# Evaluating Effectiveness of Software Testing Techniques with Emphasis on Enhancing Software Reliability


A thesis submitted on partial fulfilment of the
requirement for the degree of

**Doctor of Philosophy (Ph.D)**

*by*

**Umar Farooq**

P.G. Department of Computer Sciences
Faculty of Applied Sciences & Technology
University of Kashmir

*under the supervision of*

**Dr. S.M.K. Quadri & Dr. Nesar Ahmad**

*in*

**Computer Science**

March, 2012

# Declaration

This is to certify that the thesis entitled **"Evaluating Effectiveness of Software Testing Techniques with Emphasis on Enhancing Software Reliability"**, submitted by **Mr. Umar Farooq** in the *Department of Computer Sciences, University of Kashmir, Srinagar* for the award of the degree of **Doctor of Philosophy** in **Computer Science**, is a record of an original research work carried out by him under our supervision and guidance. The thesis has fulfilled all the requirements as per the regulations of the University and in our opinion has reached the standards required for the submission. The results embodied in this thesis have not been submitted to any other University or Institute for the award of any degree or diploma.

**Supervisor and Head**

**(Dr. S.M.K. Quadri)**
Department of Computer Sciences
University of Kashmir
Srinagar, 190 006

**Co-Supervisor**

**(Dr. Nesar Ahmad)**
Dept. of Stats. & Comp. Appl.
T. M. Bhagalpur University
Bihar, 812 007

Dated: 27-March-2012

# Abstract

Software testing is one of the most widely known and essential field in software engineering. The purpose of software testing is not only to reveal defects and eliminate them but also to serve as a tool for verification, validation and certification. Defection detection and increasing reliability are the two main goals of software testing. For decades, researchers have been inventing new techniques to test software. However, no testing technique will ever be a solution for all types of software defects. At present, we have very limited information of software testing techniques effectiveness and efficiency. Therefore, while researchers should continue to develop new testing techniques, they also need to deeply understand the abilities and limitations of existing techniques. We need to know what types of defects a particular technique can be expected to find and at what cost. We have to check whether testing technique effectiveness and efficiency depends on program to which it is applied, subject who applies it, the number of faults in the program or the type of faults in the program. However it is not sufficient if testing techniques are only compared on fault detecting ability. They should also be evaluated to check which among them enhances reliability.

The research in this thesis aims at evaluating software testing techniques in terms of effectiveness in detecting software defects, and the ability to increase the reliability of the software. The research in this thesis falls within empirical method research on the verification and validation process with a focus on software testing techniques evaluation. The work in this thesis links both research and practice and aims to continue building empirical knowledge in the field of software engineering.

The first part of this thesis surveys and analyzes empirical studies on evaluation of testing techniques. Issues with the current evaluation of software testing techniques are identified. Building upon this, we present an evaluation framework (a set of guidelines) for experiments which evaluate the software testing techniques.

In addition, we also proposed a uniform classification of software testing techniques and identified a set of factors which helps us in selecting an appropriate testing technique.

The second part of the thesis presents an experiment which evaluates and compares three defect detection techniques to evaluate the effectiveness of the software testing techniques in terms of defect detection. Moreover, we also evaluated the efficiency of these techniques. The dependence of the effectiveness and efficiency on the various parameters like programs, subjects and defects is also investigated in the experiment.

The third and final part of this thesis presents an experiment which evaluates and compares three defect detection techniques for reliability using a novel method. The efficiency of these techniques is also evaluated.

Our effort is to provide evidence that will help testing and research community to understand the effectiveness and efficiency of software testing techniques in terms of defect detection and reliability and their dependence on various factors. The ultimate goal of our work is to move software engineering from a craft towards an engineering discipline.

# Contents

# List of Figures

# List of Tables

# Glossary

**Defect** When the distinction between fault and failure is not critical, defect can be used as a generic term to refer to either a fault (cause) or a failure (effect).

**Detection** Detection refers to the observation that the programs observed behavior differs from the expected behavior.

**Effectiveness** The number of defects found by a technique

**Efficiency** The amount of time taken by a technique to find a defect. In other words, it also specifies number of defects found by a technique in an hour.

**Error** An incorrect or missing human action that result in software containing a fault (i.e. incorrect software)

**Failure** An inability of a system to perform its required functions within specified requirements or to perform in an unexpected way.

**Fault** An abnormal condition that may cause a reduction in, or loss of, the capability of a functional unit to perform a required function. It can also be defined as a requirements, design, or implementation flaw or deviation from a desired or intended state.

**Isolation** Isolation means to reveal the root cause of the failure in the program.

**Test case** A set of inputs, execution conditions, and a pass/fail criterion.

**Test case specification** It is a requirement to be satisfied by one or more test cases.

**Test obligation** A partial test case specification, requiring some property deemed important to thorough testing.

**Test or test execution** The activity of executing test cases and evaluating their results.

**Test Plan** A document describing the estimation of the test efforts, approach, required resources and schedule of intended testing activities.

**Test Procedure** A document providing detailed instructions for the execution of one or more test cases.

**Test suite** A set of test cases.

**Testing Technique** Different methods of testing particular features a computer program, system or product. Testing techniques means what methods or ways would be applied or calculations would be done to test a particular feature of software.

**Validation** The process of evaluating the to determine whether it satisfies specified customer requirements.

**Verification** The process of evaluating the software to determine whether it works according to the specified requirements.

# Chapter 1

# Introduction

## 1.1 Introduction

Our dependence on software is continuously increasing as software is now playing a primary role in present day systems. These software intensive systems demand high quality software as software failures cost us both in terms of money, time and other resources. In addition, it has become a matter of reputation for organizations to produce quality products, especially when safety of people or environment is concerned. A recent example of such a case is the Toyota Company, which issued a recall of approximately 133,000 Prius and 14,500 Lexus vehicles to update software in the vehicle's antilock braking system [Malik et al., 2010]. There are many ways to evaluate and control the quality of a software; however software testing still dominates the other methods as it is used as the primary method for quality assurance and quality control in software industry. Software testing is a process of verifying and validating that a software application or program meets the business and technical requirements that guided its design and development and works as expected; it also identifies important errors or flaws categorized as per the severity level in the application that must be fixed [Bentley, 2005]. Software testing is a broad field and has evolved over the time. Software testing is no longer seen as an activity that starts only after the coding phase is complete with the limited purpose of detecting failures only. Software testing is also used to test the software for other software quality factors like reliability, efficiency, portability, maintainability, compatibility etc. Software testing constitutes a major part of software development lifecycle.

Despite all the efforts people put in the quality of the software, effectiveness of testing remains lower than expectations. According to [Bertolino, 2007] testing is a widespread validation approach in industry, but it is still largely ad hoc, expensive, and unpredictably effective. Inadequate testing has resulted in many software related problems in past and have actually brought social problems and financial losses. A solution to this problem is to test a system exhaustively. However we are often faced with lack of time and resources which can limit our ability to effectively complete testing efforts, thus ruling out exhaustive testing. For this reason, Dijkstra stated that testing can be used to show the presence of errors, but never to show their absence [Dijkstra, 1970]. Hence, our aim is that the testing process should find maximum defects possible. For that purpose, we have to make a choice between available testing techniques as we would like to use only few not all to test our system. We would like to select proper and effective testing techniques which will increase test effectiveness to maximum extent that too very efficiently, keeping in view the limited resources available for testing. However, making this choice is next to impossible task, because we do not have

adequate information about relative effectiveness, efficiency and cost of testing techniques. To obtain this kind of information is not easy given the variability of their operations; which depends on the subject that applies it, the programming language, software under test, the type of faults etc.

Current studies suggest that we should use diverse testing techniques for testing the programs. They argue that using diverse techniques works on different aspects and as such targets different types of defects. But using numerous techniques to carry out testing means excessive use of resources (less efficiency), as it clearly involves more test cases, time and resources. Moreover, many techniques may target same types of defects resulting in the duplication of the efforts which again implies wastage of resources. So, we should evaluate and compare software testing techniques for their effectiveness and efficiency. Even though some advances have been made in evaluating effectiveness, efficiency of testing techniques but there is still a long way to go as results are very inconclusive and contradictory. Another important thing to note is that most of the experiments conducted so far are focused only on evaluating software testing technique's fault finding effectiveness and efficiency and not much attention is paid to evaluate software testing techniques on software reliability improvement criterion as producing dependable software is also one of the main objectives of software testing.

So, we argue that there is a need to evaluate software testing techniques not only for effectiveness and efficiency of finding faults but also for the ability of enhancing software reliability. Our goal is not to find a single technique that will supersede the rest of the testing techniques at all fronts but we would like to identify conditions under which a technique helps us to detect the most defects (i.e., maximum effectiveness) and conditions under which the technique helps us to detect defects most rapidly (i.e., maximum efficiency). In addition to that, the dependence of effectiveness of testing techniques on the program to which it is applied, subject who applies it, the number of faults in the program, types of faults also need to be verified.

The rest of this chapter is organized as follows: Section 1.2 explains the motivation behind this work, Section 1.3 sets the goals and objectives. Section 1.4 presents the research methodology adapted followed by Section 1.5 which presents contributions made in this work and finally, Section 1.6 gives an outline of this theses.

## 1.2 Research Motivation

Software testing is a complex and critical task in software development life cycle. Testing methods and techniques, tools, standards, measurements, and empirical knowledge etc. are the main elements of interest in the software testing domain. The area of software testing research is almost as old as the software engineering itself. It has largely been driven by the quest for quality software. [Harrold, 2000] [Taipale et al., 2005] [Bertolino, 2007] have discussed about the future research developments in software testing. The evolution of definition and targets of software testing has directed the research on testing techniques which includes the creation, refinement, extension, and popularization of better testing methods and techniques. Although, all the software testing techniques have one important thing in common i.e. they aid us in achieving goals of software testing. However, we know that software testing has diverse goals like verification and validation, reliability improvement etc. One testing technique or method is not sufficient enough to achieve all the goals of software testing. Different techniques help us in attaining different goals of software testing as different testing techniques serve the different purposes. In addition to that, techniques differ from one another in number of other ways like the purpose for which it is used, test data selection criteria, the phase in which it is used etc. At the same time many software testing techniques serve the same purpose or are similar in many aspects, for example, the information they need to generate test cases (code or specification), the criteria for selecting test cases, the aspects of program to be examined by test cases etc. Using the techniques which serve the same purpose or target same type of faults is superfluous, which will cause wastage of resources and time. [Beizer, 1990] stresses the importance of proper selection of testing techniques; he claims that the selection of an unsuitable technique can lead to an inappropriate set of test cases, which will bring with it an inaccurate (if not erroneous) evaluation of the software aspect being tested. So, in that case,s we would like to choose the most effective technique. The importance of selecting an appropriate software testing technique is widely accepted in the software testing community. [Howden, 1978] refers to the fact that a very common testing problem is the lack of well-defined techniques for selecting proper test cases. A lot of studies have focused on the importance of the information about testing techniques so that we can decide which one we should use in a given context [Kamsties and Lott, 1995] [Wong and Mathur, 1995a].

A lot of research has addressed the evaluation of the relative effectiveness of the various test criteria and especially of the factors which make one technique better than another at

fault finding but there are no clear-cut results yet. Studies carried out so far do not examine the conditions of applicability of a technique at length or assess the relevant attributes for each technique. There is no study which can tell us much about the relative benefits and limitations of testing techniques. As a result of that, one technique do not supersede other techniques on all fronts, thereby creating ambiguity in test technique selection. At present, we do not have an exact idea of what methods and techniques are available and of all the practical information of interest about every testing technique. Recent surveys on comparisons of various software testing techniques also concludes that further empirical research in software testing is needed and that much more replication has to be conducted before general results can be stated. "Demonstrating effectiveness of testing techniques" was in fact identified as a fundamental research challenge in FOSE2000 and FOSE2007, and even today this objective calls for further research, whereby the emphasis is now on empirical assessment [Bertolino, 2007].

So there is a need to further evaluate software testing techniques but the evaluation should be carried out in such a way that the results could be realistic and implementable. Besides evaluating software testing techniques for fault finding effectiveness and efficiency, we should also evaluate software testing techniques for reliability as producing dependable software is also one of the main objectives of software testing. In this research, we aim to evaluate software testing techniques for effectiveness and reliability. The problem addressed in this research is another step towards building a knowledge base which can help testing community to choose an effective and efficient testing technique.

## 1.3 Research Goals & Objectives

The ultimate goal of this research is to evaluate the software testing techniques. The research statement is as:

*Evaluating effectiveness of software testing techniques with*

*emphasis on enhancing software reliability*

The factor evaluated in this research is effectiveness of software testing techniques. In addition, we also aim to evaluate software testing techniques for reliability. The research focus is on two things: fault detection and reliability. The research goal leads us to research objectives which are stated as follows:

1. Identifying the current research status about available testing techniques evaluation and also to outline an approach to evaluate software testing techniques effectiveness.

2. To evaluate software testing techniques for effectiveness in terms of fault detection ability. The goal is to investigate the effectiveness and its dependence on the program to which it is applied, subject who applies it and the faults in the program.

3. To evaluate software testing techniques for reliability. The goal is to investigate which software testing technique is effective in enhancing software reliability.

## 1.4 Research Methodology

Two types of studies can be carried out to evaluate the relative effectiveness of software testing techniques and provide information for selecting among them: *analytical* and *empirical*.

An analytical solution would describe conditions under which one technique is guaranteed to be more effective than another, or describe in statistical terms relative effectiveness of software testing techniques. Analytical studies can produce more generalized results, i.e., results which are not tied to a particular experimental perspective. However, analytical studies remain so far quite theoretical in nature: they are highly useful to enlarge our knowledge behind testing techniques but provide little practical guidance in selecting a test technique. The reason is that the conclusions provided by such analytical comparisons are based on assumptions that are far beyond what one can reasonably expect to know or even hypothesize about a program under test [Bertolino, 2004].

Empirical solutions are closer to a practitioner's mindset in which measures from the observed experimental outcomes are taken. An empirical solution would be based on extensive studies of the effectiveness of different testing techniques in practice, including controlled studies to determine whether the relative effectiveness of different testing methods depends on the software type, subject who test it, the type of faults in the software, the kind of organization in which the software is tested, and a myriad of other potential confounding factors [Young, 2008]. Empirical approaches to measuring and comparing effectiveness of testing techniques are still in its infancy. A major open problem is to determine when, and to what extent, the results of an empirical assessment can be expected to generalize beyond the particular programs and test suites used in the investigation.

However, at present the trend in research is toward empirical, rather than theoretical comparison of the effectiveness of testing techniques. Directly observed results can sound more

realistic and convincing than mathematical formulas built on top of theoretical assumptions. Empirical studies have displaced theoretical investigation of test effectiveness to a large extent. Carrying out empirical work to understand the problems of software engineering is held to be of increasing importance [Fenton, 1994] and [Gibbs, 1994]. Over the decades, the importance of empirical studies in software engineering has been emphasized by several researchers [Basili et al., 1985], [Fenton et al., 1994], [Tichy, 1998], [Wohlin et al., 2000] and [Kitchenham et al., 2002].

There are, however, inherent problems in drawing conclusions from single studies, especially those with human subjects. For this reason, research community demand that experimental results are externally reproducible - that is, an independent group of researchers can repeat the experiment and obtain similar results. Successful replications enable a discipline's body of knowledge to grow, as the results are added to those of earlier replications. Replications are commonly considered to be important contributions to investigate the generality of empirical studies. By replicating an original study it may be shown that the results are either valid or invalid in another context, outside the specific environment in which the original study was launched. The results of the replicated study show how much confidence we could possibly have in the original study. In the last few years, the importance of replicating research studies has received growing attention in the empirical software engineering community. A finding cannot be established as the "truth", based on a single study, since small variations in the execution of a study can have a large effect on the results. In general, however, testing techniques are heuristics and their performance varies with different scenarios; thus, they must be studied empirically [Do et al., 2005]. In this thesis we have used an empirical research methodology.

## 1.5 Contributions

In this thesis, we thoroughly investigated and evaluated software testing techniques. The major contributions are listed as follows:

1. We surveyed existing studies on software testing techniques. The goal was to identify the major issues in software testing technique evaluation studies. Thereafter we proposed a set of guidelines which aim at mitigating the issues identified.

2. We empirically evaluated three testing techniques with respect to fault detection ability and compared them in terms of effectiveness taking into consideration the proposed

guidelines. In addition, the role of program and subjects was also examined. We also evaluated software testing techniques for efficiency and different fault types and classes.

3. Lastly, we evaluated three systematic testing techniques for reliability using a novel method. The goal was to check the potential of each technique to reduce risk in the software.

In addition to these major contributions, this thesis also makes following contributions.

1. A uniform classification of software testing techniques which classifies testing techniques from the root and classifies them according to the basic information like to which family the techniques belongs, the information they require and the aspect of code they examine.

2. A decision support approach for selecting the most suitable testing technique is presented which is based on number of identified factors that influence the selection of appropriate techniques.

## 1.6 Outline

This thesis is structured around these three research objectives stated in section 1.3.

Chapter 2 discusses the software testing techniques evaluation. Firstly, it proposes a uniform classification of software testing techniques. It then surveys the existing research on software testing techniques evaluation. It then focuses on identifying the issues with software testing techniques evaluation. In order to mitigate the issues identified, a framework is proposed for evaluation of software testing techniques.

Chapter 3 discusses evaluation of software testing techniques. The defect detection techniques are evaluated empirically for effectiveness and efficiency using a controlled experiment. The chapter begins by surveying the studies carried out so far to evaluate testing techniques. Finally, it discusses the execution of the experiment and the results of the experiment.

Chapter 4 discusses evaluation of software testing techniques for reliability. It begins with presenting a novel approach for evaluating the software techniques for reliability. On the basis of that approach we execute an experiment which evaluates three defect detection techniques for reliability followed by its evaluation.

Chapter 5 summarizes the major findings of this work and the contribution to knowledge made in this dissertation. Moreover, it also presents the future scope of this work which can be investigated in further research.

Some of the material presented in this thesis has been published previously. The complete list of these published articles immediately follows Chapter 5.

End

# Chapter 2

# Software Testing Techniques Evaluation: Issues and Mitigation

## 2.1 Introduction

Verification and validation activities are conducted to evaluate and enhance product quality throughout the entire cycle of software development. Verification aims at checking that the system as a whole works according to its specifications and validation aims at checking that the system behaves according to the customers' requirements. Verification and validation is a generic term which includes software testing. Software testing is an important phase in software development life cycle. Almost 30-60 % resources are spent on the testing. Despite all the efforts people put in the quality of the software, effectiveness of testing remains lower than expectations. Testing is a widespread validation approach in the industry, but it is still largely adhoc, expensive, and unpredictably effective [Bertolino, 2007]. Unfortunately the lack of understanding of software testing usually leads to incomplete testing work.

Verification and validation includes both static as well as dynamic software testing techniques. The choice of a software testing technique in software testing influences both process and product quality. Taking into account current testing problems and failure consequences using the most effective and efficient testing methods is most important need in testing. We have multitude of software testing techniques which makes testing technique selection a complex choice. When choosing a testing technique, practitioners want to know which one will detect the faults that matter most to them in the programs that they plan to test [Goodenough and Gerhart, 1975]. Which techniques should be chosen? Are there any particular benefits of using a specific technique? Which techniques are effective? Which are efficient? All testing techniques can reveal faults; but how effectively they do that and what kind of faults they find, how much resources they utilize, by which factor they increase the reliability, we do not have an exact answer to such questions. Although the utilization of these techniques is growing, we have very limited knowledge about their relative quantitative and qualitative statistics. At present, mostly selection of testing techniques is done neither systematically, nor following well-established guidelines. Despite the large number of studies which attempt to study the effectiveness and efficiency of testing techniques and its allied factors and conditions, we have very limited knowledge on testing techniques effectiveness as they are not complete in all respects and vary significantly in terms of parameters they have taken into consideration. Additionally, existing studies show contradictory results. There is no "silver bullet" testing approach and that no single technique alone is satisfactory has been pointed out by many leading researchers [Chu, 1997]. Therefore we should use a combination of several techniques to test a software. However, we should select appropriate techniques

which will target different types of defects. However, many testing techniques belong to same group and as such will target same types of defects in the program. So we should use a best candidate from each group to test a program but do we have knowledge of relative effectiveness of techniques in a group? We guess, no!

It is obvious that testing technique selection and evaluation remains a key issue in software testing. So the need of the hour is to find the effective and efficient software testing techniques which can attain the goal of testing to maximum possible extent while consuming fewer resources. It will be nice if we can first go for intra-family comparisons, then we can go inter-family comparisons; but a lack of uniform classification makes comparison process more complicated. One classification places a particular technique in one family whereas other classification places the same technique in some other family. In effect we should develop a uniform classification of software testing techniques, then only can be comparison meaningful and worthy as there will be no ambiguity related to the technique and the group to which it belongs. As we know no single study can be complete and perfect in all respects, we cannot really expect one testing technique to supersede all other techniques. There is often a specific interest or purpose of evaluating a particular test technique, based on the assumption that the technique will be more effective. Regardless of our technique, it could be wise to try to understand what types of defects a particular technique can be expected to find and at what cost. We have to check whether testing technique effectiveness and efficiency depends on program to which it is applied, subject who applies it, the number of faults in the program or the type of faults in the program. However it is not sufficient if testing techniques are only compared on fault detecting ability. They should also be evaluated to check which among them enhances reliability. To establish a useful theory for testing, we need to evaluate existing and novel testing techniques not only for defect detection effectiveness and efficiency but also for their ability of enhancing software reliability. In this chapter we describe why there is need to evaluate effectiveness of software testing techniques, problems with the existing studies and a framework is proposed for carrying out such studies so that in future such studies show results which can be useful for testing professionals in particular and testing industry in general.

The chapter is organized as: Section 2.2 gives us a brief description of software testing techniques and also proposes a uniform classification of software testing techniques in Section 2.2.1. Section 2.3 explains why we should evaluate software testing techniques. Section 2.4 surveys the existing research on software testing techniques evaluation. Evaluation results of the surveyed studies are presented in subsection 2.4.1 and the problems and shortcomings

are presented in subsection 2.4.2. Current status of testing technique selection is presented in Section 2.5. Section 2.6 describes how to choose a testing technique in the absence of concrete knowledge about the testing techniques. Section 2.7 proposes a framework (s set of guidelines) which should be taken into consideration while evaluating testing techniques effectiveness. Section 2.8 presents the conclusions and future work.

## 2.2 Software Testing Techniques

Software testing techniques are diverse methods to do software testing. We test software by selecting appropriate testing technique and applying them systematically. Testing techniques refer to different methods of testing particular features a computer program, system or product. By a testing technique, we mean a method or approach that systematically describes how a set of test cases should be created (with what intention and goals) keeping into consideration possible rules for applying the test cases. Testing techniques determine what ways would be applied or calculations would be done to test a particular feature of software. Testing techniques aids in limiting the number of test cases that can be created, since it will be targeting a specific type of input, path, fault, goal, measurement etc. [Eldh, 2011]. Test techniques provide an understanding of the complexities imposed by most systems. Using testing techniques, we can reproduce tests, as it paves the path for creating a test ware. Some techniques are easy to apply while other techniques require a little experience and knowledge before they can be used. The beauty of software testing techniques is that the more you use each the greater insight you will gain. You will understand when to use them, how to implement them, how to execute them, and will have a clear knowledge of which ones to use in any given situation. But before utilizing different testing techniques in proper manner for an appropriate purpose, we should have a profound theoretical knowledge of these testing techniques. In effect, we will study available software testing techniques and classify them according to underlying mechanism which includes what information they require, the purpose for which they can be used, when to use the. We will not describe the features of testing techniques in detail, as this information can be gathered from the classical literature on testing techniques, for example [Beizer, 1990] [Myers et al., 2011].

### 2.2.1 Proposed Software Testing Techniques Classification

Existing software testing practices are divided into two main categories: *static testing* and *dynamic testing*. Accordingly we can divide software testing techniques into two main categories

according to the criterion whether the technique requires actual execution of the software or not: static testing techniques and dynamic testing techniques [Roper, 1995]. Static testing covers the area of performing all kinds of tests on the software related documents or source code of software without executing it. Dynamic testing covers the area of performing all kinds of tests on the object code and executable that is determined from the source code by executing it. The difference between these types of testing is usually determined by the state of the software program (source code vs. executable). The other difference is that static testing techniques can be used from requirement phase to implementation phase, dynamic testing techniques can be applied from implementation phase onwards only. Another difference is that while static testing looks for faults, dynamic testing looks for failures. Software Verification and Validation use both static and dynamic techniques for system checking to ensure that the resulting program satisfies its specification and that the program as implemented meets the expectations of the stakeholders.

#### 2.2.1.1 Static testing techniques

Static testing/Non-execution based techniques focus on the range of ways that are used to verify the program without reference to actual execution of the program. Static techniques are concerned with the analysis and checking of system representations such as the requirements documents, design diagrams and the program source code, either manually or automatically, without actually executing the code [Sommerville, 2007]. Techniques in this area include code inspection, program analysis, symbolic analysis, and model checking etc. Documentation in the form of text, models or code are analyzed, often by hand. In a number of cases, e.g. in the compilation of program code, tools are used [Graham and Van Veenendaal, 2008].

Static testing techniques are also classified according to criterion whether or not the technique requires any automatic tool in addition to human analysis. If so, the techniques are grouped under automatic static testing otherwise it is manual static testing. Automatic testing is the evaluation of program or program related documents using software tools. In automatic static testing usually we input code to a static analysis tool for evaluating it for quality. In addition to code, documentation of the program can also be used in quality check of the program. Manual static testing evaluates the program and program related documents manually without the assistance of any tool. Manual static testing usually include the review of source code, program documents etc.

#### 2.2.1.2 Dynamic testing techniques

Dynamic Testing / Execution based techniques focus on the range of ways that are used to ascertain software quality and validate the software through actual executions of the software under test. We test the software with real or simulated inputs, both normal and abnormal, under controlled and expected conditions to check how a software system reacts to various input test data. It is essential to test the software in controlled and expected conditions as a complex, non deterministic system might react with different behaviors to a same input, depending on the system state. The dynamic testing of a software product implies execution, as only by studying the result of this execution is it possible to decide whether or not (or to what extent) the quality levels set for the dynamic aspects evaluated are met.

