Characterization of open-source applications and test suites

Sarojini Balasubramanian and Kristen R. Walcott

University of Colorado, Colorado Springs
sbalasub@uccs.edu, kwalcott@uccs.edu

Abstract

Software systems that meet the stakeholders needs and expectations is the ultimate objective of the software provider. Software testing is a critical phase in the software development lifecycle that is used to evaluate the software. Tests can be written by the testers or the automatic test generators in many different ways and with different goals. Yet, there is a lack of well-defined guidelines or a methodology to direct the testers to write tests

We want to understand how tests are written and why they may have been written that way. This work is a characterization study aimed at recognizing the factors that may have influenced the development of the test suite. We found that increasing the coverage of the test suites for applications with at least 500 test cases can make the test suites more costly. The correlation coeffieicent obtained was 0.543. The study also found that there is a positive correlation between the mutation score and the coverage score.

Keywords

Test Development, Open-Source Application Testing, Test Coverage, Mutation Testing

1. Introduction

Software is an integral part in many of the devices and systems that our society uses today. Software testing is the most prevalent approach in the industry that is used to evaluate the software. Testing activities can cost half the labor required to develop a working program [10]. In a typical commercial development organization, the cost of providing quality software via appropriate debugging, testing, and verification activities can easily range from 50 to 75 percent of the total development cost [19].

Despite these obvious efforts, software testing is not effective as it should be [22]. There is a lack of a commonly accepted standard for software testing that hampers efforts to test software and evaluate the tests results. There is no generalized practice of software testing that is followed across the industries. There are many standards and many practice communities specific to the domains [14]. There is a need for a well defined underlying methodology that can guide testers. This is required to understand how an ideal test case should look like or what should be the focus of a test case in a given situation.

To identify an effective test suite, it is important to measure its quality. There are studies that have provided empirical results supporting specific criteria as the best indicators of effectiveness. Test coverage is a promising metric to measure the test suites quality A coverage metric specifies which parts of program code should be executed (for example, all statements). Each coverage metric has varied criteria and therefore has specific requirements that a test suite should meet. When a test suite meets the requirements of coverage metric, testers assume it is able to find more faults than a random test suite that does not meet the requirement [5]. The second metric is the mutation score. The ability

of test suite to kill the generated mutants has been used to gauge the effectiveness of the test suite [7, 24].

Every tester may have an inherent style of his/her own in writing test cases. Given the wide variety of reasons why a test case might be written and also the different styles, one test suite may be better than the other. There is a need to characterize how real testers develop test suites to create them more systematically in the future.

In this work, we characterize the test suites of existing open source applications to better understand how developers traditionally write test cases. In this research, we study the relationship trends between widespread metrics in the source code and test code of existing test suites. We consider 24 applications sourced from repositories like GitHub, SF100, ApacheCommons and SIR. The metrics we studied include execution time, mutation score, coverage score, complexity and size of the source code and the test suites.

We learned that the test suite size was not much influenced by the complexity of the program. We found only a slight correlation of 0.116. Acceptably, the test suite execution time was not influenced much by the complexity of the program studied. Although, the test suite execution time can increase when the coverage is increased in bigger test suites (more than 500 tests). Both coverage and mutation scores were considered and compared.

In summary, the main contributions of this thesis are as follows:

- A comprehensive survey of previous studies that investigated the relationship between coverage, mutation score and effectiveness, other works that have done characterization type study for specific metrics.
- Empirical evidence that demonstrates the relationship between different metrics of the source code and test code.
- A discussion of the implications of these results for the testers, researchers and teachers.

2. Test Suite Characterization Framework

Here we explain the overall characterization study of our research. We study the software testing that is in practice and the need for a standard in this process. Then, the section describes the framework through which we aim to understand the testing process better. Finally, we discuss the metrics that will be measured and analyzed by the framework.

For better testing, we need effective test suites that can test the system thoroughly. In order to improve or redesign the software testing process, we need to have a understanding about the process. In this study, we hope to understand how to build an effective test suite by analyzing various factors that can shape a test suite's characteristics. We analyze the relationship trend between the program's attributes and its test suite's attributes. From this, we learn if there is a general test suite writing style across the targets and the domains of the open source world. Previous studies have characterized the software process or a software discipline as a whole [20] [30] or a specific software process [11] [34] [38]. In this work, we will determine how test suites vary between the case studies and discuss the relationship between the test suite and existing code based on a number of metrics. After we learn the characteristics, it is important to know why the test suites might be written that way. The second part of our study focuses in understanding the factors that might have the driven the testers during the test suite development.

The primary research interest here is to improve the test suite development process. We aim to achieve this through what we learn from the characterization framework. More specifically, we aim to help the future testers understand the testing process well.

To achieve our research goals, first, we built a metrics profile of the software and its associated test suite of the selected applications as seen in Table 2.

In order to see if there is a general test case writing style, we study the relationships between the metrics profiled in the combinations discussed below. In terms of test case cost, it can mainly depend on the executim time of the test case or the hardware cost or even the engineers' salary [39]. In this paper, we account for the execution time of the test case for the cost and study the factors that can increase the same. We have studied *execution time vs coverage* to see if targeting high coverage scores can affect the execution time of the test suites. Similarly, we study *execution time vs complexity* to see if the test suite of a more complex code can take more time to execute the test suites.

Test suite size grows in time. By studying the metrics in the fashion discussed here, we analyze the factors that can be potentially responsible for the size of the test suites during the development of the test suite. We have studied the relationship between *Number of tests vs complexity* to see if it can be an inherent nature of the testers to concentrate their test case writing effort on more complex code. Also, we explore *Number of tests vs Coverage* to understand if a high coverage target for the test suites can make them bigger in size.

In the second part of research, we aim to identify the potential focus of the tester in the test suite development. Test suite adequacy is commonly measured by the mutation score and coverage score. With this research we try to understand if these factors have driven the testers in reality. By studying the relationship trend in *coverage vs mutation score*, we see if the testers have targeted high coverage scores to achieve effective test suites. Similarly, we have considered *coverage vs complexity* to analyze if highly complex code are covered more during testing. We have also studied *Number of tests vs mutation score* to analyze if more tests are written to achieve effective test suites.

Analyzing such relationships between metrics can help us understand the kind of test suite can be developed given a high complex code and a standard coverage criteria.

We have a number of challenges in the characterization study. Sourced from a variety of repositories and domains, test suite writers may have different motivations. For example, in the avionics or automotive industries, testers may be most concerned with reaching the required statement coverage levels [12]. On the other hand, testers working with Quality analysis teams may be more concerned with covering features that the stakeholders have requested and use frequently [35]. As we are analyzing existing open source programs, no knowledge of the tools or motivations of the test writers can be assumed. It can be difficult to interpret the motiveof different stakeholders involved during the test suite development. It is a challenge to come up with the general motivations or targets of the testers despite the domain differences.

