Dynamic Weaving with .net

Wolfgang Schult and Andreas Polze
Hasso-Plattner-Institute
14440 Potsdam, Germany
{schult|apolze}@informatik.hu-berlin.de

March 1, 2002

Abstract

Aspect-oriented programming (AOP) is a relatively new approach for separation of concerns in software development. AOP makes it possible to modularize crosscutting aspects of a system. Like objects, aspects may arise at any stage of the software lifecycle, including requirements specification, design, implementation, configuration, and even runtime. Aspects often constrain the design space for a given software component. Especially if multiple aspects are applied to a component, this may have severe implications. Components may be used in different contexts, may be requiring emphasis on only a few of the aspects considered during design and implementation. Some aspects may impose contradicting requirements on a component (e.g.; the fault-tolerance aspect may require replication of data, the security aspect may prohibit it). Static interconnection of aspect code and functional code (aspect weaving) often requires compromises with respect to generality of services provided by a component.

Within this paper we focus on dynamic management of aspect-information at program runtime. We introduce a new approach called "dynamic weaving" to interconnect aspect code and functional code. Using our approch, it is possible to decide at runtime whether a component should be instantiated with support for a particular aspect or not. We have implemented our approach in context of the language C# and the .NET environment recently introduced by Microsoft.

1 Introduction

There exists a variety of application areas for Aspect-Oriented Programming (AOP). Generally, it is very acceptable to have a preprocessor-like aspect-weaver to interconnect functional code and aspect code. However, sometimes it is desirable to postpone the decision whether aspect information is added to a particular component until program runtime. For instance, one may have a huge resource consuming image processing algorithm located in a component, and depending on system load and available computing nodes one may choose data distribution, memory allocation scheme, and utilization of computing power at runtime. Perhaps you want to distribute the calculations for better performance or you want to optimize local memory usage. Both are crosscutting concerns, one may define an aspect which distributes invocation of the components' functions calls and another aspect which optimizes local and remote memory utilization during a distributed computation. Figure 1 illustrates the situation for a distributed computation. *Eagle* gets a request for a service in the component. Depending on its own utilization, the decision is to delegate it to the neighbors (*Tomcat* and *Raptor*), or to execute the service locally. However, in the case of the local computation, no aspect information at all is needed. Emphasis is rather on service execution with as little overhead as possible.

The same with our second aspect. If we do not want to restrict memory usage, we do not need any

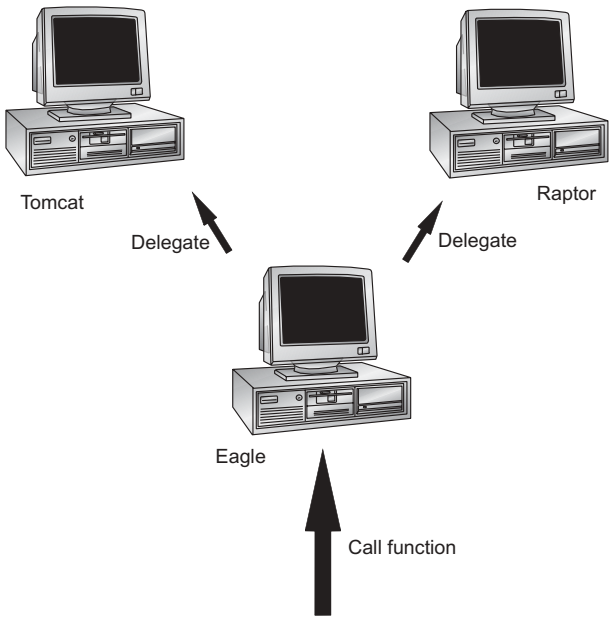

Figure 1: Distributing Calculations

overhead. At this point, our example identifies a weakness of traditional approaches to aspect oriented programming. Typically, one has to decide at compile time decide whether an aspect should be interwoven with a component or not. At the runtime you neither can 'switch off' your aspect nor interweave another aspect with the component.

