EVALUATION OF SOFTWARE TESTING COVERAGE TOOLS: AN EMPIRICAL STUDY

A Paper
Submitted to the Graduate Faculty
of the
North Dakota State University
of Agriculture and Applied Science

By

Khalid Ali Alemerien

In Partial Fulfillment
for the Degree of
MASTER OF SCIENCE

Major Program:
Software Engineering

May 2013

Fargo, North Dakota

North Dakota State University
Graduate School

Title

EVALUATION OF SOFTWARE TESTING COVERAGE TOOLS: AN EMPIRICAL STUDY

By

Khalid Alemerien

The Supervisory Committee certifies that this ***disquisition*** complies with North Dakota State University's regulations and meets the accepted standards for the degree of

MASTER OF SCIENCE

SUPERVISORY COMMITTEE:

Kenneth Magel
Chair

Hyunsook Do

James Coykendall

Approved:

5/2/2013
Date

Brian Slator
Department Chair

ABSTRACT

Code coverage is one of the most important aspects of software testing, which helps software engineers to understand which portion of code has been executed using test suite throughout the software testing process. Automatic testing tools are widely used to provide testing coverage metrics in order to gauge the quality of software, but these tools may suffer from some shortcomings such as the difference among the values of code coverage metric of a given program using different code coverage tools. Therefore, we designed and performed a controlled experiment to investigate whether these tools have a significant difference among the measured values of coverage metric or not. We collected the coverage data that consist of branch, line, statement, and method coverage metrics. Statistically, our findings show that there is a significant difference of code coverage results among the code coverage tools in terms of branch and method coverage metrics.

ACKNOWLEDGMENTS

I would like to express my gratitude to my advisor, Dr. Kenneth Magel for his time, patience, support and guidance.

I would like to express my appreciation to my advisory committee: Dr. Kenneth Magel, Dr. Hyunsook Do, and Dr. Jim Coykendall for being a part of my master’s paper committee.

The most special thanks go to my best partner and friend, my wife. Saba, you gave me your unconditional support and love through all this long process. I take this opportunity to express the profound gratitude and love to my beloved daughter "Lojain".

An honorable mention goes to our families and friends for their understandings and supports on me in completing this research.

Last but not the least, the one above all of us, Allah, for answering my prayers for giving me the strength to accomplish this research.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1. INTRODUCTION

Software testing, which is used to indicate the quality of software, is a process to detect defects and mitigate their associated effects. In fact, software testing plays a significant role in the software development process. Indeed, most of the costs and resources of developing and maintaining software are related to the software testing process [38]. One of the important aims of the software testing process is to report a high testing coverage percentage of a given program. This code coverage represents a criterion that is used to measure the completion of the testing process.

Code testing coverage shows which portion of code, for a given program, is touched by at least one test. Moreover, code testing coverage is considered by developers as an indicator of confidence level in their software. In order to facilitate the analysis of code coverage, there is a need to automate this process. Therefore, there are many code testing coverage tools, which attempt to help researchers, practitioners, and end-users to understand the software testing process. Therefore, many researchers have studied differing aspects of testing coverage analysis process. However, some of them have focused on studying the code coverage tools. To our knowledge, these researchers have investigated the testing coverage tools from a theoretical point of view only [1] [2] [3] [4] [5] [6]. Although some empirical studies have focused on specific features of code coverage tools, nevertheless other features have not been investigated [27] [28] [29] [30] [31] [32] [33].

In general, the topic of inconsistency of coverage metrics that have been calculated by code coverage tools has not received any attention in the research literature. To our knowledge, no research that considers the differences among the values of specific coverage metrics that are measured by different code coverage tools on a given program.

To address this problem, we conducted an empirical study to investigate the possible significance of value differences as calculated by different code coverage tools of a given coverage metric. In addition, we explain the effects of the possible variance among these tools. Moreover, we attempt to explain why this variance may occur. In addition, the testing process is a very expensive and this leads us to ask the following question from the point of view of developers and managers: “What is the code coverage tool that gives the most reliable coverage information?” Therefore, we designed and ran a controlled experiment using several open source java programs as well as several code coverage tools. Also, we studied four common coverage metrics; branch coverage, statement coverage, line coverage, and method coverage. To do this, we followed these steps:

1. Identifying the coverage metrics that should be collected during the experiments.
2. Choosing an appropriate set of code coverage tools that achieve the following conditions: First, these tools can be integrated with JUnit. Second, these tools can be integrated with Java Eclipse IDE. Third, these tools support some or all the coverage metrics that we want to collect. Finally, these tools have stable version to run.
3. Selecting a set of java programs that are different in size, programming domain, and number of test cases.
4. Collecting and analyzing the collected coverage data as well as presenting the results.

So, our motivation was to understand the available code coverage tools through conducting a controlled experiment. Our findings show a significant difference among code coverage tools for a given coverage metric in some cases such as large programs.

The rest of this paper is organized as follows: Chapter 2 provides the background information and related work to empirical studies that evaluated code coverage tools. Chapter 3 presents the software testing coverage including overview of software testing coverage and coverage metrics, code coverage analysis process, and illustration of selected code coverage tools. Chapter 4 describes how to perform our experiment. Chapter 5 shows the results and analysis. Chapter 6 discusses our findings and chapter 7 presents the conclusions and future work.

CHAPTER 2. BACKGROUND AND RELATED WORKS

Code testing coverage tools assist the developers to understand the testing process through test coverage reports. These reports consist of different aspects of code coverage such as code coverage metrics, visualization support of coverage granularities, common statistics about a given program, and so forth. On the one hand, these tools provide coverage information that may help developers in the process of code analysis, but on the other hand, they may make the process of code analysis complicated especially for large-scale systems.

Thus, to investigate the effectiveness of code coverage tools, many researchers have conducted numerous of empirical studies including comparisons among code coverage tools, examining the metrics of evaluating code coverage tools, relationship between code coverage tools and reliability, and impact of visualization on the effectiveness of code coverage tools. Therefore, in this chapter, we present the related work to these four areas as the following:

2.1. Comparison among Code Coverage Tools

To date, several empirical studies were conducted that compared among code coverage tools in order to investigate the features of these tools. Youngblut and Brykczynski [1] [2] surveyed theoretically the code coverage tools as a part of comprehensive study of software testing tools. In these surveys, they showed a comparison among set of coverage tools. This comparison consists of coverage metrics, reporting format, the required instrumentation and drivers, and some other features.

However, Yang et al. [4] compared, theoretically, 17 code coverage tools focusing on the following features coverage metrics, prioritization for testing, automatic generation of test cases, and ability to customize the test coverage. They focused on these features to understand the available code coverage tools, and then to compare them to eXVantage, a tool that provides code coverage metrics, reporting, and performance profiling. For each tool, they presented which programming languages support, instrumentation, levels of coverage, and reporting formats. Moreover, they provided guidelines for researchers and practitioners to select the appropriate code coverage tool. However, they did not conduct an actual experiment to examine the effectiveness of these tools.

Moreover, Shahid and Ibrahim [5] surveyed 19 code coverage tools. They compared, theoretically, five features: programming language support, instrumentation (source code, code byte), coverage metrics (statement, branch, method, and class), and GUI support and reporting format. This information was collected from literature and the websites of tools but they did not conduct an experiment to exercise the variance in coverage metric values of these tools.

In fact, some researches performed experiments that focused on the large software systems. On the one hand, the code coverage tools provide developers a huge coverage data to identify the tested areas, but on the other hand, the analysis process of this huge data is a time-consuming task. Therefore, Asaf et al. [36] proposed an approach to define numerous of views onto coverage data to improve the coverage analysis.

Furthermore, Kessis et al. [3] presented test and coverage analysis of J2EE servers. Basically, they aimed to provide a real case study that consists of test and coverage analysis of JOnAS server. To run this experiment, they used JOnAS middleware (200.000 LOC) and more than 2500 test cases as well as using clover analyzer. They had presented an empirical evidence of applicability of the coverage analysis with large Java application.

In addition to that, Kim [6] investigated, empirically, the efficient way to perform code coverage analysis on large software projects. Therefore, he examined coarse coverage analysis versus detailed coverage analysis, block coverage versus decision coverage, and cyclomatic complexity versus defect and module size. This study used a large software system with 19,800K LOC. According to his findings, he proposed a systematic approach of coverage analysis for large software systems.

Finally, Elbaum et al. [39] examined, empirically, the impact of software evolution on coverage information. They used statement coverage and function coverage metrics in their experiment. In addition, they found that the changes during evolution of software impact the quality of coverage information. However, they did not study the variance in coverage criteria using code coverage tools.

2.2. Metrics for Evaluating Code Coverage Tools

To evaluate the code coverage tools in quantitative and qualitative way, some studies have presented set of metrics for evaluating code coverage tools. Moreover, these metrics have allowed researchers and practitioners understand the features of code coverage tools. In addition, these metrics may help them to choose an appropriate tool among set of tools.

Therefore, Michael et al. [7] proposed a suite of metrics for evaluating tool features, which assist the researchers and practitioners to choose an appropriate code coverage tool. This suite of metrics consists of 13 metrics such as Human Integrate Design (HID) and Maturity and Customer Base (MCB). So, the proposed metrics have been used to evaluate the features of code coverage tools without considering the variance in coverage metric values.

Moreover, Priya et al. [8] conducted an experiment to examine the suite of metrics that was proposed in [7] to support testing procedural software. In this experiment, the researchers considered 9 small programs and 4 code coverage tools to calculate the proposed metrics but they did not focus on the variance in values of coverage metric.

Furthermore, Kajo-Mece and Tartari [9] conducted an experiment that examined 2 code coverage tools Emma and Clover using very simple java programs for search and sort algorithms. And also, they calculated 4 metrics that proposed in [7] to judge which code coverage tool can be used efficiently by testing team. These metrics are: Reporting Features (RF), Ease of Use (EU), Response Time (RT), and Human-and Interface Design (HID). However, they have not studied the difference among code coverage metrics that are calculated by code coverage tools and its causes.

2.3. Relationship between Code Coverage Tools and Reliability

Since the code coverage tools are important in the software testing process and the coverage granularities are useful to judge whether a given program is reliable or not, numerous of researcher conducted experiments to exercise the relationship between testing coverage and reliability [27] [28] [29] [30]. As result, these experiments showed a high correlation between testing coverage and reliability, this means the researchers found that the reliability of programs increases if the code coverage increases.

Moreover, Gaffney et al. [10] used JUnit and Clover to provide a very simple java program as an example to support their assumption about the code coverage tools. They assumed that the information that is provided by code coverage tools is valuable and useful but not sufficient to determine whether that a piece of code works correctly or not. Moreover, this assumption was supported in [14] as well. This assumption shows that the code fully covered does not ensure the absence of defects.

2.4. Impact of Visualization on Effectiveness of Code Coverage Tools

To simplify the coverage analysis process, most of code coverage tools provide the coverage granularities that are supported by different visualization techniques. Therefore, a few empirical studies [31] [32] [33] have investigated the impact of code coverage visualization on the effectiveness of test cases. They found that the coverage visualization is effective in testing coverage analysis. In addition, some research work [34][35] proposed new visualization techniques that support the testing coverage analysis. Furthermore, this research work evaluated empirically the effectiveness of the visualization techniques on the testing coverage analysis process.

Some researchers [14][37] studied the shortcomings of code coverage tools. Hence, Cornett [14] studied the strengths and weakness of code coverage metrics for C, C++, and Java. Basically, he found that the condition/decision coverage is the best coverage measure for these programming languages. Moreover, the combination between weak and stronger coverage metrics is effective to decide which test cases should be deferred. In addition, in testing practice, there are different types of software testing: unit testing, integration testing, and functional testing. Typically, test suites contain related test cases to these testing types for given software project. However, the test coverage measures do not distinct among those testing types. Therefore, Kanstren [37] proposed a quantitative method to measure the test coverage for each type of testing.

