Building a Java Software Engineering Tool for Testing Java Applets

Aaron S. Binns & Gary McGraw

Reliable Software Technologies Corporation
21515 Ridgetop Circle, Suite 250
Sterling, VA 20166
phone: (703) 404-9293 fax: (703) 404-9295
asbinn@rstcorp.com gem@rstcorp.com
http://www.rstcorp.com

Abstract

As the Java programming environment matures and Java starts to be used to create "real" applications, programmers will require that sophisticated software engineering tools be co-opted from C and C++ development environments for use with Java. Software testing tools make up one important class of these software engineering tools. This paper describes some of the issues that we encountered while converting a C/C++ code coverage tool for use on Java code. Our successful conversion resulted in the first code coverage tool suitable for use in testing Java applets.

1. Introduction

Many proficient C and C++ programmers find coming up to speed in Java a fairly straightforward task. However, these programmers have become accustomed to having a large set of development tools at their fingertips. It is clear that making these tools available for Java is an important step in creating a solid Java development environment. Developers creating such tools are forced to grapple with some fundamental differences between Java and C/C++. In this short article, we will discuss a few of the differences between C/C++ and Java, and explain how tool-developers' coding-styles must be changed in order to overcome these differences. We are reporting knowledge that we gained during the development of a currently-available Java software engineering product.

One good thing about the Java programming language and environment is that many of the development tools that other commercial-grade languages have enjoyed can be adapted for use in Java. This is a direct result of the common ground that Java shares with C++. With this in mind, Reliable Software Technologies recently took on the task of enhancing a version of the popular C/C++ code coverage tool — PiSCES Coverage Tracker™ — so that it could operate on Java code.[1] Throughout the development of our Java coverage tool, we discovered that many of the standard techniques for performing dynamic analysis on C and C++ programs were not amenable to the Java programming environment. We will discuss our approach to some of these problems in this paper.

Before we go on, we must briefly introduce the concept of code coverage [Myers 1979]. The PiSCES Coverage Tracker for Java™ (Tracker) is a code coverage measurement tool. Given a set of test cases and the source code for a program (in the current case a Java applet or application), Tracker measures which parts of the source code are "exercised" during program execution on the test cases. Code coverage is meant to aid in the development of a thorough and rigorous set of test cases and helps a programmer ensure that every aspect of a piece of code is exercised during the testing phase. The C/C++ version of Tracker provides many sophisticated coverage measurement techniques: function, branch,

---

[1] The PiSCES Coverage Tracker™ and the PiSCES Coverage Tracker for Java™ are part of Reliable Software Technologies Corporation's ***PiSCES Software Analysis Toolkit*** ®.

condition/decision, and multiple condition coverages [RST 1995]. The Java Tracker prototype currently supports function and branch coverages. Function coverage measures which subset of defined functions have been exercised during execution over a given test set. In the Java case, this amounts to a coverage measurement over methods associated with a class. Function coverage is the most basic level of coverage. Branch coverage (also known as decision coverage) measures the number of explicit branches that are exercised during testing on a test set (e.g., which specific branches of a case statement have been tested). Branch coverage returns numbers that are relative to the total number of branches in the source code.

Coverage measures are achieved by a software tool that is inserted into the compilation pipeline. In this pipeline, a pre-processor is invoked on the source code. The original source code itself is left intact, but new code is inserted by the coverage tool. The inserted code performs the work of recording what pieces of the original program have been executed. This process is known as "code instrumentation". Coverage information is stored dynamically as the instrumented program is executed. For this reason, coverage measurement is known as a dynamic software analysis technique.

As we will more clearly spell-out in Section 3, we were forced to change the way coverage instrumentation is integrated into the compilation pipeline. This was a direct result of divergent build environments for Java and C/C++. But not only is the compile-time pipeline different, the run-time environment is different too.

The Java applet/application run-time environment is different in many critical ways from the usual C and C++ environments. The main difference has to do with the level on which the language allows access to the machine. C and C++ can access a machine at the "bare metal" level, meaning that many interesting programming tricks can be performed in order to improve tool performance. This is important in a competitive environment where the performance of a tool is critical to its utility and ultimately to its commercial success. For Java, on the other hand, access to the machine is severely limited (mostly because of security concerns). In this paper we will discuss the "insulated" nature of Java and point out some concrete examples of why this is a problem and how it may be overcome.

