CHAPTER 1

INTRODUCTION AND BACKGROUND

1.1 Introduction

A major problem of software industry is its inability to develop bug free software. All errors are costly. Undetected errors will migrate downstream within the system to cause failures, which may result in serious financial and legal consequences and sometimes become threats to modern automated civilization. Hence software crisis has become a fixture of everyday life with many well publicized failures that have had not only major economic impact but also has become the cause of death for many human beings. Some of the major computer system failures caused by software bugs are: Explosion of European Space Agency's new Arian 5 rocket, Failure of London ambulance system, NASA Mars Climate Orbiter spacecraft loss, USS Yorktown incident, Accounting software failures, etc. The process of software testing and fault detection continues to challenge the software community.

Hetzel [60] describes testing as, "... the process of establishing confidence that a program or system does what it is supposed to do." Myers [19] defines testing as, "... the process of executing a program or system with the intent of finding errors." Software testing as defined by IEEE[152] is the process of operating a system or component under specified conditions, observing or recording the results, and making an evaluation of some aspect of the system or component.

Many of the software systems currently under development are highly interactive systems, which usually have a graphical user interface (GUI) or deal with multimedia (Real-Time)
The introduction of client-server, distributed systems, and now web based applications require specialized testing and have put a great deal of strain on current software testing methodologies.

Today's software development requires the production of high quality systems within a minimum development time. Often the time between product releases is short. This puts pressure on software testers to speed up the testing process while making sure that the software released is as reliable as economically possible. Hence there is an increasing need for effective testing of software systems.

A lot of work has been done in the field of software testing and the next section presents a brief history.

1.2 Brief History

In the early days of 1950s software development testing was regarded as "debugging", or fixing the known bugs in the software, and was usually performed by the developers themselves. By 1957 software testing was distinguished from debugging and became regarded as detecting the bugs in the software. But testing was still an after-development activity. By the 1970s "software engineering" as a term was used more often, though there was a little consequence as to what it really meant. The first formal conference on testing was held at the University of North Carolina in 1972, and a series of publications followed Hetzel[60], Myers[19]. The work of Meyers and others in 1970s was a major step in the development of satisfactory testing processes, but in the real world of industry, testing continued to be at the top of things that got dumped when the schedules and budgets became tight. By 1980s, “Quality” became the battle cry. During the 1980s, the Software Engineering Institute (SEI) at Carnegie Mellon University developed a software process assessment
method and capability maturity model for the United States Department of Defense. Their 1987 technical report proposed a five-level capability maturity model. In the 1990s, wide variety of testing tools to help in the testing process came into picture.

Despite the enormous advances in the last 50 years, the software testing process is still immature. Furthermore, the complexity and criticality of the problems which software is expected to solve have become greater, and platform complexities have become bigger. This is tending to out-run our ability to put more effective methods, which not only focus on the traditional testing (black box testing, white box testing etc.) but also emphasize on regression testing, performance testing and testing measurements.

Software maintenance activities account for two-thirds of the overall cost of software production. One necessary maintenance activity, regression testing, is performed on modified software to provide confidence that the software behaves correctly, and that modifications have not adversely impacted the software's quality. Regression testing is an expensive testing process [126]. A typical approach to regression testing is to save the test suites that are developed for a product, and reuse them to revalidate the product after it is modified. A typical regression test cycle would involve testing the entire test suite. The limited available calendar time does not allow thorough testing of all functions.

In fast moving E-business projects, risk taking is inevitable. Balancing testing against risk is essential because we never have the time to test everything. Software testers must become expert in risk and they must identify failure modes and translate these into consequences to the sponsors of the project. In this way, testers, managers and sponsors can reconcile the risks
being taken to the testing time and effort. Using risks to prioritize tests means that testers can concentrate on designing effective tests to find faults and not worry about doing too little testing and hence testing can be effective.

