Instrumenting Java bytecode

Seminar work for the Compilers-course, spring 2005

Jari Aarniala
Department of Computer Science
University of Helsinki, Finland
jari.aarniala@cs.helsinki.fi

ABSTRACT

Bytecode instrumentation is a process where new functionality is added to a program by modifying the bytecode of a set of classes before they are loaded by the virtual machine. This paper will look into the details of bytecode instrumentation in Java: the tools, APIs and real-life applications.

Keywords

Java, bytecode, virtual machines, compilers, instrumentation, profiling

Legend

The following typographic conventions are used in this paper:

- Class and method names, code fragments and names of programs are written in the `typewriter font`.
- Proper names are **bolded**.
- Concepts and terms are *in italics.*

1. INTRODUCTION

The **Merriam-Webster online dictionary** [11] gives the following meaning to the verb *instrument*:

> "To equip with instruments especially for measuring and recording data"

In this paper, we'll explore the specifics of *Java bytecode instrumentation*, a process where compiled Java classes are modified to add new functionality to them, or to facilitate different measurements, as suggested by the definition above.

1.1 Why use bytecode instrumentation?

In the field of *aspect-oriented programming* (AOP) (see section 2.2), bytecode instrumentation is often used to implement *aspects* that are applied to a set of classes and contribute to a considerable amount of their functionality thereafter.

One might argue that the functionality introduced by instrumenting bytecode could just as well be implemented in the source code in the first place. However, bytecode instrumentation is often not about adding new functionality *per se*, but enhancing a program temporarily to trace its execution, gather profiling data, monitor memory usage etc. For example, a *profiler* tool will probably want to instrument the constructors of different classes to keep track of object creation within an application. Similarly, the *breakpoint* functionality of a *debugger* might be implemented by inserting custom bytecode into the classes that are under inspection.

The next section explores different applications of bytecode instrumentation in detail. The technical details and tools of bytecode instrumentation in Java are covered in section 3. Finally, for a case study of bytecode instrumentation in action, see section 4.

2. APPLICATION DOMAINS

Bytecode instrumentation is a technique that's applicable to solving problems in various application domains. As discussed in the previous section, it can be used to change the functionality of an application, but also to monitor one without altering its behaviour in any way. This chapter illustrates the different ways in which bytecode instrumentation can be used.

2.1 Monitoring

Monitoring applications, either in real-time or by generating reports after their execution has ended is one of the core application domains for bytecode instrumentation.

2.1.1 Example: Profiling

A typical application of bytecode instrumentation is *profiling*. A *profiler* can be used to examine the time a program spends in a particular method, the memory used by a certain data structure, to spot memory leaks etc. In its simplest form, profiling can be implemented by instrumenting a set of methods so that some action is taken by the profiling library whenever a thread enters or leaves a method.

JProfiler [10] is an example of a (commercial) Java profiler that utilizes bytecode instrumentation.

2.1.2 Example: Code coverage tools

When running automated tests (unit tests, for example) against an application, *code coverage* is often used as a metric for determining whether the tests are of good quality (i.e. that they exercise as much of the codebase as possible).

Cobertura [6] is an open source code coverage tool that uses bytecode instrumentation (via **ASM** [2], discussed in section 3.2) to monitor which lines of code are actually visited during test execution.

The architecture of Cobertura is inspected in detail in section 4 (starting on p. 6).

2.1.3 Example: Dynamic sequence diagrams

Bytecode instrumentation can also be used for illustrating the control flow of arbitrary Java applications. In an example application posted to his weblog [17] in January 2003, **Bob Lee** explains how he used bytecode instrumentation (via **jAdvise** [7]) and a UML sequence diagram generation library called **SEQUENCE** [13] to implement a utility that shows the sequence diagram for any Java application in realtime.

As Lee's tool relies solely on bytecode instrumentation, the tool is very flexible in that it doesn't need access to the application's source code at all to show how the control flow proceeds when the application is executed.

2.2 Aspect-oriented programming

Aspect-oriented programming (AOP) is a programming paradigm whose name was coined by the **AspectJ** [3] research team at the **Palo Alto Research Center** [12] circa 1995 [19].

A core problem that AOP is trying to solve is that modern object-oriented systems often have code that handles multiple *concerns* (along with the *core concern*, i.e. the business logic itself) on the same level of code. Typical examples of such concerns are security, logging and transaction handling. AOP tries to extract and separate these so-called *cross-cutting concerns* into reusable *aspects* that are implemented in one place and then used all around the codebase.