Dynamic testing techniques are generally divided into the two broad categories depending on a criterion whether we require the knowledge of source code or not for test case design: if it does not require knowledge of the source code, it is known as black box testing otherwise it is known as white box testing [Sommerville, 2007] [Beizer, 1995]; which correspond with two different starting points for dynamic software testing: the requirements specification and internal structure of the software. Black Box testing gives us only the external view (behavior) of the software as it concentrates on what the software does and is not concerned about how it does it. Testing techniques under this strategy are totally focused on testing requirements and functionality of the software under test. We need to have thorough knowledge of requirement specification of the system in order to implement the black box testing strategy. In addition we need to know how the system should behave in response to the particular input. White box testing strategy deals with the internal logic and structure of the code and depends on the information how software has been designed and coded for test case design. The test cases designed based on the white box testing strategy incorporate coverage of the code written in terms of branches, paths, statements and internal logic of the code, etc. White box testing strategy is focused on examining the logic of the program or system, without concerned about the requirements of software which is under test. The expected results are evaluated on a set of coverage criteria. We need have knowledge of coding and logic of the software to implement white box testing strategy.

#### 2.2.1.3 Test data selection criteria

Dynamic testing involves selecting input test data, executing the software on that data and comparing the results to some test oracle (manual or automatic), which determines whether

the results are correct or not. We need to do test the software on the entire input domain (exhaustive testing) to be sure that software is totally error free. As a matter of fact, due to the inherent discontinuous nature of software, we cannot infer any property from an observation on some point to other points in the input domain [Chu, 1997]. However the problem is that, excluding trivial cases, the input domain is usually too large which makes it impossible for us to go for exhaustive testing. So should we keep the software untested? Well, that thing can haunt us for years. Instead what we can do is that we will select a comparatively small subset which is in some sense representative of the entire input domain and will test the software for that selected subset. From this, we then can infer the behavior of software for the entire input domain. Therefore, dynamic testing corresponds to sampling a certain number of executions of software from amongst all its possible executions. Ideally, the test data should be chosen so that executing the software on this subset will uncover all errors, thus guaranteeing that any software which produces correct results for the test data will produce correct results for any data in the input domain. However, discovering such an ideal set of test data is in general an impossible task [Rapps and Weyuker, 1985] [Bertolino, 1991]. The identification of a suitable sampling strategy is known as the test data selection problem. The selection of test data is largely dependent on the testing purpose and type of testing technique used. There are five basic strategies for selecting test data of dynamic testing techniques:

1. **Random Testing:** In random testing, we select test inputs from the possible input domain randomly without any bias.

2. **Statistical Testing:** Test data is selected according to the distribution as is expected when the software will be in actual use.

3. **Functional Testing:** Test data set is selected according to the specified functions of the program, so that all functions and sub functions are tested at least once.

4. **Structural Testing:** Test data is selected according to the structural specification which aims to get the required coverage for the specified coverage items (statements, branches or paths). The structural tests encompass control flow (program flow) and data flow. Control Flow Testing (CFT) and Data Flow Testing (DFT) help with interactions along the execution path and interactions among data items in execution respectively.

5. **Mutation Testing:** Test data is selected based on the survived mutants in the program.

The first three testing methods are included in black box testing strategy, whereas the last two are include in white box testing strategy. A large number of testing techniques fall within each of the above listed testing methods. The proposed classification of software testing techniques is shown in Figure 2.1 . Effective testing should find greatest possible number of errors with manageable amount of efforts applied over a realistic time span with a finite number of test cases. We have to make sure that we select technique(s) that will help to ensure the most efficient and effective testing of the system. However, the fundamental question is what would be the techniques that we should adopt for an efficient and effective testing

## 2.3 Why to Evaluate Software Testing Techniques?

Software testing should be effective enough to prevent critical damages on the whole system for users, by taking into consideration of potential failures of the program and its environments [Kurokawa and Shinagawa, 2008]. One way to avoid such failures is to go for exhaustive testing of the system, which tests the system with all possible combinations of inputs which includes both valid and invalid cases. However, excluding trivial cases, exhaustive testing is an impractical thing for the most software systems. Besides, we are often faced with lack of time and resources, which can limit our ability to effectively complete testing efforts. A tester do not want to go for exhaustive testing, rather he wants to select a testing technique in relation to the selected test strategy that will detect maximum possible faults and brings the product to an acceptable level while consuming less resources and time. Whether we opt for static or dynamic testing, there is a selection of testing methods to choose from. In each testing method there are so many testing techniques that are used to test a system. Each testing technique meant for testing has its own dimensions i.e. for what purpose it is used, what aspect it will test, what will be its deliverables etc. Different approaches to software development require different testing methods and techniques [Tawileh et al., 2007]. This limits our ability to use a generic technique for testing a system. So at present we prefer to use variety of testing techniques to test a system as it will ensure that a variety of defects are found, resulting in more effective testing. But how long will we use numerous testing techniques to test software. Going this way means excessive use of resources (less

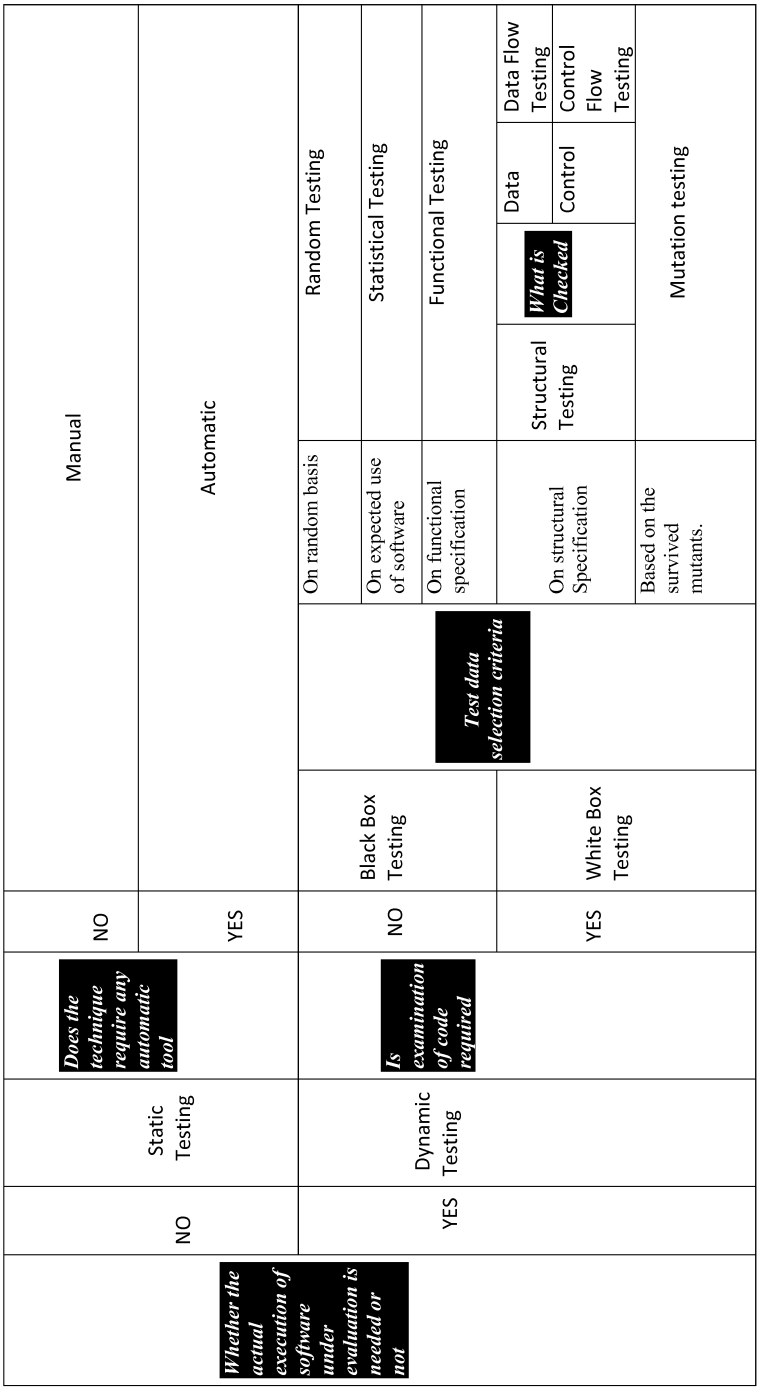



**Figure 2.1:** Proposed Classification of Software Testing Techniques

efficiency), as using many testing techniques clearly implies more test cases, more time and more resources. So the need is to select appropriate testing techniques which can make testing process effective and efficient. However, for a given testing problem, there exist several techniques of the same kind which differ by the underlying mechanism. For instance, several regression testing techniques are available; they belong to same family, yet they follow a different way to solve the problem at hand. Contrary to this are techniques which belong to different groups and also exploit totally different set of information for the purpose; for example, the control-flow and data-flow based techniques derive test cases quite differently.

Among so many techniques, which are in a competition we would like to select a technique that will detect the maximum possible significant defects, while consuming less resources and time. Unfortunately, it is not known which testing technique to select as we do not have adequate information about relative effectiveness, efficiency and cost of testing techniques. [Farooq and Quadri, 2010] also states that we do not have all the information of interest about every testing technique. This kind of information can only be obtained by evaluating software testing techniques. It is also necessary to understand what types of defects a particular technique is expected to find and at what cost. We also need to analyze testing technique dependences on program to which it is applied, subject who applies it, the number of faults in the program or the type of faults in the program. We should evaluate testing techniques to know about the relative merits and limitations of each testing technique, so that we are able to use it in appropriate scenario and for appropriate purpose. This information is useful before one has to implement a given testing technique; it is also useful (as a post mortem analysis) when one is finished with testing as this post-implementation assessment and analysis is needed for subsequent improvement of the technique to increase its effectiveness [Farooq and Dumke, 2008].

## 2.4 Existing Research on Software Testing Techniques Evaluation

A lot of research has been carried out concerning the evaluation of software testing techniques. By tracing the major research results that have contributed to the growth of software testing techniques we can analyze the maturation of software testing techniques research. We can also assess the change of research paradigms over time by tracing the types of research questions and strategies used at various stages [Luo, 2001]. Three directions of research have been found related to evaluation of testing techniques:

1. Actual evaluations and comparisons of testing techniques based either on analytical or empirical methods,

2. Evaluation frameworks or methodologies for comparing and/or selecting testing techniques.

3. Surveys of empirical studies on testing techniques which have summarized available work and have highlighted future trends.

Many experiments and case studies have been conducted so far towards the goal of evaluation of testing techniques. Some relevant works for controlled experiments are [Hetzel, 1976], [Myers, 1978] [Basili and Selby, 1987], [Weyuker, 1990], [Bieman and Schultz, 1992], [Frankl and Weiss, 1993], [Hutchins et al., 1994], [Offutt and Lee, 1994], [Kamsties and Lott, 1995], Wong and Mathur [1995b], Offutt et al. [1996], [Frankl et al., 1997], [Roper et al., 1997], [Frankl and Iakounenko, 1998], [Rothermel and Harrold, 1998], [Wong et al., 1997], [Vokolos and Frankl, 1998], Rothermel et al. [1999], [Elbaum et al., 2000], [Kim et al., 2000], [Bible et al., 2001], [Graves et al., 2001], [Juristo and Vegas, 2003] and many more. There are also some case studies which studied and evaluated different testing techniques which include [Wohlin et al., 2000], [Aurum et al., 2002], [Beck, 2003], [Host et al., 2005]. In case of frameworks and methodologies [Vegas and Basili, 2005] describe a characterization scheme for experiments which is specific for software testing techniques. The schema is similar to [Basili et al., 1985] but adapted to deal with evaluating testing techniques. [Do et al., 2004] and [Do et al., 2005] define the SIR (Software Artifact Infrastructure Repository) infrastructure to support controlled experiments with software testing techniques. The main contribution of their work is a set of benchmark programs that can be used to evaluate testing techniques. [Eldh et al., 2006] describes a straightforward framework for the comparison of the efficiency, effectiveness and applicability of different testing techniques based on fault injection or seeding. [Vos et al.] also defines general methodological framework for evaluating software testing techniques, which focuses on the evaluation of effectiveness and efficiency, but the framework is very preliminary and needs significant improvement. A significant survey of the software testing techniques can be found in [Juristo et al., 2004] which surveys the empirical studies in the last 3 decades.

Our focus in this thesis is on empirical evaluation of testing techniques, so we will have a look at empirical studies conducted so far to evaluate testing techniques. During the past few decades, a large number of empirical evaluations of numerous testing techniques have been

executed to compare various software testing techniques. The research on the comparison of testing technique traces back to as early as 35 years ago with Hetzel making a start in 1976 by conducting a controlled experiment in order to analyze three defect detection methods [Hetzel, 1976]. The empirical research on testing techniques is largely carried out through experiments comparing different dynamic testing techniques with each other or with different types of static testing techniques usually some reading technique. Most of the experimental studies are performed in a unit level testing context. A laboratory setting with student subjects is the most common design in the existing experiments. The most commonly studied factors in the experiments evaluating testing techniques are their effectiveness (i.e., number of detected defects) and efficiency (i.e., effort required to apply the technique) in programs. [Juristo et al., 2004] identified two classes of evaluation studies on testing techniques; *intra-family* and *inter-family*. Based on distinction made by [Juristo et al., 2004], major intra family and inter family studies carried till date to evaluate software testing techniques are listed in Table 2.1 and Table 2.2. In addition to these, many studies are in progression, but their results are preliminary. So they are not presented in the list/table.

**Table 2.1:** Intra-Family Comparisons

| Technique | Study | Year |
|---|---|---|
| Data-flow testing techniques | Weyuker | 1990 |
| | Bieman and Schultz | 1992 |
| Mutation testing techniques | Offut and Lee | 1994 |
| | Wong and Mathur | 1995 |
| | Offut et al. | 1996 |
| Regression testing techniques | Rothermel and Harrold | 1998 |
| | Vokolos and Frankl | 1998 |
| | Kim et al. | 2000 |
| | Bible et al. | 2001 |
| | Graves et al. | 2001 |

### 2.4.1 Evaluation Results

Summarizing the results of the studies conducted to evaluate the effectiveness and efficiency of software testing techniques. We observed that studies unfortunately have a lot of contradiction in terms of their results. There is no study which can tell us much about the

**Table 2.2:** Inter-Family Comparisons

| Comparison Groups | Study | Year |
|---|---|---|
| Control-flow, data-flow and random techniques | Frankl and Weiss | 1993 |
| | Hutchins et al. | 1994 |
| | Frankl and Iakounenko | 1998 |
| Functional and structural techniques. Note: All studies also compare manual static testing technique Code Reading with functional and structural techniques. | Hetzel | 1976 |
| | Myers | 1978 |
| | Basili and Selby | 1987 |
| | Kamsties and Lott | 1995 |
| | Roper et al. | 1997 |
| | Juristo and Vegas | 2003 |
| Mutation and data-flow techniques | Wong and Mathur | 1995 |
| | Frankl et al. | 1997 |
| Regression and improvement testing techniques | Wong et al. | 1998 |
| | Rothermel et al. | 1999 |
| | Elbaum et al. | 2000 |
| | Kim et al. | 2000 |
| | Graves et al. | 2001 |

relative effectiveness or efficiency of testing techniques because of the difference between parameters they have taken into consideration. The results also are very inconclusive and do not reveal much information. [Moreno et al., 2009] also states that experimental results are conflicting, and the experiments lack a formal foundation. From the analysis of previous studies, we can conclude that the experimental studies on software testing techniques does not provide a basis for making any strong conclusions regarding effectiveness or efficiency of different testing techniques. As a result we cannot generalize results of software testing techniques evaluation. Current studies point out that various other factors, in addition to the applied testing technique, have a strong effect on the results of defect finding effectiveness and efficiency. Even though the experiments are designed to study the effects of one or more selected testing techniques, the effects of all other factors cannot be excluded. The defect detection effectiveness and efficiency seems to depend on the person who test the software, the software being tested, and the actual defects that exist in the software. One important result that can be drawn from the existing experimental studies on testing techniques is that more faults are detected by combining individual testers than by combining different techniques [Juristo et al., 2004]. This is an important finding because it shows that the results of test execution vary significantly among individual testers despite the certain test case design strategy used. The variation between individuals seems to be greater than the variation between techniques. The different testers seem to find clearly different defects despite using the same technique. The most important results of *data flow testing techniques*, *mutation testing techniques* and *regression testing techniques* studies are presented in Table 2.3, Table 2.4 and Table 2.5 respectively. The most important results of *control flow, data flow and random testing techniques*, *code Reading, functional and structural testing techniques*, *mutation and data flow* and *regression and improvement testing techniques* are presented in Table 2.6, Table 2.7, Table 2.8 and Table 2.9 respectively. The results are discussed in more detail in [Juristo et al., 2004]. We can summarize the results of empirical studies on testing techniques as follows:

1. There is no clear, consistent evidence that one fault finding technique is stronger than others, rather the evidence to date suggests that each technique has its own merits.

2. While some studies conclude that technique A is ranked higher than technique B. Some studies conclude technique A and technique B find different kinds of defects, and are as complementary.

3. The effectiveness of verification activities is low; only 25-50% of the defects in an artifact are found using inspection, and 30-60% using testing. This makes secondary defect detection activities important.

4. Combining testing techniques uncovered more defects than did a single technique.

5. Combining individual testers seem to increase defect detection effectiveness more than combining test case design techniques.

6. Defect detection effectiveness highly depends on the individual differences between testers even if they use the same test case design technique.

7. Defect detection effectiveness seems to be correlated with the amount of test cases.

8. The effectiveness of different techniques seems to depend on the type of software tested and the types of the actual defects in the software.

9. It seems that some types of faults are not well suited to some testing techniques.

10. There appears to be a relationship between the programs, or the type of faults entered in the programs, and technique effectiveness.

**Table 2.3:** Major Results of Data-flow testing techniques comparison

| |
|---|
| 1. All-p-uses technique is better than all-uses which is better than all-du-paths, as they generate fewer test cases and generally cover the test cases generated by the other criteria. |
| 2. Even though all-c-uses generate fewer test cases, it is not clear that it is better to use it instead of all-p-uses, as coverage is not assured. |
| 3. The all-du-paths technique is usable in practice, since it does not generate too many test cases. |

### 2.4.2 Problems with Existing Studies

We must first clearly recognize the issues in any field in order to make it more mature by resolving those issues. Many years of empirical investigation in the subject have gone by though there are not definite results yet. In A look at 25 years of data, the authors have reached the same conclusion after studying various experiments on software testing [Moreno

**Table 2.4:** Major Results of Mutation testing techniques comparison

| |
|---|
| 1. When time is a critical factor, weak mutation/ selective mutation is preferred as opposed to standard mutation, as it generates fewer test cases and effectiveness is approximately the same.<br>2. In intermediate cases, it is preferable to use abs/ror mutation, because, although it generates more cases (from 50 to 100 times more), it raises effectiveness by seven points.<br>3. If time is not a critical factor, it is preferable to use standard mutation. |

**Table 2.5:** Major Results of Regression testing techniques comparison

| |
|---|
| 1. For programs with big sets of test cases, it is preferable to use deja-vu or test-tube as opposed to retest-all, as it takes less time to select and run test cases.<br>2. For small programs with small sets of test cases, it is preferable to use retest-all as opposed to deja-vu and test-tube, as it takes less time to select and run test cases.<br>3. When the number of test cases selected is an issue, it is preferable to use deja-vu as opposed to test-tube and test-tube as opposed to retest-all, since deja-vu selects fewer test cases than test-tube and test-tube selects fewer test cases than retest-all.<br>4. The percentage of selected cases over initial set depends on the structure of the program, and its modifications.<br>5. It is not clear whether to use textual differencing as opposed to retest-all. Although textual differencing selects a lower percentage of test cases over the initial set than retest-all, there are no definite results regarding the time taken to run test cases. There are also no definite results for textual differencing and deja-vu. |

**Table 2.6:** Major Results of Control-flow, data-flow and random testing techniques comparison

| |
|---|
| 1. When time is critical, the use of the random testing technique can be relied upon to yield an effectiveness similar to all-uses and all-edges (the differences being smaller the higher coverage is) in 50% of the cases. Where testing needs to be exhaustive, the application of all-uses provides assurance, as, in the other half of the cases; this criterion yielded more efficient results thanks to the actual technique, unlike all-edges, which was more efficient because it generated more test cases. |
| 2. All-edges should be complemented with all-dus, as they are equally effective and detect different faults. Additionally, they generate about the same number of test cases. The random testing technique has to generate between 50% and 160% more test cases to achieve the same effectiveness as all-edges and all-dus. |
| 3. High coverage levels are recommended for all-edges, all-uses and all-dus, as this increases their effectiveness. This is not the case for the random testing technique. |

et al., 2009]. Also, they found that it is really difficult to compare different experiments; however, they do not present any solution to it. [Briand and Labiche, 2004] discussed many issues facing empirical studies of testing techniques; criteria to quantify fault-detection ability of a technique is one such issue, while threats to validity arising out of the experimental setting (be it academic or industrial) is another. [Juristo et al., 2002] and [Juristo et al., 2004] has highlighted following issues with current studies:

1. informality of the results analysis (many studies are based solely on qualitative graph analysis)

2. limited usefulness of the response variables examined in practice, as is the case of the probability of detecting at least one fault

3. non-representativeness of the programs chosen, either because of size or the number of faults introduced

4. non-representativeness of the faults introduced in the programs

We believe that there are many reasons for this inadequacy of knowledge and restricted results regarding the evaluation of software testing techniques. After analyzing the existing studies

**Table 2.7:** Major Results of Code Reading, functional and structural testing techniques comparison

| |
|---|
| 1. For experienced subjects and when time is not an issue, it is better to use the boundary value analysis technique as opposed to sentence coverage, as subjects will detect more faults, although it will take longer. |
| 2. For inexperienced subjects and when time is short, it is better to use sentence coverage as opposed to boundary value analysis, although there could be a loss of effectiveness. The time will also depend on the program. |
| 3. It is preferable to use boundary value analysis as opposed to condition coverage, as there is no difference as regards effectiveness and it takes less time to detect and isolate faults. |
| 4. There appears to be a dependency on the subject as regards technique application time, fault detection and fault isolation. |
| 5. There appears to be a dependency on the program as regards the number and type of faults detected. |
| 6. More faults are detected by combining subjects than techniques of the two families. |
| 7. If control faults are to be detected, it is better to use boundary value analysis or condition coverage than sentence coverage. Otherwise, it does not matter which of the three are used. |
| 8. The effect of boundary value analysis and branch testing techniques on effectiveness cannot be separated from the program effect. |

**Table 2.8:** Major Results of Mutation and data-flow testing techniques comparison

| |
|---|
| 1. It is preferable to use all-uses as opposed to mutation in case when high coverage is important and time is limited, as it will be just as effective as mutation in about half of the cases. |
| 2. All-uses behave similarly as regards effectiveness to abs/ror and 10% mutation. |

**Table 2.9:** Major Results of Regression and improvement testing techniques comparison

| |
|---|
| 1. In case of time limitation, the use of regression based on modifications and minimization and prioritization can be relied upon to yield effectiveness similar to regression based on modifications only. |
| 2. In case of time limitation, the use of minimization can be relied upon to yield a lower effectiveness than safe and data-flow (these two equal) and random. |
| 3. The more faults there are in the program, the more effective minimization is. |
| 4. In case of time limitation, the use of prioritization can be relied upon to yield a higher fault detection speed than no prioritization. |
| 5. It is preferable to use fep techniques as opposed to sentence coverage technique, as they have a higher fault detection speed. |
| 6. Function coverage techniques do not behave equally. |
| 7. When time is not an issue, it is preferable to use sentence coverage techniques as opposed to function coverage techniques, as they are more effective, but also more costly. |
| 8. Effectiveness for these techniques depends on the program. |

on software testing techniques evaluation, we conclude that existing studies on evaluation of testing techniques mostly have following problems:

#### 2.4.2.1 Experimentation problems

1. Comparing testing techniques is to quantify fault detection effectiveness and efficiency. A comparison criterion for testing techniques is usually not well defined.[Farooq and Quadri, 2010] states that in the context of testing technique selection, the term best has different meanings depending on the person making comparisons.

2. Most of the studies do not take all the parameters necessary for comparison into consideration, as a result of that one technique do not supersede other techniques on all fronts; thereby creating ambiguity in test technique selection.

3. Existing studies mostly differ in the number and type of parameters they have used in their study. A common standard is missing which makes it difficult to compare these studies.

4. There are many things that are not covered by such studies. The inconclusive results indicate the presence of factors that were not under experimental control. The comparative study of the effectiveness of different techniques should be supplemented by studies of the fault types that each technique detects and not only the probability of detecting faults. That is, even if T1 and T2 are equally effective, this does not mean that they detect the same faults [Juristo et al., 2003]. This would provide a better understanding of technique complementary, even when they are equally effective.

5. Most of the studies use only fault seeding in their experiments. The result is that large numbers faults can be seeded, thus leading to less random variation in fault ratios and more statistical power. However it often results in seeding unrealistic faults. In addition, we may be biased towards seeding faults of a particular type. As a result, we are usually left with invalid results which are far away from reality.

6. In all studies, subjects have not been chosen properly according to given scenario. Even though experiment conducted by [Basili and Selby, 1987] took into account several classes of professionals. Most experiments conducted the studies in student environments, which limit the transfer of experiment results to real world. Therefore, we need to balance between academic and industry perspective.

7. Another major problem with testing technique evaluation is that experiments are mostly made on a small sample (code sample selection is in majority below 2K), and often with the demonstration that they either perform better than another specific technique. The main reason for this is the difficulty to get large amount of real data to perform research on. The number of faults on sample of this size may not be large enough to allow for quantitative, statistical analysis.

8. The experiments are often biased either towards academic or industrial system, as they are usually carried out with only academic or industrial settings into consideration. Most of the studies conducted so far are academic in nature. As a result, usually studies are less validated or hardly put to practice in testing industry. Studies in an academic setting are often a first step before studies are carried in industrial settings [Juristo and Moreno, 2001]. So we should take both systems into consideration while carrying out such experiments.

#### 2.4.2.2 Knowledge problems

1. Existing studies do not tend to share the knowledge they acquire by using a testing technique with others [Vegas, 2001]. The information related to these studies is not fully available which makes it difficult for researchers or industry professionals in drawing exact results from diverse studies. Also it becomes difficult to replicate the work already done. Everyone is going its own way, starting things from very beginning. It would have been good to validate the earlier studies so that results can be generalized and implemented at industry level.

2. Usually the main focus is often to invent a special technique and compare its effectiveness with one already known (often a similar technique). Little attention is given to evaluate effectiveness of already existing techniques which could have served professionals better.

3. The problem with testing techniques in industry is that they are not known (many testers have no training in testing techniques), not utilized, since often there is a belief of their efficiency and effectiveness, and it is seldom proven for larger complex system. The actual research setting of creating reasonable comparative models have not been totally explored. Our experience is that testers are often not trained in testing, but on system behaviour. Even if people are formally trained in test techniques, they easily

fall back to approaching testing from a system usage viewpoint rather than applying a test technique, since requirements on testers are seldom assessed as long as they find some failures.

