JBlanket: Support for Extreme Coverage in Java Unit Testing

Joy M. Agustin
Information and Computer Science
University of Hawai'i at Manoa
Honolulu, HI 96822
jagustin@hawaii.edu

Abstract

Unit testing is a tool commonly used to ensure reliability in software development and can be applied to the software development process as soon as core functionality of a program is implemented. In conventional unit testing, to properly design unit tests programmers will need to have access to specifications and source code. However, this is not possible in Extreme Programming (XP), where tests are created before any feature of a system is ever implemented. Obviously, XP's approach does not lead to improper testing, but instead leads to a different approach for testing.

JBlanket is a tool developed in the Collaborative Software Development Laboratory (CSDL) at the University of Hawai'i (UH) that is meant to assist these types of "unconventional" testing that calculates method-level coverage in Java programs, a coarse enough granularity of test case coverage whereby programmers will not only be able to ensure that all of their unit tests pass, but will also be able to test all of their currently implemented methods. Unit testing where 100% of all unit tests must pass that also exercises 100% of all non-trivial remaining implemented methods is called Extreme Coverage. This research will attempt to show that Extreme Coverage is useful in developing quality software.

1. Introduction

Unit testing is a tool commonly used to ensure reliability in software development and can be applied to the software development process as soon as core functionality of a program is implemented. It is after this first phase in coding that programmers will have access to source code with which they can begin testing [11]. The three main reasons that drive the usage of unit testing include 1) easier management of the individual units, or "modules", or combinations of modules before they are combined to form the entire system, 2) easier to find and correct bugs, or debug, since already exercising the module in which the bugs originated, i.e., eliminates wasted time searching for the guilty module containing the bugs, and 3) allows multiple modules to be tested in parallel [6].

In the conventional case, testers need to have access to a module's specifications and source code to design proper test cases. First white-box testing techniques, or "glass box testing", are applied to the source code to verify its logic. Then black-box testing techniques are derived from the specifications and then applied [6]. To ensure that intimate association with a module does not influence testing, this task is usually performed by person(s) other than the programmer. In the case of unit testing, the programmer and the tester is the same person [7].

However, in the case where the software development model used is Extreme Programming (XP), the programmer first designs the unit tests from the specifications before any code is implemented. It is only after some code is written that it can be exercised by the unit tests. One hundred percent of the unit tests must pass. This process is then repeated throughout the software development life cycle as each feature is coded. Does this mean that unit testing in XP is improperly designed or that the unit tests are not as useful as the unit tests designed in the conventional sense? Many articles have been published that say otherwise, [8] [9] [10] to name a few.

However, as the size of programs change, so do the methods used for testing and the criteria used to measure its quality. One example of change is with respect to test case coverage, i.e., a metric that measures the amount of code that is exercised by the test cases. Boris Beizer claims that during unit testing 100% coverage is necessary, and that this level of coverage usually drops as modules are combined or as testing is done on huge systems of approximately 10 million lines of code [7]. On the other hand, Brian Marick conducted a study where he examined the different granularities of coverage, which will be discussed further in Section 2 [12]. From his study, he claimed that 100% of "feasible coverage" is an acceptable level of coverage to achieve.

Of the different coverage granularities, the focus will be on method-level coverage. This level of coverage was chosen because higher levels of coverage appear to be cheaper to achieve than all the other types of coverage. For example, during highly volatile periods of software development where a system's source code is continuously evolving and refactored at a relatively quick pace, it obviously requires less effort for programmers to achieve higher levels of method-level coverage than statement-level coverage, one of the simplest measurements to calculate [17] [15], since the only way to exercise every statement is to exercise every method that contains those lines of code. (The exception, of course, being abstract methods in Java).

This is the case in XP, where source code can be considered highly volatile as test cases are created and more code is written or refactored in every iteration of development. Interestingly, there is not rule in XP that states every method written needs to be executed during testing. This rule may be implicitly implied since code should only be written to satisfy a test case, but is impossible to guarantee as programmers move further into development.

In its "purest" form, 100% method-level coverage requires invoking every method during testing. However, it is not clear whether this level of coverage a practical goal. Although method level coverage is cheap for achieving high levels of coverage, test cases solely aimed at exercising methods with only one line of code are most times worthless. Unless the single line of code contains a complicated logical expression, for example, inspection is probably better for verifying its correctness. In addition, some methods are untestable. For example, abstract methods in Java can never be invoked. Therefore, "pure" method-level coverage is not only impractical, but can also be impossible to achieve. However, method-level coverage of all "non-trivial" methods, i.e., methods that contain more than one line of code, or are not abstract, is not impractical. This approach to unit testing will be referred to as "extreme coverage".