So, to our knowledge, all the above research studies have not investigated how and why the values of coverage metrics that are measured by different code coverage tools are different for a given program? In addition, what is the impact of this difference on decision-making especially in software testing phase? Finally, what is the code coverage tool that provides the most accurate code coverage results? Therefore, in this paper, we performed experiments using 21 java programs and 5 code coverage tools in order to answer the above questions.

CHAPTER 3. SOFTWARE TESTING COVERAGE

3.1. Software Testing Coverage

Code coverage is a quality metric that calculates how thoroughly the test cases exercise a given program [12]. Thus, code coverage provides valuable information as the following: code coverage provides software developers which piece of code is tested as well as which is not. In other words, which portion of code is poorly tested? In addition, code coverage provides a quantitative measure, which is used as an indicator of reliability of software product. Furthermore, code coverage helps to quantify the progress of testing phases. This leads to enhance the test suite without affecting the defect detection process. Moreover, code coverage plays a significant role to discover the dead piece of code [12] as well as code coverage might be used to assess the progress of quality assurance process, and at the same time, plays a guidance of developers. To end this, code coverage is effective to assist in test cases prioritization and generation, which reduces the effort and cost, increases the number of effective test cases as well [11]. For example, in test cases prioritization, code coverage may help to determine which tests we need to remove from the test suite, because the redundant tests consume the resources and time. However, there are quite drawbacks such as code coverage may not be able to determine or predict how many defects likely to be found when the code coverage increases. Unfortunately, the code fully covered does not ensure the absence of defects but it is used to assure the quality of test cases [14].

To make code coverage process valuable in the software development process, the developers of code coverage tools provide several coverage metrics: Line coverage, statement coverage, branch coverage, method coverage, class coverage, path coverage, loop coverage, and requirement coverage.

- *Line coverage*

Line coverage is a simple metric that measures the number of lines of code that have been touched at least one time by test case.

- *Statement coverage*

Statement coverage is a metric that measures whether code statement has been examined by test case at least one time or not.

- *Branch coverage*

Branch coverage is a metric that measures which decision outcomes, true or false, that have been executed at least by one test case.

- *Method coverage*

Method coverage is a metric that measures whether the method has been called or not. Method coverage does not explain which piece of code is executed inside of method or how the method is called. Moreover, method coverage does not mention how many times call for that method.

- *Class coverage*

Class coverage is a metric that measures whether the class has been called as a part of an execution of test suite.

- *Path coverage*

Basically, a path is the flow of code execution from the start point to the end point in method level. Therefore, path coverage is a metric that measures whether part or all statements in a path have been executed at least one time by one test case. Furthermore, to calculate path coverage we can use a cyclomatic complexity metric to specify the basis paths in order to minimizing the number of test cases.

- *Loop coverage*

Loop coverage is a metric that calculates the percentage of loops in a program that have been examined by a test suite

- *Requirement coverage*

Requirement coverage is a metric that measures the percentage of requirements that have been exercised by a test suite.

In summary, Most of code coverage tools assist in evaluating the effectiveness of testing process by providing a set of code coverage metrics [13].

To investigate our research questions, we used line coverage, statement coverage, method coverage, and branch coverage. In addition, code coverage tools, are we used in our experiment, provide these coverage metrics. Therefore, in next subsections, we present an overview of code coverage analysis process as well as we illustrate the features of each code coverage tool that has been used in our experiment.

3.2. Code Coverage Analysis Process

In general, code coverage analysis process consists of three fundamental steps: code instrumentation, gathering of code coverage data, and analysis of coverage data.

1. *Code instrumentation*

There are two different types of instrumentation: First, instrumentation of source code, this is a preprocess that allows the code coverage tool to insert automatically additional statements to the source files before compilation in order to gather the coverage data during the testing process. Second, instrumentation of byte code is a technique to insert additional statement to the class files during the load time, which is called On-The-Fly. However, some of code coverage tools are built based on profiler monitoring, this means doing the code coverage process without using any instrumentation techniques.

2. *Gathering of code coverage data*

This step includes collecting and storing the code coverage data from the runtime of program testing.

3. *Analysis of coverage data*

It consists of analyzing the collected coverage data then views the coverage metrics to provide the developers a feedback about their code. This feedback helps to improve, mitigate, or modify the test suite.

3.3. Software Testing Tools

In our research, we have selected 5 code testing coverage tools to conduct our experiment. Briefly, we chose these tools for the following main reasons: These tools are available as eclipse plugin as well as these tools integrate with JUnit testing framework. Moreover, these tools are available to the public use and these tools provide the coverage granularities in multiple report formats. Finally, these tools are widely used in both industry and research fields. The following subsections show these tools and associated data.

3.3.1. Ecobertura

Ecobertura [15] is a free eclipse plugin that calculates the code coverage through execution of test cases. Ecobertura is used to calculate the percentage of line and branch coverage metrics for each package, each class, and for overall of program. Ecobertura shows the source code in Cobertura coverage mode. So, the part of source code that is accessed by test cases is colored in green while the untested part is highlighted in red color. Ecobertura shows the results of testing coverage process after executing the source code through JUnit framework. To enable the Ecobertura tool on the source code, we select "*Cover As*" command. And then, we can show the code coverage metrics using "*coverage code session*" command.

3.3.2. Eclemma

Eclemma [16] is an open source tool for java code coverage. Eclemma is used to calculate the statement coverage metric. In addition, Eclemma highlights the fully covered code in green, displays yellow for the partly covered code, and shows the uncovered code in red. Moreover, Eclemma adds a coverage mode in eclipse IDE as the existing modes in eclipse like run and debug modes. To enable the Eclemma tool on the source code, we have to select "*Coverage As*" command.

3.3.3. Clover

Clover [17] is a commercial code coverage tool, which is used to calculate element, statement, line, and method coverage metrics. Moreover, Clover provides a bunch of static code metrics for the executed program such as average complexity, number of classes, and so forth. Clover also provides different visualization techniques for the code coverage such as using Treemap for a whole project or a single package in order to facilitate the understanding and analysis of code coverage information. To enable the Clover tool on the source code, we select "*Clover*" option then "*enable on this project*" command.

3.3.4. Djunit

Djunit [18] is an open source tool for java code coverage. Djunit is an eclipse plugin, which performs the JUnit tests to calculate the code coverage for line, branch, package, file, and overall program. This tool generates and shows directly the code coverage granularities of the JUnit tests that are performed in eclipse IDE. To run the Djunit tool on the source code, we basically select "*run as Djunit*" command from run menu.

3.3.5. Code Pro Analytix

Code Pro Analytix [19] is a free tool for java code coverage. It was developed by google developers for eclipse IDE. Code Pro Analytix provides effective features such as static code analysis, code metrics, code dependencies, and JUnit test generation as well as code coverage. This tool calculates the code coverage at several levels of granularity as the following: class coverage, method coverage, line coverage, instruction coverage, and branch coverage. Moreover, this tool shows the historical changes of code coverage for different periods of time. Code Pro Analytix appears as "*Code Pro*" menu in menu bar. Moreover, to enable the Code Pro Analytix tool on the source code, we select "*Code Pro tools*" as well as choose "*instrument for code coverage*" command, and then run the tests using JUnit framework [26].

Table 3.1 shows, for each code coverage tool, Type (Commercial, Open Source, and Freeware), Coverage Level (Line, Statement, Branch, Method, and others), Instrumentation (Source Code, Byte Code, and on the fly "profile"), Reporting format (XML, HTML, PDF, and within eclipse IDE), Integrated with JUnit, and How to appear in eclipse.

Table 3.1. Code Coverage Tools and Associated Data.

Tool Name	Type	Coverage Level	Instrumentation	Reporting	Integrated with JUnit	How to appear in eclipse
Ecobertura	Open Source	Line, Branch	Byte Code	Direct in eclipse	Yes	Cover as
Eclemma	Open Source	Statement	Byte Code , on the fly	Direct in eclipse, Text, HTML, XML	Yes	Coverage as
Clover	Commer-cial	Statement, Branch, Method, element, Line, class, and contribut-ion	Source Code	Direct in eclipse, PDF, HTML, XML	Yes	Clover
Djunit	Open Source	Line, Branch	Byte Code	Direct in eclipse, HTML	Yes	Djunit Test
Code Pro Analytix	Freeware by google	Class, method, line, statement, branch	Byte Code	Direct in eclipse, Text, HTML, XML	Yes	CodePro

CHAPTER 4. EMPIRICAL STUDY

To investigate the potential differences in values of testing coverage of testing coverage tools, we performed a controlled experiment considering the following research questions:

RQ1: Does the value of code coverage metric, which is measured through code coverage tools, differs significantly from code coverage tool to another for a given program?

RQ2: How does program size affect the effectiveness of code coverage tools?

In this chapter, we present the following subsections, our objects of analysis, variables and measures, and experiment setup and procedure.

4.1. Objects of Analysis

We used 21 java programs as objects of the analysis process from various sources: SourceForge [20], Google Code [21], and Repository for Open Source Education (ROSE) [22], and Githup [23]. All the objects have been provided with JUnit test suites. We used lines of code (LOC) as a measure of program size: Small (size <= 3000 LOC), medium (3000 LOC < size <= 10000 LOC), and large (size > 10000 LOC). We used the program size to categorize the objects into the following three categories: Small programs (Multiplication, Token Occurrences, Elastic Map Reduce EMR, Trianglo, Vending Machine, CoffeeMaker, and Calculadora), medium programs (Monopoly, PureMVC, ApachiCli, Cinema, Hospital_Management, Jtopas 0.4, and Jpacman 3.0.1), and large programs (Elevator, iTrust 4.0, JfreeChart, Mondrian, CruiseControl, BlackJack, and FindBugs).

Briefly, our objects of analysis are: Multiplication is a program to find the multiplication of numbers, and Token Occurrences is a program to calculate the number of token occurrences in source code. EMR is a web service, and Trianglo is a program to specify the triangle type. Vending Machine is a command line program for vending machine, and CoffeeMaker is a command line program for vending machine. Calculadora is a calculator of main mathematical operations, and Monopoly is a game. PureMVC is Model-View-Control framework for creating applications, and ApachiCli is used for parsing commands that are passed to programs. Cinema is a program for managing cinema, and Hospital_Management is a simple program to provide a number of hospital functionalities. Jtopas 0.4 is a small parser for arbitrary text data, and Jpacman 3.0.1 is a 2D game. Elevator is a program for controlling the elevator, and iTrust is a program for sharing, obtaining, and viewing patient data. JfreeChart is a java chart library to display charts in applications, and Mondrian is an online analytical processing engine. CruiseControl is a continuous integration tool, and BlackJack is a game which supports four players and provides 2 difficulty levels. FindBugs is a static code analysis tool.

Table 4.1 shows, for each object of analysis, the number of lines of code (Size), the number of classes (Classes), and the number of test cases (Test Cases). For each object, we used JUnit to calculate the number of test cases in addition to using the SourceMonitor tool [24] to calculate the number of classes and number lines of code (LOC).

4.2. Variables and Measures

- *Independent Variable*

In order to investigate our research questions, we utilize the testing coverage tool as independent variable. So, we consider five code testing coverage tools, Eclemma, which is used to measure the statement coverage metric. Ecobertura, which is used to measure line and branch coverage metrics. And also, Clover, which is used to measure statement, branch, and method coverage metrics. Djunit, which is used to measure line and branch coverage metrics. CodePro Analytix, which is used to measure line, statement, method, branch, and instruction coverage metrics.

- *Dependent Variable and Measures*

To investigate our research questions, we consider the variance of code coverage metric as a dependent variable. Therefore, to measure the testing coverage, we used line, statement, method, and branch coverage metrics. The range of coverage values from 0% to 100%.

Table 4.1. Objects for Analysis and Associated Data.