## 2.5 Where do we Stand at this Moment?

The big achievement we had from conducting so many experimental experiments is: we do know with certainty that the usage of a testing technique is better than none, and that a combination of techniques is better than just one technique. We also know that the use of testing techniques supports systematic and meticulous work and that techniques are good for finding possible failures. No firm research conclusions exist about the relative merits of software testing techniques. The conclusions they drew may only apply to their specific experimental environment and are not general enough to be applied to other research environments, let alone to software testing industry [Yang, 2007]. Most of the research that has been performed is very academic and not terribly useful in the real testing world. At present we do not have adequate proof of any technique superseding other ones in terms of effectiveness or efficiency.

How should one choose testing technique at present? Current studies suggest it is just not sufficient to rely on a single method for catching all defects in a program. Actually each technique is good for certain things, and not as good for other things. Each individual technique is aimed at particular types of defect as well. For example, state transition testing is unlikely to find boundary defects. Some techniques are more applicable to certain situations and test levels; others are applicable to all test levels. Some testing techniques are never considered for use at all and others are used over again in different software projects without even examining, after use, whether or not they were really suited [Vegas, 2004]. One conclusion that seems to have been reached is: There is no “best technique. The “best depends on the nature of the product and other allied factors. The choice of which test technique to use depends on a number of factors, including the type of system, regulatory standards, customer or contractual requirements, level of risk, type of risk, test objective, documentation available, knowledge of the testers, time and budget, development life cycle, use case models and previous experience of types of defects found which are discussed more thoroughly in section 2.6. Each testing technique is good at finding one specific class of defect, using just one technique will help ensure that many (perhaps most but not all) defects of that particular class are found. Unfortunately, it may also help to ensure that many defects of other classes

are missed! Using a variety of techniques will therefore help ensure that a variety of defects are found, resulting in more effective testing. However, it will also ensure the excessive use of resources which will in turn result in less efficiency. So it is argued that more experimental work is required to evaluate testing technique so that our testing will be both effective and efficient. We need to know how to demonstrate the effectiveness of testing methods, how much effective are testing techniques in terms of effort and defect finding capability as we always want to select a testing technique that will bring the product to an acceptable level. Recent surveys on comparisons of various software testing techniques also concludes that further empirical research in software testing is needed, and that much more replication has to be conducted before general results can be stated [Juristo et al., 2004] [Moreno et al., 2009]. But the experimentation should be carried out in such a way so that results can be realistic and with very less contradictions. Then only we can have firm knowledge about the effectiveness and efficiency of the testing technique in revealing faults, the classes of faults for which the technique is useful, and other allied aspects.

## 2.6 Factors for Selecting Software Testing Technique

The big question remaining after these descriptions of wonderful test case design techniques is: Which testing technique(s) should we use? At present, the answer to that is: It depends! There is no established consensus on which technique is the most effective and efficient. At present, the decisions made regarding technique selection are mostly unsystematic. Testers actually make the selection on the basis of their particular perception of the techniques and situations, which is not necessarily incorrect, but partial (and therefore incomplete) [Vegas et al., 2006]. A firm and generalized thing that we know is that we have to use testing techniques for testing a system. With so many testing techniques to choose from how are testers to decide which ones to use? If not effective, at least we should choose a suitable testing technique. Another question that arises is that, how will we choose the most appropriate testing techniques? Rather can selecting testing techniques unsystematically, we should choose techniques according to certain factors. We have to select testing techniques anyway, so why not to do it systematically. The benefit is: If somebody asks how you did it, you are able to describe it, plus your reasoning behind it. Your test will be accountable. And you may be able to improve over time. The selection of testing techniques is related to various aspects/factors (both internal and external) [Graham and Van Veenendaal, 2008]. We have grouped those factors into three categories. There can be many other factors that

help us in selecting appropriate techniques for testing the software. The aspects/factors that aid us in selecting appropriate testing technique are listed below.

### 2.6.1 Software Related Factors

**Type of system:**
The type of system (e.g., graphical, embedded, financial, etc.) will influence the choice of techniques. For example, a financial application involving many calculations would benefit from boundary value analysis, while a GUI system will prefer GUI testing.

**Life cycle model:**
Life cycle model used in software development also impinge on the testing technique selection. For example, a sequential life cycle model will lend itself to the use of more formal techniques whereas an iterative life cycle model may be better suited to using an exploratory testing approach.

**Models used:**
Since testing techniques are based on models, the models available (i.e. developed and used during the specification, design and implementation of the system) will also govern the choice of testing technique to be used. For example, if the specification contains a state transition diagram, state transition testing would be a good technique to use.

**Likely defects:**
Knowledge of the likely defects will be very helpful in choosing testing techniques (since each technique is good at finding a particular type of defect). This knowledge could be gained through experience of testing a previous version of the system and previous levels of testing on the current version. What types of defects do the artefacts contain? There is a big difference between grammatical errors in code and missing requirements in a requirements specification.

### 2.6.2 Testing Related Factors

**Aspect:**
Which aspect of the software development are we assessing? Requirements? Design? Code? For Example, reviews are more beneficial in design as compared to code. Similarly if code is checked for consistency, code reading technique will be beneficial, whereas for checking its behavior, functional techniques are more appropriate.

**Purpose:**
The purpose of testing also defines which technique to use as activity might contribute to

validation that is, assuring that the correct system is developed or to verification that is, assuring that the system meets its specifications or to both. Static techniques contribute mostly towards verification, whereas dynamic techniques contribute to validation.

**Test objective:**

If the test objective is simply to gain confidence that the software will cope with typical operational tasks then routine techniques can be employed. If the objective is for very thorough testing (e.g. for safety-critical systems), then more rigorous and detailed techniques should be selected. Greater the risk, greater is the need for more thorough and more formal testing. Commercial risk may be influenced by quality issues (so more thorough testing would be appropriate) or by time-to market issues (so exploratory testing would be a more appropriate choice).

**Evaluation criteria:**

What are the criteria for selecting techniques? Should you choose the most effective or the most efficient method? Efficiency in this context means the number of defects found per time unit spent on verification, and effectiveness means the share of the existing defects found.

**Documentation:**

Whether or not documentation (e.g. a requirements specification or design specification) exists and whether or not it is made up to date will affect the choice of testing techniques. The content and style of the documentation will also influence the choice of techniques.

**Tester knowledge/experience:**

How much testers know about the system and about testing techniques will clearly influence their choice of testing techniques. This knowledge will in itself be influenced by their experience of testing and of the system under test.

**Time and budget:**

Ultimately how much time is available will always affect the choice of testing techniques. When more time is available we can afford to select more techniques and when time is severely limited we will be limited to those that we know have a good chance of helping us find just the most important defects.

### 2.6.3 Customers Requisites and other Requirements

**Customer/contractual requisites:**

A contract usually specifies the main objective of the system. Possible objectives can be performance, dependability etc. Accordingly we will choose a testing technique to meet that

objective.

**Regulatory requirements:**

Some industries have regulatory standards or guidelines that govern the testing techniques used. For example, the aircraft industry requires the use of equivalence partitioning, boundary value analysis and state transition testing for high integrity systems together with statement, decision or modified condition decision coverage depending on the level of software integrity required.

## 2.7 Proposed Guidelines for Software Testing Techniques Evaluation

The big question still remains there: Which are the techniques which are effective and efficient. The knowledge for selecting testing techniques should come from studies that empirically justify the benefits and application conditions of the different techniques [Juristo et al., 2003]. To get a definite answer to such question we need to carry out experimentation on a large scale using a common benchmark/framework. A common standard is also required to standardizing the evaluation process of such experiments. Empirical studies on large scale artifacts, within real world contexts, and replicated by several professional testers to attain generally valid results would be of course prohibitively expensive. A possible way out to overcome such difficult challenges could be that of combining the efforts of several research groups, currently conducting separate experimentations, and join their forces to carry out a widely replicated experiment, i.e., factorize a large experiment in pieces among several laboratories. The idea would be similar to that of launching an "Open Experiment" initiative, similarly to how some Open Source projects have been successfully conducted. This is undoubtedly a remarkable aim which without careful management is unlikely to succeed. In addition, not all open source projects are necessarily successful, and experimentation, to be credible, needs very careful planning and control. Moreover, such an enterprise could perhaps overcome the problems of scale, but the issues of context and tester's background would further require that industries be actively involved in the initiative [Bertolino, 2004].

Empirical software engineering research needs research guidelines to improve the research and reporting processes. There exists a real need in industry to have guidelines on which testing techniques to use for different testing objectives, and how usable these techniques are [Vos et al.]. Here we present a framework (a set of guidelines) how experiments/studies for evaluating the effectiveness of testing techniques should be carried out so that definite,

practical and comparable results about relative merits of testing techniques can be achieved. The proposed guidelines are general since no assumptions about the testing technique that is being evaluated, about the subjects and a program is made.

1. Studies should be carried on a set of common systems or at least similar systems. It will make comparisons of techniques much easier. [Weyuker, 1993] states that effectiveness of a testing technique is only possible to measure if you can compare two techniques for the same set (i.e. software).

2. The studies should be carried out on a large sample and on real data (most preferably on industrial data). The number of detected faults will be sufficient enough for quantitative, statistical analysis. Carrying out experiments on such data will draw results that will be near to perfection if not perfect.

3. From an external validity standpoint, experimenting on actual faults is realistic. Carrying out experimentation on actual faults is more realistic from an external validity standpoint. However, extracting proper results from such results often is time consuming as we are unaware about the number of actual faults present in product. In this method detailed fault information can be expensive to collect. On the other hand fault seeding allows us to seed as many faults as necessary in order to work with a sample that is large enough to be amenable to statistical analysis. However seeding does not always seed realistic faults. However if we still want to go for fault seeding, we need an unbiased, systematic, and inexpensive way to do so. We need to investigate and define procedures to seed faults for the purpose of experimentally assessing test techniques. So it is advisable to use both methods on different systems, this way we can achieve more concrete and applicable results.

4. We can perform software testing experiments either by using human subjects or by using simulation. Using first one allows us to access other factors like cost effectiveness and human applicability, Test suites are perhaps more realistic, as derived by human subjects performing the actual test tasks; while second one allows us to test software rigorously (100% coverage) as we can generate large of test sets, accounting for random variation but this technique can suffer from biasing problem if not implemented properly, It is good to use both techniques but role and domain of each should be well defined.

5. Criteria for comparing testing techniques should take into consideration many parameters. We cannot expect experiments to be perfect with respect to all factors which need to be taken into consideration while evaluating the effectiveness of software testing techniques. But we should strive towards carrying out experiments which takes into consideration maximum factors related to testing technique effectiveness. Taking into account diverse parameters will yield more appropriate results and will make testing techniques selection more valid and unambiguous. Some of factors necessary for comparison are

   (a) Number of faults

   (b) Fault rate

   (c) Fault type

   (d) Size (test case generated)

   (e) Coverage

   (f) Time (Usually it is execution time)

   (g) Software

   (h) Experience of subjects

   (i) Reliability improvement

6. Experimental details should be shared. Experiment should lead to an experimentation package that would allow other researchers to easily replicate experiments. Ideally, it should contain all the necessary material to perform the experiment and should be publicly available, under certain conditions. This would allow the research community to converge much faster towards credible results. This would allow other researcher to analyze the data and possibly draw different conclusions.

7. We should balance all dimensions of validity to achieve trustworthy empirical studies (i.e. the balance between internal (researcher point of view) and external (Practitioner point of view) validity). Studies in academia are often strong in terms of internal validity (i.e. our capability to draw proper conclusions from the data) and weak as far as external validity is concerned (i.e. it is hard to know the extent to which you can generalize your results to industrial contexts). Field studies have exactly the opposite strengths and weaknesses. Both academic and field studies are necessary. Field studies

are more suited to assess to difficulties to apply techniques in practice and to confirm the results obtained on real sets of faults.

8. The information available about the techniques is normally distributed across different sources of information (books, articles and even people) [Vegas, 2004]. We should work towards building a sharable centralized depository on testing techniques.

9. Removing X% of the faults in a system will not necessarily improve the reliability by X%. A study at IBM showed that removing 60% of product defects resulted in a 3% improvement in reliability. So we should not only strive to find out find out techniques which finds maximum errors, but also those techniques which increase the reliability. In fact we should execute techniques in such a way so that it not only finds faults but also increases reliability. Statistical testing can be used to enhance the effectiveness of the testing effort. Whereas software testing in practice is usually focused on finding bugs, statistical testing is focused on evaluating reliability.

## 2.8 Conclusion and Future Work

Despite the general feeling that everything is changing fast, techniques do not usually change overnight. One sure thing that we came to know is that we have to do testing anyhow. With so many testing techniques and the very inadequate quantitative and qualitative knowledge about them, we strongly believe that there is a need to further evaluate software testing techniques. Presently we are unaware about the relative ordering of software testing techniques and if we are to make software testing more effective by selecting effective testing techniques then we need to place existing software testing techniques at least on an ordinal scale. Present situation call for replication and further work on evaluation of software testing techniques so as to acquire the basic knowledge about the relative effectiveness and efficiency of software testing techniques for both fault finding and reliability criterion. To do so we need to carry out experimentation on large scale but that needs to in a way that can be compared and will have no contradictions. For that we also need to establish common and standard parameters so that there are little variations in experimentation goals. We also need to find out dimensions on the basis of which we can all agree that if one testing method A is more effective than another testing method i.e., what "effectiveness" is exactly meant for. Possible interpretations are how many tests are needed to find the first failure, or the percentage of the faults found by the testing technique to all the faults, or of how much reliability is improved.


End

# Chapter 3

# Evaluating Software Testing Techniques for Effectiveness & Efficiency

## 3.1 Introduction

Present day systems are becoming more and more software intensive rather than hardware intensive, as software has moved from a secondary to a primary role in providing critical services. Testing such systems require us to make a choice among many available testing techniques. Although the utilization of these techniques is growing, we do have a very inadequate knowledge about their relative quantitative and qualitative statistics. All of them can reveal failures, but we do not know how effectively and efficiently they do that and what kind of faults they find. The choice of a testing technique in software testing influences both process and product quality.

The software engineering community generally accepts that defect detection should be based on both dynamic and static testing techniques, as testing only code by executing it is not sufficient enough to guarantee that all faults and flaws in the software system are identified; so using techniques other than dynamic testing techniques becomes essential. In static testing, the code is examined, the aim being to discover maximum inaccuracies through observation. Static analysis techniques differ as to the way in which the code is observed. In dynamic testing, the code is executed, the aim being to discover code defects by observing system behaviour and trying to deduce whether or not it is satisfactory. Additionally, whereas static testing detects the faults the software contains (a fault is flaw or deviation from a desired or intended state in a software product), all dynamic testing can do is detect failures (the inability of a system to perform its required functions within specified requirements). As the ultimate aim of software testing is to correct any faults in the software, dynamic testing calls for a further step to identify faults from the observed failures.

Many static and dynamic techniques for evaluating software system code have been proposed. However, not much work has been put into finding out the strengths and weaknesses of each technique. For software engineering to move from a craft towards an engineering discipline, software developers need empirical evidence to help them decide what defect-detection technique to apply under various conditions [Basili et al., 1985] [Rombach et al., 1993]. Several experiments have measured effectiveness and efficiency of various defect detection methods. However, existing experimental research on testing techniques does not provide a basis for making any strong conclusions regarding effectiveness or efficiency of different testing techniques as the results are generally incompatible and do not allow one to make objective comparison on the effectiveness and efficiency of various testing approaches. The reason for this incompatibility is that most of the empirical studies to date have been

conducted under different conditions and examined programs of different size and application domains. To avoid bias in the comparison of methods, experiments should be conducted under a common framework so that they can be compared in a way that can help us to generalize the results. As mentioned in chapter 2, recent surveys on comparisons of various software testing techniques also concludes that further empirical research in software testing is needed, and that much more replication has to be conducted before general results can be stated. Replication helps validate empirical results published by other software-engineering researchers [Daly et al., 1994].

To contribute to the knowledge base of testing techniques, we replicated an experiment that evaluates three defect-detection techniques code reading, functional testing, and structural testing. The reason to select these testing methods is that they are most widely used testing methods in the practice. The experiment proposed here aims to contribute to clarifying differences between techniques for practical purposes such as how many defects they detect, what type of defects they detect and how much time they take to find a defect, etc. The origins of this experiment go back to the work of [Hetzel, 1976] and [Myers, 1978]. More precisely, it is the continuation of a line of experiments run by other authors, which have added to the knowledge provided by previous experiments. However, the most significant study was conducted by [Basili and Selby, 1987]. This experiment studied the effectiveness and efficiency of different code evaluation techniques. The work of Basili and Selby was first replicated by [Kamsties and Lott, 1995]. This replication assumed the same working hypotheses as in Basili's experiments, but the experiment differed as to the programming language used. In addition, the fault isolation phase was added in the experiment. Their work was replicated again by [Roper et al., 1997]. Their experiment followed exactly the same guidelines as the experiment run by Kamsties and Lott (who had built a laboratory package to ease external replication of the experiment), although new analyses were added. Further the experiment was replicated by [Juristo and Vegas, 2003]. Their experiment stressed on the fault types and did not considered efficiency of testing techniques. We replicated the experiment which was actually carried out by [Kamsties and Lott, 1995] and further replicated by [Roper et al., 1997] which includes fault isolation phase in addition to fault detection phase. Detection refers to the observation that the programs observed behaviour differs from the expected behaviour. Isolation means to reveal the root cause of the difference in behaviour. In our case, it is a fault in the code. The experiment package built by Kamsties and Lott was used, although some experimental conditions like hypothesis are changed which are described in more in subsequent sections.

The chapter has the following structure: Section 3.2 presents the schema used for the experiment, Section 3.3 describes goals, hypothesis and theories, Section 3.4 presents the experiment plan which describes experimental design, defect detection techniques used, programs used in subsections 3.4.1, 3.4.2 and 3.4.3 respectively. Section 3.5 describes the experiment procedures, Section 3.6 presents and discusses the results, Section 3.7 summarizes the results of this experiment and finally, Section 3.8 presents conclusion and future work.

## 3.2 Schema Used

Many researchers in the past highlighted the need to carry out experiment under a common framework or using common or similar standards and parameters. Framework proposed in chapter 2, mostly stresses on carrying experimentation and evaluation under a common benchmark. In effect, we used a characterization scheme proposed by [Lott and Rombach, 1996] to carry out our experiment. Carrying out experiment using this schema will permit the comparison of results from similar experiments and establishes a context for cross experiment analysis of those results. The schema is as follows:

1. Goals, Hypotheses, and Theories
   (a) Aspects of a goal
      i. Object of study (e.g., code reading, functional testing, . . . )
      ii. Purpose of study (e.g., compare, analyse, . . . )
      iii. Quality focus of study (e.g., effectiveness, efficiency, . . . )
      iv. Point of view (e.g., practitioner, experimenter, . . . )
      v. Context (e.g., subjects, objects, environment, . . . )
   (b) Hypotheses
      i. Type (e.g., direct observations, context factors, . . . )
      ii. Expected result (i.e., null and alternative hypotheses)
   (c) Theories
      i. Mechanisms that predict and/or explain results
      ii. Derived from beliefs or related work
2. Experiment Plan
   (a) Experimental design

i. Independent variables (e.g., techniques, objects, order, . . . )
ii. Dependent variables (e.g., defects found, time required, . . . )
iii. Randomization (e.g., match of subject, object, and technique)
iv. Repeated measures (e.g., within-subject designs)
v. Manipulation of independent variables (e.g., full-factorial, partial-factorial, . . .)
vi. Null hypotheses (e.g., technique A has no effect on . . . )

(b) Defect-detection techniques for source code
i. Type (e.g., reading, functional testing, structural testing, . . . )
ii. Other aspects (e.g., test-case development, termination criteria, . . . )

(c) Objects
i. Source-code modules (e.g., length, complexity, . . . )
ii. Faults (e.g., number, types, interactions, . . . )

(d) Subjects
i. Selection criteria (e.g., participants in a course)
ii. Experience, training, and background (e.g., students, professionals, . . . )
iii. Ethical issues (e.g., right to withdraw, anonymity, . . . )
iv. How many are required (assess power of analysis procedure)

(e) Data collection and validation procedures
i. On-line and off-line collection procedures (e.g., forms, videotape, counts of runs, . . . )
ii. Validation approaches (e.g., independent sources, interviews, . . . )

(f) Data analysis procedures
i. Significance level for inferential statistics (e.g., $p < 0 : 05$)
ii. Parametric techniques
iii. Non-parametric techniques

3. Experiment Procedures

(a) Training activities (e.g., independent work, controlled setting, . . . )

(b) Conducting the experiment (e.g., time periods, data validity, . . . )

(c) Giving feedback to subjects (e.g., comparing expectations with results, . . .)

4. Results

(a) Data (i.e., the raw data collected during the study)

(b) Interpretations (i.e., statements about the hypotheses)

## 3.3 Goals, Hypotheses and Theories

A statement of goals determines what an experiment should accomplish, and thereby assists in designing and conducting the experiment [Basili et al., 1985]. We used GQM (Goal-Question-Metrics) approach to state the goals of our experiment. Accordingly we define our main goal of the experiment as:

Analyse **code reading**, **functional testing** and **structural testing** techniques for **detecting software defects** for the **purpose of comparison** with respect to their **effectiveness, efficiency** from the point of view of the **researcher** in the context of a **controlled experiment**.

Our main goal is to evaluate the relative effectiveness and efficiency of software testing techniques. In addition we want to study the effects of other factors also. Accordingly, the two main hypotheses are:

**MH**$_{01}$: Testing techniques of code reading, functional testing and structural testing do not differ in their effectiveness.

**MH**$_{11}$: Testing techniques of code reading, functional testing and structural testing differ in their effectiveness.

**MH**$_{02}$: Testing techniques of code reading, functional testing and structural testing do not differ in their efficiency.

**MH**$_{12}$: Testing techniques of code reading, functional testing and structural testing differ in their efficiency.

### 3.3.1 Goals

**Goal 1: Effectiveness at revealing failures.**

The goal as per GQM is stated as follows: Analyze code reading, functional testing, and structural testing for the purpose of comparison with respect to their effectiveness at revealing failures/ inconsistencies from the point of view of the researcher in the context of a controlled experiment using small C programs.

**Goal 2: Efficiency at revealing failures.**

The goal as per GQM is stated as follows: Analyze code reading, functional testing, and structural testing for the purpose of comparison with respect to their efficiency at revealing failures from the point of view of the researcher in the context of a controlled experiment using small C programs.

**Goal 3: Effectiveness at isolating faults.**

The goal as per GQM is stated as follows: Analyze code reading, functional testing, and structural testing for the purpose of comparison with respect to their effectiveness at isolating faults from the point of view of the researcher in the context of a controlled experiment using small C programs.

**Goal 4: Efficiency at isolating faults.**

The goal as per GQM is stated as follows: Analyze code reading, functional testing, and structural testing for the purpose of comparison with respect to their efficiency at isolating faults from the point of view of the researcher in the context of a controlled experiment using small C programs.

### 3.3.2 Hypothesis

Statements about the expected results that can be tested using the experiment are called testable hypotheses. To support testing such statements using inferential statistical methods, these statements are eventually formulated as null hypotheses, and the original statement is called the alternative hypothesis [Judd et al., 1991].

Testable hypotheses derived from goal 1 are as follows:

**$H_{01}$:** Subjects using defect-detection techniques reveal and record the same percentage of total possible failures[1]

$H_{11}$: Subjects using the three defect-detection techniques reveal and record a different percentage of total possible failures.

Testable hypotheses derived from goal 2 are as follows:

$H_{02}$: Subjects using defect-detection techniques reveal and record a same percentage of total possible failures per hour.

$H_{12}$: Subjects using defect-detection techniques reveal and record a different percentage of total possible failures per hour.

[1] Only failures that were both revealed by the subject's detection efforts and were recorded by the subject are counted to compute this percentage.

Testable hypotheses derived from goal 3 are as follows:

$\mathbf{H}_{03}$: Subjects using defect-detection techniques isolate a same percentage of faults after failures are observed by applying one of the defect detection techniques[1].

$\mathbf{H}_{13}$: Subjects using defect-detection techniques isolate a different percentage of faults after failures are observed by applying one of the defect detection techniques.

Testable hypotheses derived from goal 4 are as follows:

$\mathbf{H}_{04}$: Subjects isolate a same number of faults per hour by applying one of the defect detection techniques.

$\mathbf{H}_{14}$: Subjects isolate a different number of faults per hour by applying one of the defect detection techniques.

These hypotheses can be tested by answering following questions:

Q1. What influence does each independent variable have on effectiveness of failure observation and fault isolation?

Q2. What influence does each independent variable have on the time to observe failures, time to isolate faults and the total time?

Q3. What influence does each independent variable have on the efficiency of failure observation and fault isolation?

In addition to this we will also extend our analysis to failure observation and fault isolation for each type and class of faults which are discussed in Section 3.4.3.1. This can be investigated by answering following question

Q4. Which technique leads to the observation of largest percentage of failures and isolation of largest percentage of faults from each type and class?

### 3.3.3 Theories and Related Work

Evaluation of software testing techniques is a main element of interest in software testing. The benefits and need of evaluating testing techniques is already explained in detail in previous chapter. As mentioned in table 2.2, five studies are significant which compares code reading, functional and structural techniques.

[Hetzel, 1976] compared functional testing, code reading and a technique that was a combination of functional testing, structural testing and code reading (selective testing).

[1] An important requirement for counting isolated faults was that a failure corresponding to the isolated fault had been revealed. Without this requirement, the fault could have been isolated purely by chance, not based on the use of a defect-detection technique.

This experiment used 39 subjects (students and inexperienced programmers) and was based on testing three PL/I programs. His main finding was that functional and selective testing was equally effective, with code reading appearing inferior.

This work was built upon by [Myers, 1978] who compared team-based, code walkthroughs/inspections with individuals using variations of structural and functional testing. Myers experiment used 59 professional programmers (averaging 11 years experience). The experiment was based on one PL/I program - the renowned Naur text formatter. The main findings of this work included that the walkthrough/inspection approach was found to be as effective in terms of fault finding as either of the two approaches to testing, there was tremendous variability amongst the performance of these experienced programmers and the ability to detect certain types of errors varied from method to method. An important aspect of this work was Myers investigation of theoretical combinations of testers. Comparing paired performance with single technique performance it was found that all pairs performed statistically better, though there was no statistical difference between any of the pairings. Whilst individuals were averaging only 33% fault detection, pairs averaged 50%.

In the 1980s this empirical research was refined by [Basili and Selby, 1987]. The three techniques that were compared were functional testing using equivalence partitioning and boundary value analysis, structural testing using 100% statement coverage, and code reading using stepwise abstraction. The 74 subjects were a mixture of experienced professionals and advanced students. The four programs studied were written in Fortran or a structured language, Simpl-T. The main findings of this work included that code reading detected more faults and had a higher detection rate than functional or structural testing, the number of faults observed, fault detection rate, and total effort in detection depended on the software type and each technique had some merit.

