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

1 Dynamic Generation of Proxies using Reflection.Emit

Dieser Abschnitt behandelt die Aufgabenstellung *Dynamic Generation of Proxies using Reflection.Emit*.

1.1 Lösungsidee

Es wird die Klasse *ProxyGenerator* implementiert, welche die beiden generischen Methoden *Create< T >(T obj)* und *Create< T >(T obj, IInterception< T > interceptor)* zur Verfügung stellt, die für das übergebene Objekt eine Proxyklasse und ein Proxyobjekt erstellen und das erstellte Proxyobjekt zurückliefern. Damit der Proxy auch bei einem *Cast* auf den implementierten Typ greift, müssen die implementierten Methoden der Schnitstelle in der implementierenden Klasse als *virtual* markiert werden, ansonsten wird die Implementierung implementierenden Klasse aufgerufen und nicht die überschriebenen Methoden des Proxy, was an der Art und Weise der Handhabung von der dynamischen Bindung in C# liegt.

Es werden alle Methoden im Proxy überschrieben, jedoch wird der *Interceptor* nur bei den Methoden, die im Typ *T* zur Verfügung stehen eingefügt. Ein Interceptor wird durch die Schnittstelle *IInterceptor* spezifiziert, wobei die beiden Methoden *Before* und *After* von einem *Interceptor* implementiert werden müssen. Diese beiden Methoden *Before* und *After* bekommen das Objekt vom Typ *T* und den Methodennamen übergeben, sodass der *Interceptor* mit dem Objekt interagieren kann und auch die Information hat welche Methode abgefangen wird.

Es wird die Klasse *LogInterception* implementiert, welche die Schnittstelle *IInterception* implementiert und bei einem Aufruf der Methoden *Before* und *After Logs* generiert.

1.2 Quelltexte

Dieser Abschnitt beinhaltet die implementierten Quelltexte und das Testprogramm.

Listing 1: IInterception.cs

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using System.Threading.Tasks;
6
7  namespace Reflection.Emit
8  {
9      /// <summary>
10     /// Specifies an interceptor which can perform before and after actions
11     /// on a intercepted method invocation.
12     /// </summary>
13     public interface IInterception<in T>
14     {
15         /// <summary>
16         /// Action performed before the actual method execution.
17         /// </summary>
18         /// <param name="intercepted">the intercepted object</param>
19         /// <param name="methodName">the intercepted method name</param>
20         void Before(T intercepted, string methodName);
21
22         /// <summary>
23         /// Action performed after the actual method execution
24         /// </summary>
25         /// <param name="intercepted">the intercepted object</param>
26         /// <param name="methodName">the intercepted method name</param>
```

```
27          void After(T intercepted, string methodName);
28      }
29  }
```

Listing 2: Interfaces.cs

```
1   using System;
2
3   namespace Reflection.Emit
4   {
5       /// <summary>
6       /// Test interface for implementing class to be proxied
7       /// </summary>
8       public interface ITest
9       {
10          int DoStuff(int i, int j);
11
12          void DoVoidStuff();
13      }
14
15      /// <summary>
16      /// Second interface wich gets not intercepted
17      /// </summary>
18      public interface ISecondTest
19      {
20          string DoOtherStuff();
21
22          void DoOtherVoidStuff();
23      }
24  }
```

Listing 3: LogInterception.cs

```
1   using System;
2   using System.Collections.Generic;
3   using System.Linq;
4   using System.Text;
5   using System.Threading.Tasks;
6
7   namespace Reflection.Emit
8   {
9       /// <summary>
10      /// Implemention of IInterception for logging the method interception.
11      /// </summary>
12      public class LogInterception<T> : IInterception<T>
13      {
14          private readonly string _typeFullName;
15
16          public LogInterception()
17          {
18              var type = typeof(LogInterception<T>);
19              _typeFullName = type.Namespace + "." + type.Name;
20          }
21
22          public void After(T intercepted, string methodName)
23          {
24              Console.WriteLine($"{_typeFullName}#After  called for '{intercepted}#{methodName}'");
25          }
26
27          public void Before(T intercepted, string methodName)
```

```
28          {
29              Console.WriteLine($"{_typeFullName}#Before called for '{intercepted}#{methodName}'");
30          }
31      }
32  }
```

Listing 4: Test.cs

```
1   using System;
2
3   namespace Reflection.Emit
4   {
5       /// <summary>
6       /// Implemention for testing proxied instance.
7       /// Methods are marked as virtual, so that proxy class can override them.
8       /// </summary>
9       public class Test : ITest, ISecondTest
10      {
11          public virtual int DoStuff(int i, int j)
12          {
13              return i + j;
14          }
15
16          public virtual void DoVoidStuff()
17          {
18              Console.WriteLine("void DoVoidStuff() called");
19          }
20
21
22          public virtual string DoOtherStuff()
23          {
24              return "string DoOtherStuff() called";
25          }
26
27          public virtual void DoOtherVoidStuff()
28          {
29              Console.WriteLine("void DoOtherVoidStuff() called");
30          }
31      }
32  }
```

Listing 5: ProxyGenerator.cs

```
1   using System;
2   using System.Collections;
3   using System.Linq;
4   using System.Reflection;
5   using System.Reflection.Emit;
6   using Reflection.Emit;
7
8   namespace Reflection.Emit
9   {
10      /// <summary>
11      /// This class provides methods for generating a proxy for any object instance.
12      /// </summary>
13      public static class ProxyGenerator
14      {
15          private static readonly string AssemblyName = typeof(ProxyGenerator).Namespace + ".Proxy";
16          private static readonly string ModuleName = AssemblyName;
17          private static readonly string AssemblyFullName = AssemblyName + ".dll";
```

