Experiment and Comparison of Automated Static Code Analyzers and Automated Dynamic Tests

Evan Martin and Karen Smiley
Department of Computer Science
North Carolina State University
Raleigh, NC, USA
{eemartin, kjsmiley}@ncsu.edu

Abstract

Code review and inspection techniques are considered vital for defect detection during analysis and design. Automated static code analyzers are essentially an approach to performing code reviews and inspections in an efficient and timely manner. Automated testing techniques are also gaining popularity and prestige. We explore various readily available code inspection tools and automated testing tools and apply a selected few to a sample application, in an attempt to evaluate the strengths and weaknesses of each tool. We then compare these results to the results obtained from execution of two suites of automated dynamic tests. Our findings indicate that these two methods have innate complementary strengths, suggesting that the joint use of both approaches (static and dynamic) can be far more effective than using either one alone.

Introduction

The techniques available to detect defects in code and isolate their cause are commonly separated into dynamic methods (e.g. testing) and static methods (e.g. inspection) [2]. Dynamic methods execute the program and attempt to detect deviations from expected behavior, whereas static methods assess the source or binary files without execution of the program to identify potential defects.

Code review and inspection techniques are considered vital for defect detection during analysis and design, but should always be used in conjunction with (not as a replacement for) software testing [1]: i.e., they are 'necessary but not sufficient'. It has long been accepted that reviews and inspections can offer high return on investment by finding and fixing defects early in the software development process [6]; yet, in spite of their acknowledged value, these methods have still not become standard practice in the software industry, in part because they require disciplined methods and special training [6]. Furthermore, even though they are cost-effective, manual inspections are a time-consuming step during the earlier stages of a development project. When a project is under time pressure, it is not unusual that inspections are performed incorrectly (e.g., at too high a review rate to be effective) or abandoned altogether, to "save time". One approach which has recently been pursued, in an attempt to mitigate this tendency to cut corners and thus offset the resulting loss of the benefits of inspections, is development of "automated code inspection" tools, i.e. static analysis tools.

This paper presents results from applying a selection of automated static code inspection tools to two versions of a small Java application: one 'good' version which was known to pass 100% of a set of automated dynamic tests, and a readily available 'bad' derivation of that version created by injecting a specific set of logical defects. Our primary goal in conducting this experiment was to attempt to identify specific static and/or dynamic techniques that may be better suited to detect or isolate certain classes of defects in our Java applications.

We begin with a brief survey of recent literature and available static analysis tools for Java; apply criteria to choose a subset of those tools for evaluation; report the results of execution of those tools on our applications; and analyze the results of these tool executions to assess the comparative benefits of the static tools vs. each other and vs. the testing previously performed. Scalability of the evaluated tools to industrial-sized applications is assessed, where possible, by benchmarking them on a large industrial application code base. In the results section of this paper, we summarize the strengths and weaknesses demonstrated by the chosen tools in this experiment, and identify possible areas for future work.

Related Work

Code inspections were developed when the procedural programming paradigm was dominant; subsequently, object-oriented (OO) programming gained popularity and use [1]. The inherent differences between these two programming paradigms introduce different challenges, requiring different inspection techniques. For example, a key challenge with inspection of OO code is that many hard-to-find defects share a common characteristic – delocalization. Delocalization is the characteristic that information required to understand a piece of code is distributed throughout the system. In other words, a trail of method invocations must be followed through many other methods and classes, possibly including inheritance hierarchies, in order to understand a piece of code [1].

Associated with the delocalization problem is the issue of a reading strategy. Two such strategies are:

- Systematic strategy: Code is traced by following a flow from the beginning through the entire program using various values to simulate program input.
- As-needed strategy: Code is inspected in portions by choosing sections that are deemed of interest for a particular enhancement or defect.

Dunsmore et al [1] present problems that may arise when inspecting Object-Oriented (OO) code and discuss the results of a long-term study investigating reading strategies developed to address these problems. In a similar study, Runeson and Andrews explore and compare usage-based inspection and coverage-based testing. It was shown that a systematic or line-by-line approach to code inspection was superior to an opportunistic or as-needed comprehension strategy [2].

Furthermore, the results Runeson and Andrews obtained indicate that white-box testing detects significantly more defects, while inspection isolates the underlying source of a larger share of the defects detected [2]. They conclude that the difference in efficiency is not statistically significant, as white-box testing requires significantly more time, suggesting that the use of a combination of methods is ideal. One of our goals is to determine if our data is consistent with these conclusions.