[Selby, 1986] also investigated all six possible pairwise combinations of the three techniques into hypothetical teams of two. In comparing the performance of the teams with individuals Selby found:

1. On average the combinations detected 67.5% of the faults in comparison to an average of 49.8% for individual techniques;

2. Code reading was found to be the most cost effective technique, both individually and as part of a combination;

3. The percentage of faults detected and the fault-detection cost-effectiveness depended on the type of software being tested.

[Kamsties and Lott, 1995] replicated the experiment of [Basili and Selby, 1987] using the same basic experimental design with some variations. They changed the programs and the language (to C) and the associated faults. They also included a fault-finding phase, and used a variation of branch coverage (which incorporated additional conditional, loop and relational operator criteria) as the structural testing criteria. The other testing techniques remained unchanged. They ran two versions of the replication, both with student subjects. The first replication had 27 subjects complete all three tasks, the second had 15. The main findings of this work included:

1. There was no statistical difference between the three techniques in either of the replications.

2. Functional testers observed failures most efficiently and were most efficient at isolating faults.

The work of [Kamsties and Lott, 1995] was replicated by [Roper et al., 1997]. They found effectiveness-related differences between functional and structural techniques depending on the program to which they were applied. The observed differences in effectiveness by fault class among the techniques suggest that a combination of the techniques might surpass the performance of any single technique.

Finally, the work of [Roper et al., 1997] was replicated by [Juristo and Vegas, 2003]. Their main purpose was to investigate the impact and relation of fault types on testing techniques. They concluded that effectiveness depends on the program, technique and fault type. However, the efficiency of the testing techniques was not considered in the experiment. They also ran two replications of their experiment.

The complete description of each study is shown in Table 3.1. The results of the [Hetzel, 1976], [Myers, 1978], [Basili and Selby, 1987], [Kamsties and Lott, 1995], [Roper et al., 1997] and [Juristo and Vegas, 2003] are presented in Table 3.2, Table 3.3, Table 3.4, Table 3.5, Table 3.6 and Table 3.7 respectively. The results presented in this section are discussed comprehensively in [Juristo et al., 2004]. The aim of this replication study is to further investigate these conclusions, and to contribute to the body of empirical evidence that is evolving in this area.

The effectiveness and efficiency statistics of these experiments is shown in Table 3.8 and Table 3.9 respectively. The (-) in some blocks denotes that the corresponding information was not taken into consideration in the experiment.

**Table 3.1:** Description of existing studies on code reading, functional and structural testing

| Author Year | Programs | Subject Count | Fault Count | Techniques |
|---|---|---|---|---|
| Hetzel 1976 | 3 Programs coded in PL/I with 64,164 and 170 LOC | 39 | 9, 15 and 25 | Disciplined Reading, Selective Testing, Specification Testing |
| Myers 1978 | Single program coded in PL/I with 63 LOC | 59 | 15 | Walk-through /inspection Functional testing Structural testing |
| Basili and Selby 1987 | 4 programs coded in Simpl-T or Fortran with 169, 145, 147 and 365 LOC | 74 | 34 | Boundary value analysis, Statement coverage and Stepwise abstraction. |
| Kamsties and Lott 1995 | 3 programs coded C with 260, 279 and 282 LOC | 50(27 in replication1 and 23 in replication 2) | 35 (11,14,11 in replication 1 and 6,9,7 in replication 2) | Boundary value analysis, Branch, multiple condition,loops and relational operators coverage and Stepwise abstraction. |
| Roper et al. 1997 | 3 programs coded C with 260, 279 and 282 LOC | 47 | 8,9,8 | Boundary value analysis, Branch coverage and Stepwise abstraction. |
| Juristo and Vegas 2003 | 4 programs coded C 400 LOC | 196 | 9*4 | Equivalence partitioning, Branch coverage and Stepwise abstraction |

**Table 3.2:** Results of Hetzel Experiment

| Aspect | Results |
|---|---|
| Effectiveness (Detection) | 1. Subjects who applied the reading technique performed less effectively than those who applied the testing techniques<br>2. No significant difference in effectiveness was detected among the two testing techniques. |

**Table 3.3:** Results of Myers Experiment

| Aspect | Results |
|---|---|
| Effectiveness (Detection) | 1. No significant differences in effectiveness among the three techniques. |
| Efficiency (Detection) | 1. The walkthrough/ inspection method required the most time, functional testing somewhat less, and structural testing the least amount of time. |

## 3.4 Experimental Plan

The Experiment plan specifies and describes the experiment design of the experiment. In addition, it also specifies the treatments and objects used in the experiment. Besides, it also specifies the data collection and measurement techniques used in the experiment. An effective experiment plan in very important in order to ensure that the right type of data and a sufficient sample size are available to meet the specified objectives of the experiment as clearly and efficiently as possible.

### 3.4.1 Experimental Design

Subjects applied three defect-detection techniques (first independent variable) to three different programs (second independent variable) in different orders (third independent variable). The experiment requires three days, but all subjects see the same program on the same day to prevent cheating. Therefore the variable "day" is confounded with "program and is not considered as a separate independent variable. We would have separated the confounded factors "program" and "day"; however we were not sure that the subjects will not discuss the program outside the experiment. Finally the subject is the fourth independent variable,

**Table 3.4:** Results of Basili and Selby Experiment

| Aspect | Results |
|---|---|
| Effectiveness (Detection) | 1. Experienced subjects: Code reading better, then functional, and then structural.<br>2. Inexperienced subjects: a) In one case, there is no difference between structural, functional and reading. b) In the other, functional is equal to reading, and both better than structural.<br>3. Depends on software type<br>4. Intermediate behave like junior and worse than advanced<br>5. Self-estimates more accurate for review, then structural. No relationship for structural. |
| Effectiveness (Observable) | 1. Functional reveals more observable faults than structural for inexperienced subjects.<br>2. Functional technique detects more of these observable faults for experienced subjects. |
| Fault detection Cost | 1. Experienced subjects: Equal time and fault rate.<br>2. Inexperienced subjects: Structural takes less time than review, which equals to functional.<br>3. The fault rate with functional and structural is less than with reading for inexperienced.<br>4. The fault rate depends on the program.<br>5. Functional testing has more computer costs than structural.<br>6. Total effort is the same for all techniques.<br>7. Fault detection rate is related to experience. |
| Fault type | 1. Review is equal to functional and both better than structural for omission and for initialization faults.<br>2. Functional is equal to structural and both better than review for interface faults.<br>3. Review is equal to structural and worse than functional for control faults.<br>4. Structural is equal to functional and both worse than review for computation faults.<br>5. For observable faults, functional and structural behave equal. |

**Table 3.5:** Results of Kamsties and Lott Experiment

| Aspect | Results |
|---|---|
| Effectiveness (Detection) | 1. Depends on the program, not the technique. |
| Effectiveness (Isolation) | 1. Depends on the program and subject, not on the technique. |
| Efficiency (detection) | 1. Boundary value analysis takes less time than condition coverage.<br>2. The time spent on finding faults also depends on the subject.<br>3. Boundary value analysis has a higher fault rate than condition coverage |
| Efficiency (isolation) | 1. Depends on the program and subject, not on the technique<br>2. With inexperienced subjects, boundary value analysis takes longer than condition coverage |
| Efficiency (Total) | 1. With inexperienced subjects, boundary value analysis takes less time than condition coverage<br>2. Time also depends on the subject. |
| Fault Type | 1. For both detected and isolated: There is no difference between techniques |

**Table 3.6:** Results of Roper *et al* Experiment

| Aspect | Results |
|---|---|
| Effectiveness (Detection) | 1. Depends on the program/technique combination<br>2. Depends on nature of faults |
| Combination of techniques | 1. Higher number of faults combining techniques |

**Table 3.7:** Results of Juristo and Vegas Experiment

| Aspect | Results |
|---|---|
| Effectiveness (detected and observable) | 1. Depends on program, technique and fault. 2. Code reading always behaves worse than the functional and structural techniques, indistinctly for the defect type. With regard to functional and structural techniques, they both behave identically. The program version influences on the number of subjects that detect a defect. |

**Table 3.8:** Average percentage of defects detected in existing experiments

| | | Effectiveness | | |
|---|---|---|---|---|
| | | **Code Reading** | **Functional** | **Structural** |
| **Hetzel** | | 37.3 | 47.7 | 46.7 |
| **Myers** | | 38 | 30 | 36 |
| **Basili and Selby** | | 54 | 54.6 | 41.2 |
| **Kamsties and Lott** | Replication 1 | 43.5 | 47.5 | 47.4 |
| | Replication 2 | 52.8 | 60.7 | 52.8 |
| **Roper *et al*** | | 32.1 | 55.2 | 57.5 |
| **Juristo and Vegas** | Replication 1 | 19.98 | 37.7 | 35.5 |
| | Replication 2 | - | 75.8 | 71.4 |

**Table 3.9:** Average defect detection rate in existing experiments

| | | Efficiency | | |
|---|---|---|---|---|
| | | **Code Reading** | **Functional** | **Structural** |
| **Hetzel** | | - | - | - |
| **Myers** | | 0.8 | 1.62 | 2.07 |
| **Basili and Selby** | | Depends on program | Depends on program | Depends on program |
| **Kamsties and Lott** | Replication 1 | 2.11 | 4.69 | 2.92 |
| | Replication 2 | 1.52 | 3.07 | 1.92 |
| **Roper *et al*** | | 1.06 | 2.47 | 2.20 |
| **Juristo and Vegas** | Replication 1 | - | - | - |
| | Replication 2 | - | - | - |

which is however an uncontrolled independent variable.

Our dependent variables focus on failures and faults as well as the time spent to apply the techniques. They include:

1. Number of failures detected
2. Number of faults isolated
3. Time to detect failures
4. Time to isolate faults

The following metrics were derived from the dependent variables:

1. % faults detected
2. % faults isolated
3. Time to detect faults
4. Time to isolate faults
5. Total time to detect and isolate

6. No. faults found/ time
7. No. faults isolated/hour
8. % faults detected/type
9. % faults isolated/type

The randomization consists of match of techniques, programs, order of applying the techniques and assigning subjects randomly to one of three groups. Membership in a group decides the match between technique and program for each subject as well as the order of applying the techniques.

Repeated measurements were taken as subjects are observed multiple times (within-subject design) as every subject applies each technique in the experiment.

The Experimental design applied to our study was randomized, fractional factorial design [Box et al., 2005]. It involves three factors (Testing Technique, Program and the order in which these techniques are applied). Our experiment like many other previous experiments measures the performance of every subject on every combination of three programs and three defect detection techniques. However, once a subject has applied one detection technique to a program, it will have some learning effects as he will acquire some knowledge of the program and some of the faults. It is therefore not reasonable to let them apply another detection technique to the same program. This constrains the experimental design to the combinations such that each subject may only apply one technique to a program as shown in Table 3.10

**Table 3.10:** Experimental Design Summary

| **Program Day** | **Program 1 (Day 1)** | **Program 2 (Day 2)** | **Program 3 (Day 3)** |
|---|---|---|---|
| Group 1 | Code Reading (CR) | Functional Testing (FT) | Structural Testing (ST) |
| Group 2 | Functional Testing (FT) | Structural Testing (ST) | Code Reading (CR) |
| Group 3 | Structural Testing (ST) | Code Reading (CR) | Functional Testing (FT) |

The hypotheses concerning external validity correspond directly to the testable hypotheses derived from the goals; the rest check for threats to internal validity [Lott and Rombach, 1996].

The experimental design allows us to test four null hypotheses in each case; these hypotheses will incorporate hypothesis stated in Section 3.3.2

**D.1:** The technique has no effect on the results (i.e., the techniques do not differ in their effectiveness and efficiency).

**D.2:** The program and day have no effect on the results; i.e., no selection effects.

**D.3:** The order in which subjects apply the techniques has no effect on the results; i.e., no learning effects.

**D.4:** The subjects have no effect on the results; i.e., all subjects perform similarly (no selection effects).

The primary null hypothesis for external validity in our case states that the different techniques have no effect on the results. Additionally, null hypotheses concerning internal validity issues help the experimenter quantify threats such as selection or learning effects. As an example, a null hypothesis may state that the different programs or subjects have no effect on the results. So depending on the hypothesis and questions stated in Section 3.3.2, overall we will analyze the results by testing following null hypothesis:

**$N_{1.1}$:** None of the independent variables (technique, program, group, or subject) affects the percentage of total possible failures observed.

**$N_{1.2}$:** None of the independent variables (technique, program, group, or subject) affects the percentage of total faults isolated.

**$N_{1.3}$:** None of the independent variables (technique, program, group, or subject) affects time taken to reveal and observe failures.

**$N_{1.4}$:** None of the independent variables (technique, program, group, or subject) affects time taken to isolate faults.

**$N_{1.5}$:** None of the independent variables (technique, program, group, or subject) affects the total time which includes both failure detection and fault isolation time.

**$N_{1.6}$:** None of the independent variables (technique, program, group, or subject) affects the failure observation rate.

**$N_{1.7}$:** None of the independent variables (technique, program, group, or subject) affects the fault isolation rate.

**$N_{1.8}$:** Techniques are equally effective at observing failures caused by the various types and classes of faults.

**$N_{1.9}$:** Techniques are equally effective at isolating faults of the various types and classes of faults.

### 3.4.2 Defect Detection Techniques

Testing techniques cover black and white box tests, structural tests, functional tests, regression tests. Moreover, there are code reading, walkthroughs and error detection techniques that belong to this category [Perry, 2006]. We used the same fault detection techniques as used in the previous experiment: code reading, functional testing and structural testing. These three testing techniques are mostly used to test software in real world. However different experiments in the past have used different criteria for each technique. In our case, code reading is applied using stepwise abstraction, functional testing using equivalence class partitioning and boundary value analysis and structural using 100% branch, multiple-condition, loop, and relational-operator coverage. For example, 100% coverage of a multiple condition using a single logical and operator means that all four combinations of true and false must be tested, and 100% coverage of a loop means that it must be executed zero, one, and many time(s). For the structural technique, like [Juristo and Vegas, 2003], the subjects have not used any tool to measure coverage like GCT (generic coverage tool) which was used in [Kamsties and Lott, 1995] and [Roper et al., 1997] experiments. This will affect the time it will take the subjects to generate the test cases (not the quality of the task performance, as the programs are simple enough for subjects to be able to do without a testing tool) [Juristo and Vegas, 2003]. All techniques are applied in a two stage process: failure observation (observable differences between programs and the official specification) and fault isolation (identifying the exact location of the cause of the failure in program code).

#### 3.4.2.1 Code reading

Code reading is applied using stepwise abstraction in a 3 step process without using any computer as program is not executed as in case of functional or structural testing. In step 1, subjects were given line numbered printed source code. They read the source code and write their own specification using stepwise abstraction by identifying prime subroutines (consecutive lines of code), write a specification for the subroutines, as formally as possible, group subroutines and their specifications together, and repeat the process until they have abstracted all of the source code After writing their own specifications, the subjects receive the official specification. In step 2, subjects compare the official specification with their own specification to observe inconsistencies (failure observation) between specified and expected program behavior (analog to failures in the other techniques). In step 3, the subjects begin

to isolate the faults that led to the inconsistencies which were observed in step 2. No special technique is specified for the fault-isolation activity. Finally, subjects hand in a list of identified inconsistencies and isolated faults.

#### 3.4.2.2 Functional testing

Functional testing is applied using equivalence partitioning and boundary value analysis in 4 step process. In step 1, subjects are provided with the official specification of the program (no source code). They identify equivalence classes in the input data and construct test cases using the equivalence class and boundary value analysis. In step 2, the subjects execute their test cases on the computer by running the executable version of the program. They are strictly instructed not to generate additional test cases during step 2, but we can neither prevent nor measure this. After executing all their test cases the subjects print out their results. In step 3, the subjects use the specification to observe failures that were revealed in their output; No automatic test oracle was used. After observing and documenting the failures, subjects receive the printed source code in exchange, and begin step 4. In step 4, the subjects use the source code to isolate the faults that caused the observed failures. No special technique is specified for the fault-isolation activity. Finally, subjects hand in a list of observed failures and isolated faults.

#### 3.4.2.3 Structural testing

Structural testing is applied using 100% branch, multiple-condition, loop, and relational-operator coverage in 4 step process. In step 1, subjects receive printed source code (no specification). They are instructed to construct test cases to get as close as possible to 100% coverage of all branches, multiple conditions, loops, and relational operators (100% coverage is usually unrealistic). The subjects develop additional test cases until they believe that they have attained 100% coverage or cannot achieve better coverage. In step 2, the subjects use an executable version of program to execute their test cases and print the output. Subjects then receive official specification and begin step 3. In step 3, the subjects use the specification to observe failures in their output. In step 4, the subjects isolate the faults that caused the observed failures. No special technique is specified for the fault-isolation activity. Finally, subjects hand in a list of observed failures and isolated faults. The requirements of testing techniques for fault detection is shown in Table 3.11.

**Table 3.11:** Requirements for testing techniques

| | Code Reading | Functional Testing | Structural Testing |
|---|---|---|---|
| **View Program Specification** | Y | Y | Y |
| **View Source Code** | Y | N | Y |
| **Execute Program** | N | Y | Y |

### 3.4.3 Programs

No programs from the Kamsties and Lott package were used in the training session. Instead we used some self-coded trivial programs, as training session was more a learning process rather than a pilot study. Those programs were simple enough to understand and they were seeded with almost all types of faults. The three programs we used in the experiment are the same as used by Kamsties and Lott and Roper *et al* in their live experiment. The following programs were used:

1. *cmdline:* evaluates a number of options that are supplied on the command line. The functions in that program fill a data structure with the results of the evaluation, which the driver function prints out upon completion.

2. *nametbl:* implements another abstract data type, namely a simple symbol table. The functions support inserting a symbol, setting two types of attributes of a symbol, searching for a symbol by name, and printing out the contents of the table. Again the driver reads commands from a file to exercise the functions.

3. *ntree:* implements an abstract data type, namely a tree with unbounded branching. The functions support creating a tree, inserting a node as a child of a named parent, searching the tree for a node, querying whether two children are siblings, and printing out the contents of the tree. The driver function reads commands from a file to exercise the functions.

All the programs were written in a C language with which the subjects were familiar. Table 3.12 gives size data for the programs. In addition to compare the relative complexities of the programs, cyclomatic complexity index measures (the number of binary decision points plus one) is also shown for each program. Looking at the source code for the programs, most of them contain no comments. This creates a worst-case situation for the code readers.

**Table 3.12:** Size and other relevant information for programs

| | cmdline.c | cmdline.h | nametbl.c | nametbl.h | ntree.c | ntree.h |
|---|---|---|---|---|---|---|
| **Total Lines** | 245 | 34 | 251 | 31 | 235 | 25 |
| **Blank Lines** | 26 | 5 | 40 | 8 | 38 | 7 |
| **Lines with Comments** | 0 | 0 | 4 | 0 | 4 | 0 |
| **Non blank Non comment Lines** | 219 | 29 | 207 | 23 | 193 | 18 |
| **Preprocessor Directives** | 3 | 17 | 6 | 3 | 5 | 4 |
| **Cyclomatic complexity index** | | | | | | |
| **Minimum** | 1 | | 1 | | 1 | |
| **Mean** | 7 | | 2 | | 3 | |
| **Maximum** | 28 | | 8 | | 8 | |

#### 3.4.3.1 Faults and fault classification

The faults used in our experiment were supplied with Kamsties and Lott package. Most faults present in the program were seeded, although we cannot guarantee that programs do not contain any other faults. All faults cause observable failures; no fault conceals another. The failure might be a total failure (no output at all), a serious problem (incorrect output), or a minor problem (misspelled word in the output). No faults identified by the compiler are taken into consideration. Fault classification was needed to classify the faults. Unfortunately, there are not many classifications in the literature. We also classify the faults using the two-faceted fault-classification scheme from the [Basili and Selby, 1987] experiment. Facet one (type) captures the absence of needed code or the presence of incorrect code (omission, commission). Facet two (class) partitions software faults into the six classes:

1. Initialization
2. Control
3. Computation
4. Interface
5. Data
6. Cosmetic

Thus we can have following combinations of fault type and class:

1. *Initialization (omission and commission):* An initialization fault is an incorrect initialization or non-initialization of a data structure. For example, failure to initialize when necessary would be an error of omission, whereas assigning an incorrect value to a variable when entering a module would be an error of commission.
2. *Control (omission and commission):* A control fault means that the program follows an incorrect control flow path in a given situation. For example, a missing predicate would be fault of omission, whereas assigning an incorrect value to a variable when entering a module would be an error of commission.
3. *Computation (omission and commission):* These are faults that lead to an incorrect calculation. For example, failure to add two numbers is error of omission whereas

an incorrect arithmetic operator on the right-hand side of an assignation would be a computation fault.

4. *Interface (omission and commission):* Interface faults occur when a module uses entities that are outside the modules local environment and assumes things that are untrue. For example, failure to send correct number of arguments to a routine is error of omission, whereas sending incorrect arguments to routine is an error of commission.

5. *Data (omission and commission):* Data faults are faults caused by the incorrect use of a data structure. For example, not terminating a string in *C* is error of omission whereas incorrectly determining the index of the last element in an array.

6. *Cosmetic (omission and commission):* Cosmetic faults of omission are faults where an error message should appear and does not. Cosmetic faults of commission can result, for example, in a spelling mistake in an error message.

There were 8 faults in total in the ntree program, 9 in the cmdline program, and 8 in the nametbl program. Table 3.13 classifies the faults in the programs used in the experiment as per the above classification. Figure 3.1, Figure 3.2 and Figure 3.3 shows the fault distribution for program cmdline, nametbl and ntree respectively. Figure 3.4 shows the collective fault distribution for all the programs.

**Table 3.13:** Count and classification of faults as per adopted classification

| | **cmdline** | **nametbl** | **ntree** | **Total** |
|---|---|---|---|---|
| Omission | 2 | 5 | 4 | 11 |
| Commission | 7 | 3 | 4 | 14 |
| Initialization | 2 | 0 | 0 | 2 |
| Control | 2 | 3 | 3 | 8 |
| Computation | 0 | 1 | 0 | 1 |
| Interface | 3 | 1 | 1 | 5 |
| Data | 1 | 2 | 2 | 5 |
| Cosmetic | 1 | 1 | 2 | 4 |
| Total | 9 | 8 | 8 | 25 |

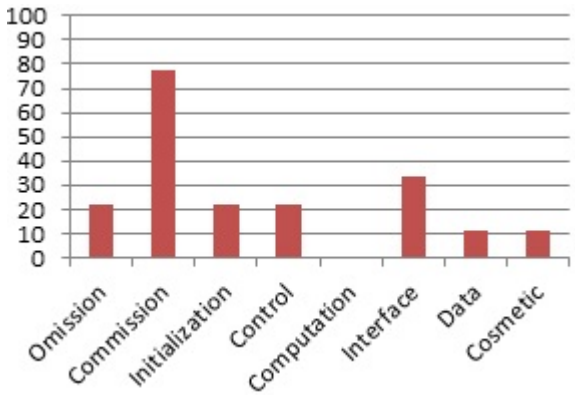



**Figure 3.1:** Fault distribution percentage for cmdline program

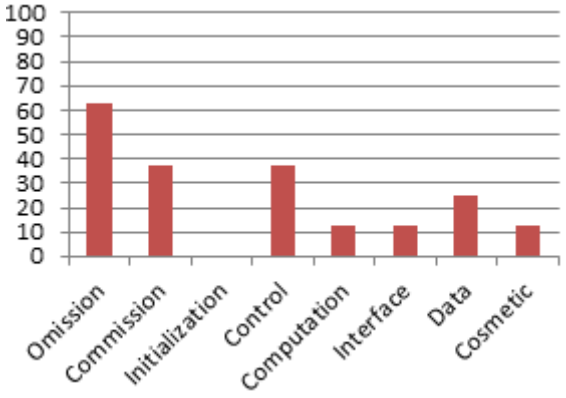



**Figure 3.2:** Fault distribution percentage for nametbl program

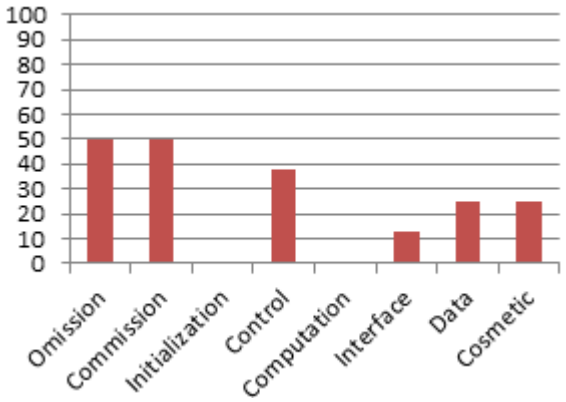



**Figure 3.3:** Fault distribution percentage for ntree program

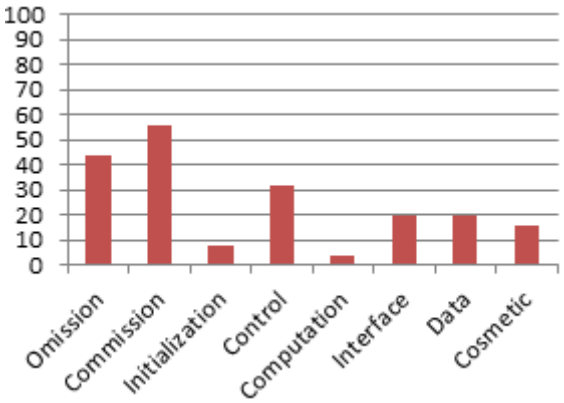



**Figure 3.4:** Fault distribution percentage for all 3 programs

#### 3.4.3.2 Failure counting scheme

There have been various efforts to determine a precise counting scheme for "defects" in software [Basili and Selby, 1987]. A failure is revealed if program output reveals behavior that deviates from the specification, and observed if the subject records the deviate behavior. In code reading, an inconsistency (analog to a failure in the other treatments) is revealed if the subject captures the deviate behavior in his/her own specification, and observed if the subject records an inconsistency between the specifications. Multiple failures (inconsistencies) caused by the same fault are only counted once. For all techniques, a fault is isolated if the subject describes the problem in the source code with sufficient precision. We distinguish between faults isolated by using the technique (i.e., after a failure was revealed and observed) and

faults isolated by chance (i.e., no failure was revealed or observed), as summarized in Table 3.14.