```
18          private const string WrappedFieldName = "proxied";
19          private const string InterceptorFieldName = "interceptor";
20
21          /// <summary>
22          /// Creates the proxied object for the given object.
23          /// </summary>
24          /// <typeparam name="T">the type of the given, proxied and intercepted object</typeparam>
25          /// <param name="obj">the object to proxy</param>
26          /// <returns>the proxied object</returns>
27          public static T Create<T>(T obj)
28          {
29              return Create<T>(obj, null);
30          }
31
32          /// <summary>
33          /// Creates the proxied object for the given object and applies the interceptor to the
    ↪   methods
34          /// accessible by the Type T.
35          /// </summary>
36          /// <typeparam name="T">the type of the given, proxied and intercepted object</typeparam>
37          /// <param name="obj">the object to proxy</param>
38          /// <param name="interceptor">the interceptor applied to the proxied object method
    ↪   invocations</param>
39          /// <returns>the proxied object</returns>
40          public static T Create<T>(T obj, IInterception<T> interceptor)
41          {
42              var wrappedType = obj.GetType();
43              var interceptorType = typeof(IInterception<T>);
44              FieldBuilder wrappedFieldBuilder, interceptorFieldBuilder;
45
46              #region Create dynamic assembly
47
48              var assemblyBuilder = AppDomain.CurrentDomain.DefineDynamicAssembly(new
    ↪   AssemblyName(AssemblyName),
49                  AssemblyBuilderAccess.RunAndSave);
50
51              var moduleBuilder = assemblyBuilder.DefineDynamicModule(ModuleName, AssemblyFullName);
52
53              #endregion
54
55              #region Create type builder
56
57              var typeBuilder = moduleBuilder.DefineType(
58                  wrappedType.FullName + "Proxy", wrappedType.Attributes, wrappedType,
    ↪   wrappedType.GetInterfaces());
59
60              #endregion
61
62              #region Create wrapped and interceptor field builder
63
64              wrappedFieldBuilder = typeBuilder.DefineField(
65                  WrappedFieldName, wrappedType, FieldAttributes.Private |
    ↪   FieldAttributes.InitOnly);
66              interceptorFieldBuilder = typeBuilder.DefineField(
67                  InterceptorFieldName, interceptorType, FieldAttributes.Private |
    ↪   FieldAttributes.InitOnly);
68
69              #endregion
70
71              #region Create proxy constructor
72
73              var constructorBuilder = typeBuilder.DefineConstructor(
74                  MethodAttributes.Public | MethodAttributes.HideBySig,
```

```
75                  CallingConventions.Standard, new[] {wrappedType, interceptorType});
76
77              var il = constructorBuilder.GetILGenerator();
78              // call this.base()
79              il.Emit(OpCodes.Ldarg_0);
80              il.Emit(OpCodes.Call, wrappedType.GetConstructor(new Type[0]));
81
82              // store argument obj into wrapped
83              il.Emit(OpCodes.Ldarg_0); // this
84              il.Emit(OpCodes.Ldarg_1); // wrapped
85              il.Emit(OpCodes.Stfld, wrappedFieldBuilder);
86              il.Emit(OpCodes.Ldarg_0); // this
87              il.Emit(OpCodes.Ldarg_2); // interceptor
88              il.Emit(OpCodes.Stfld, interceptorFieldBuilder);
89              il.Emit(OpCodes.Ret);
90
91              #endregion
92
93              #region Create proxied methods
94
95              // only intercept interface methods visible to interceptor
96              IList interceptableInterfaces;
97              if (typeof(T).IsInterface)
98              {
99                  interceptableInterfaces = new ArrayList()
100                 {
101                     typeof(T)
102                 };
103             }
104             else
105             {
106                 interceptableInterfaces = new ArrayList(wrappedType.GetInterfaces());
107             }
108
109             foreach (var interfaceType in wrappedType.GetInterfaces())
110             {
111                 foreach (var method in interfaceType.GetMethods())
112                 {
113                     var methodInfo = wrappedType.GetMethod(method.Name);
114                     var parameterTypes = methodInfo
115                         .GetParameters()
116                         .Select(x => x.ParameterType)
117                         .ToArray();
118                     var methodBuilder = typeBuilder.DefineMethod(
119                         methodInfo.Name,
120                         methodInfo.Attributes,
121                         methodInfo.ReturnType,
122                         parameterTypes);
123
124                     // Ensure that proxied method is called, even when proxied object is cast to
    ↪   implementation class
125                     // Needs the implementation class to mark methods as virtual
126                     if (methodInfo.IsVirtual)
127                     {
128                         typeBuilder.DefineMethodOverride(methodBuilder, methodInfo);
129                     }
130
131                     il = methodBuilder.GetILGenerator();
132
133                     // before interceptor call if interceptor is present
134                     if ((interceptor != null) &&
    ↪   (interceptableInterfaces.Contains(interfaceType)))
135                     {
```

