

## **Abstract**

With an ever-growing competition among software vendors in supplying customers with tailored, high-quality systems, an emphasis is put on creating products that are well-tested and reliable. During the last decade and a half numerous articles have been published that deal with code coverage and its effect, whether existent or not, on reliability. The last few years have also witnessed an increasing number of software tools for automating the data collection and presentation of code coverage information for applications being tested.

In this report we aim to present available and frequently used measures of code coverage, the practical applications and typical misconceptions of code coverage and its role in software development nowadays. Then we take a look at the notion of reliability in computer systems and which elements that constitute a software reliability model. With the basics of code coverage and reliability estimation in place, we try to assess the status of the relationship between code coverage and reliability, highlight the arguments for and against its existence and briefly survey a few proposed models for connecting code coverage to reliability. Finally, we examine an open-source tool for automated code coverage analysis, focusing on its implementation of the coverage measures it supports, before assessing the feasibility of integrating a proposed approach for reliability estimation into this software utility.

## **Preface**

This report is the product of a master thesis – formally referred to as “TDT4900 Datateknikk, masteroppgave” – at the Department of Computer and Information Science. The “Code Coverage and Software Reliability” project has been performed by two master students – Lars Fugelseth and Stian Frydenlund Lereng – during our 5<sup>th</sup> and final year at the Norwegian University of Science and Technology.

The assignment was proposed by professor Tor Stålhane, who gave us freedom to define the scope of the task ourselves. Having been given a bunch of IEEE articles on the relationship between code coverage and reliability to start with, we chose to compose the assignment with a basis on this particular matter. For this reason we found it necessary to take a broader look at both code coverage and reliability estimation, hence putting the relationship debate in a context. We also found it feasible to look at an automated tool for measuring code coverage and how this utility made practical use of code coverage theory in its implementation.

We would like to thank our supervisor Tor Stålhane for taking the time to listen to our ideas, for providing us with relevant information to get started, and for giving valuable feedback in the later stages.

Trondheim, June 20<sup>th</sup> 2005

---

Stian Frydenlund Lereng

---

Lars Fugelseth

# Table of Contents

CHAPTER 1: Introduction .....	1
1.1 Motivation.....	1
1.2 Project Context.....	2
1.3 Problem Definition.....	2
1.4 Project Scope .....	3
1.5 Project Goals.....	3
1.7 Report Outline.....	4
CHAPTER 2: Code Coverage .....	6
2.1 Code Coverage Basics .....	6
2.2 Code Coverage Measures .....	7
2.2.1 Essential Measures.....	7
2.2.1.1 Statement Coverage .....	7
2.2.1.2 Decision Coverage.....	8
2.2.1.3 Condition Coverage .....	9
2.2.1.4 Multiple Condition Coverage .....	9
2.2.1.5 Path Coverage .....	10
2.2.1.6 All-uses Coverage.....	11
2.2.2 Alternative Measures .....	12
2.2.3 Coverage Measure Hierarchy .....	14
2.3 Practical Code Coverage Application.....	15
2.3.1 Choosing the Right Coverage Threshold and Measures.....	16
2.3.2 Coverage Progress and Effort .....	16
2.4 Pitfalls and Misconceptions .....	17
2.4.1 Complete Coverage is No Guarantee.....	17
2.4.2 Code Coverage and the Design of Tests.....	18
2.5 Code Coverage Implementations .....	19
2.5.1 Implementation Approaches .....	19
2.5.2 Tool Features .....	20
2.6 The Effect of Test-driven Development Methodologies .....	20
2.6.1 The Role of Code Coverage.....	21
2.6.2 Object-oriented Testing .....	22
CHAPTER 3: Software Reliability Growth Models.....	24
3.1 Reliability of Computer Systems .....	24
3.1.1 Implications of Component Failures.....	24
3.2 The Importance of Software Reliability .....	27
3.3 Software Reliability Models .....	28
3.3.1 Introduction to Reliability Models.....	28
3.3.2 Reliability Model Ingredients .....	29
3.3.2.1 Time .....	29
3.3.2.2 Failures.....	30
3.3.3 Randomness of Variables .....	31
3.3.3.1 Causes of Randomness .....	31
3.3.3.2 Failure Probability Distribution .....	31

3.3.4 Operational Environment.....	33
3.3.5 Reliability Modeling.....	35
3.3.6 Parametrizing the Model of Choice.....	36
3.3.7 Model Variants.....	37
3.3.7.1 Jelinski and Moranda/Shooman (1971).....	38
3.3.7.2 Littlewood-Verrall (1973).....	38
3.3.7.3 Goel and Okumoto (1979).....	38
3.3.7.4 Basic Musa (1979).....	38
3.3.8 Reliability Model Usage.....	39
CHAPTER 4: Code Coverage and Reliability.....	40
4.1 Term Definitions.....	40
4.2 Operational Profiles.....	41
4.2.1 Purpose of Operational Profiles.....	41
4.2.2 Problems and Challenges.....	42
4.3 Reliability Overestimation.....	43
4.3.1 Saturation Effect.....	43
4.3.2 Pre-process Model for Improved Reliability Estimates.....	44
4.4 The Code Coverage – Reliability Relationship.....	45
4.4.1 Models for Relating Code Coverage to Reliability.....	46
4.4.1.1 Node-based Reliability Model.....	46
4.4.1.2 Logarithmic-exponential Model.....	48
4.4.1.3 Hyper-geometric Distribution Model.....	49
4.4.1.4 Detectability Profile Model.....	50
4.4.1.5 Influential Factors to the Coverage – Reliability Relationship.....	51
4.4.2 Experimental Results and Conclusions Reached.....	52
4.4.2.1 Code Coverage – Reliability Correlation.....	52
4.4.2.2 Fault Removal Behavior.....	54
4.4.3 Critics and Experimental Weaknesses.....	55
4.4.3.1 The Effect of Test Intensity on Reliability.....	55
4.4.3.2 Absence of Operational Profile Leads to Unweighted Testing.....	57
4.4.3.3 Inconclusive Results.....	59
CHAPTER 5: The Musa-Okumoto Model with Data Pre-processing.....	60
5.1 The Musa-Okumoto Model.....	60
5.1.1 Execution Time Component.....	61
5.1.2 Calendar Time Component.....	63
5.1.2.1 Description of Resource Measures.....	64
5.2 Code Coverage Pre-process Model.....	64
5.2.1 Model Rationale.....	65
5.2.2 Compression Ratio through Smoothing Parameters.....	65
5.2.3 Compression Ratio through Scaled Parameters.....	66
CHAPTER 6: A Software Implementation of Automated Code Coverage Analysis: JCoverage.....	67
6.1 Characteristics and Environment.....	67
6.2 JCoverage Operation.....	69
6.2.1 Instrumentation.....	69

6.2.2 Code Coverage Presentation and Computation .....	72
6.2.2.1 Coverage Reports .....	73
6.2.2.2 Computation of Coverage Measures .....	74
6.3 Suggested Improvements .....	79
6.4 An Approach to Implementing Reliability Estimation in JCoverage .	80
6.4.1 The Road to Obtaining Reliability Estimates .....	81
6.4.1.1 Code Coverage (1) .....	81
6.4.1.2 Failures (2) .....	83
6.4.1.3 Test Time (3) .....	83
6.4.1.4 Pre-process Technique (4) .....	84
6.4.1.5 Musa-Okumoto (7) .....	84
6.4.1.6 Reliability Estimates (8) .....	85
6.4.2 Solving the Calendar Time Issue .....	85
6.5 An Alternative Instrumentation Technique and Code Coverage Tool	87
6.5.1 A General Technique for Source Code Instrumentation .....	87
6.5.2 A Brief Presentation of Clover .....	88
CHAPTER 7: Conclusion .....	91
7.1 Summary .....	91
7.2 Further Work .....	93

## List of Figures

Figure 2.1	Subsumption hierarchy	14
Figure 2.2	An illustration of structural and functional testing	15
Figure 2.3	Implementation sketch	19
Figure 2.4	User-centered versus code-centered development	22
Figure 3.1	Fault tree sample	25
Figure 3.2	Example of a redundant system	26
Figure 3.3	Probability distribution for number of failures in one and five hours respectively	32
Figure 3.4	An illustration of equivalence partitioning	34
Figure 4.1	The operational profile deals with the identified input domain	41
Figure 4.2	Illustration of the saturation effect	44
Figure 4.3	Node representation of software	47
Figure 4.4	Internal validity represented as the black arrow between treatment and outcome	56
Figure 6.1	The respective roles of JCoverage, Ant and Eclipse	68
Figure 6.2	HTML page for presenting coverage measures for all packages and classes being tested	72
Figure 6.3	HTML page generated by JCoverage for presenting the source code of a tested class	73
Figure 6.4	XML version of the coverage report	74
Figure 6.5	Coverage report for testing of a sample program	75
Figure 6.6	An example highlighting the branch coverage dilemma	76
Figure 6.7	Illustration of the proposed reliability estimation approach	80
Figure 6.8	The cumulative frequency of test executions per line of code	81
Figure 6.9	Consequences of adding or modifying code	82
Figure 6.10	Sequence of modifications to code	88
Figure 6.11	HTML report generated by Clover	89

## **List of Tables**

Table 2.1	<i>Summary of essential coverage measures</i>	13
Table 2.2	<i>Facts and misconceptions of code coverage</i>	18
Table 3.1	<i>Time definitions</i>	29
Table 3.2	<i>Failure measures</i>	30
Table 3.3	<i>Alternative representations of reliability</i>	35
Table 4.1	<i>Summary of models incorporating code coverage</i>	51

*"There was no "before" the beginning of our universe,  
because once upon a time there was no time."*

John D. Barrow

## CHAPTER 1:

# Introduction

This introduction section will briefly state the main motivation and objectives related to the project we have been assigned: "Code Coverage and Software Reliability". Further we explain the project context, try to define the most important problems with respect to code coverage and reliability estimation, and highlight the direction that we have chosen within the theme. Finally we put our goals for the project into concrete terms and give a presentation of the report outline and what each chapter will contain.

## 1.1 Motivation

Creating high quality products is a necessity for modern day software vendors, and thus finding a testing strategy that will contribute to highly reliable software is a challenge for any company in the software industry. Code coverage is known to be an indirect indicator of product quality, determining to what extent the tests cover the program code to be evaluated. As opposed to measures such as test effort, code coverage directly measures how thoroughly a system has been exercised. Code coverage as a testing strategy has proven to be a feasible approach when testing applications that are logic-intensive and hence consist of multiple decision points.

As is the case with most fields of research, code coverage and its relationship to software reliability still has room for further exploration. There are plenty of publications available on code coverage as a testing strategy, and also on what implications code coverage has on the actual reliability of the end product. The different articles and their corresponding approaches and results do not only suggest that a definite conclusion to the matter is far from imminent, but also that the variation in theoretical reasoning and results obtained contributes to a blurry overall picture. With this in mind, we feel there is a definite need for creating an overview of existing approaches and theories.

In spite of several articles and empirical investigations on the subject, the seemingly never-ending hunt for perfect reliability estimates goes on. Several reliability models are referred to in software literature and we aim at getting to know a few of them and even take an in-depth dive into what is considered to be an extension to well-known reliability models such as Musa-Okumoto and Goel-Okumoto. This extension aims at pre-processing data to be used by



reliability models and thus improve the generated reliability estimates. Whereas there exist numerous software tools that can automatically track coverage measures as testing progresses, there are few, if any, tools that implement the existing reliability models to estimate end product reliability based on code coverage measures.

On a personal note, our motivation for choosing this topic comes as a result of recent projects that we have participated in, where we dealt with testing and quality assurance issues in a web context.

## 1.2 Project Context

This project was assigned to us as a master thesis by the department of software engineering at the Norwegian University of Science and Technology (NTNU). The “Code Coverage and Software Reliability” project is part of an ongoing research project titled Business-Critical Software – BUCS.

The main purpose of BUCS is to develop methods for improving support of development, operation and maintenance of business-critical systems. The project is funded by the Norwegian Research Council and is scheduled to run from 2003 to 2007. More information on BUCS can be found on the following web site: <http://www.idi.ntnu.no/grupper/su/bucs>

A time frame of 20 weeks for this thesis implies that we had to restrict our focus to chosen parts of the problem domain, and the resulting project scope is elaborated in chapter 1.4.

## 1.3 Problem Definition

The many existing coverage measures present an array of alternatives. Although this might seem like a blessing, it poses a real challenge in choosing what measure or set of measures that would paint the most realistic picture of the actual code coverage, and what set of measures would be the most representative with respect to the system’s actual reliability. Code coverage measures can be placed in a hierarchy where certain measures are said to subsume others, meaning that complete coverage of one measure implies complete coverage of the measure it subsumes. We will return to this issue in the next chapter.

Articles have been published that propose several ways of connecting code coverage to reliability, but would the implementation of an existing reliability model be feasible if one was to generate reliability estimates based on code coverage values? Software tools that automate code coverage analysis can no

longer be considered scarce products, however the people creating such tools seem reluctant to find ways of coming up with reliability estimation values. Perhaps it is merely a sign that the theoretical foundation for the coverage-reliability relationship has yet to be built, if researches will ever agree on such a foundation.

