Keeping the coverage green: Investigating the cost and quality of testing in agile development

Philip M. Johnson Joy M. Agustin
Collaborative Software Development Laboratory
Department of Information and Computer Sciences
University of Hawai'i
Honolulu, HI 96822
johnson@hawaii.edu

Abstract

An essential component of agile methods such as Extreme Programming is a suite of test cases that is incrementally built and maintained throughout development. This paper presents research exploring two questions regarding testing in these agile contexts. First, is there a way to validate the quality of test case suites in a manner compatible with agile development methods? Second, is there a way to assess and monitor the costs of agile test case development and maintenance? In this paper, we present the results of our recent research on these issues. Our results include a measure called XC (for Extreme Coverage) which is implemented in a system called JBlanket. XC is designed to support validation of the test-driven design methodology used in agile development. We describe how XC and JBlanket differ from other coverage measures and tools, assess their feasibility through a case study in a classroom setting, assess its external validity on a set of open source systems, and illustrate how to incorporate XC into a more global measure of testing cost and quality called Unit Test Dynamics (UTD). We conclude with suggested research directions building upon these findings to improve agile methods and tools.

1. Introduction

Agile methods such as Extreme Programming place great emphasis on the incremental development and frequent use of a suite of automated regression tests [2, 6]. Lightweight, freely available testing tool support such as JUnit has been integral to the adoption and success of this practice. It makes possible another XP mandate, that 100% of the test cases pass on a daily basis. This is often referred to as "keeping the bar green", a reference to the feedback given by the JUnit GUI when all the tests pass [13].

Of course, the fact that all the test cases pass has little bearing on software quality unless they are well designed and comprehensive. To obtain the latter, practitioners advocate "test-first programming" (recently upgraded to "test-driven development"), in which tests corresponding to a new feature are written before beginning its implementation [3]. The approach forces the developer to focus on the external client API of a feature first, and to produce a suite of tests corresponding to each feature. Advocates claim that following this method will produce not only comprehensive tests, but even more importantly, a better design.

Over the past two years, we have been incorporating various agile practices into our research, development, and teaching activities. While we have been impressed by the potential of many of the techniques, our experiences motivated research on the following two questions:

1. *Is there a way to validate the quality of test case suites in a manner compatible with agile development methods?* While agile methods such as test-drive development claim to result in high quality test cases, it is not obvious that following this method of writing tests prior to writing code is in itself sufficient to produce a high quality test suite. An independent, orthogonal measure of test case quality that is compatible with agile development practices would be useful, not only as an independent measure of test suite quality, but also as a diagnostic tool to help developers uncover areas of insufficient testing.

2. *Is there a way to assess and monitor the costs of agile test case development and maintenance?* Agile testing practices appear to generate a large amount of test code. According to some practitioners, the size of test code frequently equals the size of the system itself [3]. The development and especially the maintenance of

such code over time could potentially incur substantial overhead on development. Since speed of development is advocated as a virtue of agile methods, better understanding of the costs and benefits of large test suites would be a good first step toward determining if more cost-effective approaches might exist.

The remainder of this paper is organized as follows.

Section 2 presents XC (Extreme Coverage), a measure we developed for assessing test quality in agile contexts. XC differs in interesting ways from conventional coverage measures. For example, we desired a measure in which "keeping the coverage green", or achieving and maintaining 100% test coverage, was both feasible and practical in agile development. This led us to design a coverage measure that is substantially more coarse-grained than those advocated for use in traditional contexts.

Section 3 presents a tool called JBlanket that we developed for measuring XC on Java-based systems [1], and the design trade-offs and improvements that have occurred during the two years of its development and use.

Section 4 presents a case study we performed in a classroom setting to assess the feasibility and practicality of the XC measure. We found evidence that student developers can achieve 100% XC, and that XC can serve as a useful measure of test case quality.

Section 5 presents a case study in which we measured XC on a set of open source tools to see if such a coarse-grained measure could have value outside an agile context. None of the tools we investigated had 100% XC, providing evidence that this measure has value for test case validation outside the classroom setting.

Section 6 presents a measure that combines XC, developer test effort, and test code size data together to provide a more global perspective on the costs and benefits of testing over the life of a system. We call this composite measurement approach UTD, for "Unit Test Dynamics" (UTD). We show how implementing UTD using the Hackystat system [12] makes it practical to obtain in an agile context, and illustrate the utility of the measure through a case study.

Section 7 concludes the paper with a summary of the contributions made by this research and our views on how to build upon them to improve agile development methods through empirically based understanding of the strengths and weaknesses of their various practices.