```
136                         il.Emit(OpCodes.Ldarg_0); // load this
137                         il.Emit(OpCodes.Ldfld, interceptorFieldBuilder); // load interceptor field
138                         il.Emit(OpCodes.Ldarg_0); // load this
139                         il.Emit(OpCodes.Ldfld, wrappedFieldBuilder);     // load wrapper field
140                         il.Emit(OpCodes.Ldstr, methodInfo.Name);         // load intercepted
    ↪  method name
141                         il.Emit(OpCodes.Callvirt,
    ↪  interceptorFieldBuilder.FieldType.GetMethod(nameof(IInterception<T>.Before)));
142                     }
143
144                     il.Emit(OpCodes.Ldarg_0); // load this
145                     il.Emit(OpCodes.Ldfld, wrappedFieldBuilder); // load wrapper field
146                     // load all parameters
147                     for (byte i = 1; i <= parameterTypes.Length; i++)
148                     {
149                         il.Emit(OpCodes.Ldarg_S, i);
150                     }
151                     il.Emit(OpCodes.Callvirt,
    ↪  wrappedFieldBuilder.FieldType.GetMethod(methodInfo.Name));
152
153                     // Void return type cannot be saved as local variable in method body
154                     if (typeof(void) != methodInfo.ReturnType)
155                     {
156                         var localBuilder = il.DeclareLocal(methodInfo.ReturnType);
157                         il.Emit(OpCodes.Stloc, localBuilder);
158                         il.Emit(OpCodes.Ldloc, localBuilder);
159                     }
160
161                     // after interceptor call if interceptor is present
162                     if ((interceptor != null) &&
    ↪  (interceptableInterfaces.Contains(interfaceType)))
163                     {
164                         il.Emit(OpCodes.Ldarg_0); // load this
165                         il.Emit(OpCodes.Ldfld, interceptorFieldBuilder); // load interceptor field
166                         il.Emit(OpCodes.Ldarg_0); // load this
167                         il.Emit(OpCodes.Ldfld, wrappedFieldBuilder);     // load wrapper field
168                         il.Emit(OpCodes.Ldstr, methodInfo.Name);         // load intercepted
    ↪  method name
169                         il.Emit(OpCodes.Callvirt,
    ↪  interceptorFieldBuilder.FieldType.GetMethod(nameof(IInterception<T>.After)));
170                     }
171
172                     il.Emit(OpCodes.Ret);
173                 }
174             }
175
176             #endregion
177
178             var createdType = typeBuilder.CreateType();
179             assemblyBuilder.Save(AssemblyFullName);
180             return (T) Activator.CreateInstance(createdType, obj, interceptor);
181         }
182     }
183 }
```

Listing 6: Program.cs

```
1  using System;
2
3  namespace Reflection.Emit
4  {
5      /// <summary>
```

```
6       /// Test programm for testing the generated proxy and interceptor
7       /// </summary>
8       class Program
9       {
10          static void Main(string[] args)
11          {
12              var firstObj = new Test();
13              var secondObj = new Test();
14              var thirdObj = new Test();
15              var firstInterceptor = new LogInterception<ITest>();
16              var secondInterceptor = new LogInterception<ISecondTest>();
17              var thirdInterceptor = new LogInterception<Test>();
18
19              try
20              {
21                  Console.WriteLine("------------------------------------------------");
22                  Console.WriteLine("DynamicProxy Tests ProxyGenerator.Create<ITest>():");
23                  Console.WriteLine("------------------------------------------------");
24                  Console.WriteLine("------------------------------------------------");
25                  Console.WriteLine("testNoInterceptor:");
26                  Console.WriteLine("------------------------------------------------");
27                  InvokeMethodsITest(firstObj);
28                  Console.WriteLine("------------------------------------------------");
29                  Console.WriteLine("testWithInterceptor:");
30                  Console.WriteLine("------------------------------------------------");
31                  // Create proxy and apply interceptor on ITest interface methods
32                  firstObj = (Test)ProxyGenerator.Create(firstObj, firstInterceptor);
33                  InvokeMethodsITest(firstObj);
34                  Console.WriteLine("------------------------------------------------");
35                  Console.WriteLine("");
36
37                  Console.WriteLine("------------------------------------------------");
38                  Console.WriteLine("DynamicProxy Tests ProxyGenerator.Create<ISecondTest>():");
39                  Console.WriteLine("------------------------------------------------");
40                  Console.WriteLine("------------------------------------------------");
41                  Console.WriteLine("testNoInterceptor:");
42                  Console.WriteLine("------------------------------------------------");
43                  InvokeMethodsITest(secondObj);
44                  Console.WriteLine("------------------------------------------------");
45                  Console.WriteLine("testWithInterceptor:");
46                  Console.WriteLine("------------------------------------------------");
47                  // Create proxy and apply interceptor on ISecondTest interface methods
48                  secondObj = (Test)ProxyGenerator.Create(secondObj, secondInterceptor);
49                  InvokeMethodsITest(secondObj);
50                  Console.WriteLine("------------------------------------------------");
51
52                  Console.WriteLine("------------------------------------------------");
53                  Console.WriteLine("DynamicProxy Tests ProxyGenerator.Create<Test>():");
54                  Console.WriteLine("------------------------------------------------");
55                  Console.WriteLine("------------------------------------------------");
56                  Console.WriteLine("testNoInterceptor:");
57                  Console.WriteLine("------------------------------------------------");
58                  InvokeMethodsITest(thirdObj);
59                  Console.WriteLine("------------------------------------------------");
60                  Console.WriteLine("testWithInterceptor:");
61                  Console.WriteLine("------------------------------------------------");
62                  // Create proxy and apply interceptor on Test methods
63                  thirdObj = ProxyGenerator.Create(thirdObj, thirdInterceptor);
64                  InvokeMethodsITest(thirdObj);
65                  Console.WriteLine("------------------------------------------------");
66              }
67              catch (Exception e)
68              {
```

```
69                  Console.WriteLine($"Exception occured: {e}");
70              }
71          }
72
73          /// <summary>
74          /// Performs method invocation on ITest instances.
75          /// </summary>
76          /// <param name="obj">the ITest instance</param>
77          private static void InvokeMethodsITest(Test obj)
78          {
79              // ITest methods
80              Console.WriteLine($"DoStuff(1, 2): {obj.DoStuff(1, 2)}");
81              Console.WriteLine($"DoStuff(2, 2): {obj.DoStuff(2, 2)}");
82              Console.WriteLine($"DoStuff(4, 4): {obj.DoStuff(4, 4)}");
83              obj.DoVoidStuff();
84              // ISecondTest methods
85              Console.WriteLine($"DoOtherStuff: {obj.DoOtherStuff()}");
86              Console.WriteLine($"DoOtherStuff: {obj.DoOtherStuff()}");
87              Console.WriteLine($"DoOtherStuff: {obj.DoOtherStuff()}");
88              obj.DoOtherVoidStuff();
89          }
90      }
91  }
```

1.3 Tests

Dieser Abschnitt behandelt die Tests in Form von Ausgaben der *Logs*.