Another challenge here is to choose the suitable set of research questions to answer our characterization level goal. It takes a lot of time to come up with the right questions that meets our characterization goal. Also, this study's success largely depends on the selection of right tools and metrics for the analysis. It is important to evaluate the variety of coverage and mutation analysis tools available in the market to choose the tools that can best serve our goals. Metrics in this study are signals that can indicate a trend in the test suite writing practices. While we aim for lightweight metrics, we have to choose metrics that can provide insights about the test suites in the correlation analysis

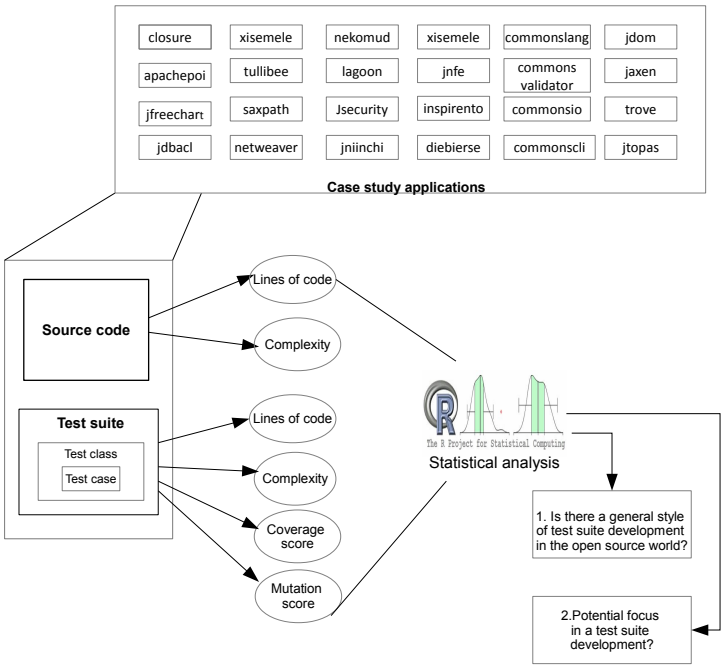

Figure 1: Test suite characterization framework

3. Evaluation

Here we will explain the case studies, metrics, and evaluation techniques used in this study.

3.1. Case Study Applications

The selection criteria for the programs used was based on open source programs developed in Java, the existence of associated JUnit suites and the size of the applications. Applications were selected from different size ranges and also with varied sized test suites.

Applications were selected from popular repositories including GitHub [1], SourceForge [2], and SIR [3], and SF100 [4]. The applications selected are listed in Table 1. The case study programs run across varied application domains. For example, Trove is a library that provides high speed regular and primitive collections for Java and hosted on SourceForge. Apache POI, is an open source API for manipulating Microsoft documents.

The size range of the applications also varies. We have selected very large application such as Closure compiler with 52KLOC, which is the largest application used in the study. SF100 application jniinchi is the smallest program used in our study with 783 LOC. Similarly, we have test suites ranging from 43KLOC in Closure to 108 lines in Diebierse.

Cyclomatic complexities of our programs ranged from 1.29 in xisemele to the highest of 3.79 in tullibee [36]. Modules exceeding a CCN of 10 were split into smaller modules. Considering that, our applications are not very complex, but there are various ranges of it. McCabe also proposed cyclomatic complexity as a measure to indicate the number of tests that will be required by the program [36]. One of our research question explores this concept to understand if the testers have an inherent nature to write more tests for complex code.

20 of our case study applications as listed in Table 1 were considered for the coverage study due to environmental constraints and issues in other applications. We have applications

Program	Statement coverage	Branch coverage	Mutation
jdbac	41%	29%	Mem err.
jfreechart	53%	45%	Mem err.
tullibee	6%	4%	Mem err.
xisemele	93%	93%	86%
saxpath	73%	58%	57&
netweaver	18%	2%	16%
nekomud	27%	7%	22%
lavalamp	96%	93%	72%
lagoon	19%	21%	20%
jsecurity	33%	24%	52%
jninchii	74%	56%	86%
jnfe	12%	25%	68%
inspirento	0%	13%	37%
diebierse	27%	20%	38%
commonslang	91%	98%	75%
commonsvalidator	83%	75%	54%
commonsio	76%	72%	90%
jdom	91%	86%	77%
jaxen	73%	59%	51%
trove	7%	7%	59%

Table 1: Case study applications and coverage scores

with branch coverage scores ranging from 98% to as low as 2%. The Apache commons application commonslang had the highest branch coverage of 98% and a statement coverage of 91%. The netweaver application had the lowest branch coverage of 2% and correpsondingly the test-LOC/source-LOC ratio in netweaver was very low, 0.07. Within the small range applications between 1KLOC - 1.5KLOC, the lavalamp application had the highest coverage of 96%. Among the moderate size applications, ranging from 2KLOC - 10KLOC, the commons-validator application had the highest coverage. The statement coverage of the commons-validator application was 83%.

Similarly, 18 of our applications were considered for the coverage study due to the memory buffer constraints caused by the tool we used. The case study applications considered for the study are provided the mutation scores in the table 1. Those with no scores were due to memory constraints. Mutation scores range from 16% - 90%. The application that has the highest mutation score is the commons-io; the netweaver application has the lowest mutation score. 33% of our applications have a mutation score greater than 75%.

3.2. LOC and Complexity

We have measured program and test attributes like size of the applications using Lines of Code (LOC) and the MCCabe's cyclomatic complexity in the study. The cyclomatic complexity of a section of source code is the count of the number of linearly independent paths through the source code. In practice, McCabe's Cyclomatic Complexity (CC) is a software complexity measured by examining the software programs flow graph. CC is calculated as: $CC = E - N + 2P$ where, E is the number of edges, N is the number of nodes, and P is the number of discrete connected components.

We have used JavaNCSS to gather Non Commenting Source Statements (NCSS) and

Case study application	**Lines of Source**	**Lines of Tests**	**Average complexity of program methods**
closure	52878	43271	3.26
apachepoi	63425	42296	2.14
jfreechart	67026	32128	2.61
jdbacl	13296	1224	2.5
xisemele	1399	2893	1.29
tullibee	2499	233	3.78
saxpath	1441	478	2.1
netweaver	17953	1363	2.82
nekomud	270	282	2.13
lavalamp	1039	1344	1.5
lagoon	6060	553	3.52
jsecurity	9470	1034	2.05
jninchii	783	1325	2.05
jnfe	1294	302	1.38
inspirento	1768	1842	1.76
diebierse	1539	108	1.74
commonslang	15825	28632	3.31
commonsvalidator	3709	4982	2.55
commonscli	1523	2237	2.69
commonsio	6037	12020	2.52
jdom	11133	17283	2.62
jaxen	8428	6415	3.21
trove	2080	10486	2.24
jtopas	3220	2996	2.36

Table 2: Characteristics of the applications used in the study

Cyclomatic Complexity Number (McCabe metric). JavaNCSS [13] is a simple command line utility which measures two standard source code metrics for the Java programming language. The metrics can be collected globally, for each class and/or for each function. The choice of JavaNCSS was made for two important reasons. First, it is a tool specific to Java and all our case study applications are limited to Java. Second, JavaNCSS measures logical lines of code. As JavaNCSS is a Java-specific tool, it is possible for the tool to read java code along the logic and express logical lines of code. The counter gets incremented for a logic (for example, a package declaration, method declaration, label etc). Physical source lines of code can vary significantly with the coder's style and the coding template followed. So it would be more apt to choose logical count (JavaNCSS) over physical count for our experiments which requires LOC metric.