By following the guidelines of extreme coverage, XP programmers will not only be able to ensure that 100% of their tests pass, but they will also be able to ensure that 100% of their non-trivial methods are exercised. Furthermore, this concept can also be applied to non-XP software models where test cases are created to exercise existing source code. In this case, extreme coverage can help testers ensure that all non-trivial methods are executed at least once during testing. As a result, this approach also has to potential to reduce the size of systems by high-lighting potentially unneeded code and to improve testing by improving the way test cases are implemented.

This research project will investigate the concept of extreme coverage and attempt to provide answers to the following questions:

1) Is knowledge of method-level coverage helpful to implementing quality software?
2) How much effort does it take to achieve and maintain 100% method-level coverage?
3) Does knowing method-level coverage influence the way systems are implemented?
4) Is extreme coverage feasible?

As an aid to this research, extreme coverage will be measured by JBlanket, a method-level coverage tool that is currently in development in the Collaborative Software Development Laboratory (CSDL) at the University of Hawai'i (UH) at Manoa.

The remainder of this paper will be presented in the following order. The next section will discuss previous studies and a selection of coverage tools that currently exist. Section 3 describes the functionality and architecture of the JBlanket system. The evaluation procedures and results of the above hypotheses are discussed in Section 4. Finally, Section 5 contains the conclusions and possible future directions of this research.

2. Related Work

Test case coverage measurement can be performed using different granularities. Numerous studies have been conducted with the different granularities to discover the ideal level of code coverage and the possible impacts they have on the quality of software. Over the years, many automated tools have also been developed to measure these different granularities of coverage. In this section, previous research and existing tools will be discussed.

2.1. Related Work

In [13], Elbaum et. al presented a study that compared the effectiveness of using either statement coverage or method-level coverage for prioritizing test cases during regression testing. Each coverage type was measured in four different ways: total coverage, additional elements invoked, total fault-exposing potential (FEP), and additional FEP potential. These eight types of coverage were executed on eight C programs with sizes ranging from 138 to 6218 lines of code (LOC), seven of which were under 520 LOC.

They found that while statement coverage performed better than method-level coverage, there were several cases in which the difference between coverages were not significant, and two cases in which a method-level measurement performed better than its statement-level counterpart. In addition, on the average, the various method-level coverage measurements performed similarly to statement coverage measurements. The ranking for both types were: 1) additional FEP potential, 2) total FEP, 3) total coverage, and 4) additional elements invoked. The authors also noted that, while some loss of effectiveness can be expected due to the coarser granularity, their findings suggest benefits of method-level coverage should be further investigated since it is the "less costly and less intrusive" approach [13].

This study relates to the helpfulness of method-level coverage. If it can perform similarly to a finer granularity during regression testing, perhaps it can be used during unit testing to obtain useful data about test cases and the amount of a system being exercised.

An experiment conducted by Marick [12] suggested that high levels of coverage are acceptable goals with various granularities of test case coverage. He measured the cost of reaching near 100% coverage with branch coverage, loop coverage, multi-condition coverage, and weak-mutation coverage. Cost was determined in terms of the amount of coverage attained, the number of test cases documented, the amount of time needed to design the test cases, and the number of conditions argued to not be feasible to test. Infeasible conditions included conditions which are either impossible to test or are not worthwhile testing.

The results of this single person experiment showed that after two tries, branch coverage reached 95% using black-box testing techniques, a level noted to be higher than those reached in previous studies. In addition, when both loop and multi-condition coverage results were combined, their total reached 92%. To exercise the remaining 8% would require "3% of total time, 2% of total test conditions, and 3% of the total test cases" [12]. By using these various granularities of coverage, the experimenter concluded "100% feasible coverage is a reasonable testing goal for unit testing" [12].

However, this experiment was conducted on a very small scale. The experimenter was the only person conducting the experiment (i.e., creating missing specifications, designing test cases, calculating the amount of time used designing the test cases, etc.). The systems measured were C programs consisting of 30 to 272 LOC. Results from such small experiments cannot be generalized to include larger systems [18] or be generalized to other granularities of coverage since each coverage type has different weaknesses [19].

Therefore, these findings cannot be generalized to method-level coverage. So how much effort is needed to reach 100% method-level coverage is unknown. In this research, effort will be measured in terms of the total LOC, total test LOC, and the amount of coverage obtained for the system measured. To ensure measuring only "feasible coverage", rules pertaining to the types of methods included in coverage will be applied.

Piwowarski et. al studied the benefits of statement coverage on a large scale software system at IBM [16]. They measured statement coverage during unit testing, function testing, and system testing. Initially, they observed that testers overestimated their coverage when they did not know their actual coverage. For example, some estimated achieving coverage of 90% or above, but actually reached only 50% to 60%. However, after measuring coverage, they found problems such as unreachable code or unexecutable code prevented 100% coverage. For example, code managing unlikely errors during normal execution cannot be reached under normal circumstances, or special hardware commands cannot be executed during testing [16].