```
------------------------------------------------
DynamicProxy Tests ProxyGenerator.Create<ITest>():
------------------------------------------------
------------------------------------------------
testNoInterceptor:
------------------------------------------------
DoStuff(1, 2): 3
DoStuff(2, 2): 4
DoStuff(4, 4): 8
void DoVoidStuff() called
DoOtherStuff: string DoOtherStuff() called
DoOtherStuff: string DoOtherStuff() called
DoOtherStuff: string DoOtherStuff() called
void DoOtherVoidStuff() called
------------------------------------------------
testWithInterceptor:
------------------------------------------------
Reflection.Emit.LogInterception`1#Before called for 'Reflection.Emit.Test#DoStuff'
Reflection.Emit.LogInterception`1#After  called for 'Reflection.Emit.Test#DoStuff'
DoStuff(1, 2): 3
Reflection.Emit.LogInterception`1#Before called for 'Reflection.Emit.Test#DoStuff'
Reflection.Emit.LogInterception`1#After  called for 'Reflection.Emit.Test#DoStuff'
DoStuff(2, 2): 4
Reflection.Emit.LogInterception`1#Before called for 'Reflection.Emit.Test#DoStuff'
Reflection.Emit.LogInterception`1#After  called for 'Reflection.Emit.Test#DoStuff'
DoStuff(4, 4): 8
Reflection.Emit.LogInterception`1#Before called for 'Reflection.Emit.Test#DoVoidStuff'
void DoVoidStuff() called
Reflection.Emit.LogInterception`1#After  called for 'Reflection.Emit.Test#DoVoidStuff'
DoOtherStuff: string DoOtherStuff() called
DoOtherStuff: string DoOtherStuff() called
DoOtherStuff: string DoOtherStuff() called
void DoOtherVoidStuff() called
------------------------------------------------
```

Abbildung 1: Test für *Interception* von *ITest*

```
-------------------------------------------------
DynamicProxy Tests ProxyGenerator.Create<ISecondTest>():
-------------------------------------------------
-------------------------------------------------
testNoInterceptor:
-------------------------------------------------
DoStuff(1, 2): 3
DoStuff(2, 2): 4
DoStuff(4, 4): 8
void DoVoidStuff() called
DoOtherStuff: string DoOtherStuff() called
DoOtherStuff: string DoOtherStuff() called
DoOtherStuff: string DoOtherStuff() called
void DoOtherVoidStuff() called
-------------------------------------------------
testWithInterceptor:
-------------------------------------------------
DoStuff(1, 2): 3
DoStuff(2, 2): 4
DoStuff(4, 4): 8
void DoVoidStuff() called
Reflection.Emit.LogInterception`1#Before called for 'Reflection.Emit.Test#DoOtherStuff'
Reflection.Emit.LogInterception`1#After  called for 'Reflection.Emit.Test#DoOtherStuff'
DoOtherStuff: string DoOtherStuff() called
Reflection.Emit.LogInterception`1#Before called for 'Reflection.Emit.Test#DoOtherStuff'
Reflection.Emit.LogInterception`1#After  called for 'Reflection.Emit.Test#DoOtherStuff'
DoOtherStuff: string DoOtherStuff() called
Reflection.Emit.LogInterception`1#Before called for 'Reflection.Emit.Test#DoOtherStuff'
Reflection.Emit.LogInterception`1#After  called for 'Reflection.Emit.Test#DoOtherStuff'
DoOtherStuff: string DoOtherStuff() called
Reflection.Emit.LogInterception`1#Before called for 'Reflection.Emit.Test#DoOtherVoidStuff'
void DoOtherVoidStuff() called
Reflection.Emit.LogInterception`1#After  called for 'Reflection.Emit.Test#DoOtherVoidStuff'
-------------------------------------------------
```

Abbildung 2: Test für *Interception* von *ISecondTest*

