WHITE PAPER

Java

Know What You're Executing: Finding a list of All Loaded Classes

By Ted Neward

Abstract

Understanding what version of your code is executing in production can be the difference between a solved bug and an embarrassing admission of ignorance. Too many times developers are caught up defending code that isn't properly migrated out to Production or a customer's machine. In some extreme cases, developers (and/or tech support staff) are expected to support code on client machines with no idea of precisely what version of the code is running there. Add to this mess the usual variety of patches, minor version releases and daily builds, and you have a recipe for complete disaster, both from a personal and professional standpoint.

One of the common means to address this problem, coming to us from the C/C++ world, is to embed a string in a compiled object file, which then gets loaded into the compiled executable. "In the field", developers and/or tech support staff can run a utility (usually the Unix "strings" command or some variant thereof) to see the exact versions of each .C/.CPP file used to build the code the customer is currently running. Unfortunately, because Java doesn't support static linking, the same approach doesn't work for us; however, we can adapt it (through one of two ways) to provide much the same level of support.

Problem Discussion

Maintenance

For many developers, it's believed that responsibility for code ends with the infamous "Ship It!". Experienced developers, however, implicitly and intuitively realize that life only gets more interesting once the code is shipped—now comes the maintenance phase of the software lifecycle, one which usually drives us all nuts. Diagnosing problems is in of itself a fine art, usually involving the principle of elimination— keep removing variables until only one possibility must remain, and thus, must be the bug. We thus only have to remove possibility after possibility until only one possibility is left, fix the

offending line of code (or broken card, or malfunctioning peripheral), and the problem is solved. Unfortunately, this principle suffers from a single problem:
It's never that easy in practice.

Back when software consisted of simple instructions executed directly by the CPU in a deterministic fashion, it was always possible to sit down with a sheet of paper in one hand, a hardcopy printout of the code in the other hand, and reason out precisely what the CPU was doing from one moment to the next. Fast-forward to circa 2000, however, and we find an incredibly complex environment in which multiple Threads can concurrently (and non-deterministically) access the CPU and execute code. I/O operations can fail, owing to minute differences between one machine and the next, something so simple as a directory, assumed to be available, not being present. Trying to nail down the differences between "my machine" and "your machine" can be a serious undertaking in of itself.

All of this becomes an order of magnitude more complex, however, when we can't even guarantee that the code running on "your machine" isn't even the same version is the code running on "my machine". Source-code control systems, like Visual SourceSafe, PVCS, or CVS, solve some of this problem, by associating versions of the source code with human-friendly labels like "1.25" or "Release 2.4" or "Build2000.02.14". That's only the first step, however—developers still need some way to identify the version of the code currently executing. Moreover, the ability to resolve the version of the code currently executing back to the version of the source file that produced it goes a long way towards finding—and ultimately eliminating—bugs.

Solution Discussion

A Historical Solution

This problem is not a new one. Developers struggled with this same problem as early as ten years ago (if not sooner), back in the heyday of C++, and numerous suggestions came as solutions to the problem. One such suggestion came from [1]: create strings with version information within it, and embed the string inside the compiled file:

```
/* C example */
static char[] rcsInfo = "Version 1.0";
```

By marking the char array "static", the string is private to the .C file in which it was declared; this way, multiple .c files can each have their own version information string embedded within it:

```
/* a.c */
static char[] rcsInfo = "RCS: a.c: Version 1.0";
/* b.c */
static char[] rcsInfo = "RCS: b.c: Version 2.1";
/* c.c */
static char[] rcsInfo = "RCS: c.c: Version 1.7.1.4";
```