Project	Size (LOC)	Classes	Test Cases
Multiplication	33	2	2
Triangulo	436	3	34
Vending Machine	594	6	36
CoffeeMaker	672	5	3
Token Occurrences	830	14	23
EMR	1151	17	33
Calculadora	2417	24	38
Hospital-Management	3103	47	58
Jpacman-3.0.1	3512	43	32
Cinema	4489	26	197
PureMVC	4817	54	26
Monopoly	6663	56	211
ApacheCli	7480	48	121
Jtopas 4.0	9967	40	432
BlackJack	17876	239	96
Cruise Control	25566	249	144
iTrust	65010	709	440
Elevator	164864	1239	50
FindBugs	179619	1798	192
JfreeChart	204677	639	364
Mondrian	237564	1832	1828

4.3. Experiment Setup and Procedure

In order to obtain the code coverage metrics, we need a set of java programs with test cases. Furthermore, the test cases must be generated and run by the JUnit framework. We collected 21 java programs with number of JUnit tests from famous online repositories.

In this experiment, we followed the steps shown in Figure 4.1 that illustrates the testing coverage analysis process. These steps are the following:

1. Select a code coverage tool, and then configure the settings of this tool on the source code.
2. Instrument the given java source code and JUnit tests using the instrumentation process as well as enable the selected code coverage tool on source code. This means, the code coverage tool inserts statements in the source code to record specific aspects and code coverage metrics.
3. Run the instrumented source code and instrumented test cases through JUnit framework and code coverage command for that tool, which is available in main menu of eclipse or right click menu.
4. The code coverage tool analyzes the testing results to gather code coverage data. And then, the tool views the code coverage information directly in eclipse ID. And also, this tool allows generating coverage reports. This report may contain statement, line, method, branch, and other code coverage metrics. The granularity level of coverage information is varying from tool to another. For instance, Clover tool provides a report contains branch, statement, method coverage, and coverage percentage for each class and package in addition to the project statistics. While the Ecobertura tool provides line and branch coverage for each package, class, and overall the program.
5. Remove all the instrumentations in order to be able to run next code coverage tool without considering the instrumentations that were done by previous code coverage tool.
6. Repeat the same steps for the next code coverage tool.

To sum this up, we ran each object program on five testing coverage tools. After each run, we disabled the tool on the source code to remove the statements that were inserted by that tool. And then, we ran that object program on another tool and so on.

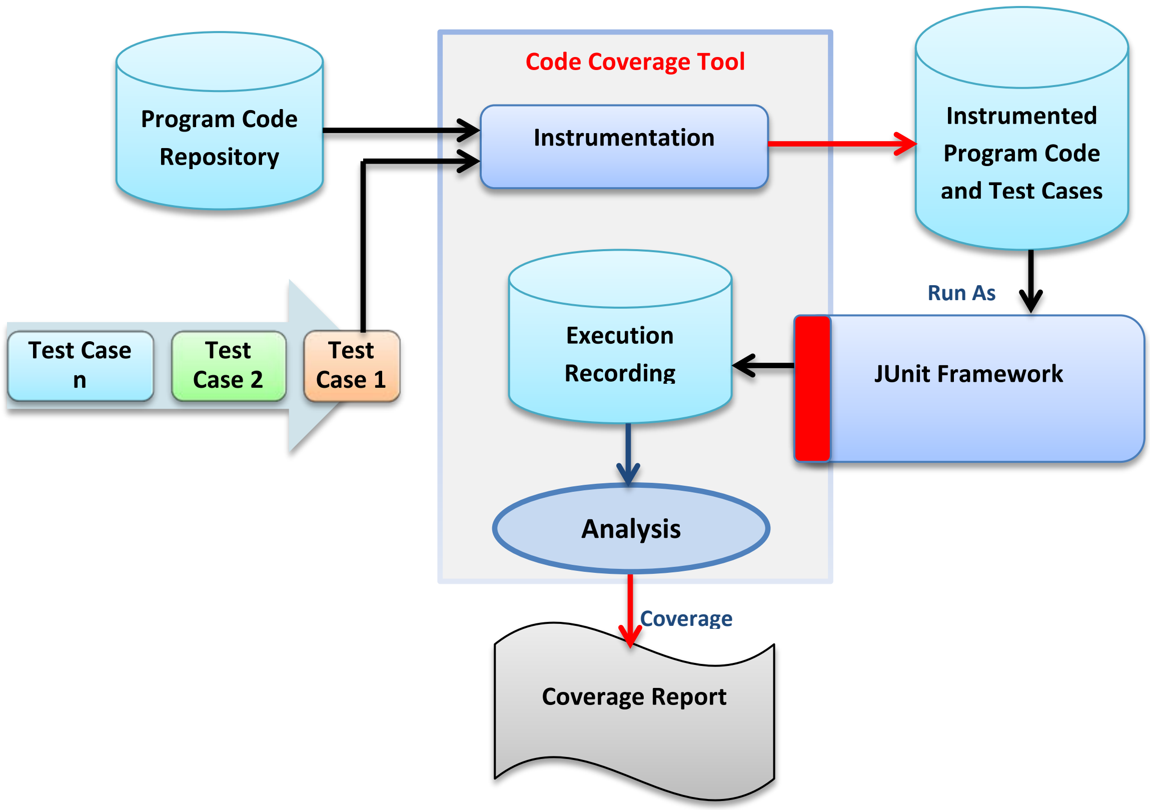


Figure 4.1. Testing Coverage Analysis Process with JUnit.

CHAPTER 5. DATA AND ANALYSIS

In this chapter, we show our findings. At first, our hypotheses associated with RQ1 were:

H1: There is a significant difference among the values that are measured by code coverage tools in terms of branch coverage.

H2: There is a significant difference among the values that are measured by code coverage tools in terms of line coverage.

H3: There is a significant difference among the values that are measured by code coverage tools in terms of statement coverage.

H4: There is a significant difference among the values that are measured by code coverage tools in terms of method coverage.

Also, the hypothesis associated with RQ2 was:

H5: Program size affects significantly the effectiveness of code coverage tools in terms of large programs.

To study the difference among code coverage tools, we used the ANOVA test to calculate the significance *p* overall, followed by Tukey's HSD test for multiple pair comparisons, which is used to show the difference between each two tools. We used the R, it is a programming language, to perform statistical analysis and we used the BoxPlotter [40], an online tool, for drawing the Boxplots.

Our findings are organized into four subsections. Each subsection shows the results of one coverage criterion. Therefore, within each coverage criterion, we present the data that we collected in two figures: the first figure shows the results for program sizes (small, medium, and large) through boxplots while the second figure shows the results of code coverage metrics of 21 object programs. And also, we provide a table that consists of the results of Tukey's HSD test for branch, line, and statement coverage.

5.1. Analysis of Results for Branch Coverage

This subsection is associated with hypothesis (H1). For evaluating the difference among code coverage tools in terms of branch coverage, we used Ecobertura, Djunit, and Clover as code coverage tools. We, also, performed the ANOVA test (df=2) for all tools per object programs, at a significance level (<0.05). Moreover, we performed the Tukey's HSD test to assess the difference between each two tools as well as the results of this test shown in Table 5.1. Finally, Figure 5.1 shows the data distribution of testing coverage for small, medium, and large programs using boxplots. Figure 5.2 presents the branch coverage results for all object programs.

- *Small programs*

Statistically, from the ANOVA test, there is a significant difference among code coverage tools with (*p= 0.04921*) overall. Table 5.1 shows the results that are calculated using Tukey's HSD test. This results show the difference between each two tools as well as *P-value* at a significance level (<0.05). We, basically, found that there is a statistical significant difference between Djunit and Clover. However, the results of comparison are slightly different trends between Ecobertura and Djunit among object programs. Figure 5.1 shows the distribution of coverage data from 0% to 100% of small object programs. Moreover, the medians of Ecobertura, Djunit, and Clover are not equal (70, 100, and 61.5 respectively). Therefore, Clover provides less code coverage while Djunit provides the highest code coverage in terms of branch coverage.

- *Medium programs*

Similar to the results of code coverage for small programs, there is, statistically, a significant difference among code coverage tools with (*p= 0.04735*) overall. Table 5.1 shows a statistical significant difference between Djunit and Ecobertura, but the results of comparison are slightly different trends between Clover and Djunit among object programs. Figure 5.1 shows the distribution of coverage of object programs. Furthermore, the median of Ecobertura, Djunit, and Clover are not equal (42.65, 93, and 63.6 respectively). Therefore, Ecobertura provides less code coverage while Djunit provides the highest code coverage.

- *Large programs*

Statistically, from the ANOVA test, there is a significant difference among code coverage tools with (*p= 0.006253*) overall. Table 5.1 shows a statistical significance difference among Djunit and Ecobertura as well as Djunit and Clover. This means the large programs affect the consistency of coverage metric values that are calculated by code coverage tools. Figure 5.1 shows the distribution of coverage of large object programs, and the median of Ecobertura, Djunit, and Clover are not equal (12.02, 7.2, and 65 respectively). Therefore, Clover provides less code coverage while Djunit provides the highest code coverage.

Moreover, Figure 5.2 shows the percentage of branch coverage among 21 object programs. Overall, the value of branch coverage metric, for the majority of object programs, is different.

In general, our hypothesis (*H1*) is supported in terms of branch coverage. Moreover, our results, in branch level, show that the hypothesis (*H5*) is supported as well. To sum up, the coverage information from Djunit, Clover, and Ecobertura are significantly different in terms of branch coverage, as we previously assumed.

5.2. Analysis of Results for Line Coverage

This subsection is associated with hypothesis (*H2*). For evaluating the difference among code coverage tools in terms of line coverage metric, we used Ecobertura, Djunit, and Code Pro as code coverage tools. In addition, we performed the ANOVA test (df=2) for all tools per object

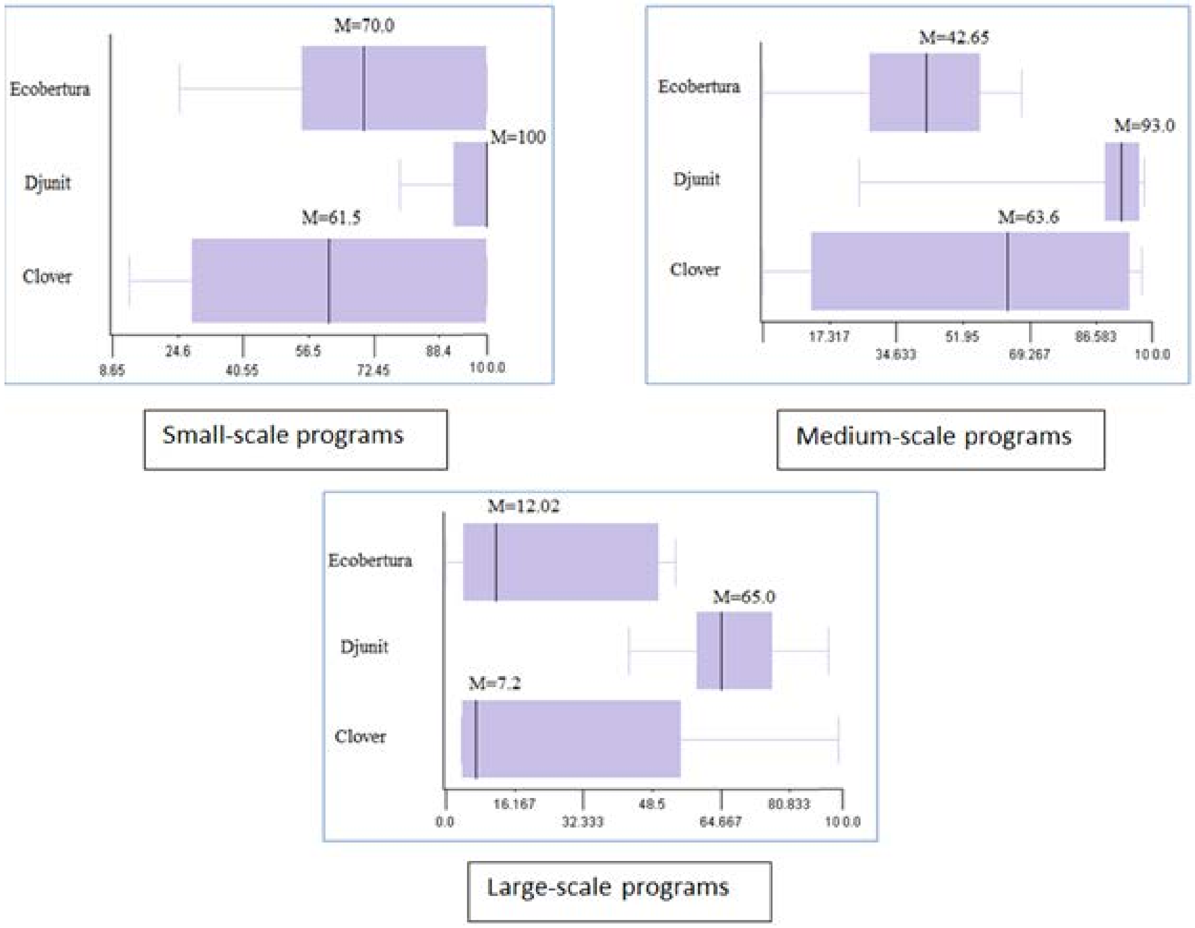


Figure 5.1. Boxplots of Branch Coverage Criterion for All Program Sizes (Small, Medium, and Large). The Horizontal Axes List the Percentage of Code Coverage and the Vertical Axes List Code Coverage Tools.

