IL Code Instrumentation with RAIL

Open source project offers easy, object-oriented modification of existing assemblies

January 12, 2004

Summary
Imagine that you have downloaded an application for the .NET platform, and that for some reason you would like to know what it is accessing on the hard drive. In particular, you would like to know which files it is reading from and writing to.

By Bruno Cabral, Paulo Marques & Luis Silva

Imagine that you have downloaded an application for the .NET platform, and that for some reason you would like to know what it is accessing on the hard drive. In particular, you would like to know which files it is reading from and writing to. This can be a problem since you may not have access to the source code of the application, and using Ildasm to iterate through the whole code is out of the question. In this article we will show you how to perform code instrumentation with the Runtime Assembly Instrumentation Library (RAIL), which allows assemblies to be easily modified according to high-level specifications.

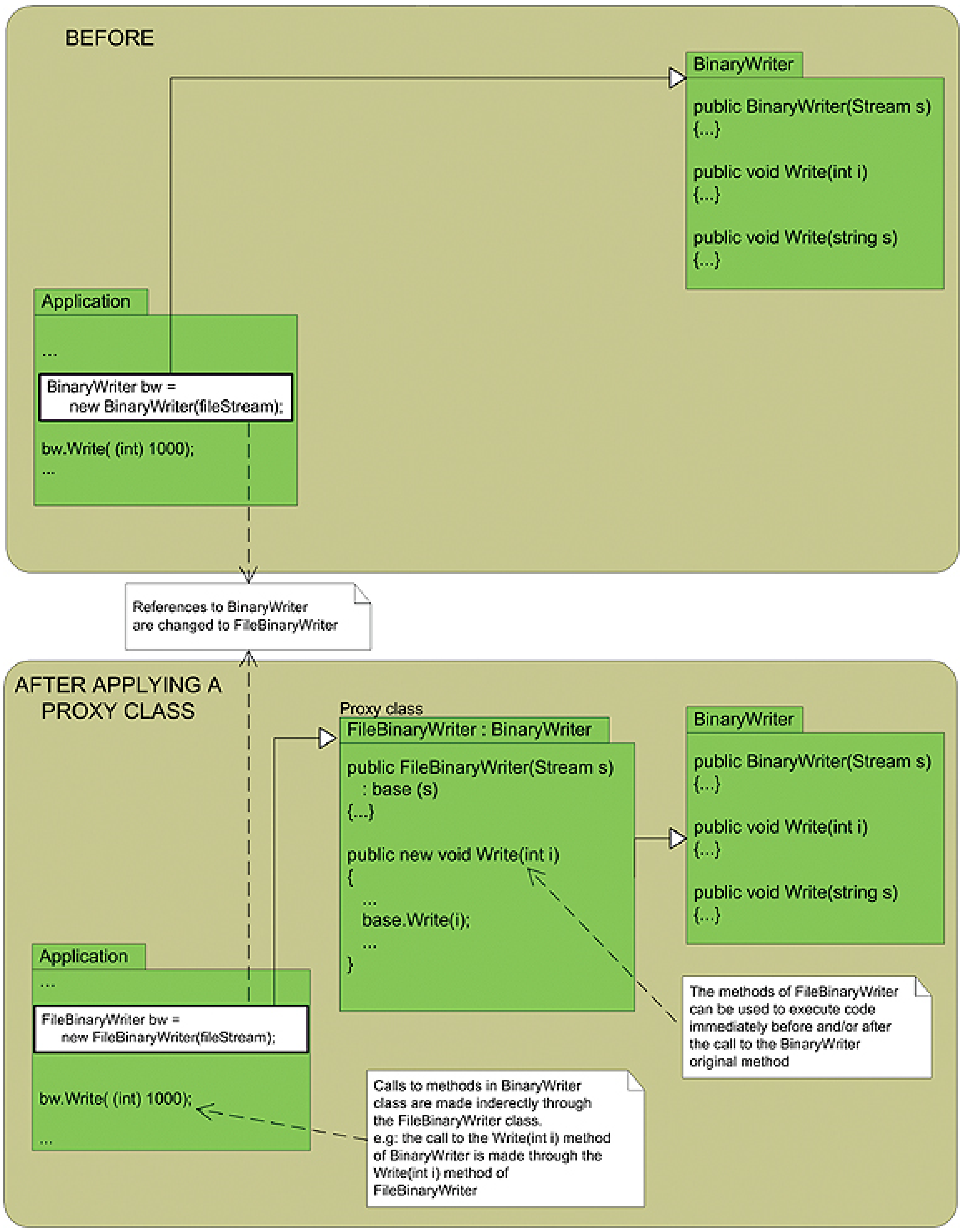
The Runtime Assembly Instrumentation Library

