ABSTRACT

NAGAPPAN, NACHIAPPAN. A Software Testing and Reliability Early Warning (STREW) Metric Suite. (Under the direction of Dr. Laurie A. Williams.)

The demand for quality in software applications has grown, and awareness of software testing-related issues plays an important role towards that. Unfortunately in industrial practice, information on software field quality of a product tends to become available too late in the software lifecycle to affordably guide corrective actions. An important step towards remediation of this problem lies in the ability to provide an early estimation of post-release field quality.

This dissertation presents a suite of nine in-process metrics, the Software Testing and Reliability Early Warning (STREW) metric suite, that leverages the software testing effort to provide (1) an estimate of post-release field quality early in software development phases, and (2) a color-coded, feedback to the developers on the quality of their testing effort to identify areas that could benefit from more testing. We built and validated our model via a three-phase case study approach which progressively involved 22 small-scale academic projects, 27 medium-sized open source projects, and five large-scale industrial projects. The ability of the STREW metric suite to estimate post-release field quality was evaluated using statistical regression models in the three different environments. The error in estimation and the sensitivity of the predictions indicate the STREW metric suite can effectively be used to predict post-release software field quality. Further, the test quality feedback was found to be statistically significant with the post-release software quality, indicating the ability of the STREW metrics to provide meaningful feedback on the quality of the testing effort.

A Software Testing and Reliability Early Warning (STREW) Metric Suite

By

Nachiappan Nagappan

A DISSERTATION SUBMITTED TO THE GRADUATE FACULTY OF
NORTH CAROLINA STATE UNIVERSITY
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

COMPUTER SCIENCE

Raleigh, NC

2005

Approved by:

Dr. Christopher G. Healey	Dr. Jason A. Osborne
Dr. Mladen A. Vouk	Dr. Laurie A. Williams Chair of Advisory Committee

Om

To Appa and Amma….[+]

[+] To my parents

Biography

Nachiappan Nagappan was born on the 24th of May 1980 in a small village, Kottaiyur in Tamil Nadu, India. He did his schooling at Vidya Mandir Senior Secondary School, Madras and graduated in 1997. He received a B.Tech degree in Information Technology from the University of Madras in 2001 and his M.S. degree from North Carolina State University in 2002. In his spare time, he enjoys building toy train models and playing soccer.

Acknowledgements

This thesis would not have been possible without the help of several people. It is impossible to acknowledge everyone I have worked with in the 22 years of my academic life. If I have missed you, I apologize for the mistake but want you to know that you are gratefully remembered.

First and foremost, I would like to thank Laurie for being a fantastic advisor. I met Laurie for the first time on a Wednesday, Nov 14 2001 at 11.00 AM. That was our first meeting. From that time on till today it's been a pleasure to work with her. I appreciate her patience, encouragement and support through these years. She has always watched out for me, helped me with all my problems and has always been open in her views. Her comments/feedback is something I will miss. Thanks, Laurie.

Dr. Vouk has been a truly inspiring mentor. I have always been amazed by his ability to think of several possible solutions to a problem whenever I have got stuck. His experience and insight have greatly influenced my work. I hope this work meets the high standards set by him.

My other committee members Dr. Jason Osborne and Dr. Chris Healey have been supportive of my work through these years. Dr. Osborne took the trouble of explaining statistics to a computer science major in addition to spending long hours explaining various statistical modeling techniques and packages and the significance of interpreting the results. My understanding of "the why and how of statistics" has been greatly influenced by him. Dr. Healey has been supportive of my work and has showed a lot of interest in my future goals and plans after graduation. I sincerely appreciate all his help.

In addition to this I would like to express my sincere thanks to all the people who have helped me professionally in different stages of my careers, in alphabetical order Pekka Abrahamsson, Annie Antón, Thomas Ball, Ed Gehringer, John Hudepohl, Brendan Murphy, Madan Musuvathi, Will Snipes, and David Thuente. The people at 165C, Lucas – thanks for clarifying all the questions I had on the data validity issues, Mark, Will, and Frank for putting up with me for so long and providing welcome breaks. The software developers who worked on the several open source projects that were

used in this dissertation, the developers in the industrial software organization, and the students of CSC 326 in Fall 2003 are gratefully acknowledged for their contribution towards this thesis.

My thanks are also due to Martin for working with me on GERT and my friends at NC State Guru, CK, Pradeep (both), Yogi, Dinesh, Prabhakar and Karthikeyan who have made my stay at NCSU very enjoyable.

On an organizational level, I would like to thank the Department of Computer Science, (especially Margery Page and Vilma Berg) and North Carolina State University, for proving an excellent environment for a student 8801 miles (14164 km) from home.

Finally last but in no way least, Chidambaram Arunachalam, Annan and Akka for keeping me sane, taking care of me and feeding me on a regular basis. Without you people, my stay here would not have been possible, and I would not have graduated…. and my brother for being the motivation for me to finish up before he did….

My work was partially funded by an IBM (IBM-Eclipse Innovation Award). This material is also based upon work supported by the National Science Foundation under CAREER award Grant No. 0346903 and DUE-CCLI 0088178. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Software engineering is the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software[43]. A more mathematical definition of Software Engineering was presented by Boehm in the classic Software Engineering Economics book [10] ,

> *Software engineering is the application of science and mathematics by which the capabilities of computer equipment are made useful to man via computer programs, procedures, and associated documentation. [10]*

Software Engineering activities include: Managing, Costing, Planning, Modeling, Analyzing, Specifying, Designing, Implementing, Testing and Maintaining of software [33].

In industry, estimates of software field quality are often available too late in the software lifecycle to affordably guide corrective actions to the quality of the software. As a result, true field quality cannot be measured before a product has been completed and delivered to an internal or external customer. Because this information is available late in the software lifecycle, corrective actions tend to be expensive [10]. Software developers can benefit from an early warning regarding the quality of their product.

In our research, we formulate this early warning from a collection of internal testing metrics that are correlated with Trouble Reports (TRs) per thousand lines of code (KLOC), an external measure

obtained from users. A TR [60] is a customer-reported problem whereby the software system does not behave as the customer expects. An internal metric, such as cylomatic complexity [58], is a measure derived from the product itself [44]. An external measure is a measure of a product derived from the external assessment of the behavior of the system [44]. For example, the number of failures found in test is an external measure.

The ISO/IEC standard [44] states that "internal metrics are of little value unless there is evidence that they are related to some externally visible quality." Internal metrics have been shown to be useful as early indicators of externally-visible product quality [3] when they are related (in a statistically significant and stable way) to the field quality/reliability of the product. The validation of such internal metrics requires a convincing demonstration that (1) the metric measures what it purports to measure and (2) the metric is associated with an important external metric, such as field reliability, maintainability, or fault-proneness[29].

Our research objective is to construct and validate a set of easy-to-measure in-process metrics that can be used as an early indication of an external measure of post-release field quality and provides meaningful feedback on the thoroughness of a testing effort. To this end, we have created a metric suite we call the Software Testing and Reliability Early Warning metric suite for Java (STREW-J) [69-71]. Software reliability is defined as the probability that the software will work without failure under specified conditions and for a specified period of time [64].

The STREW metrics are used to build a regression model to estimate the post-release field quality using the metric TRs/KLOC. The estimation of post release field quality via the STREW metric suite is applicable for development teams that write extensive automated test cases, such as is done in the Extreme Programming [8] software development methodology. The STREW method is not applicable for script-based automated testing because, as will be discussed, the metrics are primarily based upon the object-oriented (O-O) programming paradigm. Teams develop a history of the value of the STREW metrics from comparable projects with acceptable levels of field quality. These historical metric values are then used to estimate the relationship between the STREW metric

elements and the TRs/KLOC. In this dissertation, we present empirical results of an academic feasibility study (22 projects), a case study of open source projects (27 projects), and an industrial case study (five projects) designed to build and validate the STREW model.

The rest of this thesis is organized as follows. Chapter 2 provides the background introduction to software reliability, software testing, software metrics, and industrial metric programs. Chapter 3 outlines the prior research work related to the estimation of fault density and fault-proneness, and Chapter 4 presents the STREW metric suite. Chapters 5 and 6 provide the evaluation of the STREW metric suite in terms of estimating the post-release field quality and providing test quality feedback. Chapter 7 presents a retrospective analysis of the STREW metric suite, and Chapter 8 discusses the conclusions and future work.

CHAPTER 2

BACKGROUND

This section provides an introduction to the four main areas related to this proposal: software reliability, software testing, software metrics, and industrial metric programs.

2.1 SOFTWARE RELIABILITY

Software reliability is defined as the probability that the software will work without failure under specified conditions and for a specified period of time [64]. A number of software reliability models are available. They range from the simple Nelson model [73] to more sophisticated hyper-geometric coverage-based models [45], to component-based models, and object-oriented models [3]. Several reliability models use Markov Chain techniques [95]. Other models are based on the use of an operational profile, i.e., a set of software operations and their probabilities of occurrence [64]. These operational profiles are used to identify potentially-critical operational areas in the software to signal a need to increase the testing effort in those areas. A large group of software reliability growth models are described by Non-Homogenous Poisson Processes (NHPP) [98]. This group includes Musa [66] and the Goel-Okumoto [35] models.

In many test-centric methodologies, developers strive to pass all the automated tests that are written, and there are no measurable faults. Even if there are failures, these failures might not be an accurate reflection of the reliability of the software if the testing effort was not comprehensive.

Instead, “no failure” estimation models, as described in [26, 28, 59], may be more appropriate for use with such methodologies.

Software reliability models can be classified broadly into seven categories [97]:

- **Markov models:** A model belongs to this class if its probabilistic assumption of the failure process is essentially a Markov process, i.e. a birth-death process. In these models each state of the software has a transition probability associated with it that governs the operational criteria of the software.
- **Non-homogeneous Poisson process (NHPP) models**: A model is classified as a NHPP model if the main assumption is that the failure process is described by a NHPP. The main characteristic of this type of model is that there is a mean value function which is defined as the expected number of failures up to a given time.
- **Bayesian process**: In a Bayesian process model, some interesting information about the software to be studied is available before the testing starts, such as inherent fault density and defect information of previous releases. This information can be used in combination with the collected test data to make a more accurate estimation and prediction about the reliability.
- **Statistical data analysis methods:** Different statistical models and methods are applied for the analysis of software failure data. Some of these models are the time series model, proportional hazards model, and regression models.
- **Input-domain based models:** These models do not make any dynamic assumption about the failure processes. All possible input and output domains of the software are constructed and, based on the results of the testing, the faults in mapping between the input and output domains are identified, i.e. for a particular value in the input domain if the corresponding value in the output domain must be produced or a fault is identified.

- **Seeding and tagging models**: These models utilize the statistical capture-recapture technique that involves the artificial seeding of faults. The assessment of the testing is based upon the number of seeded faults that remain in the software at the conclusion of the testing effort.
- **Software metrics models:** Software reliability metrics which are measures of the software complexity can be used to estimate the number of software faults remaining in the software.

2.2 SOFTWARE TESTING

Software testing is a verification and validation (or V&V) software practice and is considered to be a software quality assurance practice. Software testing can be use to answer two main questions [10],

- Verification: Are we building the product right?
- Validation: Are we building the right product?

2.2.1 SOFTWARE TESTING CLASSIFICATIONS

As shown in Figure 2.1, testing activities can be classified as black box or white box. Black box testing [43], (also called functional testing) is testing that ignores the internal mechanism of a system or component and focuses solely on the outputs generated in response to selected inputs and execution conditions. Black box testing is used to simulate the customer behavior and focuses on input/output. White box testing [43], (also called structural testing) is testing that takes into account the internal mechanism of a system or component.

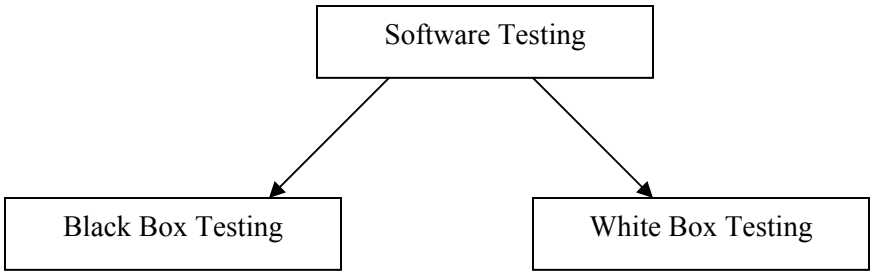


Figure 2.1: Testing Techniques

There are several levels of testing that should be done on a large software system. These levels are explained below [43].

1. **Unit Testing:** Testing of individual hardware or software units or groups of related units[43]. It is done at a very low structural level. The primary objectives of unit testing are to (1) verify the code against the component, i.e. to see if the code does what the component is expected to do with respect to the overall system; (2) execute all new and changed code to ensure all branches are executed in all directions, (3) check for the correctness of logic and data paths; and (4) exercise all error messages, return codes and response options [48].

2. **Integration testing:** Testing in which software components, hardware components, or both are combined and tested to evaluate the interaction between them. Integration testing involves uses both black and white box testing techniques[43]**.**
3. **Functional and System testing:** Using **black box testing** techniques, testing conducted on a complete, integrated system to evaluate the system's compliance with its specified requirements[43].
4. **Acceptance testing:** (1) Formal testing conducted to determine whether or not a system satisfies its acceptance criteria and to enable the customer to determine whether or not to

accept the system. (2) Formal testing conducted to enable a user, customer, or other authorized entity to determine whether to accept a system or component[43].

5. **Regression testing.** Selective retesting of a system or component to verify that modifications have not caused unintended effects and that the system or component still complies with its specified requirements[43].

2.2.2 AUTOMATED SOFTWARE TESTING EXAMPLE USING JUNIT

As this dissertation deals with leveraging the software testing effort for estimating post-release field quality, we present an example of the xUnit[1] type of software testing that our post-release field quality is based upon. Note the symmetries between the source and test code. This example involves a Java program to add, subtract, multiply and divide two numbers, as shown in Figure 2.2.

Source Program

```
//computation.java
//author-Nachiappan Nagappan
import java.io.*;
import java.lang.*;

public class computation
{
   //Add two numbers
   public static int addi(int temp1,int temp2)
   {
        int temp3;
        temp3=temp1+temp2;
        return(temp3);
   }

   //Subtract two numbers
   public static int subt(int temp1,int temp2)
   {
        int temp3;
        temp3=temp1-temp2;
        return(temp3);
   }

   //Multiply two numbers
   public static int mult(int temp1,int temp2)
   {
        int temp3;
```

[1] http://xprogramming.com/software.htm

```
            temp3=temp1*temp2;
            return(temp3);
      }

      //Divide two integers
      public static int divi(int temp1,int temp2)
      {
            int temp3;
            temp3=temp1/temp2;
            return(temp3);
      }

      public static void main(String args[])throws IOException
      {
            int x;
            computation nachi=new computation();
      }

}                        //End of Program
```

Figure 2.2: Example Java Source program

There are four methods that form the core of the source program, addi(), subt(), mult() and divi() which perform the operations of addition, subtraction, multiplication and division. The corresponding automated testing program written in JUnit[2] is given in Figure 2.3. This test program exercises the source program to check if the operations are correct based on specific test cases (e.g. testaddi() in Figure 2.3 to check if two integers are added correctly).

```
   Test Program
   //computationTest.java
   //author-Nachiappan Nagappan

   import junit.framework.*;
   import java.io.*;

   public class computationTest extends TestCase
   {
      public computation x;

      public computationTest(String name)
      {
            super(name);
      }

      public void setUp()
      {
```

[2] junit.org

```
            x=new computation();
      }

      //Test addi() method of computation.java
      public void testaddi()
      {
            x=new computation();
            assertEquals(100,x.addi(75,25));
      }

      //Test subt() method of computation.java
      public void testsubt()
      {
            x=new computation();
            assertEquals(50,x.subt(75,25));
      }

      //Test mult() method of computation.java
      public void testmult()
      {
            x=new computation();
            assertEquals(1875,x.mult(75,25));
      }

      //Test divi() method of computation.java
      public void testdivi()
      {
            x=new computation();
            assertEquals(3,x.divi(75,25));
      }

      public static void main(String[] args)
      {
      }
}               //End of Program
```

Figure 2.3: Example Java Test program

Upon execution of the above test program, all the four mathematical operations are tested and the JUnit output is produced, as shown in Figure 2.4. Figure 2.4 shows that four of four test cases pass and provides a green bar to visually demonstrate that all test cases passed.

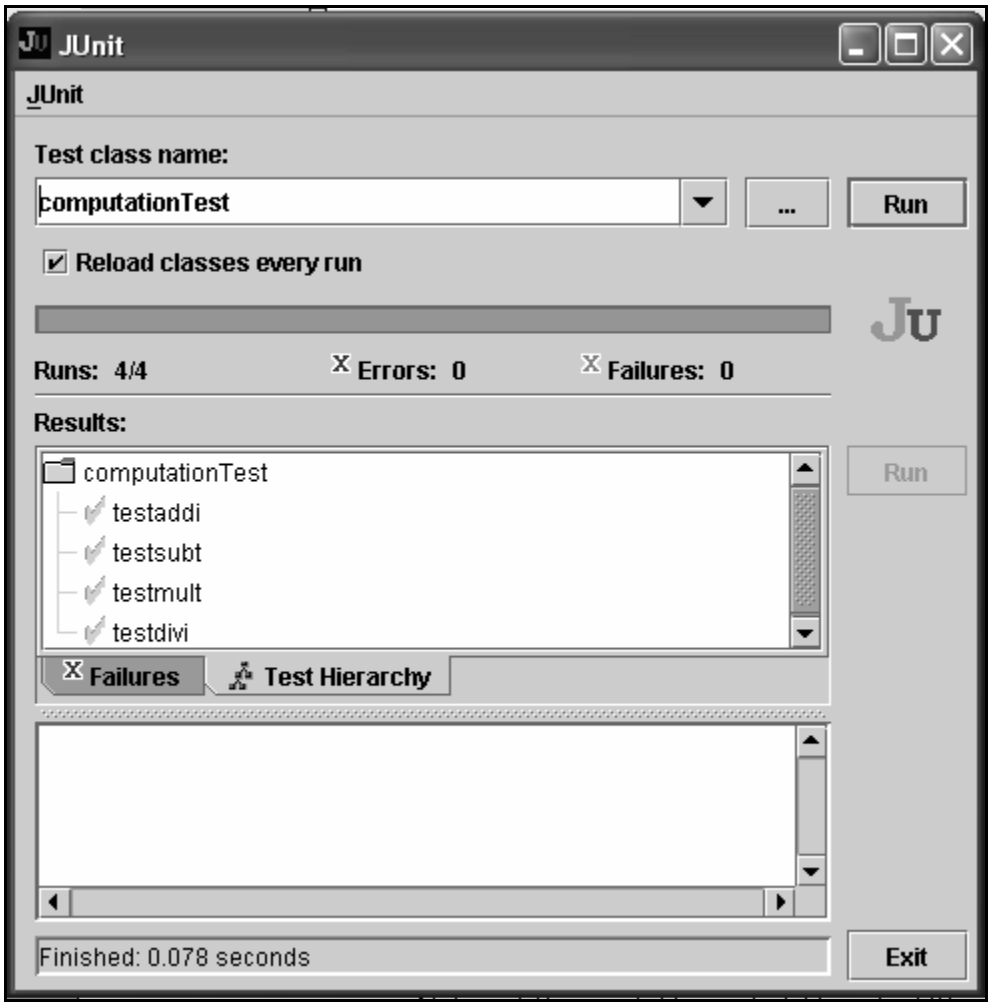


Figure 2.4: JUnit output screen shot

2.3 SOFTWARE METRICS

We present in this section a discussion on software metrics as we use in-process testing metrics in our dissertation research. The term *software metrics* explains many activities, all of which involve some degree of software measurement [33]:

- Cost and effort estimation
- Productivity measures and models
- Data collection
- Quality models and measures

- Reliability models
- Performance evaluation and models
- Structural and complexity metrics
- Capability-maturity assessment
- Management by metrics
- Evaluation of methods and tools.

Since there is virtually an infinite number of possible software metrics, developers must have some criteria for choosing which metrics to apply for a particular project. Ideally, a metric should possess the following characteristics [77]:

- *Simple*. The definition and use of the metric is simple;
- *Objective.* If different people perform the measurement, they will give similar values. Objective metrics allow for consistency and prevents individual bias;
- *Easily collected*. The cost and effort to obtain the measure is reasonable;
- *Robust*. The metric is insensitive to irrelevant changes, allowing for useful comparison; and
- *Valid*. A valid metric measures what it is supposed to measure, promoting trustworthiness of the measure.

Software metrics may be broadly classified into seven categories [77] as shown in Figure 2.5. The seven categories represent the seven different phases of software engineering from which metrics can be collected.

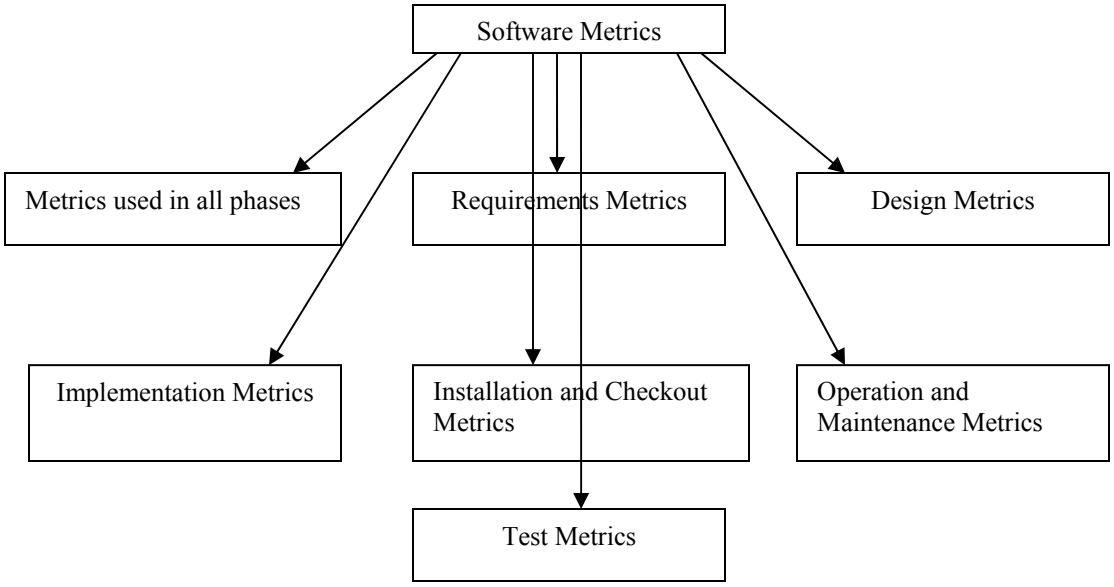


Figure 2.5: Software Metrics Classification

The STREW metric suite is comprised of metrics from both the source and test code. The following discussion presents a broad overview of the popular software metric measures. The metrics range from product, process metrics to metric measurement paradigms like the Halstead metrics, cyclomatic complexity, and function points.

2.3.1 TEST METRICS

We shall concentrate on a study of the test metrics [77] because in our work, we leverage the testing effort to assess the quality of the testing effort and to predict post-release field quality based on previous empirical data. Test metrics may be of two types [77],

1. Metrics related to test results or the quality of the product being tested.
2. Metrics used to assess the effectiveness of the testing process.

In this dissertation, we will be focusing on metrics of the second category. Further, we also provide an overview of some of the product (metrics obtained from the software) and process (metrics obtained from the software process employed in the development cycle) metrics that are commonly used.

2.3.1.1 **PRODUCT METRICS** (Defect/Error/Fault Metrics)

- Primitive defect/error/fault metrics. These are very simple, easy to collect metrics that are represented in terms of bar, line graphs and using histograms.
- Number of faults detected in each module.
- Number of requirements, design, and coding faults found during unit and integration testing.
- Number of errors by type (e.g. number of logical errors, number of computation errors, number of interface errors etc.)
- Number of errors by cause or origin
- Number of errors by severity (number of critical errors, number of major errors, number of cosmetic errors).
- Fault Density (FD): Number of faults/size in one thousand lines of code (KLOC)
    - FD may also be weighed in using the severity of errors as shown in Equation 2.1.

$$FD\ (Weighted) = W_1 S/N + W_2 A/N + W_3 M/N \qquad (2.1)$$

        - W_1, W_2, W_3 – Weights assigned.(User dependant)
        - N- Number of faults
        - S-Number of severe faults
        - A-Number of average severity faults
        - M-Number of minor faults
- Defect age: Defect age is the time between when a defect is introduced and when it is fixed. The average defect age of a product is computed using the summation of the individual defect ages for all the defects as shown in Equation 2.2.

$$defect\ age = \Sigma\forall(Phase\ detected - phase\ introduced)/\ Number\ of\ defects \qquad (2.2)$$

- Defect response time: This measure is the time between when a defect is detected to when it is fixed or closed.

- Defect Cost: The cost of a defect is the sum of the cost to analyze the defect, the cost to fix it, and the cost of failures already incurred due to the defect.
- Defect removal efficiency (DRE): The defect removal efficiency is the percentage of defects that have been removed during a process, computed via Equation 2.3.

*DRE = Number of defects removed/Number of defects at the start of the process * 100% (2.3)*

2.3.1.2 PROCESS METRICS

Test case metrics

- Total number of planned white/black box test cases run to completion.
- Number of planned integration tests run to completion
- Number of unplanned test cases required during the test phase.

Coverage metrics

- Statement coverage
- Branch coverage
- Path coverage
- Data flow coverage
- Test coverage

Failure metrics

- Mean time to failure (MTTF): This is the mean time for the next failure i.e. $(i+1)^{th}$ failure given the failure times of the previous i failures. This metric is a basic parameter required by most software reliability models.
- Failure rate: This is used to indicate the growth in the software reliability as a function of test time and is usually used with reliability models. Failure rate requires the observed time between failures at a given severity level and the number of failures in a given severity level in a particular time interval. The failure rate can calculated using R(t) as

shown in Equation 2.4, is obtained from the cumulative probability distribution (F(t)) of the time until the next failure, using a software reliability estimation model.

  - *The failure rate, $\lambda(t) = -1/R(t)\ [dR(t)]/dt$ where $R(t)=1-F(t)$ (2.4)*

- Cumulative failure profile. Uses a graphical technique to predict reliability to estimate additional testing time needed to reach an acceptable reliability level and to identify modules and subsystems that require additional testing.

Some of the more commonly used software metric(s)/suites are discussed below. These models provide an introduction to current trends in software metrics measurement.

2.3.2 HALSTEAD SOFTWARE SCIENCE

Halstead [37] proposes the measurement of a software system that is obtained by breaking down and arranging a finite number of program "tokens," which are basic syntactic units distinguished by a compiler. Halstead's measurement takes into account the number of distinct operators and operands that appear in a program; the total number of operator occurrences; and the total number of operand occurrences to calculate several program characteristics. These characteristics are program length (total number of operator and operand occurrences), volume (number of bits required to specify a program), vocabulary (total count of distinct operator and operand count), level (a measure of software complexity), and program effort.

Using the above metrics, Halstead designed a set of equations that express several of the program characteristics.

$$\textit{Vocabulary (n): } n = n_1 + n_2 \qquad (2.5)$$

$$\textit{Length (N): } N = N_1 + N_2 \qquad (2.6)$$

$$= n_1 \log_2(n_1) + n_2 \log_2(n_2)$$

$$\textit{Volume (V): } V = N \log_2 (n) \qquad (2.7)$$

$$= N \log 2\ (n_1 + n_2)$$

$$\textit{Level (L): } L = V^*/V \qquad (2.8)$$

$$= (2/n_1) * (n_2/N_2)$$

$$Difficulty\ (D) = 1/L \quad (2.9)$$

$$Effort\ (E): E = V/L \quad (2.10)$$

$$Faults\ (B): = V/S^* \quad (2.11)$$

Where:

- n_1- The number of distinct operators that appear in a program
- n_2- The number of distinct operands that appear in a program
- N_1- The total number of operator occurrences
- N_2- The total number of operator occurrences
- V* is the volume represented by the built in function performing the task of the entire program
- S* is the mean number of mental discriminations (decisions) between errors (S* is 3000 according to Halstead).

Halstead's metrics has been subject to criticism in several aspects, such as methodology, derivations of equations, human memory models [48]. Empirical support is lacking in several areas of the Halstead measures. The Halstead metrics are static metrics that ignore variations in fault rates observed in software products and among modules. However, Halstead's work was instrumental in making metrics an issue among computer scientists as it was the first formal investigation of software metrics. [48]

2.3.3 CYCLOMATIC COMPLEXITY

McCabe designed cyclomatic complexity [58] as a measure of the programs testability and understandability. Both testability and understandability impact maintainability [48]. Cyclomatic complexity is adapted from the classical graph theoretical cyclomatic number to suit software science and can be defined as the number of linearly-independent paths through a program. Cyclomatic complexity has been shown to be an indicator of the effort required to test a program [48].

The formula used to compute the complexity metric is shown in Equation 2.12.

$$M = V(G) = e-n+2p \qquad (2.12)$$

Where, V(G) = cyclomatic complexity of G

e = number of edges

n = number of nodes

p=number of unconnected parts of the graph.

To have good testability and maintainability, McCabe recommended that no program module should have a cyclomatic complexity greater than 10. Because it is based on decisions and branches, this complexity metric is consistent with the logic pattern of design and programming, which appeals to software professionals. Many experts recommend the use McCabe's cyclomatic complexity to ensure adequate test coverage, and the use of McCabe's cyclomatic measure has been gaining acceptance by practitioners [48].

2.3.4 FUNCTION-POINT ANALYSIS

Function point analysis [2] is an approach for assessing the size of a piece of software based on the functionality provided. Function points are obtained by the analysis of a requirements specification document to identify the different functions that the system is to perform. The functions are classified into different types and are given weightings according to the relative complexity of the function type [85].

For example, the unadjusted function point count 'ufc' for a specification is given by [85] Equation 2.13:

$$ufc = 4*i+5*o+4*e+7*p+10*f \qquad (2.13)$$

where, i = number of external input types

o = number of external output types

e = is the number of enquiries

p = is the number of external files (program interfaces)

f = is the number of external files

(This is a simplified equation from [85] to illustrate ufc)

Further, Albrecht suggests that a total of 14 complexity metric factors be taken into account for calculating the technical complexity factor (tcf) [85]. The 14 factors are data communications, distributed data processing, performance, heavily-used configuration, transaction rate, on-line data entry, end-user efficiency, on-line update, complex processing, reusability, installation ease, operation ease, multiple sites, facilitate change. Each factor is scored between zero (no influence) and five (very strong influence). The tcf is calculated as shown in Equation 2.13:

$$tcf = 0.65 + 0.01 * \sum_{i} DI \qquad (2.13)$$

, where DI_i is the degree of influence of the i^{th} technical complexity factor.