Table 5.1. Tukey's HSD Test Results of Branch Coverage, Per Size Category.

Program size	Tool comparison	Diff	P-value
small	Djunit-Clover	34.01429	**0.0489336***
	Ecobertura-Clover	8.22000	0.8114886
	Ecobertura-Djunit	-25.79429	0.1553821
medium	Djunit-Clover	27.61429	0.2298967
	Ecobertura-Clover	-15.41143	0.6151631
	Ecobertura-Djunit	-43.02571	**0.0403597***
large	Djunit-Clover	40.785714	**0.0183304***
	Ecobertura-Clover	-3.922857	0.9540430
	Ecobertura-Djunit	-44.708571	**0.0098260***
overall	Djunit-Clover	32.7571429	**0.0034681***
	Ecobertura-Clover	-0.4666667	0.9987123
	Ecobertura-Djunit	-33.2238095	**0.0029976***

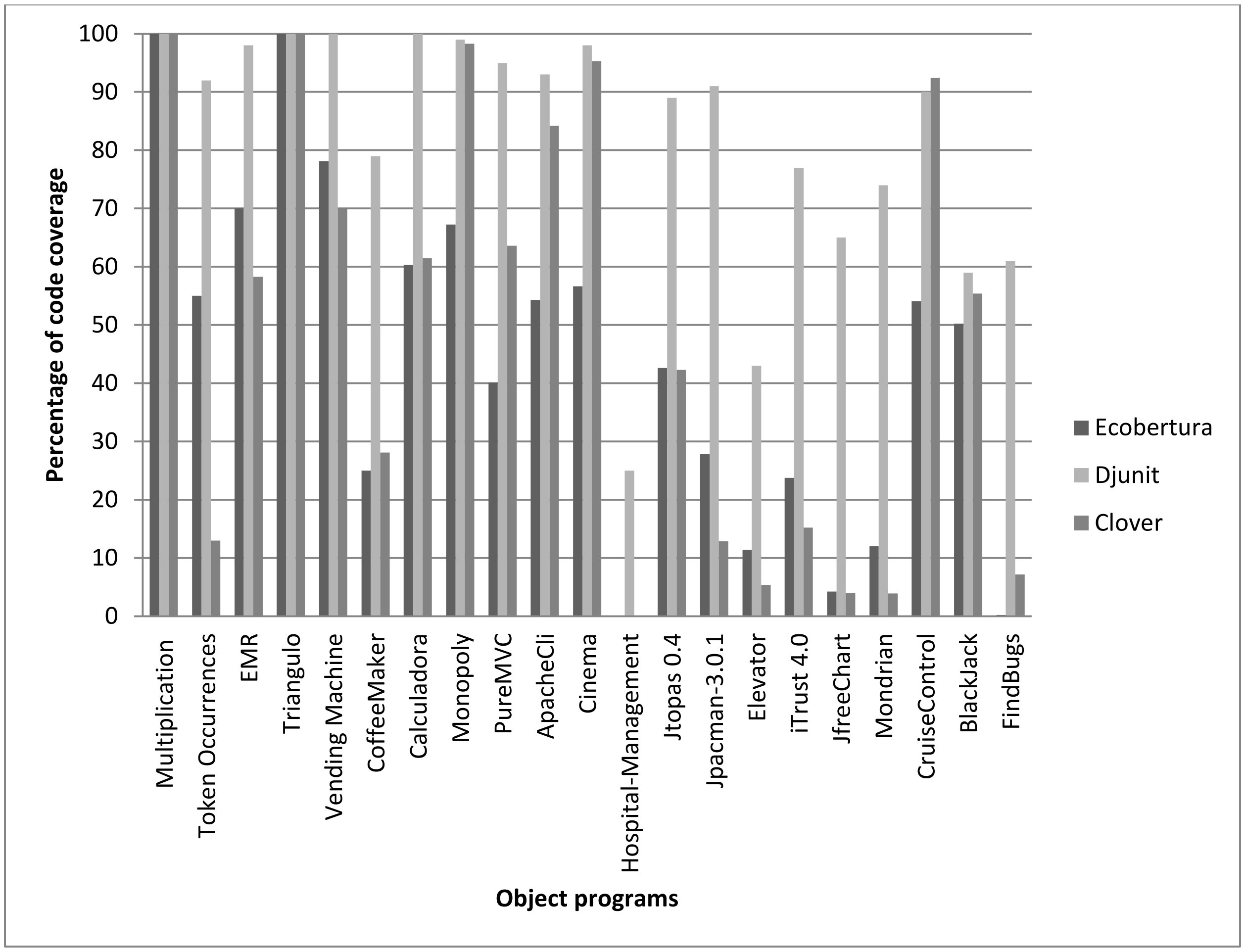


Figure 5.2. Percentage of Branch Coverage from Ecobertura, Djunit, and Clover for All Object Programs.

programs, at a significance level (<0.05). Moreover, we performed the Tukey's HSD test to assess the difference between each two tools as well as the results of this test shown in Table 5.2. Finally, Figure 5.3 shows the data distribution of testing coverage for small, medium, and large programs using boxplots. Figure 5.4 presents the line coverage results for the object programs.

- *Small programs*

Statistically, from the ANOVA test, there is no significant difference among code coverage tools with (p>0.05) overall. Table 5.2 shows the results that are calculated using Tukey's HSD test. This results show the difference between each two tools as well as *P-value* at a significance level (<0.05). Using the results shown in this table, we found that there is no statistical significant difference among the tools. However, the results of comparison are slightly different trends among them. In Figure 5.1 the boxplots of (Ecobertura and Djunit) and (Djunit and Code Pro) show quite a difference of line coverage metric but we observe the median of Ecobertura, Djunit, and Code Pro are, approximately, (equal 92.31, 95, and 93.1 respectively). Furthermore, the distribution of code coverage data is different. Obviously, the line coverage of small programs using these tools is consistent.

- *Medium programs*

Similar to the results of code coverage for small programs, there is no significant difference among code coverage tools with ($p>0.05$) overall in terms of medium programs and the results that are calculated using Tukey's HSD test as well. In Figure 5.1 the boxplots show, explicitly, quite a difference of distribution of line coverage data and we can note that the median of Ecobertura, Djunit, and Code Pro are not equal (39.02, 50, and 24.5 respectively). Therefore, the line coverage of medium programs using these tools slightly supports our hypothesis (*H2).*

- *Large programs*

Statistically, from the ANOVA test, there is a significant difference among code coverage tools with ($p= 0.04465$) overall. Obviously, Table 5.2 shows a statistical significant difference between Code Pro and Ecobertura. This means the large programs affect the consistency of coverage metric values that are computed by code coverage tools, but the results of pair comparisons are slightly different trends between Code Pro and Djunit among object programs.

Figure 5.2 shows the percentage of line coverage among object programs. Overall, the code coverage, for the majority of object programs, is different. In general, our hypothesis (*H2*) is supported in terms of line coverage especially for large programs. Moreover, our results, in

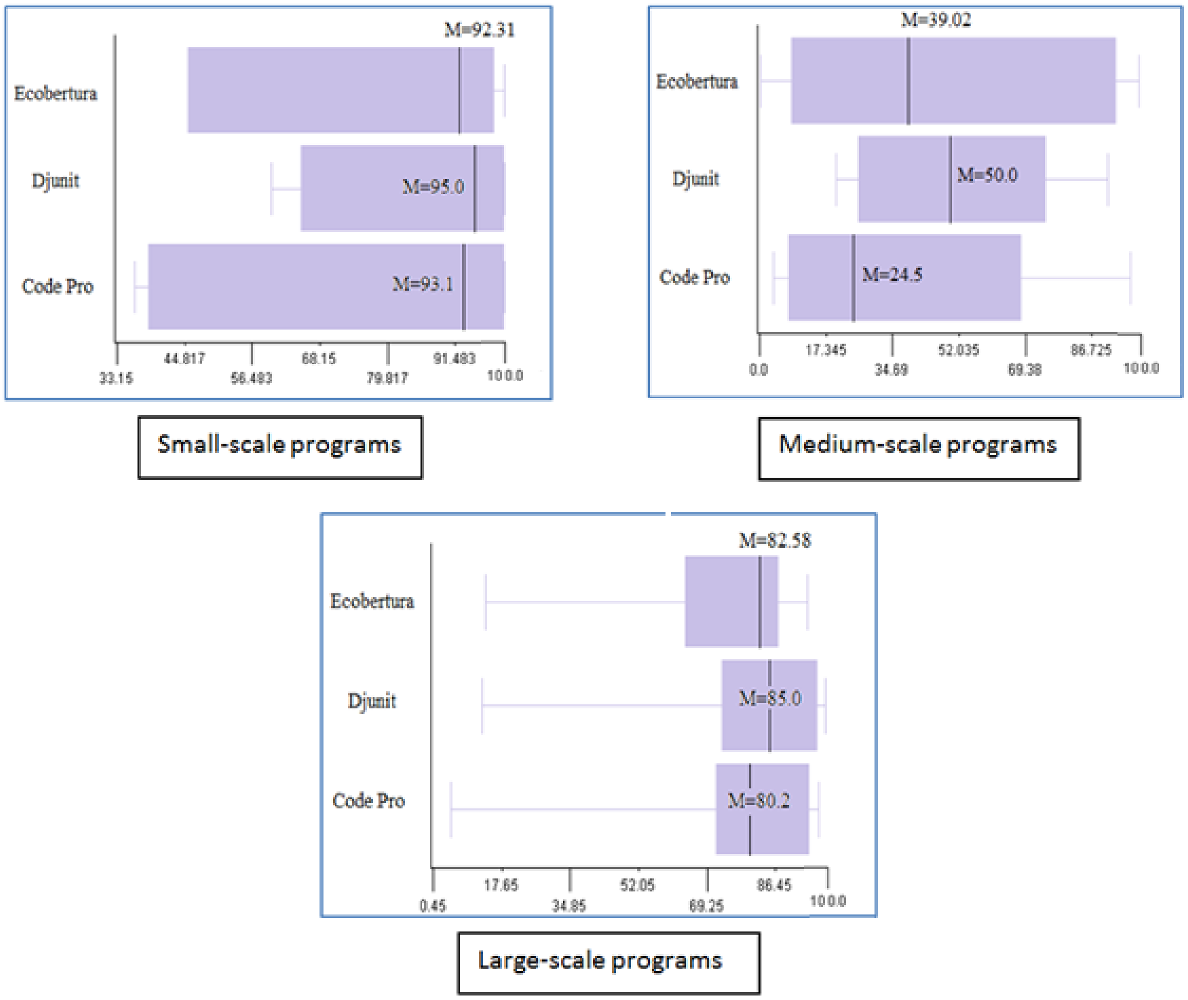


Figure 5.3. Boxplots of Line Coverage Criterion for All Program Sizes (Small, Medium, and Large). The Horizontal Axes List the Percentage of Code Coverage and the Vertical Axes List Code Coverage Tools.

line coverage level, show that the hypothesis (*H5*) is supported as well. To sum up, the results for each Djunit, Code Pro, and Ecobertura are different in terms of line coverage, as we previously assumed.

