Dynamic Aspect-Weaving with .NET

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

September 12, 2002

Abstract

Besides design and implementation of components, software engineering for component-based systems has to deal with component integration issues whose impact is not restricted to separate components but rather affects the system as a whole. Aspect-oriented programming (AOP) addresses those cross-cutting, multi-component concerns. AOP describes system properties and component interactions in terms of so-called aspects. Often, aspects express non-functional component properties, such as resource usage (CPU, memory, network bandwidth), component and object (co-) locations, fault-tolerance, timing behavior, or security settings. Typically, these properties do not manifest in the components' functional interfaces.

Like objects, aspects can be used at any stage of the software lifecycle, including requirements specification, design, implementation, configuration, and even run-time. Applicable aspects often constrain the design space for a given software components. This may have severe implications on the implementation of a component, especially if tradeoffs between multiple, possibly contradicting aspects for same component have to be made (e.g.; the fault-tolerance aspect may require replication of component data, whereas the security aspect may prohibit it).

Components may be deployed in different contexts, may be requiring emphasis on only a few of the aspects considered during design and implementation. Static interconnection of aspect code and functional code (aspect weaving) often requires compromises with respect to the the generality of services provided by a component.

Within this paper, we focus on dynamic management of aspect information during program runtime. We introduce a new approach called "dynamic aspect weaving" to interconnect aspect code and functional code. Using our approach, it is possible to decide at runtime whether objects living inside 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 Microsoft .NET environment.

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 about whether aspect information is to be added or not 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 a trade-off between data distribution, memory allocation scheme, and utilization of computing power at runtime has to be made and perhaps one wants to distribute the calculations for better performance or one wants to optimize local memory usage. Both are crosscutting concerns. One may

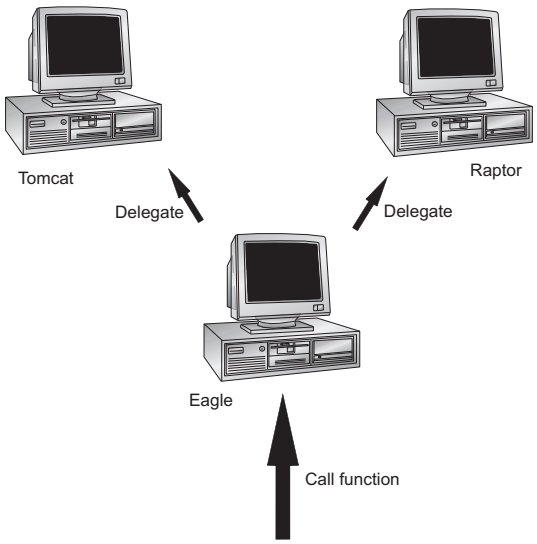

Figure 1: Distributing Calculations

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 of a distributed computation. *Eagle* gets a request for a service in the component. Depending on its own utilization, the decision is whether to delegate it to the neighbors (*Tomcat* and *Raptor*), or to execute the service locally. However, in the case of local computation, no aspect information at all is needed. Emphasis is on service execution with as little overhead as possible.

The same with the second aspect. If memory usage is not of a concern, the aspect can safely be ignored. At this point, our example identifies a weakness of traditional approaches to aspect oriented programming. Typically, one has to decide at compile time whether an aspect should be interwoven with a component or not and at the runtime one can neither 'switch off' the aspect nor interweave another aspect with the component.

Within this paper, we present a solution to this problem and and demonstrate how to interweave previously defined aspects with functional component code. This 'Dynamic Weaving' is promising because of its flexibility: neither at design nor at compilation time a definite decision has to be made whether a particular aspect should be applied to a component. Aspects specialized for a particular situation can be defined and can be interwoven depending on actual runtime requirements. Furthermore one can parameterize the aspects during the runtime. We discuss how all is done without the need of any tool 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 to deal with AOP. One of which, AspectJ [7], a Java extension, can be cited as a prominent example. The central concept of most AOP-frameworks is a joinpoint model described in [6][4].

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

Mehmet Aksit has developed the composition filters object model, which provides control over messages received and sent by an object which provides control over messages received and sent by an object [2][1]. In this 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 synchronization and communication. Most of the weaving happens during runtime.

The authors have implemented a static aspect weaver, which uses the unmanaged metadata interfaces from .NET to interweave aspect code [11].

A similar approach towards dynamic weaving for

.NET is described in [8]. However, this solution uses the current internal debug interfaces of the .NET framework implementation to interweave aspect code during the runtime and is therefore less general and portable than our approach.

3 Dynamic Weaving

Dynamic weaving means that a component (a *target class*) and an *aspect class* will become interwoven during 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, some notions have to be defined.