Within this paper, we present a solution to this problem and show one can interweave previously defined aspects with functional components code dynamically, during the runtime. This 'Dynamic Weaving' is promising because of its flexibility: neither at design nor at compilation time a definite decision whether a particular aspect should be reflected in a component has to be made. You can define aspects specialized for a particular situation and interweave them if they are needed. Furthermore one can parameterize the aspects during the runtime. And we will discuss that all is done without the need of any tool or language support.

The remainder of the paper is organized as follows: Section 2 presents related work. Section 3 describes the dynamic weaving. In Section 4 we demonstrate a simple case study with the sample described above and finally, Section 5 summarizes our conclusions.

2 Related Work

The concept of aspect-oriented programming (AOP) offers an interesting alternative for specification of non-functional component properties (such as fault-tolerance properties or timing behavior). There exists a variety of language extensions with AspectJ [8], which is a Java extension as a most prominent example. The central concept of most AOP-frameworks is a joinpoint model described in [9].

Dynamic joinpoints are an extension of the original AOP model to allow dealing with dynamic informations during the runtime [5]. A dynamic joinpoint allows to define conditions wich are compared during the runtime. Depending on the result the code will executed or not.

Mehmet Aksit has developed the composition filters object model, wich provides control over messages recived and sent by an object wich provides control over messages recived and sent by an object [1]. In their work, the component language follows traditional object-oriented programming techniques, the composition filters mechanism provides an aspect language that can be used to control a number of aspects including syncronization and communication. Most of the weaving happens runtime.

3 Dynamic Weaving

Dynamic weaving means that a component (a *target class*) and an *aspect class* will become interwoven during the runtime. There is no need for the aspect class to know something about the target class and vice versa. To understand how the weaving process works, we have to define some notions.

3.1 What is an Aspect Class?

An aspect describes crosscutting concerns. In our case an aspect is a simple class derived from **Aspect**. We will call it *aspect class*. One can define methods, properties, and members as well. In every case an

aspect class works in conjunction with another instance of a class (the *target class*). This means, that it makes no sense to instanciate an aspect class alone. You have to instanciate it together with a class. This process is called *weaving*. We will describe it later in this section.

3.2 Connection Points

As we said an AspectClass works only in conjunction with another instance of a class. At a *connection point* both will become interwoven. At this time we want only define a method as a connection point. To do this, you simply write the **call** attribute above the method definition in your aspect class. The call attribute is defined as follows:

```
[call(Invoke InvokeOrder{, Alias=AliasName})]
```

If you interweave a class (target class) with an AspectClass each connection point will become interwoven with a target class method if:

1. The method names and the signature are the same
2. If there is an *AliasName* defined and the method name from the target class is the same like the alias and - the signature of both are the same
3. If there is an *AliasName* and the alias contains a wildcard at the end, or the signature of the Aspect class method contains wildcards and the target method fit.

In any case, if a function is interwoven with a connection point. Point 1 is easy, if one define a method:

```
[call(Invoke.Instead)]
void mymethod(int i) { /* ... */ }
```

then every method **mymethod** with one **int** as parameter and void as result will interweave with this method. Now, point 2 is if one defines **Alias="myspecialmethod"** on this method, only methods named **myspecialmethod** with an **int** parameter and a **void** return value will become interwoven.

And the last point is if one modifies the alias to **Alias="my*"** every method beginning with "my" and the same parameters will become involved. Furthermore one can use *signature wildcards*. A wildcard for the result type is **object**, and for the parameters **params object**[]. This is like a method with variable arguments. But in every case you have to define an alias. If not **params object**[] will not be handled as wildcard. I.e. the following connection point:

```
[call(Invoke.Instead, Alias="*")]
object catchall(params object[] args)
```

will become interwoven with every method in the target class and *args* will contain each parameter, you pass through the function. For instance, if the target class has a method **void f(int i, double d)**, then *args[0]* will contain *i* and *args[1]* will contain *d* after the function is called.