The authors concluded that "70% statement coverage is the critical point for our function test", "50% statement coverage was generally insufficient", and "beyond a certain range (70%-80%), increasing statement coverage becomes difficult and is not cost effective" [16]. They also found that with coverage information, test cases could be improved to increase coverage 10%. Furthermore, while 100% statement coverage is not feasible during function testing, it is feasible during unit testing.

From their experiment, it is clear that knowledge of statement coverage influenced the implementation of test cases while trying to increase coverage. This is probably the case with method-level coverage also. However, in what ways are the test cases are modified? For example, does it require significantly more code, or just minor adjustments to current test cases to increase coverage?

These three case studies have influenced the evaluation of the usefulness of extreme coverage. The next section describes the influences that guided the design and implementation of JBlanket, the system used to gather data for this research.

2.2. Coverage Tools

While numerous tools exist that offer the various forms of coverage, none of them were considered to play key roles in this research. Although the tools may or may not have either offered method-level coverage, the main reasons behind this decision are that majority were Closed Source projects and the possible financial strain on the evaluators.

With Closed Source projects, they either did or did not offer method-level coverage. If method-level coverage was not offered, the tool could not be extended to include the needed coverage measure. If the tool did offer method-level coverage, the options were to either spend over a hundred dollars to purchase a copy of the tool, or use trial versions for at most 30-days. Since undergraduate college students were the evaluators, it did not seem feasible to require them to purchase individual copies of or licenses for coverage tools or be constrained by the life span of trial versions. These actions would most likely have shrunk the size of the evaluator population considerably.

Therefore, the coverage measurement tool used in this research needed to be accessible and available for use under any situation. Hence, to avoid re-downloading expired trial versions, the obvious choice was to use an Open Source Project.

The coverage tools reviewed here appear to be among the most popular (i.e., appeared higher up in the Google ranking).

2.2.1. Clover. Clover [1] is an impressive code coverage tool that determines which sections of code are not executed during testing. The current version of

Clover, version 0.6b, supplies two JAR files, clover.jar and velocity.jar, and can measure method, statement, and branch coverage. After running this tool with Jakarta Ant, class files are produced that include both the original program and Clover's methods to record trace data. This automatic addition ensures that the user does not need to manually alter their source code. Clover's output can be viewed as either XML, HTML, or through a Swing GUI. Any unexecuted code is highlighted for quick identification.

Users need to have access to the source code of the system being tested because Clover recompiles the entire system to include its ``coverage gathering mechanism''. While this restricts the tool from being used on systems in which only byte code is available, it allows users to include or exclude specific chunks of code from coverage by adding Clover specfic commands to the source code.

In addition, this is a Closed Source system and it is not clear whether it can be used with client-server systems. The projects used for the evaluation uses Jakarta Tomcat as the server.

2.2.2. JCover. With JCover(TM) [2], users can work with a program's source code, class files, or both to calculate statement, branch, method, class, file, or package coverage. It can conduct client and server-side testing with any ``standards-compliant JVM''. An additional Java API is included that allows the user to ``programmatically control JCover's(TM) coverage agent at runtime'' [2]. This API must be integrated into the user's test framework. All coverage data is archived for future analysis. The data collected can also aid in optimizing tests by including whether coverages overlap or are disjoint. The reports are formatted in HTML, XML, CSV, and MDB.

JCover(TM) is not an Open Source project, but a fully functional 15-day evaluation copy can be downloaded [2]. It is not clearly stated on this tool's web page what the process of data collection is or what servers it can be used with.

2.2.3. Optimizeit Code Coverage. Optimizeit Code Coverage is a part of Borland's Optimizeit Suite, which also contains two other tools, Optimizeit Profiler and Optimizeit Thread Debugger. It measures class, method, and statement coverage. Depending upon the type of measurement, it calculates the number of times a class, method, or line of code is executed in real-time. A GUI is also available for quick identification of results. The source code is not required for this coverage tool. Class and jar files are sufficient to receive an accurate measurement. It also works with application servers [4].

While this is not an Open Source project, it also offers a 15-day trial version [4]. In addition, Optimizeit Code Coverage seems to show coverage for every class in an application. The user does not appear to have the option to focus on a specific subset of classes.

2.2.4. Quilt. Quilt is an Open Source project created by David Dixon-Peugh and Tom Copeland. Currently it offers statement, branch, and path coverage. Through byte code instrumentation, classes are loaded into a ClassLoader specifically designed for Quilt before they are loaded into the JVM. Statistics are kept, from which coverage is calculated. Results can be displayed in HTML or XML using its reporting functionality [5].

Quilt is released under the Apache License. Therefore, someone other than the authors can extend Quilt to include method-level coverage. However, while their licensing makes Quilt available for use free of charge, modifying it to include method-level coverage and integrating it with Jakarta Tomcat did not proved to be possible due to the author's limited knowledge of Java.

