An Evaluation of Java Code Coverage Testing Tools

Elinda Kajo-Mece
Faculty of Information Technology
Polytechnic University of Tirana
ekajo@fti.edu.al

Megi Tartari
Faculty of Information Technology
Polytechnic University of Tirana
mtartari@fti.edu.al

ABSTRACT
Code coverage metric is considered as the most important metric used in analysis of software projects for testing. Code coverage analysis also helps in the testing process by finding areas of a program not exercised by a set of test cases, creating additional test cases to increase coverage, and determine the quantitative measure of the code, which is an indirect measure of quality. There are a large number of automated tools to find the coverage of test cases in Java. Choosing an appropriate tool for the application to be tested may be a complicated process. To make it ease we propose an approach for measuring characteristics of these testing tools in order to evaluate them systematically and to select the appropriate one.

Keywords
Code coverage metrics, testing tools, test case, test suite

1. INTRODUCTION
The levels of quality, maintainability, and stability of software can be improved and measured through the use of automated tools throughout the software development process. In software testing[5][6],software metrics enable the appropriate quantitative information, to support us in the decision-making on the most efficient and appropriate testing tools for our programs.

The most mentioned metric for assessment in the software field are the Code Coverage metrics. These metrics are considered as the most important metric, often used in the analysis of software projects for the testing process.

Today we have available several tools that perform this coverage analysis, but we will select the most appropriate tools, which are Java open-source code coverage tools like Emma and CodeCover.

To conclude with, according to some criteria, that we will take into consideration for the evaluation of this code coverage tools, we will judge for the most efficient tool to be used by the software testing team. These criteria are: Human-Interface Design (HID), Ease of Use (EU), Reporting Features (RF), Response Time (RT).

In Section 2 we will mention the coverage metrics [9] used in our experiments; we will shortly explain the tools [8] we have selected to perform the code coverage analysis for our tests; describe briefly how JUnit framework is implemented in each of these tools [10] [11], since JUnit is our experimental environment, where we program unit tests for our software and the last part of this section consists of selecting some criteria based on which we will then judge which of the tools is more effective to use in the testing process. In Section 3 we will summarize the results of our experiments for each tool and analyze them to bring us in the conclusion which of the tools is more effective. In Section 4 we give the conclusions of our work.

BCI'12, September 16–20, 2012, Novi Sad, Serbia.
Copyright © 2012 by the paper's authors. Copying permitted only for private and academic purposes. This volume is published and copyrighted by its editors.
Local Proceedings also appeared in ISBN 978-86-7031-200-5, Faculty of Sciences, University of Novi Sad.

2. SELECTED TOOLS AND EVALUATION CRITERIA
Among various automated testing tools [8], we have selected two tools to perform the Code Coverage Analysis [1][2][3], as a manner to evaluate the efficiency of our tests we created in the JUnit framework [4][7]. In this paragraph we will summarize briefly the main features of these to: EMMA and CodeCover coverage tools. The main reasons for which we choose them are:

1. These tools are 100 % open-source.

2. These tools have a large market share compared with the other open source coverage tools.

3. These have multiple report type format.

4. These tools are for both open-source and commercial development projects.

EMMA Tool

We used EclEmma 2.1.0, a plug-in for Eclipse, which is our Java development environment. Emma distinguishes itself from other tools by going after a unique feature combination: development while keeping individual developer's work fast and iterative. Such a tool is essential for detecting dead code and verifying which parts of an application are actually exercised by the test suite and interactive use. The main features of Emma, which represent its advantages are: Emma can instrument classes for coverage either offline (before they are loaded) or on the fly (using an instrumenting application class loader); Supported coverage types: class, method, line, basic block; Emma can detect when a single source code line is covered only partially; Output report types: plain text, HTML, XML.

CodeCover Tool

CodeCover is an extensible open source code coverage tool. It provides several ways to increase test quality. It shows the quality of test suite and helps to develop new test cases and rearrange test cases to save some of them. So we get a higher quality and a better test productivity. The main features of CodeCover are: Supports statement coverage, branch coverage, loop coverage and strict condition coverage; Performs source instrumentation for the most accurate coverage measurement; CLKI interface, for easy use from the command line; Ant interface, for easy integration into an existing build process; Correlation Matrix to find redundant test cases and optimize your test suite; The source code is highlighted according to the measured date.