The same features that make Java so attractive to developers turn out to make creating software engineering tools problematic. For example, in C and C++, a tool can easily trap signals, start child processes, communicate with the operating system, etc. Nearly all of these kinds of low-level functions are restricted in Java, severely so for an applet running across a network. We have explored some possible solutions to these problems in the development of Tracker, but no clear-cut answers are available, and much work remains to be done.

We have been forced to address many problems head-on throughout the development of the PiSCES Coverage Tracker for Java™ (now available free on the net at `http://www.rstcorp.com/java.html`). As we do more Java development and craft solutions to these problems, more questions are raised and better solutions are discovered.

2. Tool Strength Depends on the Java Implementation

The source code instrumentation process for code coverage usually begins with the creation of a global data structure where coverage information can be kept. Because of the way that C and C++ memory management is set up, it is possible to determine the size and modularity of a piece of C/C++ code in advance. This allows a coverage tool to set up one table for all modules and components at once.

In Java it is impossible to tell the size and makeup of an applet or application in advance. This is due to the admittedly more modern dynamic memory allocation strategy of Java. Though this strategy allows programmers not to worry about memory allocation (a good thing), it does this by trading off a fair amount of Java programmers' power. This problem is compounded by the ability in Java to extend and modify already compiled classes by inheritance.

The usual C/C++ strategy of storing coverage results in a global table during dynamic execution of the instrumented code and ultimately dumping this table all at once on the completion of a run will not work for Java. Instead, information must either be written out during a run (see Section 4 for further discussion of this issue) or memory must be dynamically allocated as a data structure is forced to "grow itself" to contain more coverage information during a run. Neither of these alternatives is as efficient as filling in a completely allocated table (albeit sparsely in some cases). Thus this aspect of tool strength is closely related to the strength of the Java implementation — something that tool developers have no control over.

Java programmers are unable to get "down and dirty" and hack memory management strategies. This makes the job of "transparent" code instrumentation much harder. For software coverage tools to be useful, the instrumented code (once compiled) must not perform in a noticeably different way (considered along the runtime efficiency dimension) than the compiled version of the original code. If coverage instrumentation adversely impacts code performance, then the chances that a software developer will actually want to use the tool are greatly diminished.

It is primarily because of the "grease monkey" characteristics of programming in C/C++ (including direct memory management, register access, signal trapping, etc.) that C/C++ coverage tools can be written efficiently and thus remain "transparent". In Java this power has been taken away from the programmer. As a result, the ultimate strength of a Java software tool is more closely tied to the implementation of the Java interpreter than it is to the programming ability of the tool's creator. In a competitive market this is not necessarily a good thing.

Possible solutions to this problem include the following approaches. Put simply, the first approach is to learn more Java. There may be ways to extend Java's basic object type in order to more reasonably track coverage information. This could be done by grafting a data structure and/or some methods to the object class. Whether this could be done efficiently is an open question. A second possible approach would be to set up some sort of local-to-global identification scheme such that each object can be uniquely identified. The ability to pass object identities around would go a long way towards solving some of the problems that code coverage in particular raises.

As it currently exists, the Java Tracker associates coverage results with a particular Java source file. Requiring the source code in order to provide coverage information is reasonable since results browsing (see Figures 1 and 2) is greatly enhanced by association of coverage results with particular lines in the source.

As shown in Figures 1 and 2, the PiSCES Results Browser can be used to view the results in a GUI-based browser. The results are displayed in a hierarchical format, organized by method. The original source code, augmented with the coverage results can be viewed as well. Graphs of results can be generated and printed.

3. Dynamic Linking and Loading of Classes in Java

The usual approach to code coverage for C/C++ takes advantage of the C/C++ link phase in the compilation pipeline. This phase does not exist in the Java applet/application build environment. Once the individual .class files are built in Java, the build is complete. Unfortunately, nearly all C and C++ development tools that perform dynamic analysis need to record certain information about the program they are working on during the linking phase. In fact, many C/C++ analysis tools make a point of operating exclusively at the link phase (e.g., Pure Software's Purify and Pure Coverage tools) [Pure 1996].