From the coverage tools reviewed, both Clover and Quilt were considered as possible candidates in this research. However, its price as well as the length of its trial version hindered access to Clover. It would be detrimental to this study if the evaluators were required to renew their trial version after it expired. Furthermore, if they would not be able to run Clover with Tomcat, the evaluators would not be able to use it to measure their systems (see Section 4).

With respect to Quilt, while it is an Open Source system that can be modified to also measure method-level coverage, the author found the use of ClassLoader to inhibit integration of Quilt with Jakarta Tomcat. Therefore, the decision was made to create JBlanket.

3. The JBlanket System

To gather method-level coverage data for this research, the JBlanket coverage tool was developed. It uses JUnit test cases to calculate the percent of methods in a system invoked during testing. From the various coverage systems available, JBlanket was designed to combine desirable features from the many different systems so that it would be a feasible tool for research. This means that from a research standpoint, it is readily available, relatively easy to use, and relatively easy to understand. By creating a readily available tool allows others to gain access to the tool and be able to integrate it into their own research. With relatively ease of use, people will not be discouraged from using the tool inside and outside of their research. Then with coverage results presented in a comprehensible manner, people will be able to easily understand how to apply the results.

3.1 System Functionality

JBlanket is able to measure coverage of both stand-alone and client-server systems. However, it has only been applied to client-server systems that use Jakarta's Tomcat as the web server. To calculate coverage, users will need to have access to a system's source code and byte code. With these two sources of input, four main

output sets are created: (1) the total methods measured in the system (total), (2) the methods that cannot be invoked during testing (untestable), (3) the methods invoked during testing (tested), and (4) the remaining methods that are not invoked during testing (untested), considered to be untested. Coverage is measured with the following formula:

```
% coverage = tested/untested
```

where

```
untested = (total – untestable) – tested
```

To measure extreme coverage, users have the option of excluding methods that contain one line of code. These "one-line methods" form an optional fifth output set. The percent coverage is calculated with

```
tested = total – one-line
```

and

```
untested = (total – untestable) - tested
```

To further improve the versatility of JBlanket, specific files can either be excluded from or included in coverage data. This feature allows combinations of multiple sub-packages to be measured separately, which is useful for targeting parts of a system.

Coverage results are reported in an HTML format similar to that of JUnit reports. Since users are required to implement JUnit test cases prior to running JBlanket, it was decided that mimicking JUnit's reports would simplify the use of the JBlanket reports by reducing the amount of time users need to understand and interpret the results and learn to navigate between reports.

3.2 System Architecture

JBlanket is implemented in Java, and was designed to be executed from either the command-line or integrated with Jakarta's Ant. There are four main tasks that need to be executed to calculate coverage. The first task uses LOCC (supplied by CSDL) to create a file containing the total methods included in the coverage measurement. By using the "javamethod" value for "sizetype", all method type-signatures are located in the source code and stored in the first output set mentioned in the previous section.

The second task in JBlanket modifies the byte code created from compiling the source code. The Jakarta Byte Code Engineering Library (BCEL) is used to alter each method such that when a method is executed for the first time during unit testing, its type-signature will be stored. Before each method is modified, it is checked for two separate conditions. The first condition is if the method can be invoked or should be included in coverage. If a method falls into either category it is not modified and placed in the second output set. The other condition relies on the number of lines of code the method contains. If methods containing a single line of code are to be excluded from coverage, then one-line methods are not modified and recorded in the optional fifth output set. The second output set is immediately removed from the first output set, creating a modified first output set, and is not included in the coverage measurement.

The third task to perform is executing the JUnit test cases. Prerequisite setup tasks may be needed depending upon how the system is tested. For example, the modified class files can be packaged into JAR files before running the JUnit tests when the JUnit classpath does not include the directory containing the modified class files. A WAR file can be created and copied to Tomcat, or the modified class files can be copied to one of the "classes" subdirectories in the Tomcat directory. Then Tomcat can then be launched for client-server systems. This third task outputs the third set that consists of methods invoked during testing.

After the JUnit tests are executed, the final task can be performed. This is the report task that interprets all of the accumulated results. First the fourth output set is created from the difference of the first set and the combination of the third set and the optional fifth set. Then these sets of raw coverage data are aggregated into one XML file, where each method is stored according to the package-prefixed name of its class. Therefore, each package-prefix class contains at most three different method classifications (tested, untested, one-line) under which the corresponding type-signatures are stored. This file is then transformed into HTML through XSL Transformations (XSLT).