A considerable problem when it comes to existing reliability growth models for software, is that they tend to overestimate the reliability of a given program, also known as the saturation effect. Hence, several articles suggest that a test case that does not increase coverage values, and at the same time is unsuccessful in causing one or more failures, should be considered ineffective. One of the problems with code coverage is that it is destined to increase as the number of test cases applied increases, assuming that complete code coverage has yet to be reached and that no test case is repeated. Thus, since both code coverage and defect coverage increases as time passes by or test intensity increases, it is far from surprising that empirical investigations end up concluding that a relationship exists. However, it does not necessarily mean that an increase in code coverage drives the detection of new defects.

## 1.4 Project Scope

In this project we seek to perform a general and rough literature study of what has been done with regards to code coverage and its relationship to software reliability. Our hope is that such a study will provide us with the status of reliability estimation and of code coverage as a testing strategy. We intend to use the literature study as a platform for considering the feasibility of integrating reliability estimation into an existing code coverage analysis tool.

We will only focus on the relationship between code coverage and reliability, and hence neglect other related quality attributes that might prove relevant, such as availability. Neither will we attempt to look at how the total reliability of a software product can be estimated based on reliability values for individual modules or components. Although an interesting prospect, we consider the latter challenge to be outside the scope of the current project.

## 1.5 Project Goals

Our goals for the project can be extracted from what has been written in the previous sections, but nevertheless we find it feasible to devote a separate section to our expectations. The most evident goal is to get to know the basics of code coverage and how it can be employed to form a testing strategy. As mentioned earlier in this chapter, there is a myriad of coverage measures to choose from, and we want to get an overview of the most popular ones, along with their respective strengths and weaknesses. We

also seek to familiarize ourselves with the most noteworthy reliability models with respect to software.

With the recent emergence of numerous automated tools for code coverage analysis, we want to look at one of them in detail and uncover how code coverage theory can be implemented in a software program. We will also try to highlight possible areas that need improvement and come up with suggestions as to additional features that would make the product more complete.

Finally, it would be interesting to touch upon how code coverage affects and has been affected by the existing software development methodologies. The vast majority of the articles we have read on code coverage and reliability estimation dates back to the 1990s, and it is during the latter stages of this decade that new and test-driven methodologies such as the Rational Unified Process and eXtreme Programming were introduced to the software industry. Thus, it might be of interest to determine if the presence of these methodologies have somehow interfered with the popularity of code coverage.

## 1.6. Introductory Remarks

**T**hroughout this report we have used the terms software system, application and program interchangeably. This is also the case for software reliability growth model, software reliability model and reliability model. The motivation for this has been to add language variety and thus avoid repeating one specific term over and over again. We would also like to point out that parts of the information presented early in the report will occasionally be referred to or briefly repeated in later chapters for the sake of context.

## 1.7 Report Outline

**W**e have aimed at creating a report structure that follows a logical path, starting with a presentation of code coverage, its various measures and well-known reliability models, while progressively getting more specific as the report goes on. As a consequence, the first three chapters are rather broad in their content and can be seen as an introduction or build-up for the subsequent chapters. The intention is to build a reference or context for chapters 4 through 6, which deal with the core issues of this assignment.

Chapter 1 gives an introduction to the project, defining its context, motivation and scope and explaining our goals for the assignment.

Chapter 2 takes a look at code coverage analysis, the different measures available and the impact of test-driven development methodologies on code coverage.

Chapter 3 seeks to explain the role of reliability in computer systems, before presenting the essentials of software reliability models.

Chapter 4 deals with the relationship between code coverage and reliability by looking at proposed models for connecting them, experimental results and theoretical considerations as to the existence of such a relationship.

Chapter 5 takes a closer look at the Musa-Okumoto model for reliability estimation, as well as a technique which employs coverage information to pre-process data for use in a reliability model.

Chapter 6 presents an automated tool for code coverage analysis, discussing its use and implementation of selected coverage measures, before proposing an approach and considering the feasibility of integrating reliability estimation into JCoverage.

Chapter 7 rounds off the report by drawing conclusions, describing lessons learned, and looks at the possibility of further work.

*"You got to be careful if you don't know where you're going, because you might not get there."*

Yogi Berra

## CHAPTER 2:

# Code Coverage

This chapter will look at what code coverage is and present the most popular code coverage measures. Moving on, there is a section on how code coverage can and ought to be used in practice when testing software, followed by a description of common misconceptions and possible pitfalls in using code coverage analysis. Finally, we briefly explain the main ways of implementing code coverage principles into automated software tools, before rounding off this chapter with thoughts on how test coverage has been influenced by the modern-day, test-driven software development methodologies.

## 2.1 Code Coverage Basics

Code coverage analysis, also referred to as test coverage analysis, is described as a software testing technique aimed at discovering program code that has not been exercised by a set of test cases, in Steve Cornett's *"Code Coverage Analysis"* [Cor04]. In other words, code coverage refers to what extent the designed tests exercise the code base, or simply the thoroughness of the test case suite. As mentioned in *"Introduction to Code Coverage"* by Lasse Koskela [Kos04], code coverage can serve the purpose of an indirect quality measure – indirect in the sense that it is all about to what extent the tests cover the code, and thus an indicator of the quality of the tests. Needless to say, code coverage analysis requires the availability of source code for the program to be tested.

According to [Kos04], code coverage can be classified as a white box or structural testing technique, because assertions are made on class internals as opposed to system interfaces. Structural testing compares program behavior to the apparent intention of the source code, thus investigating how the program works by taking into consideration possible pitfalls in structure and logic. Structural testing is sometimes referred to as path testing, since chosen test cases lead to different paths through the program structure being exercised [Cor04].

[Cor04] presents code coverage analysis as the process of determining a quantitative measure of coverage and then creating additional test cases with the purpose of increasing coverage values. Optionally code coverage analysis can be used to identify redundant test cases, implying test cases that do not contribute to an increase in coverage. The article *"Using Simulation for Assessing the Real Impact of Test-Coverage on Defect-Coverage"* by Lionel

C. Briand and Dietmar Pfahl [BP00] claims that test coverage increases testing control, and hence improves allocation of test resources, by using coverage measures as estimators for the fraction of defects being detected during testing. The latter statement is probably what fuels the assumption that there exists a significant, causal effect between test coverage and defect coverage. One has to keep in mind, though, that code coverage analysis by no means ensures the quality of the end product, but merely contributes to the quality of the actual test set.

## 2.2 Code Coverage Measures

There is a multitude of code coverage measures to choose from. We begin this section by describing the most well-known measures, while later on giving a brief overview of less common measures and finally looking at the hierarchy that exist among them.

### 2.2.1 Essential Measures

Statement coverage and decision coverage are probably the most straightforward and household coverage measures known to the software community, but there are a few more that deserve attention. We will look at each of them in turn below.

#### 2.2.1.1 Statement Coverage

Statement coverage, also known as line coverage or basic block coverage, indicates to what extent individual statements have been encountered during testing. One advantage of this measure is that it can be applied directly to object code, and hence does not require source code processing [Cor04]. Its widespread use is most likely a result of developers being able to easily associate statement coverage with source code lines. Another strength when

```
1: public class HelloWorld {
2:     public static void main(String[] args) {
3:         System.out.print("Hello");
4:         System.out.print(" ");
5:         System.out.println("World!");
6:     }
7: }
```

compared to alternative measures is the fact that faults are assumed to be evenly distributed through the source code, and thus the percentage of executable statements encountered reflects the percentage of faults uncovered

[Cor04]. In the code example above each line represents a statement, while line 3 to 5 forms a block of code.

However, statement coverage poses a few challenges. Its insensitivity with respect to certain control structures such as `if` statements, as well as logical operators, is a definite weakness. Hence, it comes as no surprise that [Kos04] highlights the inability of statement coverage to assess how thorough the program logic has been covered. It simply reports whether each statement has been executed at least once, and as such will not report whether or not loops have reached their termination conditions, merely if the body of the loop was executed or not.

Basic block coverage and block coverage are known as related measures or variations of statement coverage. They view each sequence of non-branched statements as its unit of code as opposed to individual statements. As a result, basic block coverage will consider each branch “equal” to the other, irrespective of how much code the branch carries [Kos04]. A code block can be seen as a sequence of statements in a program where control enters at the first statement and leaves the block at the last statement of the sequence.

### 2.2.1.2 Decision Coverage

Decision coverage, some places referred to as branch coverage or basic path coverage, is a measure based on whether boolean expressions evaluate to both `true` and `false` when used in control structures such as `if` and `while` statements. This causes both paths to be exercised, but does not pay attention to how the boolean value was set [Kos04]. Decision coverage includes coverage of `switch` statements, exception handlers as well as interrupt handlers. In the method implemented below, decision coverage will reach

```
1 : public void prnTrail(boolean greeting, boolean weekend) {
2 :     System.out.println("Thank you for shopping!\n");
3 :     if(greeting) {
4 :         System.out.print("Have a nice ");
5 :         if(weekend)
6 :             System.out.println("day!");
7 :         else
8 :             System.out.println("weekend!");
9 :     }
10: }
```

100% if the created tests trigger the boolean variables `greeting` and `weekend` to evaluate to both `true` and `false`, but for all parts of the code to be exercised the tests need to include the test pairs `true/true` and `true/false`.

As was the case with statement coverage, this measure is simple and intuitive, but it avoids the problems caused by the former. Unfortunately, the ignorance of branches within boolean expressions which occur due to short-circuited operators is a considerable weakness [Cor04]. Short-circuit operators are illustrated by the following example, where the last expression will not be evaluated given that the first expression is true. This is caused by the logical

```
5 :      if(isRegularCustomer || price > 2000000) {
```

OR operator, which makes the `if` statement evaluate to `true` if at least one of the sub-expressions has the value `true`. Similar cases occur when using the AND operator, where it is sufficient for one of the sub-expressions to evaluate to `false` to cause the entire `if` expression to be `false`.

### 2.2.1.3 Condition Coverage

Condition coverage resembles decision coverage, but has superior control flow sensitivity. This is achieved by extending the boolean evaluation of decision coverage to consider sub-expressions, separated by logical ANDs and ORs, to ensure that each of them evaluates to both `true` and `false`. Each sub-expression is considered independently, without attention being paid to whether the complete expression is evaluated both ways [Kos04]. Thus, full condition coverage does not imply full decision coverage. This is illustrated below where an `if` statement contains two boolean sub-expressions that are tied together by a logical AND operator. As far as this particular case is concerned, having one of the boolean sub-expressions never evaluating to `false` translates to complete decision coverage without achieving complete condition coverage.

```
1:      public boolean sendSMS(String cellNum, String msg) {
2:          if(cellNum.getLength() != 0 && msg.getLength() != 0 ) {
3:              ...
```

### 2.2.1.4 Multiple Condition Coverage

Contrary to condition coverage, multiple condition coverage takes into account the complete expression, as well as sub-expressions. It reports whether each possible combination of boolean sub-expressions takes place; hence the test cases required to achieve complete multiple coverage for a condition, is given by the truth table of the condition's logical operator [Cor04].

The main downside of multiple condition coverage is the time-consumption involved in using it. It is a tedious task to determine the minimum set of test cases required, and the number of test cases required may vary significantly



among conditions that have comparable complexities [Cor04]. The fact that this measure considers the complete expression as well as sub-expressions often leads to a great rise in the number of test cases required, thus underlining the tediousness. However, for short-circuiting languages such as C, C++ and Java, multiple condition coverage is virtually the same as condition coverage.

```

1:   public boolean sendSMS(String cellNum, String msg) {
2:       if (cellNum.getLength() != 0 && msg.getLength() != 0 ) {
3:           ...

```

To reach complete multiple condition coverage in the scenario above, both boolean sub-expressions must evaluate to `true` and `false`, in addition to every possible combination of these combinations being executed. Finally, the main expression needs to evaluate to both `true` and `false`. The table below shows how the `if` statement evaluates depending on the boolean sub-expressions. If existing tests, for instance, prove incapable of causing both expressions to evaluate to `false` at the same time, the criteria for multiple condition coverage will not have been satisfied. Condition coverage and decision coverage will, on the other hand, be satisfied.

Expression	Evaluates to			
<code>cellNum.getLength() != 0</code>	T	T	F	F
<code>msg.getLength() != 0</code>	T	F	T	F
<b><code>cellNum.getLength() != 0 &amp;&amp; msg.getLength() != 0</code></b>	<b>T</b>	<b>F</b>	<b>F</b>	<b>F</b>

### 2.2.1.5 Path Coverage

Another common coverage measure is path coverage which reports whether each possible path in every single function has been covered. [Cor04] defines a path as a unique sequence of branches from function or method entrance to exit, the latter typically being a return statement or a thrown exception. Loops present a delicate challenge to path coverage by possibly introducing an enormous number of paths. Making sure every single path is executed can thus prove both tedious and infeasible, although the thorough testing that path coverage requires can be seen as an advantage.

To deal with an excessive number of paths, several variations of path coverage have been proposed. Boundary-interior path testing considers two possibilities with regards to loops – zero repetitions or more than zero repetitions. Hence, one effectively reduces the number of paths by considering two scenarios, regardless of how many possible paths the loop presents. Another alternative mentioned in [Cor04] is *n*-length sub-path coverage, which reports whether each path of length *n* branches has been exercised.