2. Extreme Coverage

The development of a high quality automated test case suite is an essential goal of most agile development methods [15]. A well-established technique for assessing the quality of test suites is to measure its comprehensiveness in exercising the source code, otherwise known as "coverage". Many kinds of coverage measures have been proposed, including statement, branch, path, and loop [8], although statement coverage is the most common measure. A variety of research has investigated the relationship between coverage measures and code quality [5, 16, 18]. Some researchers advocate the use of coverage to drive the design of test suites [10], while others argue that coverage data should be restricted to use as a validation metric for some other test case design technique [14].

In the context of agile development, it is interesting to note that inconsistencies exist within various practices regarding statement coverage. On the one hand, one of the XP practices is "Test Everything That Could Possibly Break", which Kent Beck defines as follows:

> "You should test things that might break. If the code is so simple that it can't possibly break, and you measure that the code in question doesn't actually break in practice, then you shouldn't write a test for it." [2]

This seems to indicate that comprehensive coverage is not necessary. Indeed, it would be an error to achieve 100% statement coverage, since this would imply that "simple" code was tested. Other research on statement coverage in traditional testing contexts also indicate that less that 100% statement coverage might be most cost-effective [17].

On the other hand, another XP practice is Test-Driven Development (previously known as "Test-First Programming"), in which the developer designs a set of test cases for any given development task, which serves to both clarify design and external interface issues, as well as provide an unambiguous specification for when the task is completed. Beck states the following regarding TDD:

> "Test Driven Development followed religiously should result in 100% statement coverage." [3]

Unlike the first practice, TDD seems to indicate that a test suite that comprehensively exercises all of the code in the system including "simple" code is not an error, but rather the natural result of rigorous application of the method.

This inconsistency in practices within the same agile method regarding statement coverage is striking. Given the wide availability of coverage tools, it would also seem quite straightforward to gather the empirical data necessary to resolve this inconsistency. However, test coverage tools do not enjoy the same popularity as test invocation tools in agile contexts. Why might this be so?

We approached this question by investigating the features of the xUnit tools that seem to be of most value to agile practitioners, and using this list to generate an analogous

set of requirements for an "agile" test suite validation tool based upon coverage. Five major requirements emerged:

1. *Open source.* The agile development community shows a marked preference for open source tools, from the Eclipse editor to the JUnit test framework to the CVS configuration management system. Fee-for-use creates a substantial barrier to widespread adoption.

2. *Lightweight.* An essential characteristic of all agile methods, "lightweight" in this context of test case suite validation implies two things. First, it should easy to for the developer to introduce in a variety of development contexts. Second, the validation process should not significantly slow down development, even when applied many times per day as a natural adjunct to test case invocation.

3. *100% coverage as a practical, continuous goal.* Open source, lightweight testing tools such as JUnit enable agile methods to enforce a discipline of requiring all of the unit tests to pass all the time. Either all the unit test cases pass, in which case development proceeds, or one or more unit test cases fail, in which case development focusses on making the failing unit test(s) pass. A similar lack of ambiguity in interpretation would be a very desirable property of an agile coverage measure.

4. *Improves development.* Agile practitioners adopt unit test invocation tools and practices because the investment in test case development and maintenance appears to pay off in improved development speed and/or conformance to user requirements. One reason is because running test cases after each change tends to uncover introduced defects right after insertion, simplifying identification and removal. Another is that the availability of easily invoked test suites seems to reduce the cost of change by making ripple effects more apparent. The use of an agile coverage tool should appear to pay off in similar ways by increasing the quality and utility of test suites.

5. *Configurable and extensible.* Given the ill-defined understanding of the use and application of coverage measures in an agile context, the tool design should support customization to a broad family of coverage measures.

A google search on "code coverage tool" reveals a large number of commercial (Whitebox, Koalog Code Coverage, Clover, PureCoverage, Discover for Delphi) and open source (GCT, JDepend, JCoverage, Quilt) tools. While all of them address at least some of these requirements, none address all of them. The most problematic requirement, by far, is Number 3: 100% coverage.

This requirement is problematic for traditional coverage measures and tools because, as noted above, at least one agile practice recommends against testing code "that cannot possibly break." This results in the correct coverage level being less than 100% for all of the traditional coverage measures. Once a level less than 100% becomes the target, how does one decide what the appropriate level should be, and thus whether the current level is too low, or even too high? Indeed, an online workshop attended by agile researchers and practitioners raised this issue as an apparent barrier to use of coverage measures in agile development [7].