We have seen when a connection point will interweave, now focus on how we interweave. This is described by the *InvokeOrder* parameter of the call attribute. There are three possibilities:

- **Invoke.Before**: The aspect function of the connection point will invoke *before* the object function will call.
- **Invoke.After**: As to be expected, the aspect function will invoke *after* the object function has already called.
- **Invoke.Instead**: The object function will not call automatically. The aspect function has to do it.

The first case is useful if you want to trace method calls only. If you need to change the return value of the method, you should choose the second case. To get full control over the method, you need the last.

3.3 Aspect Context

In the case that you define an *Invoke.Instead* connection point, you need a mechanism to call the appropriate target class method. The problem is that you know neither the type of the target class (your aspect can become interwoven with any type) nor, in some cases, the signature of the called method (this is when you use signature wildcards). The solution is to define an **Context** property in the *Aspect* base class. With this property you get an object of type

AspectContext wich has the needed informations. We define two methods:

```
public object Invoke(params object[] parameters)

public object InvokeOn(object target, params
    object[] parameters)
```

The first simply invokes the target class method with the given parameters. With the second, one can invoke one's own instance (*target*) of the target class. This is useful if you have special instances of the target class stored in your aspect, and you want to invoke these.

3.4 Implementation Issues

In the sections above we learned what an aspect class is, how we define connection points, and what object context means. The question is how to implement it. We need a language which has the following requirements:

- a way to define attributes
- reflection and retrospection to analyse the target class and the aspect class signature (this means methods and method parameters)
- last, but not least, a possibility to emit the interwoven class

We implemented our solution in Microsoft .NET because it fullfills all these requirements. MS .NET is a framework like Java wich provides a runtime enviroment to run a system independent code. This code is present in an intermediate language (IL). Unlike java, .NET has the capability of working with a variety of languages. So we have the big advantage that we get the ability to interweave an aspect written in C++, with a component written in pascal.

Now our solution is a library for .NET. This library provides several classes and attributes defined in the namespace **Aspects**:

- **Aspect** is the base class for all defined aspects
- **Weaver** is a class wich includes the weaving functionality
- **Call** is an attribute to define connection points.

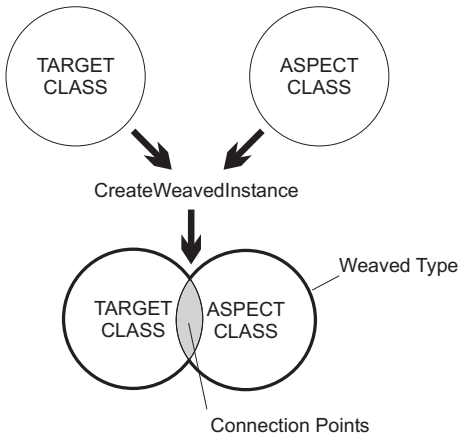

Figure 2: The Weaving Process

- **AspectContext** accessible via the *Aspect.Instance*, to invoke instance methods.

3.5 Dynamic vs. Static Weaving

Most aspect frameworks use a compiler (aspect weaver) approach. This is fine as long as all system parameters are well known at compile time. Dynamic weaving describes a process where a class will become interwoven with an aspect class during the runtime.

3.6 The Dynamic Aspect Weaver

As described above, the **Aspects** namespace contains a class called **Weaver**. It provides two functions with which to interweave an AspectClass with a specified class:

```
static object Weaver.CreateWeavedInstance(
  Type classtype,
  Aspect aspect,
  params object[] parameters)

static object Weaver.CreateInstance(
  Type classtype,
  params object[] parameters)
```

The first function generates an instance of a class *classtype*. In *aspect* you have to commit an instance of your AspectClass, *parameters* are the constuctor parameters for the target class. A possible call would be:

```
A a=Weaver.CreateWeavedInstance(typeof(A), new
    MyAspect(), ...) as A;
```