**Table 3.14:** Different defect detection and isolation cases

| Failure revealed? | Failure observed? | Fault isolated? | Explanation |
|---|---|---|---|
| N | N | N | Defect-detection technique failed |
| N | N | Y | Fault was isolated by chance |
| N | Y | N | Meaningless |
| N | Y | Y | Code reading: constructed poor abstractions Functional/structural test: meaningless |
| Y | N | N | Poor evaluation of abstractions/output |
| Y | N | Y | Fault was isolated by chance |
| Y | Y | N | Poor fault isolation |
| Y | Y | Y | Defect-detection technique succeeded |

### 3.4.4 Subjects

Eighteen students participated in the experiment. Subjects joined the experiment for learning benefits especially for acquiring practical knowledge of testing. The subjects of the experiment were an accidental sample (selected on the basis of participation, not a random one) of students of P. G. Department of Computer Sciences at the University of Kashmir. Actually, twenty two students registered themselves for voluntary participation in the experiment however only twenty one were present for training session and later only eighteen students turned up for the experiment session. They were not asked to explain why they left the experiment. The subjects were the sixth semester students pursuing **m**asters in **c**omputer **a**pplications. They were already familiar with the basics of software testing, as they took a related subject (SE) in their fourth semester, although their practical knowledge was quite limited, as no practical training is given in that subject in their course. The subjects had completed two years of programming classes (including classes in C programming).Each group represents a set of people who performed the experiment individually on the same program at the same time applying the same technique. Selection and motivation were driven by a certificate for

all participants and surprise prizes for top 3 performers. No subject was forced to join the experiment. Participation was totally based on the consent of the subjects. They were aware that this experiment is used for research purposes. Subjects were free to withdraw from the experiment at any point of time. Subjects were assured that they would not be judged personally based on their performance.

### 3.4.5 Data Collection and Validation Procedures

The output data (results in the raw form) was collected via data collection forms which were used by Kamsties and Lott in their experiment. However they were changed to make them more suitable for the experiment. After the subjects completed the experiment, we validated their data by arranging a small interactive session with them. This helped us in investigating whether the subjects applied the techniques as prescribed (process conformance), determine whether the subjects understood how to supply the data that were demanded of them (data validity), and check other issues that might cause misleading results. This also helped us in compiling raw data into usable form.

### 3.4.6 Data Analysis Procedures

Parametric techniques for analysis are usually used when randomization has been performed. Based on the randomized approach for matching techniques, programs and subjects, we used a parametric statistics method "ANOVA to test the null hypotheses. The ANOVA procedure was chosen because the design included randomization (thus satisfying a fundamental assumption), and all analyses involved more than two groups of data.

Like [Kamsties and Lott, 1995], we also included intermediate results to make analysis transparent. In all the tables in the subsection 3.6.2, *SS* refers to sum of squares and *df* refers to degree of freedom. In case of between groups column, *SS* refers to the treatment sum of squares and *df* refers to treatment degrees of freedom; whereas, in case of within group column, *SS* refers to the residual sum of squares, and *df* refers to the residual degrees of freedom. All these values are used to make a test against the F distribution to check the variation between the treatments exceeds that of the variation within the treatment; the significance level is an estimate of whether the difference is due to chance. The F value is defined as the ratio of the variance between groups to the variance within groups. Each variance is calculated as the sum of squares divided by its degree of freedom. In short, the F value is computed as: $((SS/df(\text{between groups})/(SS/df(\text{within groups}))$. The computed

value is checked to see the p-value (significance level). An important issue is the choice of significance level which specifies the probability of the result of being realistic. However, it does not specify that the results are highly important. Common practice dictates rejecting the null hypothesis when the significance level is less than or equal to 0.05 [Box et al., 2005]. We also rejected the null hypothesis if we attained a probability value below the generally accepted cut off of 0.05. This refers to a 5% probability of mistakenly rejecting the null hypothesis.

## 3.5 Experiment Procedures

Experiment procedures are actually the implementation of the experiment design. Experimental procedures describe precisely how the experiment will be performed.

### 3.5.1 Training Activities

Before running the experiment, three training sessions of 2 hours each were carried out with the subjects to give them an introduction to our experiment. Most of the subjects had some experience with C programming language. As already mentioned, training session was more a learning process rather than a pilot study. However, we had a good discussion on various aspects of the experiment and also on the testing techniques to be used. Although they had the knowledge of testing techniques; but they had never used them practically. The training session gave them a good insight into the working and purpose of the experiment.

### 3.5.2 Conducting the Experiment

The live experiment was run in three parallel sessions on three consecutive days. All subjects were aware of the 4 hours (240 minutes) time limitation for carrying out the task. The experiment was conducted at Department of Computer Science, UoK in the year 2011. All the 18 subjects participated in the experiment till the very end.

#### 3.5.2.1 Threats to validity

We tried our best to eliminate maximum threats to validity. In spite of that, some threats to validity could not be excluded.

1. Maturation effects like learning have been eliminated, as each subject applies a different technique to different program. However, the subjects comprehension of the C language could improve during the experiment.

2. In addition to that, there can be some fatigue effects causing subjects to lose motivation. There was no way to measure it; however, we tried to minimize it by using a different set of technique and program each day and by working for only 4 hours a day.

3. Selection effects are minimized by random match of subject and group and the order of the applying the testing techniques. In addition to that, all the subjects are equally experienced and have similar academic background. There are some instrumentation effects as all programs have different complexities and contain different types of faults; analysis of variance attempts to measure this effect.

4. External threats to validity are more dominant in our case. Subjects, programs and faults do not truly represent actual software engineering practice. These threats can be minimized by replicating the experiment using true representatives of these factors.

#### 3.5.2.2 Giving feedback to subjects

A feedback session was held 2 months after the experiment. The feedback session was held late because of the unavailability of the subjects. We thanked all the subjects and the results of the preliminary analyses were presented. The subjects also asked many questions and in return thanked us for the practical knowledge they gained through this experiment.

## 3.6 Results

The final outcomes of any experiment are the raw data, results of any inferential statistical analyses performed on the raw data, and interpretations of the statistical analyses [Lott and Rombach, 1996].

### 3.6.1 Raw Data

Raw data is unprocessed data collected from the sources and is not subjected to any processing or manipulation like mathematical or statistical treatments. It is also known as source data or atomic data. In our case, the raw data was collected from the experiment on the data forms. Raw data requires selective extraction, organization, and sometimes analysis and formatting

for presentation before it can be used. Table 3.15[1] and Table 3.16[2] present the raw data of the experiment. The pair of numbers in each block in Table 3.15 represents the number of failures observed and the time in minutes that was taken to reveal and observe those failures. The pair of numbers in Table 3.16 represents the number of faults isolated and the time taken to isolate those faults.

**Table 3.15:** Raw data for effectiveness

| | | Day 1: cmdline | | | Day 2: nametbl | | | Day 3: ntree | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **Subject** | **Group** | **CR** | **FT** | **ST** | **CR** | **FT** | **ST** | **CR** | **FT** | **ST** |
| 1 | 2 | | 5,120 | | | | 4,105 | 6,152 | | |
| 2 | 2 | | 4,94 | | | | 3,152 | 4,128 | | |
| 3 | 3 | | | 3,121 | 3,164 | | | | 4,160 | |
| 4 | 2 | | 3,81 | | | | 6,164 | 6,142 | | |
| 5 | 1 | 5,126 | | | | 3,98 | | | | 5,111 |
| 6 | 3 | | | 8,154 | 4,152 | | | | 5,148 | |
| 7 | 3 | | | 5,143 | 5,149 | | | | 6,152 | |
| 8 | 1 | 6,143 | | | | 5,127 | | | | 4,122 |
| 9 | 1 | 5,128 | | | | 3,105 | | | | 6,109 |
| 10 | 2 | | 6,102 | | | | 4,94 | 7,152 | | |
| 11 | 3 | | | 6,72 | 3,82 | | | | 5,127 | |
| 12 | 2 | | 7,114 | | | | 5,132 | 7,160 | | |
| 13 | 2 | | 5,117 | | | | 5,142 | 5,129 | | |
| 14 | 1 | 7,132 | | | | 6,124 | | | | 7,162 |
| 15 | 3 | | | 3,86 | 5,128 | | | | 4,81 | |
| 16 | 1 | 2,112 | | | | 4,97 | | | | 7,167 |
| 17 | 3 | | | 5,92 | 3,97 | | | | 5,161 | |
| 18 | 1 | 3,102 | | | | 5,104 | | | | 6,164 |

### 3.6.2 Interpretation

Researchers should describe their results clearly, and in a way that other researchers can compare them with their own results. Results need to be interpreted in an objective and critical way, before assessing their implications and before drawing conclusions.

[1]CR stands for Code Reading, FT stands for Functional Testing and ST stands for Structural Testing
[2]CR stands for Code Reading, FT stands for Functional Testing and ST stands for Structural Testing

**Table 3.16:** Raw data for efficiency

| | | Day 1: cmdline | | | Day 2: nametbl | | | Day 3: ntree | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **Subject** | **Group** | **CR** | **FT** | **ST** | **CR** | **FT** | **ST** | **CR** | **FT** | **ST** |
| 1 | 2 | | 2,62 | | | | 3,32 | 5,27 | | |
| 2 | 2 | | 3,47 | | | | 3,49 | 4,18 | | |
| 3 | 3 | | | 1,26 | 3,28 | | | | 3,63 | |
| 4 | 2 | | 3,38 | | | | 4,46 | 5,20 | | |
| 5 | 1 | 5,24 | | | | 3,38 | | | | 3,32 |
| 6 | 3 | | | 5,39 | 3,35 | | | | 5,81 | |
| 7 | 3 | | | 3,41 | 5,46 | | | | 5,79 | |
| 8 | 1 | 6,21 | | | | 4,82 | | | | 3,45 |
| 9 | 1 | 5,22 | | | | 3,78 | | | | 4,38 |
| 10 | 2 | | 4,56 | | | | 3,44 | 6,28 | | |
| 11 | 3 | | | 3,26 | 3,21 | | | | 3,82 | |
| 12 | 2 | | 6,67 | | | | 5,65 | 7,22 | | |
| 13 | 2 | | 4,54 | | | | 4,43 | 5,29 | | |
| 14 | 1 | 7,18 | | | | 5,73 | | | | 6,47 |
| 15 | 3 | | | 3,23 | 4,34 | | | | 3,47 | |
| 16 | 1 | 2,27 | | | | 4,68 | | | | 4,53 |
| 17 | 3 | | | 4,32 | 3,12 | | | | 5,79 | |
| 18 | 1 | 3,17 | | | | 4,76 | | | | 6,42 |

#### 3.6.2.1 Evaluation of failure observation effectiveness

For evaluating the effectiveness of software testing techniques for observing failures, we can either accept or reject the null hypothesis $N_{1.1}$: None of the independent variables (technique, program, group, or subject) affects the percentage of total possible failures observed. As

**Table 3.17:** Analysis of variance of percentage of failures observed

| **Independent Variable** | **Mean percentage of total failures observed** | **Between Groups** | | **Within Groups** | | **F** | **Sig. level** |
|---|---|---|---|---|---|---|---|
| | | *SS* | *df* | *SS* | *df* | | |
| **Technique** | CR=57.56 FT=56.71 ST=61.57 | 242.698 | 2 | 14392.25 | 51 | 0.43001 | 0.652 |
| **Program (Day)** | cm=54.32 na=52.77 nt=68.75 | 2794.139 | 2 | 11840.81 | 51 | 6.01736 | 0.004 |
| **Group** | G1=59.64 G2=61.57 G3=55.16 | 388.588 | 2 | 15564.34 | 51 | 0.636648 | 0.533 |
| **Subject** | 60.18, 43.98, 40.27, 61.11, 51.85, 67.12, 64.35, 59.72, 56.01, 68.05, 47.2, 75.92, 60.18, 80.09, 48.61, 53.24, 51.85, 56.94 | 5622.607 | 17 | 9431.584 | 36 | 1.262428 | 0.270 |

shown in Table 3.17 we can reject null hypothesis with respect to program only. Effectiveness on day 1 and day 2 are almost same; however a 30% increase in effectiveness on day 3 for ntree program certainly indicates some learning effect. Null hypothesis is accepted in case of technique, group and subject which suggest that the testing techniques were equally effective in case of failure revelation and observation. The individual statistics for independent variable program in case of failure observation are shown in Figure 3.5.

#### 3.6.2.2 Evaluation for fault isolation effectiveness

For evaluating the effectiveness of software testing techniques for isolating faults, we can either accept or reject the null hypothesis $N_{1.2}$: None of the independent variables (technique, program, group, or subject) affects the percentage of total faults isolated. We can reject null hypothesis again with respect to program only. Increase in effectiveness in isolating faults on each day certainly suggests a learning effect. However, as shown in Table 3.18, the

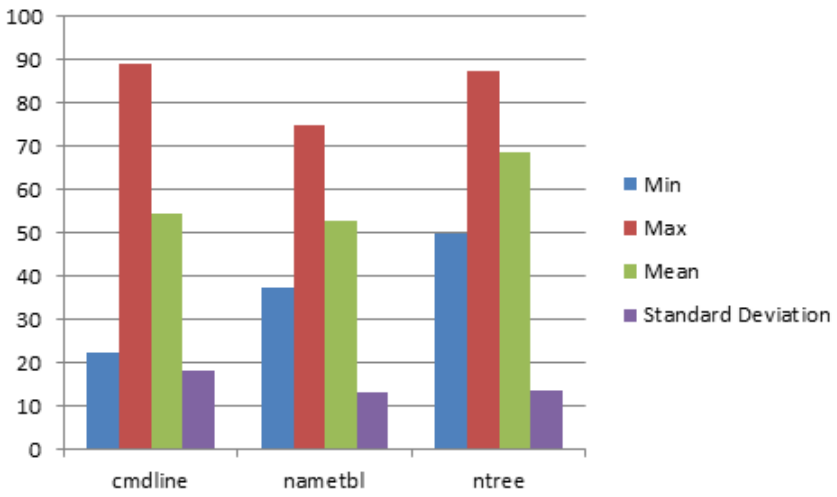



**Figure 3.5:** Statistics of program variable for failure observation effectiveness

**Table 3.18:** Analysis of variance of percentage of faults isolated

| **Independent Variable** | **Mean percentage of total faults isolated** | | | **Between Groups** | | **Within Groups** | | **F** | **Sig. level** |
|---|---|---|---|---|---|---|---|---|---|
| | | | | *SS* | *df* | *SS* | *df* | | |
| **Technique** | CR=54.08 | FT=46.14 | ST=44.52 | 944.0015 | 2 | 11340.66 | 51 | 2.11263 | 0.130 |
| **Program (Day)** | cm=42.59 | na=45.83 | nt=56.94 | 2039.609 | 2 | 10941.36 | 51 | 4.75352 | 0.012 |
| **Group** | G1=51.31 | G2=51.08 | G3=42.97 | 810.828 | 2 | 12170.14 | 51 | 1.69892 | 0.193 |
| **Subject** | 40.74, 40.27, 28.70, 48.61, 43.51 51.85, 52.77, 51.38, 47.68, 52.31 36.11, 72.22, 52.31, 71.75, 40.27 40.74, 48.14, 52.77 | | | 6046.811 | 17 | 6934.15 | 36 | 1.84665 | 0.601 |

increase factor is not uniform. The individual statistics for independent variable program for percentage of faults isolated are shown in Figure 3.6. Null hypothesis is accepted in case of technique and group which suggest that the testing techniques were equally effective in case of fault isolation.

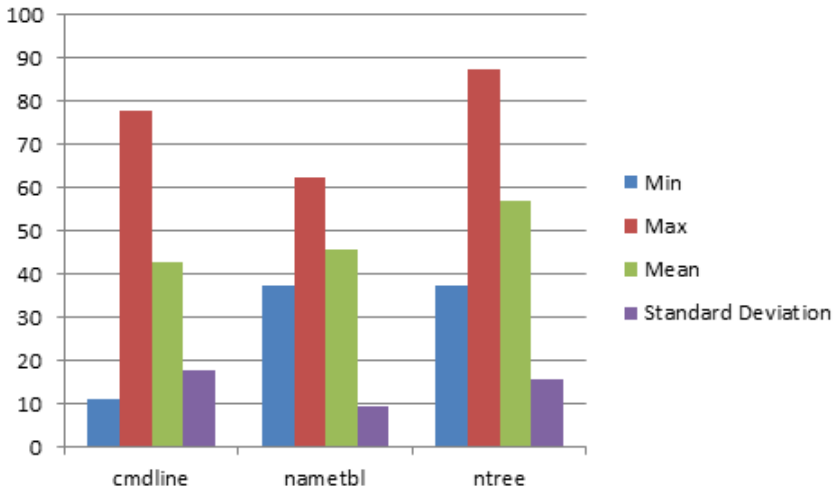



**Figure 3.6:** Statistics of program variable for fault isolation effectiveness

#### 3.6.2.3 Evaluation of time taken to observe failures

For evaluating the software testing techniques for time taken to observe failures we can either accept or reject the null hypothesis $\mathbf{N}_{1.3}$**:** None of the independent variables (technique, program, group, or subject) affects the time taken to observe the failures. We can reject null

**Table 3.19:** Analysis of variance of failure-observation time

| **Independent Variable** | **Mean failure observation time** | | | **Between Groups** | | **Within Groups** | | **F** | **Sig. level** |
|---|---|---|---|---|---|---|---|---|---|
| | | | | *SS* | *df* | *SS* | *df* | | |
| **Technique** | CR=132.11 | FT=117.33 | ST=127.33 | 2047.25 | 2 | 34857.78 | 51 | 1.49766 | 0.233 |
| **Program (Day)** | cm=113.27 | na=123.11 | nt=140.38 | 6781.37 | 2 | 30123.67 | 51 | 5.7405 | 0.005 |
| **Group** | G1=124.05 | G2=126.66 | G3=126.05 | 67.1481 | 2 | 36837.89 | 51 | 0.04648 | 0.954 |
| **Subject** | 125.66, 124.66, 148.33, 129.00 111.66, 151.33, 148.00, 130.66, 114.00, 116.00, 93.66, 135.33, 129.33, 139.33, 98.333, 125.33, 116.66, 123.33 | | | 12855.7 | 17 | 24049.33 | 36 | 1.132 | 0.364 |

hypothesis with respect to program as shown in Table 3.19 . Increase in the time taken to reveal and observe the failures suggest the instrumentation effect and indicate the difference in the complexity of the programs. Figure 3.7 shows individual statistics for independent variable program (day). Null hypothesis is accepted in case of technique, group and subject which suggest that the testing techniques does not differ in terms of the time taken to reveal and observe the failures.

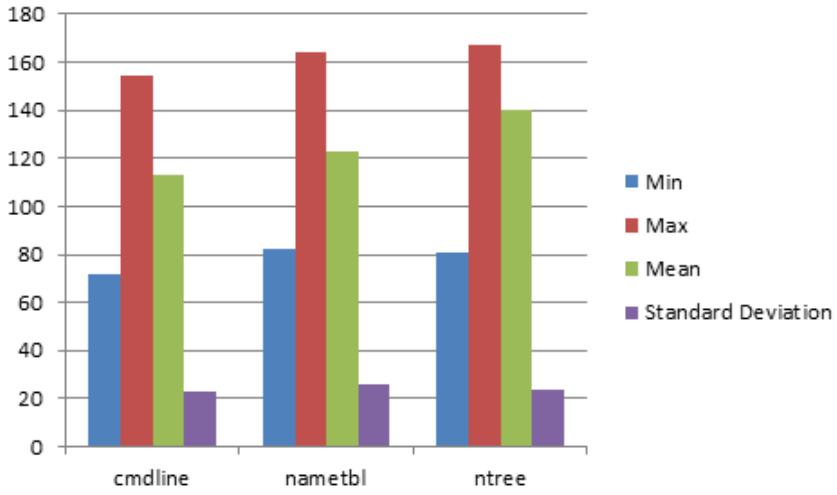



**Figure 3.7:** Statistics of program variable for failure observation time

#### 3.6.2.4 Evaluation of time taken to isolate faults

For evaluating the software testing techniques for time taken to isolate faults we can either accept or reject the null hypothesis $\mathbf{N}_{1.4}$**:** None of the independent variables (technique, program, group, or subject) affects the time taken to isolate the faults of the observed failures. We can reject null hypothesis with respect to technique only as shown in Table 3.20 . The results suggest that code reading took very less time to isolate time followed by structural testing, functional testing took the longest time ($CR < ST < FT$) which is obvious because the code readers take long time to develop their own specifications and identify inconsistencies, after which they isolated faults rapidly, whereas the functional testers reveal and observe failures rapidly but then they usually take more time to isolate faults because they had to comprehend the code for that purpose. The individual statistics for each technique is shown in Figure 3.8. Null hypothesis is accepted in case of program, group and subject which suggest that they had no effect on the time taken to isolate the faults.

**Table 3.20:** Analysis of variance of fault-isolation time

| **Independent Variable** | **Mean fault isolation time** | | | **Between Groups** | | **Within Groups** | | **F** | **Sig. level** |
|---|---|---|---|---|---|---|---|---|---|
| | | | | *SS* | *df* | *SS* | *df* | | |
| **Technique** | CR=24.94 | FT=65.00 | ST=40.16 | 14717.15 | 2 | 6877.44 | 51 | 54.5678 | 0.000 |
| **Program (Day)** | cm=35.55 | na=48.33 | nt=46.22 | 1689.03 | 2 | 19905.56 | 51 | 2.16374 | 0.125 |
| **Group** | G1=44.50 | G2=41.50 | G3=44.11 | 95.814 | 2 | 21498.78 | 51 | 0.011364 | 0.892 |
| **Subject** | 40.33, 38.00, 39.00, 34.66, 31.33, 51.66, 55.33, 49.33, 46.00, 42.66, 43.00, 51.33, 42.00, 46.00, 34.66, 49.33, 41.00, 45.00 | | | 2173.926 | 17 | 19420.67 | 36 | 0.23704 | 0.998 |

#### 3.6.2.5 Evaluation of total time (detection time + isolation time)

With respect to the total time taken to detect failures and isolate faults, we can either accept or reject the null hypothesis $\mathbf{N}_{1.5}$**:** None of the independent variables (technique, program, group, or subject) affects the total time which includes failure observation time and fault detection time. We can reject null hypothesis with respect to program only as shown in Table 3.21 . The total time taken by subjects to detect and isolate faults increased on each day which clearly suggests the presence of an instrumentation effect. The result supports the statement of subjects that programs significantly differed in terms of complexity. The individual statistics for total time with respect to variable program (day) is shown in Figure 3.9. Null hypothesis is accepted in case of technique, group and subject.

#### 3.6.2.6 Evaluation of efficiency in observing failures

For evaluating the efficiency of the testing techniques in case of failure observation, we can either accept or reject the null hypothesis $\mathbf{N}_{1.6}$**:** None of the independent variables (technique, program, group, or subject) affects the rate of failure observation. In this case, we have to accept null hypothesis for all the factors as shown in Table 3.22. The results, in principle imply that none of the independent variables affect the mean failure observation rate. However, as

**Table 3.21:** Analysis of variance for total time

| **Independent Variable** | **Mean total time** | | | **Between Groups** | | **Within Groups** | | **F** | **Sig. level** |
|---|---|---|---|---|---|---|---|---|---|
| | | | | *SS* | *df* | *SS* | *df* | | |
| **Technique** | CR=157.61 | FT=182.33 | ST=167.50 | 5574.037 | 2 | 59900.78 | 51 | 2.3728 | 0.103 |
| **Program (Day)** | cm=149.38 | na=171.44 | nt=186.61 | 12611.81 | 2 | 52863 | 51 | 6.0836 | 0.004 |
| **Group** | G1=169.11 | G2=168.16 | G3=170.16 | 36.03704 | 2 | 65438.78 | 51 | 0.014043 | 0.986 |
| **Subject** | 166.00, 162.66, 187.33, 163.66, 143.00, 203.00, 203.33, 180.00, 160.00, 158.66, 136.66, 186.66, 171.33, 188.66, 133.00, 174.66, 157.66, 168.33 | | | 20818.81 | 17 | 44656 | 36 | 0.998725 | 0.492 |

**Table 3.22:** Analysis of variance of Mean failure observation rate

| **Independent Variable** | **Mean failure observation rate** | | | **Between Groups** | | **Within Groups** | | **F** | **Sig. level** |
|---|---|---|---|---|---|---|---|---|---|
| | | | | *SS* | *df* | *SS* | *df* | | |
| **Technique** | CR=2.157 | FT=2.461 | ST=2.496 | 1.25106 | 2 | 22.5782 | 51 | 1.41295 | 0.252 |
| **Program (Day)** | cm=2.631 | na=2.103 | nt=2.380 | 2.5122 | 2 | 21.3170 | 51 | 3.005 | 0.058 |
| **Group** | G1=2.372 | G2=2.452 | G3=2.290 | 0.23578 | 2 | 23.59175 | 51 | 0.25680 | 0.774 |
| **Subject** | 2.38, 1.87, 1.36, 2.31, 2.30, 2.24, 2.15,2.28, 2.45, 2.94, 3.18, 2.86, 2.33, 2.89, 2.46, 2.02, 2.32, 2.28 | | | 9.01475 | 17 | 14.81458 | 36 | 1.2886 | 0.254 |

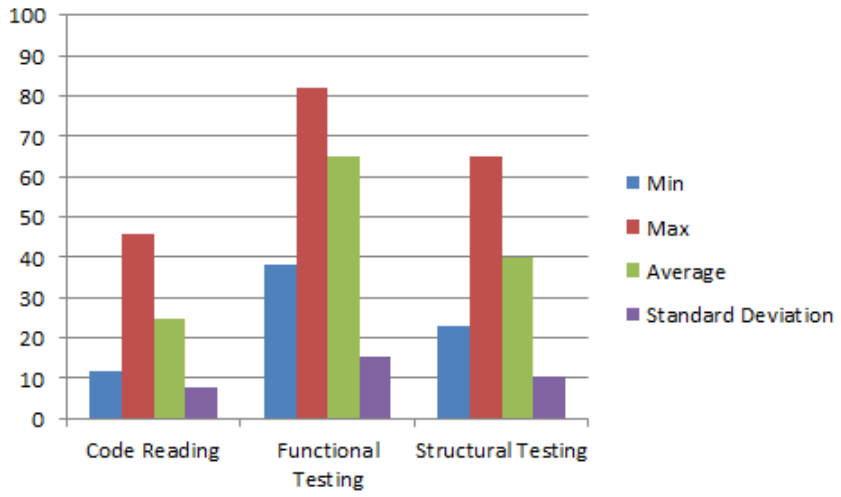



**Figure 3.8:** Statistics of technique variable for failure isolation time

the significance value for program is just above 0.05 level, we will consider it as significant. The results therefore, suggest that the program affect the rate of failure observation. The individual statistics for program (day) variable are shown in Figure 3.10. Null hypothesis is accepted in case of technique, group and subject

#### 3.6.2.7 Evaluation of efficiency in isolating faults

For evaluating the efficiency of the testing techniques in case of fault isolation, we can either accept or reject the null hypothesis $\mathbf{N}_{1.7}$**:** None of the independent variables (technique, program, group, or subject) affects the rate of fault isolation. We can reject the null hypothesis with respect to the technique. Code reading was the most efficient followed by structural testing technique. Functional testing was the least efficient. However we accept null hypothesis with respect to program, group and subject as shown in Table 3.23 . The individual statistics for variable technique are shown in Figure 3.11.

