

CC4J – Code Coverage for Java

A Load-Time Adaptation Success Story

Günter Kniesel¹ and Michael Austermann²

¹ University of Bonn, Institute of Computer Science III,
Römerstr. 164, D-53117 Bonn, Germany,
gk@cs.uni-bonn.de

² SCOOP Software GmbH, Am Kielshof 29, D-51105 Köln, Germany,
maustermann@scoop-gmbh.de

Abstract. Code coverage and tracing are extremely important for quality assurance in large scale software projects. When Java components are required to be deployed in distributed dynamic environments, e.g. as a part of an application server, load-time adaptation is the only practicable method capable of instrumenting these facilities. Load-time adaptation is, however, a relatively new technology whose scalability in industrial strength projects is so far unproven.

This paper reports on the development of a quality assurance tool, CC4J, which has been implemented using novel load-time adaptation techniques. Our evaluation, performed in the context of a large-scale, deployed, Java software project, shows that this is a resoundingly successful approach. The system's core has been implemented by one person in less than two weeks. Within less than four weeks after its first use CC4J was adopted by the entire project and the quality assurance department recommends adopting the tool in other projects.

1 Introduction

The work reported in this paper has been carried out in a large scale software project. The goal of the project is the development of a distributed system in the domain of electronic payment transactions, known for its extremely high quality and safety-requirements. These are reflected in the contract with the customer, which imposes, as a key quality measure, 100% *code coverage* during testing. This means that all unit tests together must exercise every line of code. Another important quality assurance aspect is the *tracing* of system activity at different levels of detail. This makes the overall flow of control comprehensible and verifiable during functional testing *and* deployment.

Code coverage and tracing functionality is needed in different combinations in different versions of the system. For instance, realistic performance tests require that no code coverage data is collected but tracing is still enabled at the same level as in the final system. Therefore, different versions of the functional components, instrumented for different purposes, are required. In general, the number of possible variants grows exponentially with the number of different

available adaptations. Generating these statically would be prohibitive. Even worse, many of these variants might be used just once since after the test had been performed, changes in the code would automatically render them invalid.

For these reasons, we need means of instrumenting the code base “on the fly” depending on the current suite of tests and the current quality assurance measure (code coverage, tracing at different levels, etc.).

The system is currently being developed by a team of more than 100 software engineers at T-Systems¹. Its component-based J2EE architecture makes it a good candidate for the evaluation of component adaptation techniques in a realistic environment.

Summary In the context of component-based development load-time adaptation of byte code can be regarded as a key technology. Components are delivered in binary format – so *adaptation of byte code* is ultimately required. This can be performed statically or dynamically, at load-time. Static adaptation has the advantage of adding no load-time penalty to a program’s execution. However, static instrumentation might become impractical if many different adaptation variants are to be managed. Moreover, it is inapplicable if dynamic component loading is possible and the name of dynamically loaded classes is determined at run-time, via reflection. Then the only point where it is feasible to determine and adapt all components that are actually used by a program is *during the class loading process*.

The paper is structured as follows. In section 2 we review the state of the art of code coverage and tracing solutions. In section 3, we give a short introduction to JMangler, the employed tool for load-time adaptation of Java class files. In section 4 we introduce the developed code coverage tool. Its implementation by load-time adaptation is described in section 5. The experience from the deployment of CC4J in the project is reported in section 6. Section 7 comments on the related work before concluding in section 8.

2 State of the Art

It is common practice to manually insert log statements into code. However, logging the execution of *every line of code* this way is prohibitively expensive and prone to common programming errors. In addition, the temporary nature of unit tests prohibits the integration of test code and related logging statements into the code that is being tested because in the deployed code bases all test code should be removed. On the other hand, unit tests contained in separate modules cannot be aware of every line of code that they test nor can they report on individual lines of code. In other words, there is no alternative but to keep instrumentation separate from the underlying system. This is particularly the case in the context of components, whose source code is unavailable.