In the second function there is no aspect. This is when you define the aspect as attribute. The following lines have the same meaning as the sample above:

```
[MyAspect]
class A
{ /* ... */ }
/* ... */
A a=Weaver.CreateWeavedInstance(typeof(A), ...)
    as A;
```

The first way is more flexible. One can determine the Aspect and its parameters during runtime. First the weaver looks for a custom attribute derived from **Aspect**. If there is no aspect, the call is the same as **new A([parameters])**. Otherwise, **CreateWeavedInstance** with this aspect, will be called. What happens during the creation is illustrated in figure 2. The weaver looks for connection points and tries to join them with the target class as described above. With this information, it builds a new type, and creates a new instance of this type. At the end the method *Aspect.ctor* will called. This method is overridable an has the following form:

```
virtual void ctor(Type classtype, object
    target, params object[] args)
```

- *classtype* is the type of the target class
- *target* is the new interwoven instance
- *args* are the constructor parameters

After that, the newly built instance will be returned to the caller.

4 An Example

Going back to the situation in our introduction, listing 1 shows a class which calculates a Mandelbrot. This is an image processing algorithm developed by Benoit Mandelbrot [12]. The input for the algorithm is a filename, a bounding box, and the resolution.

```
public class Mandelbrot
{
  const int m_iLimit=255; // calculation limit
  public Mandelbrot(){}
```

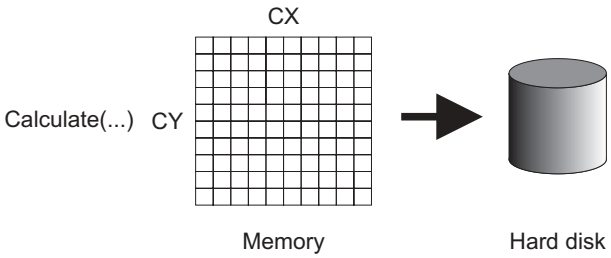

Figure 3: Mandelbrot Function Call

```
// this method calculates the mandelbrot and returns the
// result in matrix
private void InternalCalculate(double x1, double y1, double
      dAddx, double dAddy, int line, ref Byte[] matrix)
{
  int iPos=0;
  while(iPos<matrix.Length)
  {
    double dCr=x1;
    for(int iPosLine=0;iPosLine<line;iPosLine++)
    {
      Byte c=0;
      double
        dZr = 0.0,          // real component of Z
        dZi = 0.0,          // imaginary component of Z
        dZiSqr = 0.0,       // Zi squared
        dZrSqr = 0.0,       // Zr squared
        dZr1;               // temporary holder for Zr
      while (c < m_iLimit && dZiSqr + dZrSqr < 4)
      {
        dZr1 = dZrSqr - dZiSqr + dCr;
        dZi = 2 * dZr * dZi + y1;
        dZr = dZr1;
        dZiSqr = dZi * dZi;
        dZrSqr = dZr * dZr;
        ++c;
      }
      if (c >= m_iLimit)
        matrix[iPos]=0;
      else
        matrix[iPos]=c;

      dCr+=dAddx;
      iPos++;
    }
    y1+=dAddy;
  }
}
// only this method is accessible from outside
// It calls the InternalCalculate function and
// stores the result to the hard disk
public virtual void Calculate(string filename, double x1,
     double y1, double x2, double y2, int cx, int cy)
{
  double dAddx=(x2-x1)/((double)cx);
  double dAddy=(y2-y1)/((double)cy);
  // memory allocation and calculate
  Byte[] matrix=new Byte[cy*cx];
  Calculate(x1,y1,dAddx,dAddy,xRes,ref matrix);
  // store the result
```

```
    FileStream fs=new FileStream(filename, FileMode.Create,
         FileAccess.Write);
    fs.Write(matrix,0,matrix.Length);
    fs.Close();
  }
}
```

Listing 1: The Mandelbrot Class

Figure 3 shows what happens: The algorithm first calculates the whole mandel set in memory and then it stores it to the hard disk. For small resolutions this algorithm works well. But what happens if we increase the resolution? The amount of consuming memory will increase exponential (We need $cx*cy$ memory storage). A possible solution is to rewrite the algorithm. Under certain circumstances, we don't have the possibility to do that (i.e. we have the algorithm only as binary), so we need another solution.