Now, when these three .c files are compiled into an executable, the executable will contain[1] these three strings embedded inside the executable's file format, usually somewhere inside the code sections. Tools such as Unix's "strings" utility can scan through the executable file's bytes, looking for the versioning strings. In [1], Stephen Davis suggested creating a specific version-scanning tool, which searched the executable for a specific string prefix, such as the "RCS:" string above[2]. Then, when executing this version-scanning tool, a developer could see precisely which version of the source files the user was running:

```
/* main.c */
#include
// Prototypes from one.c
void one();
// Prototypes from two.c
void two();
static char rcsInfo[] = "RCS: main.c, Version 1.0";
int main(int argc, char* argv[])
{
one();
two();
}
/* one.c */
#include
static char rcsInfo[] = "RCS: one.c, Version 1.0";
void one()
{
printf("one called\n");
}
/* two.c */
#include
static char rcsInfo[] = "RCS: two.c, Version 1.0";
void two()
{
printf("two called\n");
}
C:\Projects\Papers\FindingLoadedClasses\code\c>cl main.c one.c
two.c
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 12.00.8168
for
80x86
Copyright (C) Microsoft Corp 1984-1998. All rights reserved.
main.c
one.c
two.c
Generating Code...
Microsoft (R) Incremental Linker Version 6.00.8168
Copyright (C) Microsoft Corp 1992-1998. All rights reserved.
/out:main.exe
main.obj
one.obj
two.obj
C:\Projects\Papers\FindingLoadedClasses\code\c>main
one called
two called
```

Although it's not visible at main.exe's execution time, main.exe contains three constant strings: "RCS: main.c, Version 1.0", "RCS: one.c, Version 1.0" and "RCS: two.c, Version 1.0". Verify this by opening your favorite hex editor[3] and see the three strings, starting at (for this example—your numbers may vary depending on compiler options) byte 0x000060E5 and running through 0x0000613C.
Now, when we create a new version of any of these source files, we modify the versioned string so as to keep track of the source versions as development continues:

```
/* two.c */
#include
static char rcsInfo[] = "RCS: two.c, Version 2.0";
void two()
{
printf("two called one more time\n");
}
```

And, again, compiling and examining the strings stored in the executable file shows that this version of main.exe used the 2.0 version of two.c when it was built.

Keeping track of these version strings will be something of a pain, however—every time a developer makes a modification to a source file, the version string needs to be modified. This is exactly the sort of painstaking detail that programmers (as a general rule) loathe and usually forget to obey. Fortunately, there's an easy way to keep the versioning information entirely accurate without requiring a single iota of effort on the developer's part.

Using VCS Keywords

With most source-code control systems, upon checkin (or merge), the source-code control tool can scan the source file, looking for particular keywords and expand them with versioning information. For example, Visual SourceSafe recognizes the "$Revsion: $" string as a keyword, where VSS will replace the string with the same string plus the actual VSS revision number for this checkin. So, for example, if we place the following static character string into a file "a.c",

```
static char rcsInfo[] = "$Revision: $";
every time "a.c" is checked back into VSS, the $Revision: $ string
will be updated to the
new VSS checkin version number:
static char rcsInfo[] = "$Revision: 4$";
```

By itself, the "$Revision: $" keyword does not give us all the information we need; specifically, we'd like the name of the file and a unique prefix to come in front, so that a string-scanning utility can find all of the occurrences of the version-embedded strings within the executable. So, we make use of a few other VSS keywords:

```
static char rcsInfo[] = "RCSInfo: $Logfile: $ $Revision: $";
```

Now, when this module is compiled into an executable, the string

```
"RCSInfo: $Logfile: a.c$ $Revision: 4$"
```

will be embedded inside of it. Each executable generated will have implicit tracking against the sources that were used to create it, and developers can find that information simply by examining the executable itself. Although other tools (like Davis' "WHATSIT" utility he describes in [1]) can be used, developers can, in a pinch, simply open the executable directly with Notepad or WordPad to find the strings by hand.

End-User Assistance

Developers won't be the only ones around to gather this information, however: the executable will often be in end-users' hands, away from the skilled eyes and hands of developers. Although tech-support staff could ask end-users to open the executable in Notepad, there's another, better, way.

Instead of forcing users to use an external utility to view the RCS informational strings, developers can bundle such a utility directly inside the executable itself, usually under some menu item under "Help|About" or something similar. The utility can open the executable directly, scan through the bytes looking for the rcsInfo strings, and display the results generated into a dialog box.
All in all, this is a pretty workable solution. Developers have the ability to verify which exact version is in use in the field or in Production, and this information can be communicated via tech support staff quickly and fairly easily. There's only one problem with this admittedly C++-centric discussion so far: it doesn't translate directly into Java.

