Editor: Martin Fowler ■ ThoughtWorks ■ fowler@acm.org

How .NET's Custom Attributes Affect Design

James Newkirk and Alexei A. Vorontsov

In its first release of the .NET Framework, Microsoft has provided a defined method for adding declarative information (metadata) to runtime entities in the platform. These entities include classes, methods, properties, and instance or class variables. Using .NET, you can also add declarative information to the *assembly*, which is a unit of deployment that is conceptually similar to a .dll or .exe file. An assembly includes attributes that describe its identity (name, version, and culture), informational attributes that provide additional product or company information, manifest attributes that describe configuration information, and strong name attributes that describe whether the assembly is signed using public key encryption. The program can retrieve this metadata at runtime to control how the program interacts with services such as serialization and security. Here, we compare design decisions made using custom attributes in .NET with the Java platform.

Marker interfaces

In the Java platform there is a common design trick called *marker interfaces*. A marker interface has no methods or fields and serves only to identify to the Java Virtual Machine (JVM) a particular class attribute. Here is one example:

```
public interface Serializable {}
```

If the class that you are writing must be serializable, then it must implement the interface as follows:

```
public class MyClass implements
Serializable
```

Serialization might have certain behaviors associated with it that the class developer wants to control. However, Java doesn't explicitly associate such behavior with the interface that represents the serialization contract. At runtime, when the program tells the JVM to serialize this class, it looks to see if the class has implemented the interface. It also looks to see if the class has defined any special methods associated with the serializable interface but not directly declared in it, such as `readResolve`, `readObject`, or `writeObject`.

The JVM relies on a naming convention and method signatures to locate the methods via reflection; if it finds them, it invokes them. The interface itself does not specify any methods, because implementors might then unnecessarily implement methods in the simplest case. Because the interface doesn't explicitly specify the methods used to control the process and thus might incorrectly specify the method signature, this mechanism is prone to failure. Unfortunately, no compile time check will identify this as an error.

.NET solves this problem by being explicit. In the simplest case, where the programmer wants to rely on the provided capability to serialize an object, there is a class-level attribute called `Serializable`

0740-7459/02/$17.00 © 2002 IEEE

that marks a class as having that capability. For example,

```
[Serializable()]
public class MyClass {}
```

Marking a class serializable implies nothing else. If the programmer wants to completely control the serialization process, then he or she must also implement an interface called `ISerializable`, specifying the methods used to control serialization (see Figure 1). At runtime, when the program tells the Common Language Runtime to serialize a class, the CLR looks to see if the class was marked with `SerializableAttribute`.

The Java and .NET approaches are similar in intent, but .NET's use of attributes is more explicit. Contrary to an interface's basic purpose, the marker interface reuses an existing language construct interface to represent what the attribute represents. According to the *Java Language Specification* (2nd ed., Addison-Wesley, 2000),

> *An interface declaration introduces a new reference type whose members are classes, interfaces, constants and abstract methods. This type has no implementation, but otherwise unrelated classes can implement it by providing implementations for its abstract methods.*

Stylistic naming patterns

At the method level, it is common in the Java platform to use naming conventions to identify a special method. By virtue of the name, the program finds the method at runtime using reflection. Once found, the executing program specially interprets this method. For example, when a programmer defines a test method in JUnit—a popular unit-testing framework for Java (www.junit.org)—the first four letters of a test method must be `test` (see Figure 2a). The program that executes the tests first verifies that the class inherits from `TestCase`. Then, using reflection, it looks for any methods that start with `Test`. In the code in Figure 2a, the program finds and calls `testSuccess`.

The code in Figure 2b demonstrates a common design idiom used in JUnit when the programmer wants to verify that the code throws an exception. Unfortunately, the programmer will duplicate such code in every test case that expects an exception, and the idiom is not as intuitive as you might expect.