3.3. Coverage

Code coverage measures the percentage of software that is exercised by testing process. It can be measured at various granularity like statement, branch level etc and is provided by the coverage tool. Coverage score is also considered as a measure to quantify the thoroughness of the white-box testing. We intend to use codecover to measure the code coverage in our study. JaCoCo [16] is an open source coverage library for Java, which has been created by the EclEmma team. JaCoCo reports code coverage analysis (For example, line, branch, instruction coverage) from the bytecode. JaCoCo comes with Ant tasks to launch Java programs with execution recording and for creating coverage reports from the recorded data. Execution data can be collected and managed with the Jacoco's coverage task. The standard ANT tasks to test applications in Java are junit or tests. To add coverage reports to these tasks, they are wrapped in the coverage task provided by JaCoCo. Reports in different formats are created with the report task in JaCoCo.

3.4. Mutation Score

Mutation score is the relation between the number of mutants killed by the test set and the the total number of mutants. Mutation score ranges between 0 and 1. When the mutation score is higher, a test suite is considered to be more efficient in finding faults. In this study, we have used Major [23] mutation framework to fault seed our case study applications and thereby collect their mutation scores. The following components enable Major as fundamental research tool on mutation testing as well as efficient mutation analysis of large software systems.

- An in-built compiler which aids the full fledged mutation analysis tools.
- Mutation analyzer which runs the junit tests iteratively to provide mutation analysis outputs.
- Domain specific language provides control to the user allowing to enable or disable mutation in specific parts of code.

3.5. Statistical Analyses

p-value : We have used correlation analysis to measure the relationship between metrics in our study. The Pearson correlation coeffcient determines if, and to what extent, a linear relation exists between two variables. It is the most common measure of dependence between two variables. Pearson correlation co-efficient (Pearson's r) is obtained by dividing the co-variance of the two variables by the product of their standard deviations.

Based on the co-efficient value obtained, the relationship between the two variables can be described as low, moderate or high. The Pearsons correlation is between -1 and 1. A value of 1 implies that a linear equation describes the relationship between X and Y perfectly, with all data points lying on a line for which Y increases as X increases. A value of -1 implies that all data points lie on a line for which Y decreases as X increases. A value of 0 implies that there is no linear correlation between the variables. We have used the standard Guilford scale for verbal description; this scale describes correlation of under 0.4 as "low", 0.4 to 0.7 as "moderate", 0.7 to 0.9 as "high" , and over 0.9 as "very high".

Kendall Tau : We have also calculated Rank correlation which is another form of correlation analysis between two variables. It differs from Pearson correlation by measuring a different type of association between two variables. Kendall's correlation co-efficient is similar to the Pearson correlation but does not assume that the variables are linearly related or that they are normally distributed. Rather, it measures how well an arbitrary monotonic function could fit the data. The Kendall rank correlation checks for the dependence between two variables based on ordered element. A rank correlation coefficient (Kendall's tau) measures the degree of similarity between two rankings, and can be used to assess the significance of the relation between them. The Kendall rank correlation checks for the dependence between two variables based on ordered elements.

We have performed correlation analysis, rank correlation and linear regression using R [31]. R language is one of the most widely used languages for statistical software development and data analysis.

4. Results and Discussion

Here we talk about the general experimental design followed to answer our research questions. In the next part, we discuss the results we have observed and discuss the answers to our research goals.

4.1. Experimental Design

The study required us to select a variety open source applications in Java. We have analyzed open source applications as listed in Table 2. Then, the LOC(lines of code) and cyclomatic complexity(CCN) for each of the program on the source code and the associated test suite was calculated. Number of tests for an application and the average complexity measurements were collected using the package level options available in the JavaNCSS. Shell scripts were used to run the JavaNCSS tool to collect LOC and the CCN on various levels of the source code and the test suites of all the programs. We generated mutants (faulty versions of the original subject programs) using Major, a tool for generating mutants for the Java programs. We used the Major's compiler option 'ALL' to generate mutants. The attribute set to 'ALL' during compilation generates mutants with all the mutant operators available with Major. Then, using a shell script; we ran every single test case on the mutant versions of all the programs and the outputs have been recorded. This helped us in calculating the mutation score in our research. We have measured the execution time of individual test cases by modifying the build file of the individual applications to print execution time summary. Execution time of the entire test suite of an application was gathered using the JUnit summary provided by running ant task for test. Coverage scores at the statement level and at the branch coverage were determined using a coverage tool called JaCoCo. The build files (build.xml) of the applications were modified to include the coverage tasks of JaCoCo. Finally, tests were run to collect the coverage reports.

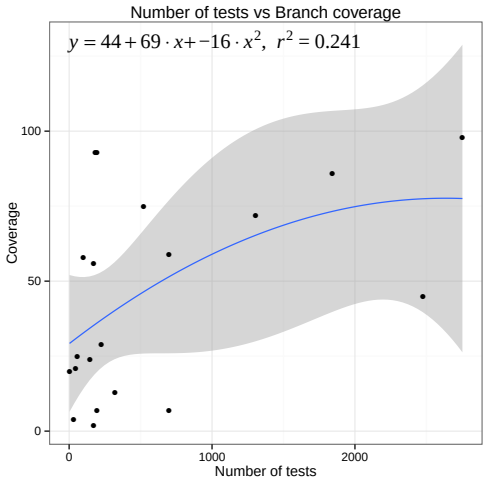

Figure 2: Number of tests compared to branch coverage for all the applications

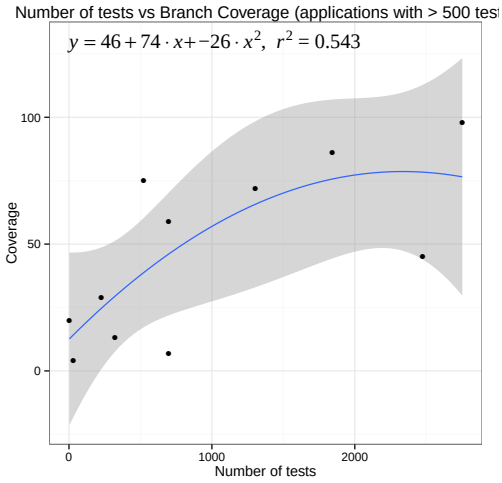

Figure 3: Number of tests compared to branch for applications with >500 tests

4.2. Test suite writing style in the open source applications

In this section we aim to find if there is a traditional test suite development style in the open source industry. We plan to answer our research goals with the following questions:

- *Is the test suite size driven by complexity or coverage score of the source code?*
- *What factors can influence the execution time of the test suite?*

The Table 2 illustrates the relationship between the test suite size (number of tests in the test suite) and the branch coverage for each program. The horizontal axis represents the number of tests while the vertical axis holds the branch coverage values . The Kendall coefficient obtained was 0.357. The Pearson's co-efficient value for this comparison is 0.241 indicating a positive but a low correlation between the coverage score and the number of test cases in the test suite. The line coverage demonstrates a trend similar to branch coverage. The correlation co-efficient achieved between the number of tests and line coverage was 0.202. The plots imply that both branch and line coverage have a slightly positive correlations and thus influence on the number of tests developed for an application.

However, we examined this relationship again by including the SF100 applications except for three applications (inspirento, diebierse and tullibee). The correlation value obtained from the Table 3 was 0.543 indicating a moderate correlation for the attributes under study. This can be due to the size of some of the test suites in the SF100 applications that ranged between 3 tests (diebierse) and 321 (inspirento) test cases. Hence in smaller applications with less than 500 number of test cases, the coverage scores might not influence the number of tests or vice versa. Although in applications with >500 tests, increasing coverage might lead to bigger test suites.

Hence in considerably bigger applications like apache commons, testers may have focused on increasing coverage which has made test suites grow in size. In order to analyze if bigger test suites can always provide higher coverage, we have to maintain the size of the application constant. To explore this is beyond the scope of our current study.

To study the effect of complexity of the application on the test suite size, we have measured the Pearsons correlation for the comparison. Graph4 illustrates the relationship between the size of the test suite and the complexity of the case study application. The horizontal

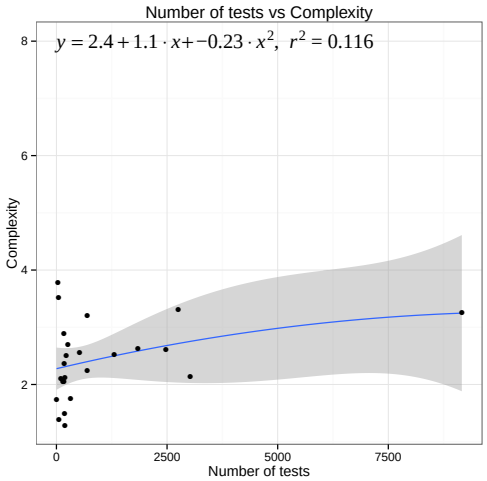

Figure 4: Number of tests compared to complexity of the programs

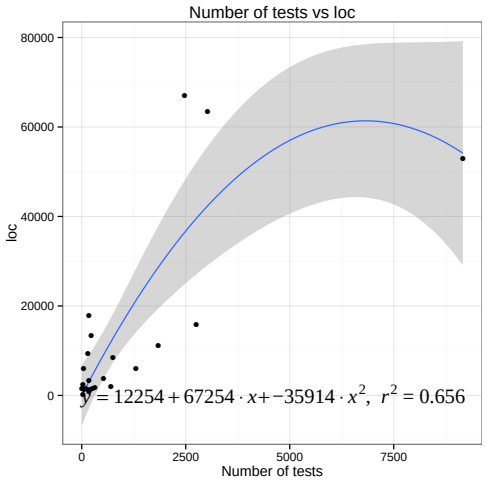

Figure 5: Number of tests compared to Lines of Code

axis represents the number of tests and the vertical axis represents the average method-level complexity of the application. The coverage score vs the complexity demonstrated a low correlation with the correlation co-efficient of 0.116. The Kendall tau value determined for this comparison was 0.241. We may say that the number of tests in a test suite is not strongly dictated by the complexity of the methods in an application. Even when we considered a subset of our case study applications, like the apache commons applications, we see that the number of tests does not grow significantly with the complexity of the code. The apache commons' applications had a cyclomatic complexity ranging from 2.52 to 3.31. We also tried the same comparison with applications ranging between 1KLOC-1.5KLOC and still obtained a very low correlation of 0.08.

Rather, the size of the application was straightforward indicator of the size of the test suite it had. Figure5 indicates that bigger applications have greater number of tests. The Pearson's correlation co-efficient considering all our case study applications for this relationship was found to be as high as 0.6. The correlation reveals that the application's size (lines of code) can drive the number of tests for it.

The complexity of an application was found to have only a slightly positive correlation(0.116) in influencing the number of tests. The slight correlation is also likely to be caused by the application's size that leads to higher number of tests.

We studied the effect of coverage and complexity on execution time of the test suites using the Pearson's and Kendall correlation. The Pearson's correlation for the test execution time versus the complexity obtained was 0.0649 as seen in Figure 6. The Kendall tau value 0.0917 obtained was in a similar trend as of the Pearson's co-efficient. This very low correlation is a likely consequence of our previous results (Number of tests vs complexity). When we compared number of tests and complexity, we inferred that complexity may not be an influential factor in the number of tests in an application. Hence, only a low correlation between the complexity of the source code and the execution time of the test suite was anticipated.This shows that the complexity of the application has not driven the execution time of the test suites in a significant way.

Similarly, we expected a low correlation between the coverage score of an application versus the execution time of the test suite. The correlation coefficient obtained was 0.249. The

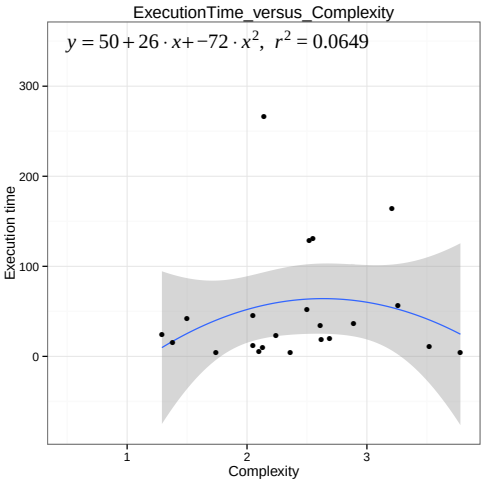

Figure 6: Execution time of the test suites compared to the complexity of the programs

Figure 7: Execution time compared to the branch coverage scores of the test suites

Kendall tau obtained was 0.317. Although in the number of tests vs coverage comparison, we found a moderate correlation between the coverage and the number of tests in a subset of applications. We did a similar comparison between the execution time and the coverage of applications with atleast 500 tests. We found the correlation coefficient to be 0.588 as seen in Table 7. Hence increasing the branch coverage in applications with more than 500 tests seems to increase the execution time of the test suites.