When Laziness Hurts

To be certain, nothing prevents us from using the first part of the C/C++ system, that of creating a string in the class that contains the versioning information:

```
public class Main
{
public static final String RCSINFO = "RCS: $ Logfile: $ $Revsion:
$";
// . . .
}
```

Again, when Main.java is checked into VSS, the Revision keyword will have its contents expanded with the current version number. Within Java, we need to be a bit more specific, however; since the String is associated with a class, instead of a file, each and every class within a given .java file must have a similar String:

```
 // Main.java: Contains Main and a few other helper classes
//
class HelperOne
{
public static final String RCSINFO = "RCS: $ Logfile: $ $Revsion:
$";
// . . .
}
class HelperTwo
{
public static final String RCSINFO = "RCS: $ Logfile: $ $Revsion:
$";
// . . .
}
public class Main
{
// . . . as before . . .
}
```

Note that, in all likelihood, only top-level classes and/or inner classes of some importance would actually carry versioning strings—placing versioning strings on anonymous classes not only would be awkward, but retrieving the information from an anonymous class would be difficult. In fact, retrieving the RCSINFO string itself is going to be difficult in just about every incarnation.

Recall from [2], [3], [4] and other sources, that Java is a dynamically-loaded, lazy-loading system. This means that, unlike C and C++, classes are not loaded into the system unless directly referred to by the currently-executing code. Thus, given a normal Java deployment (a single .jar file), even though it would be possible to scan the .jar file for string constants, there's still no guarantee that those are the classes that are used at runtime[4]. What's needed is an ability to examine the classes loaded at runtime, to extract the version information at that point.

Unfortunately, this yields an even larger problem: there is no documented API to provide a list of all classes loaded via a particular ClassLoader. While it's certainly possible to create a custom ClassLoader to provide this information, it's impossible to make these modifications to the standard ClassLoaders used by Java2 when loading code from the CLASSPATH or Extensions directory[5].

There are two possible solutions to this problem: use a Java hack to slip past the access-protection layers and read data directly from the java.lang.ClassLoader class, or else require slightly more work on the developer's part through a third class.

Solution One: Hack-O-Rama

When writing a custom ClassLoader, you extend the java.lang.ClassLoader class. Any and all ClassLoaders (with the exception of one, which we'll discuss shortly) must extend this base class in

order to work correctly. Looking within the java.lang.ClassLoader class source[6], however, reveals an interesting tidbit:

```
/*
* The classes loaded by this class loader. The only purpose of
this
* table is to keep the classes from being GC'ed until the loader
* is GC'ed.
*/
private Vector classes = new Vector();
```

A few lines later, we run across the only place where this Vector is referenced from within code (the string "classes" shows up all over the place, but mostly in comments):

```
/*
* Called by the VM to record every loaded class with this loader.
*/
void addClass(Class c)
{
classes.addElement(c);
}
```

It's that last comment that truly intrigues us—"Called by the VM to record every loaded class with this loader." Verifying this requires more than a trivial amount of work[7], but reveals that this is, in fact, true—every class loaded by a ClassLoader-extending class ultimately is registered in this Vector.

Here, then, is the information we're looking for. Getting to it, however, requires work[8]. Because the Vector is marked private on the ClassLoader class, normally the JVM will not allow us to reference it, either from outside the class or from a derived class. Another approach is necessary.

Java provides such an approach, via the Reflection API[9]. By first obtaining a ClassLoader instance, then using Reflection to dig within it and retrieve that "classes" field, we can effectively bypass the Java access-protection mechanism and get a list of all the classes loaded by a particular ClassLoader:

```
import java.util.*;
public class ListAllLoadedClasses
{
public static Iterator list(ClassLoader CL)
throws NoSuchFieldException, IllegalAccessException
{
Class CL_class = CL.getClass();
while (CL_class != java.lang.ClassLoader.class)
{
CL_class = CL_class.getSuperclass();
}
java.lang.reflect.Field ClassLoader_classes_field =
CL_class.getDeclaredField("classes");
ClassLoader_classes_field.setAccessible(true);
Vector classes = (Vector )ClassLoader_classes_field.get(CL);
return classes.iterator();
}
public static void main (String args[])
throws Exception
{
ClassLoader myCL = ListAllLoadedClasses.class.getClassLoader ();
while (myCL != null)
{
System.out.println("ClassLoader: " + myCL);
for (Iterator iter = list(myCL); iter.hasNext(); )
{
System.out.println("\t" + iter.next());
}
myCL = myCL.getParent();
}
}
}
```