Thus, the *tcf* will range from 0.65 to 1.65. For a simple system with no data communications the value will tend towards 0.65. A distributed system dealing with high transaction volumes and characterized by complex processing have a *tcf* of approximately 1.35. Combining the technical complexity factor and the unadjusted function point count, we obtain the adjusted function point count (fp) as in Equation 2.14:

$$fp = ufc*tcf \qquad (2.14)$$

The function point is usually used as a predictor of the development effort, although its inverse is often also used as a productivity index. Also, for example, once the function points of an organization are calibrated, they have been found to explain 75% of the variation in program size in a study of 15 commercial software systems [49].

2.3.5 HENRY-KAFURA STRUCTURE METRIC

Structure metrics take into account the interactions between modules in a product or system and quantify such interactions. The information-flow metric defined by Henry and Kafura [41], uses *fan-in* (a count of the number of modules that call a given module) and *fan-out* (a count of the number of modules that are called by a given module) to calculate a complexity metric.

Henry and Kafura's structure complexity metric (C_p) is defined in Equation 2.15:

$$C_p = (fan\text{-}in * fan\text{-}out)^2 \qquad (2.15)$$

In general, modules with a large fan-in are relatively small and simple. In contrast, modules that are large and complex have a small fan-in. Thus, components with a large fan-in and large fan-out may indicate poor design. Such modules have to be decomposed correctly.

2.3.6 OBJECT-ORIENTED METRICS

In recent years, object-oriented (O-O) programming has gained importance, and new suites of metrics that exploit the O-O properties are becoming popular. Two popular O-O metric suites are presented below.

2.3.6.1 CK METRICS

The CK metric suite proposed by Chidamber-Kemerer (CK) [17]identifies six O-O metrics:

- Weighted Methods per class (WMC). the weighted sum of all the methods defined in a class;
- Coupling Between Objects (CBO): the number of other classes with which a class is coupled;
- Depth of Inheritance Tree (DIT): the length of the longest inheritance path in a given class;
- Number of Children (NOC): the count of the number of children (classes) that each class has;
- Response for a class (RFC): the count of the number of methods that are invoked due to the initiation of an object of a particular class; and
- Lack of Cohesion of Methods (LCOM): is a count of the number of method pairs whose similarity is zero and minus the count of method pairs whose similarity in not zero.

Several studies have been performed assessing the effectiveness of the CK Metrics in addressing software fault-proneness [3, 86, 88]. These studies are explained in detail in Chapter 3.

2.3.6.2 MOOD METRIC SUITE

The MOOD [15] O-O metric suite consists of the following O-O Metrics:-

- Method Hiding Factor (MHF): the number of visible methods;
- Attribute Hiding Factor (AHF): the number of visible attributes;
- Method Inheritance Factor (MIF): the ratio of the sum of inherited methods to the total number of methods.
- Attribute Inheritance Factor (AIF): the ratio of, the sum of inherited attributes to the total number of attributes.
- Polymorphism Factor (PF): the degree of method overriding in the class inheritance tree. PF equals the number of actual method overrides divided by the maximum number of possible method overrides.
- Coupling Factor: the actual number of couplings among classes in relation to the maximum number of possible couplings.

2.4 INDUSTRIAL METRIC PROGRAMS

This section provides an insight into the popular metric programs employed in three popular industrial organizations (Motorola, Hewlett-Packard, and IBM) to present an overview of the types of metrics that are measured in commercial software development organizations.

2.4.1 MOTOROLA

Motorola's software metrics program [21] follows the Goal/Question/Metric paradigm [6]. The goals and measurement areas identified by the Motorola Quality Policy for Software Development (QPSD) are listed below [48]:

Goals

- Goal 1: Improve project planning
- Goal 2: Increase Defect containment
- Goal 3: Increase Software Reliability

- Goal 4: Decrease software defect density
- Goal 5: Improve customer service
- Goal 6: Reduce the cost of nonconformance
- Goal 7: Increase software productivity

Measurement areas:

- Delivered defects and delivered defects per size
- Total effectiveness throughout the process
- Adherence to schedule
- Estimation accuracy
- Number of open customer problems
- Time that problems remain open
- Cost of nonconformance
- Software reliability

2.4.2 HEWLETT-PACKARD

Hewlett- Packard's software metrics program [36] uses several metrics and ratios to assess product quality. A subset of these metrics is given below [48]:

- Average fixed defects/working day
- Average engineering hours/fixed defect
- Average reported defects/working day
- Branches covered/ Total branches
- Defects/thousands of non-commented source statements
- Defects/Lines of Documentation not included in program source code
- Defects/Testing time
- non-commented source statements /engineering month
- Percent overtime: average overtime/40 hours per week

- Engineering months/Total Engineering months

2.4.3 IBM ROCHESTER

For the software community within IBM, a set of standard 5-UP software quality metrics is defined by the IBM corporate software measurement council. The 5-UP metrics include the following [48]:

- Overall customer satisfaction
- Post release defect rate for three years
- Customer problem calls
- Fix response time
- Number of defective fixes

IBM Rochester in addition to the above 5-UP metrics uses several other in-process metrics, such as phase effectiveness (for each phase of effectiveness and test); inspection coverage; effort; defect rates; in-process inspection escape rate; compilation of failures and build/integration defects; weekly defect arrivals and backlog during testing; defect severity; defect cause; reliability; models for post release defect estimation.

CHAPTER 3

RELATED RESEARCH

In this chapter, we present the related research work in two sections. Section 3.1 is a discussion of prior related studies done with software metrics, and Section 3.2 investigates the applicability of popular software reliability models within our research context.

3.1 PRIOR RELATED WORK

The higher the failure-proneness of the software, logically the lower the reliability and the quality of the software produced, and vice-versa. Software fault-proneness is defined as the probability of the presence of faults in the software [23]. Failure-proneness is the probability that a particular software element will fail in operation. Using operational profiling information, it is possible to relate failure-proneness and fault-proneness of a product. Research on fault-proneness has focused on two areas: (1) the definition of metrics to capture software complexity and testing thoroughness and (2) the identification of and experimentation with models that relate software metrics to fault-proneness [24]. While software fault-proneness can be measured before deployment (i.e. the count of faults per structural unit such as faults per line of code), failure-proneness cannot be directly measured on software before deployment [31]. Fault-proneness can be estimated based on directly-measurable software attributes if associations can be established between these attributes and the system fault-proneness.

Structural O-O measurements, such as those defined in the CK [17] and MOOD [15] O-O metric suites, are being used to evaluate and predict the quality of software [39]. Structural object-orientation (O-O) measurements, such as those in the Chidamber-Kemerer (C-K) O-O metric suite [17], have been used to evaluate and predict fault-proneness [3, 13, 14]. The CK metric suite consists of six metrics: weighted methods per class (WMC), coupling between objects (CBO), depth of inheritance tree (DIT), number of children (NOC), response for a class (RFC) and lack of cohesion among methods (LCOM). These metrics can be a useful early internal indicator of externally-visible product quality in terms of fault-proneness [3, 86, 88].

In software systems, the actual measurable product quality (e.g., failure rate) that is derived from the behavior of the system usually cannot be measured until too late in the life-cycle to effect an affordable corrective action. In general, a multi-phase approach must be taken collecting the various metrics of these suites at different stages, since different metrics will be visible at different development phases [92, 93].

Basili et al. [3] studied the fault-proneness in a class on eight student projects. It was observed that the WMC, CBO, DIT, NOC and RFC were correlated with defects while the LCOM was not correlated with defects. Further, Briand et al. [14] performed an industrial case study and observed the CBO, RFC, LCOM to be associated with the fault-proneness of a class. A similar study done by Briand et al. [13] on eight student projects showed that classes with a higher WMC, CBO, DIT and RFC were more fault prone while classes with more children (NOC) were less fault prone (LCOM was not associated with the defects). Tang et al. [88] studied three real time systems for testing and maintenance defects. Higher WMC and RFC were found to be associated with fault-proneness. El Emam et al. [30] studied the effect of project size on fault-proneness by using a large telecommunications application. Size was found to confound the effect of all the metrics on fault-proneness. In addition to this, Chidamber et al.[17] analyzed project productivity, rework, and design effort of three financial services applications. High CBO and low LCOM were associated with lower productivity, greater rework, and greater design effort. To summarize, there is a growing body of

empirical evidence that supports the theoretical validity of the use of these internal metrics [3, 13] as predictors of fault-proneness. The consistency of these findings varies with the programming language [86]. Therefore, the metrics are still open to criticism. [19]

The relationship between product quality and process capability [82] and maturity has been recognized as a major issue in software engineering based on the premise that improvements in process will lead to higher quality products. The process capability is defined as the ability of a process to address the issue of stability, as defined and evaluated by trend or change. Such a relationship between product quality and process capability should manifest itself via meaningful metrics that would exhibit trends and other characteristics that would be indicative of the stability of the process. Using the Space Shuttle software Schneidewind reports an assessment of long term metrics, such as MTTF, total failures per KLOC change in code (churn), total test time normalized by KLOC change in code, remaining failures normalized by KLOC, change in code, and predicted time to next failure to be indicative of the stability of the software process with respect to process capability [82].

Several techniques have been used for the analysis of software quality (errors[3]) with respect to program metrics. Linear regression analysis techniques have been used to relate quality factors, such as defect density and reliability, to software metrics. These regression models are built using the best fit among all the data available and can be used to predict the software quality factors accordingly using the current values of the metrics for programs that are being analyzed[63]. Discriminant analysis, a statistical technique that is used to categorize programs into groups (high, moderate, low quality) based on the programs metric values, are also used as a tool for the detection of fault-prone programs. Munson et al. demonstrated the efficacy of discriminant analysis by using the technique called *data-splitting*. From a total of 390 programs, 260 were randomly-selected and were used to

[3] (1) The difference between a computed, observed, or measured value and the true, specified, or theoretically correct value or condition. (2) An incorrect step, process, or data definition (3) An incorrect result (4) Human action that produces an incorrect result.

build the discriminant model. The remaining 130 programs were used to test the efficacy of the model to classify programs according to software faults. Discriminant analysis has been shown to work well for programs with a low error rate, but the linear regression models shows greater promise for use with programs with a high potential for faults[62]. Further, using Binary Discriminant Factors (BDFs) makes fewer mistakes in classifying software that is of low quality than in the case with linear vectors of metrics [83].

Also, optimized set reduction (OSR) techniques and logistic regression techniques are used for modeling risk and fault data. OSR techniques attempts to determine which subsets of observations from historical data provide the best characterization of the programs being assessed. Each of these optimal subsets is characterized by a set of predicates (a pattern), which can be applied to classify new programs. OSR is sometimes better than a logistic regression analysis for multivariate empirical modeling since pattern-based classification is more accurate than logistic regression equations [12]. Further, logistic regression models can be built that relate software measures and software fault proneness for classes of homogenous software products [24]. Also, multivariate models can be built with logistic regression where Principal Component Analysis (PCA) is used on the of metrics to model the data [25]. Denaro et al. calculated 38 different software metrics for the open source Apache 1.3 and Apache 2.0 projects. Using PCA, they selected a subset of consisting of nine of these metrics was found to explained 95% of the total variance. Using combinations of these nine metrics, logistic regression models were built using the data from the Apache 1.3 project and verified against the Apache 2.0 project [25]. We believe that a judicious use of early metrics, in conjunction with an understanding of the software process can be a powerful tool in guiding the development of good quality software.

3.2 SOFTWARE RELIABILITY MODELS

For our research, we must select an appropriate estimation model which can take as input a quantification of the automated testing effort. We considered a large number of published software reliability estimation models. Early in the research, the use of an operational profile-based reliability model was determined to be impractical for use with development teams that write extensive automated test cases (i.e. employ automated testing methods (ATM)). ATM is the existence of a dual hierarchy of executable source and test code that work in parallel, for example Java source programs and Junit test programs. An example for this is provided in Chapter 4. The infeasibility of using operational profiles was determined based upon the results of three case studies [68] carried out in industrial locations, John Deere, Rolemodel Software, and Nortel. We had initially assumed there was a correspondence between the customer requirements and the developers automated acceptance test cases. This assumption came from the idea that, during the requirements elicitation process, the developers and customers would create the requirements and then the developers write acceptance tests that dealt specifically with that certain requirement. In this way, developers could prove to the customer that a given requirement was completed by demonstrating that its acceptance test(s) passed. Thus, we expected that there would be a correspondence between a requirements and its acceptance test case(s) for the life of the product.

However, we found that developers tend to aggregate a minimal set of acceptance test cases, each used to satisfy several of the customer's requirements. For example, the developers could have an acceptance test that tested a certain core capability of the program. After this test passes, this core capability is deemed to be implemented. As a result, developers consider that they do not need to keep that test in its current form any more, and they build upon this test case to demonstrate more complex behavior of another requirement. As more functionality is added to the system, developers may alter/add additional conditions to acceptance test cases so that one acceptance test may be used

for multiple requirements. Therefore, developing a mapping between requirements and acceptance test cases was shown to not be practical.

The main problems with using an operational profile-based model were identified as:

(1) Developers would need to change their automated acceptance testing habits to eliminate the reuse and alteration of test cases;

(2) It is almost impossible to tie together the customer requirements and developer's acceptance test cases as developers tend to aggregate acceptance test cases to satisfy several of the customer's requirements; and

(3) The requirements of a software system continuously evolve across a release and this evolution requires the developers to constantly reassign and recalculate the operation profiles of the acceptance and unit tests which involves tremendous overhead. The developers in the case study would not consider such a change to their current agile methodology.

This lead us to the conclusion that the use of an operational profile model may not be feasible and our approach would have to focus on using a non-operational profile models [78] for estimation of reliability.

We set out to base our estimation model upon existing reliability models. The results of the first round of the model selection effort are summarized in Tables 3.1 and 3.2. The selection criterion that was used is based on the following constraints:

1. An operational profile-based model is impractical for our work.
2. In an ATM, all the test cases pass and there are no measurable failures.
3. Time is not monitored during testing/running of test cases in ATM. Therefore, defects per unit time of operation cannot be measured.
4. No information regarding the inherent defect density (i.e. the number of defects remaining in the software) is available. The inherent defect density is generally obtained based on defects in previous released versions (or) based on results after testing stages

like Functional Verification test (FVT) at the end of which a testing group would say, for example that there are 5 defects/KLOC based on their testing principle.

Twenty-two existing models were analyzed for identifying their fit within our problem domain. We classified these models into one of four categories: Candidate, Bad Model Fit, Bad Data Fit, and Overall Poor Fit. Table 3.1 shows the legend for the classification of the models. For example, if a model could be used with an ATM but needed data that was not available with an ATM, it was classified as a "Bad Data Fit." In the data applicability criterion we assess if the product data (metrics) available from software systems can be used for the modeling purposes. Information regarding assumptions about the remaining number of failures in the system or process information about the software system which is not easy to measure forms a crucial part in classifying models as a bad data fit.

Table 3.1: Legend for Classification of Software Reliability Models

Applicability to data available	Applicability to ATMs	Model Classification
Yes	Yes	Candidate
Yes	No	Bad Model Fit
No	Yes	Bad Data Fit
No	No	Overall Poor Fit

The classification of the twenty-two software reliability models according to the above classification is shown in Table 3.2. The far majority of the models have a Bad Model Fit because they require information about the previous failures (or require failures to occur) which would make the model inapplicable if the system had no failures. Ultimately, we used a empirical metric model that is found in the Candidate classification.

Table 3.2: Software Reliability model classification

Candidate Models
Empirical Metric Models, Lipow [55]: Several metrics are collected from previously-successful software programs. Using these metrics, a regression equation is framed that is used to estimate the reliability of similar programs. If data from similar previous projects is available for calibration, this model can be used.
Regression Models[63]: A linear regression analysis is investigated between the faults and certain selected metrics and parameters, such as field project quality. Works well with all the data available.
Zero Failure Model [59]: The zero failure model determines the reliability when there are no failures, as shown below. To use the Zero-Failure model, we must identify a meaningful long term failure rate denoted by Θ and that N random tests have established an upper confidence bound $(1-\alpha)$ that Θ is below some level θ [38]. The relationship between these factors is given by $1- (1-\theta)^N <= \alpha$ [89]. Easy applicability but does not work for large systems; Used in our feasibility studies.
Bad Model Fit
Nelson Model[73]: This is a very simplistic model based on the number of test failures. $R = 1-(\check{n}/n)$. where • $\check{n}$ - number of failures during testing. • n-total number of testing runs. • R is the system reliability. If no failures are available, the reliability becomes 100% which might always not be the case as software systems usually have failures.

Table 3.2 (continued)

Fault seeding models, Schick and Wolvertone [80] and Duran and Wiorkowski [27]: The model requires fault injection whereby faults are intentionally injected into the software by the developer. The testing effort is evaluated based upon how many of these injected defects are surfaced during testing. Using the number of injected defects remaining, an estimate on the reliability based on the quality of the testing effort is computed. Infeasible because it requires process changes needing developers to do fault injection.
Hypergeometric Distribution, Tahoma et al. [87]: The number of faults experienced by test instance t(i) is required. Using the number of faults experienced by each test instance, the overall system reliability can be determined. But in ATM there are no measurable failures and usually the number of failures in each testing instance is not measured or tracked. With ATMs, all the test cases pass, and there are no faults/failures to analyze.
Fault Spreading Model, Wohlin and Korner [96]: Requires number of faults at a level (or testing cycle/stage). The number of faults at each level is utilized to make predictions about untested areas of the software. With ATMs, all the test cases pass, and there are no faults/failures to analyze.
Fault Complexity Model, Nakogawa and Hanata [72]: Ranking of faults according to complexity. Based on the number of faults in each complexity level, the reliability of the system is estimated based on the current complexity level (high, moderate, low) of the software. With ATMs, all the test cases pass, and there are no faults/failures to analyze.
Littlewood-Verall Model [56]: Requires a scale parameter, $\Psi(i)$, that is used for describing the quality of the test, and is a monotonically-increasing function. Failures are exponentially distributed with a parameter assumed to have a prior Gamma distribution. With ATMs, all the test cases pass, and there are no faults/failures to analyze.

Table 3.2: (continued)

Jelinski-Moranda (JM) Model [46]: In the JM model, the initial number of software faults in unknown but fixed (i.e. non-increasing), and the times between the failures are exponentially distributed random quantities. Using this information, the JM model is modeled as a Markov process model. But in ATMs, there are no failures, and the times of the failure are also not measured.
Bayesian Formulation of the JM model, Langberg-Singpurwalla [54]: This models the parameters in the JM model as random variables. Poor fit for the same explanation as for the JM model.
Bayesian Model for fault free probability, Thompson and Chelson [90]: This model delas with the probability of fault-free software. Reliability at time t, $R(t/\lambda,p) = (1-p) + pe^{-\lambda t}$, where λ is given by a prior gamma distribution and p (probability that software is not fault-free) is given by an beta distribution. Using these parameters, a Bayesian model is constructed to estimate the reliability. This model cannot be used because lamda and p cannot be fixed without prior information of the defect density. Also, the defects may not follow a Beta Distribution and with ATMs, all the test cases pass, and there are no faults/failures to analyze.
Bayesian Model using a Geometric Distribution, Liu [57]: In this model let, X_i be the number of test cases at the i^{th} debugging instance at which the first failure will occur. Using this value, the number of failures remaining at the current debugging instance can be determined. With ATMs, all the test cases pass, and there are no faults/failures to analyze.

Table 3.2: (continued)

Goel-Okumoto Model [35]: The mean value function of the failures at time t is given by, $m(t)=a(1-e^{-bt})$ where in time *t* the cumulative failures are observed; a and b are parameters defined from the collected failure data. With ATMs, all the test cases pass, and there are no faults/failures to analyze.
S-shaped model,Yamada, Ohba et.al. [98] : $m(t)= a[1-(1+bt)e^{-bt}]$ Where, • a is the number of faults detected • b is the failure detection rate With ATMs, all the test cases pass, and there are no faults/failures to analyze.
Basic Execution Time Model [65] : $\lambda(\tau) – fK[N_0-\mu(\tau)]$ Where, • f and K are parameters related to the testing phase, initial fault density (N_0) • $\mu(\tau)$-faults corrected after τ amount of testing • $\lambda(\tau)$ is failure rate function at the execution time τ. This model cannot be applied as we do not have the initial fault density, and the failure rate function at execution time τ with the ATMs.
Logarithmic Poisson Model [67] : Based on the Basic Execution Time Model (above). $\lambda(\tau)= \lambda_0 e^{-\Phi\mu(\tau)}$ Where, • λ_0 is the initial failure intensity • Φ is the failure intensity decay parameter. The time, t, parameter is not available as in ATMs.

Table 3.2: (continued)

Duane Model also known as the **Weibull Process Model** [26] : $m(t)= (t/\alpha)^{\beta}$ The two parameters α and β are based on failure data, m(t) is the mean value function of the failures at time t. With ATMs, all the test cases pass, and there are no faults/failures to analyze.
Markov Models, Shantikumar [84] and Whittaker and Poore [75]: Requires transition probabilities from state to state. Using this information a stochastic model is created and analyzed for stability. This is primarily because there can be a very large number of states in a large software program. Determining the states and the transition between the states would require a process change that is likely to be unwelcome to developers that utilize a ATM.
Fourier Series Model, Crow and Singpurwalla [20]: Fault clustering and time series analysis form a basis of this model. Using time series analysis, the model predicts how clustered the faults will be at a given point in time. The 'time' parameter is not available as in ATM only the number of tests (or testing runs) can be measured. With ATMs, all the test cases pass, and there are no faults/failures to analyze.
Bad Data Fit
Input Domain-based Models, Bastani and Ramamoorthy [7] and Weiss and Weyuker [94]: Input domain is denoted by I of program P. I is mapped to output space O. If there is a fault in mapping, then that mapping is identified as a potential fault to be rectified. Applicable, but infeasible to map the domains are there can be a very large number of possibilities in a large industrial case study.

Table 3.2 (continued)

Overall Poor Fit
Halstead Metrics [37], modified by Schneider [81]: Halstead metrics are collected for the programs. Using these metrics, the reliability of the system is estimated using a fixed predefined equation. Halstead's metrics has been subject to criticism in several aspects, such as methodology, derivations of equations, human memory models [48]. Empirical support is lacking in several areas of the Halstead measures. Data is calibrated according to old FORTRAN programs that are no longer valid as software development has moved to language like C, C++ and Java. Also the Halstead metrics also do not take into account OO paradigms.

CHAPTER 4

STREW METRIC SUITE

The (Software Testing and Reliability Early Warning) STREW metric suite consists of internal, in-process testing metrics that are leveraged to estimate post-release field quality with an associated confidence interval. The use of the STREW metrics is predicated on the existence of an extensive suite of automated unit test cases being created as development proceeds. These automated unit tests need to be structured as is done with the one of the O-O xUnit[4] testing frameworks, such as JUnit[5]. The STREW method is not applicable for script-based automated testing because, as will be discussed, the metrics are primarily based upon the O-O programming paradigm. When these xUnit frameworks are used with O-O programming, both test code and implementation code hierarchies emerges. Figure 4.1 presents a simplistic example of a parallel structure between the source and test class. For each implementation source code class (e.g. `computation`), there exists a corresponding test code class (e.g. `computationTest`). Often each method/function in an implementation source code class (e.g. `addi`) will have one or more corresponding test code method(s)/functions(s) (e.g. `testaddi`). In industrial practice, often such perfect parallel class structure and one-to-one method/function correspondence is not observed; our example is overlysimplistic for illustrative purposes. However, a test hierarchy which ultimately inherits from the TestCase class (in JUnit) is created to exercise the implementation code.

[4] http://xprogramming.com/software.htm
[5] http://junit.org/

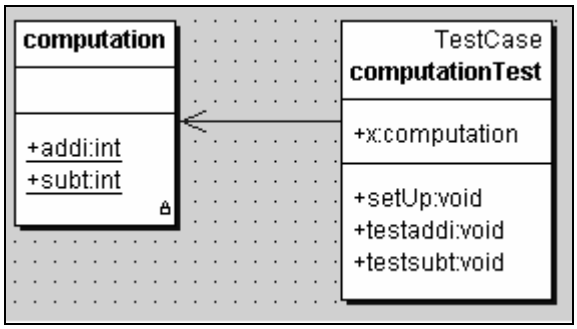


Figure 4.1: Corresponding source and test classes

Table 4.1 provides sample Java code for a computation class which adds and subtracts two integers. Notice the `assertEquals` keyword in the `testaddi` and `testsubt` methods. xUnit testing is assert-based. The number of asserts and other metrics specific to xUnit-like test automation are used in the STREW metric suite.

Table 4.1: Corresponding source and test code

```
public class computation
{
   public static int addi(int temp1,int temp2) {
         int temp3
         temp3 = temp1 + temp2;
         return(temp3);
   }
   public static int subt(int temp1,int temp2) {
         int temp3
         temp3 = temp1 - temp2;
         return(temp3);
   }
```

```
public class computationTest extends TestCase
{
      public computation x;
      public void setUp() {
             x=new computation();
      }
      public void testaddi() {
             assertEquals(100,x.addi(75,25));
      }
      public void testsubt() {
             assertEquals(40,x.subt(50,10));
      }
}
```

The STREW-J Version 2.0 metric suite consists of nine constituent metric ratios. The metrics are intended to cross-check each other and to triangulate upon an estimate of post-release field quality. Each metric makes an individual contribution towards estimation of the post-release field quality but work best when used together. Development teams record the values of these nine metrics and the actual TRs/KLOC of projects. These historical values from prior projects are used to build a regression model that is used to estimate the TRs/KLOC of the current project under development. For our case studies, we collect the TRs from customer-reported problems. These problems were screened. Duplicates and TRs involving documentation problems were removed.

The nine constituent STREW metrics (SM1 – SM9) and instructions for data collection and computation are shown in Table 4.2. The metrics can be categorized into three groups: test quantification metrics, complexity and O-O metrics, and a size adjustment metric.

The **test quantification metrics** (SM1, SM2, SM3, and SM4) are specifically intended to crosscheck each other to account for coding/testing styles. For example, one developer might write fewer test cases, each with multiple asserts [79] checking various conditions. Another developer might test the same conditions by writing many more test cases, each with only one assert. We intend for our metric suite to provide useful guidance to each of these developers without prescribing the style of writing the test cases. Assertions [79] are used in two of the metrics as a means for demonstrating that the program is behaving as expected and as an indication of how thoroughly the source classes have been tested on a per class level. SM4 serves as a control measure to counter the confounding effect of class size (as shown by El-Emam [30]) on the prediction efficiency;

Table 4.2: STREW metrics and collection and computation instructions

Test quantification		
$\frac{\text{Number of Assertions}}{SLOC^{*}}$	SM1	Count the number of assertions in all test cases for all source files.
$\frac{\text{Number of Test Cases}}{SLOC^{*}}$	SM2	Count the number of test cases to test all source files.
$\frac{\text{Number of Assertions}}{\text{Number of Test Cases}}$	SM3	Count the number of assertions in all test cases for all source files. Count the number of test cases to test all source files.
$\frac{(TLOC^{+}/SLOC^{*})}{(\text{Number of Classes}_{Test}\ \text{Number of Classes}_{Source})}$	SM4	Count the number of classes for all source files. Count the number of classes for all test files.
Complexity and O-O metrics		
$\frac{\Sigma\ \text{Cyclomatic Complexity}_{Test}}{\Sigma\ \text{Cyclomatic Complexity}_{Source}}$	SM5	Compute the sum of the cyclomatic complexity of all the test files. Compute the sum of the cyclomatic complexity of all the source files.
$\frac{\Sigma\ CBO_{Test}}{\Sigma\ CBO_{Source}}$	SM6	Compute the sum of the CBO of all the test files. Compute the sum the CBO of all the source files.
$\frac{\Sigma\ DIT_{Test}}{\Sigma\ DIT_{Source}}$	SM7	Compute the sum of the DIT of each file for all the test files. Compute the sum of the DIT of each file for all the source files.
$\frac{\Sigma\ \text{WMC}_{Test}}{\Sigma\ \text{WMC}_{Source}}$	SM8	Sum the WMC in each file for all the test files. Sum the WMC in each file for all the source files.
Size adjustment		
$\frac{SLOC^{*}}{\text{Minimum } SLOC^{*}}$	SM9	Divide the SLOC* of by the SLOC of the smallest project used to build the STREW.

** Source Lines of Code (SLOC) is computed as non-blank, non-comment source lines of code*
\+ Test Lines of Code (TLOC) is computed as non-blank, non-comment test lines of code

The **complexity and O-O metrics** (SM5, SM6, SM7, and SM8) examines the relative ratio of test to source code for control flow complexity and for a subset of the CK metrics. The dual hierarchy of the test and source code allows us to collect and relate these metrics for both test and source code. These relative ratios for a product under development can be compared with the historical values for prior comparable projects to indicate the relative complexity of the testing effort with respect to the source code. The metrics are now discussed more fully:

- The *cyclomatic complexity* [58] metric for software systems is adapted from the classical graph theoretical cyclomatic number and can be defined as the number of linearly independent paths in a program. Prior studies have found a strong correlation between the

cyclomatic complexity measure and the number of test defects [91]. Studies have also shown that code complexity correlates strongly with program size measured by lines of code [48] and is an indication of the extent to which control flow is used. The use of conditional statements increases the amount of testing required because there are more logic and data flow paths to be verified [50].

- The larger the inter-object *coupling*, the higher the sensitivity to change [17]. Therefore, maintenance of the code is more difficult [17]. Prior studies have shown CBO has been shown to be related to fault-proneness [3, 13, 14]. As a result, the higher the inter-object class coupling, the more rigorous the testing should be [17].
- A higher *DIT* indicates desirable reuse but adds to the complexity of the code because a change or a failure in a super class propagates down the inheritance tree. The relationship between the DIT and fault-proneness [3, 13] was found to be strongly correlated.
- The *number of methods* and the *complexity of methods* involved is a predictor of how much time and effort is required to develop and maintain the class [17]. The larger the number of methods in a class, the greater is the potential impact on children, since the children will inherit all the methods defined in the class. The ratio of the WMC_{test} and WMC_{source} measures the relative ratio of the number of test methods to source methods. This measure serves to compare the testing effort on a method basis. The relationship between the WMC as an indicator of fault-proneness has been demonstrated in prior studies[3, 13].

The final metric is a **relative size adjustment factor (RCAF)**. Defect density has been shown to increase with class size [30]. We account project size in terms of SLOC for the projects used to build the STREW prediction equation using the size adjustment factor.

The metrics that comprise the STREW metric suite have evolved [69-71] through our case studies. Some metrics were removed based on the lack of their ability to contribute towards the estimation of post-release field quality and due to already existing inter-correlations between the

elements. The metric revision process is illustrated in Figure 4.2. The standards for these metrics are used to provide feedback on the quality of the testing effort.