The extraordinary growth in the World Wide Web has been sweeping through business and industry. Many companies have developed or integrated their critical applications using Web technologies. As Web applications become complex, testing Web applications becomes crucial particularly performance. Performance testing is normally considered as a last stage in the development lifecycle. In some cases it is taken up as a reaction to customer complaints. This approach normally results in significant design changes in the final stages or a trade-off with the functionality or robustness of the application. Considering the factors like shorter delivery cycles, high costs of rework, competition etc., it has become necessary that the performance testing activities are properly planned and should hence run in parallel to the design and development activities. During performance testing, team faces the challenges in Workload Modeling, Load generation, simulation of the user behavior, etc. These performance-testing issues need to be examined and addressed.

Software testing has taken a new dimension in the recent past due to increasing complexity and functionality of software. In critical applications, the effort spent on testing can be as high as 35% to 45% of project effort. Hence it is necessary to define precise processes and process metrics for testing and manage the testing projects objectively. A good measure for testability and complexity can better manage the testing effort and time. Metrics like Average Cyclomatic Complexity (ACC), Response for a Class (RFC), Coupling Between Objects (CBO) and Depth Inheritance Tree (DIT) are used in measurement of object-oriented
testability and complexity but none of them is alone sufficient to give an overall reflection of software testability and complexity.

Underestimating a testing project leads to under-staffing it (resulting in staff burnout), under-scoping the quality assurance effort (running the risk of low quality deliverables), and setting too short a schedule (resulting in loss of credibility as deadlines are missed). Traditionally, estimation of efforts for testing has been more of a ballpark percentage of the rest of the development life cycle phases. This approach to estimation is more prone to errors and carries a bigger risk of delaying the launch deadlines.

1.3 Literature Survey

This section provides the details of the earlier work done in the field of Regression Testing, Performance engineering, Software Test Metrics and Software Test Size and Effort estimates.

Wong et al. [140] suggest prioritizing test cases according to the criterion of “increasing cost per additional coverage”. The authors restrict their attention to prioritization of the subset of test cases selected from a test suite by a safe regression test selection technique, and the selected test cases are only those test cases that reach modified code, but other test cases could be placed after these for later execution. No empirical results were reported on this prioritization technique.

Jones and Harrold [141] describe a technique for prioritizing test cases for use with the modified condition/decision coverage (MCDC) test adequacy criterion. This technique uses an iterative approach, updating test information as test cases are added to the ordered sequence. Their algorithm generates a list of ordered sequences of test cases.
Srivastava and Thiagarajan [142] present a technique for prioritizing test cases based on basic block coverage. In their algorithm, during each iteration, a test case that covers the maximum number of uncovered yet changed basic blocks of code is selected. When no new changed blocks can be covered, the set of covered blocks is reset, and a new sequence is started. The technique differs from previous techniques in that it computes flow graphs and coverage from binary code, and attempts to predict possible affects on control flow following from code modifications. The authors describe the application of this technique to several large systems at Microsoft, and provide data showing that the approach can be applied efficiently to those systems. The authors also provide data suggesting that their prioritized test case orders achieve coverage quickly, and can detect faults early; however, their studies do not compare their prioritized test suites to other prioritization techniques and random ordering, so it is not possible to say whether the results represent an improvement in rate of fault detection over those that would be obtained under other orderings.

Kim and Porter [143] present a technique, which they refer to as a “history-based prioritization” technique, in which information from previous regression testing cycles is used to better inform the selection of a subset of an existing test suite for use on a modified version of a system.

Avritzer and Weyuker [144] present techniques for generating test cases that apply to software that can be modeled by Markov chains, provided that operational profile data are available. Although the authors do not use the term “prioritization”, their techniques generate test cases in an order that can cover a larger proportion of the software states most likely to be reached
in the field earlier in testing, essentially, prioritizing the test cases in an order that increases the likelihood that faults more likely to be encountered in the field will be uncovered earlier in testing. Though not concerned with the prioritization of existing test cases for use in testing modified software, the approach provides an example of the application of prioritization in the case in which test suites are not available.

In [145], Rothermel et al. describe experiments with several C programs ranging in size from 138 to 6218 lines of code. Six prioritization techniques were studied including total statement coverage prioritization, additional statement coverage prioritization, total branch coverage prioritization, additional branch coverage prioritization, total fault-exposing-potential prioritization, and additional fault-exposing potential prioritization.

