

# On the Representation and Use of Metadata

Markus Lumpe

Department of Computer Science  
Iowa State University  
113 Atanasoff Hall  
Ames IA 50011, USA.

Tel: +1-515-294-2410, Fax: +1-515-294-0258

lumpe@cs.iastate.edu, WWW home page: <http://www.cs.iastate.edu/~lumpe>

**Abstract.** Metadata is the key to component-based programming. Metadata can provide additional information for every type and member defined in the program code in a language-neutral fashion. Moreover, metadata facilitates the interaction of a component with another component or application. In this position paper, we analyze the expressive power of *custom attributes*, one of the most innovative features of the .NET framework. We argue, that custom attributes, as built-in language abstractions, provide a way to represent both functional and non-functional properties of components. Furthermore, using the .NET reflection mechanism, custom attributes can be used to establish new business models, which have never been possible before.

## 1 Introduction

Traditionally, interfaces are used to specify the provided and required services of a software component in terms of *signatures*. These signatures are described in terms of dedicated language constructs or using a (language-independent) interface definition language such as the IDLs of COM [11] and CORBA [9]. In fact, all component frameworks before .NET followed this approach.

Although both provided and required interfaces of components are essential to understand the architecture of a component-based system, they generally do not contain enough information to give a detailed understanding of component interactions. In particular, “traditional” interfaces do not include dependencies between services: interfaces do not say what actually happens when a particular service is invoked. For example, in general an interface does not state any call-backs to the originating component [13]. Furthermore, required and provided interfaces are insufficient to support component composition correctly. It is possible to have compatible interfaces in a composed system, but the involved components do not match the required interaction behavior (e.g., data is encoded in a different way by the client than it is expected by the server). Therefore, a different view of the notion of an interface is

needed which includes the behavioral specification of component interactions and, in particular, the underlying contracts in which components participate.

The .NET framework is a programming model for developing, deploying, and running XML Web services that targets all types of applications [5]. In fact, the .NET framework is a rather radical approach to make the development on Windows more component-oriented [14].

The .NET framework provides a rich set of features to build *open applications*, most notably is the full support for interoperability with existing (COM) applications and the new model for language integration by means of a Common Language Specification (CLS) and a Common Language Runtime (CLR). In fact, CLS determines the minimum requirements that a programming language has to satisfy in order to be a .NET-aware language. Language compilers that conform to the CLS create objects that can interoperate with one another. Moreover, .NET-aware languages meet some or all requirements for composition languages [8]. As a consequence, .NET-aware languages can therefore be classified as *composition languages*.

Custom attributes are one of the most innovative features of the .NET framework [10]. Custom attributes provide component developers a generic means of associating information (as feature annotations) with types and members part of components, set of components, or assemblies, the units of deployment in the .NET framework.

Custom attributes represent metadata, which can be used to define design-time information (such as component documentation), runtime information (such as the names of available servers, needed to accomplish the current task), or even runtime behavioral characteristics (such as whether a transaction is allowed for a given bank account). Most notably, the number of applications of custom attributes is not limited in any way, because component developers can create an attribute based on the information they need.

Classic COM binaries are typically equipped with type libraries, which are collections of information about data types, interfaces, member functions and so on [1]. The problem with COM type information is, however, that it is not guaranteed to be present at runtime and more importantly that type libraries do not provide any support for runtime configuration.

In .NET, this problem is addressed using *metadata*. Metadata is part of every assembly and it provides additional information for every type and member defined in the assembly in a language-neutral fashion. Moreover, metadata is specified in the same file as the program code, allowing a component developer to specify the provided and required services of a component or set of components and their corresponding implementation simultaneously. Finally, every .NET-aware compiler

is able to automatically emit metadata, when building an assembly and therefore, metadata is always present. As a result, the deployment of components becomes easier and more reliable.

The definition of attributes alone is useless, since metadata written out in to a assembly will not affect the behavior of the application code per se. However, all .NET-aware languages and compilers provide a standard mechanism for defining custom attributes and for discovering them at runtime. Part of the .NET framework is the *System.Reflection* namespace. This namespace provides the abstractions for reflection, the process of runtime type discovery. Custom attributes are queried using either *GetCustomAttribute* (to retrieve for a particular piece of information) or *GetCustomAttributes* (to query an array of custom attributes associated with a type or member of an assembly). While querying an existing custom attribute, its serialized constructor <sup>1</sup> is de-serialized and invoked. All code associated with the constructor is executed. The constructor code is not limited in any way. It can instantiate any number of actions. In fact, the language used to specify an attribute may be the same as the language used to implement the assembly itself, hence facilitating meta-programming.