2.2.1 Example: Connection management

In an database-based application, the connection management code is often spread out to each method that needs to access the database. A typical (although simplified) example:

```
// connection management class
// (implementation details omitted)
public class Manager {

  // opens and returns a new
  // connection to the db
  public static Connection
     newConnection();

  // returns the active connection
  // to the db, if any
  public static Connection
     activeConnection();
}

// the class accessing the database
public class MyDatabaseClass {

  public void incrementSomeValue() {
    Connection c = Manager.newConnection();
    try {
      // arbitrary business logic..
      c.execute(...);
      c.delete(...);
      c.commit(); // all's well!
    } catch(Exception e) {
      // uh-oh, need to rollback
      c.rollback();
    } finally {
      // make sure the connection's closed
      c.close();
    }
  }

  public void doSomethingElse() {
    Connection c = Manager.newConnection();
    try {
      c.delete(...);
      c.execute(...);
      c.commit();
    } catch(Exception e) {
      // same cleanup procedure..
      c.rollback();
    } finally {
      // ..as in the previous method
      c.close();
    }
  }

  // .. other methods follow ..

}
```

As you can see, the connection management code is repeated throughout the class, making this code a good candidate for introducing an *aspect* for handling the repeated work in one, centralized place.

Using a popular bytecode instrumentation -based AOP framework, **AspectWerkz** [4], we can easily intercept the execution of the methods in `MyDatabaseClass` and add the connection management to each of these methods in a transparent manner. With the database aspect added, the code becomes more elegant and less repetitive, and thus easier to maintain:

```
public class MyDatabaseClass {

  public void incrementSomeValue() {
    c = Manager.newConnection();
    c.execute(...);
    c.delete(...);
    c.commit();
  }

  public void doSomethingElse() {
    c = Manager.newConnection();
    c.delete(...);
    c.execute(...);
    c.commit();
  }

  // .. other methods follow ..

}

@Aspect
public class MyDatabaseAspect {

  @Around execution(* MyDatabaseClass.*(*))
  public void handleConnection(JoinPoint jp) {
    try {
      jp.proceed();
    } catch(Exception e) {
      if(Manager.activeConnection() != null)
        Manager.acticeConnection().rollback();
    } finally {
      if(Manager.activeConnection() != null)
        Manager.acticeConnection().close();
    }
  }

}
```

Let's walk thru the `MyDatabaseAspect` class one step at a time:

- The first line, `@Aspect`, is an *annotation* (new feature in Java 1.5 that allows metadata to be attached to any class, method, field etc) that tells AspectWerkz that the class `MyDatabaseAspect` is an aspect class.
- The annotation `@Around` specifies that the following method, `handleConnection`, is an *around advice*, i.e. something that's to be executed around other method invocations
- The expression `execution(* MyDatabaseClass.*(*))` specifies that we want the around advice to be executed around all methods of `MyDatabaseClass`
- The parameter of type `JoinPoint` given to the `handleConnection` method represents a *join point* (i.e. a well-defined point of execution) in the code. The call to `jp.proceed()` resumes the method execution (of a method in `MyDatabaseClass`) our advice has just intercepted.
- The `proceed` -method is called inside a `try/catch/finally` -block so that any exceptions that occur when the methods of `MyDatabaseClass` access the database are handled as they were before the aspect was introduced, but this time in a transparent fashion by the advice we've implemented.

In practice, the bytecode of the `MyDatabaseClass` will be instrumented by AspectWerkz (which uses ASM [2] internally) on application startup so that our around advice is applied to the correct places in the code flow.

3. INSTRUMENTING JAVA BYTECODE

Compiled Java classes are stored in a intermediary *class file* format specified in [18]. A class file contains *bytecodes*, i.e. *instructions* that are interpreted by the Java *virtual machine* (JVM). Each instruction consists of a one-byte *opcode* followed by zero or more *operands.* The instructions manipulate the contents of the *operand stack.* For example, to multiply two integers, one would *push* the integers to the operand stack by using the instruction `iload` and then multiply them using the instruction `imul`, leaving the resulting integer on the top of the stack.

Although the binary class file format is well-defined and programming bytecode generation routines by hand is possible, there are several bytecode generation libraries available that can be used to ease the "heavy lifting" associated with instrumenting the bytecode of existing classes (as we will see in section 3.2).

