LOOM .NET- An Aspect Weaving Tool

Wolfgang Schult, Peter Troeger and Andreas Polze
Hasso-Plattner-Institute
14440 Potsdam, Germany
{wolfgang.schult|peter.troeger|andreas.polze}@hpi.uni-potsdam.de

July 16, 2003

Abstract

Aspect code can be defined separately for the various programming language-entities: namespace, class, constructor, method or field. These concrete aspect implementations are grouped together and can be attached as package to a binary assembly. During aspect weaving, the LOOM.NET tool creates automatically a derived class from the original .NET class contained in an assembly. This proxy class contains the aspect code and can be compiled and linked to produce an extended version of the original assembly.

1 LOOM .NET

LOOM .NET is an aspect weaving tool which interweaves aspect code with an already compiled .NET assembly. LOOM .NET uses metadata and reflection mechanisms to examine the compiled assemblies and to generate the proxy. Reflection information are mandatory for every .NET assembly. It does n't care weather an assembly is written in java or in C#. This means that LOOM .NET works language independent. In the next sections we will take a closer look into LOOM .NET.

1.1 Metadata and Reflection in .NET

Reflection is a language mechanism, which allows access to type information during runtime. Reflection has been implemented for various object-oriented programming languages, among them Java, C#, and C++. C++ is somewhat special as it implements reflection rather as an add-on (RTTI - runtime type information) than as an inherent language feature. With .NET reflection is not only restricted to a single language, but basically anything declared as code (any .NET assembly) can be inspected using reflection techniques.

The runtime librarys reflection classes are defined in the namespace of `System.Reflection`. They build on the fact that every type (class) is derived from `Object`. There is a public method named `GetType`, which

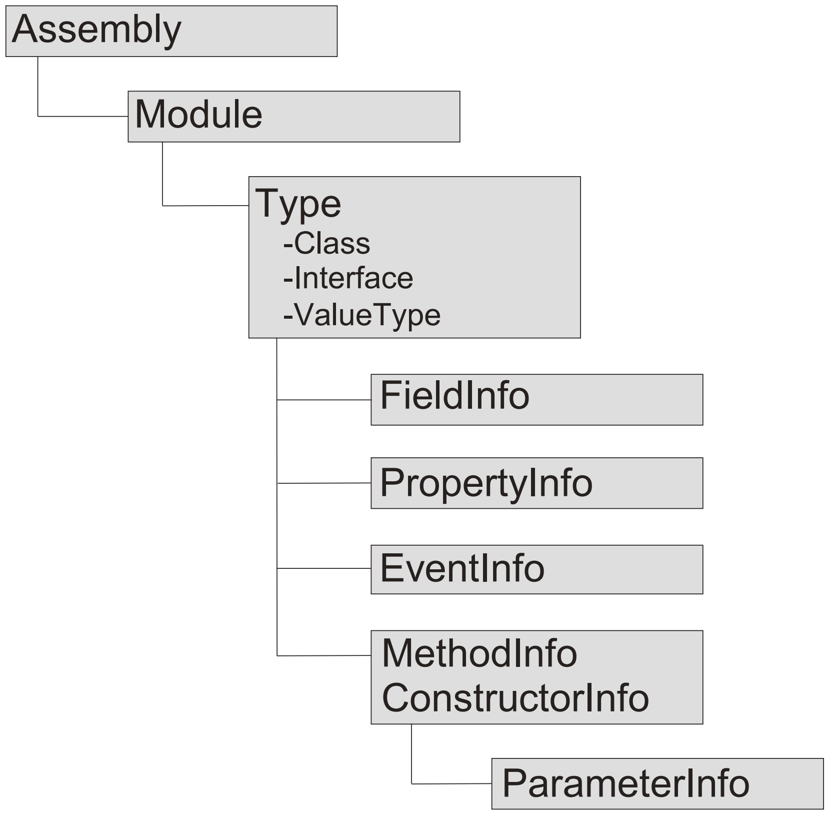


Figure 1: The Metadata Hierarchy of .NET

has as return value an object of the type `Type`. This type is defined in the namespace `System`. Every type-instance represents one of three possible definitions:

- a class definition
- an interface definition
- a value-class (usually a structure)

Via reflection, one may ask about almost every type attribute, including the type's access-modifier, whether it is a nested type and about the type's properties. Metadata information is structured in a hierarchical fashion. At the highest level stands the class `System.Reflection.Assembly`. An assembly object corresponds to one or more dynamic libraries (DLLs) from which the .NET unit in question is composed. As depicted in Figure 1, class `System.Reflection.Module` stands on the next lower level of the metadata hierarchy. A module represents a single DLL. This module class accepts inquiries about the types the module contains. Proceeding further down the metadata hierarchy reveals type information for any of the building blocks making up a member of the .NET virtual object system.