In this code, main() first obtains the ClassLoader instance used to load classes off of the CLASSPATH (called the AppClassLoader; see [4] for details), and passes it into list() to obtain an Iterator from the classes Vector stored in the java.lang.ClassLoader base. Once main() has printed out all the classes referenced by this Iterator, it then finds the parent ClassLoader to AppClassLoader and repeats the operation.

Within list(), we take the ClassLoader instance passed in, and walk the superclasses back until we get to java.lang.ClassLoader[10]. We use Reflection to find a field called "classes" on the java.lang.ClassLoader type definition, then use the get() method of the java.lang.reflect.Field class to retrieve the value of the classes field on the ClassLoader CL. This is the Vector we've been looking for; we cast the return value of get() to a Vector, call iterator() on it and return the result—an Iterator over the list of classes loaded by this ClassLoader.

Note that this code will not list any classes loaded by the bootstrap ClassLoader— because the bootstrap ClassLoader is represented within the JVM space as a null ClassLoader reference, any attempts to obtain the classes Vector will fail, because the reference we're attempting to dereference is null. (In all honesty, there's not even any assurance that the bootstrap ClassLoader even has the classes field buried within it—the bootstrap ClassLoader is implemented in native code for most VMs, anyway.)

Running this code directly ("java ListAllLoadedClasses" from the command-line) yields a list of exactly one class: ListAllLoadedClasses. As an experiment, add a few calls to other local "helper" classes to main() and re-run, just to verify that we do, in fact, list all classes loaded from the AppClassLoader:

```
// ListAllLoadedClasses.java
//
public class ListAllLoadedClasses
{
// . . . list() unchanged . . .
public static void main (String args[])
throws Exception
{
One one = new One();
Two two = new Two();
// . . . rest of main() unchanged . . .
}
}
// One.java
//
public class One
{
public One() { System.out.println("One constructed"); }
}
// Two.java
//
public class Two
{
public Two() { System.out.println("Two constructed"); }
}
C:\>java -classpath classes ListAllLoadedClasses
One constructed
Two constructed
ClassLoader: sun.misc.Launcher$AppClassLoader@404536
class ListAllLoadedClasses
class One
class Two
ClassLoader: sun.misc.Launcher$ExtClassLoader@7d8483
```

Now that we have the Class instances representing the classes loaded by this ClassLoader, obtaining the RCSINFO strings is another simple matter of Reflection:

```
// Given an Iterator from ListAllLoadedClasses.list(), find the
// RCSINFO String, if present
//
public String extractRCSInfo(Class c)
throws NoSuchFieldException, IllegalAccessException
{
java.lang.reflect.Field RCSINFO_field =
c.getDeclaredField("RCSINFO");
return (String)RCSINFO_field.get(null);
}
```

Note that if the RCSINFO field is not present on the class, a NoSuchFieldException will be thrown. In a production-quality implementation, extractRCSInfo should obtain the Field reference inside of a try/catch block, and return an empty String if the field is not present.

Solution Two: The Versions Class

Hacking this way leaves a bad taste in my mouth, personally. Just because we can do this doesn't necessarily mean we should, for reasons I'll enumerate later (see "Consequences", below). There is another approach, one which requires slightly more work on the part of developers, but doesn't require the use of Reflection to obtain the information desired.