Results Browser - tracker.cvr

File Browse Options Help

Browse Level (% Cov)	Branch Cov
tracker.cvr	42.70
ATMApp.java	42.70
ATMApp.init ( )	100.00
ATMApp.restart ( )	100.00
ATMApp.readFile ( )	51.72
Branch 122	T -
Case 124	H
Case 126	-
Case 128	H
Branch 130	T -
Branch 131	T -
Branch 133	T -
Branch 135	T -
Branch 137	T -
Branch 139	T -
Branch 141	T -
Branch 143	T -
Branch 145	T -
Branch 147	T -
Branch 151	T F
Branch 155	- -
ATMApp.checkCard (Integer cardno)	50.00
ATMApp.checkPin ( )	25.00
ATMApp.addDigit (int digit_in , int acc)	100.00
ATMApp.showBalance (int act)	0.00
ATMApp.showAmount (int action)	25.00
ATMApp.doAction (int act , float amt)	17.39
ATMApp.showString (String s)	100.00
ATMApp.appendString (String s)	100.00
ATMApp.CardPanel (Panel p)	100.00
ATMApp.ActPanel (Panel p)	100.00
ATMApp.NumPanel (Panel p)	100.00
ATMApp.handleEvent (Event evt)	41.94
Account.getCardnumber ( )	100.00
Account.getPin ( )	100.00
Account.checkChecking ( )	50.00
Account.checkSavings ( )	50.00

Figure 1: A screendump of the PiSCES Results Browser. The branch coverage results of a test of the ATM applet (<`http://www.rstcorp.com/atm.html`>) are shown organized by method. Particular results are linked directly to source-code lines as shown in Figure 2.