5.3. Analysis of Results for Statement Coverage

We examined the hypothesis (H3). For evaluating the difference among code coverage tools in terms of statement coverage metric, we used Eclemma, Clover, and Code Pro as code coverage tools and we performed the ANOVA test (df=2) for all tools per object programs, at a significance level (<0.05). Moreover, we performed the Tukey's HSD test to assess the

Table 5.2. Tukey's HSD Test Results of Line Coverage, Per Size Category.

Program size	Tool comparison	Diff	P-value
small	Djunit-CodePro	14.013243	0.5847521
	Ecobertura-CodePro	23.43533	0.2547854
	Ecobertura-Djunit	15.79429	0.8213264
medium	Djunit-CodePro	-22.61429	0.2340765
	Ecobertura-CodePro	25.48643	0.1685425
	Ecobertura-Djunit	-13.98321	0.7304267
large	Djunit-CodePro	26.914286	0.1375682
	Ecobertura-CodePro	34.712857	**0.0457245***
	Ecobertura-Djunit	7.798571	0.8305967
overall	Djunit-CodePro	13.3047619	0.3348705
	Ecobertura-CodePro	13.4204762	0.3286707
	Ecobertura-Djunit	0.1157143	0.9999154

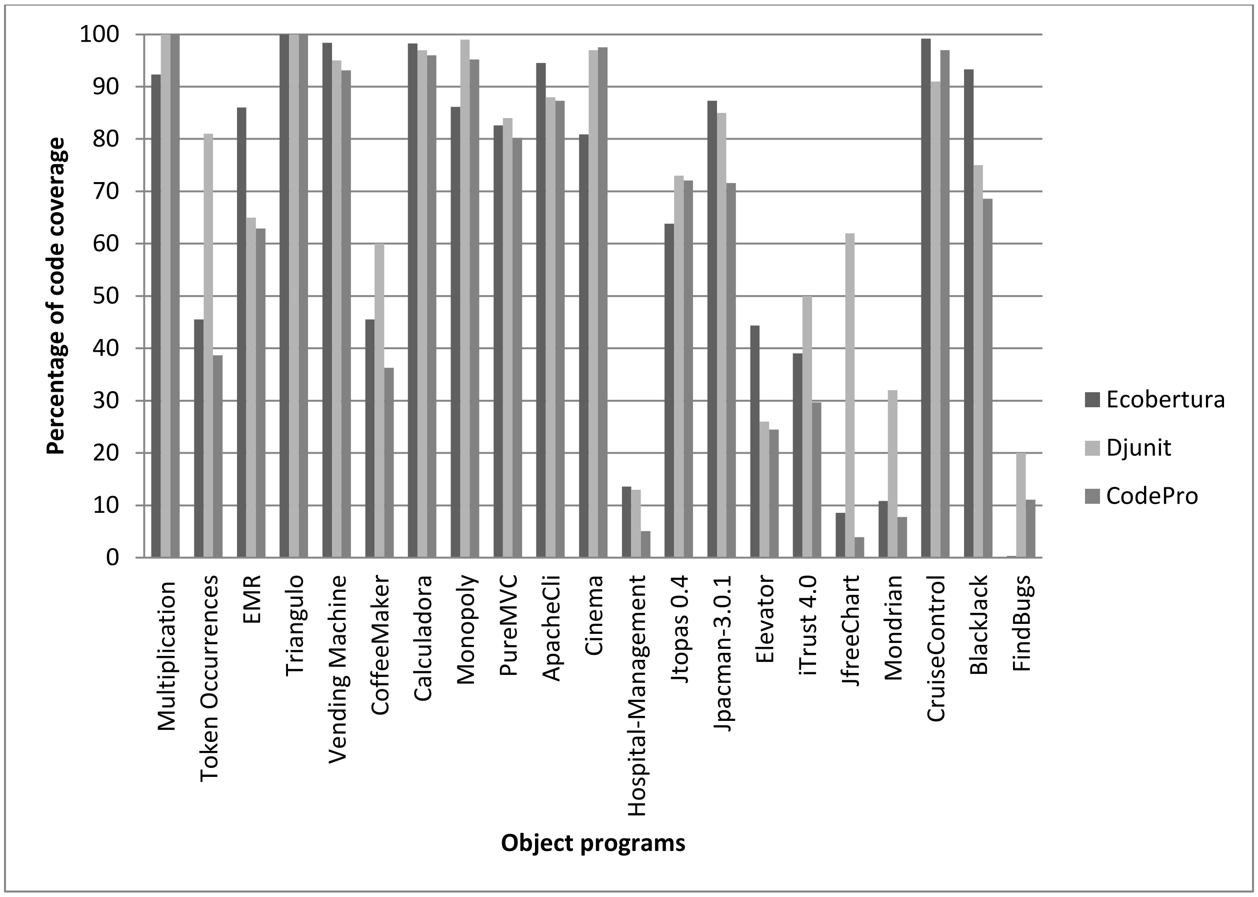


Figure 5.4. Percentage of Line Coverage from Ecobertura, Djunit, and Code Pro for All Object Programs.

significance level (<0.05). Moreover, we performed the Tukey's HSD test to assess the difference between each two tools as well as the results of this test shown in Table 5.3. Finally, Figure 5.5 shows the data distribution of testing coverage for small, medium, and large programs using boxplots. Figure 5.6 presents the statement coverage results for the object programs.

- *Small programs*

We used the ANOVA test that shows that there is no significant difference among code coverage tools with (p>0.05) overall in terms of small programs. Moreover, Table 5.3 shows the results that are calculated using Tukey's HSD test, which shows a quite difference between each two tools of small programs. Furthermore, Figure 5.5 the boxplots show, approximately, the distribution of statement coverage of object programs is slightly similar.

- *Medium programs*

Similar to the results of code coverage of small programs, there is no significant difference among code coverage tools with (p>0.05) overall. Moreover, Table 5.3 shows the results that are calculated using Tukey's HSD test, which shows quite a difference between each two tools for medium programs. Furthermore, Figure 5.5 the boxplots show, approximately, the distribution of statement coverage of object programs is slightly similar.

- *Large programs*

Statistically, from the ANOVA test, there is a significant difference among code coverage tools with (*p= 0.04441*) overall. Table 5.3 shows a statistical significance difference between Code Pro and Eclemma. This means the large programs affect the consistency of coverage metric values. However, the results of comparison are slightly different trends between Eclemma and Clover among object programs.

Figure 5.6 shows the percentage of statement coverage among object programs. Overall, the code coverage, for the majority of object programs, is different. In general, our hypothesis (*H3*) is supported in terms of statement coverage especially for large programs. Moreover, our results, in statement level, show that the hypothesis (*H5*) is supported as well. In summary, the results for each Eclmma, Code Pro, and Clover are different in terms of statement coverage, as we previously assumed.

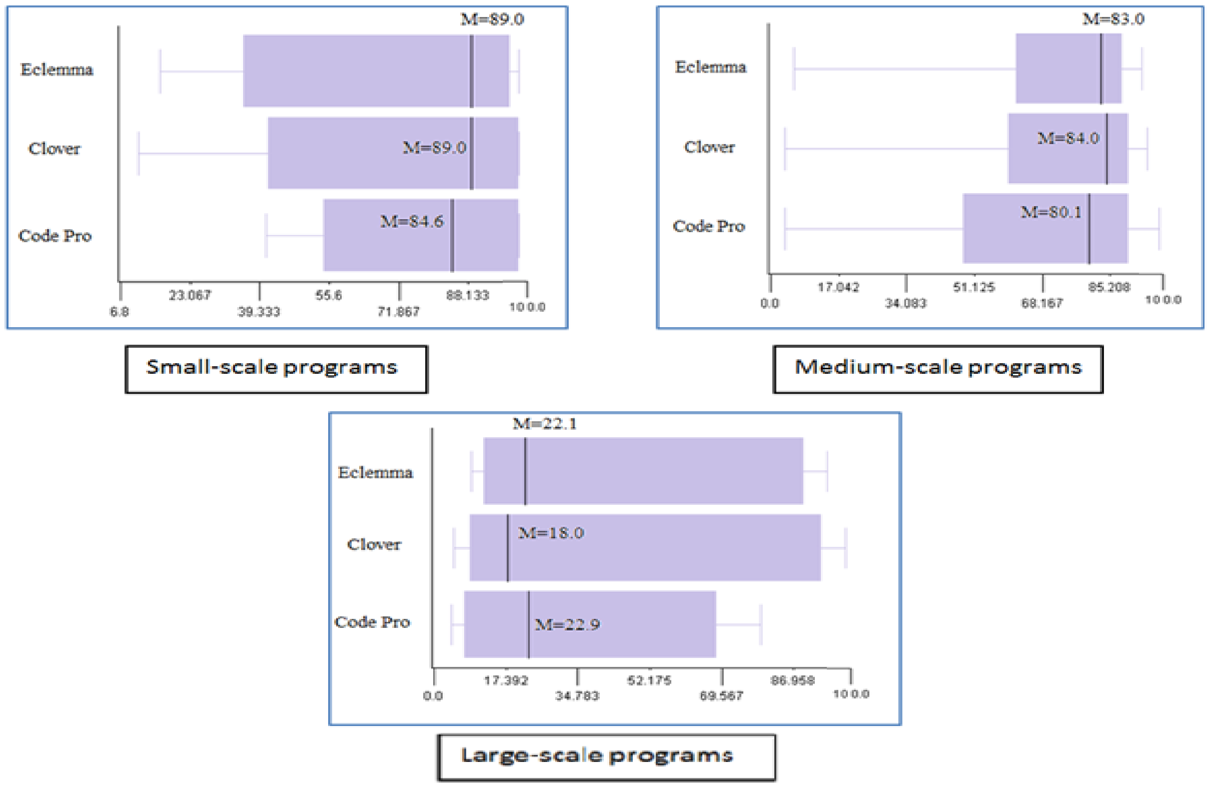


Figure 5.5. Boxplots of Statement Coverage Criterion for All Program Sizes (Small, Medium, and Large). The Horizontal Axes List the Percentage of Code Coverage and the Vertical Axes List Code Coverage Tools.

Table 5.3. Tukey's HSD Test Results of Statement Coverage, Per Size Category.