```
-------------------------------------------------
DynamicProxy Tests ProxyGenerator.Create<Test>():
-------------------------------------------------
-------------------------------------------------
testNoInterceptor:
-------------------------------------------------
DoStuff(1, 2): 3
DoStuff(2, 2): 4
DoStuff(4, 4): 8
void DoVoidStuff() called
DoOtherStuff: string DoOtherStuff() called
DoOtherStuff: string DoOtherStuff() called
DoOtherStuff: string DoOtherStuff() called
void DoOtherVoidStuff() called
-------------------------------------------------
testWithInterceptor:
-------------------------------------------------
Reflection.Emit.LogInterception`1#Before called for 'Reflection.Emit.Test#DoStuff'
Reflection.Emit.LogInterception`1#After  called for 'Reflection.Emit.Test#DoStuff'
DoStuff(1, 2): 3
Reflection.Emit.LogInterception`1#Before called for 'Reflection.Emit.Test#DoStuff'
Reflection.Emit.LogInterception`1#After  called for 'Reflection.Emit.Test#DoStuff'
DoStuff(2, 2): 4
Reflection.Emit.LogInterception`1#Before called for 'Reflection.Emit.Test#DoStuff'
Reflection.Emit.LogInterception`1#After  called for 'Reflection.Emit.Test#DoStuff'
DoStuff(4, 4): 8
Reflection.Emit.LogInterception`1#Before called for 'Reflection.Emit.Test#DoVoidStuff'
void DoVoidStuff() called
Reflection.Emit.LogInterception`1#After  called for 'Reflection.Emit.Test#DoVoidStuff'
Reflection.Emit.LogInterception`1#Before called for 'Reflection.Emit.Test#DoOtherStuff'
Reflection.Emit.LogInterception`1#After  called for 'Reflection.Emit.Test#DoOtherStuff'
DoOtherStuff: string DoOtherStuff() called
Reflection.Emit.LogInterception`1#Before called for 'Reflection.Emit.Test#DoOtherStuff'
Reflection.Emit.LogInterception`1#After  called for 'Reflection.Emit.Test#DoOtherStuff'
DoOtherStuff: string DoOtherStuff() called
Reflection.Emit.LogInterception`1#Before called for 'Reflection.Emit.Test#DoOtherStuff'
Reflection.Emit.LogInterception`1#After  called for 'Reflection.Emit.Test#DoOtherStuff'
DoOtherStuff: string DoOtherStuff() called
Reflection.Emit.LogInterception`1#Before called for 'Reflection.Emit.Test#DoOtherVoidStuff'
void DoOtherVoidStuff() called
Reflection.Emit.LogInterception`1#After  called for 'Reflection.Emit.Test#DoOtherVoidStuff'
-------------------------------------------------
```

Abbildung 3: Test für *Interception* von *Test*

2 Dynamic Generation of Functions

Dieser Abschnitt behandelt die Aufgabenstellung *Dynamic Generation of Functions*.

2.1 Lösungsidee

Ein Großteil der Implementierungen wurde bereits in der Übung implementiert, jedoch wurden die folgenden Veränderungen vorgenommen.

Die Methode *CompileTerminal* verlangt die Argumente in Form eines Strings Arrays und den Datentyp des Resultats der zu erzeugenden Methode, damit der Aufrufer nicht abhängig ist von statische Definitionen in der Implementierung dieser Methode.

Die Methode *CompileFunction* verlangt das Argument in Form eines Strings und den Datentyp des Resultats der zu erzeugenden Methode, damit der Aufrufer nicht abhängig ist von statische Definitionen in der Implementierung dieser Methode.

Die *delegate* Definitionen wurden in die Datei *Node.cs* verschoben, da die *delegates* die unterstützten Evaluierungsmethoden darstellen, die von den implementierten *Nodes* unterstützt werden. Die Methoden *CompileTerminal* und *CompileFunctional* liefern ein Objekt des Datentyps T und sind daher unabhängig von der konkreten Ausprägung der Terminal- und Funktionsmethode.

Es wird die Schnittstelle *INode* implementiert, die eine Node im Evaluierungsbaum darstellt. Es wird die abstrakte Klasse *BaseNode* implementiert, die alle gemeinsamen *Properties* kapselt. Es werden die Klassen *FunctionalNode* und *TerminalNode* implementiert, welche die Knoten der zwei Typen von Evaluierungsmethoden repräsentieren und die spezifischen Evaluierungen dieser Knotentypen implementieren.

2.2 Quelltexte

Dieser Abschnitt beinhaltet die implementierten Quelltexte und das Testprogramm.

Listing 7: AssertDouble.cs

```
1  using System;
2
3  namespace Symbolic.Computation
4  {
5      /// <summary>
6      /// Utility class for double assertion.
7      /// </summary>
8      public static class AssertDouble
9      {
10         /// <summary>
11         /// Asserts if the two double values are almost equal.
12         /// </summary>
13         /// <param name="actual">the actual value</param>
14         /// <param name="expected">the almost expected value</param>
15         /// <param name="epsilon">the epsilon value</param>
16         /// <returns>true if almost equal, false otherwise</returns>
17         public static bool AssertAlmostEqual(this double actual, double expected, double epsilon =
    ↪  1.0E-12)
18         {
19             if (double.IsInfinity(actual))
20                 return actual > 0
21                     ? double.IsPositiveInfinity(expected)
22                     : double.IsNegativeInfinity(expected);
```

```
            return Math.Abs(actual - expected) < epsilon;
        }
    }
}
```

Listing 8: Node.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Runtime.Remoting.Messaging;
using System.Text;
using System.Threading.Tasks;

namespace Symbolic.Computation
{
    #region Supported Evaluation methods
    public delegate double TerminalEvaluation(double[][] data, int variableIndex, int sampleIndex,
 ↪  double coefficient);

    public delegate double FunctionEvaluation(double[] parameters);
    #endregion

    #region Node specification
    /// <summary>
    /// Interface which specifies a Node.
    /// </summary>
    public interface INode
    {
        double Evaluate(double[][] data, int sampleIdx);
    }

    /// <summary>
    /// The abstract implementation of the INode interface which encapsulates the common members
    /// </summary>
    public abstract class BaseNode<T> : INode where T : class
    {
        protected readonly T Evaluation;

        /// <summary>
        /// Sets the evaluation for the node
        /// </summary>
        /// <param name="evaluation">to evaluation for this node</param>
        protected BaseNode(T evaluation)
        {
            Evaluation = evaluation ?? throw new ArgumentException("Node must hold an
 ↪  evaluation");
        }

        /// <summary>
        /// Evaluates the subtree started from this node.
        /// </summary>
        /// <param name="data">the data for the terminal evaluation</param>
        /// <param name="sampleIdx">the idx of the data sample contained in Data[][]</param>
        /// <returns></returns>
        public abstract double Evaluate(double[][] data, int sampleIdx);
    }
    #endregion

    #region Node implementation
    /// <summary>
    /// Node implementation for functional evaluations
```

```
54      /// </summary>
55      public class FunctionalNode : BaseNode<FunctionEvaluation>
56      {
57          private readonly List<INode> _children;
58
59          /// <summary>
60          /// Constructs a functional node.
61          /// </summary>
62          /// <param name="evaluation">the functional evaluation for this node</param>
63          /// <param name="children">the children of the functional node</param>
64          public FunctionalNode(FunctionEvaluation evaluation, List<INode> children) :
    ↪   base(evaluation)
65          {
66              this._children = children
67                               ?? throw new ArgumentException("Children list must not be null on a
    ↪   functional node");
68              if (children.Count == 0)
69              {
70                  throw new ArgumentException("Children list must not be empty on a functional
    ↪   node");
71              }
72          }
73
74          public override double Evaluate(double[][] data, int sampleIdx)
75          {
76              return Evaluation(_children.Select(child => child.Evaluate(data,
    ↪   sampleIdx)).ToArray());
77          }
78      }
79
80      /// <summary>
81      /// Node implementation for terminal evaluations
82      /// </summary>
83      public class TerminalNode : BaseNode<TerminalEvaluation>
84      {
85          private readonly int _variableIdx;
86          private readonly double _coefficient;
87
88          /// <summary>
89          /// Constructs a Node for a terminal evaluation
90          /// </summary>
91          /// <param name="evaluation">the terminal evaluation bound to this node</param>
92          /// <param name="variableIdx">the index of the variable</param>
93          /// <param name="coefficient">the coefficient for the terminal evaluation</param>
94          public TerminalNode(TerminalEvaluation evaluation, int variableIdx, double coefficient) :
    ↪   base(evaluation)
95          {
96              this._variableIdx = variableIdx;
97              this._coefficient = coefficient;
98          }
99
100         public override double Evaluate(double[][] data, int sampleIdx)
101         {
102             return Evaluation(data, _variableIdx, sampleIdx, _coefficient);
103         }
104     }
105     #endregion
106 }
```