In [146], Rothermel et al. extend these early results. Similar to the earlier experiment, in this case, additional fault-exposing-potential prioritization achieved statistically significantly better results than other techniques.

Ramesh Radhakrishnan and Lizy K. John [149] paper studies the behavior of a modern Web server to understand the interaction with the underlying hardware and operating system. The study monitor and evaluate the performance of the system using hardware performance counters, for different workloads and identify the bottlenecks in the system based on the analysis of the collected data. The workloads include static requests (to HTML files and images) which form a major percentage of the web traffic as well as dynamic requests in the form of CGI (Common Gateway Interface) scripts and Servlets, both of which are gaining popularity and hence emerging to be a significant part of today’s Web traffic.
Xiaoping Jia and Hongming Liu [147] paper presents an approach for rigorous and automatic testing of web applications. This approach uses formal specifications of web applications to test functionality, security and performance. Using the formal applications on functionality, security and performance of a web application, a test engine can automatically generate test cases, perform the test, and validate the test result against the formal specifications. A prototype tool WebTest has been developed and successfully demonstrates the feasibility and effectiveness of the proposed approach.

Jeff Offutt [148] in his paper proposes the Quality Attributes of Web Software Applications. Three of the most important quality criteria for success of web applications are given as: Reliability, Usability and Security. An additional four important criteria are: Availability, Scalability, Maintainability and Time-to-market.

Connie U. Smith and Lloyd G. Williams [150] in their paper describe how to apply the techniques of Software Performance Engineering (SPE) to Web applications. The paper focuses on using the SPE models to provide decision support during the software architectural design phase because the decisions made at that time have the largest effect on performance and scalability.

Several measures for OO designs have been proposed (Li and Henry [86]; Chidamber and Kemerer [75]; Brito e Abreu [87]; Briand et al.[88]; Bieman and Kang [89]) and validated (Basili et al. [90]; Brito e Abreu and Melo [91]; Briand et al., [92,93,96,97]; Harrison et al [94,95]). For historical reasons, the “CK metrics suite” proposed by Chidamber and Kemerer [75] are the most frequently referenced OO-design measures. Chidamber and Kemerer [75] have proposed the following 6 metrics - Weighted Methods per class (WMC),
Response for a class (RFC), Lack of Cohesion of Methods (LCOM), Coupling Between Objects (CBO), Depth Inheritance Tree (DIT) and Number of Children (NOC), which measures the different software quality attributes like efficiency, complexity, understandability, reusability, maintainability and testability. The following OO software quality attributes can be measured:

- Efficiency -- are the constructs efficiently designed?
- Complexity -- could the constructs be used more effectively to decrease the architectural complexity?
- Understandability -- does the design increase the psychological complexity?
- Reusability -- does the design quality support possible reuse?
- Testability and maintainability -- does the structure support ease of testing and changes?

K.K. Aggarwal, Y.Singh & J.Chhabra [151] have shown how the fuzzy approach could be used to provide a unified measure of software maintainability and understandability.

Kathleen Peters [118], in his book defines estimation as a four-step process. Rubin [117] has given the stage-wise effort distribution. Regarding software development distribution effort, Grady [59] reports the following breakdown from the analysis of 125 Hewlett-Packard projects:

- Specification/Requirements 18%
- Design 19%
- Code/Unit Test 34%
- System/Integration Test 29%
Allen Albrecht of IBM proposed measuring size of a software application using Function Point Analysis methodology in 1979.

Capers Jones [111] estimates that the number of test cases can be determined by the function point's estimate [113] for the corresponding effort.

1.4 Objectives of Research

This thesis presents few methodologies and approaches for improving the effectiveness of software testing. The main objective of this research has been to look at the regression testing, performance testing, software testing measurements and software testing effort estimation.

- To identify the strategy for prioritization of the regression test suite based on the cost of risk associated with each test case. The test cases are executed based on the priority derived from the cost of risk in the given duration of time.

