wurde noch auf Klassenebene ein *Modifier* eingeführt, womit auch abstrakte Klassen möglich sind.

Die *Tostring* Methode wurde so implementiert, dass wenn es eine Basisklasse gibt, zuerst an diese Basisklasse delegiert wird und dann erst die Ausgabe der konkreten Klasse erfolgt, wobei am Anfang einer jeder *Tostring* Implementierung immer die der Klassennamen ausgegeben wird.

1.2 Quelltexte

Folgender Abschnitt enthält die Quelltexte der Aufgabenstellung.

Listing 1: ClassGenerator.tt

```
<#@ template debug="true" hostspecific="true" language="C#" #>
<#@ output extension=".cs" #>
<#@ assembly name="System.Xml" #>
<#@ assembly name="System.Core" #>
<#@ assembly name="System.Xml.Linq, Version=4.0.0.0, Culture=neutral,
 ↪  PublicKeyToken=b77a5c561934e089" #>
<#@ import namespace="System.Collections.Generic" #>
<#@ import namespace="System.Linq" #>
<#@ import namespace="System.Xml.Linq" #>

<#
    // Constants for attribute names of the XElements
  const string ELEMENT_CLASS = "class";
    const string ELEMENT_PROPERTY = "property";
    const string ATTR_CLASS_NAME = "name";
  const string ATTR_CLASS_MODIFIER = "modifier";
    const string ATTR_NAMESPACE = "namespace";
    const string ATTR_BASE_CLASS = "base";
    const string ATTR_FIELD_NAME = "name";
    const string ATTR_FIELD_TYPE = "type";

    // Loading elements
    var modelFile = Host.ResolvePath("Model.xml");
    var xml = XElement.Load(modelFile);
  // group by namespaces
  var namespaceClasses = xml.Elements(ELEMENT_CLASS)
```

```
                        .GroupBy(el => (el.Attribute(ATTR_NAMESPACE) != null ?
    el.Attribute(ATTR_NAMESPACE).Value : null),
                                        el => el);

  // Create classes for each namespace
    foreach (var group in namespaceClasses)
    {

#>
<#=(group.Key != null) ? ("namespace " + group.Key + " {") : ""#>
<#
        // Create classes of namespace
        foreach (var element in group)
        {
            // Get all all class level relevant attributes
            var className = element.Attribute(ATTR_CLASS_NAME)?.Value;
            var baseClass = element.Attribute(ATTR_BASE_CLASS)?.Value;
            var classModifier = element.Attribute(ATTR_CLASS_MODIFIER);
            var propertyList = new List<Tuple<string, string>>(element.Elements(ELEMENT_PROPERTY)
                                                                .Select(prop => new Tuple<string,
    string>(prop.Attribute(ATTR_FIELD_TYPE)?.Value, prop.Attribute(ATTR_FIELD_NAME)?.Value)));
#>
public <#=(classModifier != null ? classModifier.Value : "") #> class <#=className
      + (baseClass != null ? (": " + baseClass + "{") : "{") #>
<#
            // Create properties
            foreach (var propElement in propertyList)
            {
#>
public <#=propElement.Item1 + " " + propElement.Item2 #> { get; set; }
<#
            }

      // Create Constructor ToString if properties are present
            if (propertyList.Count > 0)
            {
#>
public override string ToString() {
<#
                // Create toString with all attributes
        for (var i = 0; i < propertyList.Count; i++)
        {
            var propItem = propertyList[i];
                    var last = i + 1 == propertyList.Count;
                    var first = i == 0;
#>
<#=(first ? "return " + ((baseClass != null) ? "base.ToString() + \"\\n\" + " : "")
                      + "\"Class: " + className + " \\n\" + "
          : " + ")
            + "\"" + propItem.Item2 + ": \" + " + propItem.Item2
         + (last ? ";" : " + \"\\n\"") #>
<#
                }
#>
}
<#
            }
#>
    }
<#
  }
#>
```

```
<#=(group.Key != null) ? "}": ""#>
<#
    }
#>
```

Listing 2: Model.xml

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

  <!-- Other classes of another namespace -->
  <class name="BaseShape" namespace="Shape">
    <property name="Name" type="string"/>
    <property name="Coordinate" type="Coordinate"/>
    <property name="Stroke" type="int"/>
    <property name="StrokeHexColor" type="string"/>
    <property name="FillColor" type="string"/>
  </class>
  <class name="Rectangular" namespace="Shape" base="BaseShape">
    <property name="Width" type="double"/>
    <property name="Height" type="double"/>
  </class>
  <class name="Circle" namespace="Shape" base="BaseShape">
    <property name="Radius" type="double"/>
  </class>

