using C# reflection

by Glen McCluskey

Glen McCluskey is a consultant with 20 years of experience and has focused on programming lan-guages since 1988. He specializes in Java and C++ performance, testing, and technical documentation areas.

glenm@glenmccl.com

In this column we want to consider a set of C# features that go by the name "reflection." This term is a little hard to describe but in general refers to the ability of a running C# program to examine data about itself and modify its behavior accordingly. We'll look at several examples to help clarify this idea.

For reflection features to work, a C# system needs to keep data about the program around, so that it can be accessed at runtime. We can call this information "metadata," to distinguish it from the application data the program operates on.

Finding Types in a Running Program

Let's start with a fairly simple use of reflection, as illustrated in this program:

```
using System;
using System.Reflection;

public class DumpTypes {
    public static void Main(string[] args) {
        string targstr = (args.Length == 0 ?
            null : args[0].ToLower());

        Assembly asm = Assembly.Load("mscorlib.dll");

        Type[] typelist = asm.GetTypes();

        foreach (Type type in typelist) {
            string typestr = type.ToString().ToLower();
            if (targstr != null && targstr != typestr)
                continue;

            Console.WriteLine(type);

            MemberInfo[] memlist = type.GetMembers();
            foreach (MemberInfo mem in memlist)
                Console.WriteLine("    " + mem);
        }
    }
}
```

If you run this demo without any arguments, it displays about 25,000 lines of output, a list of all the classes and interfaces that the C# runtime system knows about, together with the members of each class. This process is sometimes called "type discovery."

The first few lines of output are as follows:

```
System.Object
    Int32 GetHashCode()
    Boolean Equals(System.Object)
    System.String ToString()
    Boolean Equals(System.Object, System.Object)
    Boolean ReferenceEquals(System.Object, System.Object)
```

```
System.Type GetType()
Void .ctor()
```

System.Object is a class known to C#, and it has class members such as GetHashCode and Equals. These members have particular parameter types such as System.Object and return types like Int32.

You can also run the program and specify the name of a class, like System.String, and only the details of that class will be displayed. The program works by opening an assembly, which is something like an archive file or dynamic link library. It then reads the types and members in a standardized way and displays them.

Obtaining a list of classes and members may not seem like much, but it's impossible to do in languages like C and C++. About the most you can do in C is to open an archive file like libc.a and read through it. There is no standard way of doing this operation, and information about parameter and return types for the various C functions is not preserved.

Dynamic Invocation

Let's go on and look at another use of reflection, one that's a little more sophisticated. Suppose that you have an application such as a C# interpreter, or a debugger, or browser, or something, and you'd like to invoke methods by name at runtime.

Or to say it another way, imagine that you are doing C programming and you have a program like this:

```
void f1() {}
void f2() {}
int main(int argc, char* argv[]) { … }
```

The user specifies "f1" or "f2" as a string on the command line, and you call the right function based on what is specified. The only way you can program such a feature is by means of a big if-then statement, or by keeping a table of function names/pointers around so that you can do dynamic dispatch. This approach is cumbersome.

Here's some C# code that solves this problem:

```
using System;
using System.Reflection;

public class InvokeDemo {
    static object run(string classname, string methodname) {
        Assembly asm = Assembly.Load("mscorlib.dll");

        Type type = asm.GetType(classname);

        object obj = asm.CreateInstance(classname);

        object[] args = new object[0];

        object ret = type.InvokeMember(
            methodname,
            BindingFlags.Default | BindingFlags.InvokeMethod,
            null,
```

```
            obj,
            args
        );

        return ret;
    }

    public static void Main(string[] args) {
        while (true) {
            try {
                Console.WriteLine();

                Console.Write("Enter class name: ");
                string classname = Console.ReadLine();

                Console.Write("Enter method name: ");
                string methodname = Console.ReadLine();

                object ret = run(classname, methodname);

                Console.WriteLine("ret value = " + ret);
            }
            catch (System.Exception exc) {
                Console.WriteLine("exception: " + exc);
            }
        }
    }
}
```

A typical interaction looks like this:

```
Enter class name: System.Object
Enter method name: GetHashCode
ret value = 21
Enter class name:
```

This interaction specifies that an object of the System.Object class is created, and then the GetHashCode method is called on the object. The return value (the hash code) is 21.

This program first queries the standard assembly for type information about the specified class (System.Object), creates an instance of that class, and then calls the method (GetHashCode) on the instance. This simple demo works only for parameterless methods, but can easily be extended to accept a list of arguments to be passed to the method.

The demo also illustrates exception handling, used to catch errors such as specifying an unknown class or method name.

Custom Attributes

A third way that you can use C# reflection is to specify and use custom attributes. An attribute is metadata about a C# program. It is not the program instructions or application data, but rather something like a "smart comment."

Let's illustrate the idea of attributes by looking at a sample application. Suppose that you'd like to incorporate code-change logs into your source code. That is, whenever

the source is edited, some information is added that describes the date, author, and nature of the change. One way to do this is through comments in the code. But comments have no structure, and are not queryable except in an ad hoc way.

Another way to solve this problem is by using C# custom attributes. Here's some code that shows how this can be done:

```
using System;
using System.Reflection;

[AttributeUsage(
    AttributeTargets.Class |
    AttributeTargets.Property, AllowMultiple = true
)]

public class CodeChangeAttribute : Attribute {
    private string id;
    private string who;
    private string date;
    private string descr;

    public CodeChangeAttribute(string id, string who,
    string date, string descr) {
        this.id = id;
        this.who = who;
        this.date = date;
        this.descr = descr;
    }

    public string GetId() {
        return id;
    }

    public string GetWho() {
        return who;
    }

    public string GetDate() {
        return date;
    }

    public string GetDescr() {
        return descr;
    }
}

[CodeChangeAttribute("123", "Jane Smith",
    "5/27/04", "initial checkin")]
[CodeChangeAttribute("456", "Bill Jones",
    "5/29/04", "fix three bugs")]

public class TestAttr {}

public class AttrDemo {
```

```
    public static void Main(String[] args) {
        MemberInfo meminfo = typeof(TestAttr);

        object[] attrlist = meminfo.GetCustomAttributes(
            typeof(CodeChangeAttribute), false);

        foreach (object attr in attrlist) {
            CodeChangeAttribute cca = (CodeChangeAttribute)attr;
            Console.WriteLine("id = " + cca.GetId());
            Console.WriteLine("who = " + cca.GetWho());
            Console.WriteLine("date = " + cca.GetDate());
            Console.WriteLine("descr = " + cca.GetDescr());
            Console.WriteLine("————————");
        }
    }
}
```

The first part of the demo defines a class called CodeChangeAttribute. This is a custom attribute, represented as a class, and objects of the class contain information on particular code changes. The information with each object is the change ID, the name of the person making the change, the date, and a description of the change.

This attribute can then be used with other classes you define, such as TestAttr. The attribute values are delimited by [...] and appear before the definition of the class.

The last part of the demo, the code in Main, shows how to query the attributes. When you run this program, the output is

```
id = 123
who = Jane Smith
date = 5/27/04
descr = initial checkin
————————
id = 456
who = Bill Jones
date = 5/29/04
descr = fix three bugs
————————
```

Attributes are annotations that you place on source code elements such as classes and method names. They can be used to provide information about an element, such as the code-change log example above, or affect the runtime behavior of a C# program. Attributes are not program data in the conventional sense, and they are not comments. They can be queried using C# reflection facilities.

Reflection is a powerful tool that you can use in your C# programs. It supports a rich and dynamic programming style, and opens the door for many innovative applications.