3.1 Instrumenting bytecode: A primer

As an example, consider a simple method that calculates and returns the square of a given integer *n*.

```
public int square(int n) {
  return n * n;
}
```

The compiled bytecode of any Java class can be disassembled into a human-readable listing of *bytecode instructions* by using `javap`, a standard tool included in the Java development kit. Here's a commented listing of the disassembled bytecode for the body of the `square` method:

```
// push the second local variable, (numbering
// starts from zero, zero being the reference to
// 'this', our object instance), 'n', to the
// operand stack twice
0: iload_1
1: iload_1

// pop two integers ('n' and 'n') off the stack,
// multiply them and push the result to the stack
2: imul

// pop an integer (the multiplication result)
// off the stack and push it to the stack of
// the invoker of this method (i.e. exit the
// method with an integer result)
3: ireturn
```

A typical application of bytecode instrumentation is *monitoring.* For example, tracing the execution of a running program can be accomplished by inserting simple logging statements to a set of compiled classes (for more examples on monitoring, see section 2.1). Let's instrument the bytecode of our `square()` -method by adding a call to `System.out.println()` in the beginning of the method:

```
// push the reference to the static field 'out'
// of type 'java.io.PrintStream' of the
// class 'java.lang.System' to the stack
0: getstatic
   java/lang/System.out Ljava/io/PrintStream

// push the String constant "in Math.square()..."
// to the stack
3: ldc "in Math.square()..."

// pop the String off the top of the stack,
// invoke the method 'println' on the object
// 'System.out' with the String as a parameter
5: invokevirtual
   java/io/PrintStream.println(Ljava/lang/String;)V

// original instructions follow
...
```

After the instrumentation, the method `square` is functionally equivalent to the source code

```
public int square(int n) {
  // line added with bytecode instrumentation
  System.out.println("in Math.square()...");
  return n * n;
}
```

and thus invoking

```
for(int i = 0; i < 3; i++) square(i);
```

in client code yields the following results in the console:

```
in Math.square()...
in Math.square()...
in Math.square()...
```

3.2 Using ASM for bytecode instrumentation

In this section, we'll delve into the details of implementing bytecode instrumentation using **ASM** [2], a widely adopted Java bytecode manipulation framework.

Other libraries, such as **BCEL** [5] and **SERP** [14], are also available for bytecode instrumentation purposes, but what sets ASM aside from the others is its small size and good efficiency:

The ASM Jar library weighs in at 21kB, while BCEL requires 350kB and SERP 150kB of storage space.

Benchmarks [15] have shown that the overhead ASM adds when instrumenting classes (the contents of JDK's `tools.jar`, 1155 classes) on the fly is only 60%, while for BCEL the minimum overhead is 700% and for SERP even more, 1100%.

These differences are largely due to a core design decision of ASM: it uses the *visitor design pattern* [16] (as do other similar tools, such as BCEL and SERP), but instead of representing the class under instrumentation as a graph of objects, the details of a field (for example) are presented as the parameters of a call to the visitor method `visitField` of the interface `ClassVisitor` in the following fashion:

```
// called once for each field in the class
FieldVisitor visitField(int access,
                        String name,
                        String desc,
                        String signature,
                        Object value)
```

3.2.1 *ASM in action: A simple profiler*

To explore ASM's bytecode manipulation capabilities in action, we'll implement an overly simplified profiler tool. The profiler exposes the following API for gathering statistics on method calls made within an application:

```
package asm;
public class Profiler {

  // get the singleton profiler instance
  public static Profiler getInstance();

  // record a method call
  public void methodAccessed(String methodSpec);

  // print statistics on the console
  public void dumpStatistics();
}
```

For the profiling to work, we can use ASM to automatically instrument all non-abstract methods of all the classes in an application so that every method call is recorded by the profiler. For a method `void myMethod()` in the class `my_app.MyClass`, this would mean preceding the original bytecode of the method by a method call to

```
Profiler.getInstance()
  .methodAccessed("my_app/MyClass.myMethod()V");
```

which corresponds to the following set of virtual machine instructions:

```
// call getInstance, push the result
// (reference to a Profiler) to the stack
invokestatic
 asm/Profiler.getInstance()Lasm/Profiler;

// push the method name to the stack
ldc "my_app/MyClass.myMethod()V"

// call methodAccessed on the Profiler instance with
// the method name as the parameter
invokevirtual
 asm/Profiler.methodAccessed(Ljava/lang/String;)V
```

As mentioned in the previous section, the architecture of ASM is based on visitors. A *class visitor* (defined by the interface `ClassVisitor`) is used to visit the structure of a class: its fields, methods etc. `ClassWriter`, the class that is used to generate the bytecode, is also implemented as a class visitor: classes can be created from scratch by creating a `ClassWriter` instance and calling the various visitor methods on it.