  <!-- Other classes in no namespace -->
  <class name="Coordinate">
    <property name="XCoordinate" type="double"/>
    <property name="YCoordinate" type="double"/>
  </class>
</model>
```

Listing 3: Program.cs

```
using CloningGenerator;
using System;
using Persons;
using Shape;

namespace ClassGenerator {
    class Program
    {
        static void Main(string[] args)
        {
            TestPersons();
```

```
12              TestShapes();
13
14              // Just to kepp console open, which it wouldn't otherwise
15              Console.Read();
16          }
17
18          public static void TestPersons()
19          {
20              Console.WriteLine("########################################################## \n"
21                              + "testPersons()\n"
22                              + "##########################################################");
23
24              Department department = new Department()
25              {
26                  Name = "Software engineering"
27              };
28
29              Teacher teacher = new Teacher()
30              {
31                  Age = 50,
32                  Department = department,
33                  Id = "P00000001",
34                  Name = "Dobler"
35              };
36
37              Student student = new Student() {
38                  Name = "Thomas",
39                  Id = "S1610454013"
40              };
41
42              Console.WriteLine(department + "\n");
43              Console.WriteLine(teacher + "\n");
44              Console.WriteLine(student + "\n");
45
46              Console.WriteLine("##########################################################\n");
47          }
48
49          public static void TestShapes()
50          {
51              Console.WriteLine("########################################################## \n"
52                              + "testShapes()\n"
53                              + "##########################################################");
54
55              Coordinate coordinate = new Coordinate();
56              coordinate.XCoordinate = 1.0;
57              coordinate.YCoordinate = 2.0;
58
59              Rectangular rectangular = new Rectangular();
60              rectangular.Height = 1.0;
61              rectangular.Width = 2.0;
62              rectangular.Name = "Rectangular";
63              rectangular.Coordinate = coordinate;
64              rectangular.FillColor = "#ffffff";
65              rectangular.StrokeHexColor = "#000000";
66              rectangular.Stroke = 1;
67
68              Circle circle = new Circle()
69              {
70                  Coordinate = coordinate,
71                  FillColor = "#ffffff",
72                  Name = "Circle",
73                  Radius = 2.0,
74                  Stroke = 1,
```

```
                StrokeHexColor = "#000000"
            };

            Console.WriteLine(coordinate + "\n");
            Console.WriteLine(rectangular + "\n");
            Console.WriteLine(circle + "\n");

            Console.WriteLine("############################################################\n");
        }
    }
}
```

1.3 Tests

Folgender Abschnitt enthält die Tests der Aufgabenstellung in Form der generierten Quelltexte sowie der Ausgaben des Testprogramms.

Listing 4: ClassGenerator.cs

```
namespace Persons
{
    public abstract class Person
    {
        public string Name { get; set; }
        public int Age { get; set; }

        public override string ToString()
        {
            return "Class: Person \n" + "Name: " + Name + "\n"
                   + "Age: " + Age;
        }
    }

    public class Student : Person
    {
        public string Id { get; set; }

        public override string ToString()
        {
            return base.ToString() + "\n" + "Class: Student \n" + "Id: " + Id;
        }
    }

    public class Teacher : Person
    {
        public string Id { get; set; }
        public Department Department { get; set; }

        public override string ToString()
        {
            return base.ToString() + "\n" + "Class: Teacher \n" + "Id: " + Id + "\n"
                   + "Department: " + Department;
        }
    }

    public class Department : Person
    {
        public string Name { get; set; }

        public override string ToString()
        {
```

```
43              return base.ToString() + "\n" + "Class: Department \n" + "Name: " + Name;
44          }
45      }
46  }
47
48  namespace Shape
49  {
50      public class BaseShape
51      {
52          public string Name { get; set; }
53          public Coordinate Coordinate { get; set; }
54          public int Stroke { get; set; }
55          public string StrokeHexColor { get; set; }
56          public string FillColor { get; set; }
57
58          public override string ToString()
59          {
60              return "Class: BaseShape \n" + "Name: " + Name + "\n"
61                     + "Coordinate: " + Coordinate + "\n"
62                     + "Stroke: " + Stroke + "\n"
63                     + "StrokeHexColor: " + StrokeHexColor + "\n"
64                     + "FillColor: " + FillColor;
65          }
66      }
67
68      public class Rectangular : BaseShape
69      {
70          public double Width { get; set; }
71          public double Height { get; set; }
72
73          public override string ToString()
74          {
75              return base.ToString() + "\n" + "Class: Rectangular \n" + "Width: " + Width + "\n"
76                     + "Height: " + Height;
77          }
78      }
79
80      public class Circle : BaseShape
81      {
82          public double Radius { get; set; }
83
84          public override string ToString()
85          {
86              return base.ToString() + "\n" + "Class: Circle \n" + "Radius: " + Radius;
87          }
88      }
89  }
90
91
92  public class Coordinate
93  {
94      public double XCoordinate { get; set; }
95      public double YCoordinate { get; set; }
96
97      public override string ToString()
98      {
99          return "Class: Coordinate \n" + "XCoordinate: " + XCoordinate + "\n"
100                + "YCoordinate: " + YCoordinate;
101     }
102 }
```