1.2 Toward a solution

There are two things what a aspect definition should provide. The first are *interceptions* and the second are *introductions*.

Interceptions means that aspect code become active within the control flow of the component. The aspect defines additional actions which should be done on an access to the component. This can happen during a field access, setting or getting a property, executing a method, creating a new object etc.

Introduction on the other hand means that we modify/extend the definition of the component. This is when the aspect defines a new interface or attributes on the component.

With an aspect definition language you should be able to describe which points of the component you want to intercept, and which additional definitions you want to introduce in you component.

An aspect definition in LOOM .NET is a set of weaving rules. A rule is defined by a symbol and a substitution. The substitution can contain further symbols. The starting point are the reflection information of the assembly. Each type of reflection object has a corresponding rule. For instance the corresponding rule for a method reflection object is a rule defined by the symbol $\langle METHOD \rangle$. The substitution for such rule is a template which defines how the method in the proxy will generated. The aspect definition can contain rules for:

- $\langle NAMESPACE \rangle$, the namespace
- $\langle CLASS \rangle$, the class
- $\langle METHOD \rangle$, $\langle CTOR \rangle$, $\langle FIELD \rangle$, $\langle PROPERTY \rangle$, the method, constructor, field, and property templates

Similar to the metadata hierarchy in Figure 1 a namespace contains classes and a class contains members (fields, constructors, methods, and properties). Such containing entities will expressed by special symbols:

- $\langle CLASSDEFINITION \rangle$ this rule defines the substitution for all classes in the namespace (usually used in the $\langle NAMESPACE \rangle$ template
- $\langle MEMBERDEFINITION \rangle$ this symbol will substituted to all members in the class (usually used in the $\langle CLASS \rangle$ template)

A typical template for $\langle CLASS \rangle$ would look like the following:

```
/* [CLASSPROTECTION] */ class /* [CLASSNAME] */ : /* [
    BASECLASS] */
{
/* [MEMBERDEFINITION] */
}
```

This rule says nothing more than the following: For every assigned class build a new class, derived from this class and if there are rules for members

level	symbol	description
$\langle CLASS\rangle$	$\langle CLASSNAME\rangle$	the class name
	$\langle BASENAMESPACE\rangle$	the namespace name of the component class
	$\langle CLASSPROTECTION\rangle$	the modifier of the class
$\langle CTOR\rangle$	$\langle PROTECTION\rangle$	the modifier of the constructor
	$\langle PARAMDECLARATION\rangle$	the argument declaration of the constructor
$\langle METHOD\rangle$	$\langle METHODNAME\rangle$	the method name
	$\langle PARAMDECLARATION\rangle$	the argument declaration of the method
	$\langle MODIFIER\rangle$	the modifier of the method
	$\langle RESULTTYPE\rangle$	the result type of the method
$\langle FIELD\rangle$	$\langle TYPE\rangle$	the type of the field
	$\langle NAME\rangle$	the name of the field

Figure 1: reflection symbols

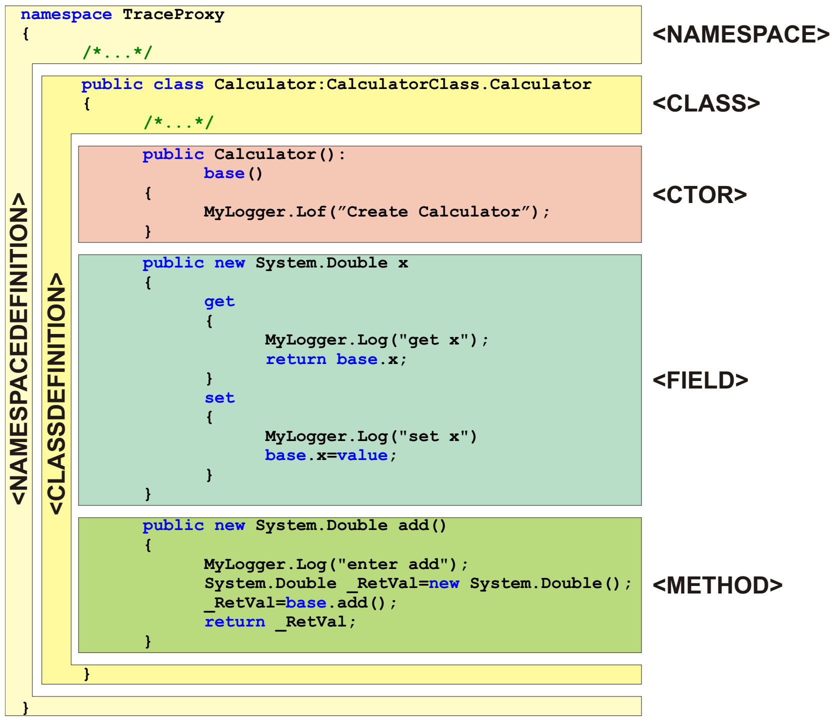
defined insert substitute these in the enclosing braces. This example shows also a second point. Here one can implement the introductions for the class.