A central claim of our research is the following: in order to make coverage useful in a variety of agile contexts, we must explore new coverage measures that are significantly more coarse-grained than traditional measures. If we can "loosen" the coverage measure appropriately, then less than 100% coverage will indicate a clear failing in the comprehensiveness of the agile test suite, though achieving 100% coverage does not necessarily imply that the test case suite is optimal.

Our proposed measure, called XC (Extreme Coverage[1]) has a three part definition:

- *Method-level invocation.* Instead of tracking which statements are invoked during test case execution, XC checks only whether a method is invoked or not. The calculated percentage expresses the number of methods in the system that were invoked.

- *One-line methods ignored.* We heuristically identify "code that cannot possibly break" as methods containing only a single line. For example, get and set methods for instance fields typically contain one line of code, and are so simple that visual inspection can verify their correctness.

- *User-specified ignore list.* Finally, a user has the option of manually removing methods from the calculation of XC coverage, in order to define additional non-one line methods as belonging to the set "that cannot possibly break."

The XC measure departs dramatically from other coverage measures by being explicitly partial rather than striving for total comprehensiveness. Its goal is to be coarse enough so that "keeping the coverage green", or 100% XC, is a reasonable goal during developent. If agile developers have less than 100% XC, then this is a clear indication of the need for more test cases. XC is designed to minimize false negatives (i.e. a coverage value less than 100% even though testing is sufficient) at the expense of increased false positives (i.e. a coverage value of 100% even though testing is insufficient).

[1]The irony of calling such a coarse-grained measure "extreme" is not lost on us.

Unlike other coverage measures, XC has at least the potential to satisfy Requirement 3. However, it is unclear from the definition alone that it satisfies any of the five requirements. Evaluating XC against these requirements involves investigating a constellation of issues, including: Can the XC measure be implemented in an open source tool that is lightweight? Is the XC measure too coarse? In other words, is XC always at 100% regardless of the testing quality? Alternatively, is the XC measure not coarse enough? In other words, is it still too difficult to reach 100% XC under agile conditions?

3. JBlanket: A tool to measure XC for Java

To support evaluation of the XC measure, we have been developing a tool called JBlanket since 2001. JBlanket is developed and released under the Gnu Public License, satisfying the first requirement, open source distribution. As our research and teaching involves development of client-server web applications, supporting collection and merging of coverage data from multiple processes became an additional self-imposed requirement.

The first version of JBlanket used the LOCC source code size tool to determine the total size of the system, and the Java Debug Interface (JDI) to monitor method invocations on both client and server sides at run-time. JDI is a low-level interface to the Java run time environment designed to support interactive debuggers, and enables callback functions to be associated with a variety of run time events, including method invocation. After running the system under the control of the JDI, we implemented a post-processor program to combine the collected data from client and server together and determine the XC coverage by comparing this to the total system size. Unfortunately, we found that the use of JDI resulted in a 50-fold increase in execution time, an unacceptable behavior for a lightweight tool.

The second version of JBlanket attempted to modify the open source coverage tool Quilt to support the XC measure. Quilt implements a customized Java class loader that dynamically inserts bytecodes into class implementations as they are loaded. The modified classes collect coverage data. The Quilt design does not suffer the performance penalties imposed by the JDI run-time system. Unfortunately, the Tomcat web application system that we use for research and teaching uses its own custom class loader, and we were unable to successfully integrate these two class loading mechanisms together to support coverage.

The current version of JBlanket is inspired by Quilt, but instead of dynamically intercepting class implementations as they are loaded, JBlanket rewrites the class files prior to execution, inserting the byte codes required to instrument method invocations. At the same time that JBlanket modifies the class files, it also identifies all of the classes and methods in the system, eliminating the need to use a separate source code size tool. Modifying Java class files with instrumentation code prior to execution also eliminates incompatibilities due to non-standard class loaders. Finally, we implemented an HTML interface for reporting the XC measurements, with summaries for the system as a whole, and drilldowns for individual packages and classes. This interface is modelled on the JUnit reporting interface.

Figure 1 illustrates one such report screen providing a package-level summary of XC measurements. The report shows a listing of eight classes in the package hackystat.client.cli.shell. For each class, it lists the total number of methods in the class, the number of one line methods (which are thus excluded from XC coverage calculation), the number of remaining "multi-line" methods, the numbers of tested and untested multi-line methods and the resulting XC coverage percentage. Clicking on a class name drills down to a report on the class listing the names of all methods in the class and whether they were tested or not. Clicking on the "Home" link drills up to the top-level summary page, where aggregate coverage data for the entire system organized by package is displayed.