```
##########################################################
testPersons()
##########################################################
Class: Person
Name:
Age: 0
Class: Department
Name: Software engineering

Class: Person
Name: Dobler
Age: 50
Class: Teacher
Id: P00000001
Department: Class: Person
Name:
Age: 0
Class: Department
Name: Software engineering

Class: Person
Name: Thomas
Age: 0
Class: Student
Id: S1610454013

##########################################################

##########################################################
testShapes()
##########################################################
Class: Coordinate
XCoordinate: 1
YCoordinate: 2

Class: BaseShape
Name: Rectangular
Coordinate: Class: Coordinate
XCoordinate: 1
YCoordinate: 2
Stroke: 1
StrokeHexColor: #000000
FillColor: #ffffff
Class: Rectangular
Width: 2
Height: 1

Class: BaseShape
Name: Circle
Coordinate: Class: Coordinate
XCoordinate: 1
YCoordinate: 2
Stroke: 1
StrokeHexColor: #000000
FillColor: #ffffff
Class: Circle
Radius: 2

##########################################################
```

Abbildung 1: *Class* Generator Testausgabe

2 Clone 'Em All

2.1 Lösungsidee

Dieser Abschnitt beschäftigt sich mit der Dokumentation der Lösungsidee der Aufgabenstellung *Clone 'Em All*. Diese Aufgabenstellung wurde bereits in der Übung implementiert und nachträglich formatiert und einige Kommentare eingefügt.

2.2 Quelltexte

In diesem Abschnit werden nur die selbst implementierten Quelltexte angeführt.

Listing 5: CloningGenerator.tt

```
<#@ template debug="true" language="C#" hostSpecific="true" #>
<#@ output extension=".cs" #>
<#@ assembly name="System.Core.dll" #>
<#@ import namespace="System.Reflection" #>
<#@ import namespace="System.IO" #>
<#@ import namespace="System.Linq" #>
<#
        string assemblyFile = @"bin\Debug\CloningGenerator.exe";
        Assembly assembly = LoadProjectAssembly(assemblyFile);
        var deepClonables = assembly.GetTypes().Where(x => !x.IsInterface && x.Name !=
 ↪  "DeepCloneable" && x.GetInterfaces()
                                                .Any(y => y.Name == "IDeepCloneable"));
#>
namespace CloningGenerator {
<#
    // Iterate over each found type
    foreach (var t in deepClonables)
        {
#>
public partial class <#= t.Name #> {

// Clone constructor
protected <#= t.Name #>(<#= t.Name #> original, Cloner cloner): base(original, cloner) {
<#
            // Iterate over each found field of the found type
      foreach (var fieldInfo in t.GetFields())
            {
                var fieldType = fieldInfo.FieldType;
                // If field type is instance of IDeepClonable
                if (fieldType.GetInterfaces().Any(y => y.Name == "IDeepCloneable"))
                {
#>
this.<#= fieldInfo.Name #> = cloner.Clone(original.<#= fieldInfo.Name #>);
<#
                }
                // If field type is instance of Clonable
                else if (typeof(ICloneable).IsAssignableFrom(fieldType))
                {
#>
this.<#= fieldInfo.Name #> = (<#= fieldType.FullName #>) original.<#= fieldInfo.Name #>.Clone();
<#
                }
                // If field type is primitve one
                else
                {
#>
this.<#= fieldInfo.Name #> = original.<#= fieldInfo.Name #>;
```

```
<#
                }
            }
#>
}

public override IDeepCloneable Clone(Cloner cloner){
  return new <#= t.Name #>(this, cloner);
}