We infer that cyclomatic complexity has a very low correlation with the execution time. Complexity has not played a considerable role in increasing the test cost in terms of execution time. Whereas, coverage has a moderate correlation with the execution time. There was a significant increase in the execution time of the bigger test suites (more than 500 tests) with an increase in branch coverage.

4.3. Potential targets during test suite development

In this section, we aim to find the focus of the testers during test suite development by answering the research questions below.

- *Is the highly complex code covered more?*
- *Does the size of the test suite matter for its effectiveness?*
- *Is high coverage targeted to achieve high effectiveness?*

One of our research goal here is to analyze if the effectiveness of a test suite is influenced by the size of the test suite. Graph 8 combines some of the data we collected to answer this question. In each sub figure, the horizontal axis indicates suite size while the vertical axis shows the range of the mutation scores determined. These mutation scores range from 16% to as high as 90%, yielding a correlation coefficient r of 0.182. The kendall tau value was 0.264. This indicates that there is a a low positive correlation between effectiveness and size for these applications. But when we have to consider that we have ignored the size of the application (LOC) during this comparison. We had case study applications ranging from 270 lines of code to 17KLOC. The low correlation obtained was of course due to the varied size range of the applications.

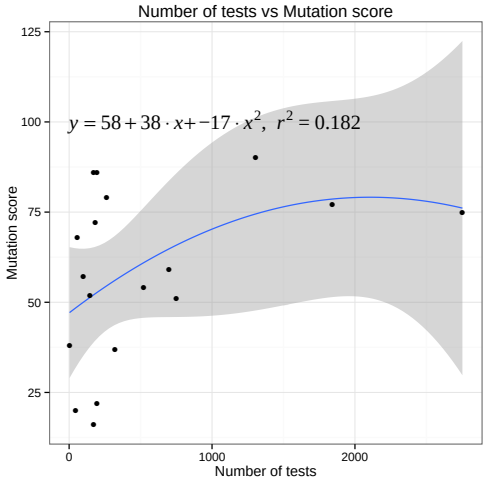

Figure 8: Number of tests compared to mutation score for all the applications

Figure 9: Number of tests compared to mutation score for applications with >500 tests

As a secondary analysis, we segregated the applications according to the size. We combined applications ranging between 2KLOC-10KLOC and recalculated the correlation co-efficient. The Pearson's correlation co-efficient obtained was 0.798 as seen in the Table 9. In this analysis, we found a high correlation between the mutation scores and the number of tests for an application. Fittingly, the ratio of test line of code to source lines of code was as high as1.9 for the commonsio application which has the highest mutation score of 90% in this category. Similarly, we combined applications between 1KLOC to 1.5KLOC and obtained a high correlation co-efficient of 0.76. Similarly, the test-LOC/source- LOC ratio was 2.0 for xisemele. Xisemele application had the highest mutation score among the applications that ranged from 1k to 1.5KLOC in size. Also, netweaver which has a poor test-LOC/ souce-LOC ratio of 0.076 has a very low mutation score of 16%.

A very low positive correlation was found between the effectiveness and the number of tests when the size of the application and test suite were ignored. However, when we combined applications with a similar size range, our results suggested that, for Java programs, there is a high correlation between the effectiveness of a test suite and the number of test methods it contains.

We compared coverage scores to the mutation scores to see if the tester's have focused on coverage to achieve high effectiveness. Comparing the coverage with the mutation scores for our set of application is a reality-check to reports that support coverage scores as an effectiveness indicator. We implemented the Pearson correlation analysis between the mutation scores and branch coverage for our case study applications 10 11. The Pearson's correlation computed to analyze this comparison was 0.574 for branch coverage and 0.474 for statement coverage. The Kendall correlation achieved was on the similar range (0.543), complimenting the results from the Pearson's correlation. Hence we see that for our set of case study applications there is a moderate correlation between coverage and mutation scores when number of tests in the suite is ignored. Although an overall positive correlation was achieved, we realized that applications that had a test suite with a greater branch coverage killed fewer mutants than test suites which had less coverage than that. For example, commonslang had a branch coverage of 98% and a mutation score of 75%.

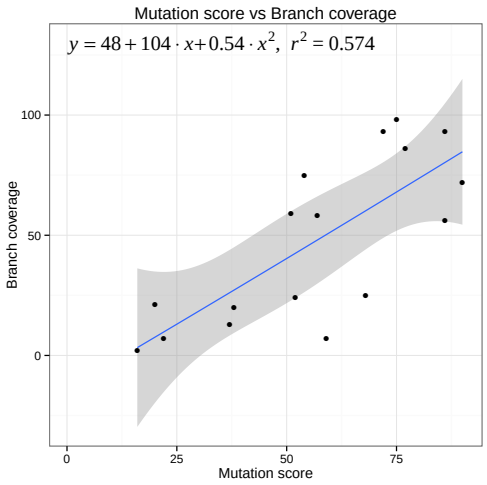

Figure 10: Branch coverage compared to the mutation scores of the test suites

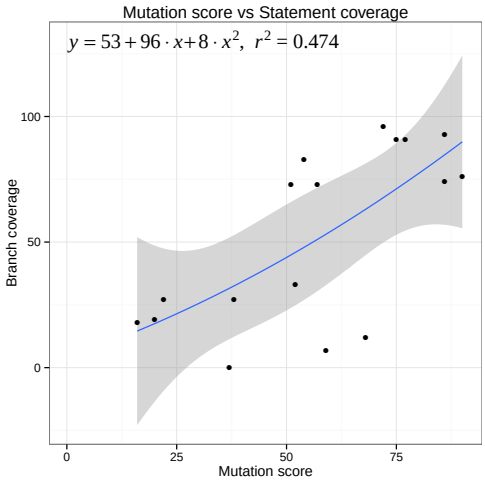

Figure 11: Statement coverage compared to the mutation scores of the test suites

Jdom had a branch coverage of 86% and a mutation score of 77%, slightly greater than commonslang's. The current positive correlation achieved may be due to the higher coverage scores of larger size test suites. It will be safe to assume coverage scores as effectiveness indicators after exploring the same correlation with test suite's size being held constant.

Our results suggest that, for many Java programs, varied in size, there is a moderate correlation between the effectiveness and the coverage of a test suite when the influence of test suite size is ignored.

A comparison between coverage and complexity was done to see if it was the inherent nature of the testers to target coverage when the complexity is high. Figure6 illustrates the relationship between the coverage scores and the cyclomatic complexity of our case study applications. The horizontal axis represents complexity while the vertical axis holds the coverage values. The Pearson's correlation coefficient determined for this comparison was 0.01 for branch coverage and 0.02 for the statement coverage. The Kendall tau value obtained was -0.04. Considering the Pearson's, there is a very low correlation between the coverage and the complexity of the applications. This can be due to the range of complexities in the applications considered. The complexities were not wide spread, but ranged between 1.29 and 3.78. Analyzing this at the method level in the future can help us understand if testers have concentrated on covering code that were highly complex.