4.1 The Save Memory Aspect

We split the function calls so that only single lines will be written to the hard disk. After that we can join these files together to the requested file. Figure 2 shows this approach. This can be done by an aspect class (we want to leave it transperent to the client). Listing 2 shows a possible implementation of this aspect.

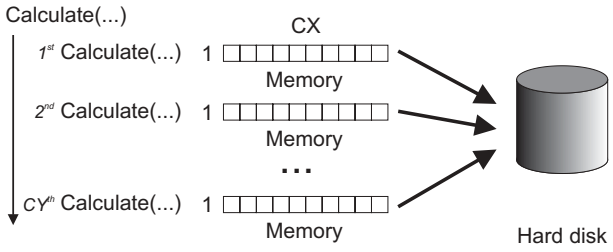

Figure 4: Function Call with the SaveMemory Aspect

```
public class SaveMemory:Aspect
{
  [Call(Invoke.Instead)] // connection point
  public void Calculate(string filename, double x1, double y1
       , double x2, double y2, int xRes, int yRes)
  {
    // split up in lines
    double dStep=(y2-y1)/((double)yRes);
    for(int i=0;i<yRes;i++)
    {
      // call original function
      Context.Invoke(filename+i.ToString(),x1,y1,x2,y1,xRes,1)
           ;
      y1+=dStep;
    }
    // join the files together
    Byte[] data=new Byte[xRes];
    FileStream fsdst=new FileStream(filename, FileMode.Create
         , FileAccess.Write);
    for(int i=0;i<yRes;i++)
    {
      FileStream fssrc=new FileStream(filename+i.ToString(),
           FileMode.Open, FileAccess.Read);
      fssrc.Read(data,0,data.Length);
      fssrc.Close();
      fsdst.Write(data,0,data.Length);
    }
    fsdst.Close();
  }
}
```

Listing 2: The Save Memory Aspect

As you see in the aspect class the function *calculate* is defined as a connection point. As described in Section 3, if the target class contains a function *Calculate* with the same signature (and in this case it has) then both will become interwoven. The for loop simply invokes via the Aspect Context our algorithm line by line. For n lines it will generate n files on the hard disk. At the end, these n files will become joined to a new file which was originally requested.

4.2 The Distributing Aspect

Our second goal was to distribute the function calls on several computers. For that problem too, one can define an aspect. Figure 5 shows what we have to do: On every function call we split the calculation up and delegate each part to our computers *Eagle* and *Tomcat*[1]. Both write the result to central location (a file Server). The aspect class now gets the result files and joins them together. Listing 3 shows an extract.

```
public class Distributing:Aspect
{
  // instances on remote computers
  private object eagle;
  private object tomcat;

  public override void ctor(Type typ, object o, params object
       [] args)
  {
    /* Create remote instances for Eagle and Tomcat */
  }
  // the connection point
  [Call(Invoke.Instead, Alias="Calculate")]
  public void Distributing(string filename, double x1, double
       y1, double x2, double y2, int xRes, int yRes)
```

[1] At this Point both computers are hard coded in our aspect class. But it is easy to extend the algorithm to more computers, and dynamic assigned computers. We only want to show the principle.