The testing environment we used to project the set of tests for our input programs was JUnit 3.

We choose as input programs six sorting algorithms: Bubble Sort, Selection Sort, Insertion Sort, Heap Sort, Merge Sort, Quick Sort. The main reason why we choose these algorithms is the facility we face on computing the Cyclomatic Complexity (CC), which is crucial on defining the number of test cases needed to achieve a good coverage percentage of the program code. To proceed in the testing process for each of this sorting algorithm, we first build Java programs for each of them.

To achieve our goal we chose some criteria, based on which we will evaluate which testing tool is the most efficient. So we chose Human Interface Design (HID) as an indicator of the level of difficulty to learn the tool's procedures on purchase and the likelihood of errors, in using the tool over a long period of time; Ease of Use (EU) to judge if the tool is easy to use to ensure timely, adequate, and continual integration into the software development process; Reporting Features (RF) to show the degree of variety regarding the formats that tools use to report their coverage results; Response Time (RT) used to evaluate the tool's performance with regards to response time. In addition to these criteria, we will also evaluate the number and quality of test cases to judge for the most appropriate tool for the software testing process.

3. EXPERIMENTS AND ANALYSIS

In this section we will summarize the experiments we have performed on the selected algorithms. Initially, we built the Java programs for each of our sorting algorithms. Then we designed the set of testing units by using the JUnit testing framework [7] in Java. Finally we performed the Analysis of Code Coverage, to evaluate these tests through the selected code coverage tools. This analysis calculates the coverage percentage, that serves as an indirect measure of the quality of tests. Based on these measurements, we can then create additional test cases [4][7]to increase code coverage.

In table 1 we summarized the quantitative information regarding our experiments. In the last column we show the number of final test cases we built for each of the Java programs of the sorting algorithms. We used the term "final test cases" because we continuously improved our coverage results by increasing the number of test cases, until the addition of another test case does not anymore affect the coverage result, that means we have achieved a high level of code coverage.

Table 1: Experimental Program Details

Input Programs	LOC	NOC	NOM	CC	No.of TestCase
Bubble	53	2	3	4	11
Selection	55	2	3	4	11
Insertion	53	2	3	4	11
Heap	84	2	11	13	16
Merge	67	1	3	11	9
Quick	63	1	6	11	7

LOC-Lines of Code,NOM-Number of Methods,NOC-Number of Classes,CC-Cyclomatic Complexity

Based on these coverage results and also the computed criteria chosen for evaluation, we performed the analysis process to define the best tool.

In the figures below we see the coverage reports produced after the execution of Emma and CodeCover for two cases: 1) When we projected a small set of tests; 2) When we projected a larger set of tests in order to improve quality of the testing process. To show briefly the experimental procedure we followed to achieve our objective, we will take as an example the experimental results for Quick Sort algorithm. For Quick Sort we initially projected only 3 test cases (Fig.1). The CodeCover tool produced low BC (Branch Coverage) and LC (Loop Coverage) coverage metrics of 66.7 %. This result contradicts the result taken after the execution of Emma tool on the same set of test cases, which is relatively high with an average of 87 % (Fig.1). This contradict, led us to increase the number of test cases for a higher quality of tests. For Quick Sort we built 4 more test cases (Fig.2), which produced a maximum result of 100 % code coverage with both tools.

Session: TestSuite_QuickSort (Aug 22, 2011 4:44:01 PM)

Counter	Coverage	Covered	Missed	Total
Instructions	85.0 %	136	24	160
Basic Blocks	84.2 %	16	3	19
Lines	84.8 %	28	5	33
Methods	85.7 %	6	1	7
Types	100.0 %	1	0	1

Figure 1: Emma Coverage report initially with three test cases for QuickSort.

Session: TestSuite_QuickSort (Aug 13, 2011 4:44:11 PM)

Counter	Coverage	Covered	Missed	Total
Instructions	100.0 %	160	0	160
Basic Blocks	100.0 %	19	0	19
Lines	100.0 %	33	0	33
Methods	100.0 %	7	0	7
Types	100.0 %	1	0	1

Figure 2: CodeCoverage report finally with seven test cases for QuickSort.