3.1 What is an Aspect Class?

An aspect describes crosscutting concerns. In this case an aspect is a simple class derived from **Aspect**. It will be called *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 instantiate an aspect class alone. It has to be instantiated together with a class. This process is called *weaving*. It will be described later in this section.

3.2 Connection Points

As mentioned above, an AspectClass works only in conjunction with another instance of a class. At a *connection point* both will become interwoven. If one wants to define a method as a connection point, one simply writes the **call** attribute above the method definition in the aspect class. The call attribute is defined as follows:

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

If one interweaves a class (target class) with an AspectClass each connection point will become interwoven with a target class method if one of the following requirements are met:

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 as 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. Requirement 1 is easy, if one defines 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, requirement 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.

Requirement basically says that 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 one has 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, one passes through the method. 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 method is called.

It has been shown, when a connection point will interweave, now the focus will be on how to interweave. This is described by the *InvokeOrder* parameter of the call attribute. There are three possibilities:

- **Invoke.Before**: The aspect method of the connection point will be invoked *before* the object method will be called.

- **Invoke.After**: As to be expected, the aspect method will be invoked *after* the object method has been called.
- **Invoke.Instead**: The object method will not be called automatically. The aspect method has to do it.

The first two cases are useful if one wants to trace method calls only. The last case is to be used in order to get full control over the method.

3.3 Aspect Context

When one defines an *Invoke.Instead* connection point, one needs a mechanism to call the appropriate target class method. The problem is that neither the type of the target class (the aspect can become interwoven with any type) nor, in some cases, the signature of the called method (this is when one uses signature wildcards) are known. The solution is to define an **Context** property in the *Aspect* base class. With this property one gets an object of type **AspectContext** which has the needed information. There are two methods defined:

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

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

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 there are special instances of the target class stored in the aspect, and one wants to invoke these.

3.4 Implementation Issues

In the sections above it has been described what an aspect class is, how connection points are defined, and what object context means. The question is how to implement it. A language is needed which has the following requirements:

- a way to define attributes
- reflection to analyse the target class and the aspect class signature (this means methods and method parameters)

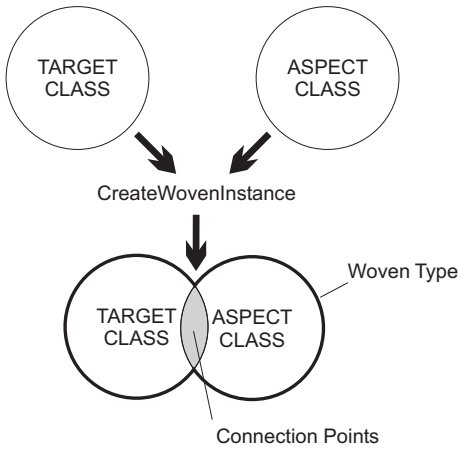

Figure 2: The Weaving Process

- last, but not least, a possibility to emit the interwoven class

We implemented our solution in Microsoft .NET because it fulfills all these requirements. MS .NET is a framework like Java which provides a runtime environment 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 it has the big advantage that one gets the ability to interweave an aspect written in C++, with a component written in pascal, for example.

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 which includes the weaving functionality
- **Call** is an attribute to define connection points.
- **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 a special function with which to interweave an AspectClass with a specified class:

```
static object Weaver.CreateInstance(
  Type classtype)

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

static object Weaver.CreateInstance(
  Type classtype,
  params object[] args,
  Aspect aspect)

static object Weaver.CreateInstance(
  Type classtype,
  params object[] args,
  Aspect[] aspectarr)
```

The first and the second version generate an instance of a class *classtype*. The objects in *arg* are the constuctor parameters for the target class. The last two versions have an additional parameter *aspect* resp. *aspectarr* where one has to commit an instance of the AspectClass(es). A possible call would be:

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

In the first two versions, there is no aspect. This is when one wants to define the aspect as attribute. The following lines have the same meaning as the sample above:

```
[MyAspect]
class A
{ /* ... */ }
/* ... */
A a=Weaver.CreateInstance(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(*args*)**. 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 and has the following form:

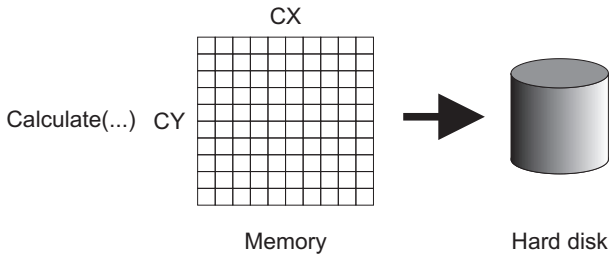

Figure 3: Mandelbrot Function Call