The key actually comes from the Singleton pattern (see [6] for details). By using a Singleton to hold versioning information fed to it by each class as it is loaded, a single class becomes our central repository for versioning information. This class, which I call the Versions class, must be called from each loaded class, passing in the necessary version information:

```
import java.util.*;
public class Versions
{
private static Map classMap = new HashMap();
public static registerClass(Class c, String rcsInfo)
{
classMap.put(c, rcsInfo);
}
public static Iterator classes()
{
return classMap.keySet().iterator();
}
public static String classInfo(Class c)
{
return (String)classMap.get(c);
}
}
```

Using it is simple—any class that wishes to be registered must call Versions.registerClass immediately upon loading. Then, when version information is required, clients call Versions.classes() to get a list of all classes, and use Versions.classInfo() to get version information for each class registered:

```
// VersionOne.java
//
public class VersionOne
{
static
{
Versions.registerClass(VersionOne.class,
"$Logfile: $ $Revsion: $");
}
public VersionOne()
{
System.out.println("VersionOne constructed");
}
}
// VersionTwo.java
//
public class VersionTwo
{
static
{
Versions.registerClass(VersionTwo.class,
"$Logfile: $ $Revsion: $");
}
public VersionTwo()
{
System.out.println("VersionTwo constructed");
}
```

```
}
// VersionMain.java
//
public class VersionMain
{
static
{
Versions.registerClass(VersionMain.class,
"$Logfile: $ $Revsion: $");
}
public static void main (String args[])
{
System.out.println("VersionMain");
System.out.println("Built using:");
for (java.util.Iterator i = Versions.classes(); i.hasNext(); )
{
Class c = (Class)i.next();
System.out.println(c + " -- " + Versions.classInfo(c));
}
VersionOne v1 = new VersionOne();
VersionTwo v2 = new VersionTwo();
System.out.println("Built using:");
for (java.util.Iterator i = Versions.classes(); i.hasNext(); )
{
Class c = (Class)i.next();
System.out.println(c + " -- " + Versions.classInfo(c));
}
}
}
C:\> java –classpath classes VersionMain
VersionMain
Built using:
class VersionMain -- $Logfile: $ $Revsion: $
VersionOne constructed
VersionTwo constructed
Built using:
class VersionTwo -- $Logfile: $ $Revsion: $
class VersionOne -- $Logfile: $ $Revsion: $
class VersionMain -- $Logfile: $ $Revsion: $
```

Notice something very important about the output, however—in the VersionMain.main() method, the list of registered classes is browsed twice, with differing results. The first time, because the classes VersionOne and VersionTwo have not as yet been loaded, the Versions singleton knows nothing about them and therefore cannot provide any version information about them. Once VersionOne and VersionTwo have been loaded into the VM[11], then listing the Versions classes list reveals them.

Solution Three?

Another approach would be to avoid the code-level approach altogether, and simply embed the information about the versions used directly into a .jar file's manifest file. While attractive at some levels, the manifest file idea suffers from a few drawbacks:

- It requires more developer overhead. Instead of simply embedding a String into the class, or making a single call from within a static initializer block, developers must make sure that the versioning information is carried over into the manifest file at build-time. This is time-consuming.

- Versioning information is carried someplace other than where it will be used. In the two solutions discussed above, using the source-code control system's keywords works because the information is stored in the same file that generates the code. If that information needs to be moved someplace out of the source-controlled file, then the updating of that information becomes an order of magnitude more difficult.

- Not all code is deployed using a .jar file. More accurately, code deployed via a custom ClassLoader will be completely outside of the .jar file format, and therefore will have no manifest file in which to store versioning information.

Because of these drawbacks (most notably the second one), it's not truly feasible to consider the manifest file an ideal place for storing versioning information (for this scenario).

Consequences

There are a number of consequences to consider with either solution.
The first solution, using Reflection, works well with code to which no source is available, or code which cannot be modified. In this case, while the RCSINFO string may not be available, other information (such as the Package information obtained via the ClassLoader's getPackages() call) may be used instead.

However, the Reflection solution also carries with it a serious drawback—because it depends on using Reflection to make the code work, any code which uses it must have permission to carry out Reflection operations. Practically speaking, this means that code which uses the Reflection approach must have the java.lang.reflect.ReflectPermission enabled for its CodeSource. Since not all environments (most notably EJB and/or Servlet containers) will enable this permission by default, its usefulness in those environments is extremely limited.

Also, using Reflection to sniff out and retrieve private-declared fields is probably the worst possible use of the API. It creates an invisible dependency on the internals of ClassLoader, and will not trigger any sort of compiler error should Sun choose to modify the internal workings of the java.lang.ClassLoader implementation. Using the source code this way is certain to draw heat from Sun[12].