In the following we first state the practical requirements of the project and then review the three categories of basic approaches that might be applicable:

¹ T-Systems is the information technology division of Deutsche Telekom.

- commercial off-the-shelf systems
- aspect-oriented languages
- aspect-oriented tools

2.1 Application Requirements

The following specific requirements were the main criteria for choosing one particular solution to be applied in the project.

Platform Independence. The system must run on any Java 1.3 compliant virtual machine, at least on Windows NT/2000 and SUN Solaris platforms.

Arbitrary Modifications. Implementing tracing and logging requires the ability to add new helper classes, extend existing classes by new methods and fields and – most importantly – to modify the *byte code* of existing methods.

General Applicability. The adaptation tool must be applicable to any legal Java program.

Component Based Architecture. Tracing and logging are orthogonal aspects of adaptation which should be implemented as independent components.

Easy Configuration. Application of different adaptations to the same classes should be possible without any changes in program code.

2.2 Commercial Off-the-Shelf Tools

The need for a solution available at short notice suggested an evaluation of commercial off-the-shelf tools in the first place. However, the choice of code coverage tools for Java is quite limited, with *Rational PureCoverage* and *Tangent JProbe Coverage* being the only realistic alternatives. Rational's tool is inapplicable, because it only supports Java applications on Windows platforms. JProbe Coverage is platform-independent and available as part of the JProbe Suite.

Neither tool provides tracing support. Even if they had done so, the related licence fees would still have been prohibitive given that tracing must also be available during deployment (in thousands of installed devices). We know of no other commercial tools that support incremental automatic instrumentation of code for tracing purposes.

Obviously, existing tools are unsuitable in the context of our project and the only way to achieve our goals was to implement our own load-time adaptation applications using a suitable language or tool.

2.3 Aspect-Oriented Languages

Novel aspect-oriented programming language extensions for Java [Asp01] are platform independent and enable modifications of existing method code. However, their high abstraction level does not allow them to refer to concepts such as the individual lines or statements of a program. Without this ability, we cannot insert instrumentation code that logs the execution of control flow statements (if, while, ...).

2.4 Aspect-Oriented Tools

The required level of detail can be addressed by aspect-oriented tools that support a lower level of abstraction. In particular, tools for the load-time transformation of Java class files come to be regarded. We are aware of only four approaches that go beyond the mere representation of Java class files by providing complete solutions for the integration into the class loading and linking process of the Java platform:

- Binary Component Adaptation [KH98]
- Java Object Instrumentation Environment [CCK98]
- Javassist [Chi00]
- JMangler [KCA01], [Aus00]

A detailed comparison of these tools can be found in [KCA01]. It shows that only JMangler is applicable in the context of the requirements listed above. In particular,

- BCA has been integrated into the implementation of the Java Virtual Machine of JDK 1.1 for Solaris, and therefore cannot be used with other JVMs. Furthermore, it does not allow adaptation at the level of individual statements.
- JOIE and Javassist cannot adapt applications that employ their own class loader, thus violating the general applicability requirement.

JMangler was therefore the obvious candidate for our evaluation. An introduction to JMangler is given in the next section.

3 JMangler

*JMangler*² is a Java framework for transformation of class files at load-time. Programmers can write their own *transformer components* that analyse the classes on target and decide which concrete transformations are to be carried out. Multiple transformer components, or simply *transformers*, can be deployed simultaneously. JMangler provides the ability to combine their transformations and to perform these transformations on all classes of a program (Figure 1). All transformations that respect binary compatibility [GJSB00] are supported, including arbitrary modifications of method bodies.

JMangler plugs neatly into any Java 1.3 platform being able to run on any compliant JVM and to work with any legal Java program. It is configured by an XML file that includes information on the actual transformer components that are to be applied.