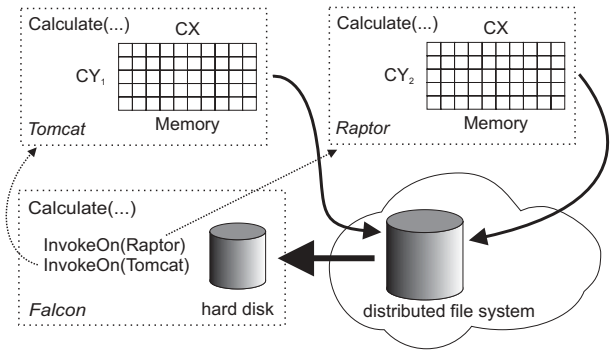

Figure 5: Function Call with the Distributing Aspect

```
{
  // calculate boundaries for both computers
  int yRes2=yRes/2;
  double yStep=(y2-y1)/((double)yRes);
  double y12=y1+yStep*yRes2;
  double y21=y12+yStep;
  // Prepare event for async call
  AutoResetEvent ev=new AutoResetEvent(false);
  workcount=2;
  // Queue function calls
  System.Threading.ThreadPool.QueueUserWorkItem(
    new WaitCallback(Distributing.Calculate),
    new WorkItem(this, ev, eagle,temppath+"/eagle.raw",x1,y1
        ,x2,y12,xRes,yRes2));
  System.Threading.ThreadPool.QueueUserWorkItem(
    new WaitCallback(Distributing.Claculate),
    new WorkItem(this, ev, tomcat,temppath+"/tomcat.raw",x1,
        y21,x2,y2,xRes,yRes-yRes2));
  // wait until ready
  while(workcount!=0) ev.WaitOne();
  // join files together
  FileStream fsdst=new FileStream(filename, FileMode.Create
      , FileAccess.Write);
  Copy(temppath+"/eagle.raw",fsdst,xRes,yRes2);
  Copy(temppath+"/tomcat.raw",fsdst,xRes,yRes-yRes2);
  fsdst.Close();
}
/* ... */
public static void Calculate(object para)
{
  WorkItem item=(WorkItem)para;
  item.aspect.Context.InvokeOn( item.target, item.filename,
      item.x1, item.y1, item.x2, item.y2, item.xRes, item.
      yRes );
  // ready
  item.aspect.workcount--;
  item.readyevent.Set();
}
}
```

Listing 3: The Distributing Aspect

Our aspect class contains three important functions. The first is **ctor**, which will be called from the Weaver when the instance is created. We use it to create further instances of the same type on which we can distribute the function calls. The second is **Distributing**. This method contains the **call** attribute, which defines it as connection point as well. Here we distribute the function calls to the instances at the computers tomcat and eagle. For that we generate a previously defined **WorkItem** and put it in a thread pool. The asynchronous callback will happen in **Calculate** where we invoke the target class.

4.3 The Client Side

In the client only the instantiation of the Mandelbrot class changes. Depending on our needs we weave one of the both aspects to our class (Listing 4).

```
Mandelbrot mb;
// we need less memory usage
if(opt_memory.Checked)
  mb=Aspects.Weaver.CreateWeavedInstance(typeof(Mandelbrot),
       new SaveMemory()) as Mandelbrot;
// we more performance
else if(opt_speed.Checked)
  mb=Aspects.Weaver.CreateWeavedInstance(typeof(Mandelbrot),
       new Distributing("d:/temp")) as Mandelbrot;
// we need nothing of both
else mb=new Mandelbrot();
```

Listing 4: The Client Side

The function call itself are the same.

5 Conclusions

Aspect-oriented programming (AOP) is a relatively new approach for separation of concerns in software development. AOP makes it possible to modularize crWithin this paper we focus on dynamic management of aspect-information at program runtime. We introduce a new approach called "dynamic weaving" to interconnect aspect code and functional code. Using our approch, it is possible to decide at runtime whether a component should be instantiated with support for a particular aspect or not. We have implemented our approach in context of the language C# and the .NET environment recently introduced by Microsoft.osscutting aspects of a system. Generally, it is very acceptable to have a preprocessor-like aspect-weaver to interconnect functional code and aspect code. However, sometimes it is desirable to postpone the decision whether aspect information is