#### 3.6.2.8 Evaluation of effectiveness of failures observed for each fault class

For evaluating the effectiveness of testing techniques in terms of failure observation for each fault class and type, we can either accept or reject the null hypothesis $\mathbf{N}_{1.8}$**:** Techniques are equally effective at observing failures caused by the various types and classes of faults. We can reject the null hypothesis with respect to cosmetic faults as shown in Table 3.24 where code reading was out performed by structural testing. Functional testing was the most effective in terms of observing cosmetic faults.

**Table 3.23:** Analysis of variance of Mean fault isolation rate

| **Independent Variable** | **Mean fault isolation rate** | | | **Between Groups** | | **Within Groups** | | **F** | **Sig. level** |
|---|---|---|---|---|---|---|---|---|---|
| | | | | *SS* | *df* | *SS* | *df* | | |
| **Technique** | CR=11.783 | FT=3.641 | ST=5.674 | 646.443 | 2 | 323.2218 | 51 | 32.91114 | 0.000 |
| **Program (Day)** | cm=7.930 | na=5.460 | nt=7.701 | 67.2756 | 2 | 1080.041 | 51 | 1.58839 | 0.214 |
| **Group** | G1=7.728 | G2=7.508 | G3=5.863 | 37.3891 | 2 | 1109.92 | 51 | 0.8589 | 0.429 |
| **Subject** | 6.22, 6.94, 3.86, 8.31, 7.62, 5.51, 4.90, 8.02, 7.41, 7.07, 5.89, 9.69, 6.79, 11.70, 6.23, 4.16, 8.76, 7.43 | | | 188.71 | 17 | 958.6041 | 36 | 0.41688 | 0.971 |

**Table 3.24:** Analysis of variance of percentage of observed failures caused by faults from each fault class and type

| **Fault Class** | **Mean percentage of failures observed** | | | **Between Groups** | | **Within Groups** | | **F** | **Sig. level** |
|---|---|---|---|---|---|---|---|---|---|
| | **CR** | **FT** | **ST** | *SS* | *df* | *SS* | *df* | | |
| **Omission** | 63.05 | 61.94 | 73.61 | 1492.593 | 2 | 35754.17 | 51 | 1.0645 | 0.35 |
| **Commission** | 58.59 | 61.24 | 61.50 | 93.2224 | 2 | 34358.78 | 51 | 0.0691 | 0.933 |
| **Initialization** | 22.22 | 16.66 | 25 | 648.148 | 2 | 67361.11 | 51 | 0.24536 | 0.783 |
| **Control** | 58.33 | 60.18 | 67.59 | 864.197 | 2 | 34922.84 | 51 | 0.63102 | 0.5361 |
| **Computation** | 27.77 | 33.33 | 33.33 | 370.37 | 2 | 116111.1 | 51 | 0.08134 | 0.922 |
| **Interface** | 68.51 | 61.11 | 61.11 | 658.43 | 2 | 10482 | 51 | 0.160852 | 0.851849 |
| **Data** | 69.44 | 50 | 52.77 | 3981.481 | 2 | 93055.56 | 51 | 1.0910 | 0.3435 |
| **Cosmetic** | 41.66 | 88.88 | 77.77 | 21944.44 | 2 | 70138.89 | 51 | 7.97821 | 0.000 |

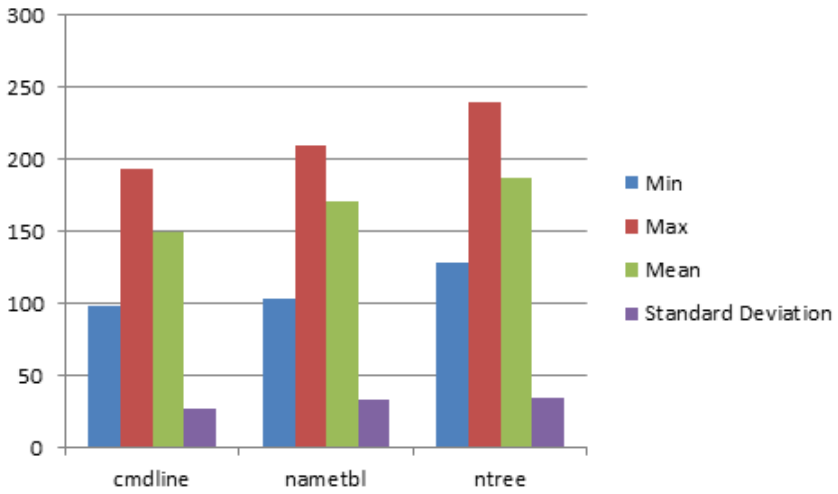



**Figure 3.9:** Statistics of program variable for total time

#### 3.6.2.9 Evaluation of effectiveness of faults isolated for each fault class

For evaluating the effectiveness of testing techniques in terms of fault isolation for each fault class and type, we can either accept or reject the null hypothesis $\mathbf{N}_{1.9}$**:** Techniques are equally effective at isolating faults of the various types and classes of faults. We can reject the null hypothesis with respect to cosmetic faults as shown in Table 3.25. In this case, functional testing isolated maximum number of faults followed by structural testing which was followed by code reading.

## 3.7 Summary

With respect to effectiveness of failure detection, our results suggest it depends on the program not on the technique. The effectiveness of fault isolation also depends on the program only. The time taken to observe the failures depends on the program only. However, the time taken to isolate faults depends on the technique; the results suggest that functional testing using boundary value analysis took the longest time while the code reading took very less time to isolate the faults from observed failures ($CR < ST < FT$). The total time depends only on the program. The efficiency of failure observation (mean failure observation rate) depends on the program only. However, the efficiency of failure isolation (mean failure isolation rate) depends on the technique only; code reading performs better than functional and structural testing ($CR > ST > FT$). With respect to fault type and class detected and

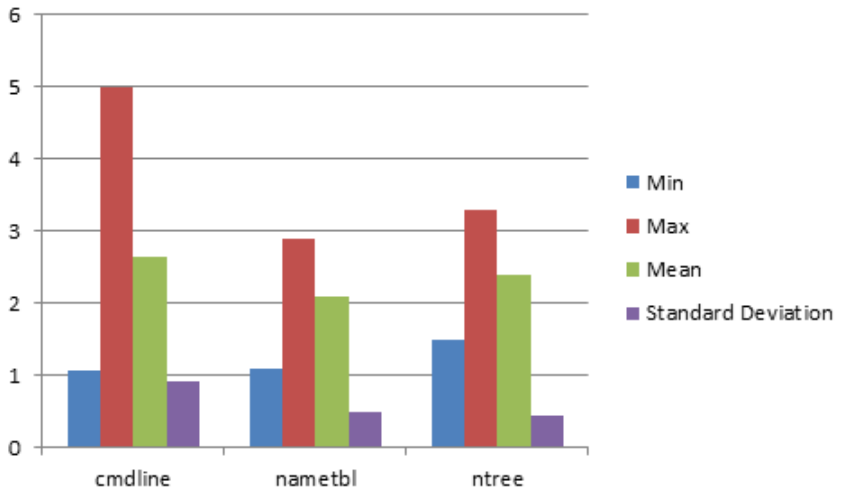



**Figure 3.10:** Statistics of program variable for failure observation efficiency

**Table 3.25:** Analysis of variance of percentage of isolated faults from each fault class and type

| **Fault Class** | **Mean percentage of faults isolated** | | | **Between Groups** | | **Within Groups** | | **F** | **Sig. level** |
|---|---|---|---|---|---|---|---|---|---|
| | **CR** | **FT** | **ST** | *SS* | *df* | *SS* | *df* | | |
| **Omission** | 59.16 | 56.38 | 56.38 | 92.59 | 2 | 33093 | 51 | 0.071348 | 0.931 |
| **Commission** | 55.35 | 45.96 | 44.51 | 1247.79 | 2 | 27748.09 | 51 | 1.14670 | 0.325 |
| **Initialization** | 22.22 | 5.55 | 11.11 | 2592.59 | 2 | 33333.33 | 51 | 1.98333 | 0.148 |
| **Control** | 54.62 | 48.14 | 52.77 | 401.234 | 2 | 26635.8 | 51 | 0.38412 | 0.683 |
| **Computation** | 22.22 | 33.33 | 33.33 | 1481.481 | 2 | 111111.1 | 51 | 0.34 | 0.713 |
| **Interface** | 68.51 | 51.85 | 33.33 | 11152.26 | 2 | 112654.3 | 51 | 2.5243 | 0.090 |
| **Data** | 63.88 | 38.88 | 36.11 | 8425.92 | 2 | 95833.33 | 51 | 2.2420 | 0.116 |
| **Cosmetic** | 41.66 | 88.88 | 75 | 21203.7 | 2 | 70277.78 | 51 | 7.6936 | 0.001 |

isolated, all testing techniques were equally effective except in case of cosmetic faults where functional testing was most effective and code reading was least effective ($FT > ST > CR$).

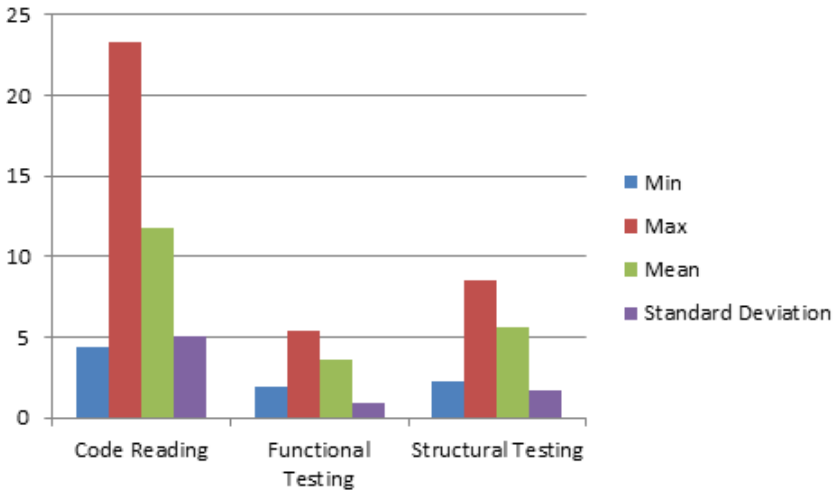



**Figure 3.11:** Statistics of technique variable for fault isolation efficiency

## 3.8 Conclusion and Future Work

The experiment was carried to evaluate the effectiveness and efficiency of the software testing techniques. We conclude that static testing technique (code reading) and dynamic testing technique (functional and structural testing) are all equivalent in terms of effectiveness. We failed to observe a significant difference between testing techniques which is very much in line with the results of the related work discussed in Section 3.3.3 with an exception of [Basili and Selby, 1987]. However, code reading performed well in fault isolation time and fault isolation rate as compared to dynamic testing techniques. Therefore we can conclude that techniques differ partially in terms of efficiency. The effect of the technique was verified only for the fault isolation time and fault isolation rate. The effect of the program was very significant in all cases, except in case of fault isolation time and fault isolation efficiency. The group (order) and subjects had no significant effect on the effectiveness or efficiency. In case of fault types, all testing techniques were equally effective in terms of detection and isolation for all types of fault classes and types except for cosmetic faults. We agree with the results of the previous experiments that the effectiveness and efficiency depends on the program and the faults. Rightly said by Roper et al, as the programs and faults vary, so do the results [Roper et al., 1997].

Our effort was to provide empirical evidence that will help testing community to understand the techniques effectiveness and efficiency and its dependence on various factors. We believe that few experiments are not sufficient to understand this phenomenon. Further repli-

cations of this work will help in making some strong conclusion regarding the effectiveness and efficiency of software testing techniques and other allied factors. However, carrying out experiment in accordance with the given schema is necessary for the reasons mentioned in chapter 2 of this thesis. The ultimate goal of our work is to move software engineering from a craft towards an engineering discipline.

# Chapter 4

# Evaluating Software Testing Techniques for Reliability

## 4.1 Introduction

Software reliability is a quantitative measure used in accessing the quality of software and is significantly considered in software development. Software reliability is a key quality attribute as identified by quality models like ISO, McCall, and FURPS etc. It is a user-oriented view of software quality and is also most readily quantified and measured. Software reliability is defined as the probability of software to perform its required functions (output that agrees with specifications) without failure under stated conditions (specified environment) for a specified period of time. Mathematically, reliability R(t) is the probability that a system will be successful in the interval from time 0 to time t:

$$R(t) = P(T > t), t \geq 0$$

*Where T is a random variable denoting the time to failure or failure time.*

Unreliability F(t), a measure of failure, is defined as the probability that the system will fail by time t:

$$F(t) = P(T \leq t), t \geq 0.$$

The Reliability R (t) is related to failure probability F (t) by:

$$\mathrm{R(t)} = 1 - \mathrm{F(t)}.$$

Developing reliable software is one of the most difficult problems facing the software industry. Schedule pressure, resource limitations, unrealistic requirements and many other factors can all negatively impact software reliability [Wood, 1997]. To achieve reliability we should use software reliability engineering techniques. Having attracted major attentions in past years from academic circles as well as industry, software reliability engineering techniques are classified into following areas: fault avoidance, fault removal, fault tolerance, and fault prediction [Lyu et al., 1996].

Traditionally, software reliability is achieved by fault removal techniques (verification and validation techniques) which include software testing, to detect and eliminate software faults. As the main fault removal technique, software testing is one of the most effort intensive and time consuming activity during software development [Beizer, 1990]. One way to achieve 100% reliability in software is to go for exhaustive testing. Exhaustive testing which tests a system with all possible inputs (valid as well as invalid), is generally not applicable, as the input domain is usually very large, even infinite. So we have to use other testing methods

in order to bring the software to an acceptable level of quality. We would like to select a testing technique that will detect maximum possible detects in an efficient manner. However, fault finding ability measure is only useful for evaluating testing techniques when the goal of testing is to gain confidence that the program is free from faults. Fault detection does not necessarily inspire confidence. It is not necessary that a software testing method which will find most faults in the software will also increase the reliability more than other methods. The impact of the failure should be taken into account. Most previous evaluations of the effectiveness of software testing techniques have considered all failures to be equivalent to one another, regardless of their severity. We all know that one important goal of software testing should be to measure the dependability of tested software and also to increase it. Alternatively, another main objective of testing is to increase our confidence in the software.

So, there are two main goals in testing software: To achieve adequate quality (systematic testing/debug testing); the objective is to probe the software for defects so that these can be removed and to assess existing quality (statistical testing); the objective is to gain confidence that the software is reliable. The names are arbitrary, and most testing techniques address both goals to some degree [Frankl et al., 1998]. So it is evident that debug testing also has the ability to reduce the risk in software by increasing the reliability of the software under test. In order to increase reliability, we should not only strive to discover maximum defects possible, we should also strive to expose those faults which affect reliability most. One sure thing is that we have to do systematic testing for testing a software as it is necessary for things like functionality, correctness etc. There is multitude of software testing techniques which can be used in systematic testing. Our approach should be to select a technique which has the maximum potential to increase the confidence in the software. But as mentioned in chapter 2 we do not have adequate knowledge about their relative quantitative and qualitative statistics. Therefore, it will be interesting to evaluate effectiveness of different testing techniques for reliability. In other words, testing techniques should be evaluated and compared in terms of effectiveness in detecting different failures types, and then checking the reliability increase. In chapter 3, we concluded that techniques are equally effective at observing faults and isolating failures. But that does not mean that they find same types of failures. We are unaware of their capability of increasing the confidence in the software. We would like to be able to compare testing strategies in a way that allows us to say that if a system has been tested using technique T1, it is likely to have less risk (more reliable) associated with its use than if it has been tested using technique T2. This will also help us to understand what types of

faults are mostly targeted by a particular testing technique. So basically we will try to find answers to following research questions:

1. Q1. Does testing methods differ from each other in terms of reliability improvement?
2. Q2. If yes, which among them enhances reliability most?

This chapter is organized as follows: Section 4.2 presents the background of this study, Section 4.3 presents the related work, Section 4.4 discusses statistical and systematic testing, Section 4.5 gives the description of the experiment which includes testing techniques, programs and faults and methodology used in the experiment which are presented in subsections 4.5.1, 4.5.2 and 4.5.3 respectively. Section 4.6 presents the experiment procedure and the results. Threats to validity are discussed in section 4.7. Section 4.8 discusses the experiment and results and Section 4.9 presents conclusion and future work of the experiment.

## 4.2 Background

An important aspect of testing is to make software quality and its characteristics visible which include the reliability of the software. Software testing has been widely used to improve software reliability, not only because verification is not practical yet at this stage, but also because sophisticated testing techniques have been developed to meet different levels of reliability requirements. As we already know software testing has many goals which include the goal of increasing confidence (reducing risk) in the software. Different goals of testing to assess risk can be distinguished as [Frankl and Weyuker, 2000]:

1. **Testing to detect risk:** In addition to counting the number of failures that occur during testing, one keeps track of the cost of those failures. Testing technique A will be considered more effective than testing technique B if the (expected) total cost of failures detected during test is higher for A than for B.
2. **Testing and debugging to reduce risk:** It is further assumed that each failure that occurs leads to the correction of the fault that caused that failure, thereby reducing the risk associated with the corrected software and results in the increase of software reliability. Testing technique A will be considered more effective than testing technique B if A reduces the risk more than B does, thus resulting in less risky software.

3. **Testing to estimate risk:** In estimating software reliability, it is assumed that some faults will remain in the software. The goal is to estimate the probability that the software will fail after deployment (during some specified time). Here, we will say that testing technique A is better than testing technique B (for a given technique of estimating risk) if A provides more accurate estimates of risk than B.

We are interested in evaluating software testing techniques for their ability to detect and reduce the risk. It is not sufficient to detect the failures to reduce risk, as only by removing root cause of the failures (faults) we can reduce the risk in the program.

## 4.3 Related Work

Most previous studies of the effectiveness of testing methods used the probability of causing a failure, and thus finding a defect, as a measure of the effectiveness of a test series [Pizza and Strigini, 1998]. They have employed such measures of test effectiveness as the likelihood of discovering at least one fault (i.e., the likelihood of at least one failure occurring), the expected number of failures that occur during test, the number of seeded faults discovered during test, and the mean time until the first failure, or between failures. This seems inappropriate when considering testing as a means for improving confidence in the software: what really matters is the reliability of the delivered software, hence the improvement that is obtained by applying the given testing method. [Li and Malaiya, 1994] [Frankl et al., 1997] and [Frankl et al., 1998] instead adopted a different measure of test effectiveness; the increment in reliability that would be obtained. Many studies in the past have incorporated the concept of weightage/cost into the evaluation of testing techniques. In the work of [Tsoukalas et al., 1993], [Gutjahr, 1995] and [Ntafos, 1997] the input domain is divided, using some partition testing strategy, and a cost $c_i$ is associated a priori with each sub domain. A failure of any element of the $i_{th}$ sub domain is assumed to cost $c_i$. However, this is not a realistic approximation of reality. In general, for the sub domains induced by the testing strategies commonly studied and in use, the consequences of failure for two different elements of the same sub domain may be very different. Furthermore, most sub domain testing strategies involve sub domains that intersect. [Weyuker, 1996] used cost/consequence of failure as the basis for an automatic test case generation algorithm, and to assess the reliability of the software that had been tested using this algorithm. In practice, some failures may represent inconsequential deviations from the specification, while others are more severe, and some

may even be catastrophic. Therefore, in evaluating the risk associated with a program, one must distinguish between different failures in accordance with their importance. To do so, we associate a cost with each failure. Our work employs both concepts i.e. we also assign a cost to a failure and then evaluate testing techniques based on the increase in confidence in the program.

## 4.4 Statistical Testing vs Systematic Testing

Testing is an essential activity for achieving adequate reliability. It aims to remove maximum defects before delivering a product, but it is an expensive activity. A well-known testing approach for reliability improvement is known as statistical testing/operational testing, where the software is subjected to the same statistical distribution of inputs that is expected in operation. Statistical testing is appealing as it offers a basis for assessing reliability; so we not only have the assurance of having tried to improve the software, but also an estimate of the reliability actually achieved. Statistical methods provide accurate assessment, but may not necessarily be as useful for achieving reliability. On the other hand, there is a great demand for systematic testing methods (i.e., criteria for choosing test cases and possibly stopping rules), giving some guarantee of thoroughness and cost-effectiveness. Systematic testing mostly employs techniques like boundary value analysis or any coverage testing technique. [Frankl et al., 1997] [Pizza and Strigini, 1998] calls these methods, collectively, "*debug*" testing methods. So we will use both terms interchangeably.

As far as the reliability is concerned, removal of faults discovered in statistical testing will have a greater impact than removal of the faults discovered in systematic testing because it focuses on those failures which have more probability of appearing more frequently in actual operation. Statistical testing is easy to understand and easy to perform. If a test is successful, we will be able to assess the reliability achieved. On the other hand, if a test fails, we will be able to improve the reliability significantly by removing the fault revealed. Furthermore, it is applicable to any software system, large or small, regardless of the language and paradigm used in building the system. However, statistical testing is not always helpful in practice, because in reality the whereabouts of failure points are unknown. For statistical testing to deliver on its promise, it is necessary for the testing profile to be truly representative of operational use. Otherwise, an operational test will become a debug test of arbitrary design. We should also be aware of the fact that operational profile of a software system often changes with time, which can create more problems for statistical testing.

Although statistical testing is a surprisingly good competitor for systematic testing, it is seldom better, and scenarios can be constructed (although their frequency of occurrence in practice is unknown) in which systematic testing is much better at failure exposure. Debug testers always have the potential advantage that by adjusting the test profile and sub domain definitions they might improve the behavior of debug methods. That is definitely problematic for statistical testing. Systematic testing is clearly better if the probability of selecting a failure point (i.e., an element of input domain that will cause the program to fail) in systematic testing is greater than the probability that the point will be used in actual operation. Moreover, a failure point in a safety-critical program may have an extremely low probability of being used in actual operation of the system but may produce catastrophic results. Such a failure point will never be selected as a test case in statistical testing. In that case, debug testing is clearly better. It should be remembered, however, that there is no guarantee that the fault will be discovered through debug testing, nor can we use debug testing to assert the absence of such a failure point in the input domain or to assess the level of reliability achieved.

A number of studies have compared statistical testing with systematic testing which include [Frankl and Weyuker, 1993], [Chen and Yu, 1996], [Frankl et al., 1997], [Frankl et al., 1998], [Frankl and Iakounenko, 1998], [Ntafos, 1998] and [Pizza and Strigini, 1998]. The original motivation for these studies was a belief that statistical testing might be a real alternative to debug testing for finding failures. However, no such conclusive result was obtained. Is it better to test by probing for defects as in "*debug*" testing, or to assess reliability directly as in statistical testing, uncovering defects by accident, so to speak? There is no simple answer, as we do not have concrete evidence.

Using systematic testing techniques is necessary, as testing has many goals other than increasing software reliability. Besides being used for other purposes, systematic testing techniques undoubtedly contribute to the improvement of the reliability of the software under test. Software whose reliability must be high could be tested in a number of different ways, and because testing is an expensive and time-consuming activity, we would like to choose among alternatives, not use them all. We will prefer to select a technique that will achieve the goal in an effective and efficient manner. As we have already mentioned in previous chapters that we have very limited knowledge of testing techniques in terms of their effectiveness and efficiency, we need to evaluate different systematic testing techniques for reliability. However, we should be aware of the potential confusion between detecting failures and achieving reliability, a confusion that occurs when testing finds only unimportant failures. Different failures may

make vastly different contributions to the (un)reliability of the program. Thus, testing with a technique that readily detects small faults may result in a less reliable program than would the testing with a technique that less readily detects some large faults. If a systematic technique consistently turns up low-frequency problems, it may be counterproductive to use it. [Frankl et al., 1997], [Frankl et al., 1998] and [Frankl and Iakounenko, 1998] show that the choice between testing methods depends on rather subtle details of the assumed scenarios; and that debug testing methods that appear promising because of their ability to detect many faults may well be vastly inferior to statistical testing, unless they preferentially discover the more important faults.

We are interested in evaluating systematic testing methods for reliability of delivered software. Studies like this one can thus be viewed as a first step towards examining which testing technique to choose in systematic testing that will reduce more risk relative to other techniques in the software.

## 4.5 Description of the Experiment

We used GQM (Goal-Question-Metrics) approach to state the goals of our experiment. The GQM goal for our experiment is:

Analyze **three defect detection techniques (systematic techniques)** for the **purpose of comparison** with respect to their **capability of increasing confidence (reliability)** from the point of view of the **(researcher)** in the context of **experiment** using small C program.

The goal is to compare code reading, functional testing and structural testing techniques. The comparison has its focus on the improvement of reliability that is measured by number of defects detected and individual weights associated with each failure which is discussed in more detail in section 4.5.3. The main question this chapter tries to answer is: Which technique is more suitable and effective in terms of increasing confidence by reducing risk in the software under test? In other words we can state that which testing technique improves the reliability of the software by a greatest factor.

### 4.5.1 Overview of Testing Methods Used

Software testing techniques cover both static and dynamic testing techniques. We selected three software testing methods for this experiment: *code reading*, *functional testing* and *structural testing*. Code reading belongs to the static testing category and come under manual

static testing approach whereas other two belong to dynamic testing and come under black box and white box testing approach respectively, as described in chapter 2. In this study, we applied code reading using stepwise abstraction, functional testing using equivalence class partitioning and boundary value analysis and structural using 100% branch coverage. The reason to select these testing methods is that they are most widely used testing methods in the practice.

Code reading applied using stepwise abstraction requires the source code to be read. Then we write our own specification by identifying prime subroutines (consecutive lines of code), write a specification for the subroutines, as formally as possible, group subroutines and their specifications together, and repeat the process until all source code will be abstracted. After writing specifications we compare the official specification with our own specification to observe inconsistencies (failure observation) between specified and expected program behavior (analog to failures in the other techniques).

Functional testing techniques are used to design test cases based on the functional requirements of the product. The goal of functional testing is to choose a set of inputs according to the specified functions of the program to test the program so that all functions and sub functions are tested at least once. Functional testing using boundary value analysis analyzes the specification to identify equivalence classes in the input data. Then we choose test cases based on that analysis by focusing on equivalence-class boundaries, run the test cases, and compare the actual output with the expected output to detect failures.