RAIL fills the gap that exists between .NET's reflection mechanisms and Reflection.Emit. While it is possible to use reflection to collect information about assemblies and types, and with Reflection.Emit it is possible to generate new assemblies at runtime, RAIL allows the modification of existing assemblies in an easy and object-oriented way. Some of the available high-level modifications include the ability to alter references in the code to types, fields, properties, and methods; insert prologue and epilogue instructions in method bodies; replace access to fields by methods; and selectively copy parts of assemblies. Many of these functionalities benefit from the use of design patterns to iterate through the object structure and apply the changes.

A Simple Example

Let's consider the example mentioned earlier: how to monitor all disk accesses of an application.With RAIL it is easy to replace all references to a certain class with new references to another class. In practice, to solve the problem it is necessary to replace all the references to classes that provide access to disk with references to an intermediate class. In this class, a log entry is created each time a method is called and then the original invocation can take place. The intermediate class is serving as a proxy between the original references and the classes that read and write from disk.

To completely solve the monitoring problem it would be necessary to proxy all the text reader and writer classes and also their binary counterparts. To keep the example short we will do this only for the System.IO.Binary Writer class, creating a File BinaryWriter class (see Figure 1 and Listing 1) (Code listings are available at www.sys-con.com/dotnet/sourcec.cfm).


BEFORE
BinaryWriter
public BinaryWriter(Stream s) {...}
public void Write(int i) {...}
public void Write(string s) {...}
Application
...
BinaryWriter bw = new BinaryWriter(fileStream);
bw.Write((int) 1000);
...
References to BinaryWriter are changed to FileBinaryWriter
AFTER APPLYING A PROXY CLASS
Proxy class
FileBinaryWriter : BinaryWriter
public FileBinaryWriter(Stream s) : base (s) {...}
public new void Write(int i) { ... base.Write(i); ... }
BinaryWriter
public BinaryWriter(Stream s) {...}
public void Write(int i) {...}
public void Write(string s) {...}
Application
...
FileBinaryWriter bw = new FileBinaryWriter(fileStream);
bw.Write((int) 1000);
...
The methods of FileBinaryWriter can be used to execute code immediately before and/or after the call to the BinaryWriter original method
Calls to methods in BinaryWriter class are made inderectly through the FileBinaryWriter class. e.g: the call to the Write(int i) method of BinaryWriter is made through the Write(int i) method of FileBinaryWriter

To avoid concurrency problems when writing to the log, a new file is created every time the FileBinary Writer constructor is called. This is achieved by dynamically generating the name of the log file using a static variable (LogVersion) in FileBinary Writer. This variable is incremented every time it is used.

When overriding the Write() methods, a call is made to the base method and a log entry is generated with a call to this.FileWriteLog.Write Line(fs.Name,v.ToString());.

The FileWriteLog code in Listing 2 implements a very simple XML log maker. The constructor creates the log file, writes the start-of-document line, and inserts the first element entry. The Close() method writes the end of the first entry, closes the document, and flushes the buffer. The WriteLine(string filePath, string val) method locks the access to the XmlTextWriter object to assure the atomicity of each log entry.

FileBinaryWriter and FileWriteLog are compiled under the MyFile namespace into the MyFile.dll assembly.

Foo.exe (see Listing 3) is the test assembly for this example. It is an adaptation of an MSDN example on how to use BinaryWriter objects. This short application writes the numbers from 0 to 10 into the Test.data file using BinaryWriter. It also checks if Test.data does not exist before the execution of Foo.exe.

Putting RAIL to Work
In this example, RAIL will be used to copy the FileBinary Writer and FileWriteLog types into the Foo.exe assembly, and to replace the calls to Binary Writer with calls to the newly inserted FileBinaryWriter. Listing 4 describes how this is done. Let's examine the implementation.