Linear code sequence and jump coverage, LCSAJ for short, as well as data flow coverage are related measures to path coverage. The former restricts its focus to consideration of sub-paths that can easily be represented in the source code. A linear code sequence may contain decisions, given that control flow continues from one line to the next at runtime [Cor04]. Not only does this measure deal with the explosive nature of path coverage, where the number of paths grows exponentially with the number of branches, but it is also known to be more thorough than decision coverage. Data flow coverage, on the other hand, merely considers sub-paths from variable assignments to subsequent variable references and has a tendency of turning out overly complex.

```
1: public float calcPrice(float price, float deliveryCosts) {
2:   if(price < 200)
3:     return price + deliveryCosts;
4:   else
5:     return price * TAX + deliveryCosts;
6: }
```

The code example above presents a method with two possible return statements at line 3 and 5. In this case, complete path coverage is only achieved when the method has returned both statements. Thus, the test set has to include data that causes the price variable to be both less than 200 and greater than or equal to 200.

### 2.2.1.6 All-uses Coverage

All-uses coverage criteria are based on a program's data flow as well as its control flow and as such all-uses coverage is considered to be an advanced coverage measure. It consists of a def-use pair, which in turn consists of two statements – the first statement assigns a value to a program variable, while the second statement uses the value of the same variable. According to Fabio Del Frate, Praerit Garg, Aditya Mathur and Alberto Pasquini in their article titled “*On the Correlation between Code Coverage and Software Reliability*” [FGMP95], a def-use pair for a given variable  $x$  is covered when control reaches the first statement of the pair, and during the same program execution control reaches the second statement without reaching a statement that assigns a value to  $x$ . All-uses coverage is the sum of computational-use and predicate-use coverage measures – c-use and p-use coverage for short – which will be explained next.

To explain what c-use and p-use coverage is, one needs to know what a c-use and p-use pair consist of. In the article “*Software Reliability Growth With Test Coverage*” written by Yashwant Malaiya, Michael Naixin Li, James Bieman and Rick Karcich [MLBK02], a c-use pair is said to include two points in the program, the first where the value of a variable is defined or modified,

followed by a point where the variable is used within a computation. C-use coverage thus reports the fraction of the total number of c-uses that have been covered during testing. As was the case with a c-use pair, a p-use pair includes two points in the program – the first point being where the value of a variable is defined or modified, followed by a point where the variable is used within a conditional expression as a predicate. Hence, complete p-use coverage implies complete decision coverage, assuming that all conditional expressions contain variables. In the code example below c-use pairs exist on the lines (45, 49), (45, 51), (46, 49) and (46, 51). P-use pairs can be found on lines (45, 48) as well as (46, 48).

```
43: ...
44: int difference = 0;
45: int myAge = 25;
46: int yourAge = 65;
47:
48: if(myAge < yourAge)
49:     difference = yourAge - myAge;
50: else
51:     difference = myAge - yourAge;
52:
53: System.out.println("There's a " + difference
                    + " year age difference!");
```

### 2.2.2 Alternative Measures

In addition to the coverage measures already described with their respective strengths and weaknesses summarized in table 2.1 on the next page, there is a wealth of more specific and less widespread measures to choose from. Function coverage is used to make sure that each function or method has been invoked, and is particularly useful when performing preliminary testing. Call coverage is a measure which is used to verify that all function calls have been executed. Its purpose is based on the hypothesis that faults typically occur in interfaces between modules [Cor04]. The same paper claims that boundary test cases often detect so-called off-by-one errors, commonly due to misunderstandings when using relational operators. Relational operator coverage thus reports whether expressions containing relational operators are tested with boundary values. In the for loop below faults may arise if the less-than-or-equal-to operator was intended as opposed to the less-than operator.

```
32: ...
33: for(int i = 0; i < array.length; i++) {
34: ...
```

Condition/decision coverage is a hybrid measure derived from more essential and basic measures already described in chapter 2.2.1, consisting of the union of condition coverage and decision coverage. Its main advantage is its simplicity, while at the same time avoiding shortcomings found in both condition and decision coverage [Cor04]. Another convenient measure when dealing with multithreaded applications is race coverage. Race coverage considers multiple threads that execute code simultaneously, thus contributing to failure detection when synchronizing access to resources.

Coverage measure	Strengths & weaknesses
<i>Statement coverage</i>	<ul style="list-style-type: none"> <li>+ intuitive</li> <li>+ direct application to object code</li> <li>+ source code processing not required</li> <li>÷ insensitivity with respect to control structures</li> </ul>
<i>Decision coverage</i>	<ul style="list-style-type: none"> <li>+ intuitive</li> <li>+ exercises control structures</li> <li>÷ ignores branches within boolean expressions which occur due to short-circuit operators found in C, C++ and Java</li> </ul>
<i>Condition coverage</i>	<ul style="list-style-type: none"> <li>+ flow sensitivity</li> <li>+ considers sub-expressions</li> </ul>
<i>Multiple condition coverage</i>	<ul style="list-style-type: none"> <li>+ considers both sub-expressions and the complete expression</li> <li>÷ time-consuming</li> </ul>
<i>Path coverage</i>	<ul style="list-style-type: none"> <li>+ thorough</li> <li>÷ tedious</li> <li>÷ complicated loop treatment</li> </ul>
<i>All-uses coverage</i>	<ul style="list-style-type: none"> <li>+ exercises the relationship between the assignment of a value to a variable and the subsequent use of that value</li> <li>÷ computationally expensive</li> </ul>

Table 2.1: *Summary of essential coverage measures*

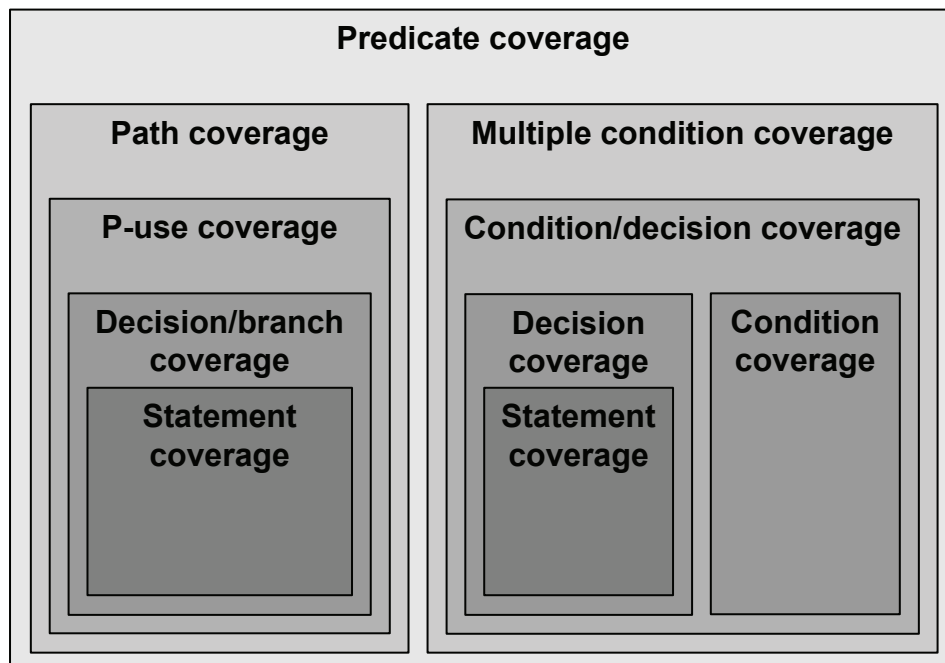
Finally, we mention mutation coverage, which tests the computational structure of a program. According to an article titled “*Connecting Test Coverage to Software Dependability*” by Dick Hamlet [Ham94], mutation can be viewed as massive fault seeding and considered a technique for estimating how many failures are yet to be found. This coverage measure is known to be

computationally expensive and challenging to employ, partly because the test data required to achieve high mutation coverage are less obvious and harder to collect systematically. [Cor04] mentions weak mutation coverage as a more general alternative to relational operator coverage. This variation of mutation coverage reports whether there exist test cases that expose the use of wrong operators and operands. Mutations typically include exchanging operators, data types and adjustment of constants.

### 2.2.3 Coverage Measure Hierarchy

As was noted in the introductory chapter of this report, relationships exist among measures, with the “stronger” measure said to subsume the “weaker” one, thus forming the basis for a subsumption hierarchy. Parts of the hierarchy are presented visually in figure 2.1 below.

Decision coverage includes statement or block coverage since execution of each branch implies that each statement has been exercised. Complete decision coverage is, according to Yashwant Malaiya et al. in the article “*The Relationship Between Test Coverage and Reliability*” [Mal+94], achieved by complete p-use coverage. The same article concludes that both branch coverage and to a lesser extent p-use coverage correlate significantly with block coverage, whereas c-use coverage appears to have no such relation to other measures.



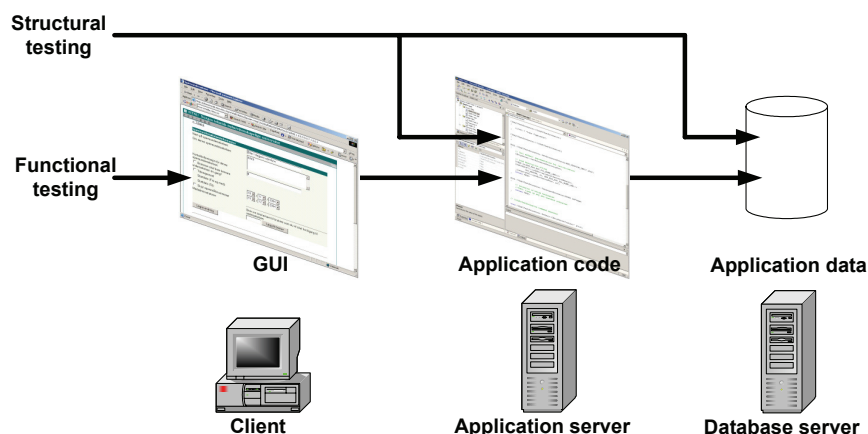
*Figure 2.1*  
*Subsumption hierarchy – the brighter*  
*rectangles subsumes the darker ones*

Yet another measure known to subsume decision coverage, along with the ones already mentioned, is path coverage. This is also the case for the hybrid measure of condition/decision coverage, which per definition includes condition coverage. As was mentioned in chapter 2.2.1.6 when presenting c-use and p-use coverage, complete p-use coverage implies complete decision or branch coverage, assuming that all conditional expressions contain variables.

Moving up in the hierarchy, exercising all paths in a program implies that all p-uses have been covered, hence resulting in path coverage subsuming p-use coverage [MLBK02]. Finally, [Cor04] places predicate coverage at the top of the hierarchy. Predicate coverage is strongly related to path coverage and considers paths as possible combinations of logical conditions, thus including strong measures such as path coverage and multiple condition coverage [Bei90].

## 2.3 Practical Code Coverage Application

We have already underlined the fact that code coverage is a more feasible and beneficial approach when testing applications that contain a large amount of decision points, as opposed to data-centric systems. Being a structural testing technique, code coverage analysis may prove particularly useful if the requirements specification lacks detail or simply has not been subjected to regular updates as the development process has progressed. Functional testing techniques, on the other hand, rely on an up-to-date specification when evaluating test program behavior [Cor04]. Figure 2.2 below illustrates the difference between structural and functional testing.



*Figure 2.2*  
*An illustration of structural and functional testing*

In his article titled “*How to Misuse Code Coverage*” [Mar99], Brian Marick acknowledges the usefulness of code coverage because of its ability to detect parts of the source code that have been neglected by the test set. He goes on to claim that code is typically covered in sections, where a section represents a considerable part of software functionality and not merely a line or two. If such a section or part of the code escaped testing unintentionally, the test team can direct focus towards that specific part of the program.

### 2.3.1 Choosing the Right Coverage Threshold and Measures

Choosing appropriate coverage measures can be quite challenging. The coverage tools available on the market tend to support different algorithms as well as using their own accent, according to [Kos04]. The author urges software developers who aim at integrating code coverage analysis with their existing development practice, to be consequent in the decisions being made. For popular programming languages such as C, C++ and Java, the general advice is to employ condition/decision coverage. Other measures might be used in addition to add coverage details and remedy possible weaknesses of the chosen measure.

When it comes to coverage thresholds, each project ought to decide on a minimum percentage value of code coverage, which has to be attained before releasing the software. Such a threshold should take into account the available test resources as well as the importance of avoiding post-release failures [Cor04]. Generally speaking, one should aim at reaching 80-90% coverage prior to release when using traditional coverage measures such as statement coverage, decision coverage or even condition/decision coverage. Code coverage is likely to increase as more test cases are applied, assuming that no test case is repeated and that complete code coverage is yet to be attained.

In the context of coverage thresholds, [Mal+94] refers to experiments where fault coverage was a mere 10% at 50% branch coverage. However, when increasing branch coverage to 84%, which relatively speaking is a modest rise compared to the initial 50% when considering the drastic improvement in fault coverage, an impressive fault coverage of 90% was attained. Such results support the general comprehension that 80% branch coverage is sufficient for most applications.

### 2.3.2 Coverage Progress and Effort

In order to make sure that coverage increases in the early stages of testing one should aim at attaining a broad coverage through the entire program before striving for high coverage percentages in specific areas of the code. This can be fulfilled by visiting each feature of the program under test and hence increasing the likelihood of detecting obvious or significant failures early on.