Name	Statement	Branch	Loop	Term
QuickSort	87.0 %	66.7 %	66.7 %	80.0 %
QuickSort	100.0 %	–	–	–
afisho	100.0 %	–	66.7 %	100.0 %
quick_sort	100.0 %	100.0 %	–	100.0 %
quicksort_celes_rasti	0.0 %	0.0 %	–	0.0 %
rendit_celes	100.0 %	100.0 %	66.7 %	100.0 %
rendit_celes_rasti	100.0 %	–	–	–
shkembe	100.0 %	–	–	–

Figure 3: Code Coverage report after execution of CodeCover initially with three test cases for QuickSort.

QuickSort	100.0 %	100.0 %	100.0 %	100.0 %
QuickSort	100.0 %	–	–	–
afisho	100.0 %	–	100.0 %	100.0 %
quick_sort	100.0 %	100.0 %	–	100.0 %
quicksort_celes_rasti	100.0 %	100.0 %	–	100.0 %
rendit_celes	100.0 %	100.0 %	100.0 %	100.0 %
rendit_celes_rasti	100.0 %	–	–	–
shkembe	100.0 %	–	–	–

Figure 4: Code Coverage report after execution of CodeCover finally with seven test cases for QuickSort.

During our experiments, we noticed that this contradict, that relates to the fact that for the same set of test cases the execution of Emma gives us a higher coverage tool than the result reported from CodeCover, we concluded that CodeCover gives a more accurate information regarding the code coverage.

In Section 2, we mentioned the Correlation Matrix as a way to find redundant test cases, which does not increase the coverage percentage. It shows a kind of dependency relationship between test cases of the same input program. In JUnit3 testing framework, dependency between tests is not supported, that is why we should always try to avoid dependency between test cases. In the figure below is shown the Correlation Matrix for Quick Sort.

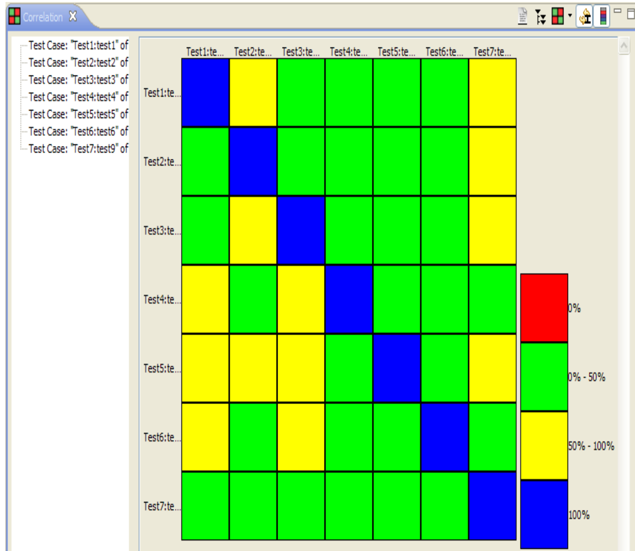
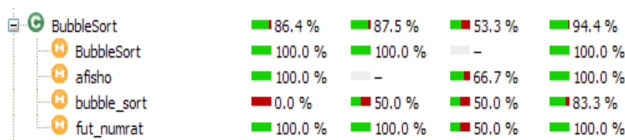


Figure 5: The Correlation Matrix produced by CodeCover for QuickSort with seven test cases.

From the figure above, we see that blue squares (meaning that there is 100 % dependency between test cases), exist only in the case where the same number of test case intersect. So we can say that we have proceeded according to the main rule of JUnit,that is to avoid dependency between test cases.

Below we will show by figures the results of the Code Coverage Analysis performed by Emma and CodeCover tools for the other five input sorting programs.

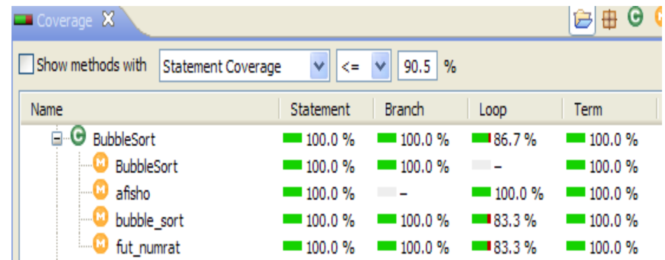
For Bubble, Selection and Insertion Sort we initially projected 7 test cases, then in order to achieve a relatively high coverage we projected 11 test cases. The coverage result report produced by CodeCover for BubbleSort is shown below for both cases.