But also interceptions are possible. The following $\langle METHOD\rangle$ rule gives an idea :

```
public /*[MODIFIER]*/ /*[RESULTTYPE]*/ /*[METHODNAME
    ]*/(/*[PARAMDECLARATION]*/)
{
  MyLogger.Log("enter␣/*[METHODNAME]*/");
  /*[RETVALINIT]*/
  /*[RETVALASSIGN]*/base./*[METHODNAME]*/(/*[
      PARAMLIST]*/);
  /*[RETVALRETURN]*/
}
```

The generated code will call the function *MyLogger.Log* and execute the original implementation. In both templates one can see further symbols enclosed in `/*[` and `]*/`. These rules will defined by the weaver depending on the associated reflection object. I.e. the Symbol $\langle METHODNAME\rangle$ is associated with a rule which replaces the name of the method. Figure 1 shows some of these reflection symbols.

Figure 1.2 shows an example for a generated proxy. On the right site one can see which rules are responsible for which parts of the generated



```
namespace TraceProxy
{
        /*...*/
        public class Calculator:CalculatorClass.Calculator
        {
                /*...*/
                public Calculator():
                        base()
                {
                        MyLogger.Lof(”Create Calculator”);
                }
                public new System.Double x
                {
                        get
                        {
                                MyLogger.Log("get x");
                                return base.x;
                        }
                        set
                        {
                                MyLogger.Log("set x")
                                base.x=value;
                        }
                }
                public new System.Double add()
                {
                        MyLogger.Log("enter add");
                        System.Double _RetVal=new System.Double();
                        _RetVal=base.add();
                        return _RetVal;
                }
        }
}
```

code. In case of the previous example for $\langle METHOD \rangle$ all reflection information are substituted.

The rules of an aspect are defined in a xml file. There exists a section for every defined aspect. This section contains the rules defined for the aspect.

```
<Aspect Name="TestAspects.Log">
  <Tag Type="file" Name="NAMESPACE">namespace_Log.
      tpl</Tag>
  <Tag Type="file" Name="CLASS">method_Log.tpl</Tag>
  <Tag Type="file" Name="METHOD">method_Log.tpl</Tag
      >
  <Tag Type="file" Name="CTOR">ctor_Log.tpl</Tag>
  <Tag Type="file" Name="FIELD">field_Log.tpl</Tag>
</Aspect>
```

Every rule is defined in a `Tag`. For instance is there a rule for $\langle NAMESPACE \rangle$ defined. The template for this rule is stored in the file *namespace_Log.tpl*.

2 Usage example : A simple checkpointing mechanism

The aspect-oriented programming technique available through LOOM.NET allows the proper integration of crosscutting functional and non-functional aspects into .NET based software systems. The following section explains an exemplary application, which shows the advantages in using .NET technologies in conjunction with aspect-oriented programing techniques.

In the research area on reliable computing one of the relevant techniques is *checkpointing*. In the context of transaction mechanisms it refers to the saving of an application state snapshot to a secondary storage in order to decrease the needed recovery time in case of a transaction rollback [?]. In the context of object based applications a checkpoint is defined as a designated place in a program at which normal processing is interrupted specifically to preserve the status information necessary to allow resumption of processing at a later time. Checkpointing is the process of saving the status information [?]. With the help of such a mechanism we can increase the fault tolerance of an application - the state is saved periodically and restored to the last consistent state in case of a crash fault. The implementation shown here acts only as a proof-of-concept scenario without taking care of the relevant consistency factors relevant in such an approach.

In the context of passive data objects the process of object state saving is well-known as *serialization*. In our earlier work we showed that the serialization mechanisms of .NET is (under some preconditions) powerful enough to allow the state saving of actively executed objects. With the help of this functionality we have implemented a framework for the transparent migration of active application during runtime in a distributed .NET environment. This work was presented on the last WORDS 2003 conference in Guadalajara, Mexico [?]. Based on this idea of active object serialization it is also possible to design an exemplary checkpointing functionality.