The initial strategy should be to look for easy-to-find failures with minimal testing [Cor04].

Test productivity is a keyword not only when talking about code coverage analysis, but also for testing techniques and strategies in general. For the purpose of maintaining a high level of test productivity one has to strive for achieving optimal results with minimal effort. This implies detecting and removing as many failures as possible, while at the same time spending a limited time on creating test cases, adding them to the existing test suite and eventually executing them. As such, focus should not rest on reaching 100% coverage for each of the initial measures, but rather on choosing appropriate intermediate coverage measures and deferring testing to areas deemed challenging and critical by an operational profile, if one exists.

Although code coverage has its advantages, it is only one of many testing techniques to choose from, and relying on code coverage alone is not the way to approach testing activities. However, it is undoubtedly a useful addition to other strategies and may serve the purpose of an alerting service, signaling the fact that the existing test suite has room for improvement. A challenging prospect unfolds when considering software applications that are under development while testing is performed. The addition of new modules to the existing core application is deemed to cause the entrance of new defects. Analysis of such programs are, however, considerably more complex and a field of future research [Mal+94].

## 2.4 Pitfalls and Misconceptions

Several articles are quick to point out that code coverage analysis by no means presents a silver bullet in software testing. Brian Marick highlights a few pivotal and common misconceptions concerning code coverage in [Mar99] that suggest how this testing technique has the potential to mislead unaware software testers.

### 2.4.1 Complete Coverage is No Guarantee

According to [Mar99], making sure that all logical expressions evaluate to both `true` and `false` is hardly sufficient for claiming that testing is completed: “*Coverage tools can only tell me how the code that exists has been exercised. It can’t tell me how code that ought to exist would have been exercised*”. For instance, faults that can be removed and fixed by adding new code – known as faults of omission – may pass tests without being discovered. Thus, there is no way of guaranteeing a faultless program in spite of running tests and making sure they cover every single line of source code. Coverage tools will, however, be able to improve overall quality by detecting possible “holes” in the existing test set.



Code coverage is also capable of revealing errors in the implementation of tests. A particular test may, for instance, do something entirely different than what it was set out to do and hence need modification in order to fulfill its initial intention. However, the bottom line is that 100% code coverage does not imply a program free of faults.

## 2.4.2 Code Coverage and the Design of Tests

When performing code coverage analysis it might be tempting to create a set of tests that aim for a rapid increase in coverage. Designing tests with high coverage percentages in the back of one's mind is, however, not the way to go [Mar99]. On the contrary, focus should be on comprehending why the tests being executed failed at exercising the parts of the software or source code that ended up untested. Coverage tools will only prove helpful if they are utilized to increase understanding, and not if they result in testers leaving the thinking and analyzing to the tools.

[Mar99] also mentions the importance of management not using code coverage percentages as a means of measuring the quality and end result of testing efforts. This will only lead to testers optimizing tests with respect to high code coverage, since this will please managers and make sure the goals set forth are met. Such a focus is likely to come at the expense of thought-through tests designed to optimize fault detection. Thus, code coverage serves a one-way purpose; notifying testers that additional testing is necessary, but not capable of telling that sufficient testing has been carried out. Table 2.2 below presents a rough overview of what code coverage contributes with and what it is not capable of.

Code coverage does	Code coverage does not
<ul style="list-style-type: none"> <li>- report how existing code has been exercised</li> <li>- improve the quality of a test set</li> <li>- reveal errors/faults in test implementation</li> <li>- increase understanding of existing tests</li> </ul>	<ul style="list-style-type: none"> <li>- detect faults of omission</li> <li>- guarantee a fault-free software application</li> <li>- ensure end product quality</li> <li>- indicate that sufficient testing has been performed</li> </ul>

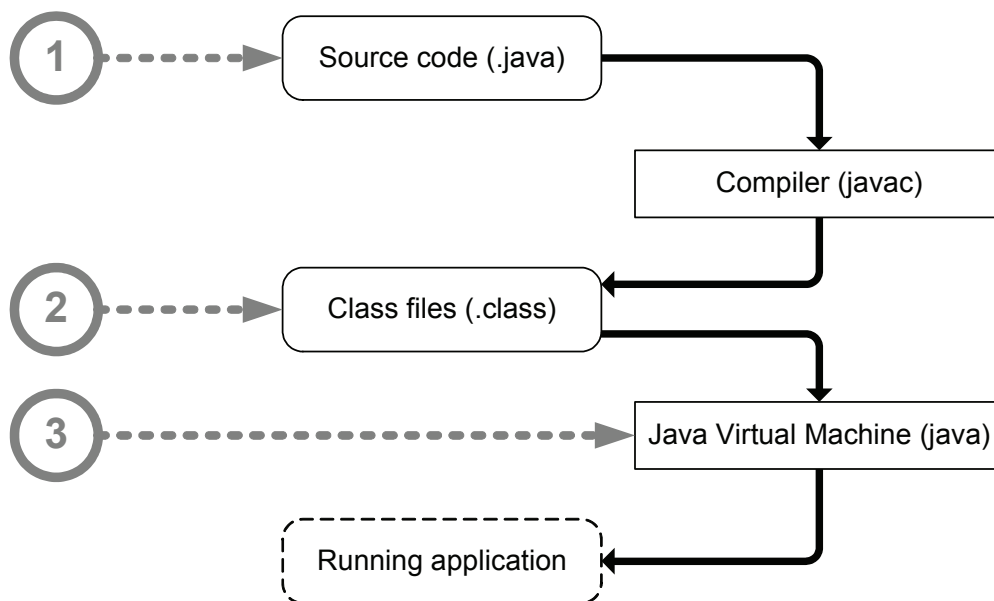
Table 2.2: *Facts and misconceptions of code coverage*

## 2.5 Code Coverage Implementations

A good number of code coverage tools have emerged lately. JCoverage, whose implementation and features we will take an in-depth look at in chapter 6, is just one of several household tools to choose from. The way these tools implement code coverage theory boils down to two main approaches briefly described in [Kos04]: Instrumentation and custom Java Virtual Machine – JVM.

### 2.5.1 Implementation Approaches

Basic instrumentation, also known as source instrumentation, is probably the most intuitive way of implementing code coverage in a software tool. It relies on manipulating application code by inserting reporting code in strategic places of the source code, whereas class instrumentation inserts reporting code directly into compiled class files, represented as byte code. This latter approach is found in our coverage tool of choice – JCoverage.



*Figure 2.3*  
*Implementation sketch*

The custom Java Virtual Machine approach causes the virtual machine to take responsibility for keeping track of the parts of the loaded classes that have been executed. Compared to the instrumentation alternative, this approach is yet to enjoy the same popularity. Both strategies are depicted in figure 2.3 above. There is also a third way of dealing with code coverage implementation mentioned in [Kos04]. It involves instrumenting application

code through reporting code explicitly, by using wrapper classes responsible for inserting code at runtime as opposed to pre-processing source code or byte code at build-time.

## 2.5.2 Tool Features

The set of features that code coverage tools offer tend to vary slightly, partly depending on the coverage measures chosen. However, the most well-known ones share a few, but nonetheless essential features, namely related to:

- ✓ Ant integration
- ✓ Report formats
- ✓ Source code linking
- ✓ Checks
- ✓ Historical reports

According to [Kos04] most Java projects taking place nowadays use Ant, or alternatively Maven, to manage the build process as well as running unit tests. As such, proper Ant integration is practically a necessity for any high-quality coverage tool. Ant is briefly explained later in chapter 6. Needless to say, presenting code coverage reports in an intuitive and well-arranged way is of utter importance, although report formats and actual layout will differ slightly from tool to tool. Some tools might also provide historical reports to illustrate coverage progress from start to finish.

Another requested feature is the linking of source code to code coverage reports, where uncovered parts of code are highlighted in an annotated copy of the original source code. This helps to guide the user's attention to code or blocks of code yet to be exercised, instead of merely reporting line numbers. Finally, incorporating checks into the tool implies notifying the user when coverage drops below a pre-defined level.

## 2.6 The Effect of Test-driven Development Methodologies

**B**y now we have concluded that code coverage should be used as an indicator as to how thorough the software has been tested and that test teams should resist the temptation of designing tests with the purpose of reaching 100% coverage. In modern-day development methodologies, tests are typically designed prior to the actual code to be tested, thus resulting in code being created to satisfy the designed tests. Logically, this might easily lead to tests reaching coverage values close to 100%. With tests attaining high coverage straight away, code coverage reporting seemingly ends up as a mere confirmation that most, if not all parts of the code were exercised. Intuitively, this could mean that code coverage analysis is turning into a redundant

supplement for applications engineered with test-driven development methodologies.

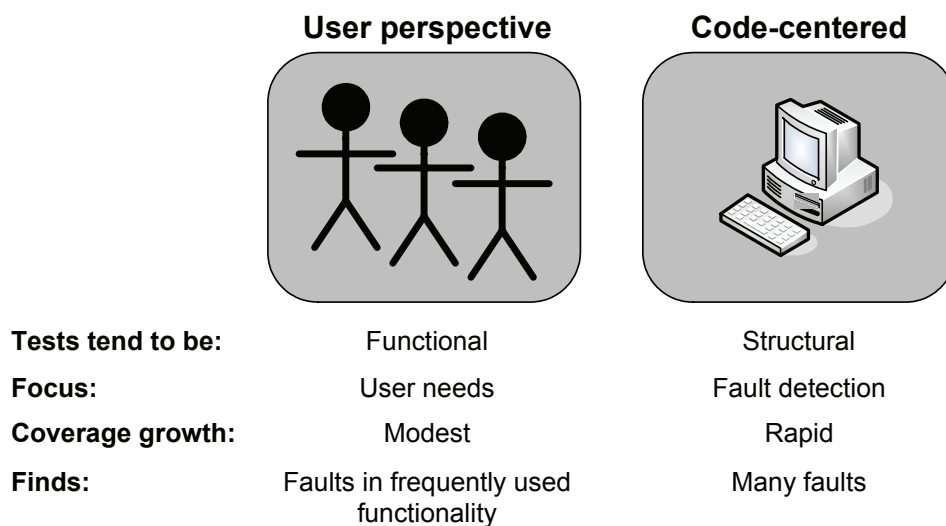
### 2.6.1 The Role of Code Coverage

One of the biggest challenges of testing is to decide how much testing should be performed. The question of determining when sufficient testing has taken place has no evident answer. Earlier in this chapter we pointed out that faults will not be found in code that has escaped testing. This fact is reasonably obvious to everyone. However, the claim that code can be tested without detecting possible faults residing in it is not as intuitive for most of us. This is where code coverage contributes by using reported coverage values as a confirmation that more testing is necessary.

It is tempting to ask oneself what is really wrong with designing tests that achieve complete code coverage. Does creating a test set without the conscious strive of having all parts of the code covered, result in more faults getting exposed? We believe that it may lead to additional tests being created and executed in order to satisfy coverage requirements and that the sum of initial tests and supplementary tests will contribute to more faults being detected. Another motivation for employing code coverage is its ability to guide testers to parts of code not yet exercised, as mentioned earlier. This raises the question of whether it is the process of gradually improving the test set or the fact that additional tests are performed which encourages this recommendation. According to our understanding, code coverage analysis provides suggestions and information regarding what parts of the program require further testing, and that this piece of information is lost when the initial tests already cover substantial parts of the code. As a result, the task of deciding how additional tests should be designed is left entirely to the test team.

This brings us to consider how test-driven development methodologies leads testing to become programming-centric as opposed to software applications engineered with traditional development frameworks, where testing is typically performed from a user perspective. Even so, tests that exercise programs developed with traditional methodologies may be characterized as programming-centric if they are designed to satisfy code and code coverage criteria intentionally. Our impression of relevant literature is that performing testing with an operational profile as a platform is both common and recommended. Operational profiles are said to contribute to realistic results by seeking to mirror the actual usage and environment of the software. This appears to be in line with [Mar99], who encourages use of coverage knowledge to encounter user needs that are insufficiently covered by the current test set. We will take a closer look at operational profiles and their impact on reliability estimation in chapter 4.

Once a product is fully developed and testing has covered all parts of the code, the question of whether enough testing has been undertaken still stands. Software literature suggests that methodologies embracing test-driven development tends to achieve 100% code coverage – an understandable assertion considering how closely connected tests and application code are. In spite of complete code coverage being incapable of guaranteeing fault-free software and high quality products, test-driven methodologies thrive on words of praise from the software community because of its contribution to end product quality. We believe that this boils down to differences beyond the testing itself, and that a successful development process must be seen as a significant tool in quality work as well as the tests it creates. Figure 2.4 highlights some aspects that typically separate user-centered testing from code-centered.



*Figure 2.4*  
*User-centered versus code-centered development*

The perceptions and opinions as far as code coverage is concerned, seem to be many and contrasting. What suits certain development environments might not match the requirements and preferences of others. Hence, discussions on the pros and cons of code coverage and its range of application are deemed to carry on.

### 2.6.2 Object-oriented Testing

Object-oriented testing introduces new elements and hence new challenges as far as software testing is concerned. The two fundamental design features of object-orientation are information encapsulation and polymorphism. In short, encapsulation hides internal structures from the rest of the program and thus creates a considerable challenge, since traditional tests usually are external.