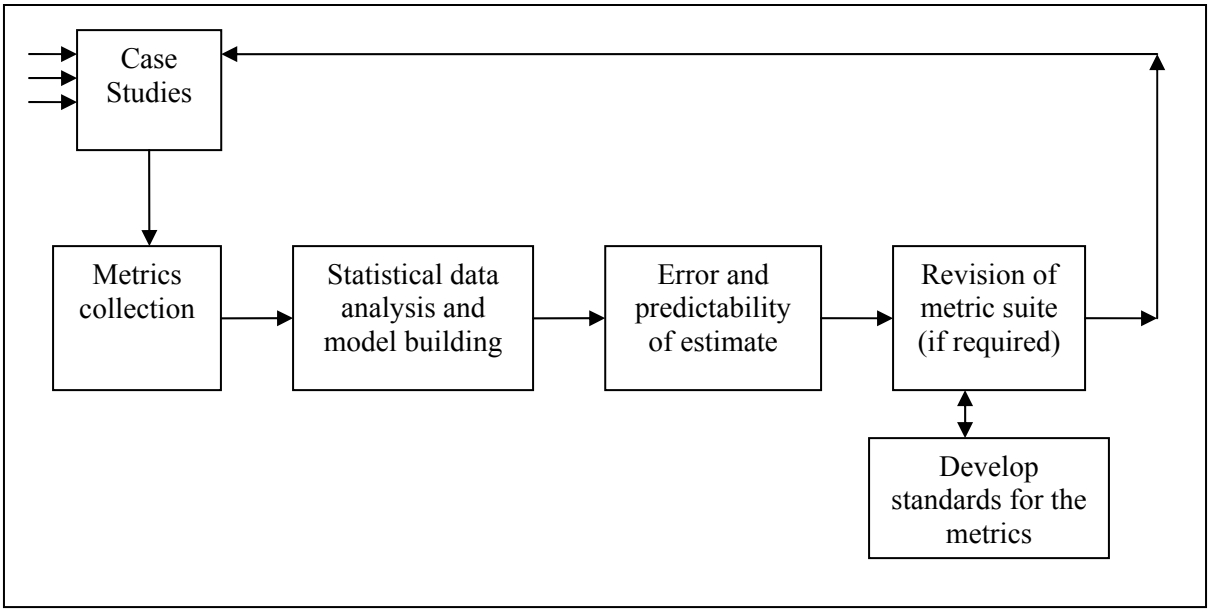


Figure 4.2: STREW metric revision process

As part of the STREW revision process, metrics were removed based upon by statistical inter-correlations, multicollinearity, stepwise, backward, and forward regression techniques [53] of case study data. The following metrics were removed:

- Statement coverage
- Branch coverage
- Number of requirements/Source lines of code
- Number of $children_{test}$/Number of $children_{source}$
- Lack of cohesion among $methods_{test}$/Lack of cohesion among $methods_{source}$

To better explain the collection of metrics, we present two examples below that highlight the metrics that are collected. The usual metrics like the LOC_{source} and LOC_{test} and the number of classes are not explained. Cyclomatic complexity is a measure of the control flow complexity in a program. Figure 4.3 shows the control flow graph for explaining the computation of the cyclomatic complexity.

Using the computation for cyclomatic complexity (Equation 2.12) we calculate cyclomatic complexity to be 7-6+2 = 3, where 7 is the number of edges, 6 is the number of nodes and one component is present in Figure 4.3.

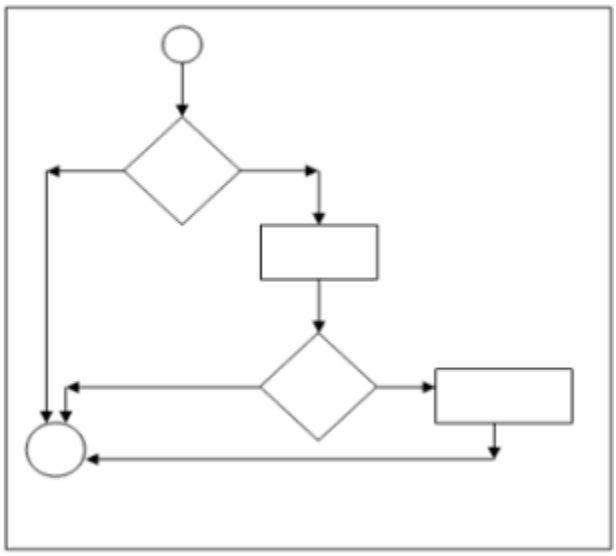


Figure 4.3: Example cyclomatic complexity computation

Consider the program shown in Figure 4.4. The left hand column shows a program to add two numbers. The corresponding test program written to test it is shown in the right hand column. The underlined parameters in the test program namely testaddi() is counted as a test case and the parameter *assertEquals*(100,x.addi(75,25)) sends in the input numbers 75 and 25 to the source program and checks if the computed sum is equal to 100. This counts as one assertion. The second example shown in Figure 4.5 explains the measurement of the O-O measures, coupling between objects, depth of inheritance tree and weighted methods per class.

```
public class computation
{
    public static int addi(int temp1,int temp2)
    {
        int temp3;
        temp3=temp1+temp2;
        return(temp3);
    }
    public static void main(String
args[])throws IOException
    {
        int x;
        computation nachi=new
computation();
    }
}
```

```
public class computationTest extends TestCase
{
    public computation x;
    public computationTest(String name)
    {
        super(name);
    }
    public void setUp()
    {
        x=new computation();
    }
    public void testaddi()
    {
        System.out.println();
        x=new computation();
        assertEquals(100,x.addi(75,25));
    }
    public static void main(String[] args) {
    }
}
```

Figure 4.4: Sample assertion and test case measurement example

```
import java.io.*;
import java.lang.*;
public class A ----------→ DIT = 1
{
    public static void w() ---→ WMC = 1
    {
    }
    public static void x() ---→ WMC = 1
    {
    }
}
class B extends A  ----------→ DIT = 2
{
}
class C extends B ----------→ DIT = 3
{
    public void z() ----------→ WMC = 1
    {
    }
}
```

```
class D ----------→ DIT = 1
{
    public void p() ---------→ WMC = 1
    {
        A TclassA = new A(); →CBO=1
TclassA.w();
    }
}

Metric Values
WMC = 4
DIT = 7
CBO = 1
```

Figure 4.5: CBO, DIT and WMC measurement example

From the above example code we calculate the WMC, DIT and CBO as shown below.

1. The WMC, considering a uniform weighting of 1 per method shows that there are 4 methods, (w(), x(), z(), p()) so WMC is 4.
2. The CBO, showing the coupling between the classes in A and D by using the object TclassA is 1.
3. The DIT, is the sum of the inheritance of the classes A(), B(), C() and D() as shown in Figure 4.5 which is 7.

The main limitations associated with the use of the STREW metrics suite are as follows:

- *The existence of automated tests*. To quantify the extent of testing done measurably in terms of assertions and test cases, we need the existence of automatic test cases which use the xUnit-style of testing. The STREW metric suite, in its current form, cannot leverage the manual black box testing results.
- *The existence of prior comparable projects*. To some extent, this limitation is alleviated by having comparable projects by some other organization that has acceptable post-release field quality. The prediction equations used in this dissertation can serve as a starting point when there is a lack of comparable projects.
- *Model robustness.* The better the consistency of data collected, the better the prediction accuracy. Consider the scenario of data from student projects being used to predict the post-release field quality of a software system for space shuttles. There will be errors in estimation as both the systems are from different environments with different post-release field quality requirements. Hence, in building the models with data collected from different environments there can be unfair bias towards one particular category of projects that cause an inflated or wrong prediction in terms of estimating the post-release field quality and providing test quality feedback.

CHAPTER 5

POST-RELEASE FIELD QUALITY ESTIMATION

This chapter discusses a preliminary feasibility study and the three-phased validation results obtained using the STREW metric suite for estimating post-release field quality. The validation studies are performed in three different environments to build an empirical body of knowledge, as shown in Figure 5.1. We discuss the model building activities, the model evaluation, and finally the results obtained using the three different set of case studies. Drawing general conclusions from empirical studies in software engineering is difficult because any process depends to a large degree on a potentially large number of relevant context variables. For this reason, we cannot assume a priori that the results of a study generalize beyond the specific environment in which it was conducted [5]. Researchers become more confident in a theory when similar findings emerge in different contexts [5]. By performing multiple case studies and/or experiments and recording the context variables of each case study, researchers can build up knowledge through a family of experiments [5] which examine the efficacy of a new practice. Replication of experiments addresses threats to experimental validity. We address these issues related to empirical studies by replicating multiple case studies through a family of experiments in three different (academic, open source and industrial) contexts. Similar results in these contexts indicate the promise of this approach at statistically significant levels.

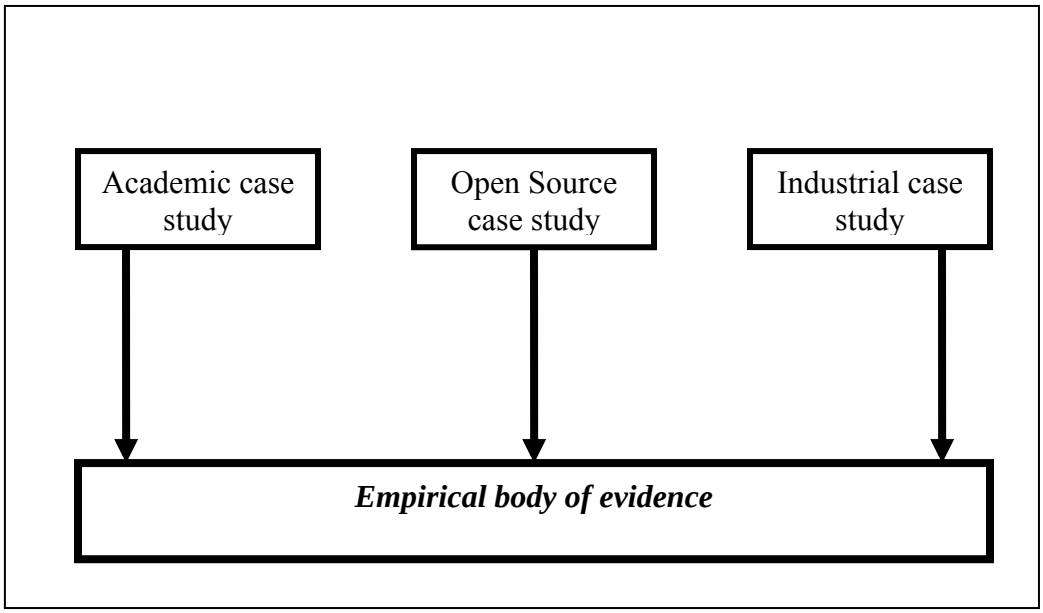


Figure 5.1: Empirical case studies

Section 5.1 presents the feasibility study on Version 1.1 of the STREW metric suite using the Zero-Failure model. Section 5.2 presents the model building techniques for the case studies performed with STREW Version 2.0 metric suite, and Section 5.3 explains the results evaluation techniques for the case studies. Section 5.4, 5.5 and 5.6 present the academic, open source and industrial case studies.

5.1 FEASIBILITY STUDY

Based on the results obtained in Table 3.2, the empirical regression model and Zero-Failure model is the best suited in our research context. In our preliminary feasibility study, we used the Zero-Failure model with STREW Version 1.1, as will be discussed below. We ran a feasibility study on 13 vending machine programs written in Java by junior/senior-level undergraduates in a software engineering course at NCSU in Fall 2002. The programs were, on average, 789 lines of code (LOC). With these programs, we analyzed an initial set of metrics defined in the STREW-J Version 1.1 [70]:

- *number of test cases/source lines of code (R1);*
- *number of test cases/number of requirements (R2);*
- *test lines of code/source lines of code (R3);*

- *number of asserts/source lines of code (R4)*; and on
- *statement coverage (C).*

We estimated reliability via the Zero-Failure model described by Hamlet et al. [38] and others [28, 59]. This model was chosen because the programmers worked until all the automated test cases passed (e.g. no failures are demonstrated by the test suite). However, it is possible and even likely that their test suites did not constitute a thorough test effort. To use the No-Failure model, we must identify a meaningful long term failure rate denoted by Θ and that N random tests have established an upper confidence bound $(1-\alpha)$ that Θ is below some level θ [38]. The relationship between these factors is given by $1- (1-\theta)^N <= \alpha$ [89]. This model is similar to the analysis performed by Frankl et al. [34] on operational testing given by $1-(1-q)^T$, where q is the failure rate for operational testing and T is the number of tests.

We ran a multiple regression analysis with the metrics collected from the program and the reliability estimate. The result of applying an ordinary least squares regression provided evidence ($F=4.325$, $p=0.041$) of a linear association between the reliability estimate and the metrics. The regression equation to predict the reliability is given by Equation 5.1.

$$Reliability = 0.669 + 1.586*R1 + 0.0513*R2 - 0.0290*R3 + 0.192*R4 - 0.0774*C \qquad (5.1)$$

Correlations between the metrics and the reliability estimate for various values of θ are given in Table 5.1. The given p-values for tests of zero correlation provide evidence of positive linear dependence of reliability on the metrics R1 and R2. Ratios R3 and R4 are not statistically significant with the reliability estimate. This lack of association may be because the coefficients were based upon programs from a small number (N=13) of different programmers and the value of R3 and R4 relies on personal programming style. The relationship between the code coverage and the estimated reliability was also not statistically significant. The lack of relationship between code coverage and the estimated reliability might be because the students were specifically assigned to create test suites with 100% statement coverage which may have created an artificial situation. While only R1

(*number of test cases/source lines of code)* and R2 (*number of test cases/number of requirements)* demonstrated a statistically significant relationship with the reliability estimate, we continued to analyze the potential of the other metrics to refine the STREW metric suite.

Table 5.1: Pearson correlation and statistical significance results between the ratios and the reliability estimate

θ	R1	R2	R3	R4	Coverage
0.01	0.629, p=.021	0.992, p=.000	0.302, p=.316	0.469, p=.106	0.118, p=.702
0.05	0.703, p=.007	0.814, p=.001	0.393, p=.184	0.541, p=.056	0.039, p=.900
0.10	0.616, p=.025	0.584, p=.036	0.414, p=.159	0.502, p=.081	-0.071, p=.817

In applying the Zero-Failure model, the meaningful long term failure rate denoted by Θ and N random tests to establish an upper confidence bound on the reliability estimate. Θ is user defined and governs the reliability estimate. In the absence of failures, it is not possible to determine Θ. Hence we use the empirical model building approach through the rest of our investigations.

5.2 MODEL BUILDING

This section describes the model-building strategies that were used for predicting post-release field quality. A number of techniques have been used for the analysis of software quality in terms of reliability, failure density, and fault-proneness. Multiple linear regression (MLR) analysis was used to model the relationship between quality and software metrics [51, 63]. In our work, the dependent variable that is to be measured is the TRs/KLOC and the independent variables are the STREW metrics.

The regression equation is of the form:

- $Y = c + a_1X1+a_2X2+.....+a_nXn$, where Y is the estimated (dependent) variable, a_1, a_2 …., a_n are regression coefficients and X1, X2 …., Xn are the known variables (independent).

One difficulty associated with MLR is multicollinearity among the metrics that can lead to inflated variance in the estimation of post-release field quality. One approach that has been used to

account for multicollinearity is Principal Component Analysis (PCA) [33]. With PCA, a smaller number of uncorrelated linear combinations of metrics that account for as much sample variance as possible are selected for use in regression (linear or logistic). PCA can be used to downweigh metrics that are highly correlated with other metrics, thus simplifying data collection with minimal impact on the accuracy of the information provided. A multivariate logistic regression equation [25] can be built to model data using the principal components as the independent or predictor variables.

As will be discussed, in our case studies the nine STREW metrics demonstrated multicollinearity. The Kaiser-Meyer-Olkin (KMO) [47] test of sampling adequacy was used to identify multicollinearity. For PCA to be applicable, the KMO measure of sampling adequacy should be greater than 0.6 [11]. The KMO measure of sampling adequacy is a test of the amount of variance within the data that can be explained by the individual measures (e.g. the STREW metrics). PCA generates a linear transformation of a set of correlated attributes, such that the transformed variables are independent, i.e. the transformed variables do not have any inter-correlations between them that lead to an inflated variance in the estimation of the post-release field quality. The PCA analysis showing the scree plots of the principal components and the respective components plots are shown for the academic and open source projects are presented in Appendix C. Scree plots explain the identification of the principal components from the individual metrics.

In our study, the KMO results indicated the applicability of using the principal components produced by PCA, rather than the nine individual metrics, in an MLR equation. Throughout this dissertation, we present the regression analysis using all the STREW metrics and the principal components obtained from the STREW metrics. The primary purpose of presenting both models is for comparison between a basic approach (all STREW metrics but suffering from multicollinearity) and a more precise approach (PCA after accounting for multicollinearity). This type of comparative analysis also serves to indicate that results obtained by PCA are more accurate for our purposes, and all the nine STREW metrics are required to construct the principal components, indicating the utility of the nine individual STREW metrics.

The generalized regression equation and associated confidence are formulated as follows:

- $Y = c + a_1PC1+a_2PC2+…..+a_nPCn$, where a_1, a_2 …., a_n are regression coefficients and PC1, PC2 …., PCn are the produced principal components.
- Further, the estimated value of the TRs/KLOC (Y_{new}) is calculated with an associated confidence bound [53] given by Equation 5.2 where *n* is the number of observations, $t_{(\alpha/2, n)}$ is the standard t-table value with n degrees of freedom at a significance level of $\alpha = 0.05$.

$$Confidence\ bounds = Y_{new} \pm t_{(\alpha/2, n)} * Mean\ Standard\ Error \quad (5.2)$$

Further, the model building strategies have the following associated factors:

- The coefficient of determination, R^2, is the ratio of the regression sum of squares to the total sum of squares. The value of the ratio can be between 0 and 1, with larger values indicating more variability explained by the model and less unexplained variation.
- The F-ratio is used to test the hypothesis that all regression coefficients are zero at statistically significant levels.
- Where parametric testing is appropriate, a significance level of $\alpha = 0.05$ was adopted for statistical inference. For example, all interval estimates are reported using confidence level 95%.

5.3 RESULTS EVALUATION TECHNIQUES

The evaluation of the TRs/KLOC estimates with the actual values is performed using two methods explained below:

- *Average Absolute Error (AAE) and Average Relative Error (ARE).* AAE and ARE are used to measure the accuracy of prediction of the estimated TRs/KLOC with numerical quantification (i.e. numerical accuracy) [52]. The smaller the values of AAE and ARE, the better are the predictions. The equations for AAE and ARE are provided as Equations 5.3 and 5.4 where *n* is the number of observations.

$$AAE = \frac{1}{n}\sum_{i=1}^{n} \quad | \textit{Estimated Value} - \textit{Actual Value} | \qquad (5.3)$$

$$ARE = \frac{1}{n}\sum_{i=1}^{n} \quad (| \textit{Estimated Value} - \textit{Actual Value} |) / \textit{Actual Value} \qquad (5.4)$$

- *Correlation between the actual and estimated TRs/KLOC.* This correlation is used to quantify the sensitivity of prediction. The two correlation techniques utilized are (1) the Spearman rank correlation which is a commonly-used robust rank correlation technique [33] because it can be applied when the association between elements is non-linear and (2) the Pearson bivariate correlation that requires the data to be distributed normally and the association between elements to be linear. A positive relationship between the actual and estimated values is desired in terms of the more robust Spearman rank correlation

A positive correlation between the actual and estimated TRs/KLOC would indicate that with an increase in estimated TRs/KLOC, there is an increase in actual TRs/KLOC. A negative correlation would indicate that with an increase in estimated TRs/KLOC, there is a decrease in actual TRs/KLOC (or vice versa).

5.4 ACADEMIC CASE STUDY

This sub-section describes a controlled study performed with junior/senior-level students at North Carolina State University (NCSU) to investigate the efficacy of the STREW metric suite elements to estimate TRs/KLOC.

5.4.1. DESCRIPTION

The students worked on a project that involved development of an Eclipse[6] plug-in to collect software metrics. The project was six weeks in duration and used Java as the programming language. The JUnit testing framework was used for unit testing; students were required to have 80% statement

[6] http://eclipse.org/

coverage. A total of 22 projects were submitted, and each group had four to five students. Figure 5.2 shows the size LOC_{source} of the projects that were developed. The projects were between 617 LOC_{source} and 3,631 LOC_{source}. On average, the ratio of LOC_{test} to LOC_{source} was 0.35. Each project was evaluated by 45 independent test cases. Actual TRs/KLOC was estimated by test case failures (or) TRs per KLOC because the student projects were not released to customers. If a 2000-line project had ten failures (out of the 45 tests) the TRs/KLOC = (10/2000)*1000 = 5 failures/KLOC.

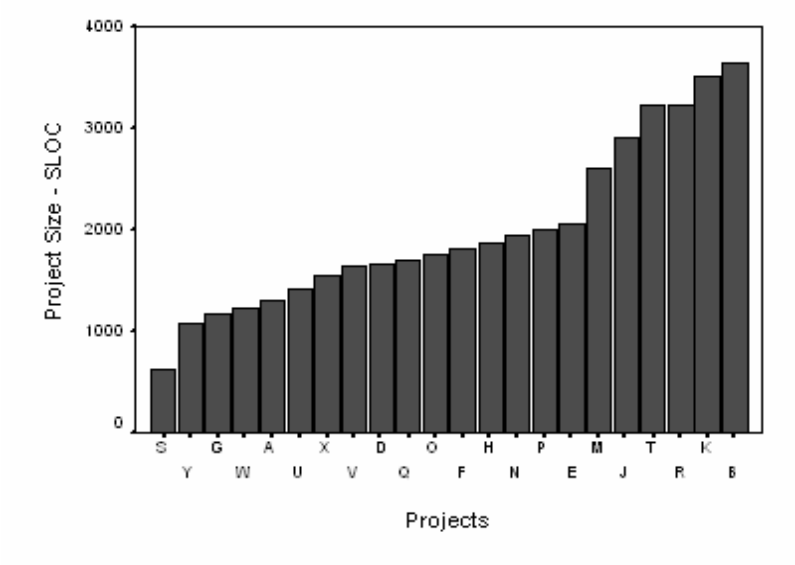


Figure 5.2: Academic projects size

The academic study is a closed structured environment that might represent ideal experimental conditions but may not reflect industrial settings. As a result, an external validity issue arises because the programs were written by students, and the projects were small relative to industry applications. This concern is mitigated to some degree because the students were advanced undergraduates (junior/senior).

5.4.2 MODEL BUILDING

The initial regression model to estimate post-release field quality was built using all the STREW metrics and the principal components obtained from the STREW metrics as the independent variables and the TRs/KLOC as the dependent variable for all 22 projects. The normal probability plots for the

built model along with the R^2 value and associated statistical significance is shown in Figure 5.3. This model serves to indicate the overall model fit obtained using all the data points.

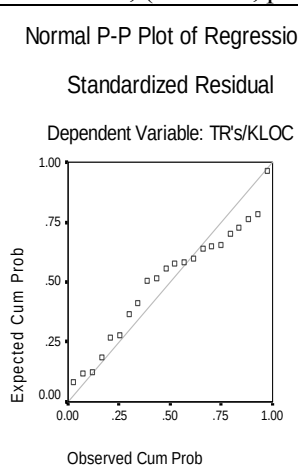
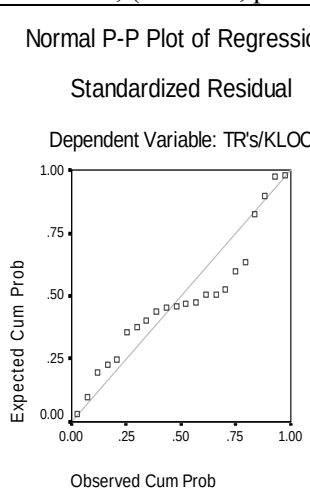
All STREW metrics	PCA
R^2= 0.807, (F=5.561, p=0.004)	R^2=0.512, (F=6.284, p=0.004)
	

Figure 5.3: Normal probability plot for model fitting

5.4.3 RANDOM DATA SPLITTING

To evaluate the efficacy of the STREW metric suite to predict TRs/KLOC, we use a random splitting approach. We randomly selected two-thirds (N=15) of the samples to build a prediction model and the remaining one-third (N=7) to evaluate the built model. We obtained the results shown in Table 5.2.

Table 5.2: Prediction evaluation results

	STREW metrics	PCA
Model Fit	R^2= 0.905, (F=5.274, p=0.041)	R^2= 0.598, (F=5.463, p=0.015)
Prediction Evaluation (7 projects)	AAE=15.29, ARE=4.19 Pearson correlation = 0.390 (p=0.377) Spearman correlation = 0.464 (p=0.294)	AAE=4.47, ARE=1.27 Pearson correlation = 0.328 (p=0.473) Spearman correlation = 0.464 (p=0.294)

The AAE and ARE results in Table 5.2 indicate that the model produces an estimate that is indicative of the true TRs/KLOC. The actual values of TRs/KLOC range from 1.43 TRs/KLOC to

35.5 TRs/KLOC indicating a wide spectrum of quality in terms of TRs for the academic projects. This range of TRs/KLOC of the projects is shown by the histogram in Figure 5.4. The ARE and AAE values are less than the standard deviation of the TRs/KLOC (standard deviation is 7.738 TRs/KLOC). In spite of the wide spectrum of the TRs/KLOC, the ARE and AAE values indicate the feasibility of using the STREW metric suite compared to the actual values of the TRs/KLOC. The Pearson and Spearman rank correlation is not statistically significant. The reason for the correlation coefficients to be positive but not statistically significant might be due to the small sample size (seven projects). The results of this feasibility study motivated further investigation using open source and industrial projects to investigate the utility of the STREW metric suite to predict TRs/KLOC on larger scale projects.

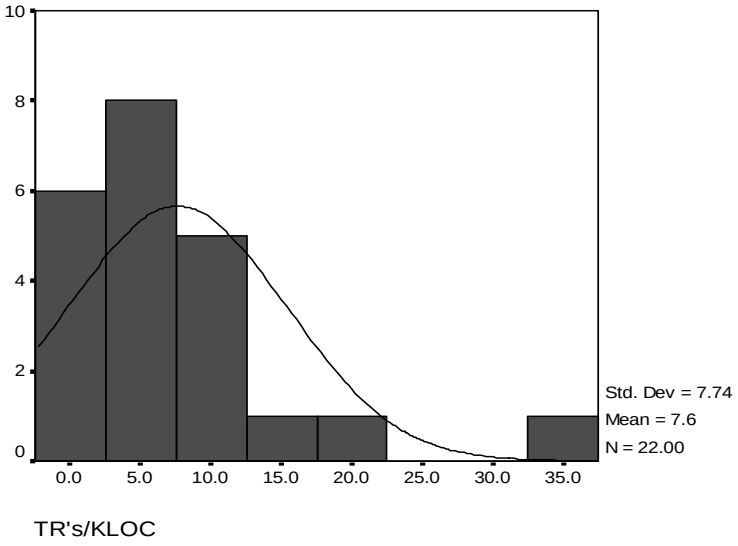


Figure 5.4: Histogram of TRs/KLOC in academic projects

5.5 OPEN SOURCE CASE STUDIES

Open source projects are convenient to perform case studies because source code, test code, and defect logs are openly available for use. Additionally, open source projects are more representative of industrial projects than the academic projects due to their size and scope. We selected 27 open source projects to apply the STREW metric suite.

5.5.1 DESCRIPTION

Twenty-seven open source projects that were developed in Java were selected from Sourceforge (http://sourceforge.net). The following criterion was used to select the projects from Sourceforge.

- *software development tools*. All of the chosen projects are software development tools, i.e. tools that are used to build and test software and to detect defects in software systems.
- *download ranking of 85% or higher.* In Sourceforge, the projects are all ranked based on their downloads on a percentile scale from 0-100%. For example, a ranking of 85% means that a product is in the top 85% of quantity of downloads. We chose this criterion because we reasoned that a comparative group of projects with similarly high download rates would be more likely to have a similar usage frequency by customers that would ultimately reflect the post-release field quality.
- *automated unit testing*. The projects needed to have JUnit automated tests.
- *defect logs available*. The defect log needed to be available for identifying TRs with the date of the TR reported.
- *active fixing of TRs*. The TR fixing rate is used to indicate the system is still in use. The time between the reporting of a TR and the developer fixing it serves as a measure of this factor. Projects that had open TRs that were not assigned to anyone over a period of three months were not considered.
- Sourceforge development stage of 4 or higher. This denotes the development stage of the project (1-6) where 1 is a planning stage and 6 is a mature phase. We chose a cut-off of 4 which indicates the project is at least a successful beta release. This criteria indicates that the projects that are at a similar stage of development and are not projects too early in the development lifecycle.

Figure 5.5 shows the names of the projects and their sizes in LOC_{source}. On average, the ratio of LOC_{test} to LOC_{source} was 0.37. The projects range from around 2.5 $KLOC_{source}$ to 80 $KLOC_{source}$. The

TRs are normally distributed with a range from 0.20 to 6.9 TRs/KLOC (Mean = 1.42). The defect logs were screened for duplicate TRs to obtain an accurate measure of TRs/KLOC. Duplicate TRs were removed before analysis.

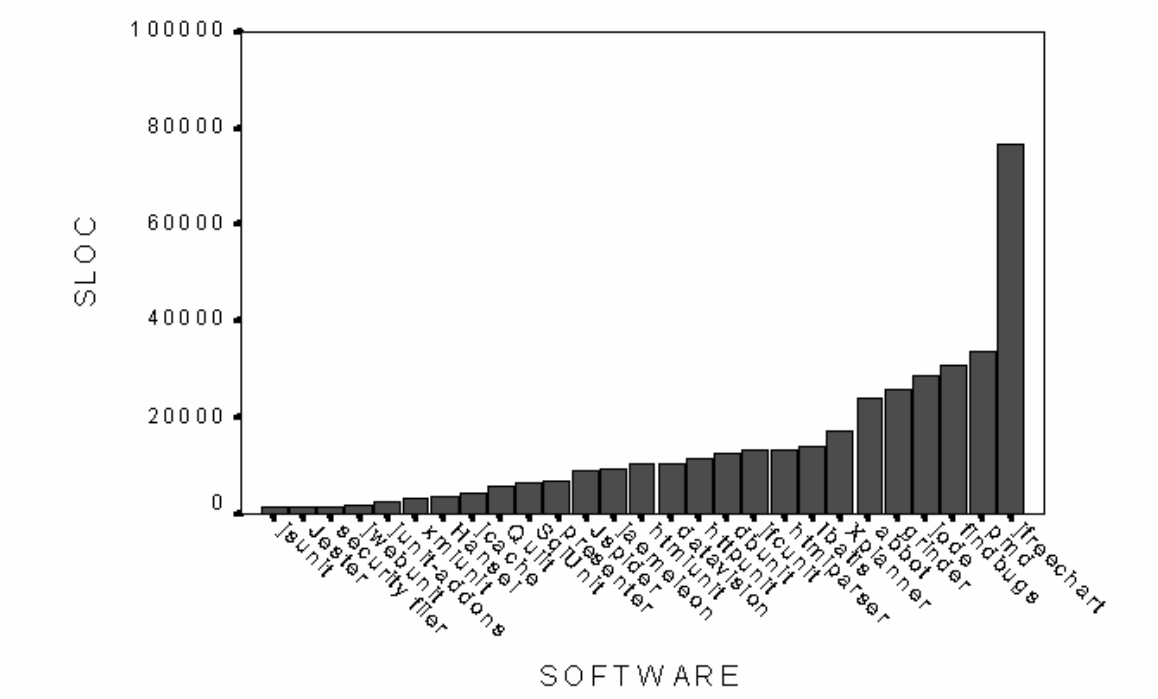


Figure 5.5: Open source project sizes

There is a validity issue with respect to the actual usage of open source software systems. The commonly-available usage metric for open source software is the download activity ratio. For example, it might be possible that Project 1 and Project 2 could have both been downloaded by 100 users, but Project 1 might have been actually used by only ten users while Project 2 by 90 users. We have no visibility to this actual usage. Also, we are dependent on the customer-reports for measuring the TRs/KLOC. However, some customers might not report the TRs in the open source environment. We assume that the TRs are representative of the operational profile [64] of the software system. If the different components of the software systems were not used equally the estimation of TRs can lead to an inflated value (i.e. a different set of customers might use the product for a different purpose and might exercise the product in different ways – and this different usage would surface a different post-release product quality). The issues of the generalizability of the actual operation profile is

negated to some extent by the uniform testing efforts of the projects, comparable TRs/KLOC across all the projects and all the projects belonging to one particular domain of software systems. A more detailed retrospective analysis is performed in Chapter 7.