Structural testing techniques are used to design test cases based on the internal structure of the component or system; most commonly internal structure is referred to as the structure of the code. Test cases are designed to get the required coverage for the specified coverage item. The goal of structural testing is to choose a set of inputs according to the structure of the program and aim that all parts (statements, branches or paths) are tested at least once. Structural testing using branch coverage analyzes a source-code listing to construct test cases that will lead to 100% branch coverage. After running the tests, we compare the actual output with the expected output to detect failures.

All techniques are applied in a two stage process: failure observation (observable differences between programs and the official specification) and fault isolation (identifying the exact location of the cause of the failure in program code).

### 4.5.2 Programs and Faults

The program used in this experiment is written in C language. The program (student.c) is used store data about students. The program contains approximately 100 lines of code. Choosing a program of this size was an obligation, as industrial or real programs were not readily available. To seed faults in the program, we firstly need a fault classification to decide which type of faults can be seeded in the program. Presently, there is no universal fault classification. We will use two-faceted fault-classification scheme from the [Basili and Selby, 1987]. Facet one (type) captures the absence of needed code or the presence of incorrect code (omission, commission). Facet two (class) partitions software faults into the six classes:

1. Initialization
2. Control.
3. Computation.
4. Interface.
5. Data.
6. Cosmetic.

Thus we can have following combinations of fault type and class.

1. Omission, Initialization
2. Commission, Initialization
3. Omission, Control
4. Commission, Control
5. Omission, Computation
6. Commission, Computation
7. Omission, Interface
8. Commission, Interface
9. Omission, Data

10. Commission, Data
11. Omission, Cosmetic
12. Commission, Cosmetic

On the basis of this classification, we manually seeded a total of 11 faults in the student program. Still, we cannot guarantee that program do not contain any other faults. All faults cause observable failures; no fault covers another. The failure might be a total failure (no output at all), a serious problem (incorrect output), or a minor problem (misspelled word in the output). No faults detected by the compiler are taken into consideration. Table 4.1 classifies the faults in the programs used in experiment as per the above classification. The description of each fault along with fault type and failure description is given in Table 4.2.

**Table 4.1:** Count and percentage of faults as per the adopted classification

| | **Student.c** | **Percentage of total faults** |
|---|---|---|
| Omission | 3 | 27.27272727 |
| Commission | 8 | 72.72727273 |
| Initialization | 2 | 18.18181818 |
| Control | 3 | 27.27272727 |
| Computation | 2 | 18.18181818 |
| Interface | 0 | 0 |
| Data | 2 | 18.18181818 |
| Cosmetic | 2 | 18.18181818 |
| Total | 11 | 100 |

### 4.5.3 Methodology

We are interested in comparing testing criteria according to their ability to detect and reduce the risk. We cannot compare the techniques for reliability based on the number of failures detected alone. Some program failures may be more important than the others, depending on the cost incurred by, or damages inflicted on, the user. For example, any failure that results from misspelling a word in the output might be considered minor, while a failure that results in the outputting of the wrong numerical value could be considered to have considerably more severe consequences. Even here, differences in the magnitude of the numerical discrepancy

might lead to different failure modes. For example, the cost of outputting the wrong numerical value might be fixed or might depend on how far the incorrect value is from the correct one. We aim to evaluate the effectiveness of systematic software testing techniques in terms of increase in reliability. To do so, we assign a weight to each defect. This weight will approximately define the relative consequence or damage that will be caused if that failure will surface during execution of the system (the severity of the failure). Following categories of defect severity were taken into consideration:

1. **Severity 1:** Defects of this severity cause catastrophic consequences for the system. A defect of this level can cause program execution to abort and can result in critical loss of data, critical loss of system availability, critical loss of security, critical loss of safety, etc.

2. **Severity 2:** Defects of this level cause serious consequences for the system. Such types of defects usually cause a program not to perform properly or to produce unreliable results. We also do not find a workaround for this type of defects For example, a function is severely broken and cannot be used.

3. **Severity 3:** Defects of this level cause significant consequences for the system. There is usually a work around for such type of defects till they are fixed. For example, losing data from a serial device during heavy loads.

4. **Severity 4:** Defects of this level cause small or insignificant consequences for the system. Such defects are easily fixed and a workaround is available for them. For example, misleading error messages.

5. **Severity 5:** Defects of this level cause no negative consequences for the system. Such defects normally produce no erroneous outputs. For example, displaying output in a font other than what the customer desired.

Each defect was assigned a weight/cost of 1 to 5. A failure with a weight/cost of 5 will be a catastrophic one (severity one), while a failure with weight/cost of 1 will have no serious consequence (severity five). Based on the severity of the defects and the corresponding weight/cost assigned to them, we assigned following weights to the defects in student program as shown in Table 4.3[1]. As can be observed from the weights assigned, it is not necessary that a fault of same type will have the same weight i.e. the same defect severity. E.g. in

[1]The weights are only an approximation as they depend on our perspective of failure consequence.

**Table 4.2:** List and description of faults and failures

| Fault No. | Line No. | Fault class | Fault Description | Causes Failure |
|---|---|---|---|---|
| F1 | 29 | Omission, Initialization | Variable count is not initialized with 0. | The student array is not accessible. |
| F2 | 32 | Omission, Cosmetic | Printf function missing | Starting message not displayed |
| F3 | 37 | Commission, Cosmetic | The exit is having wrong option i.e.4, it should be 3. | The program cannot be terminated normally. |
| F4 | 44 | Commission, Computation | The variables count is pre-incremented, it should be post-incremented. | The students are not indexed properly |
| F5 | 49 | Commission, Initialization | Variable i should be initialized with 0. | First entry and subject will not be displayed. |
| F6 | 49 | Commission, Control | The symbol $>$ should be $<$ | The loop will not execute. |
| F7 | 57 | Omission, Control | Break statement missing. | Erratic behavior of Switch Case. |
| F8 | 75 | Commission, Control | Symbol $<=$ should be $<$. | Loop will execute count times instead of count-1 times |
| F9 | 76 | Commission, Data | ‘‘Data" is used instead of ‘‘data[i]". | Only first element of array will be de-allocated |
| F10 | 97 | Commission, Data | Variable i used instead of $i+1$. | Wrong Subject ID displayed. |
| F11 | 98 | Commission, Computation | Variable i should not be incremented, as it is already incremented in the loop. | Improper indexing. |

table 4.2 fault number F6 and F8 are of same type i.e. commission, control, still they have the different weights assigned i.e. 4 and 1 respectively as shown in table 4.3. Figure 4.1 shows the percentage of faults of each severity in the student program, the number in the bold in the parenthesis represents the actual number of faults of that severity in the program. We seeded faults in such a way so that they will not be biased towards the defects of the particular weights. After assigning the weights to the failures, we applied three systematic defect detection techniques to the program separately. After that we analyzed which faults are detected and isolated by each technique. Isolation of the fault is necessary because risk is reduced only when the root cause of the failure is eradicated. We will then calculate the sum of weights of all failures detected and isolated for each technique. A technique with the highest weighted sum will be the one which have increased more confidence in the program as compared to the others. The total weight for each technique will be calculated as:

`Weighted Sum for technique X = 5*(Number of Severity 1 defects) + 4*(Number of Severity 2 defects) + 3*(Number of Severity 3 defects) + 2*(Number of Severity 4 defects) + Number of Severity 5 defects` - **(1)**

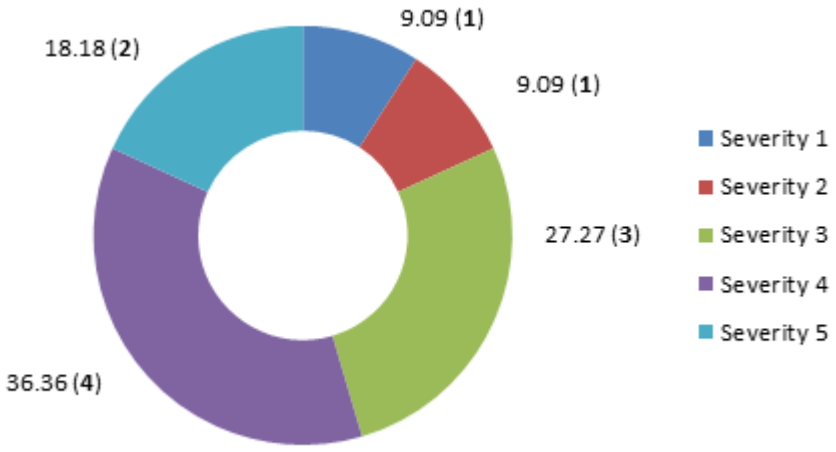



**Figure 4.1:** Percentage of faults of each severity along with the actual number of defects

### 4.5.4 Counting Scheme for the Failures

There have been various efforts to determine a precise counting scheme for "defects" in software [Basili and Selby, 1987]. A failure is "revealed" if program output reveals behavior that deviates from the specification, and "observed" if the subject records the deviate behavior.

**Table 4.3:** List of failures and their corresponding weights.

| Fault No. | Fault Description | Causes Failure | Weight/Cost Assigned |
|---|---|---|---|
| F1 | The variable count is not initialized with 0. | The student array is not accessible. | 5 |
| F2 | Printf function missing | Starting message not displayed | 2 |
| F3 | The exit is having wrong option i.e.4, it should be 3. | The program cannot be terminated normally. | 2 |
| F4 | The variables count is pre-incremented, it should be post-incremented. | The students are not indexed properly | 3 |
| F5 | Variable i should be initialized with 0. | First entry and subject will not be displayed. | 3 |
| F6 | The symbol $>$ should be $<$. | The loop will not execute. | 4 |
| F7 | Break statement missing. | Erratic behavior of Switch Case. | 2 |
| F8 | Symbol $<=$ should be $<$. | Loop will execute count times instead of count-1 times | 1 |
| F9 | ‘‘Data" is used instead of ‘‘data[i]". | Only first element of array de allocated | 1 |
| F10 | Variable i used instead of $i+1$. | Wrong Subject ID displayed. | 2 |
| F11 | Variable i should not be incremented again as it is already incremented in the loop. | Improper indexing. | 3 |

In code reading, an inconsistency (analog to a failure in the other treatments) is "revealed" if the subject captures the deviated behavior in his/her own specification, and "observed" if the subject records the inconsistency between the specifications. For all techniques, a fault is "isolated" if the subject describes the problem in the source code with sufficient precision. We distinguish between faults isolated by using the technique (i.e., after a failure was revealed and observed) and faults isolated by chance (i.e., no failure was revealed or observed), as summarized in Table 4.4. During experiment, we can isolate fault by chance without really observing/revealing the corresponding failure. Even we can have poor cases of fault isolation. In our experiment, we only counted those faults detected by a technique in consideration where a failure is revealed, observed and then isolated (case 8).

**Table 4.4:** Explanations for failure and fault data

| Case | Failure revealed? | Failure observed? | Fault isolated? | Explanation |
|---|---|---|---|---|
| 1 | N | N | N | Defect-detection technique failed |
| 2 | N | N | Y | Fault was isolated by chance |
| 3 | N | Y | N | Meaningless |
| 4 | N | Y | Y | Code reading: constructed poor abstractions Functional/structural test: meaningless |
| 5 | Y | N | N | Poor evaluation of abstractions/output |
| 6 | Y | N | Y | Fault was isolated by chance |
| 7 | Y | Y | N | Poor fault isolation |
| 8 | Y | Y | Y | Defect-detection technique succeeded |

## 4.6 The Experiment

The experiment was run at the Department of Computer Science, University of Kashmir during the fall session of 2011. The experiment was conducted on a single day, as the program was very simple. Each technique was applied on the program in three parallel sessions at different locations by different subjects who were given a preliminary training of testing techniques. No subject had any previous practical experience of software testing; however, they had standard knowledge of computer science field. Subject and technique combination was a purely random. They were given a total of 3 hours (180 minutes) to complete the experiment which include both defect detection and defect isolation task. They were not aware of the fact that techniques are being evaluated for reliability. Their focus was only on detecting and isolating defects, similar to what we did in the last chapter where we evaluated testing techniques for defect detection ability. The experiment results (raw data) were collected on forms specially designed for the experiment. After the conclusion of the experiment, the results were discussed with the subjects to avoid any sort of confusion during the analysis of the results. The results of the experiment and their analysis are as under:

### 4.6.1 Number of Faults Detected and Isolated

As mentioned earlier, only those defects were taken into account, which were revealed, observed and isolated by the subjects. Table 4.5 shows the number of defects detected and isolated and the time taken by each technique. On average 63.63% defects were detected and isolated. As shown in table 4.5 functional testing detected highest number of defects followed by structural which was followed by code reading. Regarding time, the techniques on average took 86% of the total time to detect and isolate the faults in the program. Code reading was most efficient followed by structural testing which was followed by functional testing. [Kamsties and Lott, 1995] also observed the same thing when they compared three defect detection techniques. We also observed the same thing in previous chapter in which we compared these techniques for effectiveness and efficiency. Figure 4.2 shows the effectiveness and efficiency statistics of each testing technique.

### 4.6.2 Types of Faults Detected and Isolated

After counting the number of defects that were isolated for each technique, it was necessary to find which defects were detected and isolated by which technique. Table 4.6 shows fault number and the technique which detected and isolated it. As can been in table 4.6 all faults

**Table 4.5:** Number of defects detected and isolated and time taken by each technique

| Technique | Number of Defects detected and isolated | Time taken by each technique (in minutes) |
|---|---|---|
| Code Reading | 6 | 141 |
| Functional Testing | 8 | 169 |
| Structural Testing | 7 | 154 |

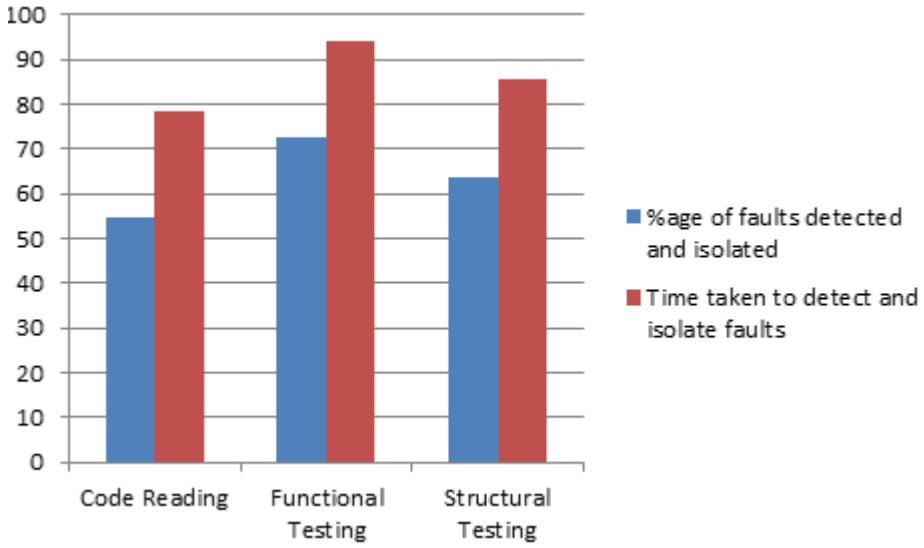



**Figure 4.2:** Effectiveness and efficiency of testing techniques.

except F2 were found. Table 4.7 shows the defects found by each technique categorized as per the severity of faults. The number in the parenthesis in each column under testing technique represents the actual number of faults of that severity in the program. As can be observed from the table 4.7 all techniques were able to find severity one and severity two defect. However, no technique was able to find all defects of other levels of severity with an exception of code reading for severity 5 defects. Figure 4.3 shows the percentage of defects found by each technique for all severity levels.

### 4.6.3 Total Weight Calculated for Each Technique

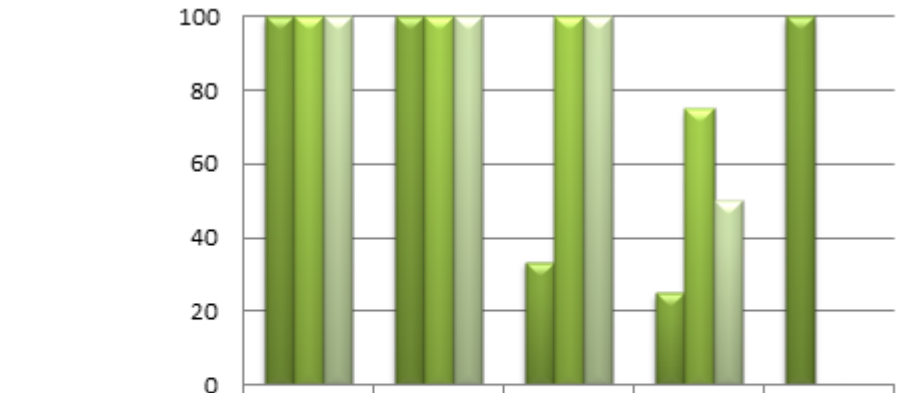
After that, we can calculate the weighted sum for each technique by multiplying each defect found (see table 4.6) with the weight assigned to it (see table 4.3) and then adding them all. The weighted sum for technique computed using equation 1 is as under:

**Table 4.6:** Defects detected and isolated by each technique

| | Testing Technique | | |
|---|---|---|---|
| **Defects detected and isolated by fault number** | **Code Reading** | **Functional Testing** | **Structural Testing** |
| F1 | Y | Y | Y |
| F2 | N | N | N |
| F3 | N | Y | Y |
| F4 | N | Y | Y |
| F5 | N | Y | Y |
| F6 | Y | Y | Y |
| F7 | Y | Y | N |
| F8 | Y | N | N |
| F9 | Y | N | N |
| F10 | N | Y | Y |
| F11 | Y | Y | Y |

**Table 4.7:** Defects detected and isolated by each technique categorized by severity

| | Testing Technique | | |
|---|---|---|---|
| **Severity Level** | **Code Reading** | **Functional Testing** | **Structural Testing** |
| 1 | 1 (1) | 1 (1) | 1 (1) |
| 2 | 1 (1) | 1 (1) | 1 (1) |
| 3 | 1 (3) | 3 (3) | 3 (3) |
| 4 | 1 (4) | 3 (4) | 2 (4) |
| 5 | 2 (2) | 0 (2) | 0 (2) |




| | Severity 1 | Severity 2 | Severity 3 | Severity 4 | Severity 5 |
|---|---|---|---|---|---|
| Code Reading | 100 | 100 | 33.3333 | 25 | 100 |
| Functional Testing | 100 | 100 | 100 | 75 | 0 |
| Structural Testing | 100 | 100 | 100 | 50 | 0 |

**Figure 4.3:** Percentage of Defects detected and isolated by each technique categorized by severity

```
Weighted Sum for Code Reading = 5*(1) + 4*(1) + 3*(1) + 2*(1) + 2
= 5+4+3+2+2
= 16
Weighted Sum for Functional Testing = 5*(1) + 4*(1) + 3*(3) + 2*(3) + 0
=5+4+9+6+0
=24
Weighted Sum for Structural Testing = 5*(1) + 4*(1) + 3*(3) + 2*(2) + 0
= 5+4+9+4+0
= 22
```

So as per the weight sum calculated for each technique, we can say that the functional testing reduced more risk as compared to the other two testing methods. As can been observed from Figure 4.4, functional testing reduced the maximum risk in the program (85.71%), which was followed by structural testing reduced the risk by 78.57% and code reading which only reduced risk by 57.14%. The red bar in the figure 4 represents the residual risk left in the program after testing it with the corresponding testing technique. So the residual risk left by code reading is 42.86%, functional testing is 14.29% and by structural is 21.43%.

### 4.6.4 Combining Testing Techniques

One thing that was observed in the survey of empirical studies in Section 2.4.1 of chapter 2 is that the techniques do appear to be finding different types of faults. So an obvious extension

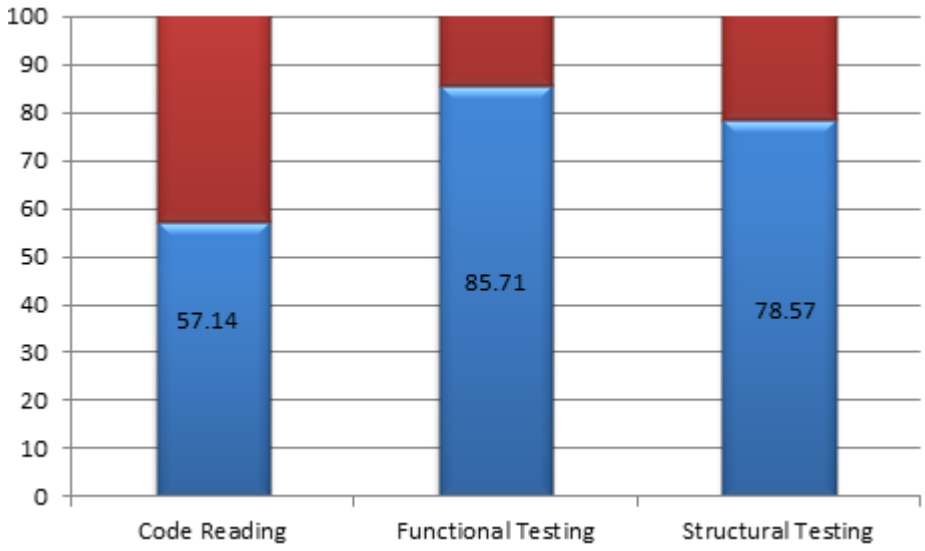



**Figure 4.4:** Percentage of reduced risk and residual risk in the program

of this work is to explore the effectiveness using combinations of techniques. All possible combinations were taken into account. They are:

1. Code Reading and Functional Testing
2. Code Reading and Structural Testing
3. Functional Testing and Structural Testing
4. Code Reading, Functional Testing and Structural Testing

Table 4.8[1] shows faults that were found by combination of techniques. This table represents the union of faults found by combining respective techniques. Table 4.9[2] shows the defects found by each technique categorized as per the severity of faults. The number in the parenthesis in each column under testing technique represents the actual number of faults of that severity in the program.

#### 4.6.4.1 Total Weight calculated for combination of technique

We calculated the weighted sum for every combination of testing techniques by multiplying each defect found (see table 4.8) with the weight assigned to it (see table 4.3) and then adding them all. The weighted sum for combination of techniques computed using 1 is as under:

[1]CR stands for Code Reading, FT stands for Functional Testing and ST stands for Structural Testing
[2]CR stands for Code Reading, FT stands for Functional Testing and ST stands for Structural Testing

```
Weighted Sum for Code Reading +Functional Testing = 5*(1) + 4*(1) + 3*(3)
+ 3*(2) + 2
= 5+4+9+6+2 = 26
Weighted Sum for Code Reading +Structural Testing = 5*(1) + 4*(1) + 3*(3)
+ 3*(2) + 2
= 5+4+9+6+2 = 26
Weighted Sum for Functional Testing + Structural Testing = 5*(1) + 4*(1) + 3*(3)
+ 3*(2) + 0
= 5+4+9+6+0 = 24
Weighted Sum for Code Reading + Functional Testing + Structural Testing = 5*(1)
+ 4*(1) + 3*(3) + 3*(2) + 2
= 5+4+9+6+2 =26
```

**Table 4.8:** Defects detected and isolated by combination of testing techniques

| | **Testing Technique** | | | |
|---|---|---|---|---|
| **Defects detected and isolated by fault number** | **CR + FT** | **CR + ST** | **FT + ST** | **CR + FT + ST** |
| F1 | Y | Y | Y | Y |
| F2 | N | N | N | N |
| F3 | Y | Y | Y | Y |
| F4 | Y | Y | Y | Y |
| F5 | Y | Y | Y | Y |
| F6 | Y | Y | Y | Y |
| F7 | Y | Y | Y | Y |
| F8 | Y | Y | N | Y |
| F9 | Y | Y | N | Y |
| F10 | Y | Y | Y | Y |
| F11 | Y | Y | Y | Y |

By combining testing techniques, we observed that the combination of code reading and functional testing, code reading and structural testing and code reading, functional and structural testing were equally effective in reducing risk in the software (92.85%), whereas combination of functional and structural testing reduced the risk by 85.71% only as shown

**Table 4.9:** Defects detected and isolated by each technique categorized by severity

| | Testing Technique | | | |
|---|---|---|---|---|
| **Severity Level** | **CR + FT** | **CR + ST** | **FT + ST** | **CR + FT + ST** |
| 1 | 1 (1) | 1 (1) | 1 (1) | 1 (1) |
| 2 | 1 (1) | 1 (1) | 1 (1) | 1 (1) |
| 3 | 3 (3) | 3 (3) | 3 (3) | 3 (3) |
| 4 | 3 (4) | 3 (4) | 3 (4) | 3 (4) |
| 5 | 2 (2) | 2 (2) | 0 (2) | 2 (2) |

in Figure 4.5. As expected, by combining the two testing techniques, we got better results; however, no combination of techniques was able to find 100% defects.

## 4.7 Threats to Validity

Validity threats like learning have been eliminated, as the experiment was conducted in three parallel sessions. The selection threat is also eliminated, as all the subjects were equally experienced and had similar academic background. The difference with respect to individual ability was minimized by random match of subject and testing technique. In spite of this following possible threats to validity (mostly external) were identified in our experiment:

1. The experiment was conducted with post graduate students of computer science who had no practical knowledge of testing. Our results depend on the way our subject characterized the faults in the program.

2. The size of the C-code module is small in comparison with the size of systems in industry. However, it is comparable to the size of programs used in other experiment which aim at evaluating software testing techniques mentioned in section 2.4 of chapter 2. In addition to that our results are based on investigation of a single program; other faulty programs may have different characteristics.