A very low correlation was obtained when we compared the branch and statement scores and the complexity of our applications. A negative correlation was obtained from Kendall analysis.

5. Threats to Validity

We have several threats to validity in our work. First, we have no contact with the testers of the applications used in the study. Thus we are not aware of any domain specific goals with which the test suites might be written. We have characterized the test suites by studying applications of varied size and domain to find the relationship between the

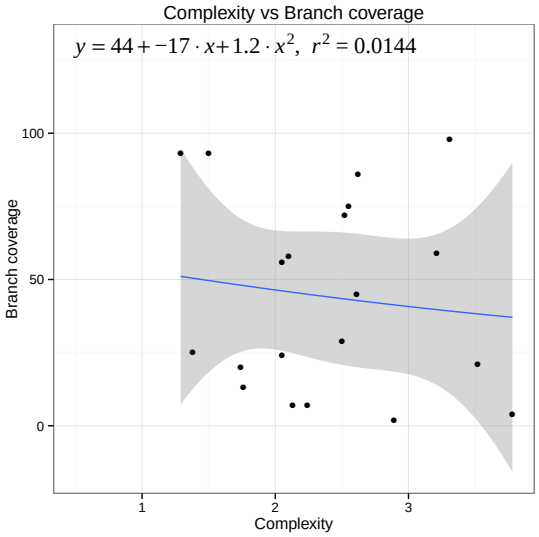

Figure 12: Branch coverage score compared to the complexity

source code and test suite attributes. The results can be different for the test suites developed for the closed source test suites where the tester's motivation can be different.

There may be equivalent mutants that may not actually introduce faults, but rather create an equivalent program given the feasible paths in the program. In this work, we assume that mutants may or may not be equivalent to the original program, given the tools we use and complexity involved in identifying the equivalent mutants. Thus, a mutation score of 100% may not be possible for all programs.

The determination of the quality of software tests can be considered a subjective measurement. Although mutation score and coverage are two ways to measure test suite quality, both the metrics have been supported as well as dispproved as quality indicators by different studies [7] [18] [21] [24] [29]. However, we have considered mutation scores as our effectiveness indicator due to the nature of our study.

The tools used for coverage and mutation analysis also lead to a potential internal threat to validity. JaCoCo was used for all coverage analysis, and Major was used for mutation analysis. JaCoCo, a well-established coverage monitor for Java programs, is based on Emma, another standard tool for analyzing Java programs. Major is an experimentally verified and well tested tool. However, other tools can be used in future work to check the uniformity of the results obtained.

In case of mutation analysis, we have modified the source code in two of our applications which faced a memory issue while runnig Major. There is a 64k byte-code size limit on Java methods and we got "code too large" error due to which we split the methods that faced this issue. The functionality of the method was not affected in any way.

6. Related Work

In this section, we will first discuss two closely related papers that have approached program and test suite metrics with similar research questions like ours. Following that, we discuss that have studied coverage scores and mutation scores of a test suite. Finally we discuss related works that have analyzed test case writing attributes in general like annotations in Java.

Characterization style studies : Our study's goals were partially influenced by a characterization type of work by Inozemtseva&Holmes (2014) [21]. Their study generated about 31,000 test suites for five systems of very different domains studied the correlation between the coverage and the effectiveness of test suites to be low or moderate when suite size is controlled. The study concludes that is not safe to consider coverage as a fault finding indicator because the mutant effectiveness was not strongly related to coverage. The study mainly explored the relationship of code coverage metric and the efficiency of the test suite when its size was ignored or held constant. Our work will neither agree nor disagree the coverage as a faultness indicator. Instead, we set out to find if predominantly practiced quality indicators like coverage have influenced testers while writing test cases.

Another study by Namin&Andrews (2009) analyzed research questions that are similar to our work [29]. Their study focussed on the relationship between test suite size, coverage and effectiveness.The study analysis primarily analyzed a relatively small set of C programs with a maximum NLOC of 513. The study did not find any linear relationship between the test suite size, coverage and effectiveness. However size and coverage can independently influence the effectiveness of the suite. Though our study accounts all the metrics used in this study, we do not intend to study if bigger test suites increase its effectiveness. One of our aim is to find if industrial test suites grow large with code size. We will also use case study applications that are comparatively huge than this study from a variety of repositories.The work [The influence of size and coverage on test suite effectiveness] intended to study the relationship between test suite size, coverage and effectiveness. Whenever a test case gets added to the test suite, it makes the test suite bigger and it can increase the coverage too. The study sets out to find if the increased coverage on the bigger test suites or if size of suite directly increases the effectiveness of the suite. The analysis is primarily with relatively small C programs with a maximum NLOC of 513. The study did not find any linear relationship between the test suite size, coverage and effectiveness. However size and coverage can independently influence the effectiveness of the suite. Though our study accounts all the metrics used in this study, we do not intend to study if bigger test suites increase its effectiveness. One of our aim is to find if industrial test suites grow large with code size. We will also use case study applications that are comparatively huge than this study from a variety of repositories.

Debates on coverage and mutation score as quality indicators : Measuring test suite quality is a major research area in software testing and it is a required study for automated testing to produce high quality suites. Most of the other related works to our characterization study investigated the link between attributes like coverage or mutation score and test suite effectiveness . First we will survey the works that have analyzed coverage score in various dimensions to explore it as a yardstick for test suite efficiency.

Empirical studies conducted on a call center reporting system with 40+ million LOC revealed that there is a direct effect of code coverage and the post-release defects. Code with higher coverage tends to have fewer field defects making it a practical technique to measure test suite quality [27]. Industrial validation of test coverage quality was a straightforward study that compared coverage driven test suites with random test suites. Each coverage metric has varied criteria and therefore has specific requirements that a test suite should meet. When a test suite meets the requirements of coverage metric, testers assume it is able to find more faults than a random test suite that does not meet the requirement . Based on this assumption, coverage metrics are used to quantify the quality of a test suite. Coverage based test suites have detected more faults than random test suites. Studies have proved that attempting to achieve a coverage level close to 100 percent can benefit in additional fault detection [8]. Another study by Rothermel et provided

empirical evidences for coverage as an efficiency metric in an indirect way. The study found that test suites reduced in size without compensating coverage were more effective than those that were reduced to the same size by randomly eliminating test cases [33]. While all these studies have investigated the positive relationship between the coverage score and the effectiveness, none of these studies have analyzed the coverage scores in a bottom-up way from the testers point of view. For example, one of our research question is to find if there is high coverage for more complex codes.We hope to find if coverage score has been a possible motivation for the testers and if so, how have they affected other characteristics like the execution time of the test suite.