5.5.2 MODEL BUILDING

Figure 5.6 shows the complete MLR model for all the 27 open source projects built using both the STREW metrics and the principal components of the STREW metrics as the independent variable and the TRs/KLOC as the dependent variable. Similar to the academic case studies, the models built is statistically significant in its ability to explain the variance in the dependent variable.

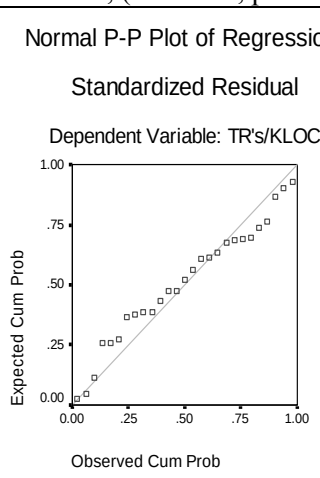
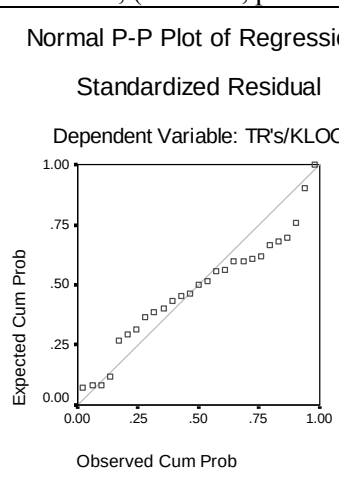
All STREW metrics	**PCA**
R^2= 0.755, (F=5. 814, p=0.001)	R^2=0.428, (F=8.993, p=0.004)
	

Figure 5.6: Model fitting results for PCA

The Kaiser-Meyer-Olkin (KMO) measure of sampling adequacy is 0.764 indicating the efficacy of the applicability of PCA. Principal component analyses of the nine STREW metrics yields two principal components, the linear transformation coefficient given by the component matrix in Table 5.3.

Using MLR on the two principal components calculated using Table 5.3, we obtain Equation 5.5.

$$TRs/KLOC = 1.424 + 0.852 * PC1 - 0.132*PC2 \qquad (5.5)$$

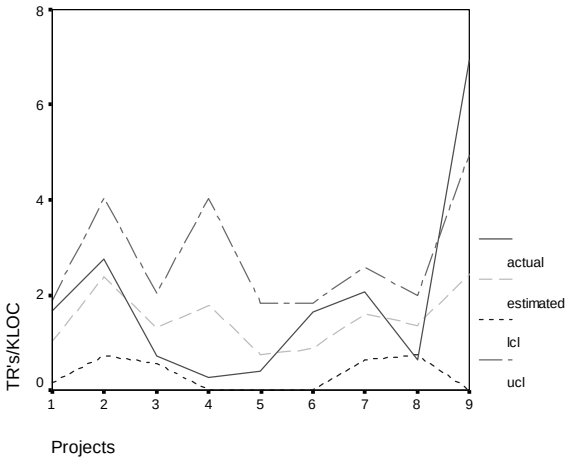
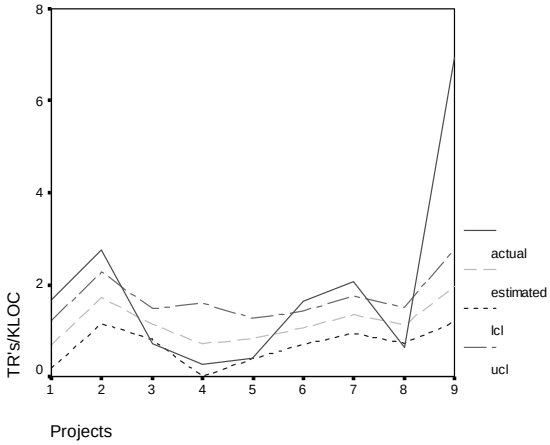
where PC1 and PC2 are the transformed principal components produced from the nine STREW metrics.

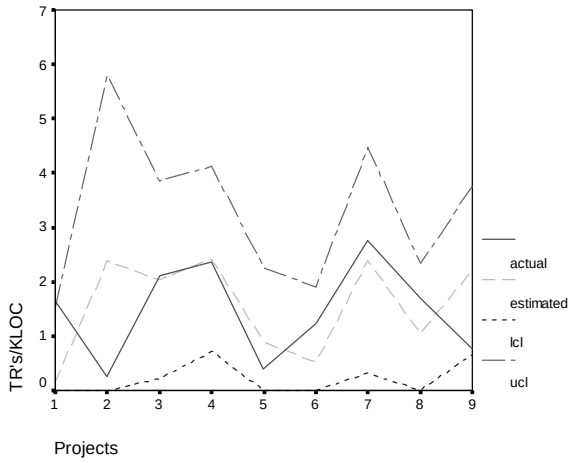
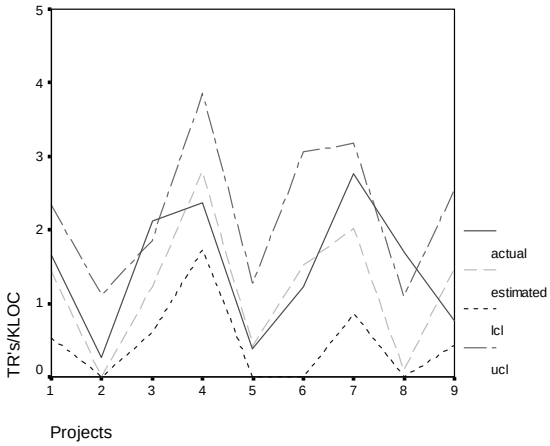
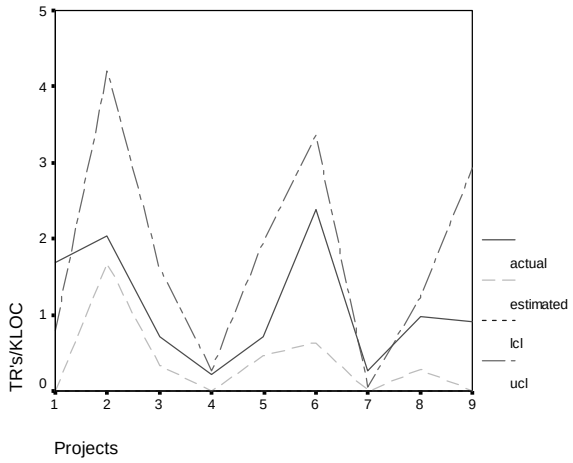
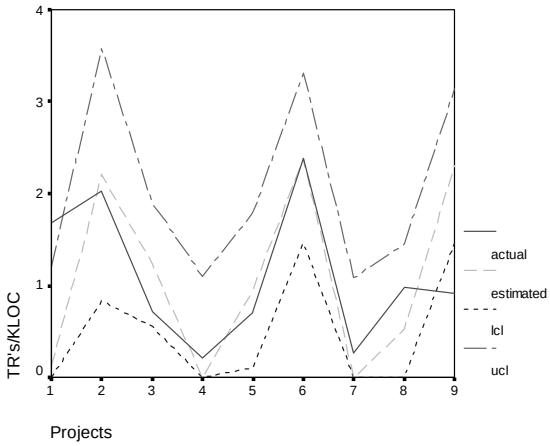
Table 5.3: Component matrix for PCA transformation

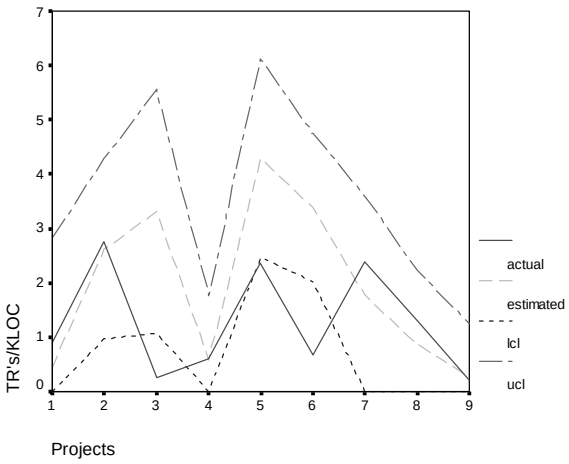
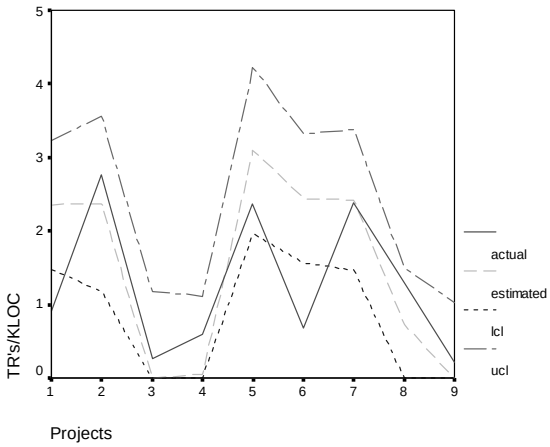
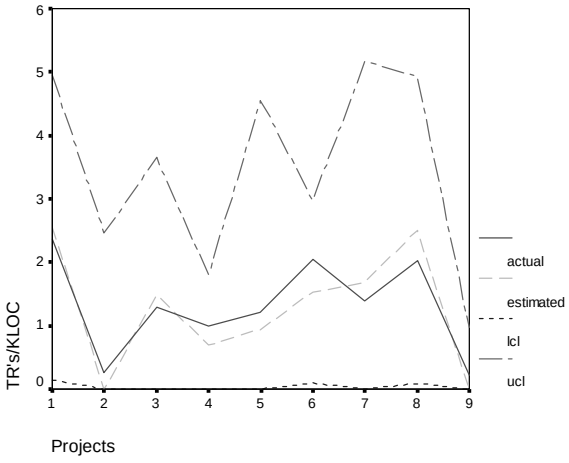
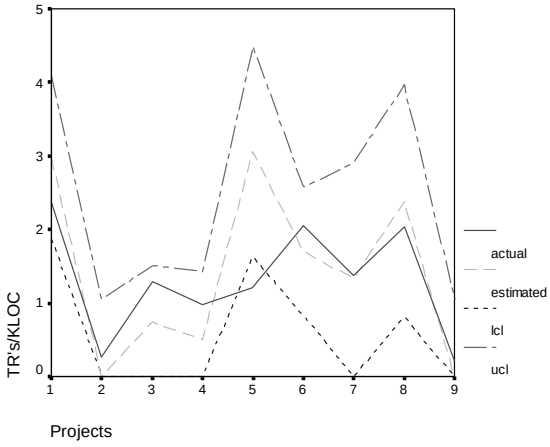
	Component	
	PC1	PC2
SM1	.793	.284
SM2	.795	-.281
SM3	-.270	.701
SM4	.935	-6.280E-03
SM5	.159	-.745
SM6	.877	.102
SM7	.699	.457
SM8	.864	4.126E-02
SM9	-.569	.233

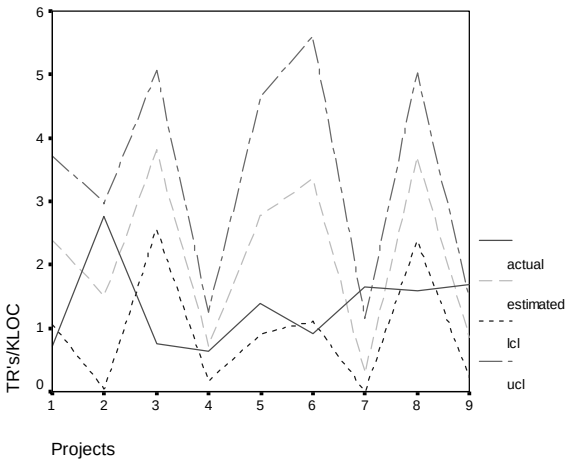
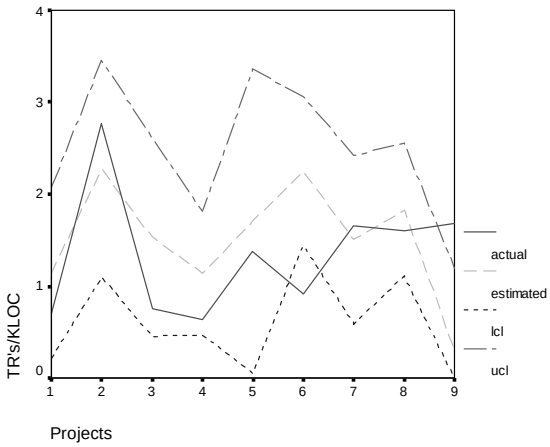
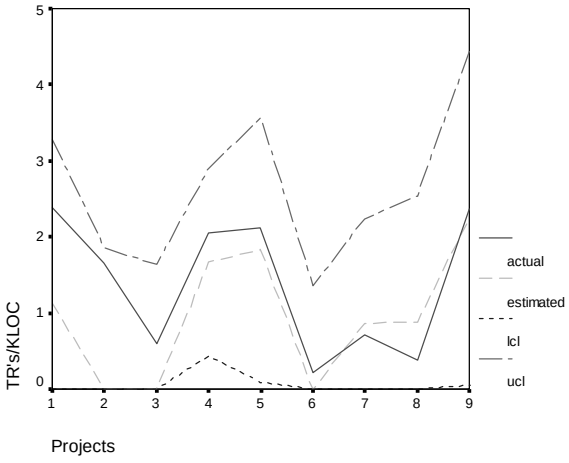
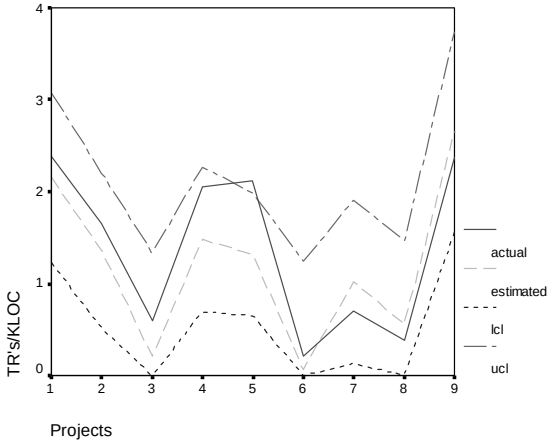
5.5.3 RANDOM DATA SPLITTING

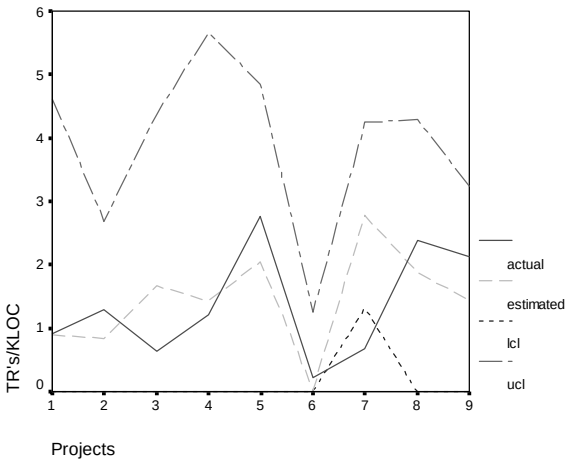
For the random data-splitting, we use two thirds (N=18) of the 27 projects to build the prediction model and the remaining one-third (N=9) to evaluate the fit of the model. We repeated the random split nine times to verify data consistency, i.e. to check if the results of our analysis were not a one-time occurrence. Figure 5.7 shows the model building and model evaluation performed using MLR on all the STREW metrics and the principal components. The results for each data split present the R^2 and F test results for each model and the prediction results evaluated using AAE, ARE, and correlation analysis. The graphs indicate the actual and the estimated TRs/KLOC along with the regression predicted upper confidence level (ucl) and lower confidence level (lcl) [53].

All STREW metrics– Random split 1	**PCA – Random split 1**
R^2= 0.748, (2.638, p=0.093)	R^2= 0.373 (4.467, (p=0.030))
	
AAE = 1.1077, ARE = 0.4000	AAE = 1.1230, ARE = 0.3276

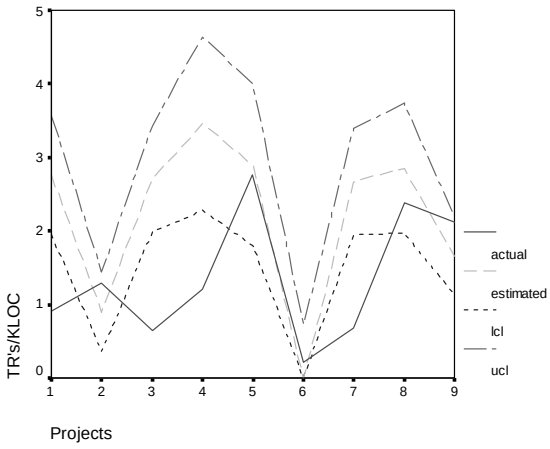
Pearson correlation = 0.672 (p=0.047) Spearman correlation = 0.500 (p=0.170)	Pearson correlation = 0.834 (p=0.005) Spearman correlation = 0.700 (p=0.036)
MLR – Random split 2	**PCA – Random split 2**
R^2= 0.877, (3.838, p=0.006)	R^2= 0.442 (5.938, p=0.013)
	
AAE =0.8222, ARE = 0.4579 Pearson correlation = 0.207 (p=0.592) Spearman correlation = 0.367 (p=0.332)	AAE = 0.6400, ARE = 0.2796 Pearson correlation = 0.694 (p=0.038) Spearman correlation = 0.617 (p=0.077)
MLR – Random split 3	**PCA – Random split 3**
R^2= 0.865, (5.706, p=0.011)	R^2= 0.450, (6.136, p=0.011)
	
AAE =1.4019, ARE = 0.8309 Pearson correlation = 0.640 (p=0.063) Spearman correlation = 0.683 (p=0.042)	AAE = 0.5600, ARE = 0.2900 Pearson correlation = 0.641 (p=0.063) Spearman correlation = 0.667 (p=0.050)
MLR – Random split 4	**PCA – Random split 4**
R^2= 0.921, (10.336, p=0.002)	R^2= 0.450, (6.138, p=0.011)

	
AAE =1.0455, ARE = 0.5922 Pearson correlation = 0.369 (p=0.329) Spearman correlation = 0.367 (p=0.332)	AAE = 0.7322, ARE = 0.4034 Pearson correlation = 0.732 (p=0.025) Spearman correlation = 0.700 (p=0.036)
MLR – Random split 5	**PCA – Random split 5**
R^2= 0.785, (3.246, p=0.056)	R^2= 0.446, (6.038, p=0.012)
	
AAE =0.7055, ARE = 0.4614 Pearson correlation = 0.885 (p=0.002) Spearman correlation = 0.933 (p<0.0005)	AAE = 0.5533, ARE = 0.2599 Pearson correlation = 0.799 (p=0.010) Spearman correlation = 0.717 (p=0.030)
MLR – Random split 6	**PCA – Random split 6**
R^2= 0.935 (12.724, p=0.001)	R^2= 0.491, (7.229, p=0.006)

	
AAE =1.5788, ARE = 0.7331 Pearson correlation = -0.290 (p=0.449) Spearman correlation = -0.217 (p=0.576)	AAE = 0.6200, ARE = 0.2900 Pearson correlation = 0.279 (p=0.467) Spearman correlation = 0.317 (p=0.406)
MLR – Random split 7	**PCA – Random split 7**
R^2= 0.792, (3.376, p=0.050)	R^2= 0.367, (4.355, p=0.032)
	
AAE =0.6233, ARE = 0.2841 Pearson correlation = 0.721 (p=0.028) Spearman correlation = 0.750 (p=0.020)	AAE = 0.3555, ARE = 0.1525 Pearson correlation = 0.905 (p=0.001) Spearman correlation = 0.917 (p=0.001)
MLR – Random split 8	**PCA – Random split 8**
R^2= 0.827, (4.247, p=0.027)	R^2= 0.642, (13.463, p<0.0005)



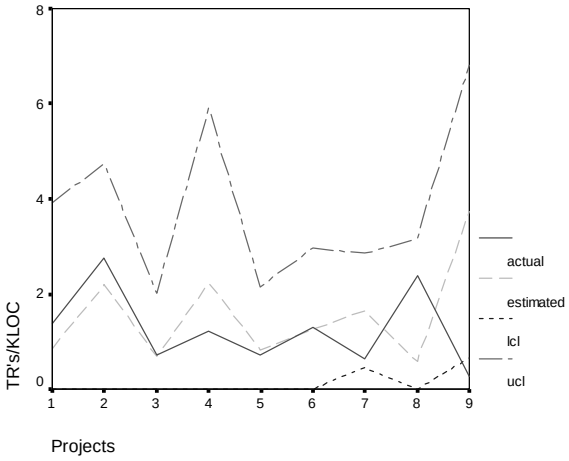
AAE =0.6533, ARE = 0.3213
Pearson correlation = 0.384 (p=0.307)
Spearman correlation = 0.367 (p=0.332)



AAE = 1.1011, ARE = 0.5740
Pearson correlation = 0.351 (p=0.354)
Spearman correlation = 0.467 (p=0.205)

MLR – Random split 9

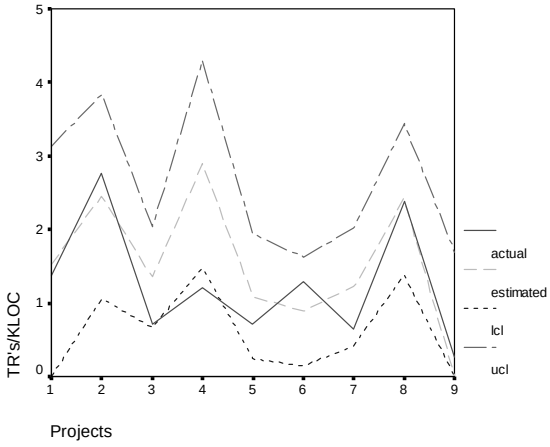
R^2= 0.838, (4.597, p=0.021)



AAE =0.9533, ARE = 0.5373
Pearson correlation = -0.239 (p=0.536)
Spearman correlation = -0.300 (p=0.433)

PCA – Random split 9

R^2= 0.435, (5.765, p=0.014)



AAE = 0.4922, ARE = 0.2493
Pearson correlation = 0.721 (p=0.028)
Spearman correlation = 0.667 (p=0.050)

Figure 5.7: Random data split – model building and evaluation results

To summarize, we present the results of our evaluation using the models built using all the STREW metrics and PCA in Table 5.4. We can assess the efficacy of the prediction model built using 18 randomly-chosen projects. The ARE values reflect the relative error in terms of the absolute magnitude of the TRs/KLOC.

Table 5.4: AAE and ARE values

Random Split	All STREW metrics		PCA	
	AAE	**ARE**	**AAE**	**ARE**
1.	1.1077	0.4000	1.1230	0.3276
2.	0.8222	0.4579	0.6400	0.2796
3.	1.4019	0.8309	0.5600	0.2900
4.	1.0455	0.5922	0.7322	0.4034
5.	0.7055	0.4614	0.5533	0.2599
6.	1.5788	0.7331	0.6200	0.2900
7.	0.6233	0.2841	0.3555	0.1525
8.	0.6533	0.3213	1.1011	0.574
9.	0.9533	0.5373	0.4922	0.2493

The overall standard deviation of the TRs/KLOC is 1.318 TRs/KLOC. The AAE in all the nine random cases (using PCA after eliminating multicollinearity) is smaller than the standard deviation indicating that the efficacy of the prediction results. Table 5.5 indicates the correlation coefficient (Pearson and Spearman) results between the actual and estimated post-release TRs/KLOC. The correlation measure serves to indicate the sensitivity of the predicted post-release TRs/KLOC. Of nine random samples in the PCA, seven are statistically significant (between the estimated and actual post-release quality), indicating the efficacy of our approach for the open source projects case study (shown in bold in Table 5.5). This indicates the sensitivity of prediction between the actual and estimated values.

Table 5.5: Sensitivity evaluation for PCA models

Random Split	All STREW metrics		PCA	
	Pearson (p value)	Spearman (p value)	Pearson (p value)	Spearman (p value)
1.	0.672 (p=0.047)	0.500 (p=0.170)	0.834 (p=0.005)	**0.700** (p=0.036)
2.	0.207 (p=0.592)	0.367 (p=0.332)	0.694 (p=0.038)	**0.617** (p=0.077)
3.	0.640 (p=0.063)	0.683 (p=0.042)	0.641 (p=0.063)	**0.667** (p=0.050)
4.	0.369 (p=0.329)	0.367 (p=0.332)	0.732 (p=0.025)	**0.700** (p=0.036)
5.	0.885 (p=0.002)	0.933 (p<0.0005)	0.799 (p=0.010)	**0.717** (p=0.030)
6.	-0.290 (p=0.449)	-0.217 (p=0.576)	0.279 (p=0.467)	0.317 (p=0.406)
7.	0.721 (p=0.028)	0.750 (p=0.020)	0.905 (p=0.001)	**0.917** (p=0.001)
8.	0.384 (p=0.307)	0.367 (p=0.332)	0.351 (p=0.354)	0.467 (p=0.205)
9.	-0.239 (p=0.536)	-0.300 (p=0.433)	0.721 (p=0.028)	**0.667** (p=0.050)

5.6 INDUSTRIAL CASE STUDY

In this section, we describe the industrial case study that was performed to investigate the efficacy of the STREW metric suite to assess TRs/KLOC for three commercial large scale software systems.

5.6.1 DESCRIPTION

Our industrial case study involved three software systems (five versions) at a company in the United States. To protect proprietary information, we keep the name and nature of the projects anonymous. These projects were critical in nature because failures could lead to loss of essential funds for the company. The project sizes that were used for analysis are shown in Table 5.6.

Table 5.6: Industrial project sizes

Project	Size
Project 1A	190 KLOC
Project 1B	193 KLOC
Project 2A	504 KLOC
Project 2B	487 KLOC
Project 3	13 KLOC

The development language used was Java, and the JUnit testing framework was used for unit and acceptance testing. Some other descriptive project characteristics measures are presented in Table 5.7.

Table 5.7: Project characteristics

Factor	Team value
Team size	Ranged from 6 to 16 developers.
Team education level	62% had a bachelors degree and 34% had a masters degree or higher.
Experience level of team	57% had more than five years of experience
Domain expertise	Medium-High.
Language expertise	Medium-High.

5.6.2 POST-RELEASE FIELD QUALITY PREDICTION

In the industrial environment, TRs found by customers are reported back to the organization. These TRs are then mapped back to the appropriate software systems. We use the STREW metrics of the open source projects to predict the TRs/KLOC of the industrial software systems. Figure 5.8 indicates the prediction plots obtained using all the STREW metrics. The axes are removed to protect proprietary information.

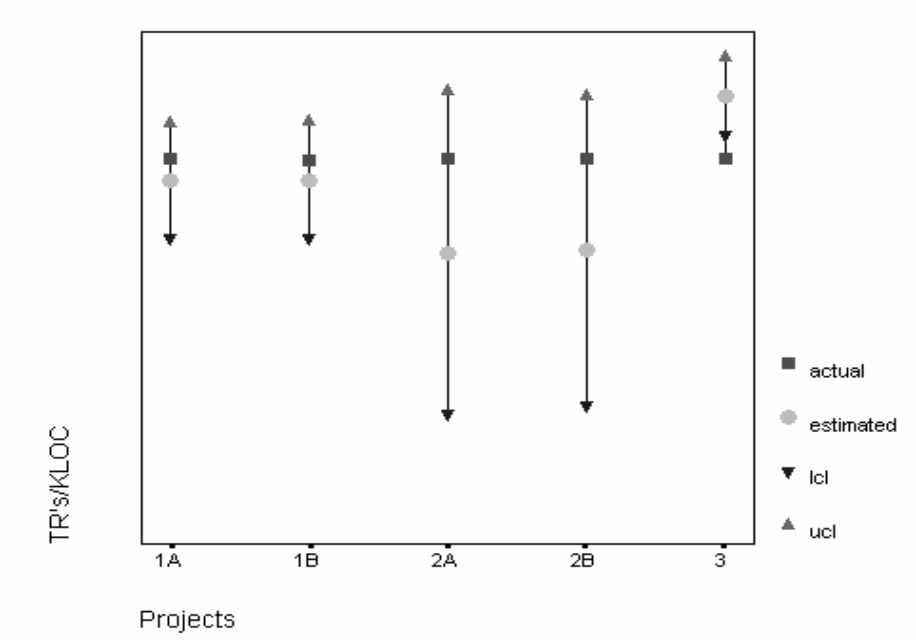


Figure 5.8: Prediction plots with all the STREW metrics

Similarly Figure 5.9 indicates the prediction plots obtained using PCA on the STREW metrics from Equation 5.5. Figure 5.9 indicates that the prediction obtained by using the PCA after

accounting for multicollinearity which is more indicative than the prediction obtained by MLR using the nine STREW metrics.