Program size	Tool comparison	Diff	P-value
small	CodePro-Clover	7.249744	0.8245721
	Eclemma-Clover	-28.45751	0.2147842
	Eclemma-CodePro	26.21478	0.1825475
medium	CodePro-Clover	-9.245871	0.7142568
	Eclemma-Clover	29.25475	0.2421244
	Eclemma-CodePro	27.54785	0.1454752
large	CodePro-Clover	-13.60000	0.6781314
	Eclemma-Clover	29.15714	0.1912470
	Eclemma-CodePro	42.75714	**0.0395174***
overall	CodePro-Clover	-5.280952	0.8680341
	Eclemma-Clover	8.371429	0.7018164
	Eclemma-CodePro	13.652381	0.3942966

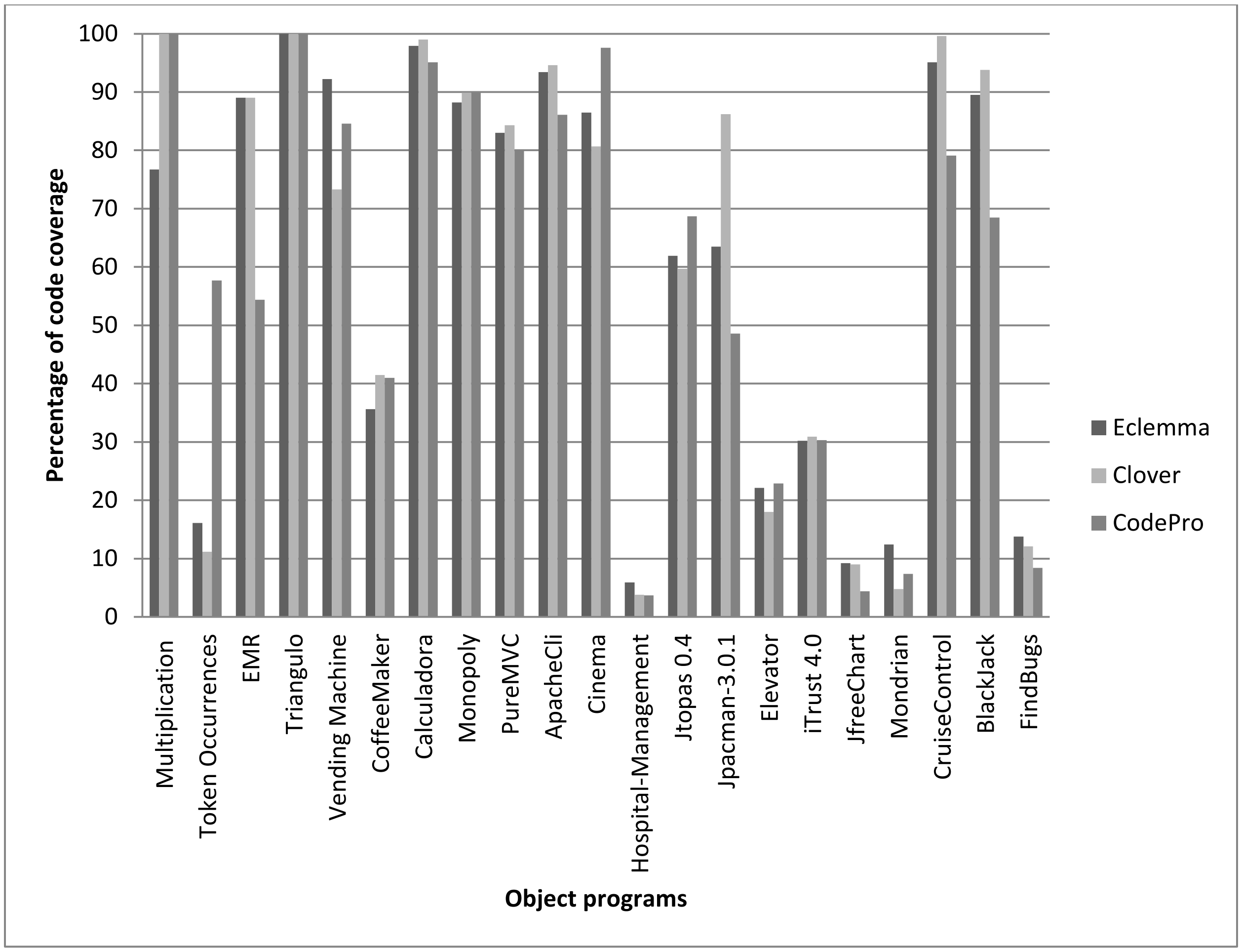


Figure 5.6. Percentage of Statement Coverage from Eclemma, Clover, and Code Pro for All Object Programs.

5.4. Analysis of Results for Method Coverage

To examine the hypothesis (*H4*), we evaluated the difference among code coverage tools in terms of method coverage metric. We used Clover, and Code Pro as code coverage tools. We performed the ANOVA test (df=1) for both tools per object programs, at a significance level (<0.05). Figure 5.7 shows the data distribution of testing coverage for small, medium, and large programs using boxplots. Figure 5.8 presents the method coverage results for object programs.

- *Small programs*

Obviously, from the ANOVA test, there is a significant difference between Clover and Code Pro with (*p= 0.0356*) in terms of small programs. Also, Figure 5.7 shows the boxplots that present the distribution of method coverage data for small programs using Clover and Code Pro. We can observe the medians of Clover and Code Pro are significantly different (75.7 and 95 respectively).

- *Medium programs*

Similar to the results of code coverage of small programs, there is a significant difference between Clover and Code Pro with (*p= 2.365e-05*) in terms of medium programs. Moreover, Figure 5.7 shows how the distribution of method coverage data is deferent between two tools, but the median of method coverage data for both Clover and Code Pro are quite equal (82.2 and 83.2 respectively).

- *Large programs*

Statistically, From the ANOVA test, there is a significant difference between Clover and Code Pro with (*p= 0.0001062*) in terms of large programs. Furthermore, Figure 5.7 shows, for both coverage tools, the distribution of coverage data is different as well as the median of Clover and Code Pro is not equal (21 and 33.3 respectively).

Figure 5.8 shows the percentage of method coverage among object programs. Overall, the method coverage, for majority of object programs, is different. In general, our hypothesis (*H4*) is supported in terms of method coverage. Moreover, our results, in method level, show that the hypothesis (*H5*) is strongly supported as well. To sum up, the results for both Code Pro and Clover, are different in terms of method coverage, as we previously assumed.

5.5. Threats to Validity

In general, any controlled experiment is subject to threats to validity. So, these threats must be taken into account in order to assess their impact on results. In this section, we describe the internal, external, and construct threats to the validity of our findings.

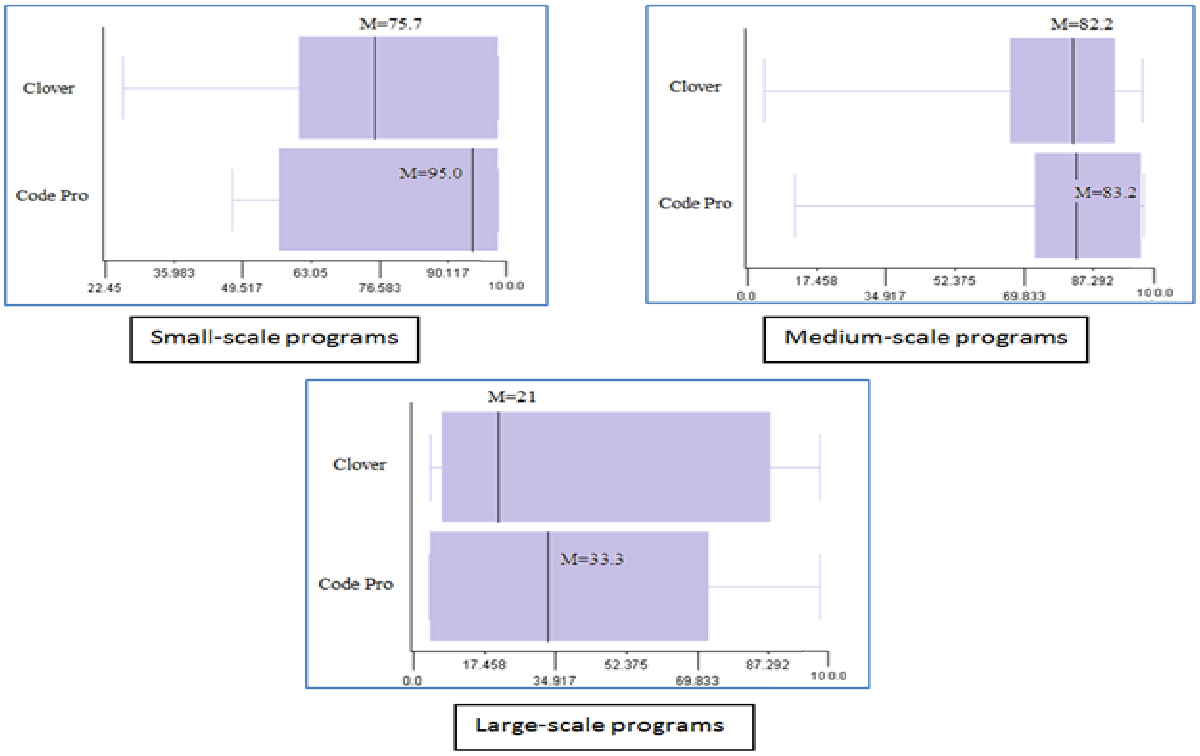


Figure 5.7. Boxplots of Method Coverage Criterion for All Program Sizes (Small, Medium, and Large). The Horizontal Axes List the Percentage of Code Coverage and the Vertical Axes List Code Coverage Tools.

5.5.1. Internal Validity

One of most expected internal threats is potential faults in the tools that we used for statistical analysis, which may effect on inferences we made about consistency and reliability of code coverage tools. To avoid this threat, we validated the tools that were used in this experiment comparing to similar purpose tools. For instance, we calculated the statistical tests using R language, and then we compared the results with other online statistical packages. Moreover, this could increase the confidence in our results.

5.5.2. External Validity

We considered three issues that affect the generalization of our findings. First, code coverage tools representativeness. We have considered 5 code coverage tools that are integrated to Eclipse IDE as plugins. Other code coverage tools may have different aspects, features, and

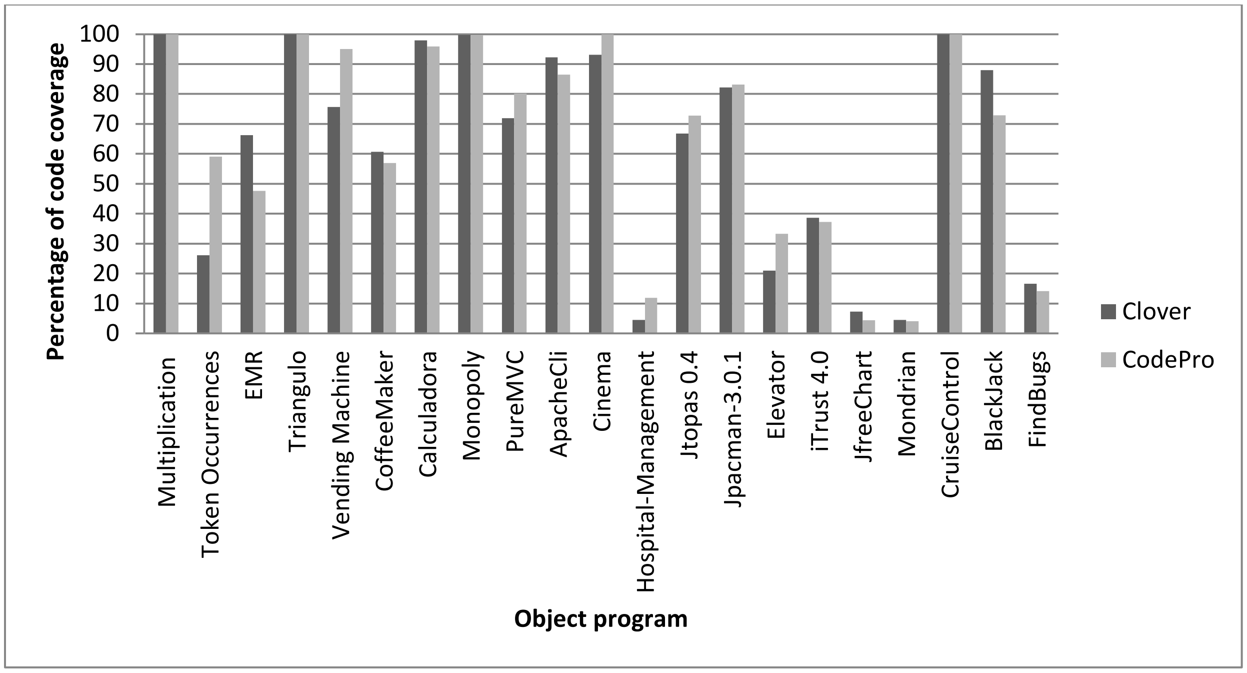


Figure 5.8. Percentage of Method Coverage from Clover and Code Pro for All Object Programs.

ways for calculating the coverage metrics. Therefore, we cannot generalize our results to code coverage tools (open source or commercial) that use other test coverage measures than the coverage metrics that we chose to study. Investigating this issue can only be done through studies involving diverse commercial and open source tools as well as involving tools for other programming languages. Therefore, we used 3 open source coverage tools, one freeware coverage tool developed by google, and only one commercial coverage tool.

Second, object program representativeness issue. We only used 21 Java object programs, which may limit the external validity of our findings. To avoid this threat, we used object programs with various sizes from 33 to 237564 LOC. In addition, these object programs represent different domains.