- To study about the Performance Testing process and different measurements captured and analyzed during the entire process. To arrive at a methodology for estimating ballpark percentage of total Software Development Life Cycle (SDLC) effort required for performance testing.

- To evaluate and address the issues faced by developers in terms of Workload Modeling, Load generator and simulation of the user behavior?, Load generator and simulation of the required number of users?, How to ensure that system can support more users? & End User Education. To arrive at certain quantitative
techniques that can be used in verifying performance and scalability of the web application. To arrive at performance optimization techniques.

- To study how process definition and process metrics help to consistently execute testing, measure quality and productivity of testing in order to improve the process. To arrive at unified measure of OO software testability and complexity.
- To find a methodology for coming out with the estimates for testing projects in terms of test case points.

1.5 Organization Of The Thesis

This chapter of the thesis presented an introduction to software testing and difficulties in testing. It presented an overview of the work done by different researchers in this field. It discussed the objectives of the research. It also presented a literature survey that was carried out with emphasis on regression testing, performance testing and testing measurements.

**Chapter 2** of the thesis discusses about the strategy for prioritization of the regression test suite based on the cost of risk associated with each test case. The technique strives to lower the costs without overly reducing the effectiveness by carefully prioritizing the test suite. The test cases are executed based on the priority derived from the cost of risk in the given duration of time. Risk exposure is calculated by considering the probability of fault occurrence and cost of fault. There are two aspects for cost estimation: one is cost to the customer and the other is the cost to the developer.

**Chapter 3** proposes a method for estimating the cost of fault from developer’s perspective using Neurofuzzy approach. The cost of fault for a development organization is based on the
effort required for rectifying the same. Also if the application delivered is faulty the development organization spends lot of effort in providing the maintenance resources.

Chapter 4 of this thesis discussed about a methodology for estimating ball-park percentage of total Software Development Life Cycle (SDLC) effort required for performance testing. The chapter also discussed about the Performance Testing process and different measurements captured and analyzed during the entire process.

There are several steps involved in performance testing – setting up the test environment, creation of test scripts using a COTS load testing tool, executing the scripts simulating a multi-user scenario and analyzing the test results. Chapter 5 of this thesis provides answers to these questions by reflecting on case studies of distributed applications and highlighting aspects that should be considered while answering these questions. In this chapter performance-testing issues are examined and we arrive at an approach to address these issues for certain classes of software systems.

Chapter 6 of this thesis proposes and details certain quantitative techniques that can be used in verifying performance and scalability of the web application. The proposed techniques are validated by following them during the performance testing of .NET architecture for a complex application with thousands of users. The chapter also discusses some of the performance optimization techniques.

Chapter 7 of this thesis discusses how process definition and process metrics help to consistently execute testing, measure quality and productivity of testing in order to improve the process.
Chapter 8 of the thesis illustrates a new integrated measure of testability of object Oriented software. The chapter combines OO software metric values into a single overall value (called Testability Index) that can be used to calculate the testability of a class using fuzzy techniques and concepts. We include empirical data of testing time of 25 different Java classes.

Chapter 9 proposes a fuzzy model for complexity measurement that integrates the effect of subset of Object Oriented Metrics (OO metrics) i.e. Weighted Methods per Class (WMC), Response for a Class (RFC), Coupling Between Objects (CBO), Depth Inheritance Tree (DIT). This model integrates the effects of different parameters of object-oriented systems, which can affect the complexity and is not biased towards one particular aspect of the system.

Chapter 10 of this thesis arrives at a methodology for coming out with the estimates for testing projects in terms of test case points. The proposed estimation methodology outlines major factors that affect testing projects and to ultimately help projects do an accurate estimation. This metric is technology independent and supports the need for estimating, project management & measuring quality and the target audience for using this approach would be anyone who would want to have a precise testing effort estimation technique for any given application under test.

The last chapter i.e. Chapter 11 includes the conclusion of the research work discussed in this thesis. The Chapter 11 also discusses the scope of improvement in the proposed areas.