Before JBlanket can be used, the javac "debug" option must be turned on when compiling the source code. The debug option ensures that line numbers from source code is included in the byte code. Without line numbers, BCEL cannot calculate the lines of code in a method in the second task. In addition, to ensure that the Java ClassLoader loads the correct class, it is recommended that all copies of the compiled class files be removed from the classpath, including the original compiled class files prior to modification. If these files linger there is no guarantee that the ClassLoader finds the modified class file. No data can be collected from invoking methods from unmodified classes. Finally, for coverage data to be reliable, all test cases need to pass. It is possible for JBlanket to calculate coverage when some test cases succeed and others fail. However, this measurement will not reflect the true coverage of the system.

In the next section, the evaluation of the research questions introduced in the first section are tested and evaluated using JBlanket.

4. Evaluation

4.1. Experimental Methodology

This research will be evaluated in an academic environment through undergraduates in an upper-division, second-semester Software Engineering course at the University of Hawai'i. There are approximately 14 students in this class, all of who are developing eight separate web services (with Java 1.4 and JSP) that will be deployed on the Information and Computer Science (ICS) home page (http://www.ics.hawaii.edu) next semester, Spring 2003. The combination of these web services is refered to as CREST. Due to the nature of their projects, it was assumed at the beginning of the semester that each student had either enrolled in the previous semester's Software Engineering course, or adequate knowledge of Java, JSP, Jakarta Tomcat, CVS, JUnit, HttpUnit, and Jakarta Ant.

The students were first given time to accustom themselves to the course and their projects for the first two months of the semester. Seven projects were assigned to teams of two people. The remaining project was assigned to one person, who is also a member of one of the aforementioned teams.

Then they were given a questionnaire to judge their current practices and beliefs towards unit testing. After the professor collected the completed questionnaires, the author presented a 20-minute introduction to JBlanket – a description of the system, how to run it with their unit tests, and how to use the output to increase their coverages. The student's Ant build files were previously modified to run the JBlanket system to eliminate the effort to include the JBlanket target(s) by the students.

After the introductory presentation, the professor instructed the students that they would be required to reach and maintain 100% "feasible" method-level coverage. By requiring the students to reach a specific level, there was an increased likelihood of discovering whether it is difficult to reach the 100% and the amount of work it takes to maintain such a high value. At the end of the semester, the students were told, they will once again fill out another survey to find out their reactions to method-level coverage.

In addition to what the students were told, their individual projects were downloaded from a shared CVS repository. For the first two weeks of November, their projects were downloaded every day at approximately the same time. These initial downloads were needed to decide upon the best course of action with respect to finding a schedule that would best reflected the students' effort and changes in their projects' coverage. It was found that checking the projects every day did not result in finding many changes. While the repository of a couple of projects did change daily, most did not, and those changes did not appear to be extremely significant. For example, one or two files changed, and the resulting coverage was modified less than 5%. Therefore, it was decided that projects would be checked once every three days.

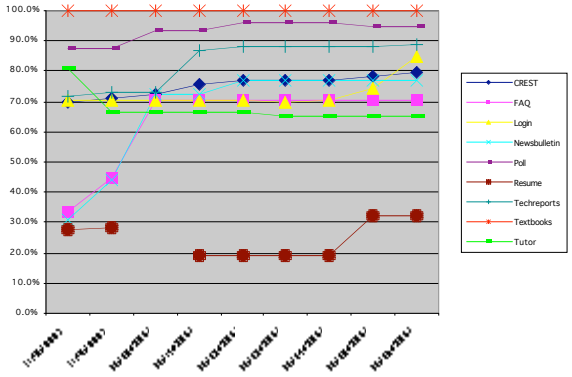

Figure 1. Daily checks of CREST coverage.

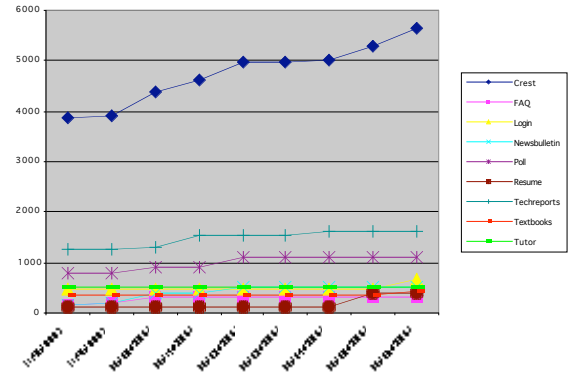

Figure 2. Daily checks of CREST LOC.

4.2. Results

4.2.1. Preliminary Results. While preparing for the introductory presentation, the author found a couple of interesting unexpected results when running JBlanket. Initially, the process of invoking the coverage system was integrated into the build file in a local copy of CREST that was checked out from the CVS repository. After reviewing results from "pure" method-level coverage, the first trend noticed was that most unexecuted methods began with "get" or "set", i.e., they were accessor methods or modification methods. Because majority of these methods are expected to contain one line of code, 'return foo' or 'this.foo = foo' for example, it was concluded that these methods were probably best tested via inspection versus writing individual test cases that would eventually exercise all of these methods. This line of reasoning was then extended to include all methods containing one line of code (one line methods) when a similar trend was noticed in other untested methods. As a result, JBlanket was modified to discard one line methods from the resulting coverage percent upon request.