The third issue involves testing process representativeness issue. In our experiment, we used only JUnit testing framework in order to perform the testing process. So, all test cases, that we utilized, were generated by JUnit framework. In practice, there are many testing frameworks available that can be used with Eclipse IDE. To control this threat, we used JUnit test cases for all object programs in addition to JUnit can support all selected code coverage tools. However, we cannot deny that we need to replicate our experiments using other testing frameworks such as TestNG, which is a multi-purpose testing framework. And also, we need to run all object programs with test cases in the same conditions and environment in order to generalize our results.

5.5.3. Construct Validity

The coverage metrics that we have considered (branch, statement, line, and method coverage) are not the only possible coverage metrics. Therefore, to handle this issue, future studies will be conducted using other coverage metrics such as path coverage and loop coverage.

CHAPTER 6. DISCUSSION

Our findings provide an evidence of our assumption that said there is a significant difference among code coverage tools in terms of branch, line, statement, and method coverage criteria. Obviously, we observed that the large programs have an impact on the consistency of code coverage of code coverage tools rather than the small and medium programs. In fact, the results of code coverage tools are close to each other in terms of statement and line coverage especially for small and medium programs. Furthermore, we revealed that the branch and method coverage are, significantly, different among code coverage tools for all program sizes.

In fact, we believe that the reasons behind this difference may relate to the following:

- Instrumentation
- Variety of code analysis
- Size and complexity object programs
- Definitions of code coverage metrics

6.1. Instrumentation

We explained that these tools instrument the object programs in different ways. On the one hand, the Clover instruments the source code, but on the other hand, the Code Pro, Djunit, Eclemma, and Ecobertura instrument the byte code. Instrumentation of byte code gathers branch coverage in an indirect way while the instrumentation of source code gathers branch coverage in a direct way. For example, if there is "*nested if statement*" in a single line, the source code instrumentation can see these statements and then provides the code coverage information. However, the byte code instrumentation may not be able to see these statements but it can provide coverage information about one line wrapped around the condition. Furthermore, Logozzo and Fähndrich [41] said that the byte code analysis process introduces precision loss compared to source code analysis. Obviously, we found a significant difference between Clover and Code Pro in terms of method coverage. Moreover, our results show a significant difference between Clover and Djunit in terms of branch coverage. Those two cases provide evidence that the instrumentation way may affect the coverage gathering process of the code coverage tools.

6.2. Variety of Code Analysis

Many coverage metrics are available and each metric needs an analyzer to be gathered. Some of these metrics are considered as simple and weak such as statement and line coverage metrics [43][12]. In contrast, some coverage metrics are considered as powerful but they are hard to measure such as branch coverage metric. On the one hand, our results show that there is a significant difference among code coverage tools in terms of branch and method coverage. So, we think that this difference may be due to the difficulties of the way of measuring and analyzing of these metrics. On the other hand, there is quite a difference among the code coverage tools in terms of statement and line coverage, which may be due to the simplicity of measuring and analyzing of these metrics.

6.3. Size and Complexity Object Programs

The increasing size and complexity of programs has increased the challenges of code coverage analysis process [42]. Our results show that the coverage criteria (branch, statement, line, and method) of large programs are significantly different. For example, our findings show a significant difference among (Djunit and Clover) and (Ecobertura and Djunit) of the large object programs in terms of branch coverage. In addition, there is a significant difference between Ecobertura and Code Pro of large object programs in terms of statement coverage. Therefore, the consistency of coverage metric values of code coverage tools has been affected by large size and complex programs but the degree of this influence varies from one tool to another.

6.4. Definitions of Code Coverage Metrics

The definitions of code coverage metrics that have been used by the developers of code coverage tools are quite different. Table 6.1 shows the definitions that are provided by the websites of these tools. For example, we can see how the definitions of branch coverage are slightly different among code coverage tools. On one hand, these definitions are expressed in simple statements that do not provide enough details about which statement, line, branch, or method is considered in code coverage and which is not. On the other hand, through practical experiment, we found some cases that interpret why the difference among the code coverage tools exists. Therefore, we can infer that the definitions of coverage metrics are unclear. In addition, the websites of Eclemma and Djunit do not provide any definition for the coverage metrics that are calculated by their tools.

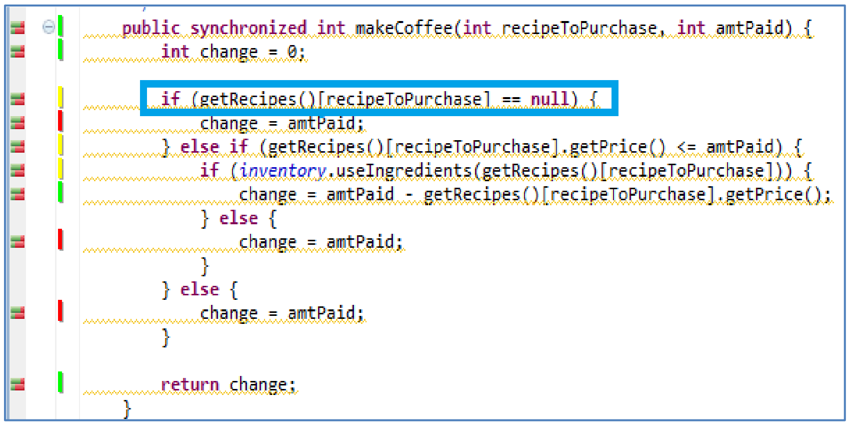
In order to explain how the definition of code coverage metric influences the percentage of code coverage, we found the following cases throughout our experiment:

6.4.1. Branch Coverage

Ecobertura and Clover count the branch coverage for both cases of conditional (true and false). By contrast, Djunit counts only the branch coverage when the conditional is true. The definition of branch coverage that is provided by Djunit does not mention any detail about this case. Figure 6.1 shows a piece of code of CoffeeMaker program that was used in our experiment. In fact, Ecobertura understands that if getRecipes() is not null, the behavior of the system might be different. Therefore, we have to generate a test case to cover this branch and also we covered the getRecipes() that is being null. In addition, Clover shows the same statement that is partially exercised and highlighted in yellow color. For that, the percentage of code coverage might be influenced by the partial coverage that was calculated in both Ecobertura and Clover. By contrast, in order for Djunit to be able to see this block of code, the statement has to be exercised at least twice, but Djunit counts this piece of code “was no hit”. This observation supports our

Table 6.1. Definitions of Code Coverage Metrics.

Tool	Coverage Metric	Definition
Clover	Branch coverage	A code coverage metric that measures which possible branches in flow control structures are followed. Clover does this by recording if the boolean expression in the control structure evaluated to both true and false during execution.
	Statement coverage	A code coverage metric that measures which statements in a body of code have been executed through a test run, and which statements have not.
	Method coverage	A code coverage metric that measures whether a method was entered at all during execution.
Code Pro Analytix	Method coverage	The number of methods containing at least one basic block that was executed at least once, divided by the total number of methods.
	Line coverage	The number of lines of code containing at least one basic block that was executed at least once, divided by the total number of lines of code.
	Statement coverage	The number of statements in basic blocks that were executed at least once, divided by the total number of statements.
Ecobertura	Line coverage	The amount of (useful) lines of code that are executed. A useful line is a line of code where something happens (so no curly braces or newlines).
	Branch coverage	The coverage of the decision points in a class (this can be a if/then/else loop. When the test suite only reaches the first part of the if-loop and skips the else part we will have 50% branch coverage).
Eclemma	Statement coverage	N/C
Djunit	Line coverage	N/C
	Brach coverage	N/C



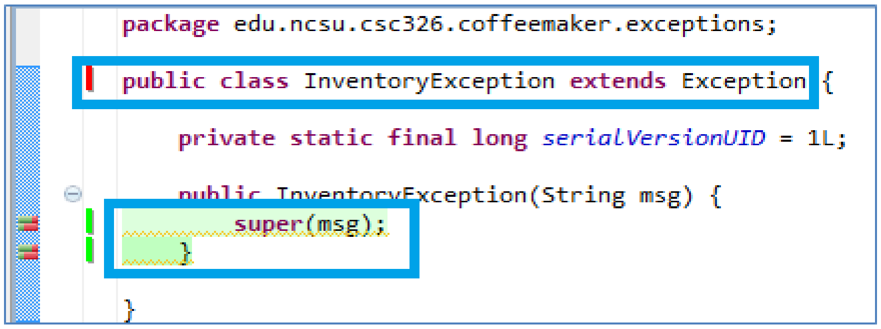
```
    public synchronized int makeCoffee(int recipeToPurchase, int amtPaid) {
        int change = 0;

        if (getRecipes()[recipeToPurchase] == null) {
            change = amtPaid;
        } else if (getRecipes()[recipeToPurchase].getPrice() <= amtPaid) {
            if (inventory.useIngredients(getRecipes()[recipeToPurchase])) {
                change = amtPaid - getRecipes()[recipeToPurchase].getPrice();
            } else {
                change = amtPaid;
            }
        } else {
            change = amtPaid;
        }

        return change;
    }
```

Figure 6.1. Screenshot of If Statement of CoffeeMaker Program under Coverage Analysis Process.

findings, which shows that there is a significant difference between Ecobertura and Djunit as well as Clover and Djunit in terms of branch coverage. As a consequence, we believe that the definition of branch coverage needs to be rewritten more precisely and in more detail for all code coverage tools.

6.4.2. Exception Class

Code Pro Analytix and Ecobertura count the line coverage for exception classes. By contrast, Djunit does not count the line coverage for exception classes. The definitions of these tools do not provide any kind of information on how to calculate the line coverage for exception classes, but we noticed that Code Pro Analytix and Ecobertura do that while Djunit does not do that. Therefore, this lack of information makes the line coverage unclear. Through measuring the line coverage metric for CoffeeMaker program, we found a slight difference among the code coverage tools. One reason is that Djunit does not measure line coverage for exception handling code as shown in the piece of code in Figure 6.2. Djunit shows that there was no test case that exercised line 5, when in fact it was executed. Therefore, line coverage for “InventoryException” class was 0%. Subsequently, the overall code coverage might be influenced by this kind of coverage calculation. By contrast, the line coverage comes out 100% for the same class using both Code Pro Analytix and Ecobertura. Moreover, percentage of method coverage can be affected as well. Therefore, absence of many exception classes in a given program can affect significantly the code coverage.



```
package edu.ncsu.csc326.coffeemaker.exceptions;

public class InventoryException extends Exception {

    private static final long serialVersionUID = 1L;

    public InventoryException(String msg) {
        super(msg);
    }

}
```

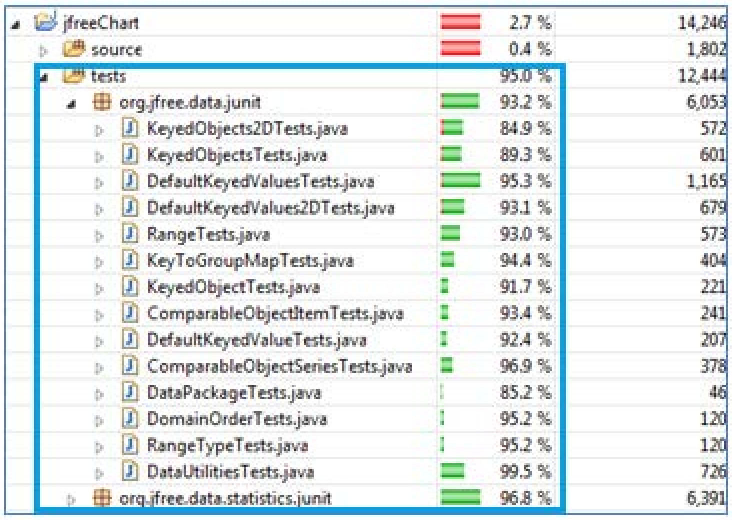
Figure 6.2. Screenshot of InventoryException Class of CoffeeMaker Program under Coverage Analysis Process.