BubbleSort	86.4 %	87.5 %	53.3 %	94.4 %
BubbleSort	100.0 %	100.0 %		100.0 %
afisho	100.0 %	–	66.7 %	100.0 %
bubble_sort	0.0 %	50.0 %	50.0 %	83.3 %
fut_numrat	100.0 %	100.0 %	50.0 %	100.0 %

Figure 6: Code Coverage report after execution of CodeCover initially with seven test case for BubbleSort.

From the figure above, we see a low percentage of 53.3 % for the LC (Loop Coverage) metric. That is why we finally projected 11 test cases to increase this low percentage as shown in the figure below, where the new LC metric is 86.7 %, which is considered a high coverage percentage. By improving our experimental work on the testing process repeatedly we came into the conclusion that to achieve a high coverage percentage the secret is to project one test case for each functional unit of the program, and to avoid programming long test cases that try to cover a considerable part of the program.



Name	Statement	Branch	Loop	Term
BubbleSort	100.0 %	100.0 %	86.7 %	100.0 %
BubbleSort	100.0 %	100.0 %	–	100.0 %
afisho	100.0 %	–	100.0 %	100.0 %
bubble_sort	100.0 %	100.0 %	83.3 %	100.0 %
fut_numrat	100.0 %	100.0 %	83.3 %	100.0 %

Figure 7: Code Coverage report after execution of CodeCover finally with eleven test cases for BubbleSort.

We haven't showed Emma coverage report, because it is relatively high since the first case, where we projected only 7 tests.

The results gained for SelectionSort are 46.7% for LC metric in the case of 7 tests and 80 % in the final case of 11 test cases; for Insertion are 60% for LC metric in the first case and 86.7% for the final case. So far, we see that in general the most "problematic" coverage metric is the Loop Coverage metric. This happens mainly because of the *for* loop, that requires more test cases to be covered. This is shown in fig.10, where yellow signifies the partial coverage of the *for* loop.

```
public void selection_sort(){
for(i=0;i<nr_termave-1;i++)
{
    min=i;
    for(j=i+1;j<nr_termave;j++)
        if(numrat[j]<numrat[min])
            min=j;
    temp=numrat[i];
    numrat[i]=numrat[min];
    numrat[min]=temp;
  }
}
```

Figure 8: A partial coverage of a *for* loop, crucial for the Lool Covrage metric (80 %).

For MergeSort we initially projected 4 test cases, which according to CodeCover produced a low LC indicator of 60 %,. Then we extended this set of test cases to 7test cases, gaining a new percentage of LC of 86.7 % (the reason why it is not 100 % is because there are many loops in the program, not only the for loops, but also while).

For Heap Sort we initially projected 8 test cases, giving a LC metric of 33.3 % and a CC metric (Condition Coverage) of 80 %.Then we improved this set of tests by extending it to 16 test cases, that improved considerably both the LC and CC metric to respectively : 88.9 % and 100 %.

Through the graph below we show the improvements we achieved in our experiments until we gained a high code by showing the initial result we gained when we projected a small set of test cases and the final result after we increased the number of test cases for a higher coverage.

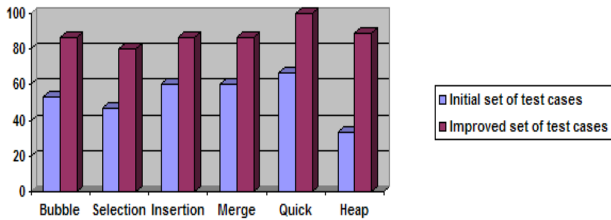


Figure 9: The percentage of improvement in code coverage achieved by increasing the number of test cases for the six sorting programs.

In table 2, we have summarized the results produced by Emma and CodeCover tools after performing the Code Coverage Analysis on each of the input programs (the sorting algorithms).

Table 2: Analysis & Implementation of Emma and CodeCover Using Various Sort Programs

Program	Emma				CodeCover			
	SC	BLC	MC	CLC	SC	BC	LC	CC
Bubble	100	100	100	100	100	100	86.7	100
Selection	88.4	85.7	93.9	100	100	90	80	95
Insertion	86	82.9	92.2	93.3	100	100	86.7	100
Merge	100	100	100	100	100	100	86.7	95.8
Quick	100	100	100	100	100	100	100	100
Heap	100	100	100	100	100	100	88.9	100

SC-Statement Coverage, BLC-Block Coverage, BC-Branch Coverage, LC-Loop Coverage, MC-Method Coverage, CC Condition Coverage, FC-File Coverage, CLC-Class Coverage

After analyzing the code coverage results produced after the execution of Emma and CodeCover on the various sorting programs, we concluded that CodeCover gives a more accurate coverage information than Emma. To complete the process of evaluating the effectiveness of these testing tools, we will show in table 3 the computed criteria [4] [5] selected to evaluate these tools.

Table 3: Analysis of Tool Metrics

Tools	HID	EU	RF	RT
Emma	14	82%	78	23min
CodeCover	13	86%	90	20min

Based on these values (which we partially gained in their official websites, as they are open-source tools), we judged that the best and more effective tool to be used during the software testing process is CodeCover.

4. CONCLUSIONS

Based on the results summarized in table 2, that shows achieved code coverage metric reported from each tool, we conclude that CodeCover tool reports a more accurate coverage information than Emma, which does not supply us with sufficient information, based on which we can judge over the quality of tests, that is why we suggest the use of the CodeCover tool. CodeCover is more efficient to perform the Code Coverage Analysis, because through the detailed coverage analysis for each program method, it allows us to define the unnecessary test cases, that does not increase coverage of the program, affecting so negatively the execution time of the test suite by decreasing it. We argued this conclusion by taking as an example QuickSort, where for an initial set of 3 test cases while Emma reported an average coverage of 87%, CodeCover reported a low Loop Coverage of 66.7 %.The same fact was present in all our set of input sorting programs. So in order to project a successful testing process for our input programs, we should base on CodeCover coverage reports, to decide whether it is necessary to increase the number of test cases or not. During our experimental work, where we continuously improved the testing process, we came into the conclusion that the most problematic coverage metric is Loop Coverage. This happens mainly because of the for loop, that requires extra tests to be fully covered. So our coverage results for all our input programs reached a Loop Coverage metric in the range 46.7 % to 66.7%, which is considered very low. But not only the Loop Coverage metric was responsible for low coverage percentages in the beginning of our work, but also the manner in which we projected our tests affects coverage result. So to achieve a high code coverage, we have to avoid programming long test cases that try to cover a considerable part of the program, but instead we must project one test case for each functional unit of the program. We arrive in the same conclusion if we see table 3, that shows the computed criteria chosen to completely evaluate the testing tools. From this table we infer that the CodeCover tool is easy to use, has a very good response time for every command given, has very good reporting features compared with Emma tool.

5. REFERENCES

[1] Lawrance, J., Clarke, S., Burnett, M., and G. Rothermel. 2005. 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* (September 2005).

[2] Tikir, M. M., and Hollingsworth, J. K. 2002. Efficient instrumentation for code coverage testing. In *Proceedings of the ACM SIGSOFT 2002 International Symposium on Software Testing and Analysis* (Rome, Italy, July 22-24, 2002).

[3] Cornett, S. 1996-2011. *Code Coverage Analysis*. Bullseye Testing Technology.

[4] Beust, C., and Suleiman, H. 2007. *Next Generation Java Testing: TestNg and Advanced Concepts*. Addison Wesley, 1-21, 132-150.

[5] Ammann, P., and Offutt, J. 2008. *Introduction to Software Testing*, Cembridge University Press, 268-277.

[6] Sommerville, I. 2007. *Software Engineering* (8th edition).Harlow: Addison Wesley, 537-565.

[7] JUnit Best Practices-Java World, http://www.javaworld.com/javaworld/jw-12-2000/jw-1221-junit.html

[8] Prasad, K.V.K.K. 2006. *Software testing tools*.

[9] Durrani, Q. 2005.Role of Software Metrics in Software Engineering and Requirements Analysis. In Proceeding of IEEE ICICT First International Conference of Information and Communication Technologies. (August 27-28).

[10] EMMA: a free Java code coverage tool http://Emma.sourceforge.net

[11] CodeCover Tutorial http://www.codecoveragetools.com/code_coverage_java.html