Code coverage, also known as test coverage, is a generic term which many developers use in misguided form. Unless one specifies the methods that were used for coverage, the term gives very little information about the quality of the coverage. Even then, since no pure coverage tools can test for logical correctness, code coverage gives only a partial characterization of a code's quality.

Code coverage, however, helps a programmer to avoid common pitfalls that arise in software development. Code coverage makes programmers more aware of their code and suggests patterns which are safer than a misguided way of development. Various authors, including Buecher; Shahid& Ibrahim; Maverik; Kessis, Ledru, and Vandome; Kim; and Gill and Runciman, have written about coverage methods and tools, common pitfalls in applying coverage, and coverage methods for remedying some of these shortcomings.

In (2012), Buechner defines various code coverage measures and their relationship to one another, including their strengths and weaknesses. Buechner defines code coverage as the measure of number of distinct program points or paths that a program executes when running test cases and their types. Coverage is expressed as the ratio of number of items executed at least once to the total numbers of items in the target code. These items can be statements, conditions, decisions branches or paths within the source code. Coverage measures also include more advanced measures of coverage, which are mostly based on combinations of these more basic program constructs.

Statement coverage measures the degree to which tests exercise statements. This measure, as Buechner notes, depends on how statements are defined. Some coverage tools interpret statements relative to a program's source code, while others translate complex statements into multiple, simpler statements. Another concern with statement coverage is its failure to include implicit code blocks, such as missing "else" blocks for “if" clauses that lack them. Buechner considers statement coverage a weak form of coverage and asserts that it does not expose all of code’s bugs.

Branch coverage requires that each branch (if clause, loops) be executed at least once during testing. Beuchner notes that branch coverage accounts for unstated else branches for “if" clauses by requiring test cases that set the condition to true and false. Since 100% branch coverage implies 100% statement coverage, branch coverage is stronger than statement coverage.

Decision coverage requires that every entry and exit point, control branch, and boolean expression be executed at least once during testing. Decision coverage implies branch coverage.

Condition coverage measures the extent of a condition’s effect on a decision’s outcome. It accounts for short-circuit evaluation: i.e., it evaluates boolean expressions from left to right and halts the evaluation if subsequent expressions have no effect on the condition’s value. The extent of a test suite's condition coverage can be characterized in five ways. Simple condition coverage is achieved if all conditions evaluate to true or false at least once; this is also called branch condition testing. Full condition / decision coverage involves 100% simple condition coverage and 100% branch coverage. Multiple condition coverage requires all atomic conditions, all compound conditions and whole decision to evaluate to both true and false. Modified condition / decision coverage (MC/DC) requires every entry and exit point to execute at least once, every condition and decision to take all possible outcomes and each condition to independently affect the decision’s outcome regardless of values of other possible conditions. Finally, multiple condition coverage (MCC) tests all possible combinations of true and false for all of a decision's conditions. MCC is measured by the completeness of truth table for these conditions.

Beuchner notes that tests are affected by how statements are ordered and structured. For example if test objects contain several conditions and decisions, condition coverage might be more appropriate for testing. Condition coverage, however, increases the number of test cases required for a particular condition. When condition coverage is used for testing, programmers should be aware of how they implement decisions. Since MCC tests all possible combinations of all conditions, MCC is less frequently used than MC/DC.

Path coverage tests the number of possible paths an expression takes during the test cases’ execution. As the numbers of a program’s branches grow, the number of possible paths for an expression increases. For complex logic it might be impractical or even impossible to achieve 100% path coverage.

A final form of coverage, boundary interior coverage, classifies paths using loops based on how often they execute.

Code coverage can be an effective method for investigating the degree to which test cases exercise program execution. Code coverage is also used to measure software quality. Code coverage, however, cannot directly measure the quality of logic. Complete coverage, in theory, can be achieved with enough test cases and randomly generated data. A better approach to testing would be to use functional point test cases derived from the specification, and then to apply code coverage only after all functional tests have passed. Coverage methods should, moreover, be chosen based on a code's criticality, complexity and frequency of usage, and the effort expended to achieve coverage balanced against other quality testing means like static analysis and code reviews.

In (2011), Shahid& Ibrahim evaluate the language support, instrumentation, GUI and reporting features of 19 test coverage tools. Shahid& Ibrahim assert that software testing identifies and improves software quality. Coverage measurement is one criterion for tools selection.

Shahid& Ibrahim divide test-based code analysis into three tasks: code instrumentation, data gathering and coverage analysis. Code instrumentation tools do runtime analysis or component testing, typically by adding code for tracking program execution to code files. Source level analyzers like Jester, Coberatura and JFeature do the analysis in a separate pre-processing phase. Byte code analyzers like JavaCodeCoverage, InsECT and Gretel do runtime instrumentation. The authors identified JFeature as a unique tool because it helped track requirements by focusing on standard development practices.

Different tools tested by Shahid& Ibrahim provided different levels of coverage. Testwell CTC++ supports modified condition/decision coverage (MC/DC). BullseyeCoverage supports C and C++ and runs on a wide range of platforms. NCover, an open source code coverage tool, supports the .NET platform. TestCocoon, a C/C++ and C# code coverage tool, analyzes the performance of software validation and reduces the number of tests by finding redundant and dead code.

Other criteria besides code coverage are often used to select test tools. Shahid& Ibrahim identified Graphical User Interface (GUI) and reporting as important features for tool selection. Tools with GUI support include JavaCodeCoverage, JFeature, Bullseye, Clover and Cobertura. Commercially developed tools generally produce automatic summary reports. Shahid& Ibrahim describe Clover as one of the best tools for report generation. Its HTML reports show red and green bars to demonstrate coverage percentages in addition to numbers. Clover also supports XML and PDF reports. It generates historical reports to show detailed overview of coverage. Emma and Bullseye support HTML outputs. Additionally, Emma and Bullseye only support XML and CSV report generation, respectively.

In (1997), Maverik describes common misuses of code coverage tools. Maverik's main concern is the tendency to misinterpret data that tools generate by failing to account for tools’ limitations. He focuses specifically on branch coverage, noting that coverage rates don't account for incomplete statements, including missing cases in "if" statements and missing catch statements for exceptions. To find such errors, codes that select from among alternatives should be analyzed to determine whether they adequately address all alternatives.

Maverik is particularly concerned with managers that measure performance solely based on coverage conditions. If, for example, a manager uses 85% coverage as a standard for adequate coverage, testers can then use easier coverage conditions to get 85% coverage. A survey found that programmers in organizations that require 85% coverage for shipping code typically failed to try for 90% coverage. Maverik argues for the use of incentives to motivate people to find weaknesses in their designs.

Maverik notes that products testers can also misuse code coverage. He notes that coverage tests, particularly black box tests, fail to identify errors involving sequences of events: e.g., a failure to execute code A after exercising a subsystem B.

Safety-critical and commonly used codes should be tested thoroughly. After fixing high risk code, medium-risk errors should be fixed as soon as they are found and as cheaply as possible. Low risk errors should be treated as annoyances: the cost of fixing these should be balanced against making time to work on other features.

In (2005), Kessis, Ledru, &Vandome provide an empirical study of code coverage analysis of a Java middleware application. Kessis et al. characterize their analysis as one of very few published empirical studies of code-coverage-based testing. Their research, which involved tests of an open source Java web server (JOnAS), was related to two studies on large scale applications at IBM: one by Kim (2003) on fault distribution and one by Asaf, Marcus, &Zivon on a new coverage method (2004).

The JOnAS web server is an open source J2EE application that hosts several enterprise applications. JOnAS also supports non functional services like security, transaction, EJB, naming, web services and communications. JOnAS uses many external components to help with servicing requests:e.g. Tomcat as its web container, Axis as its web services implementation and JORAM for a Java messaging service (JMS). These external components raise JOnAS's total lines of code to more than 1,000,000.The authors chose to test JOnAS's 200,000 lines of core code, including JOnAS's Enterprise Java Beans (EJB) container, client container, Java Management Extensions (JMX), web services and naming service.

Kessis et al tested different combinations of JOnAS-supported services: i.e., various combinations of remote method protocols (JRRMP and RMI-IIOP), databases (Oracle, PostGreSQL, MySQL, HSQLDB) and operating systems (Windows, Linux). Jonas was tested using the Jonas Test Suite (JTS), a white box testing tool developed by the JOnAS development team. JTS supports functional, structural and performance tests and provides some support for integration tests. Since JTS supports almost 43,000 tests, it was deemed suitable for testing middleware applications.

To address the difficulties created by the server's complexity and size, the authors used the Clover coverage analyzer to generate tests. Clover, which uses ant for coverage analysis, was easily integrated into JOnAS. Clover computes the Total Percent Coverage (TPC) by dividing the sum of tested conditions and statements by two, then multiplying this by the total number of conditions, statements and methods.

Tests were conducted by using an automated method to instrument JOnAS's source code, then testing the binary version generated from compilation. The results showed a TPC of 32.4 % with 32.4 % statement coverage, 31.7 % conditions coverage and 33.7% methods coverage. The coverage distribution showed two services (Web and Web services) at more than 50% coverage and three others (JDBC, Security and Naming) at 20%. The client container showed only 10% coverage. The authors also found that coverage was unbalanced inside the packages as well as between them. These imbalances included packages that were never tested: mostly, administration tools and EJB development assistance tools. Other instances of limited coverage included some JMX and debugging code.

Based on these results, the authors concluded that attaining100% coverage was nearly impossible in a large application. Their work exposed the limitations of Java based tools for what the authors noted was a quite mature platform. It showed that JOnAS's developers failed to prioritize writing test cases for debugging, exception handling and monitoring codes.70% of the defects that the tests exposed were localized in only 20% of modules.

The authors concluded that their results should be helpful for improving the JOnAS server. One such improvement would involve removing the dead code that the tests failed to cover and refactoring the JOnAS server to make it lean and memory efficient.

In (2007), Gill &Runciman describe their Haskell Program Coverage (HPC) application, comparing it to tools for imperative languages like Java and C. Imperative languages treat computations as processes that vary the state of their underlying variables, based on normal-order program evaluation. Coverage tools for such languages may treat exported modules as atomic although a program's state might have changed multiple times during the modules’ evaluation. Haskell, by contrast, implements single-assignment semantics and uses lazy evaluation. A sub-expression in Haskell that is not needed to produce output might never be evaluated even if its surrounding code is called many times. Haskell's natural units of evaluation are expressions and functions, rather than the commands and procedures that are natural units for imperative languages. These features, according to Gill &Runciman, made the design of HPC unique and challenging.

Gill &Runciman assert that HPC met many of its design goals. They note that HPC is scalable, easy to use, granular, and portable; has language extension support, and accumulates coverage over multiple executions; has open standard documents; has low running cost; generates concise results, is non-annotated, supports entry counts and can be run selectively. HPC's integration into the Glasgow Haskell Compiler (GHC)'s runtime environment allows HPC to support several Haskell-based languages.HPC detects any expression, without any coverage overhead in the output. For large applications HPC records declarations and source code modules, allowing HPC to cover small- and large-scale components of programs alike. The authors used source-to-source translation to make HPC portable between Haskell implementations. Since HPC uses open documents for storing coverage information, other developers can easily integrate them into new tools. HPC runs without source code annotations, avoiding overhead required by other coverage tools. Finally, developers can use a domain specific language to write a manual tix file that selects modules for coverage.

HPC uses numerically indexed arrays of counters called tick boxes to record coverage. HPC inserts tick functions into source code to evaluate expressions to cover, associating a generated source file with each tick box. Tick boxes allow HPC to handle concurrent threads, exceptions, foreign function call and unsafe IO. Tick boxes are thread-safe. Since tick functions are incorporated into expressions before their evaluation by the integrated HPC, exceptions propagate as usual and have no additional side effects.

HPC provides various levels of source coverage (declarations, branches and expressions) and boolean-control coverage. It displays coverage information using textual reports and color markup. The textual reports summarize coverage for all modules. The markup report shows only the unticked boxes and displays the highlights of source code in HTML file using different colors to show coverage. The authors used an example to demonstrate the report generation feature of HPC.

The authors have used HPC to instrument a ray tracing program for the ICFP 2000 programming contest. They have also used HPC to assess coverage for a lazily-streamed version of a marching cubes graphic in a table driven visual application. For source-to-source translation the authors reused the parse and pretty printer from the NHC98 compiler and the HAT tracing system. They created a rewriting pass that includes the instrumentation code that records the locations that execute and adds ticks. For GHC specific implementation they incorporated the source-to-source translator into GHC and revised the Haskell abstract syntax tree to accommodate this change.

Gill &Runciman classify compile-time, run-time and post-processing as the costs for using HPC. Although HPC-generated tick box files have no effect in the program’s execution code, they significantly increase compilation time.

Compiling with the GHC ~02 and ~fhpc options for source-to-source instrumentation yielded worst-case slowdowns of around 3x and 5x, respectively. The worst case times for unoptimized compilation and source-to-source instrumentation increased by a factor of 6.

The authors note that several opportunities for optimization are lost because ticks force the output to keep the source of the original Haskell code. The cost of ticks accounted for approximately 30% overhead for coverage gathering. However, integrating the instrumentation pass into GHC reduced compilation overhead to less than 2. This overhead (compile time and runtime) was deemed to be acceptable for practical applications, because HPC provides programmers with valuable information about their code.

Gill &Runciman view HPC as a work in progress. They state that their tests are preliminary and want others to use and test HPC for large applications.HPC is limited in a number of key ways. HPC transforms modules independently using pure syntactic rules, independently of type information, without changing type names or declarations. While this allows modules to be instrumented selectively, HPC doesn't do the transformation based on operator priorities. This lack of operator priority prevents extension of boolean coverage. Other limitations include the use of booleans to represent ticks, which improves performance at the cost of not obtaining usage counts, and HPC's incompatibility with GHCI, the GHC interpreter. GHCI permits the users to reload modules. Since HPC consists of pre-determined list of tick-boxes corresponding to entry points it does not work inside GHCI.

Code coverage is very important for any kind of software development. Software development teams should consider writing test cases very early on in their coding. I would even suggest teams to use test driven development (TDD) for writing software of considerable size and complexity. TDD encourages the development team to write code that is highly cohesive in its implementation and loosely coupled through interfaces. To support effective testing teams should write tight and cleaner code that has very little external dependencies. When developer teams use coverage tools to execute these test cases, it should provide them with the result that is true to the nature of the project.

I believe the capstone project will greatly benefit by using code coverage methods and tools. The capstone project’s team members meet each week only for a limited number of hours. Therefore they should use tools that display results that everyone can understand: e.g. graphics and statistics. Most popular coverage tools output their results using html or texts that are easily evaluated by even by members who are not developers. Capstones also have stakeholders who are non technical people. With a little training they will be able to assess the team’s performance even when there is no front-end visual output during the development.

Another theme that often arises in capstone projects is the need for maintenance. Software maintenance includes immediate bug fix and feature enhancement. The students who will be working on the project will leave the college after their graduation. Since the original developers will mostly likely not maintain the code, the need for less buggy code is apparent. Several tools such as requirements specification, UML diagrams, and project schedules provide guidelines for developers but code coverage is closest to the developer, in that enforces a certain coding style. In that regard if the developers use code coverage properly, i.e. writing good test cases, it becomes one of the best tools in their arsenal.

Some modern authorities treat test cases as a program's primary form of documentation. Test cases capture different functionalities in their individual units, therefore the codes are easier to read in isolation. This should be greatly helpful if new members work on feature enhancement of the capstone project. Test cases should help new developers to identify sections of the source code they must focus on to add new functionalities. Therefore, using code coverage tools with effective test cases should allow later developers to sustain the project's development even after the original developers leave the project. I believe this paper will provide that insight to the developers of the capstone project.

References

Asaf, S., Marcus, E., & Ziv, A. (2004). *Efficient use of code coverage in large-scale software development* (pp. 41-44)*.* Retrieved 10 11, 2014, from ACM Digital Library: http://dl.acm.org/citation.cfm?id=996579

Buechner, F. (2012). *Is 100% Code Coverage Enough?* (pp. 1-34). Retrieved 09 19, 2014, from Hitex Development Tools: http://www.hitex.com/fileadmin/pdf/products/tessy/white-papers/WP-TESSY-0104-CC\_e.pdf

Gill, A., & Runciman, C. (2007). *Haskell Program Coverage*. *Proceedings of the ACM SIGPLAN Workshop on Haskell Workshop* (pp. 1-12). New York, NY, USA: ACM.: Retrieved 10 17, 2014, from ACM Digital Library: http://doi.acm.org/10.1145/1291201.1291203

*JOnAS*. (OW2 Consortium). (2012). Retrieved 10 11, 2014, from OWN2 JOnAS: http://jonas.ow2.org/xwiki/bin/view/Main/

Kessis, M., Ledru, Y., & Vandome, G. (2005). *Experiences in coverage testing of a Java middleware* (pp. 39-45)*.* Retrieved 10 11, 2014, from ACM Digital Library: http://dl.acm.org/citation.cfm?doid=1108473.1108483

Kim, Y. W. (2003). *Efficient Use of Code Coverage in Large-Scale Software Development* (pp. 145-155)*.* Retrieved 10 11, 2014, from ACM Digital Library: http://dl.acm.org/citation.cfm?id=961347

Maverik, B. (1997). *How to Misuse Code Coverage* (pp. 1-13). Retrieved 10 4, 2014, from Exampler Consulting: http://www.exampler.com/testing-com/writings/coverage.pdf

Shahid M., & Ibrahim S. (2011). *An Evaluation of Test Coverage Tools in Software Testing.Proceedings of the CSIT vol.5*(pp. 216-222).Retrieved 09 26, 2014, from ResearchGate: http://www.researchgate.net/publication/228922323\_An\_Evaluation\_of\_Test\_Coverage\_Tools\_in\_Software\_Testing/file/60b7d52859b2442027.pdf