However, these methods appear in the final reports so that the user would be able to review all excluded methods, and individually inspect them as needed.

A second unexpected discovery occurred when the JUnit tests were included in the coverage data. At first, the debate was whether inclusion of these classes was feasible. In the end, would users be expected to write test cases for their test cases so that they would not cause the level of coverage to drop? This question ended up being a moot point. What the author found was that by including the JUnit test classes in the coverage information, they were themselves checked for possible errors, including spelling of the method names. In JUnit, the test cases need to begin with "test". If a test case does not begin with this prefix, then the test case is never executed. This is a fact that all JUnit testers are affected by. For example, one of the misspelled methods was "tstFoo". After running all of the test classes, there was no indication that anything was wrong because all tests passed. So the tester moved on to create more test classes. It wasn't until after perusing the JBlanket reports that the tester was notified of the erroneous prefix. Therefore, by using JBlanket, the tester was notified right away when the test cases themselves did not achieve 100% coverage as expected.

4.2.2. Intermediate Results. After the evaluators were notified of their required use of the system, one expected result was that the students would start to refactor and modify their code with the intention of improving their coverage. All of their initial coverage levels were between the 20-50% range. (However, it should be noted that a bug existed in that version of JBlanket such that in some cases, the files containing the method information would overwrite themselves, or methods would not be counted at all because unneeded JAR files lingered from a previous build of part of the system.) One of the methods that was constantly untested was the main method included in test classes. This method is intended for use whenever a tester wants to run a single test class instead of all of the JUnit tests. A typical main method looked like:

```
public static void main(String[] args) {
 System.out.println("Testing Foo");
 //Runs all methods starting with "test".
 TestRunner.run(new TestSuite(TestFoo.class));
}
```

As it stands, this method is considered to contain two lines of code. One solution was to include another command in JBlanket that would specifically exclude this method from coverage. However, students found that if the System.out.println method was commented out, this method was reduced to one line of code. Therefore, it was excluded from coverage.

4.2.3. Final Results. During this study, the size of the 8 services grew from a range of 1786 to 3986 LOC to the range of 1951 to 5379 LOC. (See Figure 3) The coverage of the 8 services grew from a range of 27.5% to 100% to the range of 94.9% to 100%. (See Figure 4) Within the time span of this study, 6 services were able to reach 100% method-level coverage. However, only 5 of the 8 services reached the 100% coverage at the end of this study. Two services missed complete extreme coverage by 1 method.

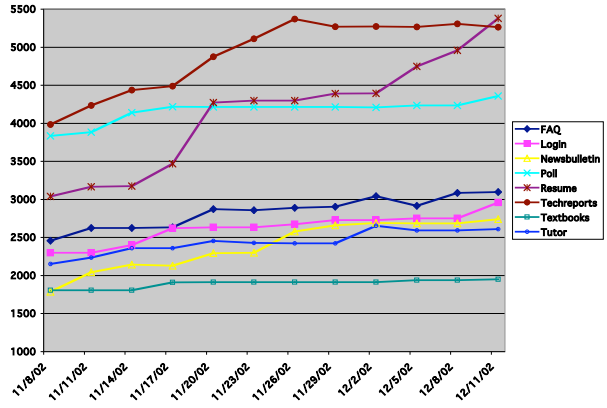

Figure 3. LOC for CREST services.

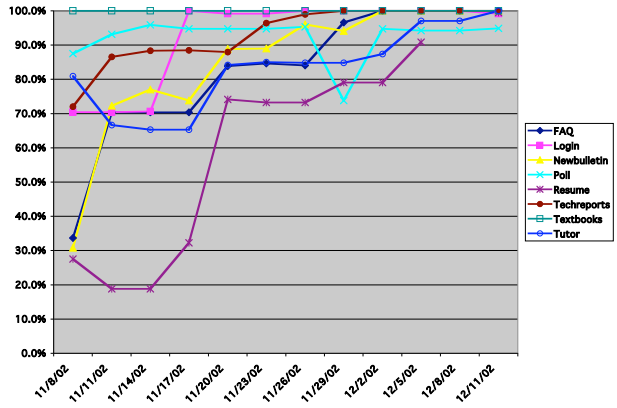

Figure 4. JBlanket coverage for CREST services.