However, in bytecode instrumentation the purpose is not to create new classes, but to modify existing ones instead. The ASM class `ClassReader` reads the bytecode of an existing class and calls the corresponding visitor methods on a class visitor. For example, when coming upon a method in the original bytecode, it calls the method `visitMethod(...)` on the class visitor given to it.

That is, while new classes can be created by calling the visitor methods by hand, a `ClassReader` instance can be used to call them in accordance to the bytecode of an existing class.

The following class, `ProfilerVisitor`, is a class visitor that implements the visitor method `visitMethod(...)` to add bytecode for the method call to `Profiler.methodAccessed(...)` in the beginning of every non-abstract method in a class:

```
package asm;
public class ProfilerVisitor
  extends ClassAdapter implements Opcodes {

  public MethodVisitor visitMethod(
    int access, String name, String desc,
    String signature, String[] exceptions) {

    // delegate the method call to the next
    // chained visitor
    MethodVisitor mv =
      cv.visitMethod(access, name, desc,
                     signature, exceptions);

    // only instrument non-abstract methods
    if((access & ACC_ABSTRACT) == 0) {

      // the following method calls
      // correspond to the three VM instructions
      // shown earlier in this section
      mv.visitMethodInsn(
        INVOKESTATIC,
        "asm/Profiler",
        "getInstance",
        "()Lasm/Profiler;"
      );

      mv.visitLdcInsn(
        className + "." + name + desc
      );

      mv.visitMethodInsn(
        INVOKEVIRTUAL,
        "asm/Profiler",
        "methodAccessed",
        "(Ljava/lang/String;)V"
      );
    }
    return mv;
  }

}
```

By chaining a `ClassReader` to our own class visitor, which is then chained to a `ClassWriter`, we can instrument classes on the fly in a high-performant fashion with a minimum amount of code. The following code snippet shows how it all fits together:

```
byte[] originalBytecode = ...;
ClassWriter writer = new ClassWriter(true);
ProfilerVisitor visitor =
  new ProfilerVisitor(writer);
ClassReader reader =
  new ClassReader(originalBytecode);
reader.accept(visitor, false);
byte[] instrumentedBytecode = writer.toByteArray();
```

After instrumenting and running an example application, a call to `Profiler.dumpStatistics()` yields the following results:

```
Method call statistics:
1. asm/app/Second.doSecondThings()V - 450 calls
2. asm/app/Second.<init>()V - 150 calls
3. asm/app/First.doFirstThings()V - 3 calls
4. asm/app/First.<init>()V - 3 calls
5. asm/app/Main.main([Ljava/lang/String;)V - 1 calls
```

3.3 Standard Java API for bytecode instrumentation

Available since the Java version 1.5.0, the package `java.lang.instrument` [9] defines a standard API for instrumenting the bytecode of classes before they're loaded by the virtual machine, as well as redefining the bytecode of classes that have already been loaded by the JVM.

Two interfaces in the package are of special interest:

Instrumentation provides a set of methods that allow custom *class transformers* to be registered with the JVM. It can also be used to inspect the set of classes loaded by the JVM and to measure the approximate storage space required for any specific object in the memory. Redefinition of already loaded classes is possible thru the method `redefineClasses(...)`. The implementation of this interface is provided by the Java runtime itself.

ClassFileTransformer is the interface that's implemented by classes that perform the actual instrumentation of bytecode. It specifies a single method, `transform(...)`, that's called on each of the registered class transformers whenever a class is being loaded by the JVM. Implementations of this interface are provided by the users of the API.

3.3.1 Using the API

In order to use the API, a special *agent* class must be implemented and registered with the JVM at runtime.

The agent class must specify a static method (much like the `main()` method seen in all command-line Java applications) with the following method signature:

```
public static void premain(String agentArgs,
Instrumentation inst);
```

The first parameter is an argument string passed to the agent via command-line (see below for details). The second parameter is the JVM-provided `Instrumentation` instance that's used to register class transformers.

The compiled agent class must be packaged into a `jar` archive. The name of the agent class is indicated in a special attribute called `Premain-Class` in the manifest of the agent's jar archive.