This paper is organized as follows: in section 2, we analyze the main mechanisms to specify components. In particular, discuss the notions of functional and non-functional properties and illustrate briefly how these concepts allow for a much more precise specification of software components. In section 3, we present small case studies, which shall illustrate the expressive power of custom attributes currently available for the .NET framework. Finally, we conclude with a summary of the main observations and a discussion about related and future work in section 4.

## 2 Functional and Non-functional Properties

An interface is a contractual specification that defines how we can communicate with a component. The actual contract defines both what clients need to do in order to use the component's required services and the provided services clients can expect when using the component. Moreover, signature-based interfaces are often enriched with pre- and post-conditions to further specify how to interact with a software component.

---

<sup>1</sup> A type or member is annotated with a custom attribute, say `UsageAttribute`, by specifying an appropriate attribute constructor (without the suffix `Attribute`) in curly brackets immediately before the feature to which this attributes should be applied(e.g., `[Usage("initialization string")]`).

In general, interfaces are used to provide the means to describe the *functional properties* of a component. An interface is an abstract, strongly typed contract between a software component and a client. However, complete interface specification is not an easy task. It is not enough to collect functional requirements and represent them appropriately in a contractual interface specification. Software components are designed and implemented to be reused. But code sharing and reuse do not happen spontaneously. Code reuse needs to be planned for, and its implications must be carefully thought through [7]. Moreover, in order to facilitate the definition of reliable and reusable code, we need appropriate language support.

However, most component frameworks require the specification of multiple, special purpose interfaces. For example, the EJB component model [6] requires that one maintains two interfaces: (i) the remote interface, which defines a bean's business methods, and (ii) the home interface, which defines a bean's life cycle methods. Both interfaces clearly represent different roles of an EJB component.

Secondly, more and more component frameworks provide or will provide support for different use cases. That is, components of a particular framework are required to offer

- a design-time interface (also known as a *composition interface*),
- a run-time interface,
- a query interface,
- a documentation interface, and
- a model checking interface.

For example, the documentation interface for Java is specified using special “doc comments”, which are turned into HTML documentation by the *javadoc* program. Similarly, in C# we use triple forward slash comments (`///`) as special XML-based comment syntax. Then C#-compiler can extract this information from a source file and will generate a corresponding XML output file.

As far as functional properties are concerned, current practice of interface specification is more or less sufficient. In fact, we can consider as a functional property any aspect of a software component, which can be directly expressed in the source code of a component that implements a given interface. However, there are other, sometimes even more important properties of software components, which cannot in general be deduced from the source code. Examples of these kinds of properties include performance issues (such as run-time and memory constraints), versioning, or deployment constraints to name a few. These properties (which are generally known as *non-functional properties*) play an important role in component software and, therefore, must be considered as well.

As an example, consider platform dependencies. Even though two software components may fit perfectly together by means of their interfaces, they may not be usable in the same application as they may have different runtime constraints. In order to make explicit these assumptions of components, Sametinger proposes the notion of a *component platform*, which refers to any soft- or hardware a component is built upon [12]. Typical examples of component platforms are operating systems, runtime systems, window systems, compilers, libraries, and network connections. The .NET framework, by means of custom attributes and reflection, is the first component framework, which has a built-in support for non-functional properties.

How do we communicate functional and non-functional properties? Before .NET, components were usually bundled up in so-called *component libraries*, a shipping and packing scheme that was proposed as early as 1969 by McIlroy [4]. In the .NET framework, the unit of deployment is an *assembly*. An assembly contains code that the common language runtime (CLR) executes, metadata that can describe every element managed by the common language runtime, and optional resources.

For the specification of functional and non-functional properties we use following approach. Functional properties that a part of the contractual specification of a component are represented by interfaces and their corresponding implementations. Non-functional properties and functional properties like documentation are represented by custom attributes, which can queried and used at both design time and runtime by means of reflection. Both are stored in a corresponding assembly.

### 3 A Case Study

In the following, we will present some small examples that illustrate how selected functional and non-functional properties can be represented as .NET metadata. We will use C# as implementation language, but the mechanisms are available for every other .NET-aware language as well.

In C# one can document the code using XML (this feature is unique to C#). The mechanism is similar to that used in Java. However, unlike Java, where one needs to use the *javadoc* utility, the C# documentation model is slightly different, in that the process “source code to XML formatting” is a built-in feature of the C#-compiler.

Unfortunately, the XML output is stored in a separate file. It is recommended to place the XML file in the same location as the assembly. But this approach has several weaknesses. First, documentation is metadata, which should be packaged in the assembly. Secondly, there is no guarantee that the XML-documentation is up-to-

```

using System;

public class DocTextAttribute : Attribute
{
    private String fText;

    public DocTextAttribute( String aText ) { fText = aText; }

    public String Text { get { return fText; } }
}

```

**Fig. 1.** The custom attribute *DocTextAttribute*.

date or even shipped with the assembly, because the assembly and its documentation are separate entities.