In the following sections we outline JMangler's basic concepts, and describe how JMangler is integrated into the Java platform. For more detailed descriptions we refer to [KCA01], [Aus00]

² See <http://javalab.cs.uni-bonn.de/research/jmangler/>

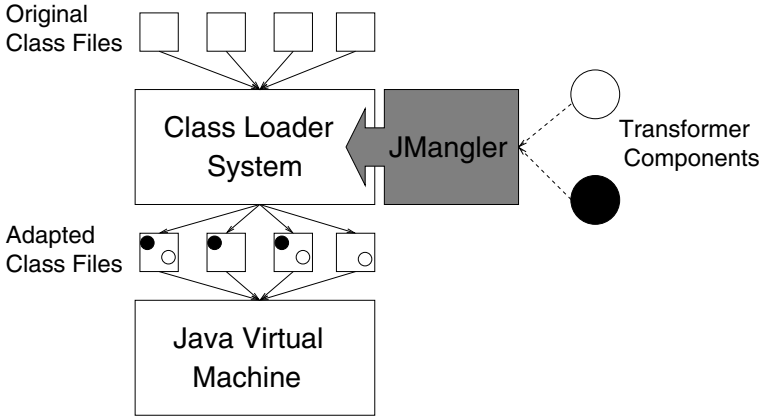


Fig. 1. Architecture of the JMangler Framework

3.1 Basics

JMangler supports all transformation of class files that do not violate binary compatibility [LY99]. In particular, it supports:

- addition of classes, interfaces, fields and methods;
- changes to a method's `throws` clause;
- changes to a class's `extends` clause that do not reduce the set of direct and indirect superclasses;
- changes to a class's `implements` clause that do not reduce the set of direct and indirect superinterfaces;
- the addition and modification of annotations that respect binary compatibility;
- changes to method code.

All transformations mentioned in the first five items of this list are called *interface transformations*. The addition of a method including an initial method body is still regarded as an interface transformation. Changes to method code are called *code transformations*.

Transformers are Java classes that implement specific interfaces (**InterfaceTransformer** and **CodeTransformer**). Implementation of these interfaces can be performed using JMangler's API. It supports three types of operations:

- analysis of class files, in order to determine whether a specific transformation is applicable,
- interface transformations and
- code transformations.

A transformer component that implements the operations of the **InterfaceTransformer** interface can perform one or many related interface transformations. The same is true for code transformations. A transformer can play both roles by implementing both interfaces. Thus it is possible for one component to provide a consistent set of related interface and code transformations.

3.2 Composition of Transformers

JMangler enables composition of independently developed transformers in the sense that multiple transformers can be jointly applied to the same program. A user who wants to transform a program at load-time can specify this easily in a configuration file. This file has a simple XML-based syntax describing:

- the set of interface and code transformers to be applied;
- parameters to be passed to the transformers;
- the ordering of code transformers;
- and some other options (debugging, etc.).

Different application-specific transformers can be easily composed from the same set of basic transformers. Each composition specification can be stored in a different XML file. Switching between different configurations simply requires providing a different file name as a parameter to the invocation of JMangler:

```
jmangler <configFile> <main> <parameters>
```

This invocation starts the JVM, loads JMangler and the transformers specified in the configuration file and then initiates execution of the program to be adapted.

In the context of the electronic payment transactions project, this open architecture makes it possible to develop transformers for code coverage and tracing separately. Further transformers, even from third parties, can be integrated later, when needed. This approach protects the coding resources already invested while still providing options for further code evolution and extension.

Last, but not least, JMangler is freely available under the terms of the GNU LGPL (<http://www.gnu.org/copyleft/lesser.html>), which explicitly allows for the development of commercial applications.