As coverage is considered as a technique to measure quality and also taking inputs from previous works, researchers have also studied on the type of coverage that can best predict the quality of the test suite. A study by Gopinath et al. reports the correlation between lightweight coverage criteria (statement, block, branch, and path coverage) and mutation kills for hundreds of Java programs, for the manually as well as automatically generated test suites [17]. For both original and generated suites, the study revealed that statement coverage is the best indicator for mutation kills and also a predictor of the test suite quality. A study's results by Gligoric et al. suggested that branch coverage can be the best criteria to be considered while comparing test suites for their quality [15]. However non-branch coverage criteria can be simpler to measure and implement as all the criteria studied predicted mutation kills in a decent way. While we measure statement, branch and modified decision coverage for our test suites, our study focuses on analyzing if the coverage was a decisive factor to the testers in the test suite development rather than exploring the efficiency of different types of coverage.

Contrasting to the studies discussed above which classified coverage score as an efficiency metric, a study criticized the use of coverage metric by the testers. Marick, author of 4 coverage tools articulates that testers often give into the urge of satisfying the coverage required rather than actually analyzing what kind of testing design flaws they point to. While low coverage numbers are definitely not a good sign for the project, coverage numbers should only be used as a measure to enhance the effort of the tester [26]. This study is a eye opener on how code coverage can be misused in the industry. Our study will provide a reverse engineered view of code coverage. We hope to discover how testers have weighed coverage while writing test cases. One of our research question is to find if complex code had high coverage score.With the increased fame of code coverage in project assessments, reviews and informal discussions, a survey of 17 coverage tools was studied and reported on various levels like language-support, platform-support provided by the tool, the tools reporting format, the overhead it can provide etc [37]. Each tool has its pros and cons depending on its application domain. Our study will take advantage of the tools that provide coverage measurements (statement, branch and modified decision coverage) for the test suites studied.

The second widely studied metric to gauge test suite effectiveness is Mutation score. In this section we will discuss previous works that have debated mutation score as an appropriate tool to produce faults in testing and also the link between mutation score and the test suites effectiveness.First we will see how researches have proved mutation score as a valid substitute for real faults in software testing. The straightforward measure of test suite effectiveness (ability to detect fault faults) is fault detection. Conversely testers may not have time to find projects big enough with real faults in it to detect the test suites quality. Researchers and testers have manually or have automatically inserted faults to the program to produce a faulty version of the application for experiment sake. Generated mutants provide the researchers the opportunity to repeat the experiments and also the

mutant volume can help enhancing the statistical proof of thel results [8]. A study proved that automatically generated mutants using standard mutation operators yields results that were similar to the real faults whereas hand seeded faults were harder to detect than the real faults [7]. The study also revealed that, to use mutant score as a trustable replacement for the real faults, mutants must be generated using the commonly uses mutant operators. Another study that was conducted on five large open source projects with 375 real faults revealed that coupling effect exists between the real faults and the mutants. Though the number of faults coupled to a single fault were small when coverage was controlled. There were faults that were not coupled to the real faults which needed modifications to become equivalent to a real fault [24]. Fault-detecting ability is also considered one of the most direct measures of the effectiveness of test adequacy criteria. In our study, we use mutation score as an efficiency metric. We plan to analyze how factors like coverage might have motivated testers for an effective test suite development.

Researchers have studied the role of mutation in test suites evaluation. The ability of test suite to kill the generated mutants has been used to gauge the effectiveness of the test suite. A study the correlation between the coverage and the mutation score is studied with an open-source application (spojo) [9]. The study showed that there is a strong positive correlation between coverage and mutation score. However the studys case study application ranged in size from 89 to 261 non-commented lines of codes. Our study will analyze 24 open source applications which are in practice and with LOC greater than that considered for the this study. In another study [8], mutants validity as a predictor of the detection effectiveness of real faults was used as the primer to investigate questions regarding the relationships between fault detection, test suite size, and control/data flow coverage. The paper also proposes an initial strategy for organizations to determine the coverage level that will be required to attain acceptable detection rates. Majority of the previous experimental results have not directly covered reports on determining the correlation between coverage score and mutation scores. However, results have proven the point that, fault detection of a software system is related with high coverage.In our study we will analyze the direct correlation between the coverage and mutation score.

Overall much research in software code coverage and mutation score has been realized and also used in the industry. Through our research questions, we aim to analyze the current test suites in practice to see what has motivated testers in reality. And also how coverage and mutation score as decisive metrics have influenced other metrics.

Test Case Writing Attributes: Work has also been completed in analyzing the attributes and style in which test cases are written. This work includes how annotations are used to identify and describe test cases and how mocks and stubs are used to create independent test cases.

One study, conducted by Rocha&Valente (2011) [32], studied 106 open source Java programs to investigate how annotations are used by the developers in the writing of test cases. They examined the most used annotations, the most annotated elements, and the "annotation hell" phenomenon. Annotation hell is a phenomenon where the source code becomes overloaded with annotations. The study revealed that atleast 10 of their case study applications had very high annotation density and thus suffered from annotation hell. Compiler generated annotations were the most popular annotations observed, while `@Test` annotations were second most common. `@Test` annotations were used by the programmers to mark methods that will be dynamically tested through reflection.

Another study in understanding the content of test cases focused on the use of mock objects and methods in industry [28]. The study directed a large empirical study that

collected 6,000 Java software projects from Github and analyzed the source code of the projects to answer a number of questions about the popularity of mocking frameworks, the usage of mocking frameworks, and the mocked classes. The study discovered that mocking frameworks are used in a large portion, about 23%, of software projects that have test code, and they learned that testers use mocking primarily for simulating source code classes rather than simulating the library classes.

Automatically created test suites versus manually generated suites are also a concern. Development enviornments also can make a difference. Kracht et al. [25] and Alkaoud and Walcott [6] discuss the differences observed in test suites in these scenarios.

While the style of writing test cases is important, in this research, we examine potential factors that may have driven the testers to focus on particular areas of code, leading to the success and quality of the test suite given analytical methods. This allows us to determine where open-source test writers concentrate their testing efforts and, potentially, why.

7. Conclusion

In this paper, we measured and analyzed different attributes of open source applications and test suites. With the correlations achieved between the attributes, we have characterized traditional test suite development fashion in the open source world. We have also examined possible aspects that can influence the testers while developing the test suite.

For this we have measured lines of code and complexity for all our programs and lines of code, complexity, execution time, mutation score and coverage score of the test suites. Our research demonstrated that in applications that have greater than 500 number of tests, the test suite size can grow when the coverage is higher. However, the test suite size was not much influenced by the complexity of the program. We found only a slight correlation of 0.116. Acceptably, the test suite execution time was not influenced much by the complexity of the program studied. We do note that the test suite execution time can increase slightly when the coverage is increased in bigger test suites.