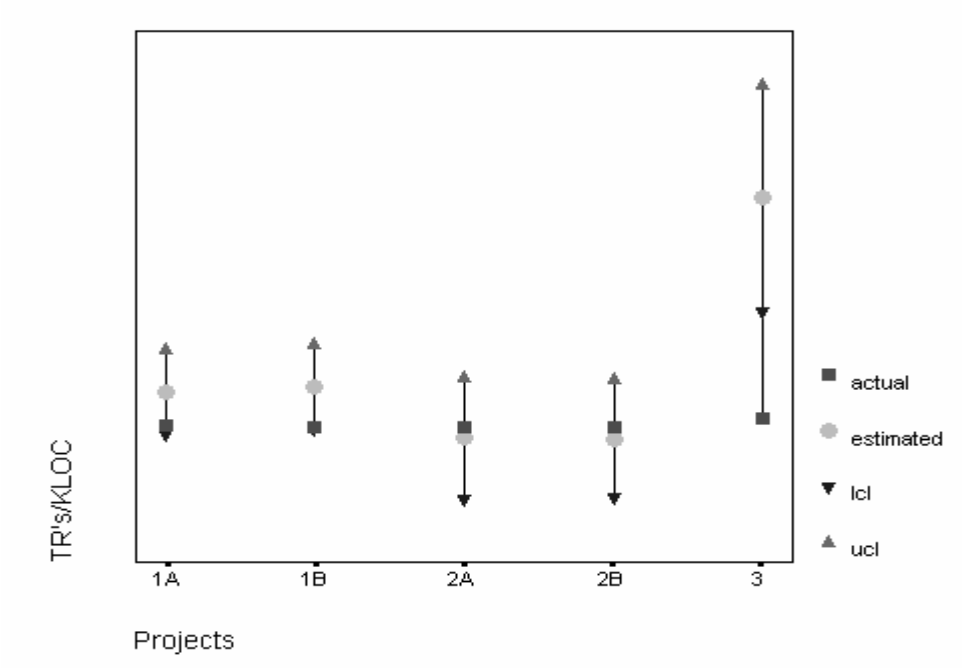


Figure 5.9: Prediction plots with PCA

Due to the critical nature of the product, the post-release TRs/KLOC are low. Further, Figure 5.9 indicates the predicted value closely overlaps the actual TRs/KLOC and bounds for Projects 1 and 2. Project 3 may not a comparable project because it is smaller than the other projects and was to form the core components of the organization's future software products. Therefore, Project 3 was particularly well tested, comparably more than the other software systems used to build or evaluate the prediction. On observing the nine STREW metric values for Project 3 we notice that metrics SM1, SM2 for Project 3 at least one order greater than all the projects (open source included). Metrics SM5, SM6, SM7, and SM8 for Project 3 are at least twice of SM5, SM6, SM7 and SM8 of all the other projects. SM5-SM8 are the cyclomatic and O-O metrics in the STREW metric suite indicating that project 3 had a high testing effort (and complexity) compared to all the other projects.

Note that in all cases, the upper confidence bounds are larger than the actual value. Organizations can take a conservative approach and use the upper bounds of TRs/KLOC to be the actual estimated

TRs/KLOC to drive the overall quality higher. Even though the sample size is small (only 5 projects), we present the correlation results of the actual and estimated TRs/KLOC indicated in Figure 7. The Pearson correlation coefficient = 0.962 ($p = 0.009$) and Spearman correlation coefficient = 0.500 ($p = 0.391$). This indicates the efficacy of the sensitivity of prediction of the TRs/KLOC but is limited by the small sample size of the available projects. (For comparative purposes for the MLR model built using all the STREW metrics the Pearson correlation coefficient = 0.873 ($p = 0.054$) and Spearman correlation coefficient = 0.700 ($p = 0.188$)).

CHAPTER 6

TEST QUALITY FEEDBACK

Providing test quality feedback allows the developers to identify areas that could benefit from more testing. Color coding [16, 42, 75] aids developers in quickly understanding if a metric is within acceptable limits. Our work is motivated by prior studies at IBM[7] [16], Nortel Networks[8] [42] that use color-coding to provide feedback on metric values based on standards (predefined or calculated).

The Enhanced Measurement for Early Risk Assessment of Latent Defects (Emerald) system at Nortel Networks combined software measurements, quality models and delivery of results to provide in process feedback to developers to improve telecommunications software reliability. The Emerald system was found to improve architectural integrity; establish design guidelines and limits; focus efforts on modules more likely to have faults; target the test effort effectively; identify patch-prone modules early; incorporate design strategies to account for the risk associated with defective patches; and help obtain a better understanding of field problems [42]. Emerald provides color-coded feedback using nine categories: green, yellow and seven shades of red using non-OO metrics related to software system link volume, testability, decision complexity and structuredness (a more detailed explanation is available in [42]). However, STREW differs from Emerald as it leverages the automatic testing effort performed to estimate the post-release field quality based on test and source code metrics and it developed for use by OO-languages.

[7] www.ibm.com
[8] www.nortelnetworks.com

Similarly at IBM, immediate feedback on the complexity of Smalltalk methods, based on the source code allows developers to modify their code to more desirable characteristics in terms of the code complexity [16]. Color coded feedback is presented in three levels, red, yellow and green using Smalltalk complexity metrics like number of blocks, number of temporary variables and arguments, number of parameterized expressions etc. STREW similar to the work done at Nortel Networks and IBM, provides feedback in three ranges red, orange and green based on in-process metrics obtained from both the source and test code.

6.1 BUILDING TEST QUALITY FEEDBACK STANDARDS

The color that is displayed is determined by the results of Equation 6.1 for each metric. The use of this equation is predicated on a normal distribution of TRs. If the TRs are not normally distributed, the Box-Cox normal transformation can be used to transform the non-normal data into normal form [74]. Using historical data from comparable projects that were successful, the lower limit (LL) of each metric ratio is calculated using Equation 5. The mean of the historical values for each metric (μ_{SMx}) serves as the upper limit. The historical data is computed from previously-successful projects with acceptable levels of TRs/KLOC. In the absence of historical data, standard values can be used that are built from projects with similar acceptable levels of TRs/KLOC. The mean and the lower limit serve as the test quality feedback standards for the STREW metrics.

$$LL(SMx) = \mu_{SMx} - z_{\alpha/2} * \frac{\text{Standard deviation of metric SMx}}{\sqrt{n}} \qquad (6.1)$$

where μ_{SMx}- Mean of Metric SMx, n is the number of samples used to calculate μ_{SMx} and $Z_{\alpha/2}$ is the upper $\alpha/2$ quantile of the standard normal distribution (For example, $Z\alpha/2=1.96$ is the t-table value at 95% confidence interval, if sample size is greater than 30).

Using the computed values, we use a color-coded approach as shown in Table 6.1. In Table 6.1, **SMx** refers to the each particular STREW metric (SM1, SM2 …) for the software system under development compared to the calculated standards from successful projects.

Table 6.1: Color coded feedback standards

Color	Interpretation
RED	**$\underline{SMx} < LL$ (*Metric*)**
ORANGE	**LL (*Metric*) $\leq \underline{SMx} \leq \mu_{SMx}$**
GREEN	**$\underline{SMx} > \mu_{SMx}$**

The purpose of the color-coding is to direct corrective action to the test suite on the part of the programmer. Table 6.2 provides detail on the specific corrective actions the programmer can take to change a metric from red to orange or from orange to green.The explanation of all these metrics is with respect to the standards built for each metric.

Table 6.2: STREW metric color coded feedback explanation

Metric	Meaning of RED or ORANGE	Corrective Action
SM1	The assertions/KLOC are lower compared to projects that were used to build the feedback standards.	**Add more assertions**. These additional assertions should be meaningful assertions (can be cross checked with increase in coverage)
SM2	The test cases/KLOC are lower compared to projects that were used to build the feedback standards.	**Add more test cases**. These test case density but should be meaningful test cases (can be cross checked with increase in coverage)
SM3	The assertions/test case are lower compared to projects that were used to build the feedback standards.	**Add more assertions per test case**.
SM4	The overall testing effort measured in terms of lines of code and classes was not comparable in terms of the projects that were used to build the feedback standards.	**Add more lines of test code**. Increasing the overall testing effort (controlling the source code size) that would lead to an increase in the test lines of code and the test classes.

Table 6.2: (continued)

SM5	The complexity of the testing effort is not comparable in terms of the previously acceptable projects	**Decrease cyclomatic complexity ratio by decreasing the cyclomatic complexity of source code.** The simplest complexity measure, the cyclomatic complexity measure, ratio of the test code and source code indicates how well the testing has taken place at the complexity level. This metric works in conjunction with metric SM4 indicating that the testing effort was not thorough enough. When metric SM4 increases, metric SM5 should also increase with respect to checking for conditionals, infinite loops, unreachable code etc., that would increase the metric SM5
SM6	The CBO ratio of the test code to source code is not comparable to the previously successful projects.	**Increase the CBO ratio by reducing the coupling between objects in source code.** The larger the inter-object coupling, the higher the sensitivity to change. Therefore, maintenance of the code is more difficult. As a result, the higher the inter-object class coupling, the more rigorous the testing should be [17]. The CBO of the source code should be reduced to eliminate the dependencies caused by coupling.
SM7	The DIT ratio of the test code to source code is not comparable to the previously successful projects.	**Increase DIT ratio by reducing DIT in the source code.** Due to DIT, a change or a failure in a super class propagates down the inheritance tree. Hence we have to reduce the DIT of the source code to acceptable levels so that the DIT ratio is comparable to the projects used to build the standards.
SM8	The WMC ratio of the test code to source code is not comparable to the previously successful projects.	**Increase WMC ratio by breaking down complex classes in source code to child classes.** The larger the number of methods in a class, the greater is the potential impact on children, since the children will inherit all the methods defined in the class. Hence we have to reduce the number of methods in the source code to acceptable levels calculated by the projects used to build the standard. This metric essentially measures the importance of the modularity of the source code.

6.2 EVALUATION OF TEST QUALITY FEEDBACK STANDARDS

To evaluate the efficacy of the test quality feedback standards, we initially use a robust Spearman rank correlation technique with respect to the number of color-coded feedbacks obtained and the post-release TRs/KLOC. The association between the test quality feedback and post-release field quality to indicate the efficacy of the color-coded mechanism is shown in Table 6.3. In Section 6.4, we investigate the overall efficacy of the test quality feedback across all three development environments (academic, open source and industrial).

Table 6.3: Desired correlation results with the color-coded feedback

Color	Desired Spearman Correlation	Interpretation
RED + ORANGE	**Positive**	**Increase in (RED + ORANGE) increases post-release TRs/KLOC.**
GREEN	**Negative**	**Increase in (GREEN) decreases post-release TRs/KLOC**

The higher the number of red and orange feedbacks, the higher we expect the TRs/KLOC to be, as red and orange feedbacks denote a lower quality of the testing effort. This is indicated by a positive correlation coefficient between the number of red and oranges with the post-release TRs/KLOC. Inversely, the correlation coefficient between the number of green feedbacks and the TRs/KLOC is expected to be negative. The feedbacks are calculated only for the STREW metric ratios SM1-SM8 as SM9 is a size adjustment factor is a not an actual metric ratio for providing feedback.

6.3 CASE STUDIES

We present academic, open source and a structured industrial case study to build an empirical body of evidence to indicate the efficacy of our approach.

6.3.1 ACADEMIC CASE STUDY

We use the same random split of the 22 projects to build the test quality feedback standards and the remaining seven to check the efficacy of the built standards to provide feedback on the quality of the testing effort. We build the standards using the 15 academic projects and evaluate the projects using the seven remaining projects. The quantity of color-coded feedbacks for each of these seven projects is shown in Figure 6.1. The normality distribution of the STREW metrics is evaluated using a Kolmogorov-Smirnov test on the metrics with the null hypothesis that the population distribution is normal. The results of the Kolmogorov-Smirnov test are presented in Table C.1 in Appendix C.

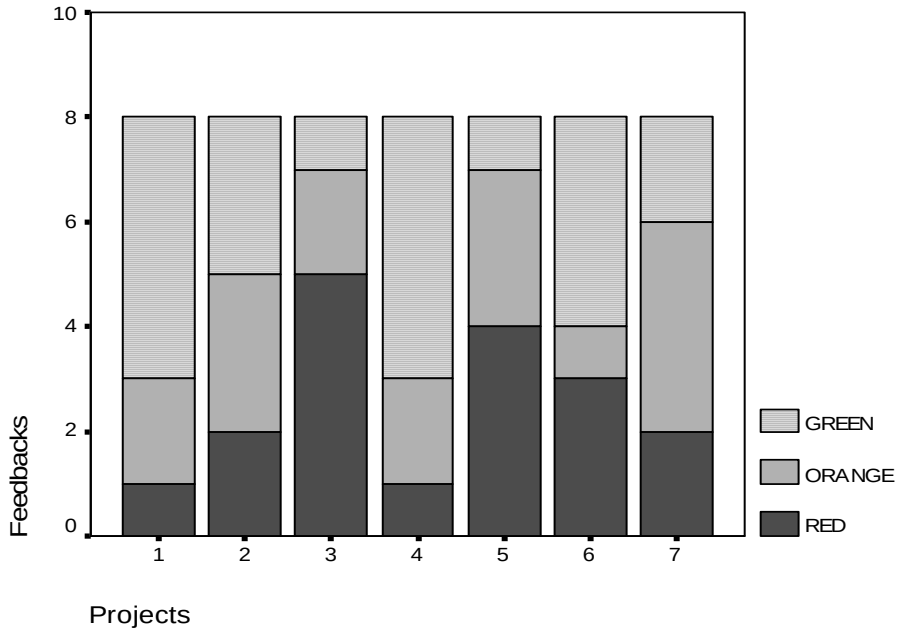


Figure 6.1: Color coded feedbacks-Academic case study

The quantity of reds and oranges and the quantity of greens is correlated to the TRs/KLOC, and the results are shown in Table 6.4. The red and orange feedbacks were added together because they represent a STREW metric ratio that is not of acceptable quality. The results in Table 6.4 indicate that the expected correlation results are as desired. However, the p values of the correlations are not statistically significant. This lack of statistical significance can be explained to a certain degree by the small size of the projects. The correlation coefficients have the same magnitude but different signs

because the number of reds and oranges is a linear function of the number of green feedbacks. The above results indicate the feasibility for further exploration of the test quality feedback standards.

Table 6.4: Academic case study – correlation results

	RED + ORANGE	**GREEN**
Correlation coefficient with TRs/ KLOC	0.145 (p=0.756)	-0.145 (p=0.756)

6.3.2 OPEN SOURCE CASE STUDY

In open source projects to evaluate test quality feedback in-process, we used thirteen versions of httpunit, one of the 27 open source software projects used in our earlier analysis. The system has a lifetime in use for three years. Each release has a time period of 2.5-3 months so that the TRs/KLOC collected are representative of equal usage. The TRs/KLOC data is available from customer-reported failure logs that are screened in the same way as the other open source projects in Chapter 5. The test quality feedback standards were calculated using the 27 open source projects, as discussed in Chapter 5. The test quality feedback for the 13 versions is evaluated against the standards built using the 27 open source projects. The 13 versions belong to a system used for software testing, i.e. belonging to the same domain as the 27 projects. The development language was in Java. Figure 6.2 shows the size of the 13 versions as they grew over a period of three years from almost 3 KLOC to 11.5 KLOC. Figure 6.2 represents only the LOC_{source} and not LOC_{test}.

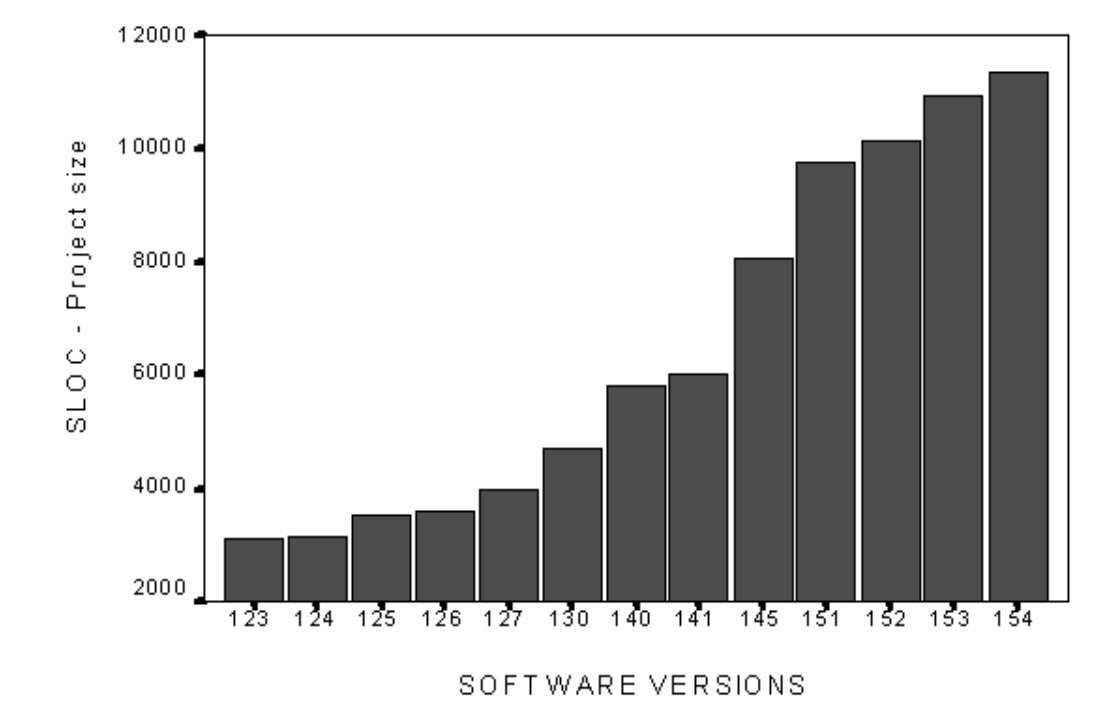


Figure 6.2: Project size across three years

The quantity of feedbacks for the thirteen versions of an open source software system are shown in Figure 6.3.

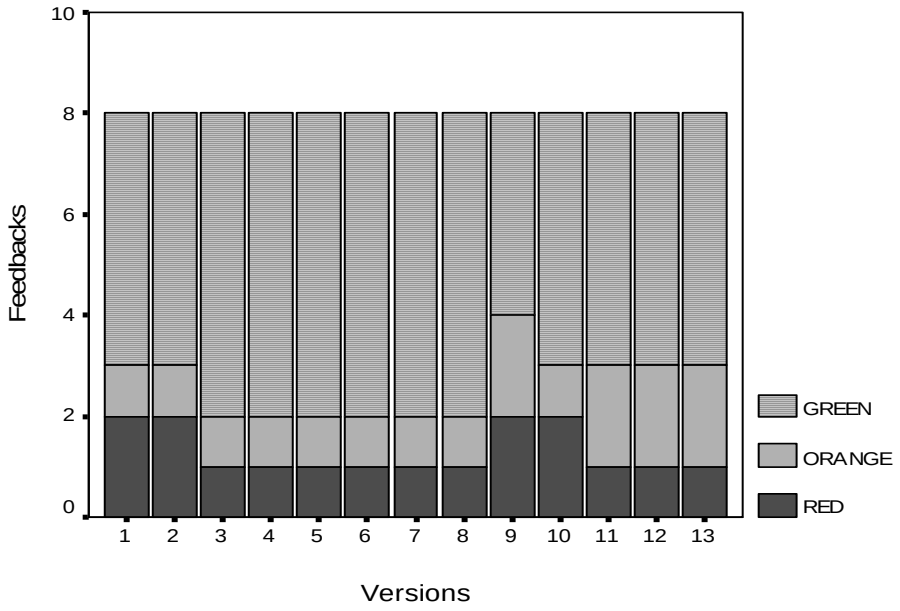


Figure 6.3: Color coded feedbacks for open source case study

Upon evaluation using a Spearman correlation we obtain the results as shown in Table 6.5.

Table 6.5: Open source case study – correlation results

	RED + ORANGE	GREEN
Correlation coefficient with TRs/KLOC	0.208 (p=0.496)	-0.208 (p=0.496)

From Table 6.5 we obtain similar results to the academic case study in terms of trend, the desired positive and negative correlation coefficients but the results not statistically significant likely due to the small sample size, as will be discussed in detail in Section 6.4.

6.3.3 STRUCTURED INDUSTRIAL CASE STUDY

To explore the efficacy of the test quality feedback mechanism, we performed a structured industrial case study to investigate the results under a more controlled environment. Six releases of a commercial software system "eXpert" were used for this case study. A metaphor that better describes the intended purpose of the system is a large sized "virtual file cabinet," which holds a large number of organized rich, i.e. annotated, links to physical or web-based resources [1]. The system has 300+ potential users and is a web-based client-server solution developed by four developers at VTT Technical Research Centre of Finland [1]. The four developers were 5-6th year university students with 1-4 years of industrial experience in software development. Team members were well-versed in the Java O-O analysis and design approaches. The overall development time was 2.1 months and post-release TRs/KLOC is available from failure logs. The development language was Java. The standards are calculated using the 27 open source projects for providing test quality feedback.

Since this was a "structured" case study, i.e. the level of control exhibited in the experiment was greater than the normal software development process. Measures such as effort and time were collected to make sure that these factors were in a comparable level across all the releases. This data is presented in Table 6.6 [1].

Table 6.6: Industrial project data description (adapted from [1])

No.	Collected Data	Release 1	Release 2	Release 3	Release 4	Release 5	Release 6 (correction phase)	Total
1.	Calendar time	2	2	2	1	1	0.4	8.4
2.	Total work effort (h)	195	190	192	111	96	36	820
3.	LOC per release	1821	2386	1962	460	842	227	7698
4.	TRs	4	5	4	4	11	0	28
5.	TRs/KLOC	2.19	2.10	2.04	8.70	13.06	0.0	1.43

The feedbacks calculated using the 27 open source projects for the six releases of an industrial software system are shown in Figure 6.4.

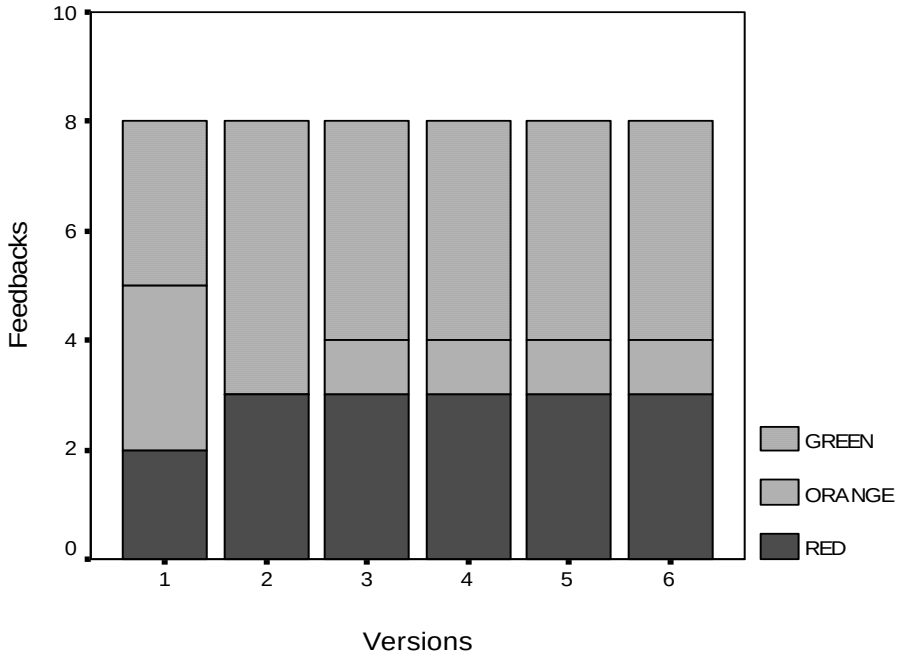


Figure 6.4: Color coded feedbacks for structured industrial case study

Upon evaluation using a Spearman correlation we obtain the results as shown in Table 6.7.

Table 6.7: Industrial case study – correlation results

	RED + ORANGE	GREEN
Correlation coefficient with post-release TRs/KLOC	0.338 (p=0.512)	- 0.338 (p=0.512)

The results in Table 6.7, demonstrate similar results to the academic case study in terms of trend (the desired positive and negative correlation coefficients) but the results are not statistically significant due to the small sample size.

6.4 STATISTICAL ANALYSIS OF TEST QUALITY FEEDBACK

Table 6.8 summarizes the correlation results of the three categories of the case. Across all the three categories of case studies, the results are indicative in terms of trend (with respect to the sign of the correlation results) but are not statistically significant. This lack of statistically significant correlations is likely due to the small size of the individual sample populations as is demonstrated by the following analysis as shown in Table 6.9. When all the three models are combined together, the Spearman correlation results indicate the desired correlation trends with the TRs/KLOC at statistically significant levels (correlation coefficient = 0.468 (p=0.016) and vice versa), as shown in Table 6.8. The positive coefficient of the red and oranges at statistically significant levels indicates that with the increase in the number of red and orange feedbacks, there is an increase in the TRs/KLOC and vice-versa with the green feedbacks. (The Pearson correlation coefficients were 0.488 (p=0.011) between the red + orange and the TRs/KLOC and -0.488 (p=0.011) for the number of greens and the TRs/KLOC.

Table 6.8: Summary correlation results of test quality feedback

Case study	Correlation Coefficient		Sample Size
	RED + ORANGE	GREEN	
Academic	0.145	-0.145	7
Open Source	0.208	-0.208	13
Industrial	0.338	-0.338	6
All three case studies combined	0.468 (p = 0.016)	- 0.468 (p = 0.016)	26

Further, to assess the effect of the environment (academic, open source and industrial case studies) on the test quality feedback, we perform a comparative modeling analysis using an ANCOVA (Analysis of Covariance) test, simple linear regression with the color-coded feedbacks and the TRs/KLOC, complete regression model including the interaction between the three types of case studies and the red + orange and green feedbacks, and an ANOVA (Analysis of Variance) test. ANCOVA can be used to test for differences among groups (in our case between the academic, open source and industrial case studies) to identify the relationship with the dependent variable (TRs/KLOC). To carry out these tests, we first fix the covariate environment variable to be 1 for academic case studies, 2 for open source case studies and 3 for industrial case studies. This nominal data allows tests like the ANCOVA to take into account that the data is from three different environments each of which may have a different ability with respect to the ability to provide test quality feedback. The results for the comparative modeling with its associate R^2, sum of squares due to regression (SS[R]), sum of squares due to error (SS[E]), and the mean square error (MSE) are shown in Table 6.9. Based on the values of F-test for model selection and the MSE, we can select the best model for integrating data across all the three environments.

Table 6.9: Comparative modeling results

Model No.	Model	R^2	SS[R]	SS[E]	MSE
1.	Simple Linear Regression	0.239	28.2	89.9	3.74
2.	ANCOVA	0.349	41.3	76.9	3.5
3.	Complete Regression Model	0.367	43.4	74.8	3.74
4.	ANOVA	0.339	40.1	78.1	3.4

Two F-tests for model selection between model 1 and model 2 (F=1.87, p=0.178) and between model 1 and model 3 (F=1.01, p=0.424) indicate that model 1, the simple linear regression model is the best model. The ANOVA test also does not indicate any significant change in the MSE factor. The results of these two F-tests provide no evidence of any effect of the environment on the TRs/KLOC. The association between the TRs/KLOC and the red + oranges and greens is consistent across environments. Thus, the color-coded feedbacks indicates the variability in the TRs/KLOC at statistically significant levels. Further, the residual diagnostics also did not indicate a violation of the model assumptions. Hence on performing simple linear regression we get an R^2 value= 0.239, (F= 7.533, p=0.011). Table 6.10 shows the regression coefficients with the TRs/KLOC as the dependent variable. The bold values indicate that the coefficient is positive and is statistically significant indicating that with an increase in the number of red and orange test quality feedbacks there is an increase in the TRs/KLOC (and vice versa for green feedbacks).

Table 6.10: Simple linear regression coefficients

	Unstandardized		Standardized	t-test	sig
	Coefficients	MSE	Coefficients		
Constant	-0.135	1.030		-0.131	0.897
Red+Orange	**0.735**	0.268	0.489	**2.745**	**0.011**

6.5 QUALITATIVE ANALYSIS

To serve as a check on the statistical studies, we did a qualitative analysis of the test quality feedback standards. For this purpose we use the "eXpert"industrial case study. The test quality feedback standards constructed using the 27 open source projects are shown in Table 6.11.

Table 6.11: Test quality feedback standards – open source case study

Metric	RED	ORANGE	GREEN
SM1	< 0.0545	[0.0545, 0.0822]	> 0.0822
SM2	< 0.0721	[0.0721, 0.1261]	> 0.1261
SM3	< 0.7790	[0.7790, 1.0958]	> 1.0958
SM4	< 0.7881	[0.7881, 1.0427]	> 1.0427
SM5	< 0.2474	[0.2474, 0.3376]	> 0.3376
SM6	< 0.3417	[0.3417, 0.4490]	> 0.4490
SM7	< 0.3498	[0.3498, 0.4931]	> 0.4931
SM8	< 0.2349	[0.2349, 0.3217]	> 0.3217

For qualitative evaluation we examine each release of the industrial case study with the standards in Table 6.11. We cannot assume that a correlation with a limited data set implies causality. This analysis is intended to study the change in the number of red, orange and green feedbacks obtained in-process with the TRs/KLOC. Table 6.12 to 6.17 explain the qualitative feedback of the STREW metrics in each release.

Based on Table 6.12, The post release field quality was 2.19 TRs/KLOC. Overall five of the STREW metrics were in the not acceptable range (red or orange). The metrics that were red or orange are assertions/SLOC, test cases/SLOC, testing effort size in terms of lines of code and classes, depth of inheritance, and weighted methods per class. We see that the test effort was not comparable to the open source standards as three of the test quantification metrics and two of the O-O metrics in terms of test complexity were not of appropriate quality. This explains the higher TRs/KLOC than the mean TRs/KLOC of the open source projects. (Complete TRs/KLOC list is available in Appendix A).

Table 6.12: Release 1 STREW metrics feedback

	SM1	SM2	SM3	SM4	SM5	SM6	SM7	SM8
	0.0805	0.0273	2.9444	0.7026	0.4916	0.8148	0.4	0.2484
Color	Orange	Red	Green	Red	Green	Green	Orange	Orange
Prior release color	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
Post-release field quality = 2.19 TRs/KLOC								

From Table 6.13 we can see that the number of red and orange feedbacks combined is reduced in Release 2 to four, (though the number of red feedbacks individually increases to three from two) in Release 1. The metrics quantifying the depth of inheritance tree changes from orange to green indicating the increase in complexity ratio of the test effort. Also, the assertion density changed to green from orange. Further, the metrics also numerically show an increase in terms of the test quantification metrics. This increase indicates a minor improve in quality reflected by a decrease in the actual TRs/KLOC to 2.10 TRs/KLOC.