A better approach is to use a custom attribute to attach documentation to the assembly as an annotation. A simple custom attribute for documentation, called *DocTextAttribute*, may be defined as shown in Figure 1.

The custom attribute *DocTextAttribute* defines two features: (i) a constructor, which expects one argument - a documentation string, and (ii) a getter property *Text* to access the documentation string. The following code fragment illustrates the use of this custom attribute:

```

[DocText( "<summary>The canonical \"Hello, World\" class.</summary>" )]
public class HelloWorld
{
    [DocText( "<summary>The main entry point of HelloWorld.</summary>" )]
    public static void Main()
    {
        Console.WriteLine( "Hello, World" );
    }
}

```

Both applications of the custom attribute *DocTextAttribute* have no direct effect on the code, that is, their application does not change the behavior of the code. Their sole purpose is to annotate the code with human readable (and well-formed) documentation<sup>2</sup>. However, more important is the fact that due to the use of a custom attribute, the documentation is now part of the assembly. There is no need to maintain an additional XML-file.

<sup>2</sup> Due to limitations of the C#-syntax, these document annotations must be decorated as strings.

```

public static void GetXMLForMain()
{
    Type t = typeof(HelloWorld);
    MethodInfo mi = t.GetMethod( "Main" );
    Object[] mattrs = mi.GetCustomAttributes( false );

    foreach ( DocTextAttribute mdt in mattrs )
    {
        Console.WriteLine( "<member name=\"M:HelloWorld.Main\">");
        Console.WriteLine( mdt.Text );
        Console.WriteLine( "</member>" );
    }
}

```

**Fig. 2.** The method `GetXMLForMain`.

If we want to recover the information in a XML-based form, we need to use reflection. In fact, we use reflection to query information about custom attributes and while querying them, we may change the behavior of the annotated item. In case, however, we are only interested in the metadata the custom attribute *DocTextAttribute* represents, that is, if we query the assembly for instances of the attribute, we should get back the corresponding XML-based information, if the attribute is defined for a particular type or member of the assembly.

The procedure *GetXMLForMain*, shown in Figure 2, illustrates how we can query the custom attribute *DocTextAttribute* and, if this attribute is defined for the method `Main`, how we can extract the corresponding XML-based information.

Custom attributes can also be used to implement different component trading models. For example, consider a scenario in which a component vendor wants to implement a *pay-by-use* scheme. That is, every instantiation of a component may require a check whether the current license still covers this instantiation. Such a policy can be considered as a non-functional property of components developed by that vendor. In order to implement such a policy, we use a:

- A *XML-Web Service* that implements the server side of the *pay-by-use* scheme. The Web Service uses a small data base to store client data and to retrieve information that verifies that a particular use of a component is still allowed.
- A custom attribute *PayByUseAttribute* that uses a *CustomerId* to verify whether to annotated component can still be used. The custom attribute uses reflection and a proxy to communicate with remote the *XML-Web Service* and provides two public getter properties that state (i) whether the current instantiation is allowed, and (ii) how often this component can still be used.

```

using WebServiceHost;

public class PayByUseAttribute : Attribute
{
    private int fCredits;

    public PayByUseAttribute( String CustomerId )
    {
        ChargeUse ws = new ChargeUse();
        fCredits = ws.DecCredit( fCustomerId );
    }

    public int Credits { get { return fCredits; } }

    public bool CanUse { get { return fCredits != 0; } }
}

```

**Fig. 3.** The custom attribute `PayByUseAttribute`.

The *Web Service* is hosted by IIS (*Internet Information Server*) under a unique virtual directory (e.g. `PayByUse`). A *Web Service* works like a CGI-script, that is, the *Web Service* is activated by IIS, when a client sends an activation request. A *Web Service* supports several standard wire protocols (e.g. HTTP GET/HTTP POST, SOAP). Besides other methods, the *PayByUse Web Service* defines a method `DecCredit` that expects a `CustomerId` as argument. If the corresponding customer is defined in the server-side database (i.e., the customer is a valid licensee), then the method `DecCredit` returns an integer that states how often this client can use a particular service. If the result is zero, then either the license has expired or the caller is not a valid licensee. For both the activation request and the response the *Web Service* uses XML.

The custom attribute *PayByUseAttribute* is shown in Figure 3. It defines three features: a constructor that expects one argument - a customer identification, and two getter properties `Credits` and `CanUse`, which are used to access the corresponding license information. The constructor implements the attribute logic. First, a proxy for the *PayByUse Web Service*, called `ws`, is instantiated. Then, we call the *Web Service* method `DecCredit` using the customer id used as constructor argument. When the method returns, the private field `fCredits` is set to the number of invocations left.