4 CC4J

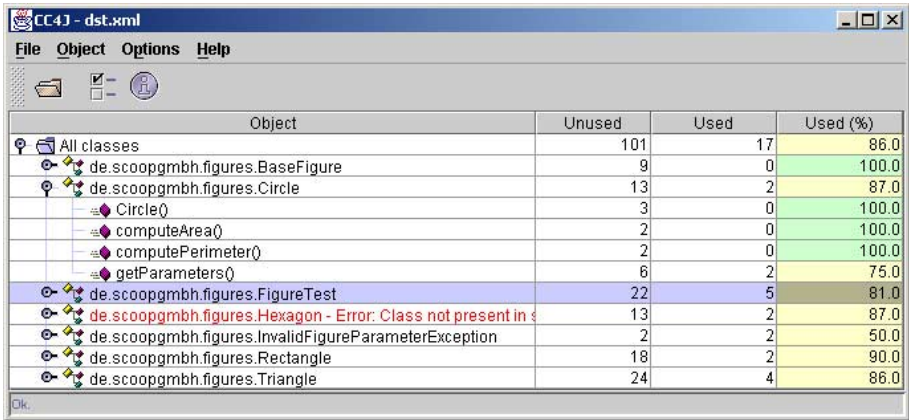
It is not the purpose of this paper to describe all the applications that have been developed using JMangler but to focus on one representative case that proves the applicability of load-time transformation in a commercial context. Therefore we will concentrate henceforth on one application, Code Coverage for Java (CC4J).

CC4J's main responsibility is to determine which lines of code have or have not been executed during one run of an application. *We define, that a line of code has been executed, if the flow of control has reached the line at least once.* It does not matter if the execution has failed to complete successfully or if an exception has been raised.

CC4J consists of the following subcomponents:

Core. The CC4J backend is implemented as a load time transformer. It instruments classes to collect coverage data for one test run. The code coverage data is stored in a *Code Coverage File (CCF)*.

Merger. This component combines the CCFs of different test runs into one file in the same format. This is needed to gain an overview of the total coverage achieved by all test runs.



Object	Unused	Used	Used (%)
All classes	101	17	86.0
de.scoopgmbh.figures.BaseFigure	9	0	100.0
de.scoopgmbh.figures.Circle	13	2	87.0
-circle()	3	0	100.0
computeArea()	2	0	100.0
computePerimeter()	2	0	100.0
getParameters()	6	2	75.0
de.scoopgmbh.figures.FigureTest	22	5	81.0
de.scoopgmbh.figures.Hexagon - Error: Class not present in s	13	2	87.0
de.scoopgmbh.figures.InvalidFigureParameterException	2	2	50.0
de.scoopgmbh.figures.Rectangle	18	2	90.0
de.scoopgmbh.figures.Triangle	24	4	86.0

Fig. 2. Screenshot of the CC4J GUI

Report Generator. The report generator transforms CCFs into human-readable reports (for instance, in HTML format for the project intranet or as postscript for printing).

GUI. The graphical user interface (Figure 2) visualizes code coverage and highlights unexecuted lines in the sourcecode.

These components and their interactions are depicted in Figure 3. By default, the core collects coverage data for every loaded class. Collection of coverage data for test classes can be prevented by specifying them in an exclusion list. This is done in the configuration file which is passed as parameter to the invocation of CC4J.

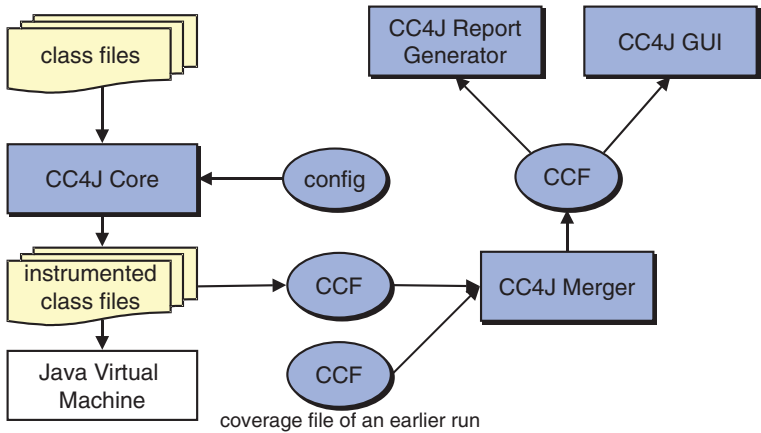


Fig. 3. Architecture of CC4J

For the code coverage files, CC4J uses a simple XML based format, which can be further processed by standard tools. This promotes easy extensibility. Appendix B shows the coverage file generated by CC4J when executing the ubiquitous “Hello World” application (Appendix A).