To enable the agent, one must specify a special command-line switch when invoking the `java` executable. It is of form

```
-javaagent:jarpath[=options]
```

where `jarpath` is the pathname of the agent jar archive and `options` is an optional configuration string that's passed to the `premain()` method of the agent class as the first parameter.

3.3.2 Putting it all together

To take advantage of the profiler functionality implemented in section 3.2.1, we can put together a simple instrumentation agent by implementing two classes: the agent class itself, and an implementation of the `ClassFileTransfromer` interface for hooking in the ASM code for doing the actual instrumentation.

```
package asm;
import java.lang.instrument.Instrumentation;
public class Agent {
  public static void premain(
    String args, Instrumentation inst) {
    inst.addTransformer(new ASMTransformer());
  }
}
...
public class ASMTransformer
  implements ClassFileTransformer {
  public byte[] transform(...) {
    ClassWriter writer = new ClassWriter(true);
    ProfilerVisitor visitor =
      new ProfilerVisitor(writer);
    ClassReader reader =
      new ClassReader(classfileBuffer);
    reader.accept(visitor, false);
    return writer.toByteArray();
  }
}
```

These two classes should be placed in a single Jar archive (say, `profiler.jar`) with the following two lines in the manifest file:

```
Premain-Class: asm.Agent
Boot-Class-Path: asm.jar
```

Enabling the agent for a specific application is then simply a matter of running the `java` interpreter in the following fashion:

```
java -javaagent:profiler.jar yourapp.MainClass
```

3.3.3 Key advantages of the API

Having a standardized instrumentation API has many advantages: for example, the AspectWerkz AOP framework [4] had to resort to other (less elegant) solutions[1] for bytecode instrumentation prior to the release of Java 1.5. These solutions include anything from *offline weaving* (where classes are instrumented in an extra phase after compilation) to *native hotswap* (where a native JVM extension module is used to hook in an instrumenting classloader). With a standardized API for instrumentation, these types of per-project solutions are no longer needed.

3.4 Options for bytecode instrumentation

There are other ways of implementing some of the features that the we've seen in the examples in this paper. However, for many applications (such as AOP frameworks), instrumentation is often the most viable solution.

For example, the Java version 1.3 introduced the concept of *dynamic proxy classes* [8]. Dynamic proxies can be used to intercept the invocation of methods and they can thus provide some of functionality offered by bytecode instrumentation. However, dynamic proxy classes only work on methods defined in *interfaces*, so they can't be used to intercept method invocations on concrete classes, let alone constructor calls.

4. CASE STUDY: COBERTURA

Up until now, we've been looking at the specifics of bytecode instrumentation using simplified examples of real-life solutions. In this section, we'll examine the inner workings of **Cobertura** [6], an open source Java code coverage tool.

Code coverage is a metric used to report how much of a specific codebase is being used when a program is run. Code coverage tools can be used to find *dead code* (i.e. code that is not used at all and can potentially be removed) and to see how well a set of automated tests exercise the codebase, for example.

4.1 Features

Cobertura functions both as a command line tool, and a *task* for the **Ant** [1] build tool. It operates by instrumenting a set of classes using ASM [2] in a separate step after compilation.

[1]See http://aspectwerkz.codehaus.org/weaving.html for a complete list of weaving solutions available in AspectWerkz

Cobertura keeps track of both the *line coverage* (i.e. execution of individual lines of code) and *branch coverage* (i.e. execution of conditional branches, such as if/else -clauses) of a program. It also calculates the **McCabe** *cyclomatic code complexity* [20] metric for the program. Both coverage metrics (line, branch) as well as the complexity metric are calculated for each class, package and the overall project. Cobertura creates coverage reports in HTML and XML formats.

4.2 Architecture

The architecture for Cobertura's class instrumentation and code coverage measurement functionality is quite simple and implemented in three central classes.

4.2.1 The CoverageData class

`CoverageData` keeps track of coverage data for a single class. It is used for both recording coverage details (for example, each instrumented line of code calls the method `touch(int line)` to report that the corresponding line was executed) and calculating coverage statistics afterwards (for example, it has methods for calculating the line coverage and branch coverage values for a single method or the whole class).

4.2.2 The ClassInstrumenter class

`ClassInstrumenter` extends `ClassAdapter` (a convenience implementation of the ASM interface `ClassVisitor` that provides basic implementations for each of the class visitor methods) and is the main driver of Cobertura's instrumentation process. It has two main functions:

- It marks each instrumented class with a marker interface (`HasBeenInstrumented`) so that no class is instrumented more than once.
- It overrides `visitMethod(...)` by returning a `MethodInstrumenter` object that's used to do method-level instrumentation.