When using RAIL, the first thing to do is create a representation of the assembly to instrument using the RAIL.Reflect. RAssemblyDef class. The RAssemblyDef.Load Assembly() creates an RAssemblyDef object, which represents the Foo.exe assembly. The same kind of object must also be created for the MyFile.dll assembly (see lines 15 and 16 of Listing 4).

The next step is to obtain RType objects representing the FileBinary Writer and FileWrite Log classes of MyFileAssembly. These instances are FileBinary WriterType and FileWriteLog Type, respectively.

Using the CopyType() method of RAIL.Reflect.RModuleDef, two copies of the RType objects are made into the FooAssembly instance. From these copy operations, two RTypeDef objects are returned.

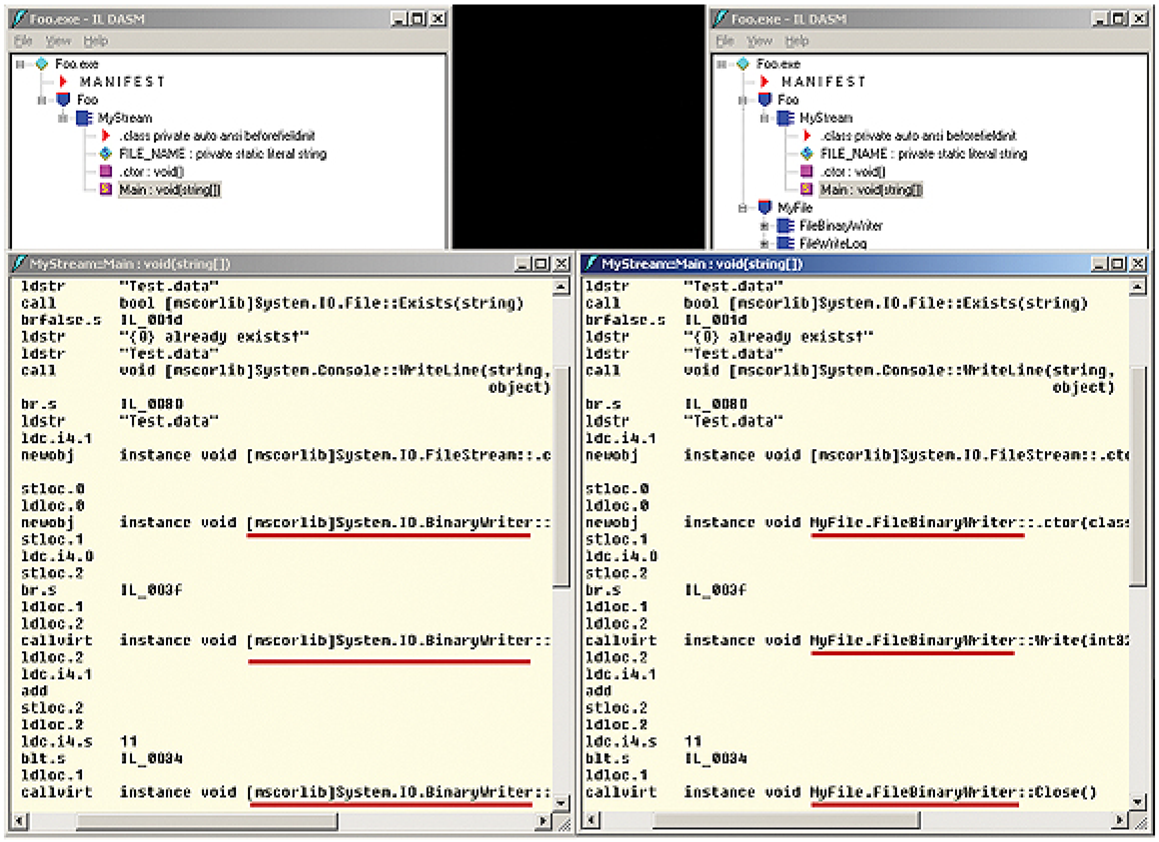
The ReferenceReplacer class allows you to change references from one type to another. It is based on the Visitor design pattern. Line 28 of Listing 4 creates a ReferenceReplacer object for changing all the references in NewFileBinaryWriterType from FileWriteLogType to NewFile WriteLogType. The changes are applied using the Accept() method. This concludes the copying of the new type into the FooAssembly. Note that this type of copying step is not really necessary for the example; an external reference to the MyFile.dll assembly would be enough. Nevertheless, it is a nicer way to demonstrate RAIL's capabilities.