The current version of JBlanket also appears to satisfy the second requirement for a lightweight impact on development. The JBlanket distribution comes with Ant tasks that make it straightforward for developers to insert XC data collection into their development process. The instrumentation process incurs overhead only for newly compiled files, making it quite suited for an incremental development process. Run-time execution of test cases is increased by approximately 20-30%, which is significant, but still small enough in most cases to permit an agile development process requiring many test runs per day.

We began using JBlanket to support our development activities in the Summer of 2002, and have been enhancing it regularly since then. JBlanket has been used in several courses, and has been downloaded by students and developers both locally and externally. The current version is well documented and quite small—approximately 36 classes, 225 methods, and 3100 lines of code. The JavaDoc design and HTML versions of the source code are available for inspection on the web at its public distribution site [11]. The petite size of the system, its public availability and documentation, and the modifications we have already made to it provide evidence of configurability and extensibility, the fifth requirement. We invite its use, evaluation, and further enhancement by the broader software engineering community.

To verify the JBlanket implementation, we compared the XC coverage data it generates on several systems against the method coverage data generated on the same systems by Clover, another Java-based code coverage tool. Clover implements coverage by instrumenting source code rather

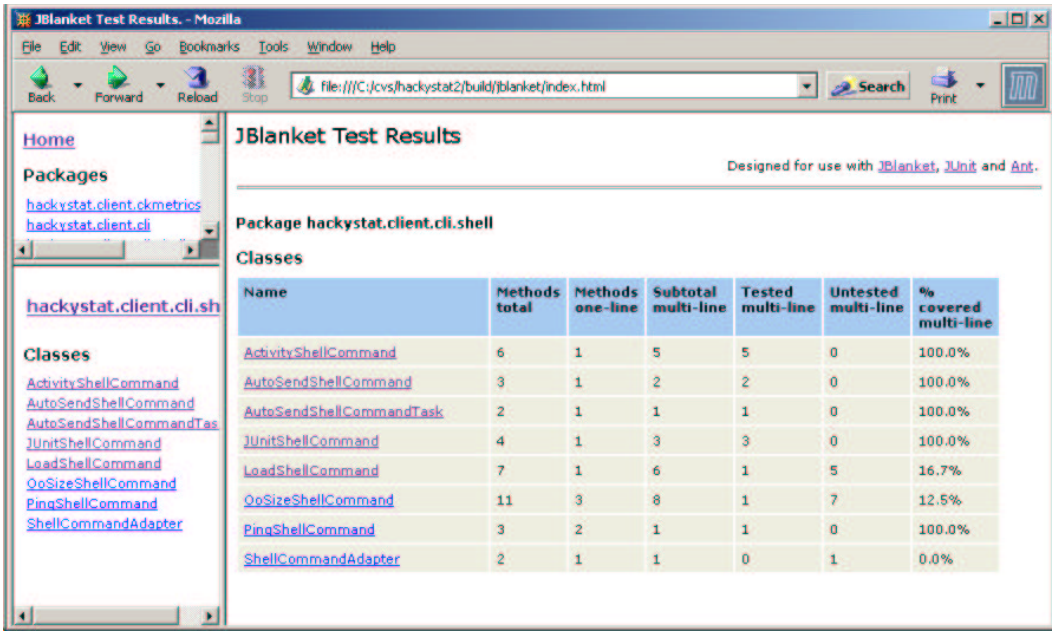

Figure 1. Example report from JBlanket providing XC measures for a Java package.

than the compiled byte code. Interestingly, there were differences in the data due to the different points at which the two tools insert their instrumentation code. In Java, if the user does not define a constructor for a class, a no-argument public constructor is implicitly defined and its byte code implementation exists in the class file. Since Clover rewrites source code, the implicit constructor is not instrumented. Since JBlanket rewrites byte codes, the implicit constructor is instrumented.

The two most important requirements remain to be evaluated, of course: is 100% XC a practical, continous goal, and does knowledge of the current XC value improve development? The next section presents a brief summary of a case study we performed to provide some initial insight into these questions. Details of the methodology and data collection are available in a thesis based on this research [1].

4. A case study of Extreme Coverage and JBlanket

In the Fall of 2002, we performed a case study of XC and JBlanket as part of a senior-level software engineering project course. The 13 students spent four months working on the development of eight interactive web services. All students worked in pairs, though some students worked on more than one service. The web services were designed to plug in to a framework system called CREST, which supports the functions of an academic department. For example, one service supports direct selling of textbooks between students, and another implements a technical report library. The students developed the software with agile practices such as pair programming, daily builds, and unit tests. They also used Java-based technologies advocated for agile development, including CVS, Ant, JUnit, and HttpUnit. The final size of each web service ranged between 2000 and 5500 non-comment source lines of code.