5 Implementation Using JMangler

Taking advantage of JMangler’s component architecture, CC4J has been realised in the form of implemented as a combined interface and code transformer component (Figure 4). In the following, we will focus on two main aspects of its implementation:

- Determination of log points
- Efficient storage of logging data

5.1 Determination of Logging Points

Since CC4J adapts classes at load-time, one has to determine where to insert logging code into the *byte code sequence* of methods. Those points within byte code sequences at which logging data has to be collected, we name *logging points*.

Fortunately Java’s class file format [LY99] helps in determining logging points. By default, Java compilers generate line number and source file information for each class. They store it in the *source file attribute* and *line number table attributes* of class files. This information is intended to be used by debuggers for mapping a method’s byte code to the corresponding lines of code in the source file (Figure 5).

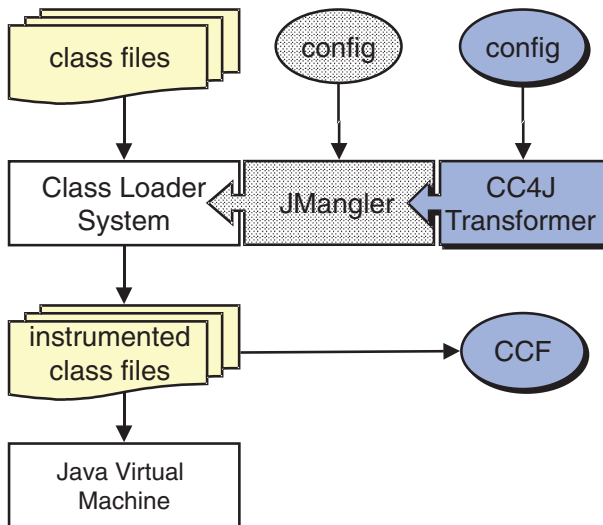


Fig. 4. Implementation of CC4J with JMangler

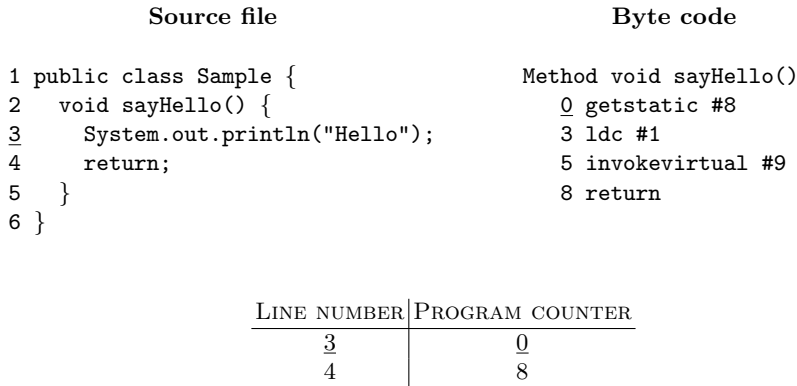


Fig. 5. Method (left) that prints “Hello”, with its byte code (right) and line number table (bottom).

For the determination of logging points the CC4J transformer analyses the line number table of each method in a class file. The logging point for a given (source code) line number is the corresponding program counter value in the line number table. For instance, in Figure 5, the logging point for line 3 is at program counter value 0.

Discussion. Since line number information is generated by default, the solution described above is sufficient in most cases. In the project for which CC4J has been developed initially, it suffices always. Still, one could argue that the line number information might not be available if explicitly turned off. Determining code coverage in such cases requires a transformer component that logs the execution of every byte code instruction or statement. Detecting individual source code statements in byte code is a more complex task than the line number based logging, which could be realized if required.

