International Journal of Science and Research (IJSR)
ISSN (Online): 2319-7064
Index Copernicus Value (2013): 6.14 | Impact Factor (2013): 4.438

Improving the Reliability of the Automation Framework based on Code Coverage Analysis

Snehalata Giraddi[1], Merin Meleet[2]

[1]R. V. College of Engineering, VTU University, Information Science and Engineering Department, Vidyaniketan Post, Mysore Road, Bangalore-560059, India

[2]R. V. College of Engineering, VTU University, Information Science and Engineering Department, Vidyaniketan Post, Mysore Road, Bangalore-560059, India

Abstract: ***Software testing is one of the challenging tasks to select the test inputs. Code coverage is a testing methodology used to measure the quality of software testing. The objective of code coverage is to ensure the adequateness of testing by providing data on different code coverage items. Code coverage is a feedback mechanism for agile development or test-driven methodologies. Both of these methods depend on a developmental feedback that stimulates the addition of features while managing a predictable quality level. The proposed study suggests that the code coverage anticipations on the test cases effectiveness that professional software developers write. By having a look at the existing methods; uncovered measurement of code coverage and gaps can be explored further.***

Keywords: JaCoCo; Code coverage; SonarQube; Cobertura; Bytecode;

1. Introduction

Code coverage is a metric used to determine the completeness of software testing, by determining which areas of source code in an application were exercised during a test. It provides an efficient way to ensure that applications are not released with untested code. The developers consider the code coverage testing as an indicator of confidence level the software applications [1].The process of code coverage analysis they need to be automated. It will help the developers to get an idea regarding the given test suite throughout the software testing process [5]. Several code coverage process of testing tools are available to help researchers and end-users understand the software testing process [2]. One of such tool is JaCoCo, which is an open source toolkit which measures and gives the report on Java programming. This Code coverage report allows developers easily to get the code which part is not executed by the test suite.

JaCoCo gives coverage on line and branch. JaCoCo processes the bytecode while running the code, in contrast to Clover, it needs to be incremented the source code, and Cobertura, which processes the bytecode offline. To do this a Java agent can be configured for the storage process to collect data in a file and send it to analysis via TCP. The files can be merged easily from several runs or code parts. SonarQube Jacoco plugin is the platform for coverage analysis within the code quality management.

2. Levels of Code Coverage

There are number of ways in which code coverage can be measured. One of the commonly used methods is to measure one or a combination of one or more of the following: function coverage, statement coverage, condition coverage, branch coverage, and Modified Condition/Decision Coverage (MC/DC).

2.1 Function Coverage

Function coverage reports whether a function in a program has been called or not, it gives explanation about which part of program is executed inside it, and how or why the function is called. And it does not give indication regarding how many of the function calls that is made in a program.

Figure 1 shows that function( ) is called in a program and it shows that every function should call at least one time in a program.

```
a = function(data);
if(a==b)
{
result = 0;
a = function(b);
}

function(...)
{
.
.
.
}
```

Figure 1: Function Coverage

2.2 Statement Coverage

Statement coverage is one of the simplest forms of the code coverage. It measures the number of lines of code which have been executed during the execution of the program. This method does not consider conditional statements or consideration loops, considers only the statements written within an executable line [5]. In many of the programming languages, typically, a semicolon will terminate a statement. In some cases, multiple lines will be there in a single statement. The result of what is and is not executed in the program is given by statement coverage, and it has some limitations.

2.2.1 Statement Coverage Limitations
Consider the following code fragment

```
int x = null;
If(cond)
        x = &var;
x= 123
```

Figure 2: Statement Coverage

If 'cond' condition is true, it is possible to reach 100% statement coverage. This test case fails the scenario when this 'cond' is false. The program will de-reference a null pointer in such scenarios. The entry level of code coverage is Statement coverage and it is also a good practice. Usually the false condition is also tested.

2.3 Branch Coverage

Branch Coverage measures whether branch points and decision are tested completely for all possible outcomes. For example an 'if' statement must take on both "true" and "false" outcomes to be considered covered [5]. The coverage is treated as partial if only one of the paths is taken. Same as Statement Coverage, there are some nuances that need to be taken care of, lazy evaluation will occur when we work with various languages. The technique of delay in computation of parts of code till those are called is known as lazy technique.

2.3.1Branch Coverage Limitations
There is a situation in which 'lazy evaluation' can occur is with complex Boolean expressions. The example code is shown in below:

```
int x = null;
If(cond1 && (cond2||function(x))
        statement;
else
```

Figure 3: Branch Coverage

Consider the cases in which 'cond1' is false. Lazy evaluation is not needed to evaluate 'cond2' or ' function(x)'. The false coverage will result for 'if (cond1 && (cond2 || function(x)))'.

Consider the cases in which 'cond1' and 'cond2' are both true. Again, it will result in lazy evaluation in 'function(x)' has not been evaluated. For the above condition this will also result in true coverage path. In these scenarios, it is possible to have 99% Branch coverage but still it has defects in the software framework.

2.4 Modified Condition / Decision Coverage (MC/DC)

MC/DC is a sort of "super branch coverage" and is a very advanced type of code coverage analysis. It gives report of true and false of a complex condition as is done in branch coverage as shown in figure 2, but it will also give report on true and false of the sub-condition in a complex condition. It reports the issue given by lazy evaluation, by requiring a manifestation that each sub condition may affect the result of the decision that is independent of the other sub condition results.

Considering the example in Branch Coverage Limitations as shown in the figure 2**,** it has to verify 'cond1' for true and false values while holding 'cond2' and 'function(x)' fixed, then it has to do the same, for 'cond2' and when holding 'cond1' and 'function(x)' fixed. Ultimately it will verify the same for 'function(x)', when holding 'cond1' and 'cond2' fixed. The evaluation of every sub-condition for the values of 'true' and 'false' when holding the other sub-conditions fixed is known as a modified condition pair.

3. Coverage from Different Types of Testing

Software testing comes in a variety of way:
1. **System / Functional Testing:** Testing the whole integrated software application
2. **Integration Testing:** Testing integrated software sub systems
3. **Unit Testing:** Testing a few individual files and classes in an application

Every project does some amount of system testing where the source code is stimulated with some of the same actions that the end users will do. One of the frequent causes of applications being fielded with the bugs, and therefore the application in the field experience untested, combinations of inputs.

Most of the projects do integration testing while some of them do unit testing. If it had done unit testing or integration testing [3]. A group of files and single file from the rest of the applications are isolated by the amount of test code that has to be created.

At the most rigorous levels of integration test or unit test, it is common for the amount of application code being tested to be smaller than the amount of test codes written. As a result, the levels of testing are involved to business and safety critical applications in standardized markets, Those are: medical devices, aviations, railway services, process control, and soon automotive. Several applications are written for these industries contain automation software. The structural testing process for regulated industries often revolves around testing the high and low-level requirements and analyzing the code coverage those results from this "requirements based" testing [3].

On many projects, high-level or functional requirements are tested first. Code Coverage can be used to capture and report on the amount coverage achieved. Regrettably, during functional or system testing it is not possible to get 100% code coverage. But, we can achieve 55%-65% code coverage during this type of testing. Using integration testing or unit testing techniques the remaining 35-45% code coverage will be achieved.

Unit testing comprises using test code in the fashion of stubs and drivers to confine specific functions in the application, and stimulating these functions with the test case. The low level requirement based test gives you much greater control over the code being tested and is used to expand the previously executed system tests and allow you to get to 95% coverage. For this reason, it is acceptable to be able to share coverage data from different methods of testing.

4. Challenges of Code Coverage in an Automation Framework

In the case of code coverage, the price to be paid is the addition of instrumentation to the source files to be tested. Instrumentation is the additional source code added to an application to allow the collection of coverage data as tests are executed. The overhead associated with instrumentation translates directly into increased source file and program size, and indirectly into increased execution time.

It is not possible however, to forecast the precise impact that instrumentation would have on a particular set of application files. No algorithm exists for this purpose and none is possible. Too many variables are involved and every application is unique in its complexities. It is possible however, to derive a set of estimates from a representative example.

5. Conclusion

During the course of this paper, we discussed the advantages and needs of test automation. This work established the impact of quality of test automation on the quality of production code. It provides scope to add tests, remove redundant tests, improve tests and cover more code, thereby ensuring higher quality and more reliable systems.

References

[1] D. Nageswara Rao, Dr. M. V. Srinath, P. Hiranmani bala, "Reliable Code Coverage Technique in Software Testing", Proceedings of International Conference on Pattern Recognition, Informatics and Mobile Engineering, Feb. 2013
[2] R.Beena, Dr.S.Sarala, "Code Coverage Based Test Case Selection and Prioritization", International Journal of Software Engineering & Applications (IJSEA), Vol.4, No.6, Nov. 2013
[3] Presitha Arathi M, Nandini V, "A Survey on Test Case Selection and Prioritization", Internation journal of Advanced Research in Computer Science and Software Engineering, Volume 5, Issue 1, Jan. 2015
[4] Kazunori Sakamoto, Hironori Washizaki, Yoshiaki Fukazawa, "Open Code Coverage Framework: A Consistent and Flexible Framework for Measuring Test Coverage Supporting Multiple Programming Languages", International Conference on Quality Software, 2010
[5] David Landoll, Michelle Lange, White Paper on "Code Coverage Explained"

Author Profile

Snehalata Giraddi is persuing the M.Tech. degree in Information Technology from R. V. College of Engineering, Bangalore.

Prof. Merin Meleet is working as Assistant professor in R. V. College of Engineering, Bangalore in the department of Information Science and Engineering.