Regardless of whether an as-needed or systematic reading strategy is used, several categories of methods exist for guiding code inspection, including checklists, abstractions, and scenarios [2]:

- An abstraction-driven reading technique requires inspectors to reverse-engineer an abstract specification for each method [1]. It was designed to address delocalization by providing many of the benefits associated with systematic reading, but in a more efficient manner.
- A use-case driven reading technique reads OO code from a dynamic model viewpoint [1]. Its purpose is to ensure that each object responds correctly to all possible ways in which it might be used.
- Finally, a checklist-driven technique is based on a series of specific questions or rules intended to focus the inspector's attention toward common sources of defects.

Ironically, the strengths of checklist-based techniques (i.e. simplicity, generality, and ease of use) are also their perceived weaknesses: the effectiveness of checklist techniques have been challenged because of their general nature, their focus on defect types driven by historical data, and because the inspectors are not forced to understand the artifacts under inspection [1]. Despite its critics, a recent study [1] found the checklist technique has the strongest performance overall and tends to be the most efficient technique to apply. The abstraction-driven technique appears to be effective at detecting delocalized defects, although its overall performance was less than that of the checklist technique. The use-case technique was the weakest performer, although several study participants noted that its main benefit was that it dealt with methods in the context of the executing system, yielding a better idea of whether the code was operating as expected (validation as opposed to verification).

Recently, advances in software technology such as strongly typed languages, automated testing, intelligent compilers, etc. have eliminated entire classes of defects; thus, the traditional cost-benefit tradeoffs associated with code inspection have changed as well. Although these advances have reduced the number of major defects detected by code inspections, Siy and Votta [3] suggest that code inspections continue to be of value as a maintenance tool by delivering another benefit, almost as a side effect: inspected code becomes easier to understand, change, and maintain. This is achieved by: ensuring the software conforms to coding standards; minimizing redundancies; improving language proficiency; improving safety and portability; and raising the quality of the documentation [3]. Furthermore, their analysis in a recent code inspection experiment shows that 60% of all issues raised by the code inspection are not problems that would be uncovered by later phases of testing or field usage [3]. This suggests that benefit can still be obtained by applying such tools even after later phases of testing have been successfully completed, as we do in this experiment.

Siy and Votta's [3] experiment suggests that the improvements as a result of code inspection did not address the fundamental problems that made the code hard to maintain, which include ambiguous requirements and some object-oriented design decisions that did not turn out to be optimal. As a result, code inspections are good for suggesting local improvements but not global ones.

Rutar, Almazan, and Foster [5] recently presented a comparison of bug finding tools for Java, which we used as a primary reference for identifying candidate tools for this experiment. We supplemented their list with our own Internet research, including the Wikipedia [15], to produce the "short list" for our tool evaluation.

Case Study

Applications Tested

1. The Prioritization Of Requirements for Testing (PORT) application [26], which was the subject of this experiment, was written as a proof of concept for a graduate-level course at North Carolina State University [25] entitled "Software Testing and Reliability". It is a standalone Java application with a Swing UI and an underlying MySql database.

2. A "defective" version of PORT was generated by deliberately introducing 20 specific defects, of varying severity levels, into the PORT source code. The application compiles and runs but exhibits unexpected behavior. See Appendix A for details on the injected defects.

3. In order to provide some insight as to how the results of the code analyzers on PORT might translate into the "real world", the selected tools were also run against a much larger proprietary Java application.

Code metrics were obtained using BCML (Byte Code Metric Library) [17]. The evaluated version of PORT consists of 6 packages, 83 Java classes, and just over 6,000 lines of code (not including lines of test code). The defective version was nearly identical.

However, only partial code metrics are available for the proprietary application: BCML was only executed on a portion of it, as it is too large for BCML to analyze. As a rough indication of the difference in size and complexity, BCML was executed on 16 out of 195 packages: this constituted 689 out of a total of 4,064 classes, yielding 2,478,676 lines of code.

Tests Performed For Experiment

Pre-Experiment Dynamic Tests on the 'Good' Version of PORT

During development of PORT, i.e. prior to this experiment, the dynamic tests performed on the application by the developers included:

- **Automated JUnit [11] Tests**

  Various criteria may be used to determine effectiveness of testing, including branch coverage, code coverage, and path coverage. The JUnit test suite applied to PORT achieved over 80% statement coverage on the non-GUI code, as measured by GERT (Good Enough Reliability Tool [16]).

- **Automated FIT [12] Tests**

  The PORT FIT tests were written to provide 100% coverage of the customer-identified acceptance tests and a small subset of the black box tests written by the development team.

- **Manual System Tests**

  For each iteration, the developers of PORT performed system testing according to their written Black Box Test Plan, which included diabolical tests and other manual tests.

No automated UI testing was performed on PORT by the developers. The resulting risk to the project of an undetected regression in UI operation was mitigated by the manual execution of tests.

The instructors performed additional testing after delivery of each PORT iteration, and the final version of PORT was also tested by other students according to their own Black Box Test Plans. These tests were not included in the evaluation of this experiment.