This matter can be solved by injecting test functions directly into the classes to be tested; however, this is not within the scope of this project.

Polymorphism, on the other hand, represents a greater problem with respect to code coverage, since classes can be sub-typed after testing has taken place. According to Craig E. Damon in his course notes from the subject “*Software Engineering*” at the University of Vermont [Dam04], additional tests have to be designed to ensure that the current class is protected from “unusual”, yet perfectly legal sub-classes, because of the introduction of polymorphism. He deems existing code coverage measures as insufficient within an object-oriented context and presents two newly suggested measures tailored to object-orientation:

- ✓ Functions or methods that have been overridden in sub-class
- ✓ Combinations of sub-class and super-class that have been tested

Thus, it seems as though code coverage is very much alive. However, as is the case with software development methodologies, programming languages and paradigms as well as technology in general, it needs modifications and additions to be tailored to modern-day standards.

“We are ready for any unforeseen event which  
may or may not happen.”  
George W. Bush

## CHAPTER 3:

# Software Reliability Growth Models

With software systems playing an increasingly prominent role in most organizations and businesses, regardless of domain, reliability is becoming a quality attribute of great focus. Software developers, as well as customers, do not only worry about releasing and eventually employing highly reliable products – they also seek predictions and estimates as to reliability during development and testing. In this chapter we take a look at general challenges in attaining reliability and what factors influence it, before moving on to existing models for estimating reliability in software products – the software reliability growth models. Substantial parts of this chapter is inspired by the representation given in *“Software Reliability: Measurement, Prediction, Application”* by John D. Musa, Anthony Iannino and Kazuhira Okumoto [MIO87].

## 3.1 Reliability of Computer Systems

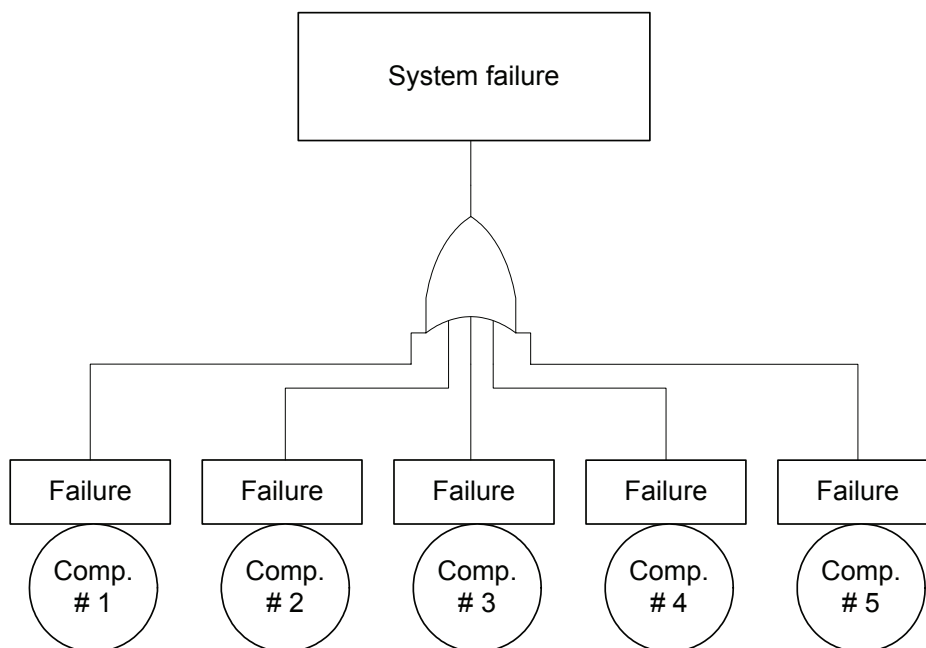
A multitude of computer systems serve as the backbone of processes and services requiring high levels of availability, either because lives depend on them or because economical considerations deem their reliable operation crucial to the company. Computer systems that manage and control day-to-day operation of nuclear plants or web-based shopping facilities such as Amazon.com are two examples. Although not all systems are labeled indispensable, proper operation and, hence, stable functionality is desirable at any time; however, this is not always possible due to the economical constraints of the market, which inevitably affect developers.

### 3.1.1 Implications of Component Failures

No matter how many considerations are taken or unlikely scenarios predicted, things can always find a way of catching users and developers by surprise. In the case of critical computer systems, vast amounts of resources are spent to avoid the presence of unforeseen actions and operations. In certain critical systems, safety might depend on reliable operation of a computer system, thus making unreliable systems a risk factor in a safety context. Marvin Rausand defines safety management as systematic efforts directed at achieving and maintaining a given level of safety, in his book on risk management, *“Risikoanalyse – Veiledning til NS5814”* [MR91]. The safety level could be determined by the owner, public authorities or other system stakeholders. The

actual work consists in surveying risk factors, managing possible deviations and considering actions and initiatives targeted at reducing overall risk. In cases where computer systems are part of a critical process there are rigid safety requirements to obey. As will be illustrated below, the structure or architecture, as well as higher level design of computer systems is likely to influence reliability significantly.

Computer systems typically consist of several components that cooperate in order to offer system services and functionality. A failure in one of these components may lead to service disruption and hence, all components constituting a computer system will affect reliability. The fault tree depicted in figure 3.1 illustrates the essence of this discussion.



*Figure 3.1  
Fault tree example – a failing  
component will cause system failure*

One way of improving reliability is to introduce redundant components with the intention of ensuring system operation in spite of a single component failure. The system approach of Redundant Array of Independent Disks – RAID – serves as an example of this, where several hard drives are employed to avoid loss of data in case of a disk crash. The duplicated functionality will lead to additional costs for the organization. An exemplified redundant system is represented in terms of a fault tree in figure 3.2 below. As far as this instance is concerned, all components are duplicated to guarantee successful operation regardless of one or more components failing. In cases where several components fail, the system may only continue operation if there exists a well-functioning, duplicated component which can resume operation



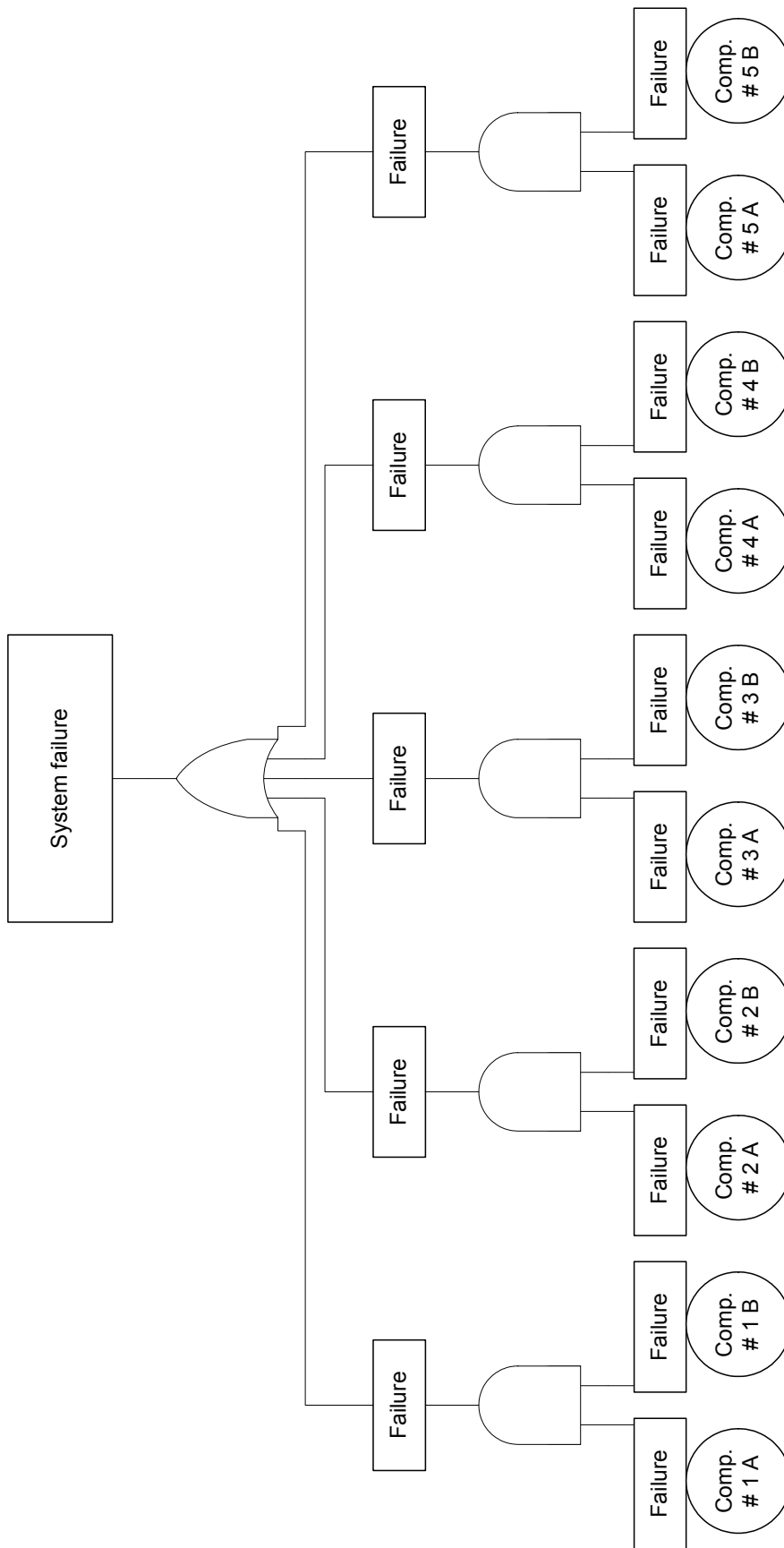


Figure 3.2  
Example of a redundant system

for the failed component. Alternative configurations may also be found; for instance one might opt for a more selective strategy and only duplicate components that are reckoned to be exceptionally vulnerable.

If probabilities of component failures are available for each and every component constituting the overall system, the probability of system failure can be computed. Several hardware manufacturers state the likelihood of failure for their respective components, hence making it a rather trivial task to calculate the probability of failure for the hardware part of a computer system. As was the case with hardware components, software modules are also capable of failing, typically caused by flaws in design or code. When hardware components are seen to fail, on the other hand, physical tear and wear is normally the main factor, whereas constructional flaws contribute to a lesser extent.

By employing comparative measures of reliability for both hardware and software components, these figures can be combined to yield an overall measure of system reliability. In order to make realistic assertions as to system reliability, fairly accurate estimates are required for each system component. Hence, software developers need a means of acquiring reliability estimates, with an appropriate level of correctness, for the software they engineer.

## 3.2 The Importance of Software Reliability

Reliability is known to be a prominent quality attribute of a software product and thus a property used to describe the qualities of a given system. [MIO87] emphasizes quality, costs and schedule as the three most essential characteristics of a software application. Because of the difficulties in measuring or quantifying quality in software, determining the relative importance of the aforementioned characteristics is a challenging task. According to [MIO87] this could be a possible explanation with regards to general quality problems in the software industry.

Finding the right balance between quality, costs and schedule is of utmost importance in order to make a product a commercial success. For instance, a computer game crashing every now and then may not be thought of as critical. However, it is nonetheless deemed to affect the customer's attitude towards the game itself and the vendor, as well as developers, if problems frequently arise as a consequence of unsatisfactory quality. Spending vast resources on quality assurance and improvement, on the other hand, runs the risk of developing a costly, hard-to-sell product. The third factor – time – comes into play in an industry of constant evolution, where computer games stand the risk of being labeled old-fashioned by the time they hit the shelves if timing is wrong. Thus, systems stemming from different areas of usage, whether computer games, financial applications or real-time monitoring systems, are

likely to have a different emphasis with respect to the three characteristics and their relative importance.

Being able to quantify the reliability of a software system can be viewed as a selling point or a platform for decision-making if the product is considered employed in a critical context. For safety-critical applications it becomes all the more important that the produced estimates closely mirror the actual, observed reliability. Achieved reliability in software is impacted by the following three factors, according to [Mal+94]:

- ✓ Test strategy – black box or white box
- ✓ The relationship between calendar time and execution time
- ✓ Testing of infrequently executed modules

The third and final factor relates to methods and functions dealing with exception handling, error recovery and the likes. [Mal+94] claims that reliability can only be predicted with a high degree of precision by having tests cover low-usage, yet critical components like those mentioned above. In spite of extensive efforts in the past, accurate estimation of reliability has proven to be a daunting challenge, thus making improved reliability prediction methods a field of active research. In the subsequent section we will present the current means of reliability estimation – software reliability models.

### 3.3 Software Reliability Models

Software developers often desire a measurable unit of how reliable a system under construction is. The reliability measures of the early days were limited to quantifying the number of failures in the software. Although far from being a feasible indicator, it was used to make rough comparisons between various projects. Aside from lacking precision, the measure did not convey particularly useful information to end users nor developers.

#### 3.3.1 Introduction to Reliability Models

Nowadays software reliability growth models are employed to statistically determine how reliable a given program is. This implies the use of failure data gathered during testing to discover trends that could give an indication as to the quality of the current system. [MIO87] argues that reliability measures are easily graspable even for people without programming knowledge or experience. This, in turn, simplifies the process of controlling whether the development organization has delivered a product in conformance with a service level agreement – not only for customers, but for the vendor as well.