One could also ask why it is necessary to instrument every statement if the goal is to test whether all statements have been executed. Consider a basic block containing several statements. If control reaches the last statement, then all previous statements must have been executed. Control can exit in the middle of a basic block only if an exception is thrown. It might appear that a big improvement in performance could have been obtained by using this property. However, to determine whether an exception has occurred each basic block has to be wrapped with a try-catch statement. We implemented and compared both approaches and found out that the apparent optimization results in a more than double increase of execution time. This is because the inserted try-catch statement inhibits standard optimisations of the HotSpot JVM.

5.2 Efficient Storage of Logging Data

At each logging point some logging data has to be stored. The overall runtime overhead of determining code coverage is almost exclusively determined by the costs of storing the logging data. Therefore, this task must be highly optimized to keep the runtime overhead small.

One option is to provide a central logging object. Calls to this object could result either in immediate writing to a log file or could be buffered in a data structure, e.g. a hashtable, for the subsequent writing of the log file. This approach has the advantage of simplicity and makes it easy to change the policy governing the writing of the log file. However, it has the disadvantage of adding one avoidable level of indirection to every logging operation. This would affect every line of code to be executed.

Another possibility is to let every class manage its logging data itself. For this distributed approach, every class is adapted to contain a buffer whose size corresponds exactly to the number of logging points for methods of this class. Logging can access this data structure without any indirection. Every log point writes to exactly one entry in this buffer.

CC4J is implemented using the distributed approach because it saves storage and minimises the increase in running time.

The logging data is saved to secondary storage in CCF format when the virtual machine begins its shutdown sequence. This avoids the high costs of continuous output to secondary storage and also prevents long-running applications from writing endless log files with mostly redundant entries.

For environments that are extremely constrained in memory we also experimented with a data structure that reduces the memory costs to the 32nd part. However, this results in increased run-time costs since thread synchronisation, required to prevent race conditions, is involved in the execution of every line of code.

6 Evaluation

When the suggestion to implement code coverage and tracing by load-time transformation of Java byte code was first put forward in the project, it received much skepticism. It was doubted that a novel technique and its implementation as a recently available research tool could be applied successfully in an industrial environment. In particular, the responsible project officers required proof of:

- feasibility,
- timeliness of delivery,
- cost-effectiveness,
- smooth integration into the development cycle and
- efficiency.

To this end, it was decided to create CC4J as a proof of concept on condition that it had to be done with minimal resources (in terms of manpower). In the

following, we report on the required manpower (which determines the timeliness of delivery and cost-effectiveness), performance measurements, and the current use of CC4J within the project development cycle.

6.1 Implementation Effort

The system's core, consisting of the JMangler transformer component described above, has been implemented by one person in less than two weeks.

6.2 Application

Evaluation of CC4J in the project was first conducted by a single programmer, who used the tool to perform automated nightly unit tests of his code including full coverage data collection. At the end of all test runs, the collected coverage data was automatically merged. The results were inspected with the CC4J GUI on the following day to identify additional tests that would include previously uncovered code.

This approach was so successful that it was adopted by the entire project within less than four weeks after the first use of CC4J. Currently, the CC4J suite is being deployed as the official code coverage tool of the project for all developed Java code, including components that run within application servers. The quality assurance department of T-Systems recommends adopting the tool in other projects.

6.3 Run-Time Performance

The run-time impact of class file instrumentation is illustrated by an example employing SUN's Java compiler and different project subsystems running on Oracle's IAS Application Server.

Javac. The `javac` compiler contained in the JDK 1.3 consists of 143 classes that compile to 582 KB of byte code. We measured the performance of `javac` when compiling itself on an AMD Athlon 1GHz processor with 384 MB of main memory, running under Windows 2000. During the measurement no other user processes were active.

The non-instrumented version of `javac` compiled its own source within 3.2 seconds. Enabling the collection of code coverage data, the compilation time was 7.3 seconds, which is an increase of 128%. This includes the overhead resulting from the load-time adaptation (in this case the determination and insertion of logging points).