We collected both qualitative and quantitative data. The qualitatitive data took the form of an anonymous pre-test questionnaire, given approximately half way through the semester. The pre-test questionnaire asked the students to indicate their level of agreement with a set of questions assessing their attitude toward unit testing, such as: “Unit tests are very important for creating correctly functioning software”. Other questions assessed student opinions regarding the perceived difficulty of creating unit tests, and the quality of their current test suites.

We then introduced JBlanket into the development process by integrating it into the Ant build file associated with each project. From this point on to the end of the course five weeks later, students had continuous feedback on the XC coverage measure for their test suites. To reduce the possibility that failure on the part of a group to reach 100% XC coverage was simply due to disinterest, the instructor told the students that a portion of their project grade would be based upon their service’s final XC value.

At the end of the semester, we gave an anonymous post-test questionnaire including the same questions from the pre-test questionnaire, plus an additional open-ended question asking the students to comment on JBlanket. We used

a key system to associate each student's pre-test with their post-test while preserving anonymity. Comparing the pre-test to post-test answers provides some evidence that for this small group, their development experiences using JBlanket strengthened their belief in the importance of unit tests, and gave several students more confidence in the quality of their testing. With one exception, all of the students indicated that JBlanket data was helpful to them in writing unit tests to support correct functioning of their systems.

An open ended question soliciting comments on JBlanket was especially revealing. Several students indicated that XC data helped them improve their test suites. One student wrote, "[It] makes me feel safer to know I'm at 100%". A second said, "I write more unit tests to test more parts of the system." On the other hand, other students recognized the danger of relying too much on the XC measure. A third student wrote: "JBlanket is excellent! It helps me pinpoint packages which are inadequate. However once it was covered I gave little thought to conditional and branch coverage." A similar danger was also noted by a fourth student: "I don't think we tested out every little detail since we were just really looking to get the system to 100%."

Quantitative metrics collected once every three days from student activities included the size in LOC of test and non-test code, the number of methods exercised during testing, number of methods not exercised during testing, and the XC coverage. Over the five weeks, six out of eight reached 100% XC at least once, with the seventh reaching 99.5% XC (a single untested method remained). One system began at 100% XC and stayed there for the entire study period. Three began below 50% XC, and the remaining began between 70-90% XC. Five out of eight systems did not lose 100% XC once it was achieved.

Figure 2 summarizes some of the quantitative data for the eight services. It reveals that the introduction of the XC measure did lead to substantial coverage change in most systems; the only system without positive coverage change began with 100% XC.

Great care must be taken in the interpretation of this data, since this study suffers from substantial experimental limitations. The study size of 13 is too small for any statistical tests of significance. A very specialized type of software was under development, which might have influenced the data values. Most significantly, the study by design is susceptible to the Hawthorne effect: students knew that their use of JBlanket was under study, and knew that their grade would depend in part upon their XC measures. It is no wonder that XC values improved to at least some extent, and that even anonymous responses would indicate generally positive opinions about the technique.

Despite these limitations, this study contributes useful evidence that the XC measure is neither excessively coarse-grained nor excessively fine-grained. Had the measure been excessively coarse-grained, most or all of the projects would have started out at 100% XC and stayed there. Had the measure been excessively fine-grained, most or all of the projects would never have achieved 100% XC. Instead, most projects started at significantly less than 100% XC, and most projects successfully achieved 100% by the end of the study. We view the qualitative and quantitative data in this study as providing at least weak evidence that XC is both useful and practical in an agile context, and thus worthy of wider experimental investigation.

Service	XC	% Change		
		XC	Test LOC	Test Methods
FAQ	100.0%	66.3%	11.1%	6.6%
Login	99.2%	28.8%	6.6%	6.3%
Newsbulletin	100.0%	69.2%	26.2%	15.0%
Poll	94.9%	7.4%	7.1%	4.5%
Resume	99.5%	72.0%	31.0%	22.0%
Techreports	100.0%	28.0%	8.7%	2.8%
Textbooks	100.0%	0.0%	3.3%	1.5%
Tutor	100.0%	19.0%	-0.1%	3.1%

Figure 2. Summary statistics for the eight services

As an example of a follow-on study, we are deploying JBlanket in two software engineering courses in Fall of 2003. In this case study, we are instructing the students that they will be graded based upon the quality and comprehensiveness of their test case suite. JBlanket will be available to both the students and instructor as a tool to probe for weaknesses in the test case suite. We will collect data to see if 100% XC is achieved and maintained when the focus is directed toward quality and comprehensiveness, as opposed to simply achieving a certain value of the measurement.