The serialization functionality of the .NET framework is one of the major features needed for the approach shown here. The .NET environment offers two serializing technologies: XML serialization is able to save the values of public fields and properties. Type information of the members are not saved by this serializer. This makes the XML serialization unusable in the conceptual scope of this work, because the restoring of an active object instance needs this kind of information. The second possibility is the binary serialization. Here all relevant informations including private, static and type data are saved. Because of this extended functionality it is the chosen serializer for the data encoding. Within its framework Microsoft allows the customization of the complete serialization procedure. The main idea behind this concept is to provide a possibility for the custom handling of member values that cannot be deserialized correctly. In practice the customization is done by implementing the ISerializable interface. This capability of the .NET serialization implementation enables the proper handling of runtime state informations from active objects.

At first we have implemented a very basic persistence aspect for LOOM.NET.

This aspect integrates a mechanisms for the persistent storage of application state to a secondary storage and can be applied to an existing binary program without the need for source code. The solution works on default unmodified .NET framework installations, but the non-intrusiveness approach comes for the price of only being usable for a single-threaded applications.

One of the main interesting topics in realizing such a functionality is the time when the saving of the object state information occurs. Our persistent aspect implementation serializes the execution object before application end (in the Finalizer method) and restores the state, if available on the storage device, before the application starts in the constructor code (see figure 2). The deserialization process can only create a new object and not overwrite the state information of the actual execution object. This leads to a new member variable that contains the object in the restored state. Because of this all subsequent calls must be redirected to the deserialized object instance. Normally this leads to overall application-wide changes in the source code, but with the help of aspect-oriented programing we can easily utilize a simple method template as shown in figure 3.

```
/*[PROTECTION]*/ /*[CLASSNAME]*/(/*[PARAMDECLARATION]*/) :
    base(/*[PARAMLIST]*/)
{
  System.IO.Stream _stream;

  try
  {
    _stream = System.IO.File.Open("serialized.bin",System.IO
        .FileMode.Open);
  }
  catch (Exception)
  {
    // seems like there is no serialization data
    return;
  }
  System.Runtime.Serialization.Formatters.Binary.
      BinaryFormatter _formatter = new System.Runtime.
      Serialization.Formatters.Binary.BinaryFormatter();
  this_deserialized = (/*[CLASSNAME]*/)_formatter.
      Deserialize(_stream);
  _stream.Close();
}
```

Figure 2: LOOM.NET constructor template for persistency aspect

After realization of the persistency aspect one could extend the functionality towards a simple checkpointing approach. In this context checkpointing can be seen as the consistent creation of persistency information at specific points during runtime. This leads to a new problem: It must

```
/*[PROTECTION]*/ /*[CLASSNAME]*/(/*[PARAMDECLARATION]*/) :
    base(/*[PARAMLIST]*/)
{
public /*[MODIFIER]*/ /*[RESULTTYPE]*/ /*[METHODNAME]*/(/*[
    PARAMDECLARATION]*/)
{
  /*[RETVALINIT]*/

  if (this_deserialized != null)
  {
    /*[RETVALASSIGN]*/this_deserialized./*[METHODNAME]*/(/*[
        PARAMLIST]*/);
    /*[RETVALRETURN]*/
  }
  else
  {
    /*[RETVALASSIGN]*/base./*[METHODNAME]*/(/*[PARAMLIST]*/)
        ;
    /*[RETVALRETURN]*/
  }
}
}
```

Figure 3: LOOM.NET method template for persistency aspect

be ensured that there is no change in state informations while the checkpointing information is created. Since we already have the prerequisite of a single-threaded application this can be reduced to the fact that there should not be any function call during the serialization process. We solved this problem by wrapping a semaphore around each function call, again with the help of LOOM.NET. The checkpointing code is running in an extra thread, waiting for the checkpointing condition. If it occures the implementation waits until it gets the semaphore, which ensures that no other function call is in progress. It can be easily seen that this kind of interruption relies on an event-driven application model. After successful interruption the checkpointing code trigggers the serialization and finally releases the semaphore to allow the continuation of the application. The triggering of the checkpoint code could simply rely on a time interval or system performance parameters. With the help of AOP and reflection mechanisms it is also possible to investigate the applications state to find a appropriate point in time for checkpointing.

The aspect code integrates a cleanup functionality that removes the state informations in case of a successful application termination. In case of a crash fault the state information resist on the storage device. Within the next application start the data is recognized by the checkpointing code, which leads to the restauration of the last checkpointing state as described in the persistency case.

The simple example shown here does not represent an exhaustive checkpointing implementation. The influence of external events between the checkpointing as well as consistency problems are not covered in detail in this proof-of-concept scenario. However it shows the elegance of solutions based on a combination of existing powerful .NET technologies with an aspect weaving toolkit.