}
<#
        }
#>
}
<#
  // Cleanup after generation
  Cleanup();
#>
<#+
    // Class feature blocks for util methods
    private string outputDir;

    private Assembly LoadProjectAssembly(string assemblyFile)
    {
        if (!Path.IsPathRooted(assemblyFile))
        {
            assemblyFile = Path.Combine(Path.GetDirectoryName(Host.TemplateFile), assemblyFile);
        }
        outputDir = Path.GetDirectoryName(assemblyFile);
        if (!File.Exists(assemblyFile))
        {
            throw new ArgumentException(string.Format("Project assembly file could not be located
 ↪  at {0}.",
                assemblyFile));
        }
        AppDomain.CurrentDomain.AssemblyResolve += ResolveAssembly;

        return Assembly.Load(File.ReadAllBytes(assemblyFile));
    }

    private void Cleanup()
    {
        AppDomain.CurrentDomain.AssemblyResolve -= ResolveAssembly;
    }

    private Assembly ResolveAssembly(object sender, ResolveEventArgs args)
    {
        string dependency = Path.Combine(outputDir, args.Name.Substring(0, args.Name.IndexOf(','))
 ↪  + ".dll");
        if (File.Exists(dependency))
        {
            return Assembly.Load(File.ReadAllBytes(dependency));
        }

        return null;
    }
#>
```

Listing 6: A.cs

```cs
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace CloningGenerator
{
    // Test class for cloning generator
    public partial class A : DeepCloneable
    {
        public int i;
        public B b;
        public A() { }
    }
}
```

Listing 7: B.cs

```cs
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace CloningGenerator
{
    // Test class for cloning generator
    public partial class B : DeepCloneable
    {
        public string s;
        public int[] arr;
        public B() { }
    }

}
```

Listing 8: Program.cs

```cs
using CloningGenerator;
using System;

namespace ClassGenerator
{
    class Program
    {
        static void Main(string[] args)
        {
            var a = new A();
            var b = new B
            {
                s = "Hello world",
                arr = new[] { 1, 2, 3 }
            };

            a.b = b;

            Console.WriteLine("######################################################## \n"
                              + "Before clone\n"
                              + "########################################################");
            Console.WriteLine("a.i    : " + a.i);
```

```
23              Console.WriteLine("a.b.s  : " + a.b.s);
24              Console.WriteLine("a.b.arr: " + a.b.arr[0]);
25              Console.WriteLine("#########################################################\n");
26
27              // clone object
28              var clone = (A)a.Clone();
29
30              // change value
31              a.i = 1337;
32              a.b.s = "bye world";
33              a.b.arr[0] = 0;
34
35              Console.WriteLine("######################################################## \n"
36                              + "After clone\n"
37                              + "########################################################");
38              Console.WriteLine("a.i    : " + a.i);
39              Console.WriteLine("a.b.s  : " + a.b.s);
40              Console.WriteLine("a.b.arr: " + a.b.arr[0] + "\n");
41
42              Console.WriteLine("clone.i    : " + clone.i);
43              Console.WriteLine("clone.b.s  : " + clone.b.s);
44              Console.WriteLine("clone.b.arr: " + clone.b.arr[0] + "\n");
45              Console.WriteLine("#########################################################\n");
46
47              Console.ReadLine();
48          }
49      }
50  }
```

2.3 Tests

Dieser Abschnitt enthält die Tests in Form von den generierten Quelltexten und der Ausgabe des Testprogramms.

Listing 9: CloningGenerator.cs

```
1   namespace CloningGenerator
2   {
3       public partial class A
4       {
5   // Clone constructor
6           protected A(A original, Cloner cloner) : base(original, cloner)
7           {
8               this.i = original.i;
9               this.b = cloner.Clone(original.b);
10          }
11
12          public override IDeepCloneable Clone(Cloner cloner)
13          {
14              return new A(this, cloner);
15          }
16      }
17
18      public partial class B
19      {
20  // Clone constructor
21          protected B(B original, Cloner cloner) : base(original, cloner)
22          {
23              this.s = (System.String) original.s.Clone();
24              this.arr = (System.Int32[]) original.arr.Clone();
25          }
```

```
26
27          public override IDeepCloneable Clone(Cloner cloner)
28          {
29              return new B(this, cloner);
30          }
31      }
32  }
```

```
##########################################################
Before clone
##########################################################
a.i    : 0
a.b.s  : Hello world
a.b.arr: 1
##########################################################

##########################################################
After clone
##########################################################
a.i    : 1337
a.b.s  : bye world
a.b.arr: 0

clone.i    : 0
clone.b.s  : Hello world
clone.b.arr: 1

##########################################################
```

Abbildung 2: *Clone* Generator Testausgabe

3 SVG Generator

Dieser Abschnitt beschäftigt sich mit der Dokumentation der Lösungsidee der Aufgabenstellung *SVG Generator*.

3.1 Lösungsidee

3.2 Quelltexte

3.3 Tests