The second solution, by virtue of the fact that it has no dependencies on Reflection whatsoever, avoids the problems discussed above, with the drawback that it requires source-code modifications in order to work. This means that if a developer (perhaps that "new guy" on the team who's not quite as detail-oriented as you'd like) forgets to create the Versions.registerClass call in a static-initializer block, you'll have no versioning information for that class. Fortunately, this is a fairly easy syndrome to correct, either by creating a wizard[13] to generate the class boilerplate for you, or else by catching it via simple code-review upon first checkin of the file. The more serious situation is when you'd like to version code over which you have no source control—you can't add the static initializer block to register the class. In this situation, you have no choice.

Summary

Having the ability to list the source version used to build an executable is a valuable tool in diagnosing software bugs. If a developer can make sure that the version the customer is using is, in fact, the version he or she thinks it is, a large number of miscalculations and inaccurate assumptions can be removed from the debugging process.

Because of Java's lazy-loading approach, embedded strings within source files works only when the class containing the string is loaded into the VM. Ideally, Sun would provide the ability to obtain a list of all classes loaded via a particular ClassLoader, but no such API exists. As a result, either we must hack to find that list, or require classes to register themselves with a third party (in this case, the Versions class) to be listed.

Bibliography

[1] Stephen R. Davis, C++ Programmer's Companion, Addison-Wesley, 1993

[2] Bill Venners, Inside the Java2 Virtual Machine

[3] Sheng Liang and Gilad Bracha, "Dynamic Class Loading in the Java Virtual Machine" (presented at 13th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '98), Vancouver, BC, Canada, October, 1998).

[4] Ted Neward, Server-Based Java Programming, Manning Publications, 2000

[5] "Using the Boot Classpath", Ted Neward, 2000, available from javageeks.com

Endnotes

1. Some C/C++ compilers will issue a warning about rcsInfo being unreferenced within the .C file, and therefore attempt to optimize it away (thus reducing the code size). Users of this idiom are required to use whatever compiler means necessary—usually a #pragma warning directive of some sort—to prevent this optimization from taking place.
2. To be quite specific, on pp. 398 – 402 of C++ Programmer's Companion, he describes a macro-based system, which generates the unique strings into static char arrays, as described above. I'd repeat his system here, except that it depends somewhat on C/C++ preprocessor magic to work, and it would take too long to explain the preprocessor's quirks.
3. Even Notepad.exe or WordPad (write.exe) will serve in a pinch. Alternatively, use dumpbin.exe (from the Visual Studio command-line tools) with the "/rawdata" option.
4. For example, it's possible that the same class of the same package/class name exists within the Extensions directory, and would be loaded first, instead of the class within the .jar file. See [2], [3], and [4] for more details on the delegating nature of ClassLoaders in Java2.
5. OK, it's not impossible, but doing so would definitely place you squarely in the world of non-portable Java. See [5] for details on how to do it if you're truly interested.
6. Available in the src.jar file that ships with the JDK.
7. One way to verify this would be to create a custom ClassLoader that overrides the addClass() method and spits a line out to the console or a logfile on each classload, then cross-checking that list with the list of classes loaded by that ClassLoader. Another way would be to write a custom version of java.lang.ClassLoader to do the same, and use the tricks described in [5] to use this new custom version instead of the Java2 version. Either way, describing how to do this is beyond the scope of this paper; see [5] for guidelines on how to make this work.

8. It would always be possible to replace the java.lang.ClassLoader definition itself, but doing so would mean that the code would only work within JVMs with the customized ClassLoader definition. For purposes of this paper, I'm assuming such a restriction is unacceptable.
9. For more information on Reflection, see [2] and/or [4].
10. We could just use java.lang.ClassLoader.class to get this class, but fooling around with the type system at this level has so many subtle places where things could go wrong that this seems a slightly more robust approach.
11. Again, remember that Java is a lazy-loading system, so classes won't be loaded until absolutely necessary— see [2], [3], and [4].
12. (With sincere apologies to George Lucas) Remember, "A Jedi uses the Source only for knowledge and defense. Never for a hack."
13. See Gen<X>, for example, at http://www.develop.com/genx

Copyright