GP 2 | **Generative Programming** | **ST 17, Exercise 6**

Deadline: 23.06.2017, 09:40

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

Name ______________________________ **Effort in h** _____

Points ______________________________ **Lecturer** ______

1. Dynamic Generation of XML Trees with the DLR (4 + 3 + 3 Points)

The Dynamic Language Runtime (DLR) API includes *DynamicObject*, a base class for specifying dynamic behavior at runtime.

a) Derive from this class and implement *XmlCreator*, a class that offers functionality for creating arbitrary XML trees at runtime. Realize this requirement by overriding *TryGetMember*, a method that is called each time a member of the respective class is accessed via a getter. For example, the following code in a C# program

```
String xml = XmlGenerator.Create()
             .Customer
               .BeginChildElements
                 .Name
                 .Phone
                 .Address.BeginChildElements
                   .Street
                   .City
                   .Zip
                 .EndChildElements
               .EndChildElements
               .Generate();
```

should lead to the generation of the following XML tree:

```
<Customer>
  <Name />
  <Phone />
  <Address>
    <Street />
    <City />
    <Zip />
  </Address>
</Customer>
```

Implement an auxiliary class *NodeGenerator* (also derived from *DynamicObject*) that is responsible for the correct handling of XML (child) node definitions.

b) Extend your XML creator so that the values of XML nodes can be defined at their creation. E.g., the following code shall lead to the definition of a XML tree with the given information:

```
String xml = XmlGenerator.Create()
             .Customer.BeginChildElements
               .Name("Franz")
               .Phone("033577021")
               .Address.BeginChildElements
                 .Street("Street 5")
                 .City("Linz")
                 .Zip("4020")
               .EndChildElements
             .EndChildElements
             .Generate();
```

c) Finally, extend your XML creator so that it also supports the generation of attributes. E.g., the following code

```
String xml = XmlGenerator.Create()
               .Customer.BeginChildElements
                 .Name("Mayr", "FirstName", "Franz")
                 .Phone("033577021", "CountryCode", "43")
                 .Address.BeginChildElements
                   .Street("Street 5")
                   .City("Linz")
                   .Zip("4020")
                 .EndChildElements
               .EndChildElements
               .Generate();
```

should lead to the generation of the following XML tree:

```
<Customer>
  <Name FirstName="Franz">Mayr</Name>
  <Phone CountryCode="43">033577021</Phone>
  <Address>
    <Street>Street 5</Street>
    <City>Linz</City>
    <Zip>4020</Zip>
  </Address>
</Customer>
```

2. Dynamic Generation of Proxies with the DLR (6 + 4 + 4 Points)

As we have already discussed the advantages and importance of the design pattern *Proxy* we will now implement a framework for generating proxies for arbitrary types at runtime using .NET and the Dynamic Language Runtime:

a) Use the .NET DLR for generating proxy objects at runtime. Implement the class *ProxyGenerator* that is derived from *DynamicMetaObject*, the base class for dynamic logic, and override the methods *BindGetMember*, *BindSetMember*, and *BindInvokeMember* for delegating the execution of called accesses and methods to the wrapped 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.

Again, use a simple interface *IInterception* that defines the methods *void Before()* and *void After()*, and extend the interface of the *Create* method of *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.

c) Add around advice functionality to your implementation. *IInterception* should define another method *object Around(Delegate proceed, object[] args)*. Implementations of this method can call the *proceed* delegate to propagate method execution and have access to the method arguments in *args*. *Before()* and *After()* shall only be called if method execution is propagated. E.g., the following *IInterception* implementation

```
public void Before() {
  Console.WriteLine("before");
}

public void After() {
  Console.WriteLine("after");
}
```

```
public object Around(Delegate proceed, object[] args) {
  Console.WriteLine("around (before)");
  var result = proceed.DynamicInvoke();
  Console.WriteLine("around (after)");
  return result;
}
```

should produce the following output

```
around (before)
before
after
around (after)
```

You don’t have to take care of argument changes prior to the execution of *proceed*, i.e. *proceed* can always use the arguments that were passed by the caller. It should be possible to change the *result* after the *proceed* call though.

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

Please note: - Test all implementations extensively

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