Listing 9: FunctionalBasis.cs

```
1  using System;
2  using System.CodeDom.Compiler;
```

```
3  using Microsoft.CSharp;
4
5  namespace Symbolic.Computation
6  {
7      /// <summary>
8      /// This class provides static methods for creating evaluation methods
9      /// </summary>
10     public static class FunctionalBasis
11     {
12         private const string ClassName = "Definitions";
13
14         /// <summary>
15         /// Compiles a terminal evaluation method
16         /// </summary>
17         /// <typeparam name="T">The type of the create terminal method</typeparam>
18         /// <param name="methodCode">the code for the method body</param>
19         /// <param name="arguments">the string array representing the methd arguments</param>
20         /// <param name="compilationResults">the out parameter holdng the comiplation
   ↪  result</param>
21         /// <returns>the created terminal evaluation delegate</returns>
22         /// <summary>
23         public static T CompileTerminal<T>(
24             string methodCode, string[] arguments, Type resType, out CompilationResults
   ↪  compilationResults) where T:class
25         {
26             string namespaceName = typeof(TerminalEvaluation).Namespace;
27             string methodName = typeof(TerminalEvaluation).Name;
28             string resultType = resType.Name;
29             string argumentsCode = string.Join(",", arguments);
30
31             return Compile<T>(
32                 namespaceName, ClassName, methodName, resultType, argumentsCode, methodCode,
33                 out compilationResults);
34         }
35
36         /// <summary>
37         /// Creates a functional evaluation method.
38         /// </summary>
39         /// <typeparam name="T">The type of the create functional method</typeparam>
40         /// <param name="methodCode">the code for the method body</param>
41         /// <param name="argument">the string representing the method argument</param>
42         /// <param name="resType">the type of the method result</param>
43         /// <param name="compilationResults">the out parameter holdng the comiplation
   ↪  result</param>
44         /// <returns>the created functional evaluation delegate</returns>
45         public static T CompileFunction<T>(
46             string methodCode, string argument, Type resType, out CompilationResults
   ↪  compilationResults) where T : class
47         {
48             string namespaceName = typeof(TerminalEvaluation).Namespace;
49             string methodName = typeof(TerminalEvaluation).Name;
50             string resultType = resType.Name;
51             string argumentsCode = argument;
52
53             return Compile<T>(
54                 namespaceName, ClassName, methodName, resultType, argumentsCode, methodCode,
55                 out compilationResults);
56         }
57
58         /// <summary>
59         /// Compiles a evaluation method
60         /// </summary>
61         /// <typeparam name="T">the concrete type of the compilation result</typeparam>
```

```
62          /// <param name="namespaceName">the name of the namespace the owning class is in</param>
63          /// <param name="className">the name of the class</param>
64          /// <param name="methodName">the name of the method</param>
65          /// <param name="resultType">the type of the method result</param>
66          /// <param name="argumentsCode">the code representing the arguments</param>
67          /// <param name="methodCode">the code of the method body</param>
68          /// <param name="compilationResults">the out parameter holding the compilation
    ↪   result</param>
69          /// <returns>the compiled evaluation method</returns>
70          private static T Compile<T>(string namespaceName, string className,
71              string methodName, string resultType, string argumentsCode, string methodCode,
72              out CompilationResults compilationResults) where T : class
73          {
74              string code = Auxiliary.CompilableCode(
75                  namespaceName, className, methodName, resultType,
76                  argumentsCode, methodCode);
77
78              var provider = new CSharpCodeProvider();
79              var parameters = new CompilerParameters
80              {
81                  GenerateInMemory = true,
82                  TreatWarningsAsErrors = false
83              };
84
85              var compilerResults = provider.CompileAssemblyFromSource(parameters, code);
86              var result = default(T);
87              if (!compilerResults.Errors.HasErrors)
88              {
89                  var assembly = compilerResults.CompiledAssembly;
90                  var definitionsClass = assembly.GetType(
91                      string.Format("{0}.{1}", namespaceName, className));
92                  var methodInfo = definitionsClass.GetMethod(methodName);
93                  result = Delegate.CreateDelegate(typeof(T), methodInfo) as T;
94              }
95              compilationResults = new CompilationResults(compilerResults, code);
96
97              return result;
98          }
99      }
100 }
```