6.4.3. JUnit Tests Package

Eclemma counts the statement coverage for both source code and JUnit test packages. In other words, Eclemma considers the JUnit test packages as source code. By contrast, Code Pro Analytix and Clover count only the statement coverage for the source code. This case shows us the definition of statement coverage for Eclemma is unclear and not precise. Figure 6.3 shows that the coverage report that was generated by Eclemma for the JfreeChart program. This report includes the coverage for both the source code and JUnit tests package. Unexpectedly, Eclemma counted the statement coverage for both the source code and JUnit tests packages. Furthermore, Code Pro Analytix and Clover do not include the coverage for JUnit tests package in their coverage results. Therefore, this is one of the main reasons behind the significant difference between Code Pro Analytix and Eclemma. Our observation shows that there is no code coverage tool that includes code coverage for test files except Eclemma. Therefore, statement coverage metric is used to measure the coverage of source code files of the program not the coverage of test files.

6.4.4. Executable Methods and Classes for Code Coverage

The definitions of code coverage metrics that are provided by the code coverage tools do not provide any information about the classes and methods that are excluded by default from code coverage such as abstract methods, native methods, and Java interface classes. To run the programs correctly, the compiler implements certain Java language constructs like inner class access. For example, Clover provides a feature to exclude any method or class to be considered in code coverage. Therefore, there is a need to know what are the executable methods and classes for each code coverage tool to understand how these tools calculate the code coverage.



jfreeChart	2.7 %	14,246
source	0.4 %	1,802
tests	95.0 %	12,444
org.jfree.data.junit	93.2 %	6,053
KeyedObjects2DTests.java	84.9 %	572
KeyedObjectsTests.java	89.3 %	601
DefaultKeyedValuesTests.java	95.3 %	1,165
DefaultKeyedValues2DTests.java	93.1 %	679
RangeTests.java	93.0 %	573
KeyToGroupMapTests.java	94.4 %	404
KeyedObjectTests.java	91.7 %	221
ComparableObjectItemTests.java	93.4 %	241
DefaultKeyedValueTests.java	92.4 %	207
ComparableObjectSeriesTests.java	96.9 %	378
DataPackageTests.java	85.2 %	46
DomainOrderTests.java	95.2 %	120
RangeTypeTests.java	95.2 %	120
DataUtilitiesTests.java	99.5 %	726
org.jfree.data.statistics.junit	96.8 %	6,391

Figure 6.3. Screenshot of Coverage Report for JfreeChart Program.

All the above cases that we discussed show how the definitions of code coverage metrics are unclear. The developers of these tools need to provide more precise and detailed definitions of code coverage metrics. In addition, the users of these tools need to be aware about what methods and classes are included and what are excluded from the code coverage.

To sum this, the coverage criteria that are measured using code coverage tools may be affected by instrumentation method, variety of code analysis, unclear definitions of code coverage metrics, and size and complexity of object programs.

CHAPTER 7. CONCLUSION AND FUTURE WORK

In this paper, we have conducted a controlled experiment in order to investigate how the values of coverage metric of the code coverage tools are consistent in terms of line, statement, branch, and method coverage. Our findings show that the branch and method coverage metrics are significantly different. By contrast, the statement and line coverage metrics are slightly different. Therefore, some code coverage tools may mislead the practitioners as well as researchers. For example, these tools could be used to judge the effectiveness of proposed technique in software testing.

To our knowledge, there is no previous empirical study that has investigated the consistency of the coverage information that is calculated using code coverage tools. In addition, no empirical study has made the specific contributions we make here. This paper makes four contributions. First, this is the first study to our knowledge to investigate, empirically, the differences among code coverage tools in terms of line, statement, branch, and method coverage metrics. Second, this is the first study to our knowledge to investigate the effect of program size on the effectiveness of code coverage tools. Third, this study sheds insights into the effect of coverage results variance on decision making. Fourth, this study attempts to interpret why this variance among code coverage tools is found.

More importantly, we hope that this effort assists the practitioners and research community to increase the understanding of the code coverage tools, to choose the appropriate code coverage tool that improves the software quality, and to enhance the software testing process.

One direction for our future work is to perform various large scale studies to evaluate the impact of program versions on effectiveness of code coverage tools in terms of branch and method coverage metrics. Another direction for future work is to extend our current study to use additional object programs and code coverage tools for other programming languages. Finally, we will consider other aspects of code coverage tools such as the path coverage, loop coverage, and the visualization provided through these tools.

REFERENCES

[1] C. Youngblut and B. Brykczynski, “An examination of selected software testing tools,” IDA Paper P- 2769, Inst. for Defense Analyses, Alexandria, Va., Dec. 1992.

[2] C. Youngblut and B. Brykczynski, “An examination of selected software testing tools,” Supp. IDA Paper P- 2925, Inst. for Defense Analyses, Alexandria, Va., Oct. 1993.

[3] M. Kessis, Y. Ledru, and G. Vandome, “Experiences in Coverage Testing of a Java Middleware,” In Fifth International Workshop on Software Engineering and Middleware (SEM’05), Lisbon, Portugal, ACM Press, 2005, pages 39–45.

[4] Q. Yang, J. J. Li, and D. Weiss, “A survey of coverage based testing tools,” In AST’06. ACM, 2006, pp. 99–103.

[5] M. Shahid and S. Ibrahim, “An Evaluation of Test Coverage Tools in Software Testing,” 2011 International Conference on Telecommunication Technology and Applications Proc. of CSIT vol.5, 2011.

[6] Y. Woo Kim, “Efficient Use of Code Coverage in Large-Scale Software Development,” IBM Center for Advanced Studies Conference, Proceedings of the 2003 Conference of the Centre for Advanced Studies on Collaborative research**,** Toronto, Ontario, Canada, 2003, pp. 145 – 155.

[7] J.B. Michael, B.J. Bossuyt, B.B. Snyder, “Metrics for Measuring the Effectiveness of Software-Testing Tools,” In: 13th International Symposium on Software Reliability Engineering (ISSRE 2002). Annapolis, Maryland, Nov 12-15, 2002.

[8] L.S. Priya, A. Askarunisa, and N. Ramaraj, “Measuring the Effectiveness of Open Coverage Based Testing Tools,” Journal of Theoretical and Applied Information Technology , Vol.5, No.5, 2005, pages 499-514.

[9] E. Kajo-Mece and M., Tartari, “An Evaluation of Java Code Coverage Testing Tools,” *Local Proceedings* © 2012 Faculty of Sciences, University of Novi Sad, *BCI’12*, Novi Sad, Serbia. Sep 16–20, 2012.

[10] C. Gaffney, C. Trefftz, and P. Jorgensen, “Tools for coverage testing: Necessary but not sufficient,” Journal of Computing Sciences in Colleges, v.20 n.1, Oct. 2004, pp.27-33.

[11] J.J. Li, D.Weiss, and H. Yee, “Code-coverage guided prioritized test generation,” Inf. Softw. Technol., 48, 2006, pp. 1187-1198.

[12] B. Beizer, “Software testing techniques,” 2nd edition. New York: Van Nostrand Reinhold, 1990.

[13] R. Lingampally, A. Gupta, and P. Jalote, “A Multipurpose Code Coverage Tool for Java,” In Proceedings of the 40th Annual Hawaii International Conference on System Sciences, IEEE Computer Society, 261b, 2007.

[14] S. Cornett “Code Coverage Analysis“, available on the web at: URL: http://www.bullseye.com/coverage.html.

[15] J. Hofer, “Ecobertura” http://ecobertura.johoop.de/ , retrieved on February 2013.

[16] M.R. Hoffmann, B. Janiczak, and E. Mandrikov, “Eclemma” http://www.eclemma.o rg/index.html, retrieved on February 2013.

[17] “Clover” http://www.atlassian.com/software/clover/overview, retrieved on February 2013.

[18] “Djunit” http://works.dgic.co.jp/djunit/, retrieved on February 2013.

[19] “Code Pro Analytix” http://dl.google.com/eclipse/inst/codepro/latest/3.6, retrieved on February 2013.

[20] http://sourceforge.net, retrieved on March 2013.

[21] https://code.google.com, retrieved on March 2013.

[22] http://realsearchgroup.com/rose, retrieved on March 2013.

[23] https://github.com, retrieved on March 2013.

[24] J. Wanner, “SourceMonitor” http://www.campwoodsw.com/sourcemonitor.html, retrieved on March 2013.

[25] http://commons.apache.org/proper/commons-cli, retrieved on March 2013.

[26] K. Beck and E. Gamma, “JUnit Testing Framework” http://junit.org/, retrieved on March 2013.

[27] M. H. Chen, M. R. Lyu, and W. E. Wong, "An empirical study of the correlation between code coverage and reliability estimation," In Proceedings of the 3rd International Software Metrics Symposium, 25-26 Mar. 1996, pp. 133-141.

[28] M.H. Chen and M.R. Lyu, "Effect of code coverage on software reliability measurement, "IEEE Transaction on Reliability, vol. 50, no. 2, Jun. 2001, pp.165-170.

[29] Y.K. Malaiya and R. Karcich, “The relationship between test coverage and reliability,” In Proceedings of Fifth International Symposium on Software Reliability Engineering , 1994, pp. 186–195.

[30] F.D. Rate, P. Garg, A.P. Mathur, and A. Pasquini, "On the correlation between code coverage and software reliability," proceedings of the sixth international symposium on software reliability, 24-27Oct. 1995, pp.124-132

[31] J. Lawrance, S. Clarke, M. Burnett, and G. Rothermel, “How Well Do Professional Developers Test with Code Coverage Visualizations? An Empirical Study,” In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*, Sep. 2005.

[32] J. A. Jones, M. J. Harrold, and J. Stasko, “Visualization of test information to assist fault localization,” in ICSE ’02: 24th International Conf. Software Engineering, 2002,

pp. 467–477.

[33] K. J. Rothermel, C. R. Cook, M. M. Burnett, J. Schonfeld, T. R. G. Green, and G. Rothermel, “WYSIWYT testing in the spreadsheet paradigm: An empirical evaluation,” In ICSE ’00: 22nd International Conf. Software Engineering, 2000, pp. 230–239.

[34] V. P. Araya, "Test Blueprint: An Effective Visual Support for Test Coverage," in ICSE '11 : 33rd International Conference on Software Engineering, 2011, pp. 1140-1142.

[35] M. Perscheid, D. Cassou, and R. Hirschfeld, “Test Quality Feedback: Improving Effectivity and Efficiency of Unit Testing,” in Proc. C5, 2012, pp. 60-67

[36] S. Asaf, E. Marcus, and A. Ziv, "Defining coverage views to improve functional coverage analysis,” Proceedings of the 41st annual Conference on Design automation**,** San Diego, CA, USA, 2004, pp. 41 – 44.

[37] T. Kanstr´en, “Towards a deeper understanding of test coverage,” J. Softw. Maint. Evol. , vol. 20, no. 1, 2008, pp. 59–76.

[38] H. Do, S. Elbaum, and G. Rothermel, “Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact,” Empirical Software Engineering: An International Journal, vol. 10, no. 4, 2005, pp. 405–435.

[39] S. Elbaum, D. Gable, and G. Rothermel, “The impact of software evolution on code coverage,” In Proceedings of International Conference on Software Maintenance, Nov. 2001, pp. 169-179.

[40] G.P. Vennebush, “BoxPlotter,” http://illuminations.nctm.org/ActivityDetail.aspx?ID=77 , retrieved on April 2013.

[41] F. Logozzo and M. Fähndrich, “On the relative completeness of bytecode analysis versus source code analysis,” Proceedings of the Joint European Conferences on

Theory and Practice of Software 17th international conference on Compiler construction, Budapest, Hungary, March 29-April 06, 2008.

[42] M. Gittens, K. Romanufa, D. Godwin, and J. Racicot, "All code coverage is not created equal: A case study in prioritized code coverage," CASCON 2006, pp. 131-145.

[43] H. Zhu, P.A.V. Hall, and J.H.R. May, "Software unit test coverage and adequacy," ACM Computing Surveys (CSUR), v.29 n.4, Dec. 1997, pp.366-427.