Having the testing framework support such a common case directly would be nice. However, relying on the naming convention could lead to some variation of the code in Figure 2c. In this case, we use a naming convention to specify not only a test method but also additional information about how to interpret the test result. We expect that this method's execution will throw `MyException`. This example might seem somewhat ridiculous, but it demonstrates the limitations of naming conventions, because of how much information the name itself can convey. In fact, JUnit doesn't implement the functionality to check boundry conditions in this way. Other approaches used in Java (such as JavaDoc tags) can provide additional information. However, they are not present at runtime and usually require preprocessing the code to identify and process the tags.

In .NET, stylistic naming patterns are not needed because, in addition to attributes that the Framework specifies, a programmer can create custom attributes that are defined and used in the same way. These attributes are not just names but are instances of classes that might have additional data. Figure 3a shows a similar test class defined with Nunit (www.nunit.org), a unit testing tool for the .NET platform. Nunit, a derivative of JUnit, supports all languages in the .NET framework and uses attributes at the class and method levels. The class attribute is called `TestFixture`; it tells the program that runs the tests to look for test methods in this class. A `Test` attribute then identifies test methods. This overall solution makes for a more consistent approach.

In addition, this solution is more extensible because more than one attribute can be associated with a method, and attributes can have additional data. For example, Nunit has another attribute defined for a method that expects an exception. This leaves the name not only unencumbered by the context in which it is run but also more relevant to what is being tested (see Figure 3b).

```
[Serializable()]
public class MyClass : ISerializable
{
    public MyClass(
        SerializationInfo info,
        StreamingContext context)
    {
        // ...
    }
    public void GetObjectData(
        SerializationInfo info,
        StreamingContext context)
    {
        // ...
    }
}
```

Figure 1. Implementing the `ISerializable` interface, which specifies the methods for controlling serialization.

Attributes in .NET provide an elegant, consistent approach to adding declarative information to runtime entities. Because the runtime entities interact with the supporting services via declarative information, the set of services and supporting attributes does not have to be closed. By providing a standard mechanism to extend built-in metadata with cus-

```
public class MyClass extends TestCase
{
    public void testSuccess()
    { /* ... */ }
}
```
(a)

```
public class MyClass extends TestCase
{
    public void testMyException()
    {
        try {
            /* code that throws exception */
            fail("Code should have thrown MyException");
        }
        catch(MyException e)
        { /* expected exception – success */ }
     }
}
```
(b)

```
public class MyClass extends TestCase
{
     public void testSuccess_ExpectException_MyException()
     { /* ... */ }
}
```
(c)

Figure 2. (a) A test method in JUnit (the method's first four letters must be `test`); (b) a test for the boundary conditions that verify that an exception is thrown when expected; (c) a naming convention to specify not only a test method but also some additional information about how to interpret the test result.

```
[TestFixture]
public class MyClass
{
    [Test]
    public void Success()
    { /* ... */ }
}
```
(a)

```
[TestFixture]
public class MyClass

{
    [Test]
    [ExpectedException(typeof(MyException))]
    public void Success()
    { /* would throw my exception */ }
}
```
(b)

Figure 3. A test class (a) defined with Nunit and (b) with another attribute defined for a method that expects an exception.

tom attributes, .NET lets programmers develop applications that can interact with services not yet defined or supported by CLR. In fact, Nunit version 2.0 was written with custom attributes and provides much of the flexibility we've demonstrated here.

In contrast, the most common ad hoc mechanisms in Java to add declarative information include marker interfaces, stylistic naming patterns, and JavaDoc tags. These inconsistently solve the problem of adding declarative information to runtime entities. They are also error prone and too simplistic for today's applications. The Java community recognizes this limitation and has started working on JSR-175 (see www.jcp.org/jsr/detail/175.jsp), which specifies a similar facility for Java that is already in .NET.

James Newkirk is a software project manager for Thoughtworks. He has been working with the .NET Framework since its introduction in the summer of 2000. Contact him at jim@nunit.org.

Alexei A. Vorontsov is a software technical lead for Thoughtworks. He has worked on an enterprise transforming application for the past three years. Contact him at alexei@nunit.org.