The reliability growth models have evolved empirically through steady development and refinement. Typically, researchers launch hypotheses as to what affects reliability and subsequently challenge them through empirical experiments. In these experiments, reliability models are put to use and their estimates are compared to the real, observed reliability. The actual reliability can be measured once the software under scrutiny has been operational for a while and hence provided system administrators with useful information of its operation. An alternative approach encountered is to utilize existing software applications with proven levels of outstanding reliability, and then expose the source code to a fault-seeding process. Moving on, the faulty code is tested and subsequently investigated through a chosen reliability model. The deviations observed from the different models are finally used to discuss the pros and cons of each of them.

### 3.3.2 Reliability Model Ingredients

A software reliability growth model is a mathematical representation of various program properties. The operation of the most renowned models relies on two main ingredients – time and failures. We will now take a closer look at these two input parameters to the models, which yield reliability estimates for a given application.

#### 3.3.2.1 Time

Most reliability measures connect reliability to some notion of time. However, as [MIO87] points out, there is no real hinderance as to the employment of other variables considered feasible. This could for instance be the number of executed transactions or some other variables capable of quantifying system usage. The use of time in reliability estimation requires a set of specific

<b>Time notion</b>	<b>Definition</b>
Execution time	Time during which the program utilizes the central processing unit (CPU)
Clock time	Time elapsed between program start and termination on a computer. This includes idle time – time where the software awaits user input or information from a different system before resuming operation
Calendar time	Regular time corresponding to the human-made calendar – the way we usually deal with the notion of time

Table 3.1: *Time definitions*

definitions of the term, with the purpose of avoiding ambiguity. Table 3.1 presents time definitions in accordance with [MIO87].

Reliability growth models that make use of execution time in their estimation are considered to be superior [MIO87]. This assertion makes sense assuming that idle time does not contribute to increased reliability. Yet there is an evident need for calendar time, since it carries an intuitive meaning and represents the notion of time which we relate to in our everyday lives. To illustrate this case, a reliability measure of 0.95 probability for error-free operation in a time period of five execution hours, does not make immediate sense to most of us. This is because we do not relate to execution time in the same way we relate to calendar time. Thus, by converting the notion of CPU time to the familiar calendar time we end up with a probability measure of error-free operation that actually conveys processable information. An execution time of five hours could, for instance, translate into 48 calendar hours of software operation. Naturally, these translations between the different time notions are not fixed, but will vary from program to program and from environment to environment.

### 3.3.2.2 Failures

The second and final input parameter to the reliability growth models is the number of failures detected during testing. Failures are measured with respect to time, with the possibility of employing any of the three notions of time defined in the previous section. Table 3.2 below gives an overview of the failure measures as presented in [MIO87].

Failure measure	Definition
Time of failure	Time elapsed between program start and failure detection, typically measured in seconds
Time interval between failures	Time elapsed between observed failure $f_n$ and the previous failure detection $f_{n-1}$
Cumulative failures experienced up to a given time	The total number of failures observed during testing up until time $t$ . This measure can be calculated at fixed intervals during testing
Failures experienced in a time interval	Division of testing into time intervals, with the number of discovered failures in each interval reported

Table 3.2: Failure measures

### 3.3.3 Randomness of Variables

There is some uncertainty involved in the variables entering the reliability model, thus encouraging model users to take into account the possible imprecision in generated estimates. The fact that the values of variables are random does not imply that the same values are unpredictable, but rather that the exact value is unknown [MIO87]. The different values which variables can take have a distinctive probability associated with them, denoting the likelihood of that particular value occurring. The probability of a variable being assigned the value  $x$  corresponds to the fraction of tests where  $x$  is registered. Usually, the value range of a variable is known along with the average value and to a certain extent its statistical dispersion. It is a common perception among people that the assignment of values to variables adheres to a uniform distribution, but according to [MIO87] this assignment does not need one specific probability distribution as a basis.

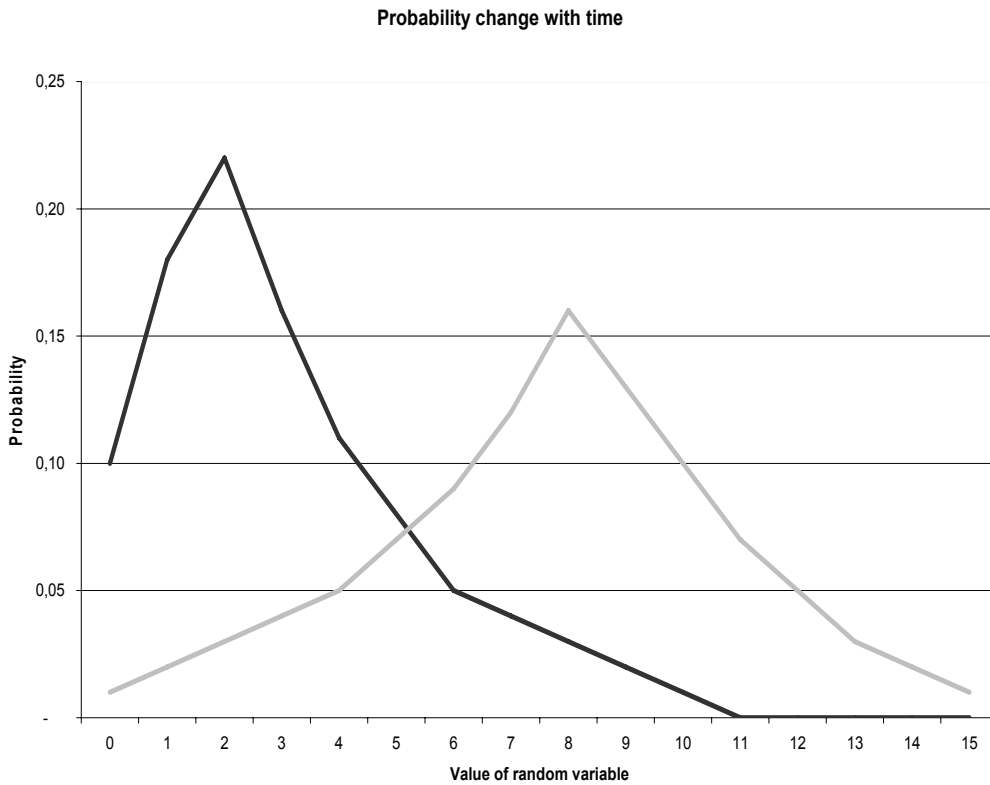
#### 3.3.3.1 Causes of Randomness

Three main reasons are stated in [MIO87] for the randomness in assigning values to variables. First, the process of a programmer unintentionally injecting faults into the code, resulting in failures when executed, is highly complex and unpredictable. Second, software typically runs in environments where human knowledge of conditions under which it operates might be incomplete. A third and final issue of interest is that failures depend on faults and that a program is run under specific conditions which, together with one or more faults, trigger a failure. The latter can be viewed as a combination of the first and second cause. Knowing the probability of a certain amount of failures appearing within a given time interval, a measure of the average number of failures can be found by combining probabilities and the cumulative number of failures discovered during that period.

#### 3.3.3.2 Failure Probability Distribution

The more testing and fault removal is performed, the harder it becomes to uncover additional failures, and hence the probability of failure detection is not constant. The probability of a variable being assigned a given value is destined to change with time; for instance, the likelihood of discovering exactly two failures during one hour of testing will differ from the probability of detecting the same amount of failures in, say, three hours. This is a characteristic of random processes, which *“can be viewed as a set of random variables, each corresponding to a point in time”* [MIO87]. Thus, the number of failures detected in the application can be represented as a random process, characterized by the probability distribution of the random variables and the variance with respect to time. [MIO87] elaborates: *“A random process whose probability distribution varies with time is called nonhomogeneous. Most*

failure processes during test fit this situation”. Poisson distributions are frequently employed probability models when dealing with random processes. When a process changes continuously with respect to time it is referred to as a non-homogenous Poisson process – NHPP.



*Figure 3.3*  
Probability distribution for number of failures found in one and five hours respectively

How time affects a random process is tried illustrated in the example of figure 3.3 above. Both graphs depict the probability of assigning a given value to a variable, based on a table “showing failure distributions of the cumulative number of failures experienced at two different time points” in [MIO87]. The darker curve denotes the probability of having found  $x$  number of failures after an hour of testing, while the brighter of the two graphs presents the probability after five hours of testing. As one might expect, it is more likely to uncover a higher number of failures during a five hour session than during a mere hour of testing.

In order to identify and subsequently remove any faults from the software prior to its release, faults need to manifest themselves in the form of detectable failures. Two main factors of contribution are listed in [MIO87] with regards to failure behavior:

- ✓ The number of faults in the software being executed
- ✓ The execution environment or operational profile of execution

The occurrence of failures is impacted by how long the software has been running, the environment in which it operates and, finally, the operational profile used during program execution, which in turn is influenced by the overall testing strategy.

### 3.3.4 Operational Environment

The environment of a software application is made up of external factors such as input to the system and output from the system, which describe the operational profile [MIO87]. Practically all software systems process some kind of input and produce output although it might not be visible to the end user. These variables of input and output may have a multitude of states, and as a consequence it is not feasible to test all of them. To accommodate for this, the amount of data to be processed is reduced to a subset of all possible states during testing. The states which variables may “enter” carry different probabilities of application during an operational phase. Thus, it is desired that state selection for use during testing corresponds to the state probabilities of actual operational use. The main intention is to ensure that tests mirror expected system usage once the software becomes operational.

Some systems are dependent on requested input to be of a particular format in order to function properly. In the case of unexpected input this may lead to failure for systems lacking robustness. Two classes of states can be deduced, namely valid and invalid states, which can be dynamically modified by conditions and other states. The values classified as invalid may in many cases have less likelihood of being put to use according to an operational profile, but are perfectly capable of causing failures if exception handling has not been implemented for erroneous input data. Although test data have been selected from an operational profile there is no guarantee that all possible input data from the entire input space will yield expected results. Thus, to be able to assess the precision of the reliability estimate one needs to make sure that chosen data cover the system’s input space well, in addition to using the operational profile as a basis for input selection.

Probabilities for assigning specific states to system variables are defined in the operational profile, hence providing a measure as to the probability of a given run being chosen during an operational phase. The cumulative probability of all possible runs will, logically, be 100%, but because of resource limitations it is highly unlikely that all combinations will be exercised during testing. A measure referred to as input space coverage is defined in [MIO87], which denotes the sum of all probabilities associated with executed test runs. For a given run taking place during the operational phase, this measure will state the



probability of the same run having been tested prior to operational use. When dealing with vast input spaces the measure will be accordingly modest.

As mentioned earlier, it is an infeasible task for most systems, possibly with the exception of simple and easy-to-understand applications, to perform tests covering the entire input space. In order to increase code coverage, techniques are employed which exploit the fact that a lot of input data are similar. Input exhibiting a certain level of similarity are hence grouped together and assigned a probability which is the sum of each individual input probability, which in turn refers to the likelihood of the current input being selected during operational use. A particular value or input is chosen from the input class to represent its entire group. This technique, referred to as equivalence partitioning in [MIO87] and exemplified in figure 3.4, causes a drastic reduction as far as the input space is concerned, but increases the risk of neglecting states that may result in system failure.

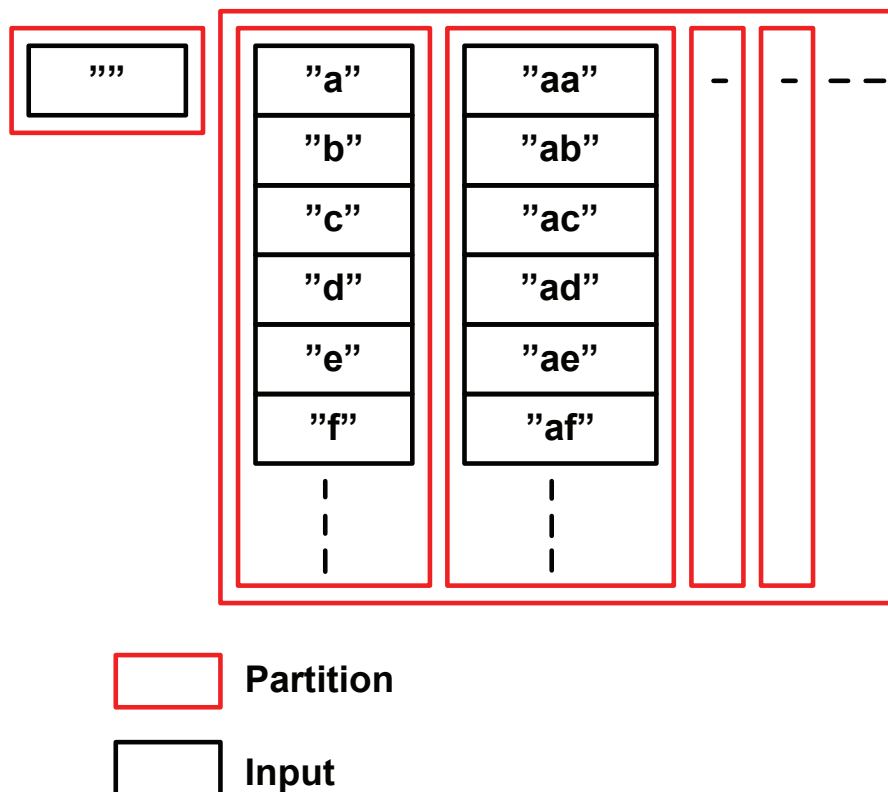


Figure 3.4  
An illustration of equivalence partitioning

All in all, the picture of what parts of the input space ought to be covered appears to be a rather blurry one. In [MIO87] it may seem contradictory to first emphasize a broad coverage of the input space, thus running additional tests to complement the operational profile, while later stating that in practice there is no passable way unless some kind of simplification technique is used, leaving considerable input data untested. If there is a conclusion to be drawn

from this matter it should be that the process of selecting input data for testing purposes needs thorough consideration, in order to satisfy input space coverage. A more dedicated and in-depth look at operational profiles is taken in chapter 4.