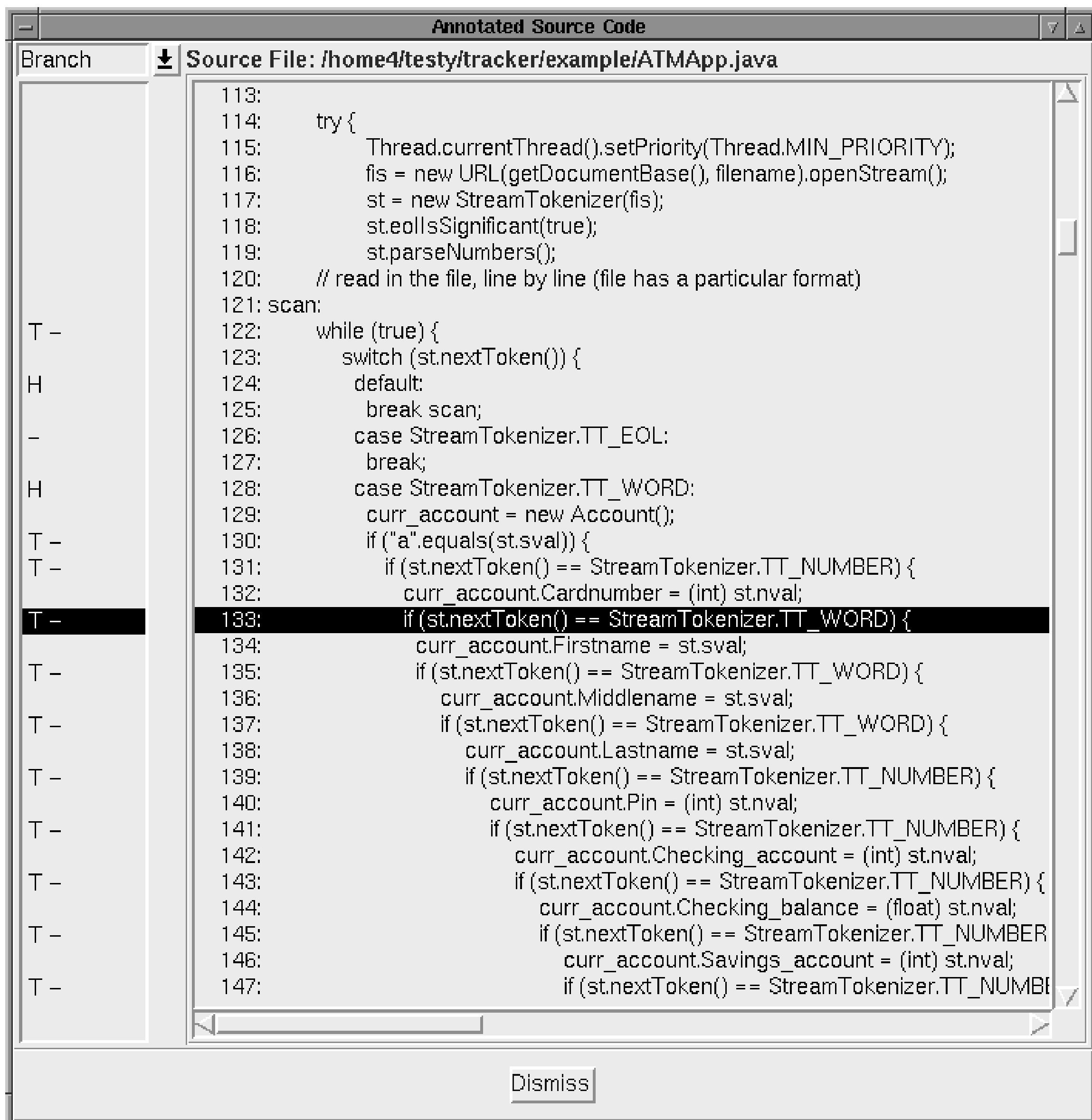

Figure 2: Coverage results (see Figure 1) are linked to lines of the original source code. This allows a developer to see which sections of the code have and have not been tested.

One aspect of linking-phase "housekeeping" includes building a global coverage table. (The impossibility of making such a table for Java is discussed above). Another "housekeeping" chore involves deciding what to name results files. In C/C++ it is easy to create a coverage file for each executable. For Java, no information other than the class name is available. This makes maintaining dates on results files and source files more complicated than it should be.

As an example of this problem, consider the following scenario. Suppose that a development team of two programmers is co-writing a Java applet in an Intranet-based environment. One programmer could make an instrumented version of the applet at some given time. Then that programmer could start collecting results. Meanwhile, suppose the other programmer makes some changes to the applet code and fails to compile the resulting applet with the coverage instrumentation. Anybody browsing the coverage results at

a later date may have no way of knowing whether the coverage results are up-to-date or not. The usual solution to this problem (correctly naming results files) is not possible in Java.

Some potential solutions to this problem do exist. For example explicitly listing all .class files used might help. Also, searching the `CLASSPATH` environment for all .class files might work. Or, as suggested above, giving every object a unique id (that might also include a time-stamp) in order to record run-time information may work. We have not yet determined which of these possible solutions might work best.

4. Web-based I/O Presents a Unique Challenge

In a web-based environment it often pays to be paranoid. Protecting data and other computational resources from exploitation by malicious users who may mount an attack on the system becomes a hard and very important problem. Because of the security measures that Java has in place, Java is a reasonably secure environment for Internet-networking. Though the existence of strong security is of paramount in Java and will be critical in its acceptance as a standard, it also plays a somewhat detrimental role by seriously impacting the power of the language. This becomes especially apparent in an Intranet situation in which it may pay to loosen security constraints.

One way to make this problem concrete is to consider testing and development in an Intranet environment where all users are considered trustworthy. (Though this may not necessarily be the case in all organizations, it will certainly hold in some.) At the heart of this problem lies the fact that Java makes no distinction between applets intended for Intranet use and applets intended for Internet use (the difference being, of course, that one network is "in house" and the other is "connected to the world"). How might testing in an Intranet environment be performed? How might two programmers on two different machines of the same Intranet test an applet?