5. XC on open source software

To provide some external validation of our claim that the XC measure is neither too coarse-grained nor fine-grained, we used JBlanket to compute XC on a sample of open source, Java-based systems with JUnit test cases. Most of the systems that we found were obtained from the Jakarta project. The goal of this study was to ensure that the variation in XC values that we observed in the classroom setting was also present in external settings. Figure 4 presents the results.

This data provides additional evidence in support of the usefulness of the XC measure, by showing that external systems also show wide variability in XC. For the eight systems studied, XC ranged from approximately 4% to 87% with

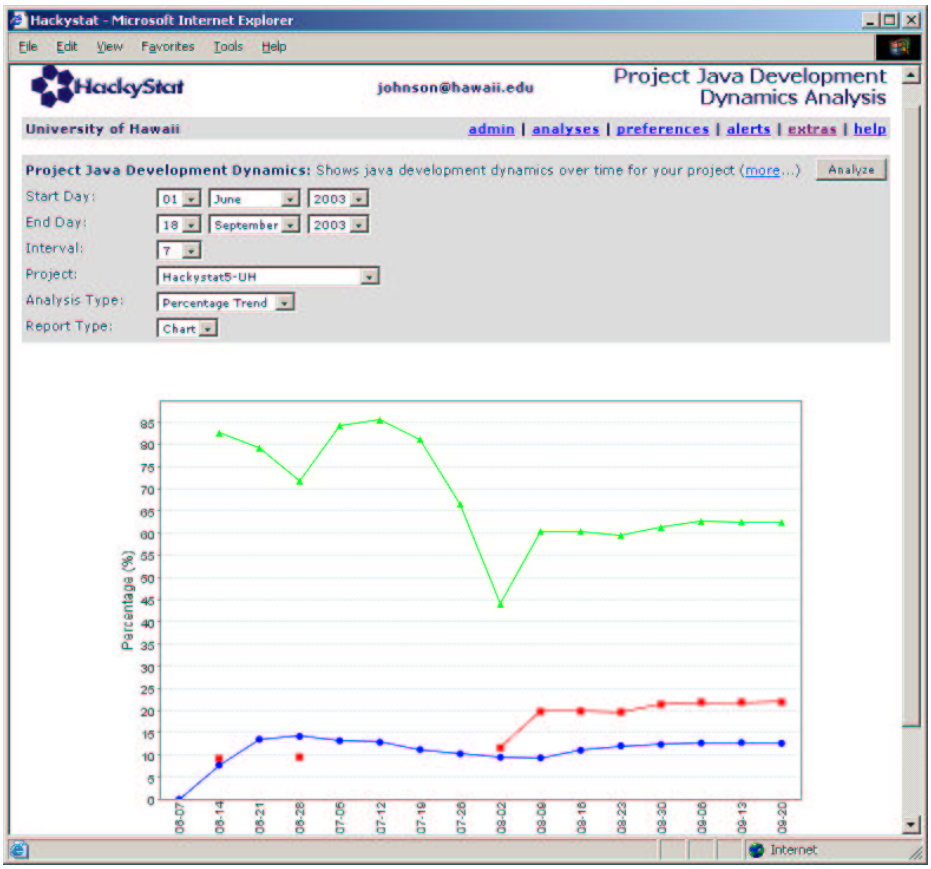

Figure 3. UTD data for the Hackystat project from June to September, 2003. The top line is the % coverage (XC), the middle line is the % system code devoted to testing, and the bottom line is the % of developer effort devoted to testing.

none achieving 100% XC. By adding the Covered and Uncovered numbers together and subtracting this result from the Total Methods, you can derive the number of one line methods for any of the systems. For example, Antelope had 53 (41 + 12) multi-line methods subject to XC calculation, indicating that more than half of the methods in the system were single line and thus excluded from the coverage measurement. All of the systems studied had substantial numbers of one line methods, indicating that this aspect of the XC definition has a significant impact on the way coverage is measured.

Finally, the low XC values found for external systems raise an interesting question: Do finer-grained coverage measurements actually add value, at least for these types of projects at this point in their evolution? Rather than research into more complex and sophisticated coverage tools, whose results are correspondingly more difficult to interpret, should we instead focus on making very simple coverage measurement tools as easily used and as widely available as current test invocation tools like JUnit?