The next step is to create an RType object to represent the BinaryWriter class reference in the FooAssembly. Then, the ReferenceReplacer instance is instructed to redirect all the references from OriginalBinary Writer to New FileBinaryWriter Type.

This ReferenceReplacer is applied to the RTypeDef instance of Foo.MyStream on line 38.

Finally, FooAssembly.Save Assembly ("Foo.exe") saves the new version of the assembly one directory up in the file system.

Figure 2 shows a screenshot of Ildasm. On the left the original assembly is shown; on the right the modified version is presented. It is easy to see that all references to [mscorlib] System.IO.Binary Writer have been replaced with references to MyFile.FileBinary Writer.


Foo.exe - IL DASM
File View Help
Foo.exe
MANIFEST
Foo
MyStream
.class private auto ansi beforefieldinit
FILE_NAME : private static literal string
.ctor : void()
Main : void(string[])
Foo.exe - IL DASM
File View Help
Foo.exe
MANIFEST
Foo
MyStream
.class private auto ansi beforefieldinit
FILE_NAME : private static literal string
.ctor : void()
Main : void(string[])
MyFile
FileBinaryWriter
FileWriteLog
MyStream::Main : void(string[])
MyStream::Main : void(string[])

```
ldstr        "Test.data"
call         bool [mscorlib]System.IO.File::Exists(string)
brfalse.s    IL_001d
ldstr        "{0} already existst"
ldstr        "Test.data"
call         void [mscorlib]System.Console::WriteLine(string,
                                                       object)
br.s         IL_0080
ldstr        "Test.data"
ldc.i4.1
newobj       instance void [mscorlib]System.IO.FileStream::.c

stloc.0
ldloc.0
newobj       instance void [mscorlib]System.IO.BinaryWriter::
stloc.1
ldc.i4.0
stloc.2
br.s         IL_003f
ldloc.1
ldloc.2
callvirt     instance void [mscorlib]System.IO.BinaryWriter::
ldloc.2
ldc.i4.1
add
stloc.2
ldloc.2
ldc.i4.s     11
blt.s        IL_0034
ldloc.1
callvirt     instance void [mscorlib]System.IO.BinaryWriter::
```

```
ldstr        "Test.data"
call         bool [mscorlib]System.IO.File::Exists(string)
brfalse.s    IL_001d
ldstr        "{0} already existst"
ldstr        "Test.data"
call         void [mscorlib]System.Console::WriteLine(string,
                                                       object)
br.s         IL_0080
ldstr        "Test.data"
ldc.i4.1
newobj       instance void [mscorlib]System.IO.FileStream::.ct

stloc.0
ldloc.0
newobj       instance void MyFile.FileBinaryWriter::.ctor(clas
stloc.1
ldc.i4.0
stloc.2
br.s         IL_003f
ldloc.1
ldloc.2
callvirt     instance void MyFile.FileBinaryWriter::Write(int3
ldloc.2
ldc.i4.1
add
stloc.2
ldloc.2
ldc.i4.s     11
blt.s        IL_0034
ldloc.1
callvirt     instance void MyFile.FileBinaryWriter::Close()
```

When executing the original Foo.exe, a file named "test.data" is produced. When executing the modified version, a file called "log0.xml" will also be created. This file contains a log of the accesses that Foo.exe has made to the hard drive.

Conclusion

Although RAIL is still very new, it already provides powerful features in terms of code instrumentation, including code walkthrough, reference replacement, type copying, and method prologues and epilogues. The RAIL team is currently working on v1.0 of the software, which should be available by the end of the first trimester of 2004. Among other things, it will support assemblies with multiple modules.

The objective of RAIL is to simplify the task of code instrumentation, making it possible to use it in areas such as aspect-oriented programming, runtime analysis tools, security verification, and software fault injection.

Resource

- RAIL is an open source project of the Dependable Systems Group of the University of Coimbra, is distributed under the Mozilla Public License, and is available for download from: http://rail.dei.uc.pt