Listing 10: Program.cs

```
1  using System;
2  using System.Collections.Generic;
3
4  namespace Symbolic.Computation
5  {
6      /// <summary>
7      /// Console application for testing the implementations
8      /// </summary>
9      class Program
10     {
11         static void Main(string[] args)
12         {
13             // create varaible function
14             TerminalEvaluation variable = CreateTerminal(new []{
15                 "double[][] data",
16                 "int varIdx",
17                 "int sampleIdx",
18                 "double coefficient"
19             }, typeof(double), "return data[varIdx][sampleIdx] * coefficient;");
```

```
20
21              // create addition function
22              FunctionEvaluation addition = CreateFunctional("double[] data", typeof(double),
23                  "double result = 0.0; for (int i = 0; i < data.Length; i++) result += data[i];
    ↪  return result;");
24
25              // create multiplication function
26              FunctionEvaluation multiplication = CreateFunctional("double[] data", typeof(double),
27                  "double result = 1.0; for (int i = 0; i < data.Length; i++) result *= data[i];
    ↪  return result;");
28
29              Console.WriteLine("-------------------------------------------------");
30              Console.WriteLine("Test functions:");
31              Console.WriteLine("-------------------------------------------------");
32              double[][] data = CreateTestData(4, 4);
33              Console.WriteLine("TestEvaluation(() => addition([1,2,3]))");
34              Console.WriteLine("-------------------------------------------------");
35              TestEvaluation(()=> addition(new double[]{1,2,3}), 6.0);
36              Console.WriteLine("-------------------------------------------------");
37              Console.WriteLine("TestEvaluation(() => multiplication([1,2,3]))");
38              Console.WriteLine("-------------------------------------------------");
39              TestEvaluation(() => multiplication(new double[] { 1, 2, 3 }), 6.0);
40              Console.WriteLine("-------------------------------------------------");
41              Console.WriteLine("TestEvaluation(() => varaible([1,2,3], 0, 2, 2))");
42              Console.WriteLine("-------------------------------------------------");
43              TestEvaluation(() => variable(data, 0, 2, 2.0), (data[0][2] * 2.0));
44              Console.WriteLine("-------------------------------------------------");
45              Console.WriteLine();
46
47              Console.WriteLine("-------------------------------------------------");
48              Console.WriteLine("Test evaluation trees:");
49              Console.WriteLine("-------------------------------------------------");
50              Console.WriteLine("TestExample1()");
51              Console.WriteLine("-------------------------------------------------");
52              TestExample1(addition, multiplication, variable);
53              Console.WriteLine("-------------------------------------------------");
54              Console.WriteLine("TestExample2()");
55              Console.WriteLine("-------------------------------------------------");
56              TestExample2(addition, multiplication, variable);
57              Console.WriteLine("-------------------------------------------------");
58          }
59
60          /// <summary>
61          /// Tests the evaluation methods created via CodeDOM
62          /// </summary>
63          /// <param name="function">the function executing the evaluation method</param>
64          /// <param name="expected">the expected result</param>
65          private static void TestEvaluation(Func<double> function, double expected)
66          {
67              double actual = function();
68              Console.WriteLine($"actual:{actual} / expected: {actual} /
    ↪  result={actual.AssertAlmostEqual(expected)}");
69          }
70
71          /// <summary>
72          /// Tests with the first formula
73          /// </summary>
74          private static void TestExample1(FunctionEvaluation addition, FunctionEvaluation
    ↪  multiplication,
75              TerminalEvaluation variable)
76          {
77              Console.WriteLine("ADD([1.2*VAR_1], MULT([-2.9*VAR_0], [0.3*VAR_2]))");
78              Console.WriteLine("-------------------------------------------------");
```

```
79
80              const int variableCount = 3;
81              const int sampleCount = 5;
82              double[][] data = CreateTestData(variableCount, sampleCount);
83
84              INode multNode = new FunctionalNode(multiplication, new List<INode>()
85              {
86                  new TerminalNode(variable, 2, 0.3),
87                  new TerminalNode(variable, 0, -2.9)
88              });
89              INode rootNode = new FunctionalNode(addition, new List<INode>()
90              {
91                  multNode,
92                  new TerminalNode(variable, 1, 1.2)
93              });
94
95              for (var sampleIdx = 0; sampleIdx < sampleCount; sampleIdx++)
96              {
97                  double actual = rootNode.Evaluate(data, sampleIdx);
98                  double expected = ((1.2 * data[1][sampleIdx]) + (-2.9 * data[0][sampleIdx] * 0.3 *
    ↪  data[2][sampleIdx]));
99                  Console.WriteLine(
100                     $"ADD([1.2*{data[1][sampleIdx]}], "
101                     + $"MULT([-2.9*{data[0][sampleIdx]}], [0.3*{data[2][sampleIdx]}])) "
102                     + $"= {actual} "
103                     + $"= {expected} "
104                     + $"= {actual.AssertAlmostEqual(expected)} ");
105             }
106         }
107
108         /// <summary>
109         /// Tests with the second formula
110         /// </summary>
111         private static void TestExample2(FunctionEvaluation addition, FunctionEvaluation
    ↪  multiplication,
112             TerminalEvaluation variable)
113         {
114             Console.WriteLine("ADD([2*VAR_0], MULT([4*VAR_1], ADD([-2*VAR_2], MULT([4*VAR_3],
    ↪  [-2*VAR_4]))))");
115             Console.WriteLine("-------------------------------------------------");
116
117             const int variableCount = 5;
118             const int sampleCount = 5;
119             double[][] data = CreateTestData(variableCount, sampleCount);
120
121             INode mult0Node = new FunctionalNode(multiplication, new List<INode>()
122             {
123                 new TerminalNode(variable, 4, -2),
124                 new TerminalNode(variable, 3, 4)
125             });
126             INode add1Node = new FunctionalNode(addition, new List<INode>()
127             {
128                 mult0Node,
129                 new TerminalNode(variable, 2, -2)
130             });
131             INode mult2Node = new FunctionalNode(multiplication, new List<INode>()
132             {
133                 add1Node,
134                 new TerminalNode(variable, 1, 4)
135             });
136             INode rootNode = new FunctionalNode(addition, new List<INode>()
137             {
138                 mult2Node,
```

```
139                 new TerminalNode(variable, 0, 2)
140             });
141 
142             for (var sampleIdx = 0; sampleIdx < sampleCount; sampleIdx++)
143             {
144                 double actual = rootNode.Evaluate(data, sampleIdx);
145                 double expected = ((2 * data[0][sampleIdx]) +
146                                    ((4 * data[1][sampleIdx]) *
147                                     (((-2) * data[2][sampleIdx]) +
148                                      (4 * data[3][sampleIdx] * (-2) * data[4][sampleIdx]))));
149                 Console.WriteLine(
150                     $"ADD([2*{data[0][sampleIdx]}], "
151                     + $"MULT([4*{data[1][sampleIdx]}], "
152                     + $"ADD([-2*{data[2][sampleIdx]}], "
153                     + $"MULT([4*{data[3][sampleIdx]}], [-2*{data[4][sampleIdx]}]))))) "
154                     + $"= {actual} "
155                     + $"= {expected} "
156                     + $"= {actual.AssertAlmostEqual(expected)} ");
157             }
158         }
159 
160         /// <summary>
161         /// Creates the deterministic test data array
162         /// </summary>
163         /// <param name="variableCount">the count of variables to create data for</param>
164         /// <param name="sampleCount">the count of samples to create for the varaibles</param>
165         /// <returns>the created deterministic test data array</returns>
166         private static double[][] CreateTestData(int variableCount, int sampleCount)
167         {
168             double[][] testData = new double[variableCount][];
169 
170             for (int i = 1; i <= variableCount; i++)
171             {
172                 int iIdx = i - 1;
173                 testData[iIdx] = new double[sampleCount];
174                 for (int j = 1; j <= sampleCount; j++)
175                 {
176                     int jIdx = j - 1;
177                     testData[iIdx][jIdx] = i * j;
178                 }
179             }
180             return testData;
181         }
182 
183         /// <summary>
184         /// Creates a terminal evaluation method
185         /// </summary>
186         /// <param name="arguments">the string[] representing the method arguments</param>
187         /// <param name="resultType">the method result type</param>
188         /// <param name="code">the source code of the method body</param>
189         /// <returns>the created terminal evaluation method delegate</returns>
190         private static TerminalEvaluation CreateTerminal(string[] arguments, Type resultType,
    ↪   string code)
191         {
192             CompilationResults compilationResults;
193             TerminalEvaluation variable =FunctionalBasis.CompileTerminal<TerminalEvaluation>(code,
    ↪   arguments, resultType, out compilationResults);
194             if (compilationResults.HasErrors)
195             {
196                 throw new InvalidProgramException(
197                     $"Variable terminal evaluation creation failes: Error:
    ↪   {compilationResults.ErrorsExplanation}");
198             }
```

```
199
200             return variable;
201         }
202
203         /// <summary>
204         /// Creates a functional evaluation method
205         /// </summary>
206         /// <param name="argument">the string representing the method argument</param>
207         /// <param name="resultType">the method result type</param>
208         /// <param name="code">the source code of the method body</param>
209         /// <returns>the create functional evaluation method delegate</returns>
210         private static FunctionEvaluation CreateFunctional(string argument, Type resultType,
    ↪   string code)
211         {
212             CompilationResults compilationResults;
213             FunctionEvaluation function =
    ↪   FunctionalBasis.CompileFunction<FunctionEvaluation>(code, argument, resultType, out
    ↪   compilationResults);
214             if (compilationResults.HasErrors)
215             {
216                 throw new InvalidProgramException(
217                     $"Addition terminal evaluation failes: Error:
    ↪   {compilationResults.ErrorsExplanation}");
218             }
219
220             return function;
221         }
222     }
223 }
```