6. Assessing agile test costs using Unit Test Dynamics

It is a widely acknowledged that testing and debugging consume significant development resources [4]. An industrial case study indicated that 10-25% of code in the systems studied were devoted to test cases [19]. The agile literature indicates a much higher proportion: 50% of the code in the system can be devoted to test cases [3]. A test case suite equal in size to the system itself must surely present significant development, maintenance, and quality challenges of its own. It creates at least the possibility of thrashing, where the impact of changes to the test case suite begins to inhibit the development and enhancement of the system itself. An important research issue for the agile community is to better understand of developmental dynamics of the test case suites created using their methods.

We have begun work on this research issue by using the Hackystat system to automatically collect and analyze test

System	Total Classes	Total Methods	XC		
			Covered	Uncovered	%
Antelope v2.55	96	112	41	12	77.4
Checkstyle v3.1 *	224	1437	944	327	74.3
Jakarta Commons BeanUtils v1.6.1	130	1526	688	363	65.5
Jakarta Commons CLI v1.0 *	33	307	103	122	45.8
Jakarta Commons Collections v2.1	540	5662	1432	2616	35.4
Jakarta Commons Sandbox Graph2	57	497	296	85	77.7
Jakarta Tomcat v4.1.27 Catalina subpackage	338	4306	105	2579	3.9
JWebUnit v1.1.1	30	443	224	33	87.2

Figure 4. XC data for a variety of open source Java systems

case cost and quality data over time. Hackystat is a system in which developers attach sensors to their tools which allow unobtrusive collection and analysis of development activities and products [9]. We designed a composite measure, called Unit Test Dynamics (UTD), and implemented it in the Hackystat system. UTD measures three values of a development project over time: the percentage of the system's code devoted to test cases, the percentage of developer effort devoted to test case development and maintenance, and XC: the percentage of code covered by the test cases. The unobtrusive nature of Hackystat data collection and analysis make it well suited to agile development contexts, where traditional, non-automated process and product data collection activities tend to be avoided due to their overhead.

To collect UTD, the developers of a system attach a Hackystat sensor to their editors. After configuration, the sensor automatically monitors the developers' activities within the editor and measures the effort devoted to development of both the test and non-test code for the system under study. A different sensor, after being attached to the system's build process and similarly configured, automatically collects the total size of the system, the percentage of it devoted to test code, and the value of XC at that point in time.

We began implementing the UTD analysis mechanisms for Hackystat in the Spring of 2003, and finished connecting the sensor to our automated build system in August. Figure 3 illustrates the UTD data collected from June to September for the six active developers on the Hackystat project. Due to build process issues, test code percentage data was collected only intermittently until August.

This analysis reveals several interesting features regarding testing in the Hackystat project during this period. First, XC has varied between a high of 85% and a low of 45%, with the final values at approximately 60%. A more detailed analysis indicates wide variability in the XC value: many of the older packages and classes have 100% XC, which is balanced by a significant amount of new, experimental code with low or zero XC. The plummet from 85% in mid-July corresponds to the incorporation of this new code into the Hackystat baseline.

On the other hand, the other two lines indicate that the cost of testing is currently quite low: around 20% of the total code is devoted to test cases, and less than 15% of developer effort over the three months was devoted to test code development and maintenance.

This UTD analysis indicates a decrease in software quality assurance over the past three months for this project. It reveals both the need for additional tests on existing code, as well as improved integration of testing into ongoing development so that incorporation of new code in future does not impact so negatively on XC. It also provides an interesting baseline: as we work to raise our XC measure to 100% and maintain it there, will the test code double or triple in size to comprise half the total size of the system? What will be the cost in terms of developer effort? Most importantly, will the resulting system be more robust and easier to enhance?

7. Contributions and Future Directions

We believe that our investigation into the two questions motivating this research has resulted in six contributions. First, it has produced a novel coverage measure, Extreme Coverage (XC), whose design is based upon requirements generated from unit test invocation tools of widespread use in agile development. The design of XC makes it more amenable to use in agile programming contexts, providing this community with a means to validate comprehensiveness of test case suites. Second, it has produced a novel open source implementation of XC, for Java systems called JBlanket that is available to the software development community for use, evaluation, and modification. Third, it has provided case study evidence from a classroom setting indicating that the XC measure is neither too coarse grained nor too fine grained for application in agile programming contexts, and that 100% XC is feasible and practical in an agile setting. Fourth, a case study of several other open source tools indicates that XC is a useful measure for test case suite validation. Fifth, it has produced a second novel measure, Unit Test Dynamics (UTD), which provides a way to understand the costs and benefits associated with agile test development and maintenance. Finally, a Hackystat implementation of UTD makes this measure practical within an agile context. UTD data for the hackystat project itself over a three month interval illustrates the application of the measure.