Pre-Experiment Static Tests on the 'Good' Version of PORT

The only static analysis methods used on the 'good' PORT application, prior to execution of the above dynamic tests, were the checks built into the Eclipse [10] IDE (Integrated Development Environment), and the continuous peer review which results from the use of pair programming [24].

Pre-Experiment Dynamic Tests on the 'Bad' Version of PORT

No dynamic tests were performed on the 'bad' version of PORT prior to this experiment: we simply ensured that it compiled and launched so that it could serve its intended purpose.

Experimental Dynamic Tests on the 'Bad' Version of PORT

As part of this experiment, the same automated dynamic test suites (JUnit and FIT) described in the preceding section were executed on the 'bad' version of PORT. The purpose of this testing was to see if these automated tests were capable of detecting the injected defects, enabling comparison of their effectiveness vs. the selected automated code inspection tools.

Manual tests were not performed on this application.

Experimental Dynamic Tests on Proprietary Application

No attempt was made to assess or compare the effectiveness of any existing dynamic test suites for the proprietary application.

Static Methods – Java Code Analysis Tools

Tool Selection Criteria

Given the timeframe for completion of the experiment, we decided to select at least two, but no more than four, tools from our short list to utilize in this study. We identified, then applied, the following set of criteria to aid in selection:

1. Purchase cost and availability: due to time and monetary constraints, only free or open source tools that could be obtained in less than one day were considered.

2. Ease of setup: due to the brief timeframe of the experiment, tools likely to require more than a day or two to install, configure, and prepare for use were eliminated from consideration. A strong preference was given to tools which could run with un-annotated source or byte code. Checklist-based tools, mentioned above, generally rated well against this criterion – i.e., checklist-based tools are the easiest to apply since they do not require special annotations in order to function. The amount of effort required to be able to use a tool was a major concern: the size and complexity of even the smaller application we wished to evaluate prohibited us from inserting the necessary annotations required by abstraction-based tools (i.e. Bandera, ESC/Java) within the timeframe for conducting the experiment. The trade-off is logical defects requiring knowledge of correct program behavior will not be detected by most checklist-based tools.

3. Execution time: again, due to the brief timeframe of the experiment, tools which, according to the available literature, would be likely to take hours to execute on our Java application were not considered.

4. Likely signal-to-noise ratio of output: tools with a reputation for generating a high number of false positives, requiring significant effort to manually analyze the output to determine whether a legitimate bug had been identified, were eliminated from consideration.

5. Classes of bugs typically found: based on our analysis of the types of defects each tool was likely to find, we wanted to choose tools which would be likely to provide some unique benefit, i.e. an ability to identify a defect type which the other tools in the evaluation set did not typically find. A tool whose sample outputs did not seem likely to be easy to classify and compare to the other tools would not be desirable for this experiment.

Other Factors

While tools which supported multiple platforms (i.e. PC, Mac, and Unix) offer an advantage in convenience and would warrant preferential consideration in tool selection in "real life", platform availability and portability were not considered for this experiment under the second criterion. Nor were other factors such as scalability for larger applications.

Our evaluation of the candidate tools according to these criteria is summarized in Table 1 below.

Table 1 – Static Analysis Tool Selection Criteria and Ratings

Name	Ease of Use	Execution Speed	Readability of Output	Detection of Unique, Useful Defect Types	Selected?
Bandera [13]	Model Checking – Low	*(not evaluated)*	*(not evaluated)*	Logical defects and deadlocks	No
BCML [17]	High	Medium	High	N/A (metrics tool)	Yes (metrics)
CheckStyle [18]	High	High	Medium (lines decorated in Eclipse UI)	Syntax, Coding standards and practices	No
ESC/Java [14]	Theorem proving – Low	*(not evaluated)*	*(not evaluated)*	Logical Defects and Deadlocks	No
FindBugs [9]	High	High	Medium (GUI and text)	Syntax, Coding standards and practices	**Yes**
JLint [7]	High	High	Medium (text)	Syntax, Coding standards and practices, Deadlocks	**Yes**
JML [19]	Low	*(not evaluated)*	*(not evaluated)*	Logical Defects	No
PMD [8]	High	High?	Medium (text)	Syntax, Coding standards and practices	**Yes**

Tools Chosen For Experiment

A summary of the key characteristics of each of the selected tools follows. All are checklist-driven.

FindBugs

(For curious minds, "Finding Bugs Is Easy" [5] is an excellent paper discussing various automated code analysis techniques and FindBugs in particular.) The primary bug-finding approach used by FindBugs is to match source code patterns to known suspicious programming practices. This is complemented by other ad-hoc strategies, such as bytecode-level dataflow analysis [4]. For example, a simple dataflow analysis is used to check for null pointer dereferences within a single method. FindBugs is expandable by writing custom bug detectors in Java, although this capability was not used in the experiment. It is available both standalone and as an Eclipse plugin.