To determine the overhead resulting exclusively from the collection of code coverage data, we used a JMangler option to dump the transformed classes. Executing these dumped classes, it is possible to measure the pure run-time overhead without the influence of the adaptation process itself. For these dumped – already adapted – classes the compilation time was 4.0 seconds. This is an increase of

just 25%, which is remarkable, given that the instrumented application executes additional instructions for *every line of the original code*.

Another remarkable experience was that it is more important to generate code that does not inhibit standard optimisations of the HotSpot JVM than to reduce the number of inserted instructions (see 5.1).

Applications on Oracle IAS. The results reported above were compared to a typical regression testing scenario of the project. We tested a subsystem with 412 classes that compiled to 1,3 MB of byte code. Automatic unit testing of the whole system took 503 seconds without instrumentation and 887 seconds with CC4J. This is an increase of 76%.

We were also interested in the influence of CC4J on I/O-intensive server applications and found out that it is negligible. There was no measurable difference in the throughput of instrumented applications compared to the original ones.

The relatively low impact on run-time performance contributed to the acceptance of CC4J and to its fast adoption in the project.

7 Related Work

In addition to the tools and technologies discussed in section 2 there are several existing and emerging Java APIs that are related to our project: reflection, JVMPi and JPDA.

For a portable and generally applicable implementation of CC4J’s core functionality the *reflective capabilities* of Java promote no assistance because they do not allow programmatic modification of existing method code.

The information that can be gained by using the *Java Virtual Machine Profiler Interface (JVMPi)*³ is too coarse grained to be used for coverage or tracing. Due to restrictions in JVMPi, one cannot record information about the execution of statements other than class instantiation, method entries, and method calls.

The *Java Debug Interface (JDI)*⁴ allows more detailed examination of running programs but does not allow for customization of the gathered information. For instance, tracing should ideally provide application-specific semantic information (e.g. reporting “Missing payment.” instead of “Return from method *m* in class *C* of package *P*”). Furthermore, permanently running deployed applications in debug mode in order to gather relevant tracing information is no practical option, given the performance impact of the debug mode in JDK 1.3.

Last, but not least, the recent implementation of dynamic class exchange [Dmi01] as an extension of the Java Platform Debugger Architecture (JPDA) is a powerful complement to JMangler’s capabilities but no replacement. Support for dynamic class redefinition, when in a new class version only method bodies are allowed to change, is available in the HotSpot JVM, which is included in

³ See <http://java.sun.com/j2se/1.4/docs/guide/jvmpi/index.html>

⁴ See <http://java.sun.com/j2se/1.4/docs/guide/jpda/>

the JDK 1.4 release⁵. However, HotSwap is just for redefinition of classes that have already been loaded. Its `RedefineClasses()` method takes just old and new class versions, and it is the programmer's responsibility to prepare the new version (in the form of a complete `.class` file). Interception of loaded classes and their modification is not a part of the HotSwap API.

8 Conclusions and Future Work

In this paper we have reported on the development and evaluation of a commercial code coverage tool, CC4J, which is implemented using the JMangler load-time adaptation framework.

Our evaluation has shown that load-time adaptation is a mature technology. Its applicability for the timely and cost-effective implementation of quality assurance measures has been proven within a large scale industrial project.

Load-time transformation of byte code has the potential to support arbitrary adaptations beyond those required for quality assurance during software development. In the context of component-based development this can even be regarded as a key technology because components are delivered in binary format and often deployed in dynamic environments like application servers.

Our run-time performance evaluations indicate that JMangler is immediately useful in scenarios where adaptation is performed before deployment at the customer (like in our quality assurance task) and during deployment of long-running applications. In the first case, the time spent on adapting classes is not an issue at all. In the second case, it just accounts for a small percentage increase of overall run-time.