Table 6.13: Release 2 STREW metrics feedback

	SM1	SM2	SM3	SM4	SM5	SM6	SM7	SM8
	0.0927	0.0279	3.3157	0.6417	0.5130	1.0980	1	0.2181
Color	Green	Red	Green	Red	Green	Green	Green	Orange
Prior release color	Orange	Red	Green	Red	Green	Green	Orange	Orange
Post-release field quality = 2.10 TRs/KLOC								

Table 6.14 indicates that the total number of red and orange feedbacks increases by one for Release 3. The assertion density (SM1) changed from green to orange indicating that the testing in terms of assertions undergoing a decrease (i.e. more new code was added but the corresponding assertions added were not up to the previous assertion density ratio). The weighted methods per class metric (SM8) changed from orange to red indicating a decrease in quality with respect to the number of test methods per class. The quantitative chamges of the STREW metrics is relatively less compared

to Release 2 with Release 1. This leads to an explanation for a similar TRs/KLOC of 2.04 TRs/KLOC like release 3.

Table 6.14: Release 3 STREW metrics feedback

	SM1	**SM2**	**SM3**	**SM4**	**SM5**	**SM6**	**SM7**	**SM8**
	0.0749	0.0196	3.8181	0.6362	0.4405	0.8666	0.8666	0.1702
Color	Orange	Red	Green	Red	Green	Green	Green	Red
Prior release color	Green	Red	Green	Red	Green	Green	Green	Orange
Post-release field quality = 2.04 TRs/KLOC								

As seen from Table 6.15, Release 4 had minor change in code. Primarily the in-process development was completed by Release 3. Releases 4, 5, and 6 were primarily to improve quality. Most of the effort in Release 4 was concentrated on fixing testing defects. This is indicated by the numerical quantification of all the eight STREW metrics, which each only have a minor change, (only in the second decimal place) compared to Release 3. This indicates the high TRs/KLOC associated with Release 4.

Table 6.15: Release 4 STREW metrics feedback

	SM1	**SM2**	**SM3**	**SM4**	**SM5**	**SM6**	**SM7**	**SM8**
	0.0751	0.0196	3.8181	0.6422	0.4510	0.8901	0.8666	0.1781
Color	Orange	Red	Green	Green	Red	Green	Green	Red
Prior release color	Orange	Red	Green	Red	Green	Green	Green	Red
Post-release field quality = 8.70 TRs/KLOC								

Similar to Release 4, Release 5 shown in Table 6.16 also has even fewer new feature additions to code. Most of the changes were bug fixes which can be used to explain the high TRs/KLOC associated with Release 5.

Table 6.16: Release 5 STREW metrics feedback

	SM1	SM2	SM3	SM4	SM5	SM6	SM7	SM8
	0.0730	0.0183	3.9782	0.6686	0.4234	0.8645	0.7777	0.1724
Color	Orange	Red	Green	Green	Red	Green	Green	Red
Prior release color	Orange	Red	Green	Green	Red	Green	Green	Red
Post-release field quality = 13.06 TRs/KLOC								

Release 6, shown in Table 6.17 was a correction phase in which there was no new feature development. This affects the STREW metrics as the only changes to them will occur when there are bug fixes. These changes will primarily not affect any of the feedbacks in the metrics compared to Release 5. If the lower quality metrics (red feedbacks) are addressed namely, for example increasing test effort (more test code → in turn test cases → in turn assertions) will also increase the number of methods per class compared to the source code leading to better quality of code with fewer failures.

Table 6.17: Release 6 STREW metrics feedback

	SM1	SM2	SM3	SM4	SM5	SM6	SM7	SM8
	0.0720	0.0177	4.0652	0.6471	0.3995	0.8469	0.7777	0.1696
Color	Orange	Red	Green	Red	Green	Green	Green	Red
Prior release color	Orange	Red	Green	Green	Red	Green	Green	Red
Post-release field quality = 0.0 TRs/KLOC								

There are two limitations to our investigation of test quality feedback. One is the small size of the samples (seven academic, 13 open source, and six industrial) which make it difficult to obtain a statistical significance at a high 95% confidence. Secondly, the analyses were all done post-mortem, i.e. the feedback results were not used to improve the testing effort as development proceeded. This post-mortem feedback explains to a certain degree the similar number of color-coded feedbacks in Figure 6.2 and Figure 6.3. Changing the red and orange feedbacks in these figures to green would further serve in strengthening the current results with significantly higher levels of statistical acceptance.

6.6. DISCRIMINATIVE POWER

In addition to providing test quality feedback, we investigate the ability of the STREW metrics to identify programs of low and high quality. For this purpose we use logistic regression with the STREW metrics as the covariates and the post-release field quality (1- low quality and 0-high quality) as the dependent variable. The field quality is calculated using Equation 6.1 on the TRs/KLOC such that all projects with a TRs/KLOC lower than the calculated lower bound is "High Quality", otherwise it is classified as "Low Quality". The quality of the logistic regression (Type I and Type II errors) is evaluated as shown in Table 6.18.

Table 6.18: Type 1 and Type II errors

		Predicted	
		High Quality	**Low Quality**
Observed	**High Quality**	**Correct**	**Type 1 Error**
	Low Quality	**Type II Error**	**Correct**

The results for the Type I and Type II errors for the academic and open source case study are shown in Table 6.18. The results shown in Table 6.19 indicate that a high proportion of "Low Quality" and "High Quality" projects are correctly identified. This result adds knowledge to our already existing empirical knowledge base of the test quality feedback results indicating the efficacy of the STREW metric suite.

Table 6.19: Type I, Type II errors for case studies

	Type I Error	**Type II Error**	**Overall Model Fit**
Academic Case Study (n=22)	0 (0%)	0 (0%)	**100%** (9 +13)/22 (Low Quality =13, High Quality =9)
Open Source Case Study (n=27)	2/11 (18.15%) [2.94%, 33.36%]	1/16 (6.25%) [0%, 15.74%]	**88.88% [74.68%, 100.00%]** (9+15)/27 (Low Quality =16, High Quality =11)

Also Table 6.19 presents the error in the estimates within box brackets. The error in the estimates for classification cannot be computed for the academic case study as there are no misclassifications. Further, for providing test quality feedback, we have automated the collection and analysis of the STREW metrics Version 1.4 via an open source Eclipse plug-in GERT (Good Enough Reliability Tool)[9] [22, 69]. In addition to providing a reliability estimate, the tool provides color-coded feedback on the quality of the testing effort relative to historical data from comparable projects. Appendix B presents a brief summary of the tool and its deployment in the Sourceforge software repository.

[9] gert.sourceforge.net

CHAPTER 7

RETROSPECTIVE STREW METRIC ANALYSIS

This chapter deals with a retrospective analysis of the STREW metric suite, more specifically identifying the relationship between the STREW metrics and the TRs/KLOC similar to the work done by Vouk et. al.[93]. We investigate several hypothesis with relation to the STREW metrics and other project metrics. These hypotheses are discussed below. In all our discussions, we restrict ourselves to the academic and open source case study only because the industrial case study has only five points.

H_0: There is no relationship between the number of TRs and the size of the code.

Figure 7.1 presents the scatter plots between the TRs and the size of the code. It also presents the Pearson correlation results between the two measures to numerically quantify the scatter plots. The correlation results in Figure 7.1 indicate that for the open source projects the large projects have more TRs at statistically significant levels. A contrary relationship is observed with respect to the academic projects. But this relationship is not statistically significant and is on a much smaller scale of projects indicating that this evidence is not in direct contradiction to the open source projects. *Hence we can reject the null hypothesis and state that there exists a positive association between the size of a project and the number of TRs.*

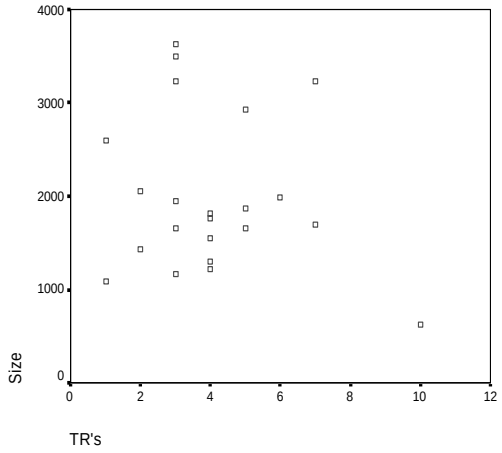
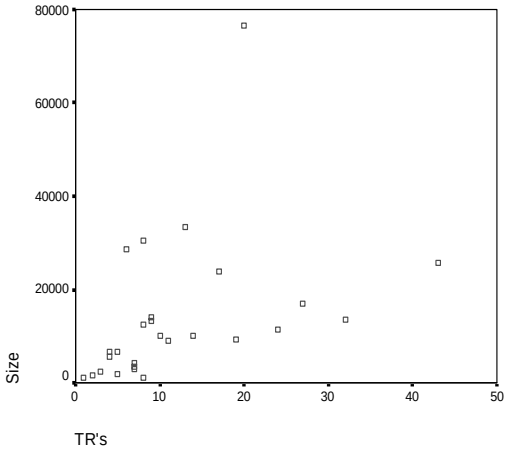
Academic Case Study	Open Source Case Study
	
Correlation = -0.168 (p=0.455)	Correlation = 0.393 (p=0.043)

Figure 7.1: Scatterplots of TRs Vs. Size

H_0: There is no relationship between the number of TRs/KLOC and the size of the code.

We perform a similar scatter plot analysis on the TRs/KLOC and the size of the code along with correlation results for both the academic and open source projects, the results of which are shown in Figure 7.2.

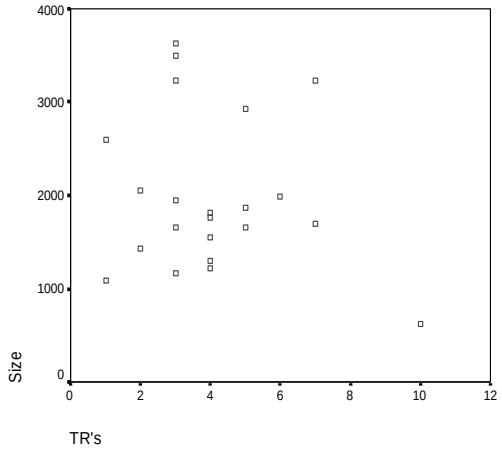
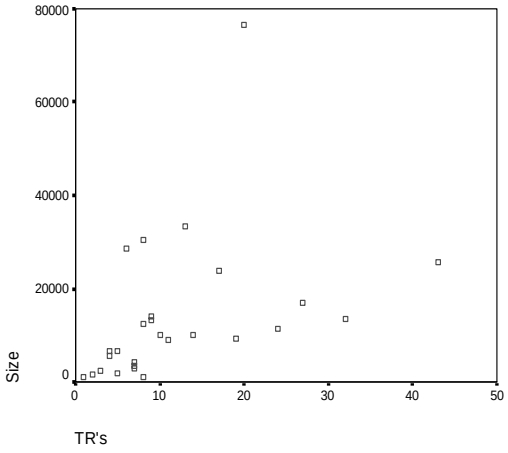
Academic Case Study	Open Source Case Study
	
Correlation = -0.627 (p=0.002)	Correlation = -0.397 (p=0.041)

Figure 7.2 Scatterplots of TRs/KLOC Vs. Size

The results of both the academic and the open source case study indicate that the there is a negative statistically significant relationship between the size and the TRs/KLOC. With increase in

the size of the systems there is a decrease in the TRs/KLOC. These results are in agreement with previously published empirical studies [4, 32, 40, 61, 76]. *Thus we can reject the null hypothesis that there is no relationship between the TRs/KLOC and the size of the code.* This research hypothesis indicates that our case study software systems were similar to previously studied software systems, and the STREW metric suite results are not likely to be due to a difference in the underlying relationship between the TRs/KLOC and the project sizes.

H_0: There is no relationship between the number of TRs/KLOC and the asserts and test cases of the code.

To address this hypothesis, we plot a three dimensional plot with the TRs/KLOC on the x axis and the asserts/KLOC and test cases/KLOC on the y and z axis. The plots for the open source and academic projects are shown in Figure 7.3.

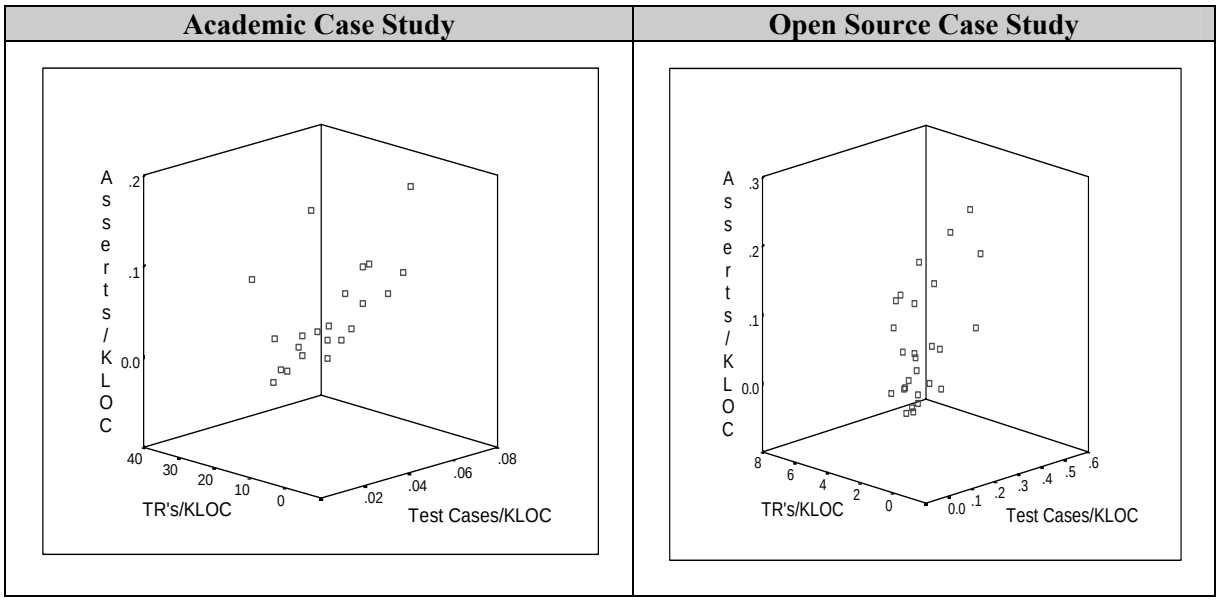


Figure 7.3: 3-D plots of Asserts/KLOC vs. Test Cases/KLOC vs. TRs/KLOC

From Figure 7.3, we observe a clustered pattern indicating the three way relationship between the asserts/KLOC, test cases/KLOC and TRs/KLOC. *This leads us to reject the null hypothesis that there is no relationship between asserts/KLOC, test cases/KLOC and TRs/KLOC.* To check for consistency

when normalized by size we perform the same analysis using the absolute measures of the asserts, test cases, and TRs. The results are shown in Figure 7.4.

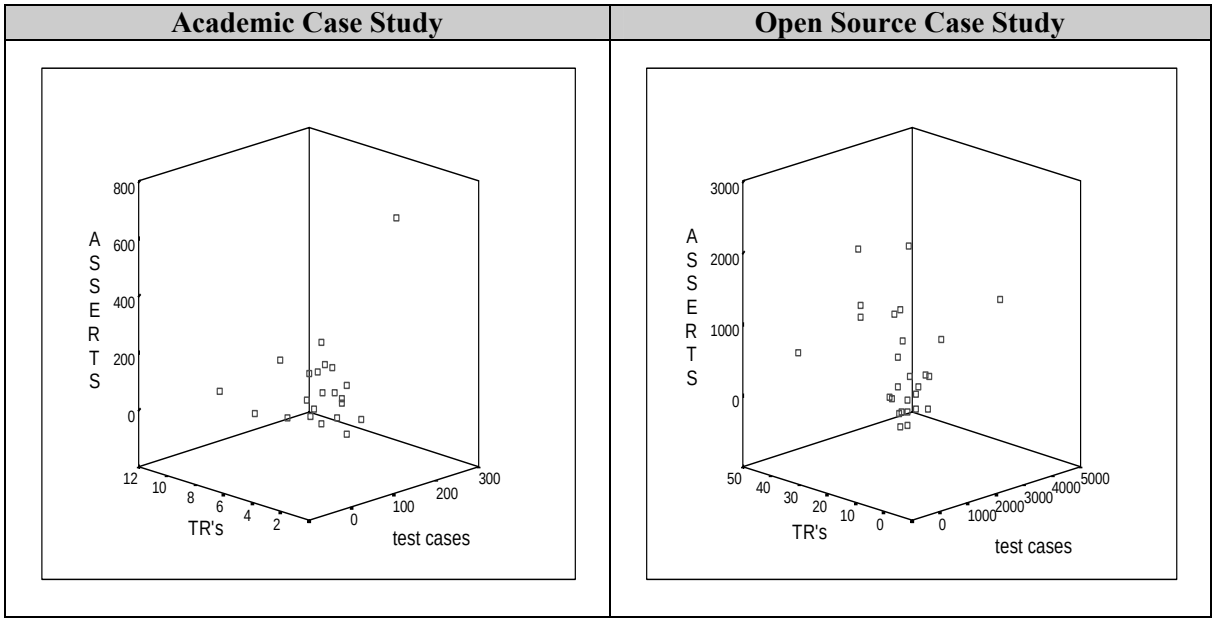


Figure 7.4: 3-D plots of Asserts vs. Test Cases vs. TRs

The 3-D plots in Figure 7.4 also show clustering across both the academic and open source case studies indicating the results hold across the normalizing effect of size on the asserts, test cases and TRs. Figure 7.5 shows the relationship between the statement and branch coverage and the TRs/KLOC for the academic projects.

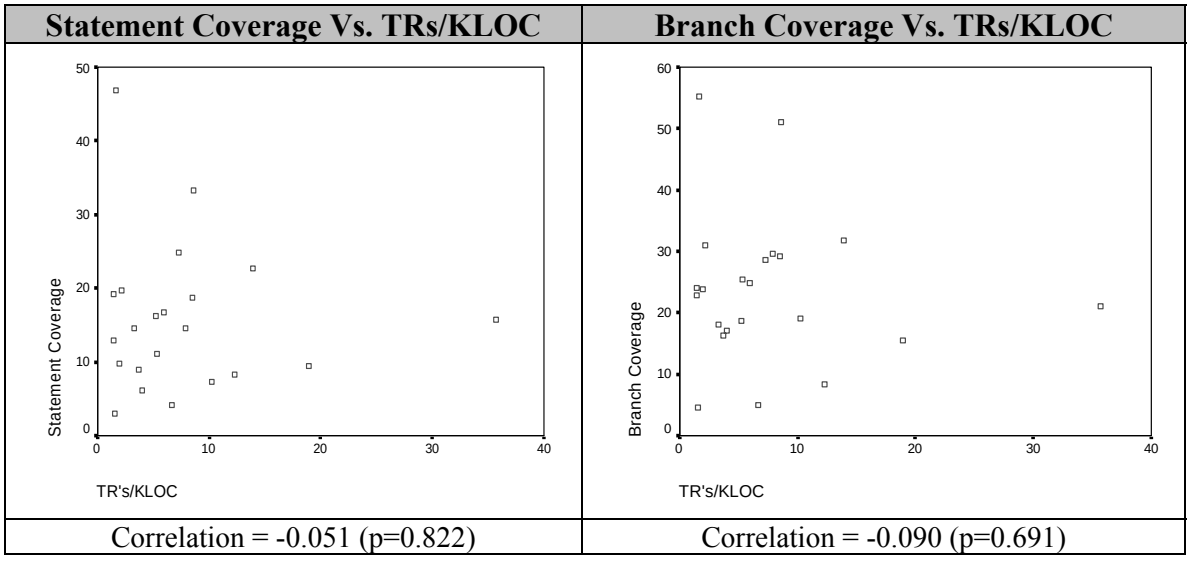


Figure 7.5: Statement and Branch Coverage with TRs/KLOC

The statistical results indicate that there is no correlation between the coverage and the TRs/KLOC. Further, the majority of the overall statement and branch coverage is below 30%. The projects were required to have a minimum of 80% statement coverage in their JUnit unit test suite for non-GUI code (i.e. program logic), indicating a uniformness in the testing effort. The 30% is when both GUI and non-GUI code is considered. The analysis when performed with the TRs also leads to similar results.

Table 7.1 presents the correlation matrix of the academic case studies. The inter correlation between the elements indicates the multicollinearity and the appropriateness of principal component analysis. The TRs/KLOC is correlated with the CBO ratio and the RCAF. The other elements may not have been correlated because of the small sample size of the academic projects.

Table 7.1: Correlation Matrix – Academic Projects

		TRs	TRs/ KLOC	Asserts/ KLOC	TCs/ KLOC	Asserts/ TCs	TLOC/tcl/ SLOC/scl	CCT/ CCS	CBO T/ CBOS	DITT/ DITS	WMCT/ WMCS	RCAF
TRs	**Pearson Correlation**	**1**	**.752**	**-.309**	**.216**	**-.413**	**-.075**	**-.032**	**.199**	**-.008**	**-.178**	**-.335**
	Sig. (2-tailed)	**.**	**.000**	**.162**	**.335**	**.056**	**.741**	**.888**	**.374**	**.972**	**.429**	**.127**
TRs/ KLOC	**Pearson Correlation**	**.752**	**1**	**-.185**	**.226**	**-.307**	**-.103**	**.393**	**.642**	**.101**	**.118**	**-.627**
	Sig. (2-tailed)	**.000**	**.**	**.410**	**.312**	**.165**	**.647**	**.071**	**.001**	**.655**	**.600**	**.002**
asserts/ KLOC	Pearson Correlation	-.309	-.185	1	**.450**	**.569**	**.510**	.011	.021	**.447**	-.020	.106
	Sig. (2-tailed)	.162	.410	.	.036	.006	.015	.963	.927	.037	.929	.637
TCs/ KLOC	Pearson Correlation	.216	.226	**.450**	1	-.310	**.564**	.096	.069	.402	-.032	-.169
	Sig. (2-tailed)	.335	.312	.036	.	.160	.006	.670	.760	.063	.886	.452
asserts/ TCs	Pearson Correlation	-.413	-.307	**.569**	-.310	1	-.095	.097	.153	.045	.183	.239
	Sig. (2-tailed)	.056	.165	.006	.160	.	.675	.668	.497	.842	.416	.285
TLOC/tcl/ SLOC/scl	Pearson Correlation	-.075	-.103	**.510**	**.564**	-.095	1	-.127	-.127	**.660**	-.184	.121
	Sig. (2-tailed)	.741	.647	.015	.006	.675	.	.573	.572	.001	.412	.590
CCT/ CCS	Pearson Correlation	-.032	.393	.011	.096	.097	-.127	1	**.819**	.205	**.834**	-.112
	Sig. (2-tailed)	.888	.071	.963	.670	.668	.573	.	.000	.361	.000	.619

Table 7.1 (continued)

CBOT/ CBOS	Pearson Correlation	.199	**.642**	.021	.069	.153	-.127	**.819**	1	.283	**.520**	-.214
	Sig. (2-tailed)	.374	.001	.927	.760	.497	.572	.000	.	.202	.013	.340
DITT/ DITS	Pearson Correlation	-.008	.101	**.447**	.402	.045	**.660**	.205	.283	1	.116	.281
	Sig. (2-tailed)	.972	.655	.037	.063	.842	.001	.361	.202	.	.606	.205
WMCT/ WMCS	Pearson Correlation	-.178	.118	-.020	-.032	.183	-.184	**.834**	**.520**	.116	1	-.087
	Sig. (2-tailed)	.429	.600	.929	.886	.416	.412	.000	.013	.606	.	.700
RCAF	Pearson Correlation	-.335	**-.627**	.106	-.169	.239	.121	-.112	-.214	.281	-.087	1
	Sig. (2-tailed)	.127	.002	.637	.452	.285	.590	.619	.340	.205	.700	.

Similarly Table 7.2 shows the correlation matrix for the open source case studies. The correlations for the TRs and TRs/KLOC are highlighted in bold to show their individual correlations with the STREW metrics. The TRs/KLOC are correlated with the STREW metrics, except SM1, SM3, and SM4.

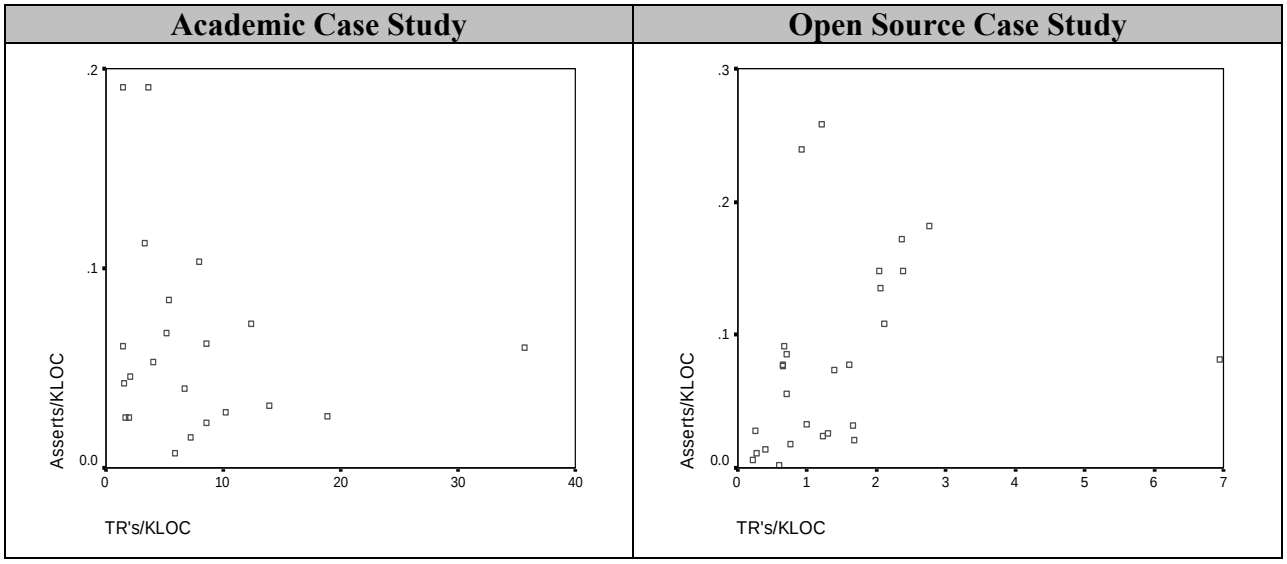
Table 7.2: Correlation Matrix – Open Source projects

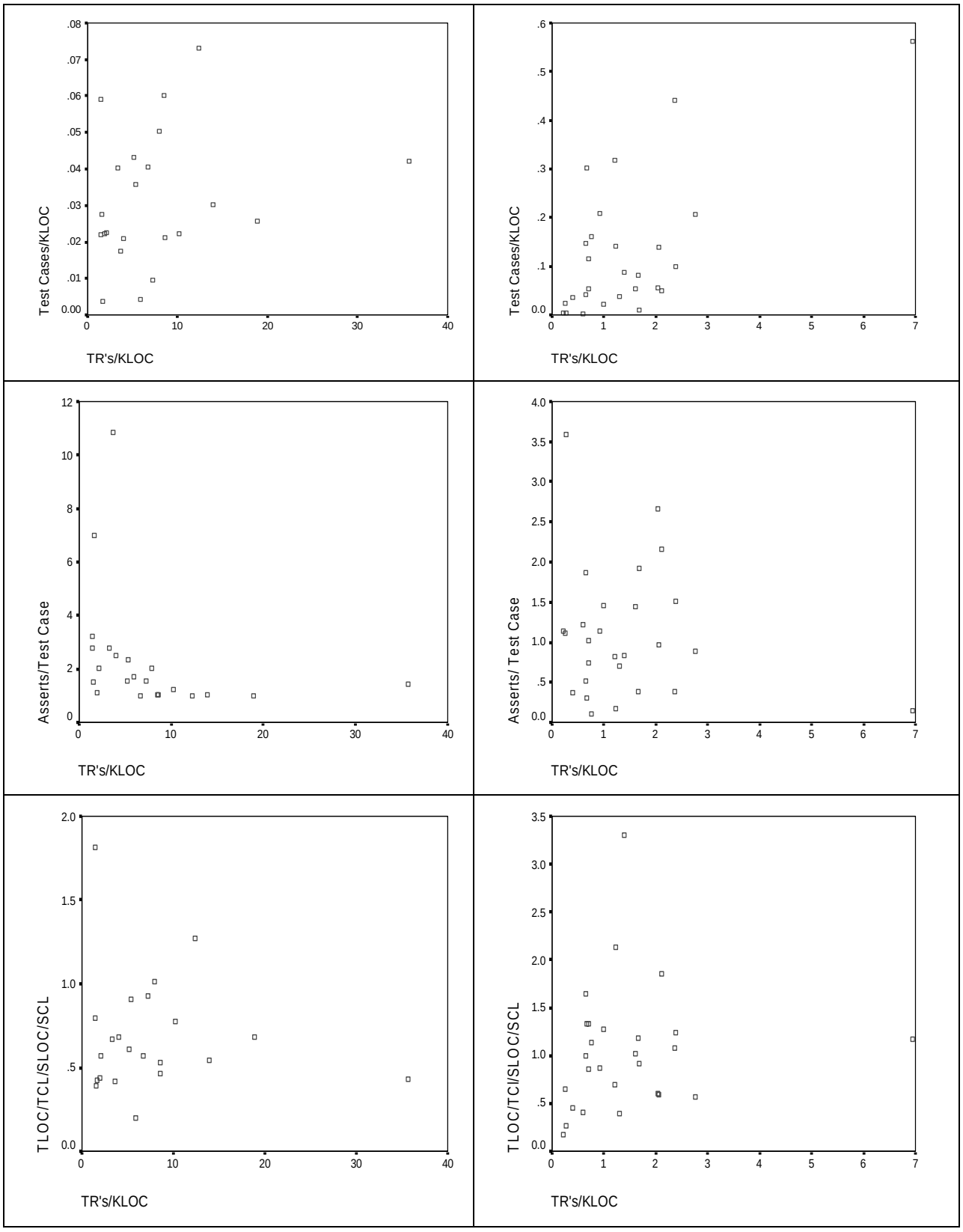
		TRs	TRs/ KLOC	asserts/ KLOC	TCs/ KLOC	asserts/ TCs	TLOC/tcl/ SLOC/scl	CCT/ CCS	CBOT /CBOS	DITT/ DITS	WMCT/ WMCS	RCAF
TRs	**Pearson Correlation**	**1**	**.102**	**-.143**	**-.293**	**.289**	**.162**	**-.074**	**.107**	**-.115**	**.017**	**.393**
	Sig. (2-tailed)	**.**	**.612**	**.478**	**.138**	**.144**	**.419**	**.715**	**.594**	**.567**	**.932**	**.043**
TRs/ KLOC	**Pearson Correlation**	**.102**	**1**	**.288**	**.667**	**-.174**	**.129**	**.585**	**.583**	**.406**	**.644**	**-.397**
	Sig. (2-tailed)	**.612**	**.**	**.146**	**.000**	**.384**	**.523**	**.001**	**.001**	**.036**	**.000**	**.041**
asserts/ KLOC	Pearson Correlation	-.143	.288	1	.528	-.011	.022	.654	.654	.715	.537	-.435
	Sig. (2-tailed)	.478	.146	.	.005	.958	.912	.000	.000	.000	.004	.023
TCs/KLOC	Pearson Correlation	-.293	.667	.528	1	-.514	.141	.718	.579	.437	.607	-.424
	Sig. (2-tailed)	.138	.000	.005	.	.006	.482	.000	.002	.023	.001	.028
asserts/TCs	Pearson Correlation	.289	-.174	-.011	-.514	1	-.257	-.164	-.127	-.064	-.081	.216
	Sig. (2-tailed)	.144	.384	.958	.006	.	.196	.413	.528	.753	.688	.280
TLOC/tcl/ SLOC/scl	Pearson Correlation	.162	.129	.022	.141	-.257	1	.183	.150	-.251	.134	-.220
	Sig. (2-tailed)	.419	.523	.912	.482	.196	.	.361	.456	.206	.505	.269