## 4.8 Discussion

One thing that is worth mentioning is that no testing technique was able to reduce the known risk of the program to the zero level, even for this very trivial program. This thing proves

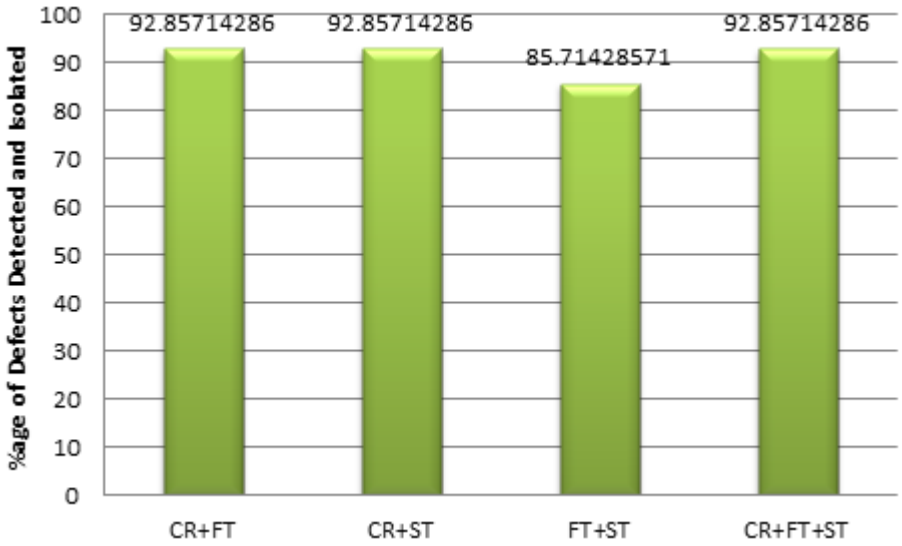



**Figure 4.5:** Effectiveness of combination of testing techniques

the claim of many researchers that there is no silver bullet technique for software testing. We require different types of appropriate techniques for detecting different types of faults in the program. We believe that the achieved level of confidence in our study will be much lower for a program of significant size and complexity. Another interesting thing is that no technique was able to find the fault F2 in the program. The possible reason for this is the type of fault (Omission, Cosmetic) and its location in the source code. The subjects may have confused themselves with the next printf () statement and treated that as a welcome message. In addition to that, functional and structural were unable to detect fault number F8 and F9. Although they are assigned a weight of 1 in our case, such types of fault may cause severe problems in other cases. It also points to the significance of using code reading in the software testing. One thing that can be observed from the Figure 4.5 is that if we would have skipped structural testing in our case, it would have caused no effect on the result, as combination of code reading and functional give us the same result, and combination of functional and structural increases confidence by 85% which is 7% less what we achieved by the combination of code reading and functional testing, and that 7% deficit in the achieved confidence level is because of structural testing as can been observed from Table 4.8 and Table 4.9. However, we strongly believe that this thing is the outcome of the types of faults and nature of program, we have used in our study.

## 4.9 Conclusion and Future Work

We conducted a simple but novel experiment to evaluate three software testing techniques, code reading, functional testing and structured testing for reliability. We conclude that every testing technique contributes to reduction of risk in the software. However, we found functional testing to be more effective in terms of the reducing the risk in the program; whereas, it was the least efficient in terms of time. However, at this point of time, the results cannot be generalized due to the external validity threats. One thing that is clear is that combination of different techniques is more effective solution to increase reliability in the software; however that can decrease the overall efficiency. We also observed that no technique or combination of techniques was able to reduce known risk in the software to the zero level. Results also indicated that no technique is perfect and has its weaknesses.

Although the experiment was not run on the large scale, we strongly believe that it is a significant step towards evaluating systematic software testing techniques for reliability. Studies of this kind demand replication, as the results can be generalized only when we have substantial empirical evidence. The future work includes carrying out similar experiment on a large scale with different subjects and real programs with realistic faults of significant size. In addition to that, other testing techniques need to be evaluated for the reliability. We also need to examine the interaction between testing techniques and the program, subjects, faults and other allied parameters.

# Chapter 5

# Conclusions & Future Work

## 5.1 Conclusions Drawn

The knowledge of the testing techniques that contribute to quality of the software is important to the advancement of the software testing research. This thesis presents three studies that evaluate software testing techniques.

Chapter 2 is an exploratory work which identifies current issues in software testing techniques evaluation. Thereafter, we propose a framework (set of guidelines) which aim at mitigating the current issues in software testing techniques experimentation. In addition to this, a set of factors are also presented which can help us in choosing appropriate testing technique in the absence of concrete knowledge about the testing techniques statistics. The conclusions drawn from the chapter are:

1. Present situation calls for further work on evaluation of software testing techniques so as to acquire the basic knowledge about the relative effectiveness and efficiency of software testing techniques for both fault finding and reliability criterion. We need to carry out many other experiments in order to get reliable and generalizable results. Replication of earlier work will be more useful.

2. We need to establish and follow common and standard framework for testing techniques evaluation so that there are little variations in experimentation goals and results.

Chapter 3 presents an empirical study of testing techniques using a controlled experiment which evaluates and compares three defect detection techniques for their effectiveness. The experiment compared one static testing technique (code reading) and two dynamic testing techniques (functional and structural testing). We conclude that they are all equally effective in terms of failure detection and fault isolation. However, a difference was observed in case of efficiency. While the effect of program was significant in most of the cases, the effect of testing techniques was observed only in two cases. Moreover, the order and subjects had no significant effect on the effectiveness or efficiency. In case of fault types, all testing techniques were equally effective in terms of detection and isolation for all types of fault classes and types except for cosmetic faults.

Chapter 4 presents an empirical study of testing techniques using a controlled experiment which evaluates and compares three defect detection techniques (code reading, functional testing and structural testing) for reliability using a novel method. A major conclusion of the study was that every testing technique contributes to reduction of risk in the software. However, we found functional testing most effective in terms of the reducing the risk in the

program relative to other techniques; whereas, it was the least efficient in terms of time. However, at this point of time, we cannot generalize the results due to the validity threats. Results also indicated that no technique is perfect and has its weaknesses. One thing that is clear is that combination of different techniques is more effective solution to increase reliability in the software; however that can decrease the overall efficiency.

## 5.2 Future Work

There are some limitations in the present research that stem from the particular research focus and scope that have been chosen. With so many testing techniques and the very inadequate quantitative and qualitative knowledge about them, we strongly believe that there is a need of much more research and evidence in the software testing techniques evaluation field. The work in this thesis can be extended or replicated further to produce more realistic, generalized and implementable results.

The work in the chapter 2 can be extended in the following ways:

1. Developing a uniform working definition for effectiveness which can be used as a standard comparison criteria for comparing software testing techniques.

2. Validating the proposed framework in different scenarios, so as to check whether the proposed framework is viable for evaluation of testing techniques.

3. Refining the proposed framework, if required.

We believe that few experiments are not sufficient to understand the relative strengths and weakness of testing techniques. Further replications of the experiments will help in making strong conclusions regarding the effectiveness and efficiency of software testing techniques and other allied factors. In particular, important studies would be empirical evaluations in an industrial context which can facilitate the transfer of these techniques to practice. However, carrying out such experiments in accordance with the common and standard is necessary for the reasons mentioned in chapter 2 of this thesis.

The work presented in chapter 3 can be extended by evaluating different testing techniques. In addition, a important extension will be the use of programs which are of significant size and contain realistic faults. We also need to understand the effects of subject, program type and fault type on the evaluation results in more detail by varying them under controlled manner.

The work in chapter 4 can be carried out on a large scale with different subjects and programs with realistic faults of significant size. In addition to that, other testing techniques need to be evaluated for the reliability. We also need to examine the interaction between testing techniques and the program, subjects, faults and other allied parameters.

# Publications

# PUBLICATIONS

## Refereed Journal Papers

2012 **A Novel Approach for Evaluating Software Testing Techniques for Reliability** (Sheikh Umar Farooq, S.M.K. Quadri), *ARPN Journal of Systems and Software*, Volume 2, Number 3, Pages 84-96, eISSN: 2222-9833, March 2012.

2011 **Evaluating Effectiveness of Software Testing Techniques With Emphasis on Enhancing Software Reliability** (Sheikh Umar Farooq, S.M.K. Quadri), *Journal of Emerging Trends in Computing and Information Science*, Volume 2, Issue 12, Pages 740-745, eISSN: 2079-8407, December 2011.

2011 **Quality Practices in Open Source Software Development Affecting Quality** (Sheikh Umar Farooq, S.M.K. Quadri), *Trends in Information Management*, Volume 7, Issue 2, Pages 108-126, pISSN: 0973-4163, December 2011.

2011 **3Ws of Static Software Testing Techniques** (Sheikh Umar Farooq, S.M.K. Quadri), *Global Journal of Computer Science & Technology*, Volume 11, Issue 6, Pages 77-86, pISSN: 0975-4350, eISSN: 0975-4172, April 2011.

2011 **Software Reliability Growth Modeling with New Generalized Exponential Growth Models and Optimal Release Policy** (S. M. K Quadri, Nesar Ahmad, Sheikh Umar Farooq), *Global Journal of Computer Science & Technology*, Volume 2, Issue 6, Pages 26-41, pISSN: 0975-4350, eISSN: 0975-4172, February 2011.

2011 **Software Measurements and Metrics: Role in Effective Software Testing** (Sheikh Umar Farooq, S.M.K. Quadri, Nesar Ahmad), *International Journal of Engineering Science & Technology*, Volume 3, Issue 1, Pages 671-680, eISSN: 0975-5462, February 2011.

2010 **Identifying Some Problems with Selection of Software Testing Techniques** (Sheikh Umar Farooq, S.M.K. Quadri), *Oriental Journal of Computer Science & Technology*, Volume 3, Issue 2, Pages 265-268, pISSN: 0974-6471, December 2010.

2010 **Software Testing - Goals, Principles and Limitations** (S.M.K. Quadri, Sheikh Umar Farooq), *International Journal of Computer Applications*, Volume 6, Number 9, Pages 07-10, eISSN: 0975 8887, September 2010.

2010 **Effectiveness of software testing techniques on a measurement scale** (Sheikh Umar Farooq, S.M.K. Quadri), *Oriental Journal of Computer Science & Technology*, Volume 3, Issue 1, Pages 109-113, pISSN: 0974-6471, June 2010.

## Refereed Conference Papers

2012 **Metrics, Models and Measurements in Software Reliability** (Sheikh Umar Farooq, S.M.K. Quadri, Nesar Ahmad), *In Proceedings of 10th IEEE Jubilee International Symposium on Applied Machine Intelligence and Informatics*, Pages 441-449, pISBN: 978-1-4577-0197-9, January 26-28, Herlany, Slovakia, 2012.

2011 **A Roadmap for Effective Software Testing** (Sheikh Umar Farooq, S.M.K. Quadri, ), *In Proceedings of 5th National Conference on Computing for Nation Development*, Bharati Vidyapeeth's Institute of Computer Applications and Management, Pages 263-264, pISSN:0973-7529, pISBN:978-93-80544-00-7, March 10-11, New Delhi, India, 2011.

2011 **Notable Metrics in Software Testing** (S.M.K. Quadri, Sheikh Umar Farooq), *In Proceedings of 5th National Conference on Computing for Nation Development*, Bharati Vidyapeeth's Institute of Computer Applications and Management, Pages 273-276, pISSN:0973-7529, pISBN:978-93-80544-00-7, March 10-11, New Delhi, India, 2011.

2011 **Testing Techniques Selection: A Systematic Approach** (S.M.K. Quadri, Sheikh Umar Farooq), *In Proceedings of 5th National Conference on Computing for Nation Development*, Bharati Vidyapeeth's Institute of Computer Applications and Management, Pages 279-282, pISSN:0973-7529, pISBN:978-93-80544-00-7, March 10-11, New Delhi, India, 2011.

Presentations & Talks

2011 **Quality Practices in Open Source Software Development Affecting Quality**, *Presented at National Seminar on Open Source Softwares: Challenges & Opportunities*, June 20-22, University of Kashmir, India, 2011.

2010 **What Should be Measured During Software Testing?**, *Presented at 6th JK Science Congress*, December, University of Kashmir, India, 2010.

2010 **Towards More Successful Software Testing**, *Presented at 6th JK Science Congress*, December, University of Kashmir, India, 2010.

2010 **Decisive Factors for Selecting Software Testing Techniques**, *Presented at 6th JK Science Congress*, December, University of Kashmir, India, 2010.

# References


Aurum, A., Petersson, H., and Wohlin, C. State-of-the-art: software inspections after 25 years. *Software Testing, Verification and Reliability*, 12(3):133–154, 2002. 20

Basili, V. and Selby, R. Comparing the effectiveness of software testing strategies. *Software Engineering, IEEE Transactions on*, (12):1278–1296, 1987. 20, 29, 42, 48, 49, 62, 64, 81, 92, 96

Basili, V., Selby Jr, R., and Hutchens, D. Experimentation in software engineering. Technical report, DTIC Document, 1985. 7, 20, 41, 45

Beck, K. *Test-driven development: by example*. Addison-Wesley Professional, 2003. 20

Beizer, B. *Black-box testing: techniques for functional testing of software and systems*. John Wiley & Sons, Inc., 1995. 15

Beizer, B. *Software testing techniques (2nd ed.)*. Van Nostrand Reinhold Co., New York, NY, USA, 1990. ISBN 0-442-20672-0. 4, 13, 84

Bentley, J. Software testing fundamentals–concepts, roles, and terminology. In *SUGI*, volume 30, 2005. 2

Bertolino, A. An overview of automated software testing. *Journal of Systems and Software*, 15(2):133–138, 1991. 16

Bertolino, A. The (im) maturity level of software testing. *ACM SIGSOFT Software Engineering Notes*, 29 (5):1–4, 2004. 6, 35

Bertolino, A. Software testing research: Achievements, challenges, dreams. In *Future of Software Engineering, 2007. FOSE'07*, pages 85–103. IEEE, 2007. 2, 4, 5, 11

Bible, J., Rothermel, G., and Rosenblum, D. A comparative study of coarse-and fine-grained safe regression test-selection techniques. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 10 (2):149–183, 2001. 20

Bieman, J. and Schultz, J. An empirical evaluation (and specification) of the all-du-paths testing criterion. *Software Engineering Journal*, 7(1):43–51, 1992. 20

Box, G., Hunter, J., and Hunter, W. *Statistics for experimenters: design, innovation, and discovery*, volume 2. Wiley Online Library, 2005. 56, 67

Briand, L. and Labiche, Y. Empirical studies of software testing techniques: Challenges, practical strategies, and future research. *ACM SIGSOFT Software Engineering Notes*, 29(5):1–3, 2004. 26

Chen, T. and Yu, Y. On the expected number of failures detected by subdomain testing and random testing. *Software Engineering, IEEE Transactions on*, 22(2):109–119, 1996. 89

Chu, H. An evaluation scheme of software testing techniques. *Technical Report Series*, 1997. 11, 16

Daly, J., Brooks, A., Miller, J., Roper, M., and Wood, M. Verification of results in software maintenance through external replication. In *Software Maintenance, 1994. Proceedings., International Conference on*, pages 50–57. IEEE, 1994. 42

Dijkstra, E. On the reliability of mechanisms. *Notes on Structured Programming*, 1970. 2

Do, H., Elbaum, S., and Rothermel, G. Infrastructure support for controlled experimentation with software testing and regression testing techniques. In *Empirical Software Engineering, 2004. ISESE'04. Proceedings. 2004 International Symposium on*, pages 60–70. IEEE, 2004. 20

Do, H., Elbaum, S., and Rothermel, G. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, 10(4):405–435, 2005. 7, 20

Elbaum, S., Malishevsky, A., and Rothermel, G. Prioritizing test cases for regression testing. *ACM SIGSOFT Software Engineering Notes*, 25(5), 2000. 20

Eldh, S., Hansson, H., Punnekkat, S., Pettersson, A., and Sundmark, D. A framework for comparing efficiency, effectiveness and applicability of software testing techniques. In *Testing: Academic and Industrial Conference-Practice And Research Techniques, 2006. TAIC PART 2006. Proceedings*, pages 159–170. IEEE, 2006. 20

Eldh, S. *On Test Design*. PhD thesis, Mälardalen University Press, October 2011. URL http://www.mrtc.mdh.se/index.php?choice=publications&id=2635. 13

Farooq, A. and Dumke, R. *Evaluation approaches in software testing*. Univ.-Bibliothek, Hochschulschr.-und Tauschstelle, 2008. 19

Farooq, S. and Quadri, S. Identifying some problems with selection of software testing techniques. *Oriental Journal of Computer Science & Technology*, 3(2):266–269, 2010. 19, 29

Fenton, N. Software measurement: A necessary scientific basis. *Software Engineering, IEEE Transactions on*, 20(3):199–206, 1994. 7

Fenton, N., Pfleeger, S., and Glass, R. Science and substance: a challenge to software engineers. *Software, IEEE*, 11(4):86–95, 1994. 7

Frankl, P., Hamlet, D., Littlewood, B., and Strigini, L. Choosing a testing method to deliver reliability. In *Proceedings of the 19th international conference on Software engineering*, pages 68–78. ACM, 1997. 20, 87, 88, 89, 90

Frankl, P. and Iakounenko, O. Further empirical studies of test effectiveness. In *ACM SIGSOFT Software Engineering Notes*, volume 23, pages 153–162. ACM, 1998. 20, 89, 90

Frankl, P. and Weiss, S. An experimental comparison of the effectiveness of branch testing and data flow testing. *Software Engineering, IEEE Transactions on*, 19(8):774–787, 1993. 20

Frankl, P. and Weyuker, E. A formal analysis of the fault-detecting ability of testing methods. *Software Engineering, IEEE Transactions on*, 19(3):202–213, 1993. 89

Frankl, P. and Weyuker, E. Testing software to detect and reduce risk. *Journal of Systems and Software*, 53(3):275–286, 2000. 86

Frankl, P., Hamlet, R., Littlewood, B., and Strigini, L. Evaluating testing methods by delivered reliability [software]. *Software Engineering, IEEE Transactions on*, 24(8):586–601, 1998. 85, 87, 89, 90

Gibbs, W. Software's chronic crisis. *Scientific American*, 271(3):72–81, 1994. 7

Goodenough, J. and Gerhart, S. Toward a theory of test data selection. *ACM SIGPLAN Notices*, 10(6): 493–510, 1975. 11

Graham, D. and Van Veenendaal, E. *Foundations of Software Testing: ISTQB Certification*. Cengage Learning Emea, 2008. 14, 32

Graves, T., Harrold, M., Kim, J., Porter, A., and Rothermel, G. An empirical study of regression test selection techniques. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 10(2): 184–208, 2001. 20

Gutjahr, W. Optimal test distributions for software failure cost estimation. *Software Engineering, IEEE Transactions on*, 21(3):219–228, 1995. 87

Harrold, M. Testing: a roadmap. In *Proceedings of the Conference on the Future of Software Engineering*, pages 61–72. ACM, 2000. 4

Hetzel, W. An experimental analysis of program verification methods. 1976. 20, 21, 42, 47, 49

Host, M., Wohlin, C., and Thelin, T. Experimental context classification: incentives and experience of subjects. In *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, pages 470–478. IEEE, 2005. 20

Howden, W. Theoretical and empirical studies of program testing. *Software Engineering, IEEE Transactions on*, (4):293–298, 1978. 4

Hutchins, M., Foster, H., Goradia, T., and Ostrand, T. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of the 16th international conference on Software engineering*, pages 191–200. IEEE Computer Society Press, 1994. 20

Judd, C., Smith, E., and Kidder, L. Research methods in social relations. 1991. 46

Juristo, N. and Moreno, A. *Basics of software engineering experimentation*. Springer, 2001. 30

Juristo, N. and Vegas, S. Functional testing, structural testing and code reading: what fault type do they each detect? *Empirical Methods and Studies in Software Engineering*, pages 208–232, 2003. 20, 42, 49, 58

Juristo, N., Moreno, A., and Vegas, S. A survey on testing technique empirical studies: How limited is our knowledge. In *Empirical Software Engineering, 2002. Proceedings. 2002 International Symposium n*, pages 161–172. IEEE, 2002. 26

Juristo, N., Moreno, A., and Vegas, S. Limitations of empirical testing technique knowledge. *SERIES ON SOFTWARE ENGINEERING AND KNOWLEDGE ENGINEERING*, 12:1–38, 2003. 29, 35

Juristo, N., Moreno, A., and Vegas, S. Reviewing 25 years of testing technique experiments. *Empirical Software Engineering*, 9(1):7–44, 2004. 20, 21, 23, 26, 32, 49

Kamsties, E. and Lott, C. An empirical evaluation of three defect-detection techniques. *Software EngineeringESEC'95*, pages 362–383, 1995. 4, 20, 42, 49, 58, 66, 99

Kim, J., Porter, A., and Rothermel, G. An empirical study of regression test application frequency. In *Software Engineering, 2000. Proceedings of the 2000 International Conference on*, pages 126–135. IEEE, 2000. 20

Kitchenham, B., Pfleeger, S., Pickard, L., Jones, P., Hoaglin, D., El Emam, K., and Rosenberg, J. Preliminary guidelines for empirical research in software engineering. *Software Engineering, IEEE Transactions on*, 28(8):721–734, 2002. 7

Kurokawa, T. and Shinagawa, M. Technical trends and challenges of software testing. *http://www. nistep. go. jp/achiev/ftx/eng/stfc/stt029e/qr29pdf/STTqr2902. pdf, Lat visited on*, 1(03):2010, 2008. 17

Li, N. and Malaiya, Y. On input profile selection for software testing. In *Software Reliability Engineering, 1994. Proceedings., 5th International Symposium on*, pages 196–205. IEEE, 1994. 87

Lott, C. and Rombach, H. Repeatable software engineering experiments for comparing defect-detection techniques. *Empirical Software Engineering*, 1(3):241–277, 1996. 43, 56, 68

Luo, L. Software testing techniques. *Institute for software research international Carnegie mellon university Pittsburgh, PA*, 15232:1–19, 2001. 19

Lyu, M. et al. Handbook of software reliability engineering. 1996. 84

Malik, Q. et al. *Combining model-based testing and stepwise formal development*. PhD thesis, 2010. 2

Moreno, A., Shull, F., Juristo, N., and Vegas, S. A look at 25 years of data. *IEEE Software*, 26(1): 15–17, 2009. 23, 24, 32

Myers, G. A controlled experiment in program testing and code walkthroughs/inspections. *Communications of the ACM*, 21(9):760–768, 1978. 20, 42, 48, 49

Myers, G., Sandler, C., and Badgett, T. *The art of software testing*. Wiley, 2011. 13

Ntafos, S. On random and partition testing. In *ACM SIGSOFT Software Engineering Notes*, volume 23, pages 42–48. ACM, 1998. 89

Ntafos, S. The cost of software failures. In *Proc. IASTED Software Engineering Conference*, pages 53–57, 1997. 87

Offutt, A. and Lee, S. An empirical evaluation of weak mutation. *Software Engineering, IEEE Transactions on*, 20(5):337–344, 1994. 20

Offutt, A., Lee, A., Rothermel, G., Untch, R., and Zapf, C. An experimental determination of sufficient mutant operators. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 5(2): 99–118, 1996. 20

Perry, W. *Effective methods for software testing*. John Wiley & Sons, Inc., 2006. 58

Pizza, M. and Strigini, L. Comparing the effectiveness of testing methods in improving programs: the effect of variations in program quality. In *Software Reliability Engineering, 1998. Proceedings. The Ninth International Symposium on*, pages 144–153. IEEE, 1998. 87, 88, 89

Rapps, S. and Weyuker, E. Selecting software test data using data flow information. *Software Engineering, IEEE Transactions on*, (4):367–375, 1985. 16

Rombach, H., Basili, V., and Selby, R. *Experimental software engineering issues: critical assessment and future directions: international workshop, Dagstuhl Castle, Germany, September 14-18, 1992: proceedings*, volume 706. Springer, 1993. 41

Roper, M. *Software testing*. McGraw-Hill, Inc., 1995. 14

Roper, M., Wood, M., and Miller, J. An empirical evaluation of defect detection techniques. *Information and Software Technology*, 39(11):763–775, 1997. 20, 42, 49, 58, 81

Rothermel, G. and Harrold, M. Empirical studies of a safe regression test selection technique. *Software Engineering, IEEE Transactions on*, 24(6):401–419, 1998. 20

Rothermel, G., Untch, R., Chu, C., and Harrold, M. Test case prioritization: An empirical study. In *Software Maintenance, 1999.(ICSM'99) Proceedings. IEEE International Conference on*, pages 179–188. IEEE, 1999. 20

Selby, R. Combining software testing strategies: An empirical evaluation. In *Proceedings of the ACM/SIGSOFT IEEE Workshop on Software Testing*, pages 82–90, 1986. 48

Sommerville, I. *Software Engineering*. Addison-Wesley, 2007. 14, 15

Taipale, O., Smolander, K., and Kälviäinen, H. Finding and ranking research directions for software testing. *Software Process Improvement*, pages 39–48, 2005. 4

Tawileh, A., McIntosh, S., Work, B., and Ivins, W. The dynamics of software testing. In *Proceedings of the 25th System Dynamics Conference, July*, 2007. 17

Tichy, W. Should computer scientists experiment more? *Computer*, 31(5):32–40, 1998. 7

Tsoukalas, M., Duran, J., and Ntafos, S. On some reliability estimation problems in random and partition testing. *Software Engineering, IEEE Transactions on*, 19(7):687–697, 1993. 87

Vegas, S. What information is relevant when selecting testing techniques. In *Proceedings of the 13th International Conference on Software Engineering and Knowledge Engineering*, pages 45–52, 2001. 30

Vegas, S. Identifying the relevant information for software testing technique selection. In *Empirical Software Engineering, 2004. ISESE'04. Proceedings. 2004 International Symposium on*, pages 39–48. IEEE, 2004. 31, 38

Vegas, S. and Basili, V. A characterisation schema for software testing techniques. *Empirical Software Engineering*, 10(4):437–466, 2005. 20

Vegas, S., Juristo, N., and Basili, V. Packaging experiences for improving testing technique selection. *Journal of Systems and Software*, 79(11):1606–1618, 2006. 32

Vokolos, F. and Frankl, P. Empirical evaluation of the textual differencing regression testing technique. In *Software Maintenance, 1998. Proceedings. International Conference on*, pages 44–53. IEEE, 1998. 20

Vos, T., Marin, B., Panach, I., Baars, A., Ayala, C., and Franch, X. Evaluating software testing techniques and tools. 20, 35

Weyuker, E. The cost of data flow testing: An empirical study. *Software Engineering, IEEE Transactions on*, 16(2):121–128, 1990. 20

Weyuker, E. Can we measure software testing effectiveness? In *Software Metrics Symposium, 1993. Proceedings., First International*, pages 100–107. IEEE, 1993. 36

Weyuker, E. Using failure cost information for testing and reliability assessment. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 5(2):87–98, 1996. 87

Wohlin, C., Runeson, P., Host, M., Ohlsson, C., Regnell, B., and Wesslén, A. *Experimentation in software engineering: an introduction*. Kluver Academic Publishers, 2000. 7, 20

Wong, W. and Mathur, A. Reducing the cost of mutation testing: An empirical study. *Journal of Systems and Software*, 31(3):185–196, 1995a. 4

Wong, W. and Mathur, A. Fault detection effectiveness of mutation and data flow testing. *Software Quality Journal*, 4(1):69–83, 1995b. 20

Wong, W., Horgan, J., London, S., and Agrawal, H. A study of effective regression testing in practice. In *PROCEEDINGS The Eighth International Symposium On Software Reliability Engineering*, pages 264–274. IEEE, 1997. 20

Wood, A. Software reliability growth models: Assumptions vs. reality. In *PROCEEDINGS The Eighth International Symposium On Software Reliability Engineering*, pages 136–141. IEEE, 1997. 84

Yang, X. *Towards a self-evolving software defect detection process*. PhD thesis, University of Saskatchewan, 2007. 31

Young, M. *Software Testing and Analysis: Process, Principles, and Techniques*. John Wiley & Sons, 2008. 6