GP 2	Generative Programming	ST 17, Exercise 5

Deadline: 09.06.2017, 10:30

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

Name _____ **Effort in h** _____

Points _____ **Lecturer** _____

1. Dynamic Generation of Proxies using Reflection.Emit (6 + 6 Points)

The design pattern *Proxy* and its importance in, e.g., distributed applications (for example remote proxies) should be known to you; in order to make the usage of proxies as efficient and easy as possible, it would be nice to be able to generate proxies for arbitrary objects at runtime. Luckily, this can be realized using generative concepts:

a) Use the.NET Reflection.Emit API for generating proxy objects at runtime; implement the static class *ProxyGenerator* with the static method *Create* that takes the object, for which the proxy shall be generated, as parameter, and returns a reference to the generated proxy object.

   Using Reflection.Emit the *Create* method shall dynamically generate a new type that stores a reference to the original object as well as implements all methods of the interfaces implemented by the original object. In these methods simply delegate the execution to the encapsulated object.

b) Extend your implementation of dynamic proxies by implementing a mechanism for inducing arbitrary code before and after a call of the original object.

   Write a simple interface *IInterception* that defines the methods *void Before()* and *void After(),* and extend the interface of the *Create* method of the *ProxyGenerator* by a parameter of this type. The proxy object generated by the *ProxyGenerator* shall then call the methods *Before* and *After* each time a method is called and delegated to the encapsulated object.

   Test your implementation by implementing a class *LogInterception* that writes simple log messages to the console.

2. Dynamic Generation of Functions (6 + 6 Points)

The basis of many applications in symbolic computation (as for example genetic programming) is the definition of functions and terminals. Functions are in this context entities that take inputs and calculate return values, whereas terminals read and return data from a data base.

a) First, your task is to use the NET CodeDOM API framework for implementing a mechanism that takes functions and terminals (i.e., their code), compiles these and returns compiled methods that are ready to use.

   The basis of these functions and terminals shall be the following definitions:

```
public delegate double TerminalEvaluation(double[][] data,
  int sampleIndex, int variableIndex, double coefficient)
public delegate double FunctionEvaluation(double[] parameters)
```

   For evaluating terminals we obviously need information, which variable of the data base and which sample shall be evaluated, whereas functions do not need any further parameters.

For generating test data you can for example use the following code:

```
int n = 1000;
double[][] testData = new double[4][];
for (int i = 0; i < 4; i++) {
  testData[i] = new double[n];
}
for (int i = 0; i < n; i++) {
  testData[0][i] = i + random.NextDouble() - 0.5;
  testData[1][i] = Math.Sin(i / 100.0 * 2.0 * Math.PI);
  testData[2][i] = testData[0][i] + testData[1][i];
  testData[3][i] = testData[2][i] * testData[1][i];
}
```

b) Develop a mechanism for representing formulas based on structure trees that use the functions and terminals generated in the previous task. The structure node is the core element of structure trees. Nodes shall be able to represent terminals as well as functions and store references to child nodes; additionally we want to manage relevant parameters for evaluating terminals (*VariableIndex*, *Coefficient*).

An exemplary formula, that should be able to be represented, could look as follows: ADD ([1.2*VAR_1], MULT ([-2.9*VAR_0], [0.3*VAR_2]))

The structure of the nodes developed here should be based on the following definition:

```
public class Node {
  public TerminalEvaluation TerminalEvaluation
  public FunctionEvaluation FunctionEvaluation
  public int VariableIndex
  public double Coefficient
  public List<Node> Children
  public Node(TerminalEvaluation TerminalEvaluation, int VariableIndex,
    double Coefficient)
  public Node(FunctionEvaluation FunctionEvaluation, List<Node> Children)
  public double Evaluate(double[][] Data, int SampleIndex)
}
```

Please note: - Test all implementations extensively

- Take care about the proper structuring of your code as well as comments