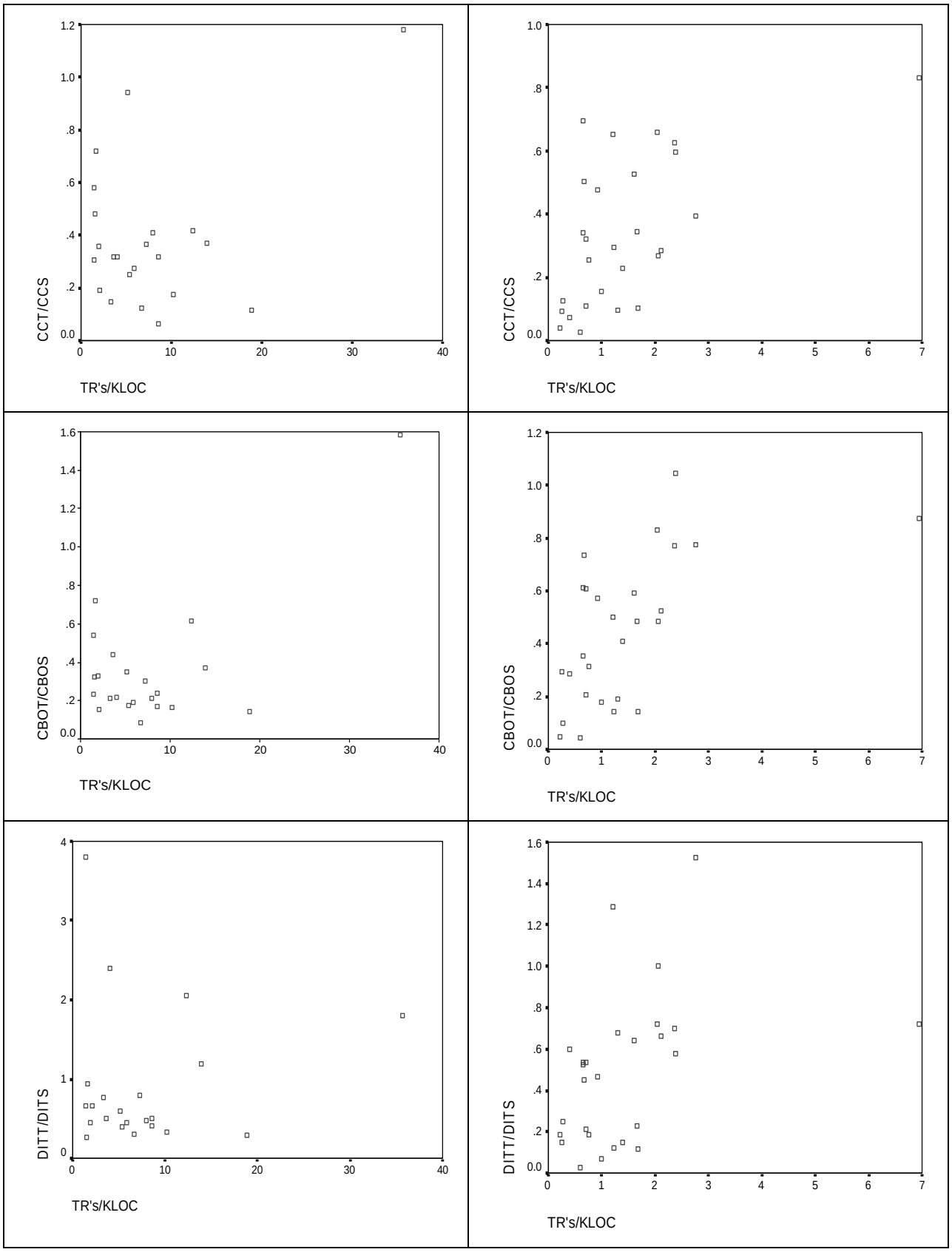
Table 7.2 (continued)

CCT/CCS	Pearson Correlation	-.074	.585	.654	.718	-.164	.183	1	.854	.525	.931	-.434
	Sig. (2-tailed)	.715	.001	.000	.000	.413	.361	.	.000	.005	.000	.024
CBOT/ CBOS	Pearson Correlation	.107	.583	.654	.579	-.127	.150	.854	1	.592	.810	-.310
	Sig. (2-tailed)	.594	.001	.000	.002	.528	.456	.000	.	.001	.000	.116
DITT/DITS	Pearson Correlation	-.115	.406	.715	.437	-.064	-.251	.525	.592	1	.457	-.336
	Sig. (2-tailed)	.567	.036	.000	.023	.753	.206	.005	.001	.	.016	.086
WMCT/ WMCS	Pearson Correlation	.017	.644	.537	.607	-.081	.134	.931	.810	.457	1	-.391
	Sig. (2-tailed)	.932	.000	.004	.001	.688	.505	.000	.000	.016	.	.044
RCAF	Pearson Correlation	.393	-.397	-.435	-.424	.216	-.220	-.434	-.310	-.336	-.391	1
	Sig. (2-tailed)	.043	.041	.023	.028	.280	.269	.024	.116	.086	.044	.

The scatter plots for the STREW metric elements with TRs/KLOC are shown in Figure 7.6 for both the academic and open source projects. These plots are graphical representation of the correlations in Table 7.1 and Table 7.2. These correlations are to graphical represent the association between the TRs/KLOC and the STREW metrics.




Test Cases/KLOC
TR's/KLOC
Test Cases/KLOC
TR's/KLOC
Asserts/Test Case
TR's/KLOC
Asserts/ Test Case
TR's/KLOC
TLOC/TCL/SLOC/SCL
TR's/KLOC
TLOC/TCl/SLOC/SCL
TR's/KLOC


CCT/CCS
TR's/KLOC
CCT/CCS
TR's/KLOC
CBOT/CBOS
TR's/KLOC
CBOT/CBOS
TR's/KLOC
DITT/DITS
TR's/KLOC
DITT/DITS
TR's/KLOC

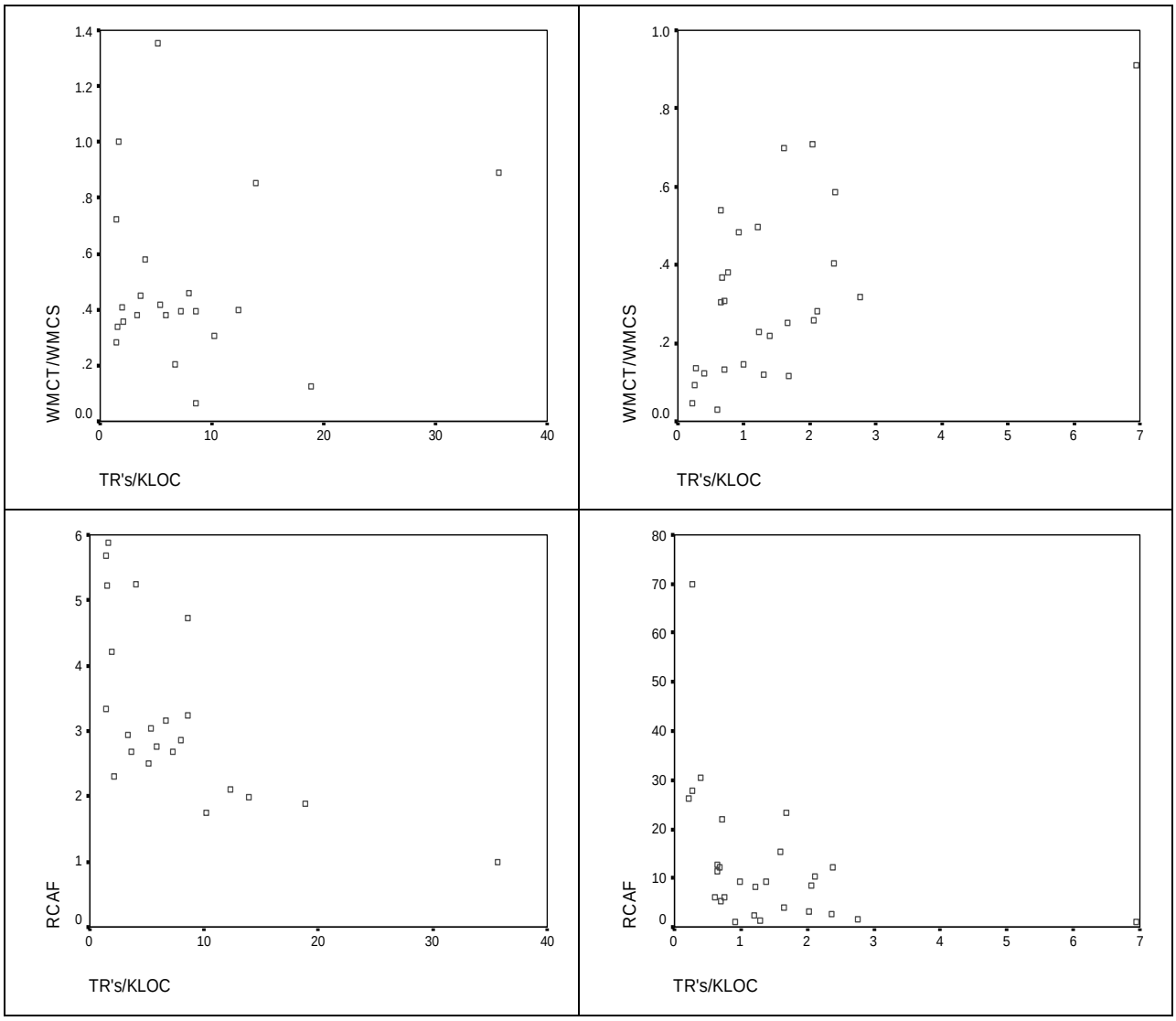


Table 7.6: Scatter plots of STREW metrics with TRs/KLOC

In Appendix C, Table C.3 and C.4 show a similar correlation matrix between the individual project metrics and TRs to highlight the association between the project metrics for both the open source and the industrial projects.

CHAPTER 8

CONCLUSIONS AND FUTURE WORK

An early estimation of potential software field quality is very useful to developers because it helps in identifying the overall quality of the software system. However, in most production environments, field quality is measured too late to affordably guide significant corrective actions. In this dissertation, we have reported on the usage of an in-process suite of test metrics for providing an early warning regarding post-release field quality and for providing meaningful feedback on the thoroughness of a testing effort.

The STREW metric suite is intended to be easy to gather in the development environment so that developers can receive an early indication of system TRs/KLOC throughout development. This estimation allows developers to take corrective actions early. However, these benefits will not accrue unless an adequate number of test cases that stress the system are written in the tight feedback loops, as is done in the TDD [9] paradigm. The STREW metric suite has been designed to provide the developer feedback to guide in the incremental creation of a thorough test suite. Additionally, the STREW metric suite leverages the utility of the extensive suite of automated tests to identify low quality projects throughout development. We note that the STREW metric suite can be used with any system with an automatic test suite (not script based testing), not only for systems that are developed using TDD.

To summarize the main contributions of this dissertation are:

- *Development of an in-process STREW metric suite that leverages the testing effort to estimate the post-release field quality of the software. The STREW metric suite is a set of static code measures that can be used to estimate the dynamic measure of post-release field quality.*
- *Empirical evidence of the ability of the STREW metric suite to estimate post-release field quality at statistically significant levels using three case studies in the academic, open source and industrial environments.*
- *Empirically assess the ability of the STREW metric suite to provide meaningful feedback on the thoroughness of a testing effort using three case studies in the academic, open source and industrial environment.*

Our case studies reflect only a relatively small set of data points and contexts. We plan to follow a process similar to the research done with the COCOMO II cost estimation model [18] by Barry Boehm at USC, over a period of 15 years with 83 data points from commercial, aerospace, government, and nonprofit organizations. We will build a more general, robust STREW-based post-release field quality estimation model by obtaining data from many industrial organizations, in multiple domains, with varying degrees of quality. Similar to the COCOMO methodology we plan to make our model robust to handle projects with a wide spectrum of post-release field quality.

We will continue to validate the metric suite under different industrial and academic environments and refine metric suite according to specific industry characteristics with emphasis on smaller granularity of measurement, i.e. instead on a project basis, measure the STREW metrics on a function/class basis to provide more individual feedback to developers. We will continue to refine the metric suite by add/deleting new metrics based on the results of further studies. An important step towards generalizing STREW would involve modifying STREW metrics for non – OO languages as languages like C that form a large part of the legacy code base. We plan to use non-automated test

results with STREW to further enhance the measurement of the testing effort. In this thesis, we use projects from teams that develop a history of the value of the STREW metrics from comparable projects with acceptable levels of post-release field quality.

REFERENCES

[1] Abrahamsson, P., Koskela, J., "Extreme Programming: A Survey of Empirical Data from a Controlled Case Study," Proceedings of International Symposium on Empirical Software Engineering, 2004, pp. 73-82.

[2] Albrecht, A. J., "Software function, source lines of code, and development effort prediction," *IEEE Transactions on Software Engineering*, Vol. 9, No. 6, pp. 639-648, 1983.

[3] Basili, V., Briand, L., Melo, W., "A Validation of Object Oriented Design Metrics as Quality Indicators," *IEEE Transactions on Software Engineering*, Vol. 22, No. 10, pp. 751 - 761, 1996.

[4] Basili, V., Perricone, B., "Software Errors and Complexity : An Empirical Investigation," *Communications of the ACM*, Vol. 27, No. 1, pp. 42-52, 1984.

[5] Basili, V., Shull, F., Lanubile, F., "Building Knowledge through Families of Experiments," *IEEE Transactions on Software Engineering*, Vol. 25, No. 4, pp. 456-473, 1999.

[6] Basili, V., Weiss, D.M., "A Methodology for Collecting Valid Software Engineering Data," *IEEE Transactions on Software Engineering*, Vol. 10, No. 6, pp. 728-738, 1984.

[7] Bastani, F. B., Ramamoorthy, C.V., "Input-domain-based models for estimating the correctness of process control programs," in *Reliability Theory*, A. Serra, Barlow, R.E., Ed. Amsterdam: North-Holland, 1986, pp. 321-378.

[8] Beck, K., *Extreme Programming Explained, Embrace Change*. Boston: Addison Wesley, 2000.

[9] Beck, K., *Test Driven Development: By Example*. Boston: Addison-Wesley, 2003.

[10] Boehm, B. W., *Software Engineering Economics*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1981.

[11] Brace, N., Kemp, R., Snelgar, R., *SPSS for Psychologists*: Palgrave Macmillan, 2003.

[12] Briand, L. C., Thomas, W.M., Hetmanski, C.J., "Modeling and Managing Risk Early in Software Development," Proceedings of International Conference on Software Engineering, 1993, pp. 55-65.

[13] Briand, L. C., Wuest, J., Daly, J.W., Porter, D.V., "Exploring the Relationship between Design Measures and Software Quality in Object Oriented Systems," *Journal of Systems and Software*, Vol. 51, No. 3, pp. 245-273, 2000.

[14] Briand, L. C., Wuest, J., Ikonomovski, S., Lounis, H., "Investigating Quality Factors in Object-Oriented Designs : An Industrial Case Study," Proceedings of International Conference on Software Engineering, 1999, pp. 345-354.

[15] Brito e Abreu, F., "The MOOD Metrics Set," Proceedings of ECOOP '95 Workshop on Metrics, 1995.

[16] Burbeck, S. L., "Real-time complexity metrics for Smalltalk methods," *IBM Systems Journal*, Vol. 35, No. 2, pp. 204-226, 1996.

[17] Chidamber, S. R., Kemerer, C.F., "A Metrics Suite for Object Oriented Design," *IEEE Transactions on Software Engineering*, Vol. 20, No. 6, pp. 476 - 493, 1994.

[18] Chulani, S., Boehm, B., Steece, B., "Bayesian Analysis of Empirical Software Engineering Cost Models," *IEEE Transactions on Software Engineering*, Vol. 25, No. 4, pp. 573-583, 1999.

[19] Churcher, N. I. and M. J. Shepperd, "Comments on 'A Metrics Suite for Object-Oriented Design'," *IEEE Transactions on Software Engineering*, Vol. 21, No. 3, pp. 263-5, 1995.

[20] Crow, L. H., Singpurwalla, N.D., "An Empirically Developed Fourier Series Model for Describing Software Failures," *IEEE Transactions on Reliability*, Vol. 33, No. 2, pp. 176-183, 1984.

[21] Daskalantonakis, M. K., "A Practical View of Software Measurement and Implementation Experiences within Motorola," *IEEE Transactions on Software Engineering*, Vol. 18, No. 11, pp. 998-1010, 1992.

[22] Davidsson, M., Zheng, J., Nagappan, N., Williams, L., Vouk, M., "GERT: An Empirical Reliability Estimation and Testing Feedback Tool," Proceedings of International Symposium on Software Reliability Engineering, St. Malo, France, 2004, pp. 269-280.

[23] Denaro, G., "Estimating software fault-proneness for tuning testing activities," Proceedings of International Conference on Software Engineering, 2000, pp. 704-706.

[24] Denaro, G., Morasca, S., Pezze, M., "Deriving Models of Software Fault-Proneness," Proceedings of Software Engineering Knowledge Engineering, 2002, pp. 361-368.

[25] Denaro, G., Pezze, M., "An Empirical Evaluation of Fault-Proneness Models," Proceedings of International Conference on Software Engineering, 2002, pp. 241 - 251.

[26] Duane, J. T., "Learning Curve Approach to Reliability Monitoring," *IEEE Transactions on Aerospace*, Vol. 2, No. 2, 1964.

[27] Duran, J. W., Wiorkowaski, J.J., "Capture-recapture sampling for estimating software error content," *IEEE Transactions on Software Engineering*, Vol. 7, No. 1, pp. 147-148, 1981.

[28] Ehrenberger, W., "Statistical Testing of Real-Time Software," in *Verification and Validation of Real-Time Software*, W. Quirk, J., Ed. New York: Springer-Verlag, 1985.

[29] El Emam, K., "A Methodology for Validating Software Product Metrics," National Research Council of Canada, Ottawa, Canada NCR/ERC-1076, 2000.

[30] El Emam, K., Benlarbi, S., Goel, N., Rai, S.N., "The Confounding Effect of Class Size on the Validity of Object-Oriented Metrics," *IEEE Transactions on Software Engineering*, Vol. 27, No. 7, pp. 630 - 650, 2001.

[31] Fenton, N., Neil, M., "A critique of software defect prediction models," *IEEE Transactions on Software Engineering*, Vol. 25, No. 5, pp. 675-689, 1999.

[32] Fenton, N. E., Ohlsson, N., "Quantitative analysis of faults and failures in a complex software system," *IEEE Transactions on Software Engineering*, Vol. 26, No. 8, pp. 797-814, 2000.

[33] Fenton, N. E., Pfleeger, S.L., *Software Metrics*. Boston, MA: International Thompson Publishing, 1997.

[34] Frankl, P. G., Hamlet, R.G., Littlewood, B., Strigini, L., "Evaluating testing methods by delivered reliability," *IEEE Transactions on Software Engineering*, Vol. 24, No. 8, pp. 586-601, 1998.

[35] Goel, A., Okumoto, K., "Time-Dependant Error-Detection Rate Model for Software Reliability and other Performance Measures.," *IEEE Transactions on Reliability*, Vol. 28, No. 3, pp. 206-211, 1979.

[36] Grady, R. B., Caswell, D.L., *Software Metrics: Establishing a company-wide program*. Englewood Cliffs, NJ: Prentice Hall, 1986.

[37] Halstead, M. H., *Elements of Software Science*. New York: Elsevier North Holland, 1977.

[38] Hamlet, D., Voas J., "Faults on Its Sleeve: Amplifying Software Reliability Testing," Proceedings of International Symposium on Software Testing and Analysis, Cambridge, MA, 1993, pp. 89-98.

[39] Harrison, R., S. J. Counsell, and R. V. Nithi, "An Evaluation of the MOOD Set of Object-Oriented Software Metrics," *IEEE Transactions on Software Engineering*, Vol. 24, No. 6, pp. 491-496, June 1998.

[40] Hatton, L., "Reexamining the Fault-Density - Component Size Connection," *IEEE Software*, Vol. 14, No. 2, pp. 89-97, March/April 1997.

[41] Henry, S. M., Kafura, D., "Software Structure Metrics based on Information Flow," *IEEE Transactions on Software Engineering*, Vol. 7, No. 5, pp. 510-518, 1981.

[42] Hudepohl, J., Aud, S.J., Khoshgoftaar, T.M., Allen, E.B., Mayrand, J., "Emerald: Software Metrics and Models on the Desktop," *IEEE Software*, Vol. 13, No. 5, pp. 56-60, 1996.

[43] IEEE, "Standard 610.12-1990, IEEE Standard Glossary of Software Engineering Terminology," 1990.

[44] ISO/IEC, "DIS 14598-1 Information Technology - Software Product Evaluation," 1996.

[45] Jacoby, R., Masuzawa, K., "Test coverage dependent software reliability estimation by the HGD model," Proceedings of International Symposium on Software Reliability Engineering, 1992, pp. 193-204.

[46] Jelinksi, Z., Moranda, P.B., "Software reliability research," in *Statistical Computer Performance Evaluation*, W. Freiberger, Ed. New York: Academic press, 1972, pp. 465-497.

[47] Kaiser, H. F., "An Index of Factorial Simplicity," *Psychometrika*, Vol. 39, pp. 31-36, 1974.

[48] Kan, S. H., *Metrics and Models in Software Quality Engineering*. Reading, MA: Addison-Wesley, 1995.

[49] Kemerer, C. F., "An empirical validation of software cost estimation models," *Communications of the ACM*, Vol. 30, No. 5, pp. 416-429, 1987.

[50] Khoshgoftaar, T. M., Munson, J.C.,, "Predicting software development errors using software complexity metrics," *IEEE Journal on Selected Areas in Communications*, Vol. 8, No. 2, pp. 253-261, 1990.

[51] Khoshgoftaar, T. M., Munson, J.C., Lanning, D.L., "A Comparative Study of Predictive Models for Program Changes During System Testing and Maintenance," Proceedings of International Conference on Software Maintenance, 1993, pp. 72-79.

[52] Khoshgoftaar, T. M., Seliya, N., "Fault Prediction Modeling for Software Quality Estimation: Comparing Commonly Used Techniques," *Empirical Software Engineering*, Vol. 8, No. 3, pp. 255-283, 2003.

[53] Kleinbaum, D. G., Kupper, L.L., Muller, K.E., *Applied Regression Analysis and Other Multivariable Methods*. Boston: PWS-KENT Publishing Company, 1987.

[54] Langberg, N., Singpurwalla, N.D., "A unification of some software reliability models," *SIAM Journal of Scientific and Statistical Computation*, Vol. 6, No. 3, pp. 781-790, 1985.

[55] Lipow, M., "Number of faults per line of code," *IEEE Transactions on Software Engineering*, Vol. 8, No. 4, pp. 437-440, 1982.

[56] Littlewood, B., Verrall, J.L., "A Bayesian reliability growth model for computer software," *Applied Statistics*, Vol. 22, pp. 332-346, 1973.

[57] Liu, G., "A Bayesian assessing method of software reliability growth.," in *Reliability Theory and Applications*, S. Osaki, Cao, J., Ed. Singapore: World Scientific, 1987, pp. 237-244.

[58] McCabe, T. J., "A Complexity Measure," *IEEE Transactions on Software Engineering*, Vol. 2, No. 4, pp. 308-320, 1976.

[59] Miller, K. W., Morell, L.J., Noonan, R.E., Park, S.K., Nicol, D.M., Murrill, B.W., Voas, J.M., "Estimating the Probability of Failure When Testing Reveals No Failures," *IEEE Transactions on Software Engineering*, Vol. 18, No. 1, pp. 33 - 43, 1992.

[60] Mohagheghi, P., Conradi, R., Killi, O.M., Schwarz, H., "An Empirical Study of Software Reuse vs. Reliability and Stability," Proceedings of International Conference on Software Engineering, 2004, pp. 282-292.

[61] Moller, K.-H., Paulish, D.J., "An Empirical Investigation of Software Fault Distribution," Proceedings of IEEE International Software Metrics Symposium, 1993, pp. 82-90.

[62] Munson, J. C., Khoshgoftaar, T.M., "The Detection of Fault-Prone Programs," *IEEE Transactions on Software Engineering*, Vol. 18, No. 5, pp. 423-433, 1992.

[63] Munson, J. C., Khoshgoftaar,T.M., "Regression Modeling of Software quality : Empirical Investigation," *Information and Software Technology*, Vol. 32, No. 2, pp. 106-114, 1990.

[64] Musa, J., *Software Reliability Engineering*: McGraw-Hill, 1998.

[65] Musa, J., "A Theory of Software Reliability and its Applications," *IEEE Transactions on Software Engineering*, Vol. 1, No. 3, pp. 312-327, 1975.

[66] Musa, J., Ianino, A., Okumoto, K., *Software Reliability:Measurement, Prediction, Application*. New York: McGraw Hill, 1987.

[67] Musa, J., Okumoto, K., "A Comparison of Time Domains for Software Reliability Models," *Journal of Systems and Software*, Vol. 4, No. 4, pp. 277-287, 1984.

[68] Nagappan, N., Sherriff, M., Williams, L., "On the Feasibility of Using Operational Profiles to Determine Software Reliability in Extreme Programming," North Carolina State University CSC-TR-2003-15, 2003.

[69] Nagappan, N., Williams, L., Vouk M.A., ""Good Enough" Software Reliability Estimation Plug-in for Eclipse," Proceedings of IBM-ETX Workshop, in conjunction with OOPSLA 2003, 2003, pp. 36-40.

[70] Nagappan, N., Williams, L., Vouk M.A., "Towards a Metric Suite for Early Software Reliability Assessment," Proceedings of International Symposium on Software Reliability Engineering, FastAbstract, Denver,CO, 2003, pp. 238-239.

[71] Nagappan, N., Williams, L., Vouk M.A., Osborne, J., "Using In-Process Testing Metrics to Estimate Software Reliability: A Feasibility Study," Proceedings of IEEE International Symposium on Software Reliability Engineering, FastAbstract, Saint Malo, France, 2004, pp. 21-22.

[72] Nakagawa, Y., Hanata, S., "An error complexity model for software reliability measurement," Proceedings of International Conference on Software Engineering, 1989, pp. 230-236.

[73] Nelson, E. C., "Estimating software reliability from test data," *Microelectronics and Reliability*, Vol. 17, No., pp. 67-74, 1978.

[74] NIST/SEMATECH, *e-Handbook of Statistical Methods*: http://www.itl.nist.gov/div898/handbook/.

[75] Ohlsson, M. C., Wohlin, C., "Identification of Green, Yellow and Red Legacy Components," Proceedings of International Conference on Software Maintenance, 1998, pp. 6-15.

[76] Ostrand, T. J., Weyuker, E.J., "The Distribution of Faults in a Large Industrial Software System," Proceedings of ACM International Symposium on Software Testing and Analysis, 2002, pp. 55-64.

[77] Peng, W. W., Wallace, D. R., *Software Error Analysis*. Summit, NJ: Silicon Press, 1994.

[78] Rivers, A. T., Vouk, M.A., "Resource-Constrained Non-Operational Testing of Software," Proceedings of International Symposium on Software Reliability Engineering, Paderborn, Germany, 1998, pp. 154-163.

[79] Rosenblum, D. S., "A practical approach to programming with assertions," *IEEE Transactions on Software Engineering*, Vol. 21, No. 1, pp. 19-31, 1995.

[80] Schick, G. J., Wolverton, R.W., "An Analysis of Competing Software Reliability Models," *IEEE Transactions on Software Engineering*, Vol. 4, No. 2, pp. 104-120, 1978.

[81] Schneider, V., "Some experimental estimators for developmental and developed errors in software development projects," Proceedings of the ACM Workshop/Symposium on Measurement and Evaluation of Software Quality, 1981, pp. 169-171.

[82] Schneidewind, N. F., "An integrated process and product model," Proceedings of Fifth International Software Metrics Symposium, 1998, pp. 224-234.

[83] Schneidewind, N. F., "Software metrics model for quality control," Proceedings of Fourth International Software Metrics Symposium, 1997, pp. 127-136.

[84] Shantikumar, J. G., "A General Software Reliability Model for Performance Prediction," *Microelectronics and Reliability*, Vol. 21, No. 5, pp. 671-682, 1981.

[85] Shepperd, M., Ince, D., *Derivation and Validation of Software Metrics*: Oxford University Press, 1993.

[86] Subramanyam, R., Krishnan, M.S., "Empirical Analysis of CK Metrics for Object-Oriented Design Complexity: Implications for Software Defects," *IEEE Transactions on Software Engineering*, Vol. 29, No. 4, pp. 297 - 310, 2003.

[87] Tahoma, Y., Tokunaga, K., Nagase, S., Murata, Y., "Structural Approach to the Estimation of the Number of Residual Software Faults Based on the Hyper-Geometric Distribution," *IEEE Transactions on Software Engineering*, Vol. 15, No. 3, pp. 345-362, 1989.

[88] Tang, M.-H., Kao, M-H., Chen, M-H., "An empirical study on object-oriented metrics," Proceedings of Sixth International Software Metrics Symposium, 1999, pp. 242-249.

[89] Thayer, R., Lipow, M., Nelson, E., *Software Reliability*. Amsterdam: North-Holland, 1978.

[90] Thomson, W. E., Chelson, P.O., "On the specification and testing of software reliability," Proceedings of Annual Reliability and Maintainability Symposium, 1980, pp. 379-383.

[91] Troster, J., "Assessing Design-Quality Metrics on Legacy Software," Software Engineering Process Group, IBM Canada Ltd. Laboratory, North York, Ontario 1992.

[92] Vouk, M. A., Jones, W., "Software Reliability Field Data Analysis," in *Handbook of Software Reliability Engineering*, M. Lyu, Ed.: McGraw Hill, 1996.

[93] Vouk, M. A., Tai, K.C., "Multi-Phase Coverage- and Risk-Based Software Reliability Modeling," Proceedings of CASCON '93, 1993, pp. 513-523.