JLint

Similar to FindBugs, JLint performs syntactic checks and dataflow analysis on Java bytecode [4]. JLint also builds a lock graph in order to find possible deadlocks in multithreaded applications. Unfortunately, JLint is not easily expandable. However, it is easy to run, very fast to execute, and according to the literature, is capable of identifying classes of defects not detected by FindBugs [4].

PMD

PMD performs syntactic checks on source code, but does not use any types of dataflow analysis. PMD searches for suspicious stylistic conventions to trigger defect warnings. Since PMD relies heavily on programming style, it is possible to select detectors or groups of detectors that should or should not be run. For developers who may purposely violate a stylistic convention which PMD identifies as a possible fault, this ability to disable detectors (and thereby automatically

suppress the many false positives triggered by such violations) is a necessary feature for the tool to be considered usable. A further benefit is that these detectors are defined as rulesets which are easy to customize and supplement, potentially making PMD a very flexible and powerful tool. However, only standard rulesets included with the software were used for this experiment.

Tools Not Chosen For Experiment

Bandera

Bandera is based on model checking and abstraction [4]. Bandera supports several model checkers, including SPIN [20] and the Java PathFinder [21]. The programmer must annotate source code with specifications describing what should be checked. When working with source code without annotations, Bandera merely verifies the absence of deadlocks. Deadlocks were not considered to be a high-risk item for our target Java application, and thus Bandera's usefulness for this experiment was considered to be too low to warrant investing the effort to manually annotate our source. Even if this were not a disqualifier, Bandera has another substantial limitation: it cannot analyze standard Java library calls, although the Java library is used extensively by virtually every Java application. Bandera's usefulness and effectiveness seems to be severely hindered by these two characteristics.

CheckStyle

This tool primarily verifies that a source code base is compliant against various coding standards. While this capability is potentially useful for enhancing maintainability, identifying possible defects, and enforcing company wide coding practices, adherence to an externally defined coding standard was not deemed to be of great value for this experiment. An initial run of this tool as an Eclipse plug-in showed a high number of stylistic messages (for example: separating operands from operators with a space) in a fairly typical PORT class, indicating that CheckStyle's signal-to-noise ratio might be low for our purposes. Therefore, this tool was not given further consideration.

ESC/Java

The Extended Static Checking system for Java is based on theorem-proving, which is used to perform formal verification of properties of Java source code [4]. The programmer must annotate source code with pre-conditions, post-conditions, and loop invariants in the form of comments. A theorem prover then uses these annotations to verify that the program matches the specifications. Some useful analysis is performed on source code without annotations: for example, possible null pointer dereferences, and array out-of-bounds errors. However, other available tools also detect these kinds of errors. Annotations may also be used to suppress false positives and perform other types of program specification. Unfortunately, without annotations, ESC/Java produces a large number of warnings. In fact, other tools have been developed to automatically annotate source code in an attempt to reduce the warnings produced by ESC/Java. We decided that the effort required to annotate our Java application manually or with these tools did not warrant inclusion of ESC/Java in our experiment.

Java Modeling Language (JML)

The Java Modeling Language (JML) specifies detailed Java classes and interfaces. It specifies both the behavior and the syntactic interface of Java code meaning that method signatures (the names, the types of its fields, etc.) as well as its function are all specified [30]. This facilitates the definition of contracts. Various tools are available that use JML to test (either statically or dynamically) if code adheres to its intended use and function. For a more detailed summary of its purpose, uses, and basic operation as well as a survey of all known tools using JML, see [30].

Results

FindBugs, JLint, and PMD were applied to these three Java applications ('good' PORT, 'bad'

PORT, and the proprietary application) in order to compare the outputs. We used BCML (Byte Code Metric Library [17]) as a way to measure the size of the applications, to establish a frame of reference for interpretation of the tool findings.

Dynamic Tests on the 'Bad' Version of PORT

For comparison to the outputs of these three static analysis tools, the automated dynamic testing tools discussed above were executed on the 'bad' version of the PORT application, revealing the following:

- JUnit executed 435 assertions with 13 failures revealing 6 of the 20 injected defects, all of which are severity 3.
- FIT executed 71 correct tests, 132 incorrect tests, encountered 571 exceptions, and detected 4 of the 20 injected defects, again all of severity 3.
- All 4 defects identified by the FIT suite were also identified by execution of the JUnit suite. However, it is worth noting that each FIT test operates on the assumption that previous steps in the test succeed. As soon as an error is encountered in a given FIT test, it is unlikely that any other tests will pass within that spec, so additional defects that would otherwise be detected are not. As defects are fixed and FIT is run, in an iterative fashion, the FIT tests will detect more of the injected defects.