Java presents a host of problems for the remote execution of applets. The worse case is that of an applet running on a website page. For security reasons, an applet of this sort is not allowed to write files on the machine that served it up. (Reading a file from the "serving" machine can be done through a URL with the command `URL(getDocumentBase(), <filename>).openStream()`.) The fact that it is impossible to write back to an applet-serving host makes doing things like tracking coverage information difficult.

For example, suppose an applet has been created, instrumented for coverage, and installed on a webpage. Whenever this applet runs, there is the potential for updating coverage information based on use of the applet. This may be interesting for a variety of reasons, not the least of which may be "feature-tuning" an applet based on real-world usage statistics. Unfortunately, an applet running over the net cannot write to a file. This means that any coverage information that might be gathered when an applet runs across the net (by being executed on a remote machines) cannot be collected at the applet developers site.

This problem shows up in various guises in all kinds of Java applications. Much bandwidth in `comp.lang.java` has been devoted to discussing this issue and various ways to work around it. The most alarming workaround (involving the Netscape browser in particular), is hacking a few bytes of information in the Netscape executable so that it can write remote files [Back 1996]. This "solution" calls into question the actual security of Java applets versus the perceived security. Also see [Dean 1996] for related Java/Netscape security concerns.

A less horrible workaround would be to write and run a daemon on the remote (applet developer's) machine that accepts input from across the net and is able to merge it into a global results file. Another is to use electronic mail as a means of communication. Many people are thinking about solutions to the I/O issue (especially O). It will be interesting to see what emerges.

The most obvious solution to this problem is to do testing on one "local" filesystem. As it stands now, the Java Tracker must be run under the appletviewer in order to be tested for coverage. We have yet to grapple with performing coverage testing over the web.

5. Taking Over a Java Environment

In C and C++ it is easy to trap signals, capture exit calls and basically interpose a program between some original code and the actual machine. Interrupting code execution pre-crash is fairly straightforward. This makes writing debuggers and related code analysis tools for C/C++ somewhat non-mysterious.

In Java this is not as straightforward. Because Sun Microsystems has chosen not to release the debugger code, it is far less clear how such low-level interposition might be done in Java. It is our impression that the source to the Java debugger and Java compiler can be obtained from Sun, but only after signing an expensive royalty agreement. The debugger source is not freely available as part of the JDK 1.0.

Java's important threading capability, and the way that Java handles exceptions make this all the more difficult. For example, what can be considered a "basic block" of code if exceptions allow control to pop around schizophrenically? We have only scratched the surface of these questions throughout the development of the Java Tracker. Much work remains to be done.

6. Conclusions

As the Java programming environment matures, we expect many sophisticated software engineering tools to become available. The experience we have gained in converting our C/C++ code coverage tool to Java can be used to help devise a general strategy for tool conversion. Although this paper may have raised more questions than it has answered, our preliminary approach resulted in encouraging concrete results (in the form of a working code coverage tool for Java). The fruits of our effort — the PiSCES Coverage Tracker for Java ™ — are available free of charge on the Web at URL: `<http://www.rstcorp.com/java-trac.html>`. We encourage all Java developers to devise a reasonable and thorough testing strategy before releasing their applets. The Java Tracker should prove useful to such efforts.

References

[Back 1996] Back, G. (1996) How to Bypass Netscape's Security Manager. A web document at URL `<http://www.cs.utah.edu/~gback/netscape/bypass.html>`. Posted to comp.lang.java 1/29/96.

[Dean 1996] Dean, D., Felton, E., and Wallach, D. (1996) Java Security: From HotJava to Netscape and Beyond. A web document at URL `<http://www.cs.princeton.edu/~ddean/java>`. Also see the related CERT Alert document CA-96.05.

[Myers 1979] Myers, G. (1979) The Art of Software Testing. Wiley.

[Pure 1996] Pure Software. (1996) PureCoverage Data Sheet. A web document at URL `<http://www.pure.com/products/purecoverage/PCdatasheet.html>`.

[RST 1995] Reliable Software Technologies Corp. (1995) White paper: PiSCES Coverage Tracker TechnicalBrief. A web document at URL `<http://www.rstcorp.com/techbrf.html>`.