```
virtual void ctor(Weaver weaver, object
    target, params object[] args)
```

- *weaver* the aspect weaver itself
- *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

Now going back to the situation in the introduction, listing 1 shows a class which calculates a Mandelbrot set [9]. 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(){}
  // 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 Mandelbrot set in memory and then stores it to the hard disk. For small resolutions this algorithm works well. But what happens if the resolution is increased? The amount of consuming memory will increase polynomial (one needs $cx*cy$ memory storage). A possible solution is to rewrite the algorithm. But under certain circumstances, there is not the possibility to do that (i.e. the algorithm is only as binary available), so another solution is needed.

4.1 The Save Memory Aspect

The idea is that the function calls are split so that only single lines will be written to the hard disk. After that one can join these files together to the requested file. Figure 2 shows this approach. This can be done by an aspect class (it should be left transparent to the client). Listing 2 shows a possible implementation of this aspect.

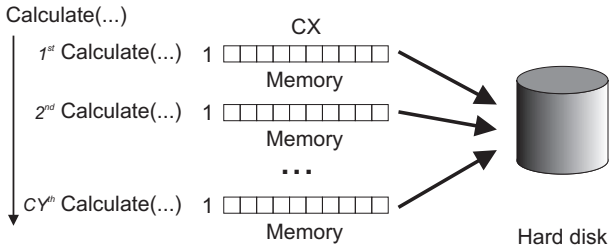

Figure 4: Function Call with the SaveMemory Aspect

```
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 one sees 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 in-

vokes, via the Aspect Context, the 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 Distribution Aspect

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

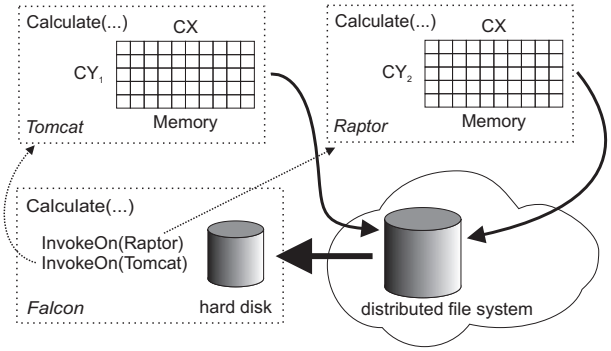

Figure 5: Function Call with the Distribution Aspect

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

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

The aspect class here contains three important functions. The first is **ctor**, which will be called from the Weaver when the instance is created. It is used to create further instances of the same type on which one can distribute the function calls. The second is **Distribution**. This method contains the **call** attribute, which defines it as connection point as well. Here the function calls are we distributed to the instances at the computers *tomcat* and *eagle*. To do that a previously defined **WorkItem** is generated and put in a thread pool. The asynchronous callback will happen in **Calculate** where the target class is invoked.

4.3 The Client Side

In the client only the instantiation of the Mandelbrot class changes. Depending on what is needed one of both of the aspects will become interwoven to the class (Listing 4). The function call itself does not change.

[1] At this point both computer names are hard coded in our aspect class. However, the algorithm sketched out here can easily be extended to use dynamically assigned computers. In fact, this would be an example of the next aspect describing system configuration.

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

Listing 4: The Client Side

5 Conclusions

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.

We 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 approach, 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. 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 some constraints for the programmer of a component. Currently, only virtual methods can be interwoven dynamically. The reason for this lies in 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, recursively applying the AOP techniques described here and in [11], it is a simple task to generate proxy classes which substitute non virtual member functions and fields with their virtual counterparts.

References

[1] M. Aksit and L. Bergmans. Composing multible concerns using composition filters. *Communications of the ACM*, 44, Issue 10:51–57, Oktober 2001.

[2] M. Aksit and B. Tekinerdogan. Solving the modeling problems of object-oriented languages by composing multiple aspects using composition filters. AOP'98 workshop position paper, 1998.

[3] T. Archer. *Inside Microsoft C#*. Microsoft Press, 1 edition, 2001.

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

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

[6] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. Getting started with aspectj. *Communications of the ACM*, 44, Issue 10:59–65, October 2001.

[7] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect oriented programming. In *European Conference on Object-Oriented Programming (ECOOP)*, Finnland, June 1997. Springer Verlag LNCS 1241.

[8] J. Lam. http://www.iunknown.com, 2002.

[9] B. Mandelbrot. *The Fractal Geometry of Nature*. Freeman, San Francisco, 1982.

[10] Microsoft. *Common Language Infrastructure*. Internal Working Document.

[11] W. Schult and A. Polze. Aspect-oriented programming with C# and .NET. In *International Symposium on Object-oriented Real-time distributed Computing (ISORC)*, pages 241–248, Crystal City, VA, USA, April 29 - May 1 2002.