2.3 Tests

Dieser Abschnitt beinhaltet die implementierten Quelltexte und das Testprogramm.

```
------------------------------------------------
Test functions:
------------------------------------------------
TestEvaluation(() => addition([1,2,3]))
------------------------------------------------
actual:6 / expected: 6 / result=True
------------------------------------------------
TestEvaluation(() => multiplication([1,2,3]))
------------------------------------------------
actual:6 / expected: 6 / result=True
------------------------------------------------
TestEvaluation(() => varaible([1,2,3], 0, 2, 2))
------------------------------------------------
actual:6 / expected: 6 / result=True
------------------------------------------------

------------------------------------------------
Test evaluation trees:
------------------------------------------------
TestExample1()
------------------------------------------------
ADD([1.2*VAR_1], MULT([-2.9*VAR_0], [0.3*VAR_2]))
------------------------------------------------
ADD([1.2*2], MULT([-2.9*1], [0.3*3])) = -0.21 = -0.21 = True
ADD([1.2*4], MULT([-2.9*2], [0.3*6])) = -5.64 = -5.64 = True
ADD([1.2*6], MULT([-2.9*3], [0.3*9])) = -16.29 = -16.29 = True
ADD([1.2*8], MULT([-2.9*4], [0.3*12])) = -32.16 = -32.16 = True
ADD([1.2*10], MULT([-2.9*5], [0.3*15])) = -53.25 = -53.25 = True
------------------------------------------------
TestExample2()
------------------------------------------------
ADD([2*VAR_0], MULT([4*VAR_1], ADD([-2*VAR_2], MULT([4*VAR_3], [-2*VAR_4]))))
------------------------------------------------
ADD([2*1], MULT([4*2], ADD([-2*3], MULT([4*4], [-2*5])))) = -1326 = -1326 = True
ADD([2*2], MULT([4*4], ADD([-2*6], MULT([4*8], [-2*10])))) = -10428 = -10428 = True
ADD([2*3], MULT([4*6], ADD([-2*9], MULT([4*12], [-2*15])))) = -34986 = -34986 = True
ADD([2*4], MULT([4*8], ADD([-2*12], MULT([4*16], [-2*20])))) = -82680 = -82680 = True
ADD([2*5], MULT([4*10], ADD([-2*15], MULT([4*20], [-2*25])))) = -161190 = -161190 = True
------------------------------------------------
```

Abbildung 4: Ausgabe des Testprogramms