Similar to the findings in [14] that measured block coverage, decision (branch) coverage, c-use (computational expression use) coverage, and p-use (predicate use) coverage on C programs, the levels of coverage obtained by each service were not always consistent. For example, the coverage for the Resume service initially decreased before increasing. The same drop in coverage (over 5%) occurred in the Poll and Tutor services, but at different times. Other services like FAQ, Newsbulletin, Login, and Techreports experienced a continuous increase in coverage with drops of less than 5%. Textbooks were able to maintain their 100% coverage throughout the study.

Table 1 displays the changes observed in the 8 CREST services. The change in total LOC is calculated as the difference between the final total LOC and the

starting total LOC divided by the beginning total LOC. Test LOC was calculated similarly. Services marked with a '*' indicate those that achieved 100% extreme coverage at the end of this study. The service marked with a '+' indicates the service that achieved 100% extreme coverage, but was not able to maintain it.

Interestingly, the Resume service had the most change in coverage, total LOC, and test LOC, but missed total extreme coverage by 1 method. The Textbooks service was able to maintain total extreme coverage with the least amount of change in total LOC but not in test LOC. Only the Tutor service experienced a negative change in test LOC while increasing the amount of total LOC and reaching 100% extreme coverage.

Service	% Change Coverage	% Change Total LOC	% Change Test LOC
Resume	72.0	76.9	31.0
Newsbulletin*	69.2	53.4	26.2
FAQ*	66.3	26.0	11.1
Techreports*	28.0	32.1	8.7
Poll	7.4	13.6	7.1
Login+	28.8	28.7	6.6
Textbooks*	0.0	8.0	3.3
Tutor*	19.0	21.2	-0.1

Table 1. Percent change in coverage, total LOC, and test LOC of CREST services

Figure 5 is a graph representation of the data in Table 1. From the graph, the only noticeable pattern for majority of the services that reached 100% extreme coverage and the change in their test LOC is the increase in test code. For those services (FAQ, Newsbulletin, and Techreports) the change in test LOC was less than half the change in total LOC. The other two services displayed unique behaviors, as mentioned previously.

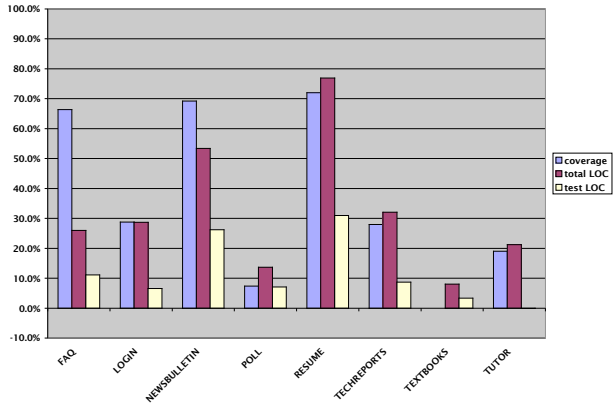

Figure 5. Change in coverage, total LOC, and test LOC.

5. Contributions and future directions

Knowledge of method-level coverage through use of JBlanket was helpful by the students in the ICS 414 class. They were able to discover how thorough their unit tests were. Students found that results indicated which methods were not invoked during unit testing where they previously thought "it was testing everything". Therefore, they were able to improve their test cases to include those methods and remove methods that were on longer needed.

Achieving 100% pure method-level coverage was not found to be a reasonable goal. Therefore, the goal was modified to 100% extreme coverage. Those students that achieved total extreme coverage increased their test LOC by less than 27%. One team reduced the amount of their test LOC by 0.1%. With respect to maintaining the 100% extreme coverage, as can be seen from Figure 3 and 4, once a team reached the goal, their total LOC did not change beyond 500 LOC. This behavior could be attributed to the maturity of the service or the fear from programmers of dropping their coverage. Only one service obtained total extreme coverage and was not able to maintain it. Therefore, the effort needed to maintain total coverage remains unknown.

When armed with knowledge of their method-level coverage, the size of services with respect to total LOC did increase, some more than others. The increases can be due to the state of the service when JBlanket was introduced. If a service were mature, i.e., almost complete for the semester, its total LOC would not dramatically increase. For example, the total LOC for the Textbook service increased only 8% while the Resume service total LOC increased 76.9%, implying the Textbook service was more mature. However, it is not clear that the amount of test LOC change could also be attributed to a service's maturity. For example, the Tutor service decreased its test LOC by 0.1% with an increase in its total LOC by of 21.2% while the test LOC for the Textbook service increased 3.3% with 8% total LOC change. Therefore, knowing method-level coverage does appear to influence the implementation (and testing) of software.

Most reactions from the ICS 414 students were favorable to the idea of using both extreme coverage and JBlanket. While they understood that total extreme coverage "does not mean the system is fault-free", they claimed that knowing their service's coverage was helpful in designing their unit tests and that invoking every method was important. Interestingly, although extreme coverage was used instead of pure method-level coverage, one team still insisted that they "didn't think 100% method coverage [was] necessary." Nonetheless, extreme coverage can be considered to be feasible because its measurement rules are flexible for adaptation to different situations.