4.2.3 The MethodInstrumenter class

`MethodInstrumenter` extends ASM's `MethodAdapter` class (and thus implements the `MethodVisitor` interface). It is the main workhorse of the instrumentation process in Cobertura by implementing the method-level bytecode manipulations in the following methods:

- `visitLineNumber(..)` is overridden so that each line of code in the instrumented method is registered with the class's `CoverageData` instance. Also, each of the lines is instrumented so that when the method is executed, `CoverageData.touch(lineNumber)` is called for each executed line.
- Both `visitJumpInsn(..)` and `visitLookupSwitchInsn(..)` are called when a *conditional* is found in the class under instrumentation. The implementation of these methods therefore marks the current line as a conditional with a call to `CoverageData.markLineAsConditional(currentLine)`

4.2.4 The reporting module

After Cobertura has finished instrumenting the classes of an application, running the application creates a serialized set of `CoverageData` objects on the local harddrive. An instance of the `CoverageReport` class brings each of these class-specific coverage statistic objects together and presents them in the format chosen by the user[2].

4.3 Advantages of using bytecode instrumentation

While Cobertura relies on bytecode instrumentation when gathering code coverage statistics, the instrumentation could also be done on the source code level. By working on the bytecode level directly, Cobertura doesn't need to compile the target application's source code, which can be cumbersome in larger projects with lots of dependencies. Also, instrumenting source code can actually be trickier than working with bytecode: libraries such as ASM provide a coherent view to the strictly defined bytecode format, while adding lines to a source file requires custom code for the parsing process.

5. CONCLUSION

Bytecode instrumentation is a widely-used technique with its share of pros and cons, as illustrated in this paper. It is an ideal solution for some scenarios, such as developing applications for monitoring and profiling purposes.

However, utilizing bytecode instrumentation is quite error-prone: although tools such as ASM make the actual process of bytecode generation straightforward, it is still quite easy to accidentally create invalid sequences of VM instructions which can lead to exotic class format exceptions at runtime. These kinds of errors are very hard to track down.

For regular development work, this is not an issue: developers don't usually work with bytecode generation libraries directly, but instead use higher-level tools that utilize instrumentation at a lower level.

While the learning curve is quite steep, bytecode instrumentation is a valuable technique in the toolbox of a developer working on frameworks and tools that require control over the execution of arbitrary Java applications.

[2]See http://cobertura.sourceforge.net/sample/ for a sample HTML report

6. REFERENCES

[1] *Apache Ant.* http://ant.apache.org/.

[2] *ASM Java bytecode manipulation framework.* http://asm.objectweb.org/.

[3] *AspectJ.* http://aspectj.org/.

[4] *AspectWerkz AOP framework.* http://aspectwerkz.codehaus.org/.

[5] *BCEL: The Byte Code Engineering Library.* http://jakarta.apache.org/bcel/.

[6] *Cobertura code coverage tool.* http://cobertura.sourceforge.net/.

[7] *jAdvise AOP framework.* http://crazybob.org/downloads/jadvise-javadoc/.

[8] *Java 1.3 Dynamic Proxy Classes.* http://java.sun.com/j2se/1.3/docs/guide/reflection/proxy.html.

[9] *Java 1.5.0 API documentation.* http://java.sun.com/j2se/1.5.0/docs/api/.

[10] *JProfiler.* http://www.ej-technologies.com/.

[11] *Merriam-Webster Online.* http://m-w.com/.

[12] *Palo Alto Research Center.* http://www.parc.com/.

[13] *SEQUENCE UML sequence diagram library.* http://www.zanthan.com/itymbi/archives/000596.html.

[14] *SERP.* http://serp.sourceforge.net/.

[15] E. Bruneton, R. Lenglet, and T. Coupaye. *ASM: a code manipulation tool to implement adaptable systems.* http://asm.objectweb.org/current/asm-eng.pdf.

[16] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns.* 1995.

[17] B. Lee. *Introducing jAdvise SEQUENCE.* http://crazybob.org/roller/page/crazybob/20030105.

[18] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification, Second Edition.* http://java.sun.com/docs/books/vmspec/.

[19] C. V. Lopes. *AOP: A Historical Perspective.* http://www.isr.uci.edu/tech_reports/UCI-ISR-02-5.pdf.

[20] T. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, 1976.