added to a particular component until program runtime. Within this paper we focus on dynamic management of aspect-information at program runtime. We introduce a new approach called "dynamic weaving" to interconnect aspect code and functional code. Using our approch, it is possible to decide at runtime whether a component should be instantiated with support for a particular aspect or not. We have implemented our approach in context of the language C# and the .NET environment recently introduced by Microsoft. Within this paper have presented our approach to dynamic management of aspect-information at program runtime. We have introduced a new approach called "dynamic weaving" which allows for late binding (weaving) of aspect code and functional code. Using our approch, it is possible to decide at runtime whether a component should be instantiated with support for a particular aspect or not. We have implemented our approach in context of the language C# and the .NET environment recently introduWithin this paper we focus on dynamic management of aspect-information at program runtime. We introduce a new approach called "dynamic weaving" to interconnect aspect code and functional code. Using our approch, it is possible to decide at runtime whether a component should be instantiated with support for a particular aspect or not. We have implemented our approach in context of the language C# and the .NET environment recently introduced by Microsoft.ced by Microsoft. Relying on the .NET support for a variety of programming languages, our approach is not restricted to C# but works for all of the .NET languages.

Our current implementation has puts some constraints on the programmer of a component. Currently, only virtual methods can be interwoven dynamically. The reason is our implementation of late binding of the function calls. Currently the Weaver "overrides" the function so that the virtual method table maintained inside the .NET virtual machine points to the woven function (the version enriched with aspect information). Other members of a class, such as fields, properties, static, and class functions currently cannot be accessed this way. However, we are working on a solution to lift this restriction.

References

[1] Mehmet Aksit, Bedir Tekinerdogan, "Aspect-Oriented Programming Using Composition-Filters", ECOOP Workshops 1998: 435.

[2] T. Archer, "Inside Microsoft C#", ISBN 0-7356-1288-9, Microsoft Press.

[3] AspectJ Hompage, http://www.aspectj.org/, 2002

[4] "Common Language Infrastructure", Microsoft, Internal Working Document.

[5] Kris Gybels, "Using a logic language to express cross-cutting through dynamic joinpoints", In proceedings of Second Workshop on Aspect-Oriented Software Development, Bonn, February 21-22, 2002.

[6] S. Hanenberg, Rainer Unland, "A Proposal For Classifying Tangeled Code", In proceedings of Second Workshop on Aspect-Oriented Software Development, Bonn, February 21-22, 2002.

[7] S. Hanenberg, R. Unland, "Concerning AOP and Inheritance", Dept. of Mathematics and Computer Science University of Essen.

[8] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Christina Videira Lopes, Jean-Marc Loingtier, John Irwin, "Aspect Oriented Programming", In proceedings of the European Conference on Object-Oriented Programming (ECOOP), Finnland, Springer Verlag LNCS 1241; June 1997.

[9] G.Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. G. Griswold, "Getting Started with AspectJ", Communications of the ACM, Vol. 44, Issue 10, October 2001, pp. 59-65.

[10] K.Lieberherr, D. Orleans, J. Ovlinger; "Aspect-Oriented Programming with Adaptive Methods", Communications of the ACM, Vol. 44, Issue 10, Oktober 2001, pp. 39-41

[11] C. V. Lopes, G. Kiczales, "Recent Developments in AspectJ", Xerox Palo Alto Research Center.

[12] B. Mandelbrot, "The Fractal Geometry of Nature", San Francisco: Freeman, 1982.

[13] M. Pietrek, http://msdn.microsoft.com/ msdnmag/issues/1000/metadata/metadata.asp.

[14] J. Richter, D. Box, several articles about .NET; SYSTEM-Journal 02/2001 to 05/2001, redtec publishing, Unterschleiheim, Germany.

[15] Workshop "Microsoft .net Crash Course for Faculty and PhDs", Microsoft Research, Cambridge, England, September 3-6, 2001.