Note that 11 of the 20 injected defects reside entirely in the GUI: since both FIT and JUnit exercise the PORT application just below the GUI level, these defects are not expected to be detected by JUnit or FIT. See Appendix B for the details of the dynamic tests.

Static Tests on the 'Good' and 'Bad' Versions of PORT

The results of running the static code analysis tools on PORT were identical for both the good and bad versions. Specifically, the findings of the tools are:

- FindBugs ran relatively quickly and produced output that was both easy to read and navigate. A majority of the 67 potential problems identified by FindBugs are not likely to represent defects that an end user of the PORT application would ever see. However, a quick read indicated that the tool had likely found some genuine defects.
- JLint was extremely quick to execute as well, but threw a high number of false positive messages (63 out of the 81 total found) in the area of inheritance. The 6 identified dataflow bugs appear legitimate and warrant further examination.
- Both FindBugs and JLint highlight potential synchronization problems that would, in many applications, represent true defects but in this PORT application were not likely to be issues.
- PMD, due to its simplicity, ran very quickly but generated few, if any, interesting results with the rulesets used. It may be far less effective at finding legitimate defects compared to JLint or FindBugs, however, due to its flexibility; it seems to be useful for general housekeeping and enforcing company wide coding standards. PMD operates at the source level, whereas JLint and FindBugs operate at the byte code level, making their uses quite different.

None of these three checklist-based code analyzers detected any of the injected anomalies in the defective version of PORT.

See Appendix C for the full details of the results from BCML, FindBugs, JLint, and PMD.

Conclusion

The conclusions which can be drawn from this experiment are limited by the relatively small number of injected defects and, more importantly, how they were chosen. This set of injected logical defects was used because it had already been created for other purposes and was readily available, and because we assumed that at least a few defects of the other types were probably latent in the application. To some extent this assumption was borne out, but it is still unlikely that the exact mix of latent defects and injected logical defects would coincidentally happen to closely approximate a typical real-life distribution of defect types. This clearly limits the applicability and potential predictive value of our results.

Strengths and Weaknesses Summary

Despite the stated limitations of this case study due to the set of injected defects used, our results are consistent with the cited works in demonstrating the complementary value of static and dynamic testing:

- The static analyzers completely failed to find the logical defects, but notably succeeded in identifying some possible bugs that had not been revealed by testing.

- Testing required significantly more time and effort to implement, but was far more effective at detecting the injected logical defects.

However, even the combination of the dynamic testing methods with all three static analyzers was insufficient to identify the presence of the injected defects which affected operation of the Swing user interface.

We recognize that it is theoretically possible, even likely, that a more robust set of dynamic tests using these same tools (e.g. a JUnit suite providing a significantly higher degree of coverage as measured by GERT or a "test tester" such as Jester [29], or a FIT suite which automated 100% of the automatable tests in the Black Box Test Plan) would be capable of finding more of the injected defects. Similarly, it is possible that with the easy extendibility of FindBugs and PMD, a higher defect identification rate could be achieved.

However, we consider it unlikely that these tools alone would be able to find them all, given that none of them seem capable of finding UI defects and that all of the injected defects were logical defects deliberately inserted (often by commenting out necessary lines of code) to produce incorrect program behavior.

- The chosen dynamic tests used during PORT development deliberately excluded automation of UI testing. Inability to detect UI defects was an acknowledged risk of this testing strategy, mitigated during by development by manual test execution (a dynamic testing method which was not included in this experiment due to time constraints).

- With 20/20 hindsight, it is not surprising that the selected static analysis tools failed to find any of the injected defects. Only abstraction-driven or use-case driven static code analyzers such as Bandera or ESC/Java would be likely to reveal such defects, since they are not a result of the types of poor programming practices that the checklist-driven tools strive to identify.

In summary, we consider the experiment a qualified success, in that it demonstrated to us that our own effectiveness as developers will likely improve if we become more proficient in using static testing methods (since they exhibited complementary strengths vs. dynamic methods even under the limited circumstances of this experiment), and using additional dynamic testing tools (since our current test suites were clearly inadequate to find all of the defects we injected).

Future Work

An experiment based upon careful creation of a set of injected defects which more closely approximates the type and frequency of defects in a well-established taxonomy, e.g. IBM's Orthogonal Defect Classification [27] or the defect type categories used in the Software Engineering Institute's Personal Software Process[SM] [28], would be more likely to yield meaningful results.

A test suite which would automatically exercise the functionality of the GUI layer might possibly have been able to find the injected defects which were missed by the applied automated dynamic tests and the evaluated static analysis tests. Examples of tools which could potentially be used for this type of testing are JFCUnit [22] and Abbot [23].