### 3.3.5 Reliability Modeling

Three factors are said to impact the reliability of software systems. The first factor is the introduction of faults in the code, which depends on characteristics of the development process as well as the resulting code, such as the size of code and developer tools and knowledge [MIO87]. The second factor is removal of faults in the code. Because software engineers know there is no silver bullet present that will prevent faults from entering program code as development progresses, the fault detection and removal processes need to be optimized. Fault removal is dependent on time spent on testing, operational profile and the quality of repair activities. Some of the revealed faults might not be eliminated or correctly removed as a result of unsatisfactory documentation of detected failures or lack of structure and clarity in the source code [MIO87]. The third and final factor is the environment, which directly governs the operational profile of the system. If the environmental representation of the operational profile turns out to be imprecise, it is set to affect reliability estimates more or less significantly.

Reliability representation	Definition
Failure intensity	Average number of failures per unit of time, typically per hour. The advantage of this measure is the fact that it requires only one number to be stated
Reliability	Probability of failure-free operation in a given time interval
Mean Time To Failure	Denoting the average number of a chosen time unit (minutes, hours, days) between successive failures in a system. Frequently used by hardware vendors, but less common in software reliability

Table 3.3: *Alternative representations of reliability*

[MIO87] claims that “*since some of the foregoing factors are probabilistic in nature and operate over time, software reliability models are generally formulated in terms of random processes*”. Existing reliability models are distinguished by the probability distribution of failure times and number of failures detected, as well as variations in the random process over time. A software reliability model specifies a general dependency between the failure

process and the aforementioned factors. Because reliability is defined with respect to time these models are referred to as time-based models. However, this does not rule out the possibility of alternative representations of reliability conveying useful information about software reliability. There are a multitude of ways of describing the failure process, and this fact is put to good use by the models.

The estimates generated by the reliability models can be applied in several ways. The most common representations of reliability are presented in table 3.3 above. Failure intensity is usually the preferred choice because it exists at all times and “*because they combine additively*” [MIO87].

### 3.3.6 Parametrizing the Model of Choice

A reliability model is not a mere function that can be put to use straight away. The models are presented as general mathematical expressions with non-defined parameters that depend on the particular system for which reliability estimates are being produced. Thus, the set of parameters associated with the model of preference has to be determined before its intended function can be performed. The process of determining the parameters can be accomplished in one out of two proposed ways [MIO87]:

- ✓ Estimation
  - parameters are determined by estimating a value based on failure data from early rounds of testing
- ✓ Prediction
  - parameters are established by considering properties of the system and the development process

Both of these approaches introduce added uncertainty to the numbers used in estimating reliability. Uncertainty is typically expressed in the form of confidence intervals for parameters used to create a specific instance of the chosen model – a model parametrized for a particular software application.

With a parametrized reliability model in place, [MIO87] lists the following findings possible:

- ✓ The average number of failures experienced at any point in time
- ✓ The average number of failures in a time interval
- ✓ The failure intensity at any point in time
- ✓ The probability distribution of failure intervals

It is desirable for a reliability model to possess a variety of properties. Among other things, a model should be capable of giving “*good predictions of future failure behavior*” and be “*based on sound assumptions*” [MIO87]. The initial

property can be viewed as elementary and a matter of course, since the fundamental intention behind a reliability model is to quantify the reliability of a software system. The second property regarding sound assumptions is somehow connected to the first property in that complete confidence can not be put in a model whose assumptions are not supported. Model assumptions generally relate to how the operational environment of a system, as specified in the operational profile, does not change, but is kept constant.

A reliability model may prove useful beyond generating reliability estimates for applications. Its ability to enhance communication is underlined in [MIO87], stating that high-quality models improve project communication and provide a common framework for increased understanding of the development process. Additionally, a model is capable of promoting visibility to management and other stakeholders. These are essential advantages even if the resulting predictions were to end up imprecise or useless at worst, especially considering the relatively modest resource consumption involved in making use of an existing model. Developing a reliability model from scratch, however, requires substantial amounts of theoretical work, tool building and accumulation of practical knowledge [MIO87].

### 3.3.7 Model Variants

Several reliability models exist that are geared towards software. The first models appeared during the early 1970s, and as of today well over 100 distinct models are published. This section is devoted to models fitting the black box category – models which view the system as a monolithic unit. White box models, on the other hand, regard software systems as a constituted set of modules. Each module carries its own reliability and the overall system reliability can be estimated by exploiting knowledge of how the individual modules execute. For the purpose of this report we have chosen to focus on black box reliability models, since models belonging to this category have been the ones mentioned explicitly in published articles on code coverage and reliability. Black box models distinguish themselves from one another by means of:

- ✓ Different statistical distributions are used in the models to represent the average number of failures experienced versus execution time
- ✓ Diverse assumptions are made
- ✓ Different factors are taken into account that are said to affect reliability

Some models are more renowned and more frequently employed than others, and several models are strikingly similar because their foundation is based on earlier models. A selection of principal models are presented briefly and in chronological order below. The presentation is merely intended to give a flavor of the respective elements of each model and the main assumptions they make. A more thorough approach is taken in chapter 5 with regards to the

Musa-Okumoto model, which will be utilized in considering the possibility of reliability model implementation in JCoverage.

#### 3.3.7.1 Jelinski and Moranda/Shooman (1971)

These practically identical models were two of the earliest reliability models to be developed. According to "*Software Reliability Engineering*" by Michael Rung-Tsong Lyu [Lyu05], they were basically published at the same time with no knowledge of one another and make a number of assumptions. Firstly, the number of faults in the code is assumed to be fixed prior to testing and fault removal is not expected to introduce new faults. Moving on, the number of machine instructions is assumed to be more or less constant and an operational profile ought to be present. Finally, the detection rate is presumed to be proportional to the remaining number of faults in the code. As a result, the model is said to have a "*hazard rate*" that decreases linearly with time [Lyu05].

#### 3.3.7.2 Littlewood-Verrall (1973)

According to "*A Survey of Software Reliability Models*" authored by Ganesh J. Pai [Pai02], the Littlewood-Verrall model is one of the most complex ones, requiring substantial resources and hence considered to be an expensive alternative. The model is referred to as a Bayesian software reliability growth model which consider "*reliability growth in the context of both the number of faults that have been detected and the failure-free operation. Further, in the absence of failure data, Bayesian models consider that the model parameters have a prior distribution, which reflects judgement on the unknown data based on history e.g. a prior version and perhaps expert opinion about the software*" [Pai02].

#### 3.3.7.3 Goel and Okumoto (1979)

The Goel-Okumoto model assumes that the average number of failures experienced can be described by a Poisson distribution and that the expected number of failures observed is a finite number in infinite time. Further, the amount of failures discovered is assumed to be constant with respect to time, as was the case with the Jelinski and Moranda-Shooman models [Pai02].

#### 3.3.7.4 Basic Musa (1979)

The same year in which the Goel-Okumoto model was introduced also saw the emergence of the Basic Musa reliability model. The main assumption of this model is that the number of failures observed at a time  $t$  is finite and follows a Poisson process. Contrary to existing models at the time, execution time is used in the estimation process. Execution time between failures is

assumed to be exponentially distributed and the number of failures encountered is taken to be constant with regards to time [Pai02].

#### 3.3.7.5 Musa-Okumoto (1984)

The Musa-Okumoto model, appearing five years after both Okumoto and Musa launched their initial reliability models, employs execution time in estimation, as was the case with the Basic Musa model. The Musa-Okumoto model, exhaustively described in [MIO87] and known as the "*logarithmic Poisson execution time model*", predicts fewer failures to be found as time and testing progresses. The number of failures is, contrary to the Goel-Okumoto model, not considered to be a finite number as time approaches infinity.

### 3.3.8 Reliability Model Usage

For smaller projects, estimation is likely to yield imprecise results, and hence the use of reliability models is generally not recommended. For more comprehensive projects it is important to take into account the assumptions made by the available models and to ensure consistency between the assumptions and system data; for instance deciding whether data obtained from executed tests fit the probability distribution of the chosen model. As [Pai02] points out, there exists no universal model that can be completely trusted in all possible situations, since it is virtually impossible to determine what factors affect the correctness of any model.

Software reliability models are used for a number of purposes. One intention can be to perform retrospective estimation to determine achieved reliability from one specific point in time to the present. An alternative target can be reliability prediction by parametrizing chosen models and utilizing available data. With the existence of numerous models it might be tempting to employ all of them and subsequently compare the results to obtain a realistic estimate. This strategy is feasible for objects of research, but the use of more than two models for real-life projects is seen as economically impractical [MIO87].

*"If you think you can, you can. And if you think you can't, you're right."*  
Henry Ford

## CHAPTER 4:

# Code Coverage and Reliability

Now that we have covered the essentials of code coverage analysis and taken a look at some issues of reliability and estimation through the use of reliability growth models, it is time to consider the relationship between code coverage and reliability. We begin this chapter by defining some critical terms within this context, before moving on to operational profiles and the saturation effect – two pivotal factors when it comes to testing and software reliability in general. With these elements in place, we further seek to uncover what findings and conclusions have been made with respect to the relationship between code coverage and reliability.

## 4.1 Term Definitions

In order to add clarity and avoid ambiguity in the later sections of this chapter, we find it useful to define and explain a few essential terms and concepts. According to IEEE, reliability is defined as *"the ability of the system or component to perform its required functions under stated conditions for a specified period of time"*. A more mathematical interpretation defines reliability as the probability that no error will occur in a given time interval:  $\text{Reliability} = P(\text{no error in } [0, t >))$ .

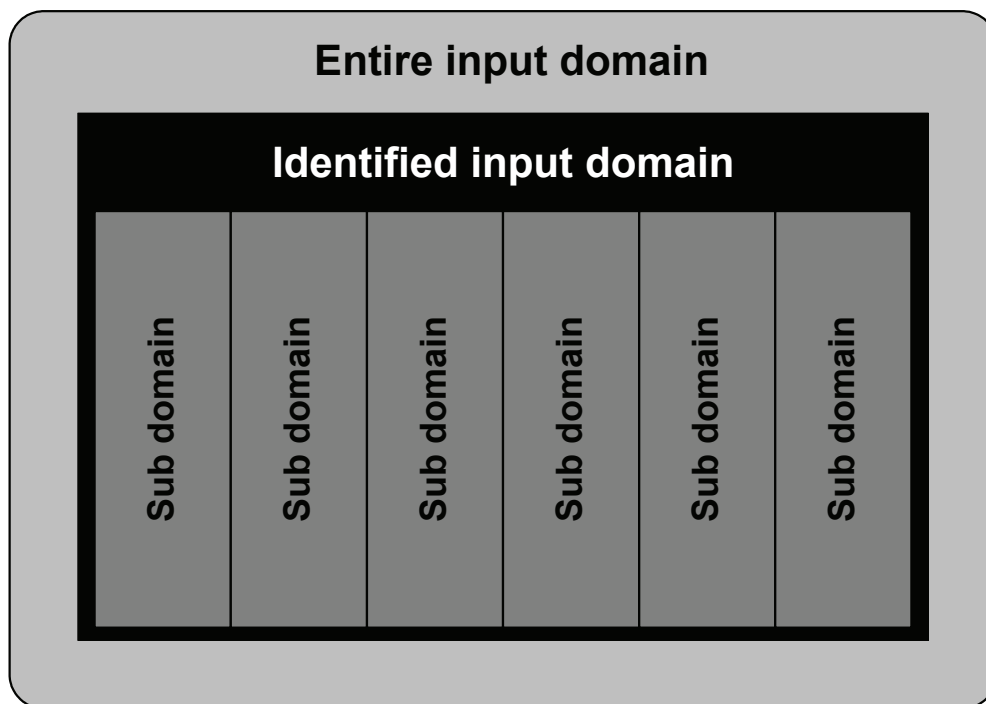
However, some articles such as [Ham94], find it necessary to distinguish between reliability and dependability – a formal notion of trustworthiness, by claiming that an application may well be trustworthy if failures occur that are of negligible importance, yet these failures will affect reliability negatively. On the other hand, a potentially catastrophic failure may influence reliability insignificantly if it occurs in low-usage functionality, yet it is destined to affect the user's confidence in the software greatly. We will get back to the issue of dependability, as well as operational profiles, later in this chapter.