Extreme coverage is an immature concept that has quickly evolved throughout this study. With further research, this concept can be improved to include more

precise rules that improve the accuracy of its measurement.

Future research includes conducting a study in which a system is measured from the very beginning. This would more accurately indicate the difficulty of obtaining and maintaining the 100% method-level coverage. By conducting this type of study within companies of various sizes, a better indication of the effects method-level coverage on software quality.

Furthermore, with each system that uses JBlanket, the more refined rules become in which measure coverage. For example, from this study, it was discovered that one possible future enhancement to the system would be a method call tree whereby users will be able to easily identify "hotspots" from the unexercised methods, and thus increase coverage by exercising a maximum number of methods with the least amount of test case modifications. This call tree would reduce the amount of time it could take a tester to scan through all of the untested methods and decide which method should be tested next.

However, before the concept of extreme coverage can be refined, the tool used for measurement also needs to be improved. For example, the process of integrating JBlanket with another system needs to be simplified to encourage others to use it. This could be achieved by reducing the amount of steps for running the system. Currently, the process of executing JBlanket is separated into four steps. These steps could be reduced if LOCC is integrated into JBlanket. By removing the need to run LOCC separately, the user would only need a system's byte code, improving the versatility of the system.

6. Acknowledgments

I would like to thank Professor Philip Johnson for providing guidance throughout this research project and allowing me to deploy the JBlanket system in his second-semester undergraduate-level Software Engineering class (ICS 414).

I would like to thank the students in the ICS 414 for using JBlanket and providing the data that made this research possible.

Finally, I would like to thank the students in the graduate-level Special Topics class (ICS 691) for providing feedback on the contents of this research project.

References

[1] Clover: A Code Coverage Tool for Java. Online at http://www.thecortex.net/clover.

[2] JCover. Java Code Coverage Testing and Analysis. Online at http://www.codework.com/JCover/product.html.

[3] Glass JAR Toolkit. Online at http://glassjartoolkit.com/gjtk.html.

[4] Optimizeit suite: Code coverage. Onilne at http://www.borland.com/optimizeit/code_coverage/innex.html.

[5] JUnit Quilt. Online at http://quilt.sourceforge.net/overview.html.

[6] Myers, Glenford. *The Art of Software Testing*, John Wiley & Sons, New York, 1979.

[7] Beizer, Boris. *Software Testing Techniques*, second edition, Van Nostrand Reinhold, New York, 1990.

[8] Highsmith, Jim. *Agile Project Management Advisory Service White Paper*, January 2002. Online at Online at http://www.sdmagazine.com/documents/s=7147/sdm0206a/sdm0206a.htm.

[9] Crispin, Lisa. *How XP Solves Testing and Quality Assurance Problems*, January 2002. Online at Online at http://www.xptester.org/_ZABLE[0]_/tab/9/excerpts/xpsolvesqa.htm.

[10] Bossi, Piergiuliano and Cirillo, *Francesco. Repo Margining System: Applying XP in the Financial Industry*, November 2002. Online at Online at http://www.agilealliance.org/articles/articles/RepoMarginingSystem.pdf.

[11] Kaner, Cem, Falk, Jack and Nguyen, Hung Quoc. *Testing Computer Software*, John Wiley & Sons, Inc., New York, 1999.

[12] Marick, Brian. Experience With the Cost of Different Coverage Goals For Testing. In *Proceedings of the Ninth Pacific Northwest Software Quality Conference*, pages 147-164, Portland, Oregon, 1991. Also online at <http://www.testing.com/writings/experience.pdf>.

[13] Rothermel, Gregg. Test Case Prioritization: A Family of Empirical Studies. In *IEEE Transactions on Software Engineering*, pages 159-182, vol. 28, No. 2, February 2002.

[14] Horgan, Joseph, London, Saul and Bellcore, Michael. Achieving Software Quality with Testing Coverage Measures. In *Computer*, pages 60-69, September 1994.

[15] Kaner, Cem. Software Negligence and Testing Coverage. In *Software QA Quaterly*, page 18, vol. 2, No. 2, 1995

[16] Piwowarshi, Paul, Ohba, Mitsuru and Caruso, Joe. Coverage Measurement Experience During Function test. In *IEEE*, pages 287-299, 1993.

[17] Marick, Brian. *How to Misuse Code Coverage*, 1997. Online at http://www.testing.com/writings/coverage.pdf.

[18] Glass, Robert. Persistent Software Errors. In *IEEE transactions on Software Engineering*, pages 162-168, vol. SE-7, No. 2, March 1981.

[19] Cornett, Steve. Code Coverage Analysis. Online at http://www.bullseye.com/coverage.html