A tool that automatically generated a model of expected program behavior (e.g. from the JUnit test suites) might be of value, in that it could eliminate a significant cost of applying model-based tools such as Bandera and ESC/Java. Creation of such a tool would enable evaluation of the benefits or limitations of running them, vs. the benefits and limitations of the checklist-driven static analyzers evaluated in this experiment. A related interesting research question would be to compare and analyze the use of Bandera or ESC/Java with a JUnit-generated model vs. a manually generated model. While a well-done manually generated model might be more likely to deliver superior results, an automatically generated model could potentially deliver "good enough" results to be beneficial while being both practical and cost-effective due to a reduction in time and effort.

The authors plan to extend this analysis to include additional code bases and to address some of the stated limitations of this experiment.

References

[1] Dunsmore, A.; Roper, M.; Wood, M. Practical code inspection techniques for object-oriented systems: an experimental comparison. *IEEE Software*, Vol.20, Iss.4, July-Aug. 2003, pp. 21- 29.

[2] Runeson, P.; Andrews, A. Detection or isolation of defects? an experimental comparison of unit testing and code inspection. *14th International Symposium on Software Reliability Engineering*, (ISSRE 2003), Vol., Iss., 17-20 Nov. 2003, pp. 3- 13.

[3] Siy, H.; Votta, L. Does the modern code inspection have value? *IEEE International Conference on Software Maintenance*, Vol., Iss., 2001, pp.281-289.

[4] Rutar, N; Almazan, C; Foster, J. A comparison of bug finding tools for java. *15th IEEE International Symposium on Software Reliability Engineering* (ISSRE2004), Saint-Malo, Bretagne, France. Nov. 2004, pp. 245-256.

[5] Hovemeyer, D.; Pugh, W. Finding bugs is easy. *19th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, (OOPSLA 2004), Oct. 2004.

[6] Humphrey, Watts S. Winning with software. Addison-Wesley, Boston, 2002.

[7] JLint. http://artho.com/jlint/

[8] PMD. http://pmd.sourceforge.net/

[9] FindBugs. http://findbugs.sourceforge.net/

[10] Eclipse. http://www.eclipse.org/

[11] JUnit. http://www.junit.org/

[12] FIT (Framework for Integrated Testing). http://fit.c2.com/

[13] Bandera. http://bandera.projects.cis.ksu.edu/

[14] ESC/Java. http://research.compaq.com/SRC/esc/

[15] Wikipedia. http://en.wikipedia.org/wiki/Main_Page

[16] GERT (Good Enough Reliability Tool). http://gert.sourceforge.net/

[17] BCML (Byte Code Metric Library) http://csdl.ics.hawaii.edu/Tools/BCML/

[18] CheckStyle. http://checkstyle.sourceforge.net/

[19] JML (Java Modeling Language). http://www.cs.iastate.edu/~leavens/JML/

[20] SPIN. http://www.spinroot.com

[21] Java PathFinder. http://ase.arc.nasa.gov/visser/jpf/

[22] jfcUnit. http://jfcunit.sourceforge.net/

[23] Abbot. http://abbot.sourceforge.net/

[24] Pair Programming. http://www.pairprogramming.com/

[25] North Carolina State University (NCSU). http://www.ncsu.edu

[26] Srikanth, H., and Williams, L. Requirements-Based Test Case Prioritization, Grace Hopper Poster, 2004. http://collaboration.csc.ncsu.edu/laurie/Papers/GHCPoster.pdf

[27] Chillarege, R. et.al., Orthogonal defect classification - a concept for in-process measurements. *IEEE Transactions on Software Engineering*, Vol.18, Iss.11, Nov. 1992, pp. 943–956.

[28] Humphrey, Watts. A discipline for software engineering. Addison-Wesley, Boston, 1995.

[29] Jester. http://jester.sourceforge.net/

[30] Burdy, L; Cheon, Y; Cok, D; Ernst, M; Kiniry, J; Leavens, G; Leino, R; and Poll, E. An overview of JML tools and applications. To appear in *International Journal on Software Tools for Technology Transfer*. ftp://ftp.cs.iastate.edu/pub/leavens/JML/sttt04.pdf