When we researched about the potential focus of the testers while developing the test suites, we found that coverage can influence the effectiveness (mutation score) and vice versa. This relationship has been much debated and there are results that supports as well as contradicting this. However for the set of applications we considered, there is a positive correlation indicating coverage can be indicator for the effectiveness, when we ignore the size of the test suite. However, test suite developed for the applications did not achieve coverage corresponding to the complexity of the program. The number of tests for an application can increase the effectiveness of the test suite, when the size of the application is taken into consideration.

Our next step is to perform analysis on the programs and the corresponding test suites at the method level. Analyzing some of the relationships considered in this study at the method level can provide results in a different perspective. For example, we anticipated the test suite size coverage to be higher when the complexity is high. However, only a low correlation coefficient was observed when we analyzed the coverage scores and the complexity of the code. Also, exploring all the possible relationships at the method level granularity can reveal the motivation behind the test suite developed (e.g, unit testing, integration testing etc). We will also need more empirical evidences for the results to be presented as a predictable trend in test suite development.

8. References

[1] https://github.com/. GitHub is a web-based repository hosting service, which offers all of the distributed revision control and source code management (SCM) functionality.

[2] http://sourceforge.net/.

[3] http://sir.unl.edu/portal/index.php/.

[4] http://www.evosuite.org/subjects/sf100/.

[5] M. Adolfsen. Industrial validation of test coverage quality. 2011.

[6] H. Alkaoud and K. R. Walcott. Quality metrics of test suites in test-driven designed applications. In *International Journal of Software Engineering Applications (IJSEA), 2018*, 2018.

[7] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Software Engineering, 2005. International Conference on Software Engineering 2005. Proceedings. 27th International Conference on*, pages 402–411. IEEE, 2005.

[8] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin. Using mutation analysis for assessing and comparing testing coverage criteria. *Software Engineering, IEEE Transactions on*, 32(8):608–624, 2006.

[9] B. Assylbekov, E. Gaspar, N. Uddin, and P. Egan. Investigating the correlation between mutation score and coverage score. In *Computer Modelling and Simulation (UKSim), 2013 UKSim 15th International Conference on*, pages 347–352. IEEE, 2013.

[10] B. Beizer. *Software system testing and quality assurance*. Van Nostrand Reinhold Co., 1984.

[11] A. Carzaniga, A. Fuggetta, R. S. Hall, D. Heimbigner, A. Van Der Hoek, and A. L. Wolf. A characterization framework for software deployment technologies. Technical report, DTIC Document, 1998.

[12] S. Cornett. Minimum acceptable code coverage, 2006.

[13] C. L. et al. Javancss - a source measurement suite for java, May 2009.

[14] R. L. Glass, R. Collard, A. Bertolino, J. Bach, and C. Kaner. Software testing and industry needs. *IEEE Software*, 23(4):55–57, 2006.

[15] M. Gligoric, A. Groce, C. Zhang, R. Sharma, M. A. Alipour, and D. Marinov. Comparing non-adequate test suites using coverage criteria. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pages 302–313. ACM, 2013.

[16] C. K. M. GmbH and Contributors. http://www.eclemma.org/jacoco/.

[17] R. Gopinath, C. Jensen, and A. Groce. Code coverage for suite evaluation by developers. In *International Conference on Software Engineering*, pages 72–82, 2014.

[18] A. Gupta and P. Jalote. An approach for experimentally evaluating effectiveness and efficiency of coverage criteria for software testing. *International Journal on Software Tools for Technology Transfer*, 10(2):145–160, 2008.

[19] B. Hailpern and P. Santhanam. Software debugging, testing, and verification. *IBM Systems Journal*, 41(1):4–12, 2002.

[20] W. S. Humphrey. Characterizing the software process: a maturity framework. *Software, IEEE*, 5(2):73–79, 1988.

[21] L. Inozemtseva and R. Holmes. Coverage is not strongly correlated with test suite effectiveness. In *International Conference on Software Engineering*, pages 435–445, 2014.

[22] N. Juristo, A. M. Moreno, and W. Strigel. Software testing practices in industry. *IEEE software*, 23(4):19–21, 2006.

[23] R. Just. http://mutation-testing.org/, Oct 2014.

[24] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser. Are mutants a valid substitute for real faults in software testing. Technical report, Technical Report UW-CSE-14-02-02, University of Washington, 2014.

[25] J. S. Kracht, J. Z. Petrovic, and K. R. Walcott-Justice. Empirically evaluating the quality of automatically generated and manually written test suites. In *Quality Software (QSIC), 2014 14th International Conference on*, pages 256–265. IEEE, 2014.

[26] B. Marick. How to misuse code coverage. In *Proceedings of the 16th Interational Conference on Testing Computer Software*, pages 16–18, 1999.

[27] A. Mockus, N. Nagappan, and T. T. Dinh-Trong. Test coverage and post-verification defects: A multiple case study. In *Empirical Software Engineering and Measurement, 2009. ESEM 2009. 3rd International Symposium on*, pages 291–301. IEEE, 2009.

[28] S. Mostafa and X. Wang. An empirical study on the usage of mocking frameworks in software testing. In *Quality Software (QSIC), 2014 14th International Conference on*, pages 127–132. IEEE, 2014.

[29] A. S. Namin and J. H. Andrews. The influence of size and coverage on test suite effectiveness. In *Proceedings of the 18th International Symposium on Software Testing and Analysis*, pages 57–68. ACM, 2009.

[30] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, 1992.

[31] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2014.

[32] H. Rocha and M. T. Valente. How annotations are used in java: An empirical study. In *SEKE*, pages 426–431, 2011.

[33] G. Rothermel, M. J. Harrold, J. Von Ronne, and C. Hong. Empirical studies of test-suite reduction. *Software Testing, Verification and Reliability*, 12(4):219–249, 2002.

[34] S. Vegas and V. Basili. A characterisation schema for software testing techniques. *Empirical Software Engineering*, 10(4):437–466, 2005.

[35] J. A. Whittaker. What is software testing? and why is it so hard? *Software, IEEE*, 17(1):70–79, 2000.

[36] Wikipedia. Cyclomatic complexity, Mar. 2015.

[37] Q. Yang, J. J. Li, and D. M. Weiss. A survey of coverage-based testing tools. *The Computer Journal*, 52(5):589–597, 2009.

[38] E. S. Yu and J. Mylopoulos. Understanding why in software process modelling, analysis, and design. In *Proceedings of the 16th international conference on Software engineering*, pages 159–168. IEEE Computer Society Press, 1994.

[39] X. Zhang, C. Nie, B. Xu, and B. Qu. Test case prioritization based on varying testing requirement priorities and test case costs. In *Quality Software, 2007. QSIC'07. Seventh International Conference on*, pages 15–24. IEEE, 2007.