These contributions suggest a number of promising future research directions.

First, work needs to be done to resolve the inconsisten-

cies in the literature regarding statement-level coverage and agile practices. Should 100% statement level coverage be strived for, or avoided?

Second, should coverage be used as a driver for test case design, or only as a quality assurance metric for some other test case design method? Are some coverage measures (such as XC) more suited to "driving" test case design, while others (such as statement or branch) are more suited to validation?

Third, our UTD data provides evidence that very low levels of test effort during development can result in moderate levels of XC coverage. How much additional effort will be required to raise the XC level to 100%? We plan to investigate this issue during Fall of 2003 by raising the XC level in Hackystat to 100% and monitoring the resulting change in proportion of test code and test effort that results.

Finally, we hope that the availability of XC, JBlanket, UTD, and Hackystat along with our initial research results will encourage members of the agile community to begin collecting empirical data concerning the cost and quality of test suite development, and that this will result in new insight on how to make agile development faster, better, and cheaper.

8. Acknowledgments

We gratefully acknowledge the hardworking and disciplined students from our current and prior software engineering courses, and the code reviews, pair programming, and good advice provided by the Hackystat Hackers (Hongbing Kou, Cam Moore, Cedric Zhang, Aaron Kagawa, and Takuya Yamashita). Support for this research was provided in part by grants CCR98-04010 and CCR02-34568 from the National Science Foundation.

References

[1] J. Agustin. Improving software quality through extreme coverage with JBlanket. M.S. Thesis CSDL-02-06, Department of Information and Computer Sciences, University of Hawaii, Honolulu, Hawaii 96822, May 2003.

[2] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison Wesley, Massachusetts, 2000.

[3] K. Beck. *Test-Driven Development*. Addison Wesley, Massachusetts, 2003.

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

[5] P. Bishop. Estimating residual faults from code coverage. In *Proceedings of SAFECOMP 2002*, 2002.

[6] A. Cockburn. *Agile Software Development*. Addison Wesley, Massachusetts, 2002.

[7] P. Costa. Summary of the third eWorkshop on agile methods. http:fc-md.umd.edu/projects/Agile/3rdeWorkshop/summary3rdeWorksh.htm, October 2002. Fraunhofer Center for Experimental Software Engineering-Maryland.

[8] R. DeMillo, W. McCracken, R. Martin, and J. Passaume. *Software Testing and Evaluation*. Benjamin/Cummings, 1987.

[9] The Hackystat home page. http://csdl.ics.hawaii.edu/Tools/Hackystat.

[10] J. R. Horgan, S. London, and M. R. Lyu. Achieving Software Quality with Testing Coverage Measures. *Computer*, 27:60–69, September 1994.

[11] The JBlanket home page. http://csdl.ics.hawaii.edu/Tools/JBlanket/.

[12] P. M. Johnson, H. Kou, J. M. Agustin, C. Chan, C. A. Moore, J. Miglani, S. Zhen, and W. E. Doane. Beyond the personal software process: Metrics collection and analysis for the differently disciplined. In *Proceedings of the 2003 International Conference on Software Engineering*, Portland, Oregon, May 2003.

[13] The JUnit home page. http://www.junit.org/.

[14] B. Marick. How to misuse code coverage. Technical report, Reliable Software Technologies, http://www.testing.com/writings/coverage.pdf, 1997.

[15] B. Marick. Agile methods and agile testing. *Software Testing and Quality Engineering*, 3(5), 2001.

[16] J. Morgan and G. Knafl. Residual fault density prediction using regression methods. In *Proceedings of the Seventh International Symposium on Software Reliability Engineering*, 1996.

[17] P. Piwowarski, M. Ohba, and J. Caruso. Coverage measurement experience during function test. In *Proceedings of the 15th International Conference in Software Engineering*, pages 287–301, California, 1993. IEEE CS Press.

[18] K. Saileshwar and A. Mathur. On predicting reliability of modules using code coverage. In *Proceedings of the 1996 conference of the Centre for Advanced studies on collaborative research*, 1996.

[19] T. Yamaura. How to design practical test cases. *IEEE Software*, 15(6), November 1998.