[94] Weiss, S. N., Weyuker, E.J., "An Extended Domain-Bases Model of Software Reliability," *IEEE Transactions on Software Engineering*, Vol. 14, No. 10, pp. 1512-1524, 1988.

[95] Whittaker, J. A., "Markov Chain Techniques for Software Testing and Reliability Analysis, PhD Dissertation," *Department of Computer Science, University of Tenn.*, Vol., No., 1992.

[96] Wohlin, C., Korner, U., "Software Faults : Spreading, Detection and Costs," *Software Engineering Journal*, Vol. 5, No. 1, pp. 38-42, 1990.

[97] Xie, M., *Software Reliability Modeling*. Singapore: World Scientific Publishing Co. Pte. Ltd., 1991.

[98] Yamada, S., Osaki, S., "Software Reliability Growth Modeling: Models and Applications," *IEEE Transactions on Software Engineering*, Vol. 11, No. 12, pp. 1431-1437, 1985.

Appendix A – Data sets

Table A.1: Academic case study data set

SM1	SM2	SM3	SM4	SM5	SM6	SM7	SM8	SM9	TRs/ KLOC
0.072419	0.07319	0.989474	0.416084	1.270539	0.613636	2.047619	0.398693	2.103728	12.32666
0.025062	0.00358	7	0.72093	0.426019	0.719101	0.9375	1	5.884927	1.652437
0.01506	0.009639	1.5625	0.364621	0.924269	0.301887	0.8	0.396296	2.690438	7.228916
0.061195	0.021855	2.8	0.579399	0.792186	0.538462	0.666667	0.72449	3.337115	1.457018
0.112583	0.040287	2.794521	0.145374	0.671358	0.210526	0.769231	0.38	2.936791	3.311258
0.025751	0.025751	1	0.113485	0.682403	0.140845	0.294479	0.127094	1.888169	18.88412
0.083912	0.03581	2.343284	0.248815	0.908605	0.176471	0.4	0.419087	3.032415	5.344735
0.022245	0.021218	1.048387	0.06383	0.528747	0.24	0.411765	0.064865	4.735818	8.555784
0.190463	0.059109	3.222222	0.307851	1.811822	0.235294	3.8	0.283433	5.675851	1.427756
0.025	0.022308	1.12069	0.35786	0.437308	0.329268	0.444444	0.405858	4.213938	1.923077
0.039487	0.040513	0.974684	0.125	0.571874	0.086957	0.307692	0.20202	3.160454	6.666667
0.103058	0.050396	2.044944	0.408907	1.01359	0.211765	0.48	0.46087	2.862237	7.92752
0.062218	0.060211	1.033333	0.316384	0.466006	0.169014	0.5	0.39375	3.230146	8.529854
0.007067	0.004122	1.714286	0.273723	0.201413	0.188119	0.45	0.38191	2.752026	5.889282
0.052908	0.02104	2.514706	0.319149	0.679146	0.21875	2.4	0.580153	5.23825	4.022277
0.059968	0.042139	1.423077	1.181518	0.431767	1.584416	1.8	0.892157	1	35.6564
0.042157	0.027588	1.52809	0.47973	0.389182	0.320755	0.263158	0.33913	5.228525	1.549907
0.045518	0.022409	2.03125	0.190476	0.571822	0.15625	0.666667	0.355769	2.314425	2.10084
0.190793	0.017565	10.86207	0.319703	0.419594	0.438356	0.5	0.448454	2.675851	3.634161
0.031097	0.030278	1.027027	0.37037	0.545008	0.369565	1.190476	0.852273	1.980551	13.91162
0.067829	0.043282	1.567164	0.941176	0.608312	0.351064	0.6	1.354497	2.508914	5.167959
0.027778	0.022222	1.25	0.175711	0.773148	0.166667	0.333333	0.304878	1.750405	10.18519

Table A.2: Open source case study data set

SM1	SM2	SM3	SM4	SM5	SM6	SM7	SM8	SM9	TRs/ KLOC
0.055602	0.054349	1.023059	0.319667	0.859121	0.607487	0.534314	0.308992	21.881	0.710168
0.032401	0.022159	1.462222	0.155299	1.270924	0.179862	0.070686	0.145636	9.282	0.984834
0.077072	0.147949	0.520935	0.694059	1.640944	0.61275	0.534	0.53862	11.362	0.643604
0.011397	0.003177	3.587629	0.125165	0.269654	0.098648	0.246886	0.134135	27.911	0.261994
0.020498	0.01064	1.926471	0.101405	0.910326	0.143173	0.118299	0.114993	23.367	1.682119
0.14775	0.055443	2.664921	0.659643	0.598961	0.82906	0.721805	0.707219	3.149	2.03193
0.148361	0.098659	1.503776	0.595318	1.244509	1.045028	0.576369	0.585379	12.267	2.384501
0.073409	0.088209	0.832215	0.226997	3.303744	0.411226	0.146104	0.218008	9.264	1.381352
0.10863	0.050388	2.155867	0.286179	1.857822	0.524051	0.66263	0.283009	10.358	2.117896
0.076912	0.04131	1.861831	0.34194	0.997198	0.35492	0.523077	0.305928	12.812	0.642123
0.134711	0.138715	0.971139	0.266631	0.595705	0.485356	1	0.258913	8.448	2.055832
0.031641	0.082172	0.385057	0.345133	1.185235	0.486154	0.2287	0.250254	3.871	1.652893
0.081526	0.562012	0.145062	0.832558	1.166894	0.873874	0.717949	0.910714	1.054	6.938422
0.09146	0.302223	0.302625	0.501784	1.334559	0.733911	0.452012	0.368778	12.213	0.673602
0.027399	0.024707	1.108937	0.093508	0.64379	0.294118	0.147216	0.092116	69.96	0.261315
0.005519	0.004821	1.144928	0.039031	0.172479	0.049563	0.186869	0.047142	26.167	0.209592
0.023736	0.140639	0.16877	0.294632	2.132135	0.141527	0.119617	0.229282	8.241	1.220053
0.239488	0.209324	1.144105	0.477679	0.874162	0.571429	0.464286	0.48249	1	0.914077
0.258738	0.317397	0.81519	0.652941	0.691388	0.5	1.289474	0.496523	2.275	1.205303
0.181868	0.205638	0.884409	0.393939	0.572576	0.774194	1.526316	0.318182	1.654	2.763958
0.013482	0.036382	0.370583	0.072683	0.45747	0.287869	0.599688	0.121825	30.577	0.388628
0.001651	0.001351	1.222222	0.027144	0.405223	0.044061	0.028112	0.029683	6.09	0.60033
0.0852	0.114835	0.741935	0.110818	1.332436	0.207127	0.212963	0.133277	5.182	0.705592
0.02584	0.036822	0.701754	0.095238	0.399655	0.192593	0.68	0.120846	1.415	1.29199
0.0176	0.161736	0.108818	0.2566	1.134122	0.313576	0.18797	0.381517	6.025	0.75861
0.171854	0.441137	0.389571	0.625709	1.077885	0.772093	0.698113	0.403027	2.702	2.368065
0.077032	0.053366	1.443459	0.525633	1.024454	0.591343	0.641115	0.699861	15.45	1.597444

Appendix B – Tool support

Tool support for partial collection of the metrics used in the STREW metric suite Version. 1.4 is performed using an open source plug-in GERT[10] (Good Enough Reliability Tool) under the Common Public License (CPL[11]) for the open source Eclipse[12] development environment. Figure B.1 is a snap shot of the tool in use with different functionalities highlighted.

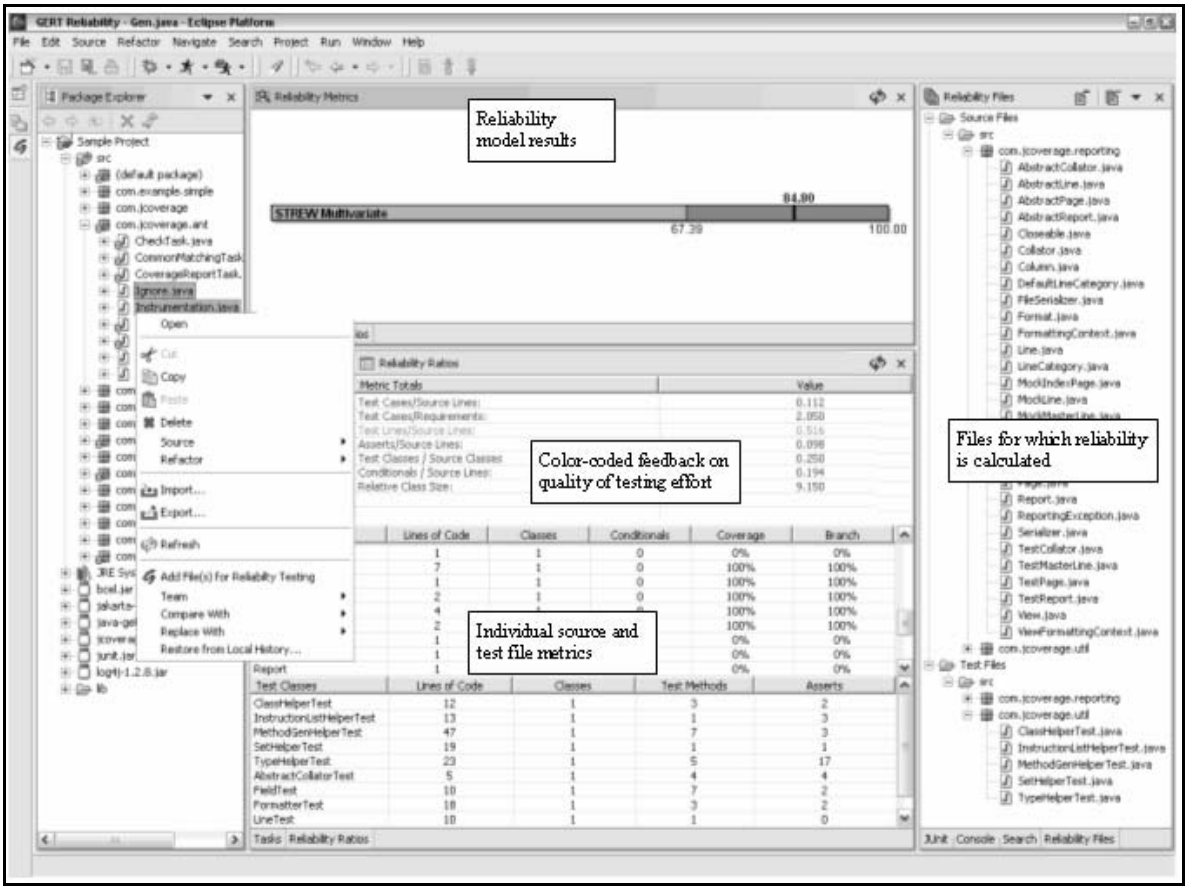


Figure B.1: GERT snap shot

All preferences for the tool are editable through a sub-page of the Eclipse IDE preference pages. These settings are retained throughout subsequent sessions as is the layout of the tool's different graphical views. The configurations available consist of defining the parameters for the STREW

[10] GERT can be obtained from http://gert.sourceforge.net. GERT was a winner in the International Challenge for Eclipse competition in the student project category

[11] http://www-124.ibm.com/developerworks/oss/CPLv1.0.htm

[12] Eclipse is an open source integrated development environment. For more information see http://www.eclipse.org

model. If the user wishes to employ a different reliability model, the expression of metrics that determine the point estimate can be entered among these preferences. Optionally, the colors used to highlight the "Reliability Ratios" view and their associated ranges are defined from the same menu.

GERT provides an easy-to-use tool for empirical reliability estimation and test feedback. Some of GERT's functionality is handled by other open source tools that have been incorporated in GERT's source code. Currently, the task of performing coverage analysis and administrating unit testing is handled by JCoverage and JUnit, respectively.

- JCoverage, licensed under the GNU General Public License (GPL[13]) is an extension to the Apache Ant build tool.
- JUnit, licensed in similar fashion under the CPL, provides a framework for running unit testing. GERT calculates coverage based upon JUnit test cases.

From the home page of GERT (gert.sourceforge.net) we present some statistics about the download rate and the activity profile over the past nine months[14] (274 days). Figure B.2 presents the usage statistics from sourceforge.net.

[13] http://www.gnu.org/copyleft/gpl.html
[14] The statistics snapshot was taken on November 11 2004.

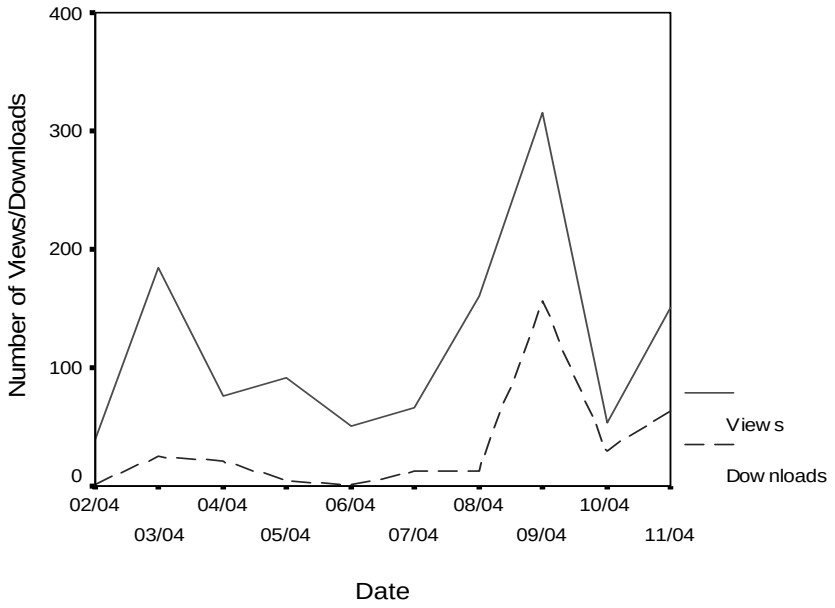


Figure B.2: Usage Statistics (Courtesy sourceforge.net)

Table B1 shows the number of downloads (D/L) and the rank of the GERT tool in sourceforge.net on a monthly basis.

Table B.1: Statistics for the past 10 months (Courtesy sourceforge.net)

Month	Rank	Page Views	D/l
November 2004	7764 (55.15)	150	64
October 2004	11282 (34.62)	54	30
September 2004	5793 (65.44)	316	156
August 2004	11022 (38.42)	161	13
July 2004	7181 (58.30)	66	12
June 2004	7294 (60.36)	51	1
May 2004	10814 (37.40)	92	5
April 2004	10906 (33.42)	76	21
March 2004	7530 (54.06)	184	25
February 2004	3042 (80.88)	39	2

Table B.2 shows a cumulative number of downloads over the entire life time of the tool and its current rank with respect to other products in souceforge.net.

Table B.1: Statistics for All Time (Courtesy sourceforge.net)

Lifespan	Rank	Page Views	D/l
276 days	8263 (51.81)	1,189	329

Appendix C

Miscellaneous Statistical Analysis

Appendix C details some of the statistical analysis that form the underlying criterion for the analysis explained in this dissertation. We assess the normality of the STREW metrics, and the principal components produced by the STREW metrics in Appendix C.

Table C.1 and C.2 are used to check for the normality of data distribution for application of equation 1. The results indicate that the STREW metrics are normally distributed.

Table C.1: One-Sample Kolmogorov-Smirnov Test – Academic Projects

	SM1	SM2	SM3	SM4	SM5	SM6	SM7	SM8
Kolmogorov-Smirnov Z Statistic	.890	.692	1.349	.880	1.049	1.182	1.306	1.241
Asymp. Sig. (2-tailed)	.406	.724	.053	.421	.221	.122	.066	.092

Table C.2: One-Sample Kolmogorov-Smirnov Test – 27 Open Source Projects

	SM1	SM2	SM3	SM4	SM5	SM6	SM7	SM8
Kolmogorov-Smirnov Z Statistic	.860	.957	.741	.933	.611	.551	.811	.706
Asymp. Sig. (2-tailed)	.450	.318	.642	.348	.850	.922	.526	.700

In statistical studies, sometimes it is possible that the distribution of data points may be bimodal. This can be avoided by using dependent variable residuals from the regression equation in the test for normality. The plot of the residuals for the academic case study is shown in Figure C.1. We also ran a one sample Kolmogorov-Smirnov test on the unstandardized regression predicted residuals with the null hypothesis that the population distribution is normal. The results ($Z=0.656$, $p=0.782$) indicate that we can accept the null hypothesis that the distribution is normal.

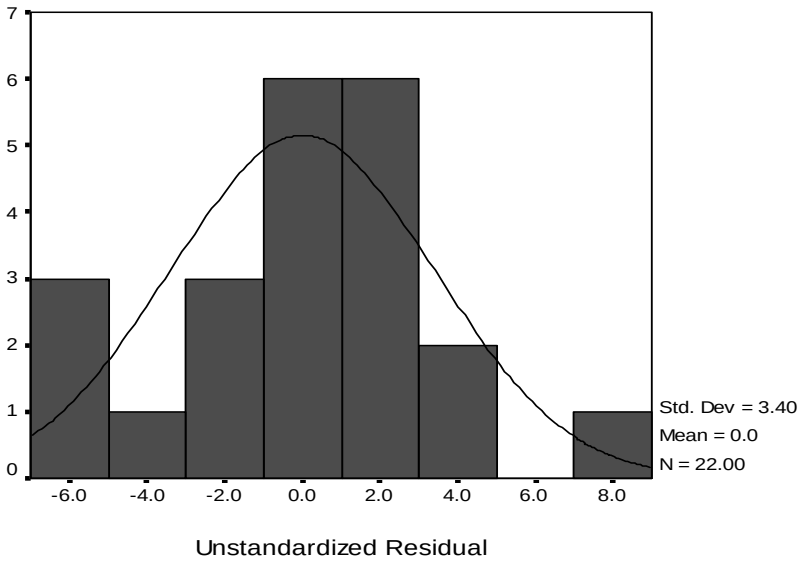


Figure C.1 : Normality plot of regression residuals – Academic case study

Similarly for the 27 open source projects the results of a Kolmogorov-Smirnov test on the unstandardized regression predicted residuals yielded (Z=0.656, p=0.782) indicating that the distribution was normal. The normality plots are shown in Figure C.2.

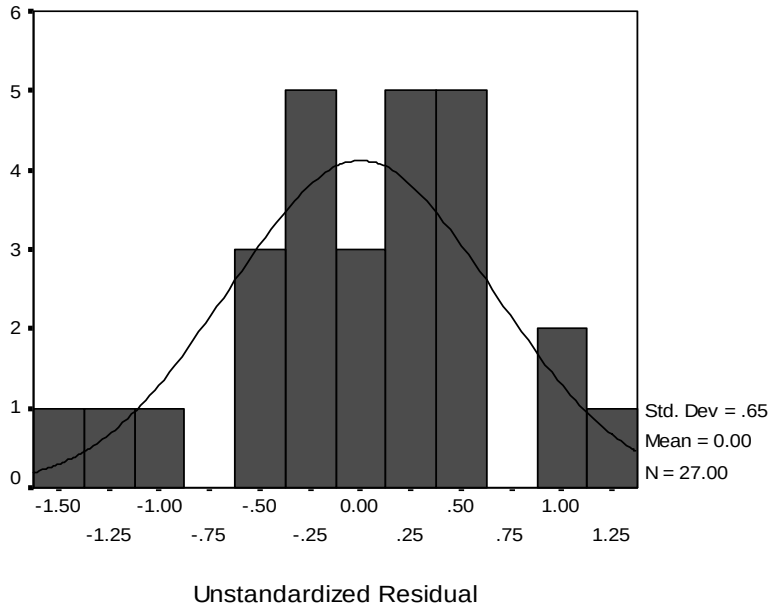


Figure C.2: Normality plot of regression residuals –Open source case study

We present more information on the principal component analysis in Figure C.3 - the scree plot and Figure C.4 - the component plot of the academic projects. The scree plot is useful in determining the number of principal components that should be extracted.

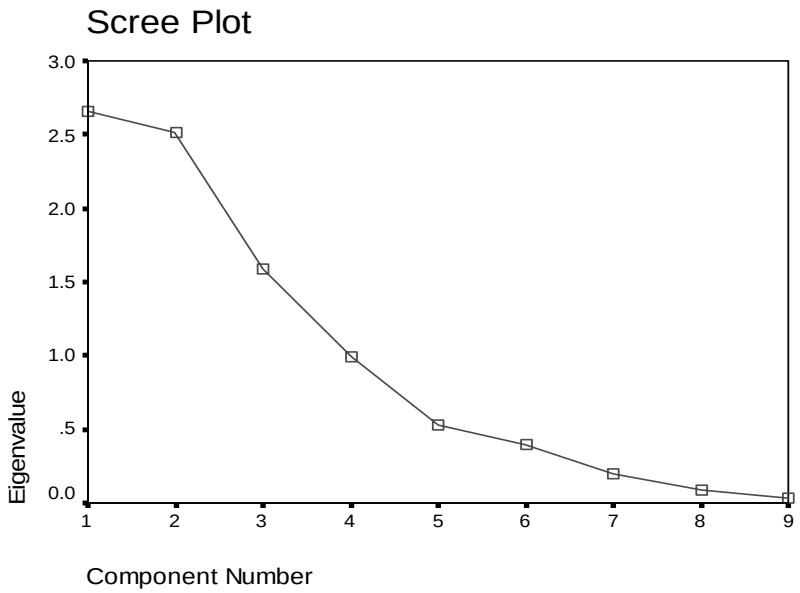


Figure C.3: Scree plot

The component plot shows the location of the individual metrics in terms of the loaded components to identify relationships between the elements in terms of the principal components.

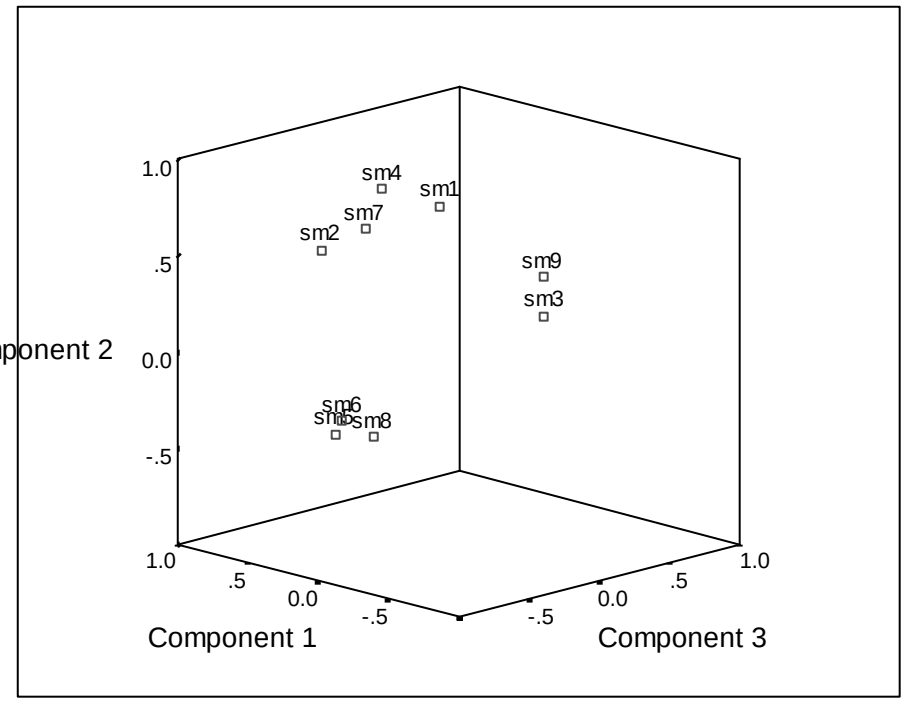


Figure C.4: Component plot

Similarly we present in Figure C.5 - the scree plot and Figure C.6 - the component plot of the open source projects.

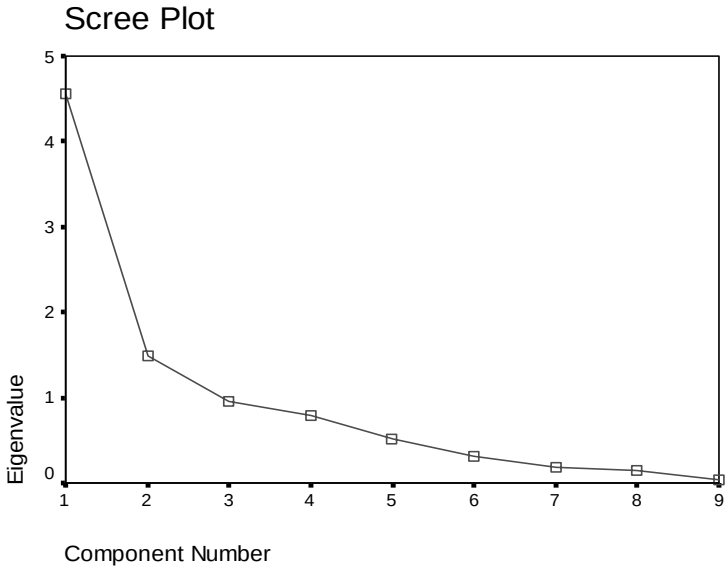


Figure C.5: Scree plot

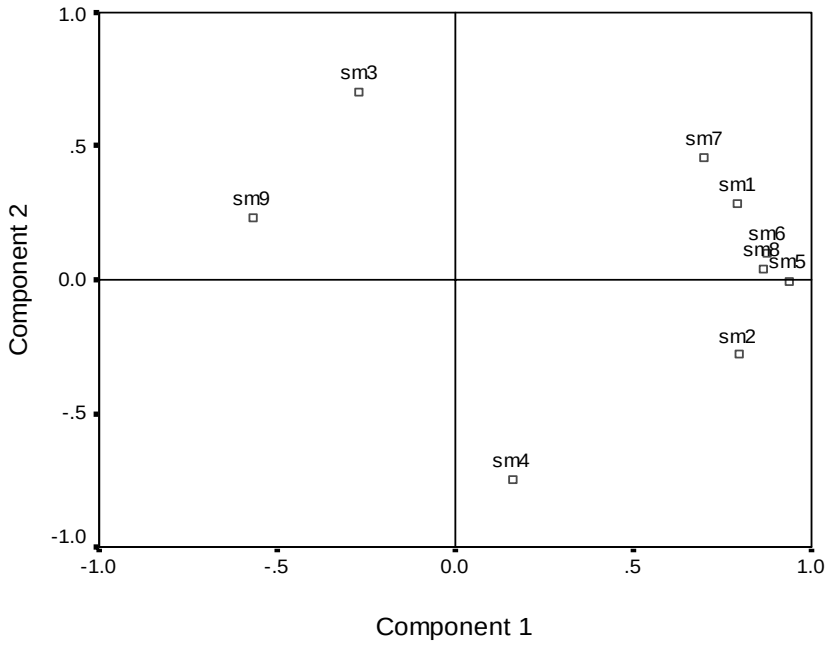


Figure C.6: Component plot

Table C.3 shows the association between project metrics and TRs without the normalization of metrics for the academic projects. From Table C.3 we observe that none of the metrics(shown in bold) are correlated with the TRs, indicating the absence of any relationship between the number of TRs and the individual project metrics.

Table C.3: Correlation matrix with individual project metrics - Academic

		TRs	SLOC	CCS	CBOS	DITS	WMPCS	Asserts	Test Cases
TRs	**Pearson Correlation**	**1**	**-.335**	**.167**	**.062**	**.296**	**.308**	**-.369**	**-.089**
	Sig. (2-tailed)	**.**	**.127**	**.459**	**.783**	**.181**	**.164**	**.091**	**.694**
SLOC	Pearson Correlation	-.335	1	-.216	.056	-.114	-.208	.459	.424
	Sig. (2-tailed)	.127	.	.334	.806	.613	.354	.032	.049
CCS	Pearson Correlation	.167	-.216	1	.304	.695	.870	.186	.026
	Sig. (2-tailed)	.459	.334	.	.169	.000	.000	.407	.909
CBOS	Pearson Correlation	.062	.056	.304	1	.238	.121	.065	.003
	Sig. (2-tailed)	.783	.806	.169	.	.287	.591	.773	.991
DITS	Pearson Correlation	.296	-.114	.695	.238	1	.810	-.161	-.229
	Sig. (2-tailed)	.181	.613	.000	.287	.	.000	.475	.304
WMPCS	Pearson Correlation	.308	-.208	.870	.121	.810	1	.160	.106
	Sig. (2-tailed)	.164	.354	.000	.591	.000	.	.476	.638
Asserts	Pearson Correlation	-.369	.459	.186	.065	-.161	.160	1	.762
	Sig. (2-tailed)	.091	.032	.407	.773	.475	.476	.	.000
Test Cases	Pearson Correlation	-.089	.424	.026	.003	-.229	.106	.762	1
	Sig. (2-tailed)	.694	.049	.909	.991	.304	.638	.000	.

Similarly Table C.4 shows the association between project metrics and TRs without the normalization of metrics for the open source projects. From Table C.4 we observe that except the test cases all the other metrics are correlated with the TRs at statistically significant levels implying that with an increase in these metrics there is an increase in the TRs. This could imply that for the academic projects the lack of correlation might be due to the large variability of the TRs in the projects.

Table C.4: Correlation matrix with individual project metrics – Open source

		TR's	SLOC	CCS	WMCS	CBOS	DITS	Asserts	Test Cases
TR's	**Pearson Correlation**	**1**	**.393**	**.381**	**.395**	**.374**	**.431**	**.593**	**.131**
	Sig. (2-tailed)	**.**	**.043**	**.050**	**.041**	**.055**	**.025**	**.001**	**.515**
SLOC	Pearson Correlation	.393	1	.965	.962	.966	.854	.512	.243
	Sig. (2-tailed)	.043	.	.000	.000	.000	.000	.006	.223
CCS	Pearson Correlation	.381	.965	1	.934	.917	.838	.474	.231
	Sig. (2-tailed)	.050	.000	.	.000	.000	.000	.013	.247
WMCS	Pearson Correlation	.395	.962	.934	1	.976	.863	.619	.321
	Sig. (2-tailed)	.041	.000	.000	.	.000	.000	.001	.102
CBOS	Pearson Correlation	.374	.966	.917	.976	1	.903	.530	.241
	Sig. (2-tailed)	.055	.000	.000	.000	.	.000	.005	.227
DITS	Pearson Correlation	.431	.854	.838	.863	.903	1	.526	.230
	Sig. (2-tailed)	.025	.000	.000	.000	.000	.	.005	.249
Asserts	Pearson Correlation	.593	.512	.474	.619	.530	.526	1	.551
	Sig. (2-tailed)	.001	.006	.013	.001	.005	.005	.	.003
Test Cases	Pearson Correlation	.131	.243	.231	.321	.241	.230	.551	1
	Sig. (2-tailed)	.515	.223	.247	.102	.227	.249	.003	.