Appendix A – Defects Injected To Create Defective 'PORT' Application

Injected Defects		
ID	**Class**	**Description**
Severity 1 Defects		
1.1	DefectDataView.java	This will prevent the defect severity level from being set on any edit or add operation.
1.2	ProductDataView.java	This will prevent the customer priority weight from being set on any add or edit operation. Products cannot be added.
1.3	RequirementDataView.java	This will prevent the requirement type from being set on any edit or add operation. Only the default type can be added.
1.4	TestCaseDataView.java	This will prevent the test case description from being set on any add or edit operation. Test cases cannot be added.
1.5	EditAction.java	This will prohibit the editing of any test case entry.
Severity 2 Defects		
2.1	DefectDataView.java	This will prevent the defect comment from being set on any edit or add operation.
2.2	DefectDataView.java	This will allow the open date to be edited in edit mode, although it should not be editable.
2.3	RequirementDataView.java	This will allow the origination date to be changed for a requirement being edited, although it should not be editable.
2.4	TestCaseDataView.java	This will allow the test case id to be edited for an existing test case, although it should not be editable.
2.5	RequirementDataView.java	Although data is stored correctly in the database, this will prevent the displayed requirement completion date from reflecting the date in the database.
Severity 3 Defects		
3.1	PortControllerView.java	Right-click in the tree will launch popup action against last selected tree node instead of the node clicked on by the user.
3.2	NewDefectAction.java	This will cause a new product to be created when a new defect is desired.
3.3	NewRequirementAction.java	This will cause a new product to be created when a new requirement is desired.
3.4	NewTestCaseAction.java	This will cause a new product to be created when a new test case is desired.
3.5	NewProductAction.java	This will cause a new defect to be created when a new product is desired.
3.6	NewAction.java	This will cause a new entry to be added to the tree but not automatically selected.
3.7	CancelAction.java	This will cause the cancel button to do nothing at all.
3.8	RequirementsDataView.java	This will cause the title and type title fields to be reversed in the requirements table.
3.9	RequirementsDataModel.java	This will cause the requirements data view to display the wrong name.
3.10	RequirementsDataModel.java	This will cause the requirements node in the tree to not be displayed as a folder if it is empty, which is inconsistent with the defects and test cases folder.

Appendix B – Results of Dynamic Tests on Defective 'PORT' Application

Injected Defects Detected by JUnit			
	Test Class	**Test Method**	**Defect Discovered (Level.ID)**
1	NewDefectActionTest	testActionPerformedImpl	3.2
2	RequirementsDataModelTest	TestIsLeaf	3.10
3	NewActionTest	testAddNewProdsEntry	3.6
4	NewActionTest	testAddNewProdEntry	3.6
5	NewActionTest	testAddNewReqsEntry	3.6
6	NewActionTest	testAddNewReqEntry	3.6
7	NewActionTest	testAddNewTCsEntry	3.6
8	NewActionTest	testAddNewTCEntry	3.6
9	NewActionTest	testAddNewDefsEntry	3.6
10	NewActionTest	testAddDefProdEntry	3.6
11	NewRequirementActionTest	testActionPerformedImpl	3.3
12	NewProductActionTest	testActionPerformedImpl	3.5
13	NewTestCaseActionTest	testActionPerformedImpl	3.4
Injected Defects Detected by FIT (duplications are ignored)			
1	I4BBtestsComputeASFD.html	new product action	3.5
2	I4BBtestsComputeASFD.html	new requirement action	3.3
3	I4BBtestsComputeWeightedPriority.html	new test case action	3.4
4	I4BBtestsComputeWeightedPriorityMap.html	new defect action	3.2

FIT Results Summary			
Rights	**Wrongs**	**Exceptions**	**Ignored**
71	132	571	0

Notes on FIT tests:

1. Errors in FIT have a cascading effect; that is, as soon as an error is encountered in a given FIT test, it is unlikely that any other tests will pass within that spec, so additional defects that would otherwise be detected are not. Specifically, FIT is unable to create any new entries due to the injected defects, so it cannot execute any subsequent tests because the object modeling the entry does not exist. Iterative fix-and-test cycles are required to find all of the defects these tests can potentially identify.
2. Since all of the FIT fixtures detected the same injected defects, duplicate entries are not recorded in the table above.

Appendix C – Results of Static Tests on PORT, Defective PORT, and Proprietary Application

	PORT (good)	PORT (20 defects injected)	Proprietary Application
BCML Statistics Report			
Execution Time (verbose mode)	22.09 sec	22.14 sec	10 - 20 minutes *(on a dual 2.2 GHz hyperthreaded, 1 GB ram IBM server)*
Packages Analyzed	6	6	**16**
Classes	83	83	689/4064
CBO (Coupling Between Objects)	882	881	7779
DIT (Depth of Inheritance Tree)	234	234	2121
LOC (Lines of Code, approximate)	6026	6008	36099
NOC (Number of Children)	44	44	393
RFC (Response For a Class)	1860	1850	14556
WMC (Weighted Methods Per Class)	820	819	3901
Size (object code size in bytes)	316829	316085	2478676

Notes on BCML [17]:

1. BCML was only executed on a portion of the proprietary application, as it is too large for BCML to analyze. Specifically, only 16 of 195 packages (689 out of 4064 classes) were used to generate the metrics above.
2. Below we offer for reference brief descriptions of the code metrics in the table, per the BCML user guide [17]:
   - 2.1. Weighted Methods per Class (WMC): Calculated with weight=1, so this metric represents the total number of methods in this class.
   - 2.2. Depth of Inheritance Tree (DIT): Calculated as the number of parent classes (using the 'extends hierarchy' relation). A class that inherits from Object has DIT=0. Interface inheritance (i.e. using the 'extends' relation) is not represented.
   - 2.3. Number of Children (NOC): Calculated as the number of sub-classes of this class (using the 'extends' relation).
   - 2.4. Coupling Between Objects (CBO): Calculated as the total number of other classes whose methods are invoked in this class.
   - 2.5. Response for a Class (RFC): Calculated as the total number of unique methods (local and external) invoked by this class.
   - 2.6. Class size in bytes (SIZE): The size of the object code representing this class. (Not a CK metric).
   - 2.7. Last modified date (LastModified): The date and time that the .class file representing this class was generated. (Not a CK metric).
   - 2.8. Lines of Code (LOC): The number of non-comment lines of source code, as determined from inspection of the .class file. These LOC values are only approximate, since they cannot be measured accurately from byte code data.

PMD Summary Report	PORT (good)	PORT (20 defects injected)	Proprietary Application
Execution Time	*< 1 minute*	*< 1 minute*	*< 5 minutes*
Ruleset: Unused code	15	15	918

JLint Summary Report	**PORT (good)**	**PORT (20 defects injected)**	**Proprietary Application**
Execution Time	0.14	0.11	
***Number of Messages by type*:**	**81**	**81**	**14054**
AntiC – bugs in tokens	0	0	0
AntiC – operator priorities	0	0	0
AntiC – statement body	0	0	0
JLint – synchronization	**12 (14.8% of total)**	**12 (14.8% of total)**	**10222 (72.7% of total)**
3.1.? multi_lock			1
3.1.1 loop_deadlock			820
3.1.2 loop_graph			340
3.1.3 deadlock_thread			10
3.1.4 wait_lock			7
3.1.5 deadlock			7
3.1.6 sync_override			22
3.1.7 diff_threads	1	1	6515
3.1.8 not_volatile (field)	6	6	2351
3.1.9 runnable_not_sync	5	5	117
3.1.10 value_changed			24
3.1.11 lock_changed			1
3.1.12 call_wo_sync			7
JLint – inheritance	**63 (77.8% of total)**	**63 (77.8% of total)**	**2834 (20.2% of total)**
3.2.? equals_hashcode	2 (real bug)	2 (real bug)	94
3.2.1 not_overridden	2 (false pos)	2 (false pos)	2
3.2.2 shadow_local (component)	47 (false pos)	47 (false pos)	2007
3.2.3 shadow_local (variable)	12 (false pos)	12 (false pos)	650
3.2.4 finalize			81
JLint – data flow	**6 (7.4% of total)**	**6 (7.4% of total)**	**999 (7.1% of total)**
3.3.? bounds	2	2	92
3.3.1 unchecked_null			14
3.3.2 null_reference	2	2	320
3.3.3			0
3.3.4 zero_operand	1 (false pos)	1	200
3.3.5 always_zero			32
3.3.6 shift_count			4
3.3.7 out_of_domain			29
3.3.8 range			27
3.3.9 truncation			54
3.3.10 typecast			32
3.3.11 compare_result			57
3.3.12 operand_compare			33
3.3.13 remainder			1
3.3.14			0
3.3.15 string_cmp	1 (bug)	1 (bug)	20
3.3.16 compare_eq			84

FindBugs Summary Report	**PORT (good)**	**PORT (20 defects injected)**	**Proprietary Application**
Execution Time	< 1 minute	< 1 minute	?
Packages Analyzed	6	6	**195**
Outer Classes – Bugs/Count (%)	62/53 (116.98%)	62/53 (116.98%)	2773/2754 (100.69%)
Inner Classes – Bugs/Count (%)	5/30 (16.67%)	5/30 (16.67%)	221/842 (26.25%)
Interfaces – Bugs/Count (%)	0/0 (0.00%)	0/0 (0.00%)	0/468 (0.00%)
Total – Bugs/Count (%)	**67**/83 (79.52%)	**67**/83 (79.52%)	**3044**/4064 (74.83%)
By Bug Type:			
Co: Covariant compareTo()	1	1	***See screen shots on next page***
E1: Method returning array may expose internal representation	1	1	
E12: Storing reference to mutable object	2	2	
MF: Masked Field	1	1	
MS: Mutable Static Field	15	15	
ODR: Database resource not closed on all paths	1	1	
SIC: Inner class could be made static	3	3	
Se: Incorrect definition of serializable class	12	12	
SnVI: Serializable class with no Version Id	6	6	
UIL: Unsafe Inheritance	1	1	

FindBugs Summary Report (continued) – for Proprietary Application: