DEGREE PROJECT IN COMPUTER SCIENCE AND ENGINEERING, SECOND CYCLE, 30 CREDITS

STOCKHOLM, SWEDEN 2017

Analysis of test coverage metrics in a business critical setup

SHASHANK MISHRA

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF COMPUTER SCIENCE AND COMMUNICATION

Analysis of test coverage metrics in a business critical setup

SHASHANK MISHRA
shamis@kth.se

DD221X, Degree project in Computer Science (30 ECTS credits)
Master program in Computer Science (120 ECTS credits)
KTH Royal Institute of Technology, Year 2017
Supervisor at CSC: Karl Meinke
Examiner at CSC: Cristian M Bogdan
Master thesis work carried out at NASDAQ Technology AB

Royal Institute of Technology
School of Computer Science and Communication
KTH CSC
SE-100 44 Stockholm, Sweden
URL: www.kth.se

Abstract

Test coverage is an important parameter of analyzing how well the product is being tested in any domain within the IT industry. Unit testing is one of the important processes that have gained even more popularity with the rise in Test driven development (TDD) culture.
This degree project, conducted at NASDAQ Technology AB, analyzes the existing unit tests in one of the products, and compares various coverage models in terms of quality. Further, the study examines the factors that affect code coverage, presents the best practices for unit testing, and a proven test process used in a real world project.
To conclude, recommendations are given to NASDAQ based on the findings of this study and industry standards.

Sammanfattning

Testtäckning är en viktig parameter för att analysera hur väl en produkt är testad inom alla domäner i IT-industrin. Enhetstestning är en av de viktiga processerna som har ökat sin popularitet med testdriven utveckling.
Detta examensarbete, utfört på NASDAQ Technology AB, analyserar de befintliga testen i en av produkterna, och jämför olika kvalitetsmodeller. Vidare undersöker undersökningen de faktorer som påverkar koddekning, presenterar de bästa metoderna för enhetstestning och en beprövad testprocess som används i ett verkligt världsprojekt.
Avslutningsvis ges rekommendationer till NASDAQ baserat på resultaten från denna studie och industristandarder.

Keywords

Code coverage, line coverage, branch coverage, path coverage, mutation testing, unit testing, test process, complexity

Preface

This degree project was the final requirement of my Master in Computer Science degree at KTH, and I would like to thank my team at NASDAQ that has helped and supported me throughout the project. My experience working on this project was wonderful and intriguing.

Special thanks to:

Kjell Paulson, NASDAQ, for his support, engagement, and invaluable advice.

Karl Meinke, KTH, for his support and taking me on as a thesis student.

Cristian M Bogdan, KTH, for his guidance and time to evaluate my work.

Ann Bengtsson, KTH, for ensuring that the thesis complied with all administrative aspects.

Bill Bonnici, NASDAQ, for providing me insight with the test process.

Finally, I would like to thank **my parents**, for their endless love, support and encouragement.

Contents

Chapter 1: Introduction

Test coverage is of value to all business and products, big or small. The focus of this study was a product in the fin-tech domain. The degree project was carried out at NASDAQ Technology AB; an American fin-tech company with a big market cap. NASDAQ [15] is the largest electronic equities exchange in the U.S. and lists stocks of over 3,100 companies. It also offers trading in derivatives, debt, commodities, structured products and ETFs.
The software used for trading on the stock exchange and in the clearing houses need to be glitch free. Even minor mistakes in calculations in such an environment can lead to huge losses in the markets, and consequences that can cause panic. Thus, testing all functionalities, and covering all aspects of the user requirements are of utmost importance in such a business critical setup. The objective of this degree project is to examine the existing unit tests in the product, trace issues, and suggest improvements to them. Further, suggest which coverage model is best suited for unit testing in such a set up, and suggest improvements to the test process in the project.

Problem

The software product to be analyzed is one of the widely used products from NASDAQ in the fin-tech industry all over the world. Thus, it is crucial for the company to have a stellar product with minimum bugs in production or delivery. To ensure this, it should be tested thoroughly. Thus, unit testing was identified as an area to be examined. The task was to do an analysis of the existing unit tests, and find problems or deviations, from the best practices of unit testing. Further, some unit test coverage metrics were to be compared in terms of quality, and complexity to establish the best coverage metric for unit testing in a business critical and financial setup.

Therefore, three questions had to be examined. The first question was specific to the business critical setup of this project, while the other two questions were applicable to software testing in general.

The research questions were as under:
RQ 1: *Which test coverage metric is most suited for unit testing in a business critical setup?*
RQ 2: *What factors affect code coverage?*
RQ 3: *What kind of unit testing practices and test process can help in improving the code coverage level and quality of a software product?*

Thesis objective

The degree project aims to investigate how the current unit tests perform on various coverage scales such as line, branch, and path coverage, find gaps and suggest improvements, thereby analyzing and improving the overall testing process, and test suites used for the particular product.

Delimitations

The analysis and comparison of the test suites and test process is limited by the structure of the legacy code relevant to the product in test, and this is determined in consultation with NASDAQ.
The coverage approaches possible could be structural or requirement based. It was established that a structural approach would be preferred over a requirement based approach. However, performing a thorough structural coverage analysis of the whole system and realistically producing the desired results was a task that would take more than the desired time for this project. Thus, the set-up limitations, and discussions with the project management, helped narrow the focus to a critical part of the system that had good unit test coverage.
It is assumed that the restrictions and limitations laid upon us are realistic and representative of the actual limitations of the testing team for the product under test.

Choice of methodology

This thesis work is based on a study using an empirical and experimental approach to achieve validity. At the first stage of the project, a detailed analysis is performed to be able to establish the scope of improvement in the test process for a part of the product under test. The results are then compared to establish the quality of the suites with respect to different coverage criteria. The resulting conclusions and recommendations are based on the analysis done.

Declaration

This thesis work was carried out along with Hikari Watanabe (hikari@kth.se), another master student at KTH, and thesis intern at NASDAQ. The primary focus of the research was different for both (test coverage for this study and mutation testing for the other). The common parts in the thesis work include the data set, the tools used, and the comparison drawn between coverage and quality.

Chapter 2: Background

This chapter lays the foundation of the theory necessary to understand the problem and the intended approach.

Financial markets

A financial market is where people trade financial instruments or assets such as stocks, bonds, securities, commodities, and other items of value at low transaction costs. Financial markets strengthen the economy by making finance available at the right place. [1]
The main functions of a financial market include mobilization of savings and their channelization to more productive users, facilitate price discovery to provide liquidity for financial assets, and reduce transaction costs.
Based on what asset or instrument is being traded, the financial markets are divided into several types. These include Capital markets for stocks, bonds, equities, etc.; Commodity markets for electricity, gas, oil, etc., and so on.

Trade lifecycle

The lifecycle of a trade refers to the sequence of events that occurs and processes that are implemented when a trade takes place. The key stages of the trade lifecycle, namely pre trade, trade execution, trade clearing (the set up of this project), trade settlement, and ongoing position & risk management are shown in figure 2.1:

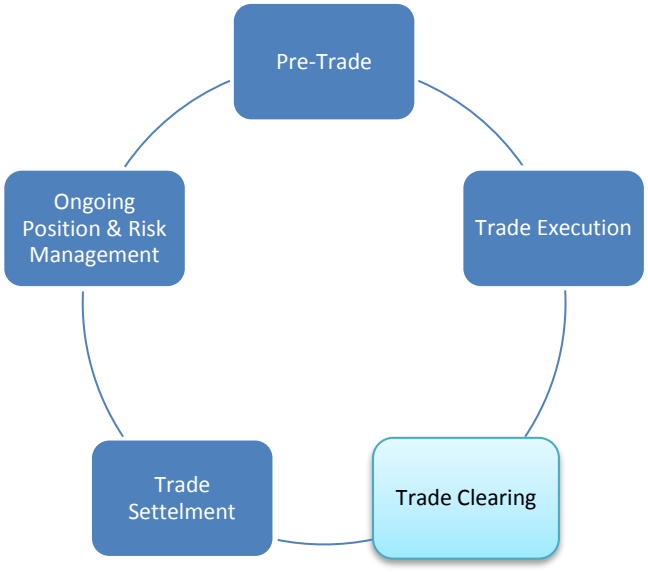

Figure 2.1: The trade lifecycle

Clearing house and CCP

Clearing a trade typically takes place through a clearing system, which implements clearing process at a single entity known as clearing house. Thus, a clearinghouse [2] [3] represents a central processing mechanism through which the exchange of securities and funds is facilitated using the rules set by the clearinghouse.
It is important to distinguish between a clearinghouse and a central counterparty clearinghouse, known as a CCP [2] [3]. The key difference between the two is that a CCP assumes counterparty risk while a clearing house does not. A CCP acts as a middleman between a buyer and seller of financial instruments.

Software testing

Software testing [2] [4] is undertaken to ensure that a process does what it is supposed to do. Broadly speaking, testing is divided in two categories, namely verification, and validation. While verification is the process of ensuring that the product is being built according to the requirements and design specifications; validation makes sure that the product meets user needs i.e. it fulfills intended use in intended environment.

Software testing is important because it creates a safe and robust process, thereby reducing the cost of product maintenance. Testing in a planned manner reduces uncertainty and ensures that problems are fixed in a timely manner while all the resources are still available. Organizations generally devote time to testing in order to enhance performance and reduce operational risks.

There are several distinct types of testing at various stages in the software development process.
We define a few, relevant to this study as [2] [4] [7]:

Unit testing: Testing single component in the system, in isolation from the rest of the system, usually done by the developer of the unit.

Black box testing: System is treated as a black box, meaning that output is validated for the input that is fed into the system.

White box testing: Tester possesses the knowledge of the internal system under test (SUT), and attempts to test most possible paths of the system.

When to stop testing?

Testing is potentially endless. We can never be sure to have covered all the possibilities for a product. Neither can we test until all the issues are traced and debugged - it is simply impossible. At some point, we have to stop testing and ship the software. The question is when.
The factors [5] [6] that directly influence the scope of testing are size and complexity of the software, amount of software integration work required, knowledge and experience of the team, and the time and budget constraints.

Test requirement

A test requirement [4] is a specific element of a software artifact that a test case must satisfy or cover. Test requirements can be described with respect to source code, design components, specification modelling elements, or even descriptions of the input space.
Example: If a trade is registered in the system, it must have an auto assigned unique trade ID.
A test set with minimum redundancy that can satisfy all the test requirements is called a minimal test set. In technical terms, given a set of requirements TR and a test set T that satisfies all test requirements, T is minimal if removing any single test from T will cause T to no longer satisfy all test requirements TR.

Coverage criteria

A coverage criterion [4] is a recipe for generating test requirements in a systematic way. It is a rule or set of rules that impose test requirements on a test set i.e. the coverage criterion describes the requirements in a complete and unambiguous way. Since the test engineers need to know how good a collection of tests is, the test set is measured against a criterion in terms of coverage.
Example: Function coverage, branch coverage, and statement coverage.

Test coverage

Dijkstra, in 1972 said that "program testing can be used to show the presence of bugs, but never their absence" [8]. There has been an increase in research about testing, test coverage criteria, and when to stop testing. One question that needs to be thought about is "what is effective test criterion or test coverage" that is, the criterion that defines what constitutes an adequate test. To measure what percentage of code has been exercised by a test suite, one or more test coverage criteria can be used. Test coverage criteria are usually defined as a rule or requirement, which the test suite needs to satisfy.

Given a set of test requirements TR for a coverage criterion C, a test set T satisfies C if and only if for every test requirement tr in TR, at least one test t in T exists such that t satisfies tr.
The test coverage by two test cases executed can be the same but the input data of 1 test case may find a defect while the input data of the other may not. Thus, 100% coverage does not mean 100% tested. Coverage is extremely important. It not only helps better the overall quality of the software, but also helps identify the parts of the code that need further testing, and weed out potential bugs early in the software development life cycle. With code coverage, not only developers and test engineers, but also the project managers and clients have a clear overview of the quality of the code, and reliability of the code of their product and services.

Given a set of requirements TR and a test set T, the coverage level is the ratio of the number of test requirements satisfied by T to the size of TR.
Coverage can be measured by the formula:

$$\text{Test coverage } \% = \frac{\text{Number of coverage items exercised}}{\text{Total number of coverage items}} \; x \; 100\%$$

Coverage models

For this research work, we will look into two possible code coverage models:

Requirement based coverage: Specifies the required testing based on identified features of the program and requirements of the software.

Structural coverage: Specifies the test requirements with respect to the program under test, and decides if a test set is adequate going by how thoroughly has a program been exercised.

Requirement based coverage

The requirement-based coverage, as name suggests, is measured against requirements of the product under test. That requirement coverage is adequate, is known by examining two factors i.e. has the product been tested against all functional and normal use requirements, and has the product been tested against all non-functional and abnormal usage requirement [12].
Requirement based coverage is also primarily linked to black box test technique, where the tester is not concerned about the internal structure of implementation. There is a close relation between high-level requirements and the part of the software or product to be tested.

Example 1: Consider the below requirement statement:
"If a trade is registered in the system, it must have an auto assigned unique trade ID."
An analyst developing a test case from the above requirement may derive the following scenario:

1. Enter primary trade data
2. Save to register the trade in the system
3. Verify that the new trade has a unique trade ID

Does the above test case adequately cover the high-level requirement discussed in the example above? Does passing such a test case indicate that the model has correctly captured the behavior required through this requirement? If not, what would additional test cases look like? The specification of the requirement as a property allows us to define several objective criteria with which to determine whether we have adequately tested the requirement. Thus, the coverage of such criteria can serve as a reliable measure of the thoroughness of the requirements-based testing activities [12].

Structural coverage

Structural coverage [13] is correlated to white box testing technique, where the tester has knowledge of the internal mechanism of the product. Structural coverage criteria are thus defined with respect to the source code, and can be more reliable measure of adequate testing. In other words, we check if each element of the software has been exercised during testing.

Typically, the single positive test case shown as part of example 1 is too weak of a coverage, since it leaves too many untested scenarios. A better approach would be to adopt one of the more rigorous

coverage criteria, as part of the structural coverage. Few things that can be covered as part of the structural coverage include:

1. Every point of entry and exit invoked at least once
2. Every basic condition has taken on all possible outcomes at least once
3. Each basic condition has been shown to independently affect the decision's outcome

Note that each instance of a condition within a formula is treated separately: the formula (A ^ B) _ A) has three basic conditions, and we must determine the independence of each instance of A separately. Structural coverage can be divided into various types depending on the method and measurement criteria being used namely graph coverage, logic coverage, syntax coverage, and input domain coverage.

Graph coverage

Bezier, in his book wrote that "*Testing is simple; all a tester needs to do is find a graph and cover it* [4]." Graph coverage can be applied to source code, specifications, and use cases among other things. It is also one of the most widely used models for the program under test.

To apply graph coverage criteria [4] [13], the first step is to define a graph abstraction of the source code. Next, we build a graph model, based on the information to be captured, and the way to present that information. The test requirements are identified i.e. structural entities in the graph that must be covered during testing. Lastly, based on the requirements, test paths are selected, and test data is derived such that the test paths can be executed. A graph based coverage criteria evaluates a test set in terms of how well the paths corresponding to the test cases cover the graph abstraction.

Example of graph coverage:
A graph G (refer figure 2.2) = (N, N_o, N_f, E) is [4]:

- A set N of nodes
- A set No of initial nodes, where No is a non-empty subset of N
- A set Nf of final nodes, where Nf is a non-empty subset of N
- A set E of edges, where E is a subset of N x N

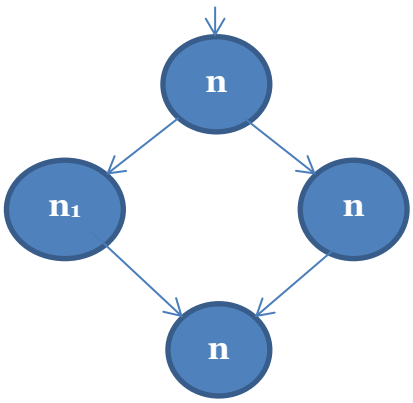

Figure 2.2: Graph coverage

$$N = \{n_0, n_1, n_2, n_3\}$$
$$N_0 = \{n_0\}$$
$$E = \{(n_0, n_1), (n_0, n_2), (n_1, n_3), (n_2, n_3)\}$$

Given a set of test requirements TR for a graph criteria C, a test set T satisfies C on a graph G if and only if for every test requirement tr in TR, there is at least one test path p in path T such that p meets tr.

Some commonly used graph models include:

Control flow graph: Captures information about control transfer in a program

Data flow graph: Augments control flow graph with data flow information

Dependency graph: Captures the data/ flow dependencies among the program statements

Cause-effect graph: Modelling relationships among program input conditions known as causes, and output conditions, known as effects

Control flow coverage criteria

Control flow coverage criteria use the control structure of a program to develop tests for the program. These tests cover the overall control structure of the program, which is represented using control flow graphs.

The control flow graph G = (N, E) of a program consists of a set of nodes N and a set of edge E. The nodes of a control flow graph are statements of the program and the edges represent the control flow between the statements. The approaches differ with respect to the handling of branching and the merging of branches, and the representation of segments of statements that are always executed together.

Moving on, we look into how control flow criteria is further divided into node coverage, edge coverage, condition coverage, complete path coverage, simple path coverage, and prime path coverage.

Node coverage

Node coverage [4] [13], also called statement coverage is widely used in the industry as a common criterion for thoroughness of software testing. Different standards govern how high the statement coverage should be. For example, avionics industry, the finance, and automotive industry would demand close to 100% statement coverage.
Statement coverage represents a specific coverage criterion of white box testing approach. For statement coverage, the test requirements are all the statements in the program, making it the simplest coverage criteria in white box testing. What statement coverage does best is to find the faulty elements in the code by executing it. Statement coverage covers only the true conditions i.e. the conditions that are expected, as shown in figure 2.3. Code coverage metric tells whether the flow of control reached every executable statement of source code at least once.

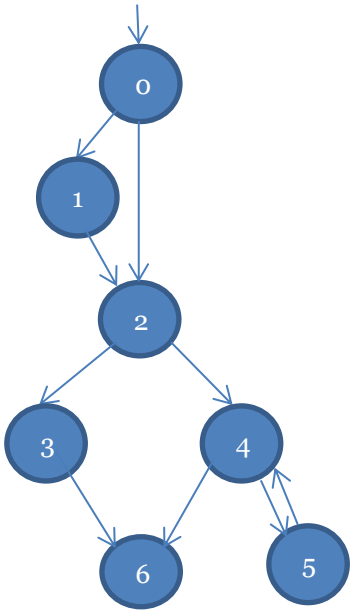

Figure 2.3: Node coverage

Node coverage
TR = {0, 1, 2, 3, 4, 5, 6}
Test paths = [0, 1, 2, 3, 6] [1, 2, 4, 5, 4, 6]

In the example above, TR are the test requirements to be covered, while test paths are the minimum optimal paths that allow us to cover all the requirements.
A set P of execution paths satisfies the statement coverage criterion if and only if, there is at least one path p in P such that node n is on the path p, for all nodes n in the flow graph.
The main advantages of statement coverage include verification of what the written code is expected to do or not do, and that it checks the flow of different paths in the program and it also checks whether those path are tested or not. The disadvantages include inability to test false conditions i.e. the conditions that are not expected, no execution of logical operators, and no execution of loops.

Edge coverage

Edge coverage [4] [13], also known as branch coverage or decision coverage is a testing method, which aims to ensure that each one of the possible branch from each decision point is executed at least once and thereby ensuring that all reachable code is executed. A branch is the outcome of a decision, so branch coverage simply measures, which decision outcomes, have been tested.
Branch coverage is stronger than statement coverage because covering all edges in a flow graph means that all the nodes are also covered. Thus, a test set that satisfies branch coverage criteria will have to satisfy statement coverage. As shown in figure 2.4, edge coverage is measured by the portion of condition branches that are covered by the test cases.

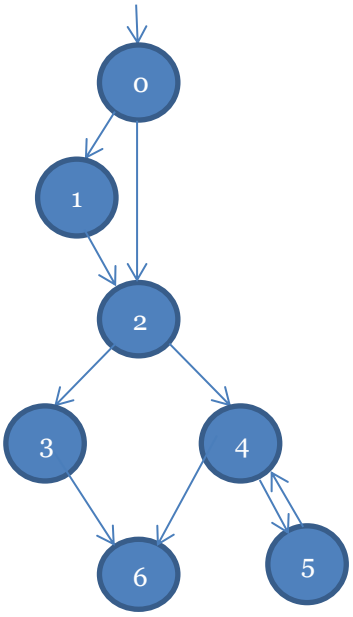

Figure 2.4: Edge coverage

Edge coverage
TR = {(0, 1), (0, 2), (1, 2), (2, 3), (2, 4), (3, 6), (4, 5), (4, 6), (5, 4)}
Test paths = [0, 1, 2, 3, 6] [0, 2, 4, 5, 4, 6]

In the example above, TR are the test requirements to be covered, while test paths are the minimum optimal paths that allow us to cover all the requirements.

A set P of execution paths satisfies the branch coverage criterion if and only if, there is at least one path p in P such that p contains the edge e, for all edges e in the flow graph.

Branch coverage reports whether Boolean expressions tested in conditional statements evaluated to both true and false. The Boolean expression is considered one true-or-false predicate regardless of the logical operators it contains. Additionally, this metric includes coverage of switch-statement cases, exception handlers, and all points of entry and exit. The advantages of branch coverage include easy assessment of coverage, while it may not always execute and trace dead code.

Condition coverage

Condition coverage [4] is closely related to decision or branch coverage but has better sensitivity to the control flow. This is possible because every condition in a decision has to take all possible outcomes at least once.

A condition is a Boolean expression containing no Boolean operators, as depicted in figure 2.5. A decision is a Boolean expression composed of conditions and zero or more Boolean operators. A decision without a Boolean operator is a condition. If a condition appears more than once in a decision, each occurrence is a distinct condition.

A && B (A && B) || C

Figure 2.5: Condition coverage

The decision (A && B) || C is one decision and (A && B) is actually another decision. Here, A, B, and C are conditions since there are no Boolean operators.

Complete path coverage

Path coverage [4] [13] is a white box testing technique where tests are executed in such a way that each condition in a decision takes on all possible outcomes at least once, and each point of entry to a program is invoked at least once i.e. every branch taken each way, true and false. It helps in ensuring that all branches in the code behave as expected. Figure 2.6 shows how path coverage is the most comprehensive test that a suite can provide. It can find more bugs, however, it is hard to achieve on complex code having a number of paths and decisions involved. Thus, complete path coverage is a metric, more feasible for smaller and critical parts of the code.

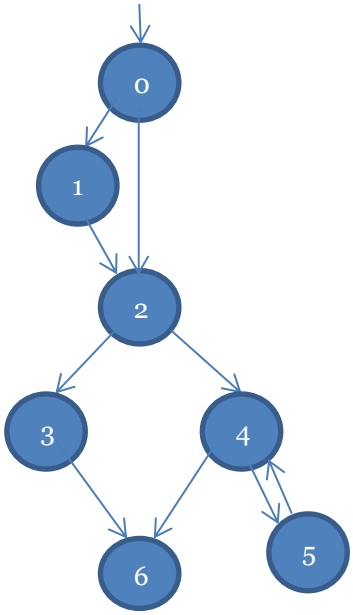

Figure 2.6: Complete path coverage

Complete path coverage
TR = {(0, 1, 2), (0, 2, 3), (0, 2, 4), (1, 2, 3), (1, 2, 4), (2, 3, 6), (2, 4, 5), (2, 4, 6), (4, 5, 4) , (5, 4, 5) , (5, 4, 6)}
Test paths = [0, 1, 2, 3, 6] [0, 1, 2, 4, 6] [0, 2, 3, 6] [0, 2, 4, 5, 4, 5, 4, 6]

In the example above, TR are the test requirements to be covered, while test paths are the minimum optimal paths that allow us to cover all the requirements.

A set P of execution paths satisfies the path coverage criterion if and only if P contains all execution paths from the start node to the end node in the flow graph.

The biggest advantage of path coverage is the amount of logical flow coverage it gives, thus attaining very high level of test coverage; however, there are uncertainties too, such as, whether or not all paths are necessary, and the number of paths being exponential to the number of branches.

Simple path coverage

A path from node n_i to n_j is simple if no node appears more than once, except the possibility of the first and last nodes being the same.
Even a small program may have a very large number of simple paths, as shown in figure 2.7 [4] [13]. Most of these simple paths are not worth addressing since they are sub paths of other simple paths.

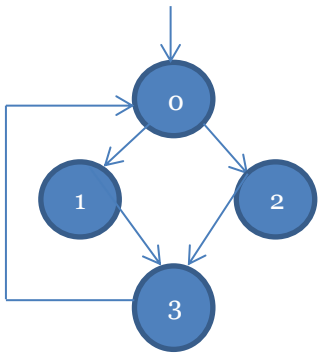

Figure 2.7: Simple path coverage

Simple path coverage
Simple paths = [0, 1, 3, 0], [0, 2, 3, 0], [1, 3, 0, 1], [2, 3, 0, 2], [3, 0, 1, 3], [3, 0, 2, 3], [1, 3, 0, 2], [2, 3, 0, 1], [0, 1, 3], [0, 2, 3], [1, 3, 0], [2, 3, 0], [3, 0, 1], [3, 0, 2], [0, 1], [0, 2], [1, 3], [2, 3], [3, 0], [0], [1], [2], [3]

In the example above, the simple paths are all the possible paths, covered exactly once. Covering code using simple path is difficult and not optimal because of the number of paths possible on even a small code base.

Prime path coverage

The simple path that does not appear as a sub path of any other simple path is known as prime path.
To avoid enumerating the entire set of simple paths we list only maximal length simple paths, as shown in figure 2.8, referred to as prime paths [4] [13].

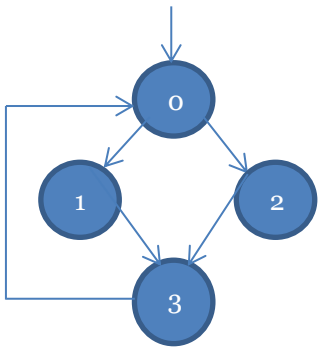

Figure 2.8: Prime path coverage

Prime path coverage
Prime paths = [0, 1, 3, 0], [0, 2, 3, 0], [1, 3, 0, 1], [2, 3, 0, 2], [3, 0, 1, 3], [3, 0, 2, 3], [1, 3, 0, 2], [2, 3, 0, 1]

In the example above, prime paths are the maximal length simple paths from the example in figure 2.7. This ensures that all simple paths are covered as a subset of the prime paths, thus reducing the overhead.

Criteria subsumption

Sometimes one coverage criterion is more powerful than another one [4]: any test set that satisfies C1 might automatically satisfy C2. Coverage criterion C1 subsumes C2 if and only if every test set that satisfies C1 also satisfies C2.

There are two ways to subsume:
1. Test requirements for C1 are always a superset of the test requirements for C2;
E.g., Calculus 2 test requirements are a superset of Calculus 1 test requirements.
2. Many-to-one mapping exists between TRs for C1 and C2; i.e. for any test requirement tr that C2 imposes, C1 has at least one test requirement tr 0 which, when satisfied, also satisfies tr. (key point: tr 0 may be different from tr).
Software example: Branch coverage (Edge coverage) subsumes statement coverage (Node coverage).

Mutation adequacy (quality)

Software testing is aimed at detecting faults in software. A way to measure how well this objective has been achieved is to plant some artificial faults into the program and check if they are detected by the test. A program with a planted fault is called a mutant of the original program. If a mutant and the original program produce different outputs on at least one test case, the fault is detected. In this case, we say that the mutant is dead or killed by the test set. Otherwise, the mutant is still alive. The percentage of dead mutants compared to the mutants that are not equivalent to the original program is an adequacy measurement, called the mutation score or mutation adequacy [4].

Unit testing

Unit tests are not an effective way to find bugs or detect regressions. Unit tests, by definition, examine each unit of the code separately. When these units are integrated to work as an application, the process is much more complex. Proving that components X and Y both work independently does not prove that they’re compatible with one another or configured correctly. In addition, defects in an individual component may bear no relationship to the symptoms an end user would experience and report. Moreover, since the preconditions are designed for specific unit tests, they will not ever detect problems triggered by preconditions that were not anticipated.

A suite of good unit tests is valuable: it documents the design, and makes it easier to refactor and expand the code while retaining a clear overview of each component's behavior. However, a suite of bad unit tests is immensely worrisome: it doesn't prove anything clearly, and can severely inhibit the ability to refactor or alter the code in any way.

Related work

Mockus et al. conducted a study [18] on two dissimilar industrial software projects, and concluded that higher test coverage resulted in fewer reported problems, thus proving that code coverage was a sensible and practical measure of testing effectiveness. The research also concludes that code coverage is moderately related to code complexity.
The study looks into defect detection effectiveness curve for coverage levels, effect of code complexity on coverage, and cost effective measures for particular requirement sets. However, it does not compare coverage models in terms of their effect on quality to decide which coverage metrics are best suited to effectively test software products based on imperative programming languages such as Java.

A study by Muhammad et al. on test coverage in software testing [16], at the Technological University of Malaysia, found that the main focus of researchers in the area of software test coverage has been related to code coverage as an exit criteria, comparing various coverage tools, and techniques available to generate test cases to satisfy test coverage criteria. There are not many studies that compare performance of various coverage metrics for industrial use.

In a work related to this thesis, Kochhar et al. conducted a study on adequacy of testing [28] using over 300 large open source projects in Java. They analyzed correlations between various metrics such as lines of code, cyclomatic complexity, and number of developers in the project. This study concludes a correlation between cyclomatic complexity and lines of code, while proving that there is no relation between number of developers and coverage.

Another closely related research work done to this thesis, is Comparing Non-adequate Test Suites using Coverage Criteria [20], where various coverage criteria are evaluated for non-adequate test suites resulting in branch coverage being the best coverage criteria in such cases. Another closely related work Code Coverage for suite evaluation by developers [19], examines the relation between code coverage and quality in a different domain and a sample space having better coverage than used in this thesis work. This study concludes that statement coverage was the best-suited criteria for unit tests, in terms of reflecting the code quality.

Since two very closely related studies [19][20], with relatively consistent data, compared to the one available during this study, had different conclusions in the form of branch coverage[20] and statement coverage[19], it was an interesting prospect to look into the results of defect detection analysis conducted on a totally different data set in the fin-tech domain.

Chapter 3: Method

This chapter givers an overview of the process followed during this study, and explains the methodology used.

Process overview

Test suites are identified as a consistent sample to check how well they perform on the coverage scales. The code base is Java, and the test suites have test cases written in JUnit. The test coverage is measured using different tools such as JaCoCo, JMockit and PITest. Once the results are recorded, an analysis is done to compare how well various coverage criteria perform in terms of quality, and what factors impact the coverage. A brief study is conducted into the test process followed by product teams within the organization and the industry best practices for unit test coverage. Improvements are suggested, including best practices, and some new unit tests are implemented using the recommendations as proof of concept.

Figure 3.1 illustrates the overview of the process followed during this degree project. It starts with an analysis of the project code base to select the sample space under test, followed by an analysis of the unit tests in the sample space with respect to industry standards. Further, an analysis is performed to establish a relation between test coverage criteria and the quality of tests. This is followed by analyzing the test process in the project, and concluded by recommendations to the project team.

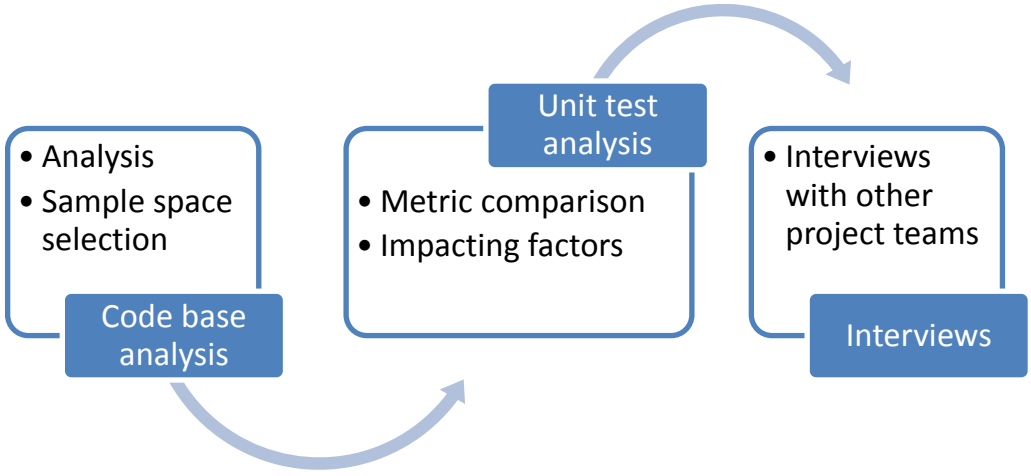

Figure 3.1: Process overview of the project

Tools used

One of the first decision points was to select the tools to be used for measuring test coverage and quality in the project. Below open source tools were used for the analysis:

SonarQube

SonarQube [21] collects and analyzes the source code to measure reliability of the code along with providing many other useful statistical data. It takes into account factors that affect the code base, including minor details such as indentations and critical design errors. Thereby it lets developers to access and track code analysis data ranging from indentation errors, potential bugs, and code defects to design inefficiencies, code duplication, lack of test coverage, and excess complexity. SonarQube uses JaCoCo to measure code coverage on two coverage criteria i.e. line coverage and branch coverage.

JaCoCo

JaCoCo [22] is an open source toolkit for measuring code coverage. It measures line and branch coverage based on the code covered by running unit test cases and provides a visual report, including highlighted lines of code and the total percentage of code executed in each method. JaCoCo uses the standard JVM Tool Interface. During a build, a JaCoCo agent attaches itself to a JVM. When the JVM starts, and whenever a class is loaded, JaCoCo can use the agent to analyze when the class is called and what lines are executed. This is how code coverage statistics are collected.

JMockit

It was important to look beyond the coverage criterion such as line and branch coverage. This would ensure that the research takes into account the more complex coverage models that subsume line and branch coverage by criteria. To be able to do this, path coverage was measured using JMockit. JMockit [23] was easy to use, and set up. It was also one of the few tools available to work with the kind of complex legacy code base that was to be analyzed.

PITest

In order to analyze the quality of unit tests and assess which coverage model would work best for the fin-tech domain, PITest [24] was used. PIT runs the unit tests against automatically modified versions of the code base. Upon changing, the application code should produce different results and cause the unit tests to fail.

Data Analysis

The code base to be analyzed was written in Java. The project itself was a multi-module project with varying module sizes in terms of both lines of code and number of classes, different levels of coverage per module and varying complexity. It was important to attain certain statistical measurements of each module to understand the dataset available. Thus, SonarQube was used to analyze the quantitative and qualitative information about the available dataset. For this thesis, the most relevant information like

lines of code, cyclomatic complexity, line coverage, branch coverage, number of functions in the code, and code smells were taken for analysis using SonarQube.

How to measure coverage?

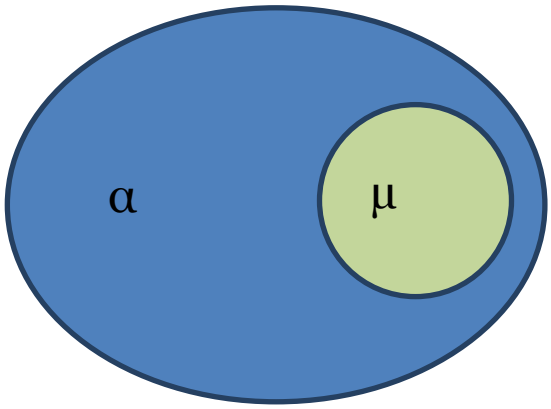

Figure 3.2: Measurement scope of the project
α is the entire product, and μ is the part of the code which has good unit test coverage.

The analysis of test coverage was done using structural coverage metrics. Metrics such as statement, branch, and path coverage were used during this study. This was done because these are the most commonly used metrics in the open source tools such as JaCoCo, JMockit and PITest, available for this study, and thus most likely to be used in a real world development environment. In addition, since the clearing system project was one with huge amounts of data and legacy code to be analyzed, these metrics would be practical in terms of implementation and data reliability.

Regression analysis for metric comparison

In order to compare the performance of various test coverage metrics to the quality they are able to provide, a regression analysis was performed. Regression analysis [25] is a statistical approach to show the relation between variables. It is a way of mathematically sorting out the variables (factors) that actually have an impact on the dataset. It answers the questions: Which factors matter most? Which can we ignore? How do those factors interact with each other?

Regression analysis has also been used in some studies [18] [19] that also analyze the defect detection of various coverage metrics. A key difference, however, is the data under analysis, since this study focuses on a project with hundreds of thousands of lines of code, with a large section of legacy code, all of which had highly inadequate test suites in terms of coverage levels, while others use data from multiple smaller projects having more than 70% code coverage levels.

There are dependent variables i.e. the main factor to understand or predict. Then, there are independent variables i.e. the factors suspected to have an impact on the dependent variable.

R-Squared value

R-squared [17] is a statistical measure of how close the data are to the fitted regression line. It is also known as the coefficient of determination, or the coefficient of multiple determinations for multiple regressions.
R-Squared is defined as the percentage of the response variable variation that is explained by a linear model. An R-square of 1 indicates that the regression line perfectly fits the data while an R-square value of 0 indicates no relation of the datasets.
The R-square value [17] was calculated to establish a relation between the coverage models and quality, and to check the relationship of code coverage and cyclomatic complexity. This is a statistic that gives information about the fit of a model in terms of how well the regression line approximates to the real data points in a graph.

Correlation analysis for factors impacting coverage

In order to understand the results, it was important to understand what factors can impact coverage. To do this, a correlation analysis was performed. Unlike regression analysis that is done to predict the behavior of one variable based on the other variable, a correlation analysis is conducted in order to understand if two variables share a positive, negative or neutral relation. The correlation coefficient r measures the strength and direction of a linear relationship between two variables. An *r-value* of zero suggests a neutral relation, while 1 and -1 suggest positive and negative correlation respectively.

The factors that were considered in this analysis were code complexity, number of functions in the code, lines of code and code smells i.e. syntactical faults with the code structure. These practical and measurable factors were likely to have an impact on the coverage levels.

Since the parts of the sample space with no coverage have no relational dependency to any of the factors concerned, and only increased the noise in the results, it was decided not to use those uncovered parts in the correlation analysis.

The r-values [17] for the concerned factors were calculated for the sample space with coverage and a comparison was drawn to understand the relation they had with line, branch, and path coverage metrics.

Interviews

Since one of the objectives of this study was to understand the unit testing practices and test process that can help in improving coverage and quality of a software, it was crucial to understand how these are practiced in the IT industry and within other teams at NASDAQ. This would not only help in establishing basics of unit testing, such that the unit test suites available for this study can be examined, but also would help in making recommendations to the clearing system project.

These interviews were conducted with managers and engineers from different teams within the organization, which had an established test process and above 65% code coverage in their specific projects. This included test managers, product managers, and the developers who were working closely with the products concerned. The interviews were conducted using predefined questions that were targeted at understanding the test processes, and defect prevention strategy followed in the specific teams.

The following questions were asked:

1. How do developers write unit tests?
2. How can unit tests be improved?
3. How do you ensure minimal defects?
4. What kind of test process do you follow?

The answers to these questions, and the detailed discussions during the interviews helped come up with a set of best practices to follow while performing unit tests and visualize the test process followed in a real world business critical setup.

Chapter 4: Results

In this chapter, the data selection is motivated, and the results of the study are presented.

Module wise coverage

The overall test coverage (line and condition) in the system was found to be below 10%. The module wise breakdown of the test coverage in the system (figure 4.1) shows that the tests are very unevenly distributed. This proved to be far below the desired unit test coverage levels, which are above 70% across the IT industry.

Module #	# Classes	# LoC	# Unit Tests	Line Coverage (%)
Module 1	**1149**	**81448**	**233**	**5.6**
Module 2	1402	61248	7	1.2
Module 3	**430**	**37728**	**172**	**28.5**
Module 4	181	16757	33	0.9
Module 5	103	7269	0	0
Module 6	145	6074	74	13.1
Module 7	34	5278	34	8.4
Module 8	46	3285	10	14.4
Module 9	66	3076	1	2.4
Module 10	46	1745	0	0
Module 11	21	1137	0	0
Module 12	10	1132	5	29.8
Module 13	12	917	2	19.7
Module 14	18	831	17	27.8
Module 15	16	819	0	0
Module 16	14	585	0	0
Module 17	15	369	0	0
Module 18	1	164	0	0

Figure 4.1: Line coverage per module of code

Data distribution

The overall project code base i.e. module's 1 – 18 in figure 4.1 had more than 220,000 lines of code spanning over 3700 classes. For this size of code base, it had roughly 600 unit tests, where many modules had zero coverage. This caused a problem of inconsistent dataset to choose from, hence making the data selection challenging.

Thus, for the experiments done during this work, only two modules with a major chunk of code base and unit tests i.e. module 1 and module 3 shown in figure 4.1 were considered. The selected code base comprised of roughly 120,000 lines of code, 1600 classes, and 400 unit tests. This was done keeping in

mind that the remaining modules that had no code coverage, or were being covered scarcely, may have a biased impact on the data and eventually the results of the study.

To ensure that the dataset considered in the study was not biased, a comparison was done between the complexity of the code, and lines of code of the code base to the dataset selected for this study. This comparison, as shown in figures 4.2, and 4.3 that were created using the data from SonarQube, establishes a relationship between the code base (all), the part of the code that has unit tests (covered), and the actual dataset used for this study (core). The structural tendency of this relation is discussed in the following section.

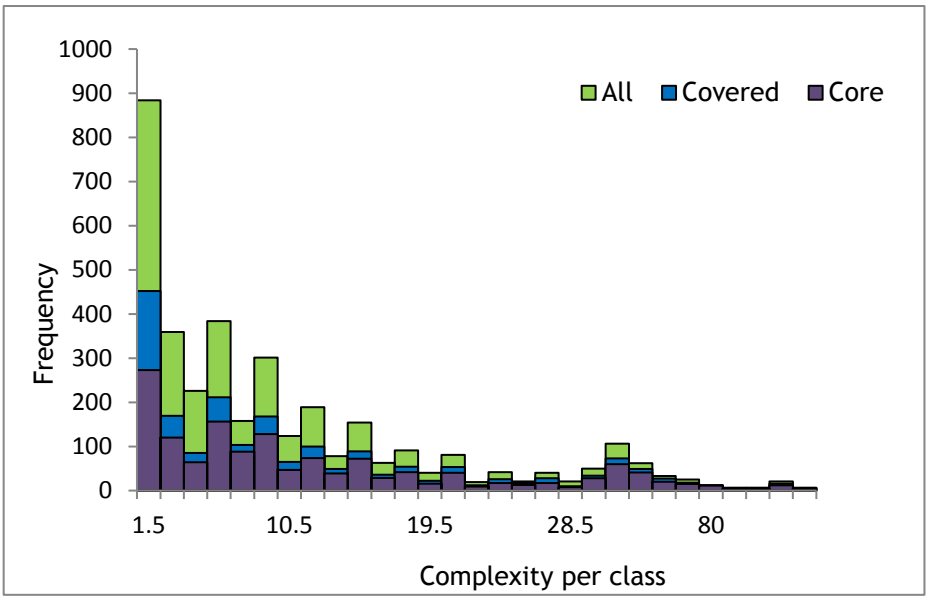

Figure 4.2: Complexity of the code

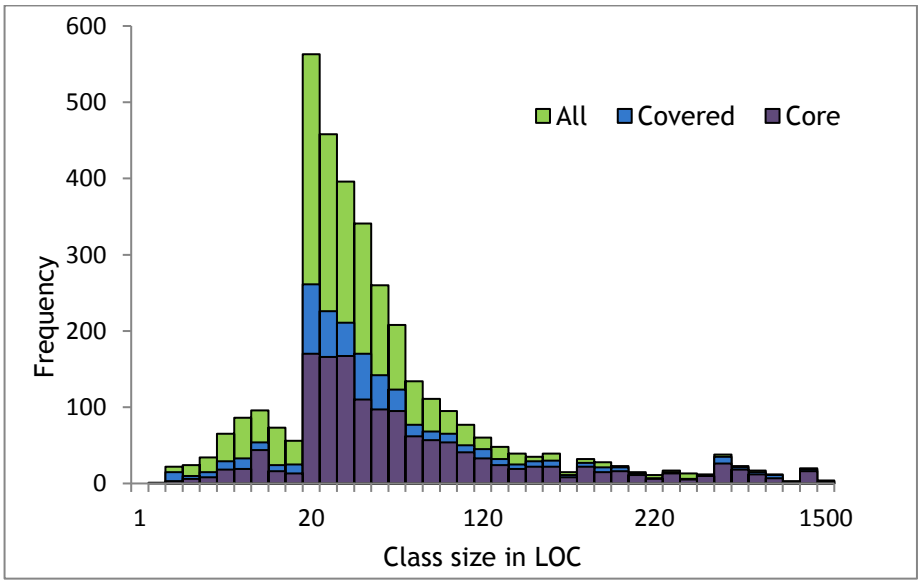

Figure 4.3: Lines of code

Structural tendency

Looking at the difference of size between the actual code base and the selected sample for this study, it was important to compare the distributions for the lines of code and complexity to verify that the selection did not bias the sample data in these dimensions. An important dimension where bias could occur is the complexity of the code since the simpler coverage metrics are closely related to the complexity of the code. The histograms in figures 4.2 and 4.3, due to their consistent representation of data for all three types of data shown, suggest that the selection did not bias the sample data. Thus, the selected dataset actually represents the entire project code base, and any results obtained on this dataset, will be reflective similarly on the entire code base.

Metric comparison

The relationship between test coverage and mutation (quality) in the selected data set had a relational tendency, shown by R^2 in the graphs 4.4, 4.5, and 4.6, which were created using the data from SonarQube. This relational tendency was unexpected because of the uneven test coverage distribution within the software under test.

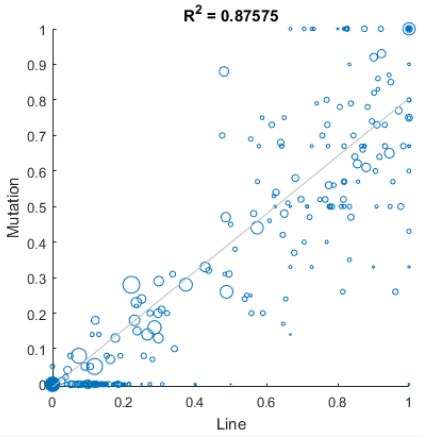

Figure 4.4: Line coverage – Mutation

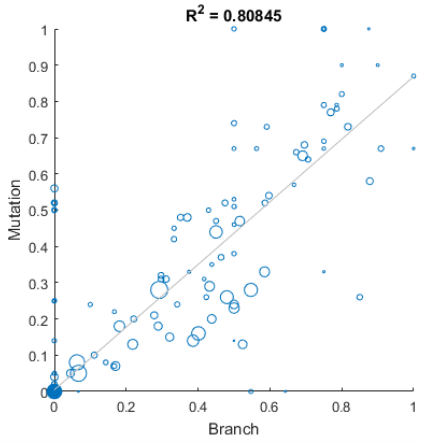

Figure 4.5: Branch coverage – Mutation

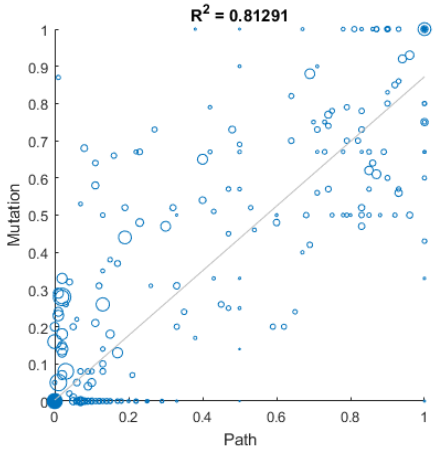

Figure 4.6: Path coverage – Mutation relation

The coverage mutation relation shows which coverage model best fits the quality graph. Each data point in the graphs corresponds to a class in either of the modules from the dataset under test. The size of the circles represents the size of the class in terms of lines of code. Based on the regression line, and the R-square values, it was observed that line coverage seemed to perform better than branch and path coverage in terms of predicting the quality of the tests.

Factors impacting coverage

Figure 4.7 shows the correlation graphs when those were drawn on the complete sample space and the reduced sample space, leaving out uncovered classes. Figure 4.8 shows the r-values or the correlation values for line coverage and complexity of the code when taken with noise i.e. classes with zero coverage and without noise i.e. classes with some coverage. The uncovered part of the code does not seem contribute to this analysis, in any way other than taking the correlation values toward neutral. Since lines of code remains an important contributing factor to the code complexity [29] [30], hence relating to coverage levels, a neutral correlation does not seem to be true. The results are similar to that of line coverage for both branch and path coverage as well. Therefore, the removal of uncovered code from this analysis seemed to be justified.

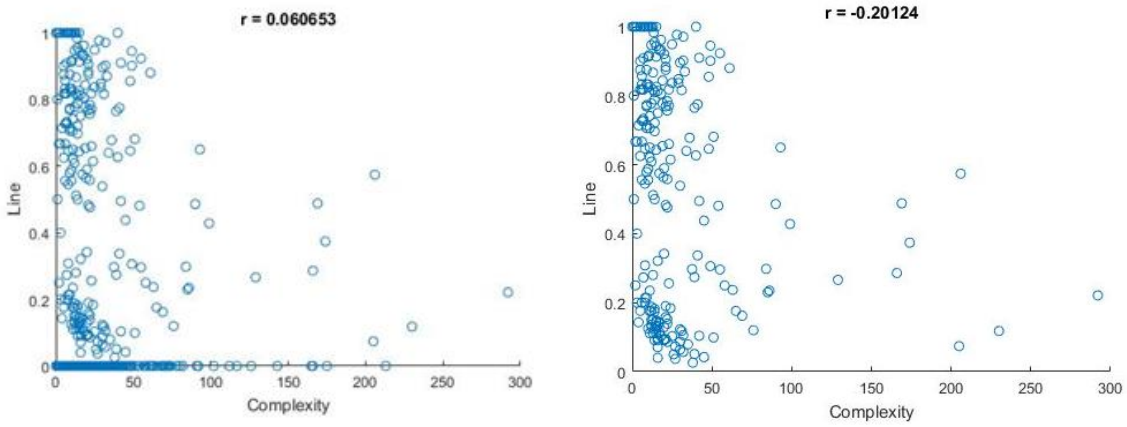

Figure 4.7: Correlation graphs of line coverage and complexity with and without the noise

r-value/ line cove.	Complexity	LoC	#Functions	Code Smells
No Noise	-0.20	-0.20	-0.12	0.05
With noise	0.06	0.04	0.08	0.04

Figure 4.8: Line coverage correlation coefficients with and without noise

The correlation between coverage and complexity exists, but is weak when compared to lines of code. The correlation between coverage, number of functions, and code smells was negligible, as shown in figure 4.9. The relationship between test coverage and lines of code in the selected data set had somewhat negative relational tendency, shown by the graphs 4.10, 4.11, and 4.12, which were plotted using the data from SonarQube.

r-value /metric	Complexity	LoC	#Functions	Code Smells
Line	-0.20	-0.20	-0.12	0.05
Branch	-0.31	-0.34	-0.02	0.20
Path	-0.29	-0.27	-0.07	0.08

Figure 4.9: Correlation of factors impacting coverage

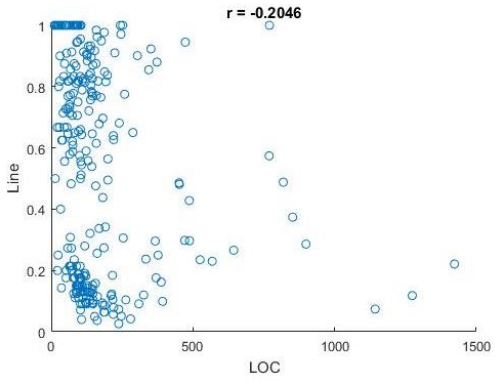

Figure 4.10: Line coverage – LoC

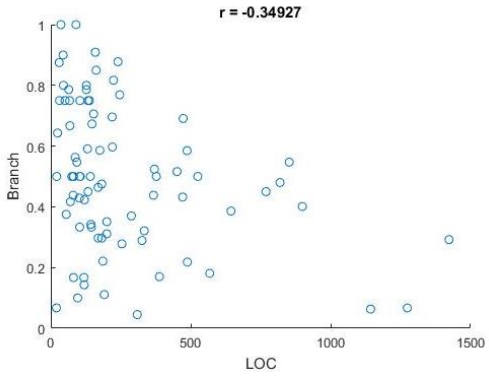

Figure 4.11: Branch coverage – LoC

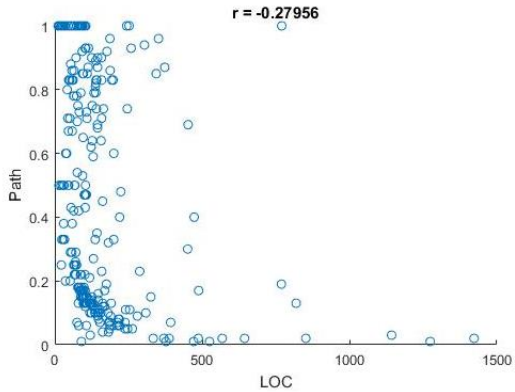

Figure 4.12: Path coverage – LoC

Interview results

Unit testing best practices

Based on the interviews and literature, the following set of practices [26] [27] for writing unit tests were found to be necessary to achieve good unit tests in terms of coverage and quality:

1. Unit tests should run completely in-memory:
   Avoid writing unit tests that access a database, or read from the file system.
2. Unit tests should not be skipped:
   Unit tests that are skipped provide no benefit, but still have to be checked out of source control and compiled. Instead of skipping unit tests, remove them from source control.
3. Ideally, each unit test method should perform exactly one assertion:
   Without strong assertions, unit tests are not able to catch bugs. In other words, unit tests that lack strong assertions ensure that the production code does not throw an exception.
   In order of decreasing strength, assertions fall into the falling categories:
   Strongest assertions: asserting on the return value of a method
   Strong assertions: verifying that vital dependent mock objects were interacted with correctly
   Weak assertions: verifying that non-vital dependent mock objects (such as a logger) were interacted with correctly
4. Assertion parameter order:
   The order of assertion parameters should be correct. Example, use `assertEquals (expected, actual)`.
5. Naming convention:
   Naming convention should be decided and followed consistently across the project, such as `methodUnderTest_condition`.
6. Test classes should be in the same Java package as the production code under test:
   When in the same package as the production class being tested, test classes can use package-private classes and invoke package-private methods.
7. Test code should be separate from production code:
   The default project structures should be designed in a way so as to separate the test and production code.
8. Do not use static members in a test class:
   Static members make unit test methods dependent. Don't use them! Instead, strive to write test methods that are completely independent.
9. Do not write catch blocks only to fail a test or pass a test:
   It is unnecessary to write your own catch blocks that exist only to fail or pass a test because the JUnit framework takes care of the situation for you.

10. Test classes directly:
Indirect testing occurs when a class is tested only because it is a dependency of another class. Indirect testing is not recommended because we lose coverage of the dependent class if we change the implementation of the main class.

Problems in existing unit tests

Upon conducting an analysis of the clearing systems unit tests based on the interview results, a few issues were found. Some of those are listed below:

- Unit tests spanning over multiple classes
- Initializations within the methods that make testing difficult
- High interdependency of the methods
- Calls to several static methods not mocked
- One naming convention not followed by all developers

These problems not only made the code hard to understand, but also made it difficult for the developers to refactor the code, and conduct proper unit tests.

Test process in the trading system

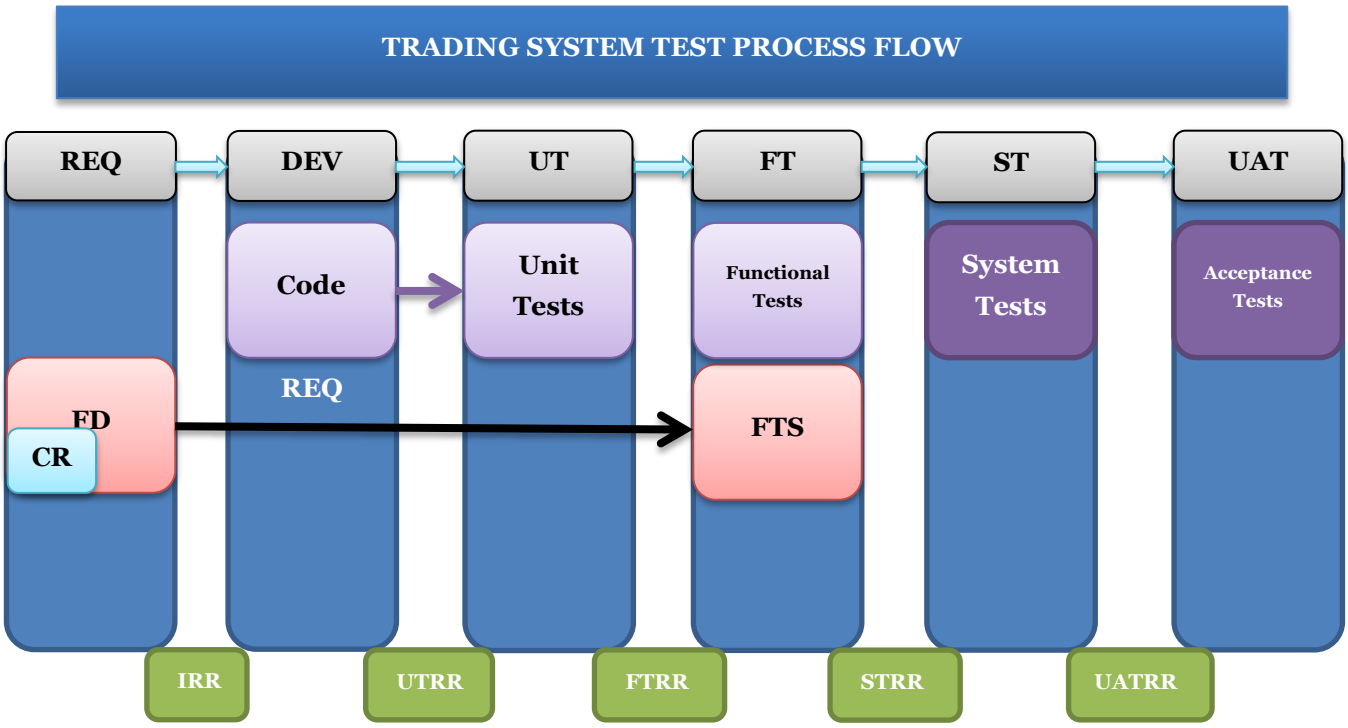

Figure 4.13: Trading system test process

REQ	Requirement analysis	**IRR**	Implementation readiness review
DEV	Development	**UTRR**	Unit test readiness review
UT	Unit testing	**FTRR**	Functional test readiness review
FT	Functional testing	**STRR**	System test readiness review
ST	System testing	**UATRR**	User acceptance test readiness review
UAT	User acceptance test	**CR**	Change requests
FD	Functional description	**FTS**	Functional test specifications

Figure 4.14: Legend

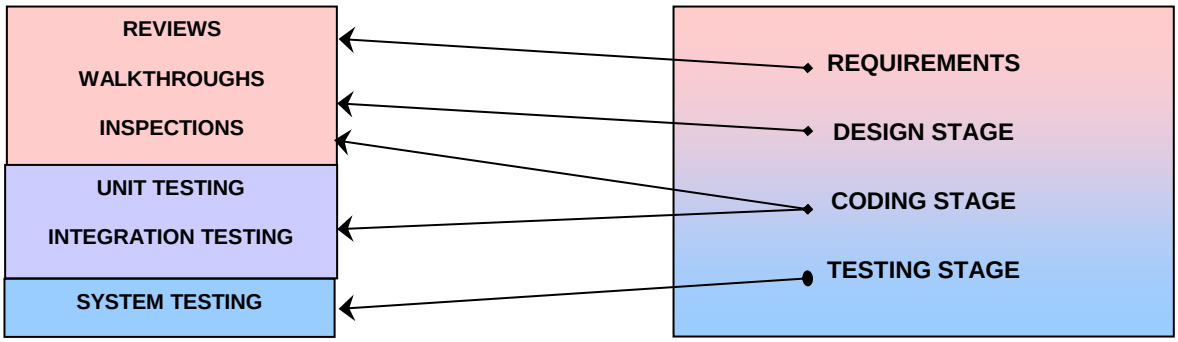

Figure 4.15: Defect prevention strategy graph

Based on the interviews, figure 4.13 shows the established test process in another project within the organization. Refer the legend in figure 4.14 for terms used in the test process flow diagram. The test process in the trading team depends on multiple kinds of tests performed by different teams that are responsible for that work area. In order to ensure a smooth transition and that the product is ready for the next level of testing, a readiness review meeting is conducted before taking up any testing, and any exceptions are laid out during this meeting.

The strategy graph in figure 4.15 shows the defect prevention strategy followed in the same project. This approach not only assures that fewer faults occur in production build, but also make it easier to identify the root cause of any issues.

Test process in the clearing project

It was found that clearing system did not have an established test process to drive the project and ensure quality in the end product. This was being done by conducting very few unit tests and some functional test and regression, on a need basis.

An example of this need based testing was noted in the scenario of day light savings time. During production, the lot size from two regions did not match in the clearing engine, and it was later identified that the system was not accounting for day light savings time adjustment. Thus, a test case was later developed to cover this corner case.

Technically, such an approach is known as context driven testing [14], which is based on the test objectives, techniques, and deliverables, instead of following the best practices of testing as discussed earlier in this section.

Chapter 5: Discussion

In this chapter, the research questions are answered and the results are discussed.

RQ 1: Which test coverage metric is most suited for unit testing in a business critical setup?

Three coverage metrics including line coverage, branch coverage, and path coverage were examined during this study. *Line coverage* proved to be the best suited unit testing metric. Figure 5.1 shows the R-squared values, i.e. the correlation coefficient for line coverage is best at 87%, followed closely by path coverage at 81% and branch coverage at 80%. This shows that having good and meaningful unit tests that provide adequate line coverage would ultimately ensure that the units of the product have good quality. Given that the line coverage is one of the simplest to measure test coverage model, and perhaps one of the simplest to implement, it provides enough reasons to be implemented in similar projects.

	R-squared
*Mutation x **Line** coverage*	**0.87575**
*Mutation x **Branch** coverage*	0.80845
*Mutation x **Path** coverage*	0.81291

Figure 5.1: Correlation coefficients

The reason for better performance of line coverage seems to be problems with the code structure itself, since that makes it difficult to have better branch and path coverage compared to current levels. While a study [19] shows line coverage performing better on this scale, another study [20] concludes that branch coverage was the best suited metric. However, it is worth noting that the coverage and code structures used in the latter were better than the ones available for this work, which would explain why branch and path coverage could not perform better than the simpler line coverage metric.

RQ 2: What factors affect code coverage?

Out of the four factors examined during this study i.e. cyclomatic complexity, lines of code, number of functions and code smells, only two factors proved to be relevant to some extent in influencing code coverage. These were *lines of code*, and *cyclomatic complexity*.

The scatter graphs in figures 4.10, 4.11, and 4.12 shows the negative correlation between code coverage and lines of code for line, branch, and path coverage respectively. This means that as the number of lines of code increases, the coverage level decreases. Kochhar et al. in their study [28] also got results

similar to that in this study. A low coverage level from existing unit tests seems to be one of the reasons for this result, since both the studies reach similar conclusions.

The cyclomatic complexity of software generally increases with the increase in number of lines of code and functions it has [29] [30]. The software tends to become more prone to errors with the increase in its complexity. The figure 4.9 shows the negative correlation between complexity and line, branch, and path coverage metrics. This means that the coverage decreases with the increase in the code complexity. While complexity should tie into code coverage, i.e. obtaining code coverage of 80% on a binary with cyclomatic complexity 100 is more difficult than getting coverage of 80% on a binary with cyclomatic complexity 10. In this case, the relation between coverage and complexity was rather weak. This may be because the sample space in itself had legacy code that had structural issues along with significantly low (less than 10%) levels of unit test coverage.

The figure 4.9 also shows the negligible (close to neutral) relation between the coverage metrics, number of functions, and code smells. While code smells do not have an impact on the coverage, the neutral relation between code coverage and number of functions was something that stood out. This is because the number of functions in the code is one of the important factors of increase or decrease in code complexity. And since code complexity had a weak relation with the coverage levels, a neutral relation between coverage levels and number of functions was not expected. On analysis, it was found that this may be because a lot of the methods were helper methods that do not serve any purpose other than calling methods from other classes, and only add to the number of functions but do not contribute to the complexity of the code.

RQ 3: What kind of unit testing practices and test process can help in improving the code coverage level of a software product?

Upon analysis during this thesis work, it was found that the existing unit tests in the clearing system project lacked basics such as proper naming conventions, and lack of mocking data in case of static methods. This not only makes testing difficult, but also increases the chances of the unit tests not being fully utilized.

The fact that the interviews conducted for this study were with engineers and managers from the teams that had good unit test coverage, and were following the best practices was one of the indicators of good unit test practices helping achieve good coverage. In any work domain, teams like to be responsive to business, and be able to make changes to the software product upon request. These changes should be implemented in a low risk manner, which can be achieved by having good unit tests, that not only provide adequate coverage, but are also effective. Following the best practices listed in the chapter 4 of this report, should ensure that the unit tests provide good coverage and can be good predictors of the software quality by making it easier for the developers to understand and modify the tests in case part of the code is being refactored or changed due to requirement changes. These best practices will also ensure that bugs can be detected and isolated in a better way.

Having a test process from requirements to production will not only help in increasing the testing efficiency in the clearing system project, but also help ensure good coverage levels for unit tests, and even other tests being done before production release. This will be possible during the readiness review meetings, where the responsible team members will have to lay out and justify the uncovered part of the code as to why the code was left untested. During discussions in the interviews conducted, one of the managers mentioned that although the review meetings last less than half an hour, they have proven to be very effective in improving the overall testing of the product. And, that since incorporating such a structure one and a half years ago, they have noticed a significant improvement in the code coverage and drop in the number of defects being reported by the customer.

Chapter 6: Conclusion

In this chapter, a conclusion of the thesis work is drawn and recommendations and future work are suggested.

This degree project empirically proves that the test coverage in the clearing system project is below the desired level of 70% coverage and that among the three coverage metrics examined (line, branch, and path coverage) for unit testing; line coverage is the most effective coverage metric in terms of quality. Line coverage also correlates the most closely to lines of code. If implemented in a correct manner, line coverage also remains one of the most cost effective test coverage metric, in terms of ease of implementation and effect on the code quality. The project also examines the factors that affect code coverage, and concludes that lines of code and cyclomatic complexity of the code have a weak linear correlation to coverage.

In addition to the coverage data, this degree project also suggests unit test best practices, and looks upon a detailed test mechanism for such projects, based on interviews conducted within the organization.

Recommendations

After the analysis and comparisons drawn in the results and discussion sections, NASDAQ clearing system is advised to bring in an exhaustive test process for their project. It will however, not be practical to simply implement the exact process discussed in this report. Thus, clearing system team needs to factor out what parts of the process will be easily implemented and suitable in the long run. This will in turn enhance the project quality and ensure fewer bugs in production code. NASDAQ clearing system is also advised to refactor some parts of their code to make testing process easier and more convenient. This includes the legacy code in the system, and other parts that may have not been written keeping unit testing in mind. NASDAQ clearing system is also advised to follow standard best practices while testing their code. The areas of improvements here include the naming conventions, mocking, and reducing the dependencies. Integrating tools like SonarQube, which are relatively easy to set up and display good data and statistics in terms of quality, would be a good idea for the project. Other tools used during this study, including JMockit, and PITest can also be easily incorporated in the project, if path coverage and mutation are to be implemented going forward. The observations from this study also highlight that developers need to increase the testing effort, to increase the code coverage level with the increase in size or complexity of the software.

Future work

The topic of how test coverage can be improved, and how do various coverage models perform on the same code base in terms of quality, in general, is vastly studied and promising. My belief after having done this thesis work is that testing is a crucial aspect in projects of all domains and not only the fin-

tech domain, and thus the testing mechanisms will keep evolving based on the growing complexities in the software under test.

The continuation of the work presented in this thesis report is to continue exploring more complex coverage models on perhaps better covered code bases to establish the relationship between coverage and quality. In what cases do the coverage models such as branch and path coverage work better than line coverage will also be something to examine. One more aspect to look at can be if the relationship between coverage and quality is consistent to the findings of this study when the data set is from non-imperative languages such as Haskell, Erlang, or Prolog.

References

[1] Schmidt, Anatoly B. *Financial Markets and Trading*. 1st ed. Hoboken, N.J.: Wiley, 2013.

[2] Baker, Robert P. *The Trade Lifecycle: Behind the Scenes of the Trading Process*. 2nd ed. John Wiley & Sons, 2015.

[3] Lalor, John J. *Cyclopedia of Political Science, Political Economy, And Of the Political History of the United States*. 1st ed. Chicago: Rand, McNally, 1882.

[4] Ammann, Paul and Jeff Offutt. *Introduction to Software Testing*. Cambridge: Cambridge University Press, 2017.

[5] Jangra, Ajay, Nitin Goel, and Chander Kant. "When to Stop Testing". *High Performance Architecture and Grid Computing: International Conference HPAGC 2011*. Kurukshetra, India: Kurukshetra University, 2011. 626- 630.

[6] Yang, M.C.K. and A. Chao. "Reliability-Estimation and Stopping-Rules for Software Testing Based On Repeated Appearances of Bugs". *IEEE Transactions on Reliability* 44.2 (1995): 315-321.

[7] Elssamadisy, Amr and Jean Whitmore. "Functional Testing: A Pattern to Follow and the Smells to Avoid". *Pattern Languages of Programs*. Portland, Oregon, USA: PLoP, 2006. Article 27.

[8] Dahl, Ole-Johan, Edsger Wybe Dijkstra, and C. A. R Hoare. *Structured Programming*. 1st ed. London [etc.]: Academic Press, 1990.

[9] Goodenough, John B. and Susan L. Gerhart. "Toward A Theory of Test Data Selection". *International Conference on Reliable Software*. Los Angeles, California: IEEE Trans. Software Engineering SE-3, 1975. 493 - 510.

[10] Howden, W. E. "Reliability of the Path Analysis Testing Strategy". *International Conference on Reliable Software*. New Jersey, US: IEEE, 1976. 208 - 215.

[11] Weyuker, E. J. and T. J. Ostrand. "Theories of Program Testing and the Application of Revealing Sub-Domains". *IEEE Trans. Software Engineering*. NJ, USA: IEEE Press, 1980. 236 - 246.

[12] Whalen, Michael William et al. "Coverage Metrics for Requirements-Based Testing". *International Symposium on Software Testing and Analysis*. Portland, USA: ISSTA, 2006.

[13] Zhu, Hong, Patrick A. V. Hall, and John H. R. May. "Software Unit Test Coverage and Adequacy". *ACM Computing Surveys* 29.4 (1997): 366-427. Web.

[14] Bolton, Michael. "Context Driven Testing". *CAST 2011*. Seattle: CAST, 2011. Web. 9 May 2017.

[15] "The Nasdaq Story: History, Business, Awards". *Business.nasdaq.com*. N.p., 2017. Web. 9 May 2017.

[16] Shahid, Muhammad, Suhaimi Ibrahim, and Mohd Naz'ri Mahrin. "A Study On Test Coverage In Software Testing". *International Conference On Telecommunication Technology And Applications*. Singapore: IACSIT Press, 2011. Print.

[17] Cameron, A. Colin, and Frank A.G. Windmeijer. "An R-Squared Measure Of Goodness Of Fit For Some Common Nonlinear Regression Models". University of California, and University College London, 1995. Print.

[18] Mockus, Audris, Nachiappan Nagappan, and Trung T. Dinh-Trong. "Test Coverage And Post-Verification Defects: A Multiple Case Study". *International Symposium On Empirical Software Engineering And Measurement*. Florida: IEEE, 2009. Web. 10 Feb 2017.

[19] Gopinath, Rahul, Carlos Jensen, and Alex Groce. "Code Coverage For Suite Evaluation By Developers". *36Th International Conference On Software Engineering*. New York, USA: ICSE, 2014. Print.

[20] Gligoric, Milos et al. "Comparing Non-Adequate Test Suites Using Coverage Criteria". *2013 International Symposium On Software Testing And Analysis*. New York, USA: ISSTA, 2013. Print.

[21] "Continuous Code Quality | Sonarqube". *Sonarqube.org*. N.p., 2017. Web. 18 Feb 2017.

[22] "Eclemma - Jacoco Java Code Coverage Library". *Eclemma.org*. N.p., 2017. Web. 18 Feb 2017.

[23] "The Jmockit Testing Toolkit". *Jmockit.org*. N.p., 2017. Web. 19 Feb. 2017.

[24] "PIT Mutation Testing". *Pitest.org*. N.p., 2017. Web. 19 Feb. 2017.

[25] "Linear Regression". *Stat.yale.edu*. N.p., 2017. Web. 29 May 2017.

[26] Daka, Ermira, and Gordon Fraser. "A Survey On Unit Testing Practices And Problems". Ph.D. University of Sheffield, U.K., 2010. Print.

[27] Ellims, Michael, James Bridges, and Darrel C. Ince. "Unit Testing In Practice". *15Th International Symposium On Software Reliability Engineering*. IEEE, 2004. Print.

[28] Kochhar, Pavneet Singh et al. "An Empirical Study On The Adequacy Of Testing In Open Source Projects". Ph.D. Singapore Management Univesity, 2013. Print.

[29] N. Fenton and N. Ohlsson, “Quantitative analysis of faults and failures in a complex software system,” IEEE Transactions on Software Engineering, pp. 797–814, 2000.

[30] Y. W. Kim, “Efficient use of code coverage in large-scale software development,” in CASCON, 2003, pp. 145–155.