This license verification process starts at runtime. More precisely, when the custom attribute *PayByUseAttribute* is queried. Figure 4 illustrates the application of the *PayByUseAttribute* for the class *TestPayByUse*. This class implements the

```

class TestPayByUse
{
    [PayByUseAttribute("349490202")]
    public static void Main(string[] args)
    {
        evalAttributes();
    }

    static void evalAttributes()
    {
        Type t = typeof(MainClass);
        MethodInfo mi = t.GetMethod( "Main" );
        int Credits = 0;
        Object[] mattribs = mi.GetCustomAttributes( false );

        for ( int i = 0; i < mattribs.Length; i++ )
            if ( mattribs[i] is PayByUseAttribute )
                Credits = ((PayByUseAttribute)mattribs[i]).Credits;

        if ( Credits != 0 )
            Console.WriteLine( "You have " + Credits.ToString() + " runs left!" );
        else
            Console.WriteLine( "You cannot use this application anymore!" );
    }
}

```

**Fig. 4.** The class `TestByUse`.

method `Main`, which is annotated with our custom attribute. In the method `Main`, we call `evalAttributes` to verify that we can still use this application or component. The reader should note that we do not actually enforce the policy. We rather use this example to illustrate the mechanism. When the license for customer “349490202” is still valid, say 20 credits left, then the method `evalAttributes` prints the following message on the console: You have 20 runs left!

## 4 Conclusion and Future Work

Custom attributes, as available in the .NET framework, can be used to represent a broad spectrum of functional and non-functional properties. We have only shown a small example here, but we believe that a generalization of this mechanism will significantly improve the way we construct, compose, and deploy components.

In component frameworks before .NET, the developer could only use interfaces to specify functional properties (e.g., IDL). However, as interfaces are used to define a contractual specification, they are not powerful enough to cover different use cases

of a component. Custom attributes, on the other hand, support different use cases by construction (we simply annotate a particular element of a component with an attribute like *DocTextAttribute*). Furthermore, custom attributes provide the means to incorporate non-functional properties like deployment constraints in the component.

At the moment, custom attribute are in general not evaluated at compile time<sup>3</sup>. Such an approach would be useful, for example, to generate metadata from an external source, or to change the compiler's behavior on-the-fly. Such an approach has already been used in the PICCOLA 1 [2, 3]. While compiling a PICCOLA source, the compiler is consulting a component library in order to find necessary parameters for the code generation.

## References

1. K. Brockschmidt. *Inside OLE 2: the Fast Track to Building Powerful Object-Oriented Applications*. Microsoft Press, 1993.
2. Markus Lumpe. *A  $\pi$ -Calculus Based Approach to Software Composition*. PhD thesis, University of Bern, Institute of Computer Science and Applied Mathematics, January 1999.
3. Markus Lumpe, Franz Achermann, and Oscar Nierstrasz. A Formal Language for Composition. In Gary Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, chapter 4, pages 69–90. Cambridge University Press, March 2000.
4. M.D. McIlroy. Mass Produced Software Components. In P. Naur and B. Randell, editors, *Software Engineering*. NATO Science Committee, January 1969.
5. An Introduction to Microsoft .NET. White Paper, Microsoft Corporation, 2001.
6. Richard Monson-Haefel. *Enterprise JavaBeans*. O'Reilly, second edition, 2000.
7. Oscar Nierstrasz, Simon Gibbs, and Dennis Tsichritzis. Component-Oriented Software Development. *Communications of the ACM*, 35(9):160–165, September 1992.
8. Oscar Nierstrasz and Theo Dirk Meijler. Requirements for a Composition Language. In Paolo Ciancarini, Oscar Nierstrasz, and Akinori Yonezawa, editors, *Object-Based Models and Languages for Concurrent Systems*, LNCS 924, pages 147–161. Springer, 1995.
9. Object Management Group. *The Common Object Request Broker: Architecture and Specification*, July 1996.
10. Jeffrey Richter. *Applied Microsoft .NET Framework Programming*. Microsoft Press, 2002.
11. Dale Rogerson. *Inside COM: Microsoft's Component Object Model*. Microsoft Press, 1997.
12. Johannes Sametinger. *Software Engineering with Reusable Components*. Springer, 1997.
13. Patrick Steyaert, Carine Lucas, Kim Mens, and Theo D'Hondt. Reuse Contracts: Managing the Evolution of Reusable Assets. In *Proceedings OOPSLA '96*, volume 31 of *ACM SIGPLAN Notices*, pages 268–285, October 1996.
14. Andrew Troelsen. *C# and the .NET Platform*. Apress, 2001.

---

<sup>3</sup> It seems, however, that certain Microsoft attributes are used to change the behavior of the C#-compiler. In particular, all *Web Service* attributes can trigger the compiler to emit additional XML-code.