In contrast, use of JMangler for adaptation of short-running and time-critical applications would require further optimization of the framework. The goal must be to lower the fixed costs of system startup and tune the API for class file analysis and modification. Corresponding improvements of JMangler are subject of ongoing work. An interesting option for future work is to take advantage of the current and upcoming ability of Java to perform dynamic class replacement [Dmi01]. This would allow instrumented byte code to be added just when needed and to be removed again later on. Regarding CC4J, a possible future enhancement might be the support of a finer grained notion of code coverage. For example, a single source line may sometimes contain a number of statements. Therefore, logging the entry point to individual statements might be technically more adequate. However, this would require significantly more sophisticated analysis of byte code.

Acknowledgements

Misha Dmitriev helped us understand the relation of JMangler to the HotSwap project at Sun Microsystems and contributed many insightful comments on technical aspects and writing style. We are further indebted to Tom Arbuckle Pascal

⁵ See <http://java.sun.com/j2se/1.4/docs/guide/jpda/enhancements.html>

Costanza and the anonymous reviewers for careful proofreading and numerous suggestions that significantly improved the paper and the quality of our English writing.

References

- [Asp01] Aspect oriented software development home page. <http://aosd.net>, 2001.
- [Aus00] Michael Austermann. Ladezeittransformation von Java-Programmen. Master's thesis, Universität Bonn, Institut für Informatik III, 2000.
- [CCK98] Geoff A. Cohen, Jeffrey S. Chase, and David L. Kaminsky. Automatic program transformation with JOIE. In *Proceedings of the USENIX 1998 Annual Technical Conference*, pages 167–178, Berkeley, USA, 1998. USENIX Association.
- [Chi00] Shigeru Chiba. Load-Time Structural Reflection in Java. In Elisa Bertino, editor, *Proceedings of ECOOP2000*, LNCS 1850. Springer, 2000.
- [Dmi01] Mikhail Dmitriev. Towards flexible and safe technology for runtime evolution of java language applications, 2001. In proceedings of Workshop on Engineering Complex Object-Oriented Systems for Evolution (ECOOSE) at OOPSLA 2001, <http://www.dsg.cs.tcd.ie/ecoose/oopsla2001/papers.shtml>.
- [GJSB00] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification Second Edition*. Addison-Wesley, 2000.
- [KCA01] Günter Kniesel, Pascal Costanza, and Michael Austermann. JMangler - A Framework for Load-Time Transformation of Java Class Files. In *Proceedings of International Workshop on Source Code Analysis and Manipulation (SCAM)*. IEEE Computer Society Press, 2001.
- [KH98] Ralph Keller and Urs Hölzle. Binary Component Adaptation. In Eric Jul, editor, *Proceedings ECOOP '98*, LNCS 1445, 1998.
- [LY99] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification (2nd Ed)*. Java Series. Addison Wesley, 1999.

A The Hello World Application

```
public class HelloWorld {
    public HelloWorld() {
        super();
        return;
    }
    public static void main(String[] args) {
        System.out.println("Hello World!");
        return;
    }
}
```

B Sample Code Coverage File

This section shows the coverage file generated by CC4J when executing the “Hello World” application printed in Appendix A.

```
<?xml version="1.0" encoding="US-ASCII" ?>
<SCOOP_CC4J>
  <SEQUENCE_OF_CLASSES>
    <CLASS name="HelloWorld" sourcefile="HelloWorld.java">
      <METHOD name="HelloWorld()">
        <LN l="3" s="n"/>
        <LN l="4" s="n"/>
      </METHOD>
      <METHOD name="main([Ljava/lang/String;)V">
        <LN l="7" s="e"/>
        <LN l="8" s="e"/>
      </METHOD>
    </CLASS>
  </SEQUENCE_OF_CLASSES>
  <ERRORLOG/>
</SCOOP_CC4J>
```

Each line of code in the source of a method results in a LN-tag in the coverage file. The *l* attribute contains the corresponding line number, the *s* attribute contains the status of this line. The value *e* represents *executed*, *n* represents *not executed*.