Another term frequently referred to, is testability. Statements that are easy to reach are said to have high testability, whereas statements that will only get exercised in rare scenarios, such as error management code, typically have low testability [Mal+94]. Hence, pieces of code or statements with high testability are likely to be covered by running a moderate number of tests. However, a fault in the source code needs to result in an observable failure during testing in order for testers to detect it, and this is not always the case. According to [Ham94], a software program with high testability will be seen

to fail if capable of failing. The same article believes that testability might be a practical way of measuring dependability by using sub-domains of coverage testing as a basis for testability measurements.

## 4.2 Operational Profiles

Operational profiles have come to play a vital part in testing and software reliability engineering. An operational profile strives to mirror actual system usage by allocating realistic probabilities “to various subdomains of the identified target input domain”, according to Garg Praerit in his article titled “*Investigating Coverage-Reliability Relationship and Sensitivity of Reliability to Errors in the Operational Profile*” [Gar94]. The majority of existing time-based reliability models rely on testing being executed in accordance with an operational profile of the system. However, as we will see, operational profiles pose some intricate challenges as well. A conceptual division of the input domain is captured in figure 4.1.



*Figure 4.1*  
*The operational profile deals with*  
*the identified input domain*

### 4.2.1 Purpose of Operational Profiles

The intuitive purpose of an operational profile is to be able to perform realistic testing that resembles how the end product will be utilized. By acquiring



knowledge of an operational profile one is able to better understand what test cases to apply and in what order. Based on this, testing should be able to detect failures and their respective faults in accordance with their occurrence frequency. According to John D. Musa in his book *“Software Reliability Engineering: More Reliable Software Faster And Cheaper”* [Mus04], the approach of employing operational profiles *“rapidly reduces failure intensity as test proceeds, and the faults that cause frequent failures are found and removed first”*.

The Testing Standards Working Party and its *“Reliability Guidelines”* [TS04] emphasizes the usefulness in guiding system testing so that the most critical parts and features of the application have benefited from extensive effort and hence contributed to optimizing reliability by finding faults in high-usage functions. Additionally, the assignment of probabilities to functions in the software can drive resource management through both development and testing. Another feature of operational profiles is how operations easily map to use cases and thus fits in neatly with modern-day object-orientation. For some applications it might prove feasible to develop supplementary operational profiles that are tailored to different modes of operation and their criticality [TS04].

## 4.2.2 Problems and Challenges

Most time-based reliability models, such as the well-known Musa-Okumoto model, presume that testing is performed based on an operational profile that mirrors expected system use. Inevitably, this approach might leave parts of the code untested – code that could potentially contain faults resulting in failures. Thus, the accuracy of the operational profile is essential. If the match between operational profile and actual system usage turns out to be unsatisfactory, the parts of code and functionality not exercised or only superficially tested could turn out to be considerably more important than what the profile suggested. Thus, reliability estimates are at the mercy of profile precision.

There are, however, situations where an operational profile is unknown. If a completely new software application is being developed, an operational profile would be unavailable. Other systems might not have an existing profile because it seems infeasible to estimate it, or that the prospect of developing one – facilitating data collection and subsequent analysis – would prove too costly. According to [Gar94], an existing operational profile might turn out to be insufficient in estimating the reliability for new or considerably modified software. An experiment was performed that sought to determine the sensitivity of reliability to errors in an operational profile. Two profiles were developed and both were employed while testing a UNIX program that contained ten injected faults. The experiment revealed that two different operational profiles resulted in vastly differing outcomes when testing the same program. Hence, there is little doubt that, because of the sensitivity,

incorrect estimates in the operational profile may lead to drastically misleading reliability estimates

In their article "*On Software Reliability and Code Coverage*", Richard M. Karcich, Robert Skibbe, Aditya P. Mathur and Praerit Garg [KSMG96] highlight the "danger" in relying on incorrect or inaccurate operational profiles when choosing a set of tests. They rightfully claim that undiscovered faults may lurk in parts of code that will remain untested because of the profile and, as we pointed out earlier in this section, instances of inaccuracy could result in frequent user execution of poorly tested functionality. The conclusion has to be that if a company opts for operational profiles as a corner-stone in testing efforts, they better make sure they have the necessary prerequisites and resources to attain a satisfactory level of precision. Developing operational profiles half-heartedly will most likely result in a less reliable product than if testing had been undertaken without the presence of a profile.

### 4.3 Reliability Overestimation

Existing reliability growth models tend to overestimate the reliability of software. One of the main contributors to overestimation is the saturation effect. In this section we will look at the latter effect and a proposed way of how code coverage can help to make reliability estimates more accurate.

#### 4.3.1 Saturation Effect

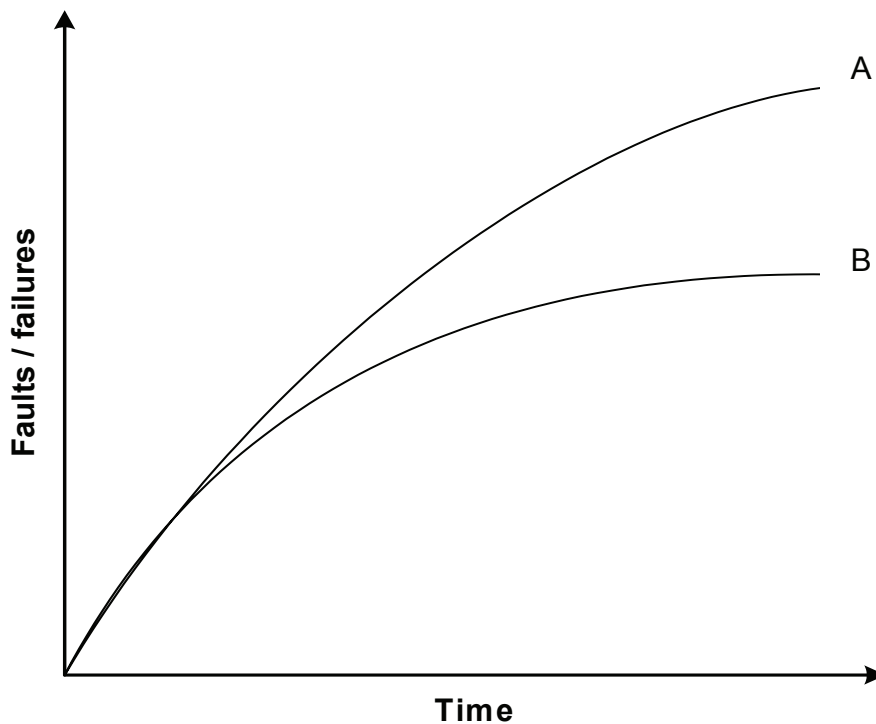
Aditya P. Mathur and Vernon J. Rego claim that functional or structural testing methods suffer from the phenomenon of saturation effect, in their article "*White-box Models for the Estimation of Software Reliability*" [MR96]. This effect refers to limitations of the functional testing methods in revealing faults in the tested application. Typically, as testing proceeds fewer and fewer faults are being discovered, and some faults are destined to survive no matter how many tests are applied. The reason for this is that remaining faults may hide in parts of the code that, according to the operational profile, represent low-usage functionality, in combination with tests being incapable of revealing all faults. Chapter 4.4.3.2 will look at how the use of mutation coverage can demonstrate the inability of tests to discover all faults in the code. As a result, the fault-detecting ability of functional testing is said to saturate with time and effort spent on testing. This is illustrated in figure 4.2, where curve *A* represents fault-detecting behavior as seen by most reliability models, whereas *B* mirrors actual or more realistic progress as testing proceeds.

According to Mei-Hwa Chen, Michael R. Lyu and Erik Wong in the article "*Effect of Code Coverage on Software Reliability Measurement*" [CLW01],

empirical studies suggest that overestimation exists because of the reliability growth models' inability to take the saturation effect into consideration. These models predict an increase in reliability as more time and effort is spent on testing, no matter the increase in number of faults detected or the improvement in code coverage.

### 4.3.2 Pre-process Model for Improved Reliability Estimates

Time-domain reliability models use failure history obtained during testing to predict program behavior, often with contribution from one or more operational profiles [CLW01]. However, as we have seen, developing an accurate operational profile and dealing with the saturation effect of functional testing methods pose significant challenges. According to [CLW01], empirical and analytical studies reveal an over-optimistic tendency in reliability estimates. Clearly, there is a need for techniques that pre-process test data before passing them on to the reliability models presented in chapter 3, with the purpose of producing improved estimates.



*Figure 4.2*  
*Illustration of the saturation effect*

Time-domain models estimate reliability based on the failure rate and the time spent on testing. Hence, they predict an increase in reliability when running tests that do not discover additional failures. As testing progresses and faults causing failures are removed, the time interval between successive failures increases, as do the estimates from reliability growth models. The risk of

overestimation grows the more redundant a testing effort turns out to be. In order to deal with this challenge, an approach is recommended that uses code coverage information to adjust failure rates prior to reliability estimation. [CLW01] refers to empirical studies suggesting that fault detectability statistically correlates with code coverage, and hence that software reliability correlates with code coverage. Intuitively then, code coverage information can be exploited to assess the effect of a particular test case. Time intervals between failures are modified for testing efforts that are redundant with respect to a chosen coverage criteria, with the intention of reducing their influence on results produced by time-domain reliability models [CLW01]. Test cases that neither encounter failures, nor increases code coverage are considered ineffective.

We have chosen to follow the test case definition given in “*Testing Applications on the Web, Second Edition*” by Hung Q. Nguyen, Bob Johnson and Michael Hackett [NJH03]. They consider a test case to be “*a test that (ideally) executes a single well-defined test objective (e.g., a specific behavior of a feature under a specific condition). Early in testing, a test case might be extremely simple; later, however, the program is more stable, so you will need more complex test cases to provide useful information*”. Since so-called ineffective test cases contribute to an increase in the estimated reliability, even after test case data have been pre-processed and hence been given a reduced effect on reliability, this technique mainly strikes us as a means of compelling testers to further testing. Thus, the practical outcome of employing this approach appears to be an expanded test set which may include supplementary tests that uncover no additional failures.

[CWL01] views time and code coverage as essential factors when predicting failures, and consequently combines them to extract effective testing efforts. A measure,  $\rho$ , is used to denote the effective part of the execution time for a given test case, and is computed based on the relative increase in time and code coverage. An experiment is referred to that was performed in a simulation environment, where reliability overestimation from the renowned Goel-Okumoto and Musa-Okumoto models was shown to decrease significantly by considering effective testing efforts. Moreover, the study confirmed that adjusted values for testing efforts based on code coverage information resulted in far more accurate estimates from reliability growth models when compared to the actual, observed reliability [CLW01].

## 4.4 The Code Coverage – Reliability Relationship

**T**he extensive use of failures detected during testing as an indirect measure of reliability requires strong assumptions about test cases and testing in general. Hence, employing code coverage to estimate reliability seems like a more direct approach, in the sense that thorough, well-

covered testing is assumed to decrease the likelihood of experiencing failures and hence contribute to the trustworthiness and reliability of software. The use of code coverage also solves problems related to data collection and assumptions regarding test case distribution. Intuitively, a causal effect will exist between code coverage and defect coverage. Since both code coverage and defect coverage increase with test intensity or time, it is, however, hardly surprising that empirical data suggest the presence of a relationship [BP00]. A wealth of articles propose their own models and present findings and results – both positive and negative – from empirical investigation and theoretical analysis. In this section we try to summarize models encountered and conclusions reached within the code coverage – reliability relationship, based on available publications.

#### 4.4.1 Models for Relating Code Coverage to Reliability

A few selected models will be presented below to give an overview of ways of connecting code coverage to software reliability. The model descriptions aim at giving an overall impression of the models and their feasibility, while specific details and the determination of parameter values are left to the respective articles.

##### 4.4.1.1 Node-based Reliability Model

Pankaj Jalote and Y. R. Muralidhara describes a coverage-based model in their article “*A Coverage Based Model for Software Reliability Estimation*” [JM94]. The model bases its estimation on the coverage history of a program and the fact that most software applications consist of several modules. A program is represented as a flow graph where each node corresponds to a module. The number of times that each of the modules are being exercised is registered and used to estimate module reliability along with the runtime during system testing. The reliability of a node is assumed to increase in step with the number of executions. In the example flow graph depicted in figure 4.3 below, the edges connecting the nodes represent a possible transfer of control and carry a computed weight denoting the probability of a control transfer taking place from node  $a$  to node  $b$ . These probabilities are then used to determine the overall reliability of the system. Logically, the sum of all edges going out of a node should be 1.0.

The reliability of a node is given by the following equation:

$$R_i(t) = e^{(-\lambda_i t)}$$

Here,  $t$  denotes the time spent in the system, while  $\lambda_i$  reports the average failure rate of module  $i$  with respect to time [JM94]. Total system reliability is obtained by traversing different paths, preferably in accordance with an operational profile, where path reliability is computed as the product of all

reliability values of the nodes or modules it consists of. Each path going from start to finish represents a valid execution of the program. According to [JM94], the two parameters which require value assignment can be viewed as reasonably constant throughout, assuming that all nodes have comparable sizes and that similar development techniques have been employed. These parameters are, to a large extent, a function of general software properties and hence relatively stable within the organization undertaking development projects.

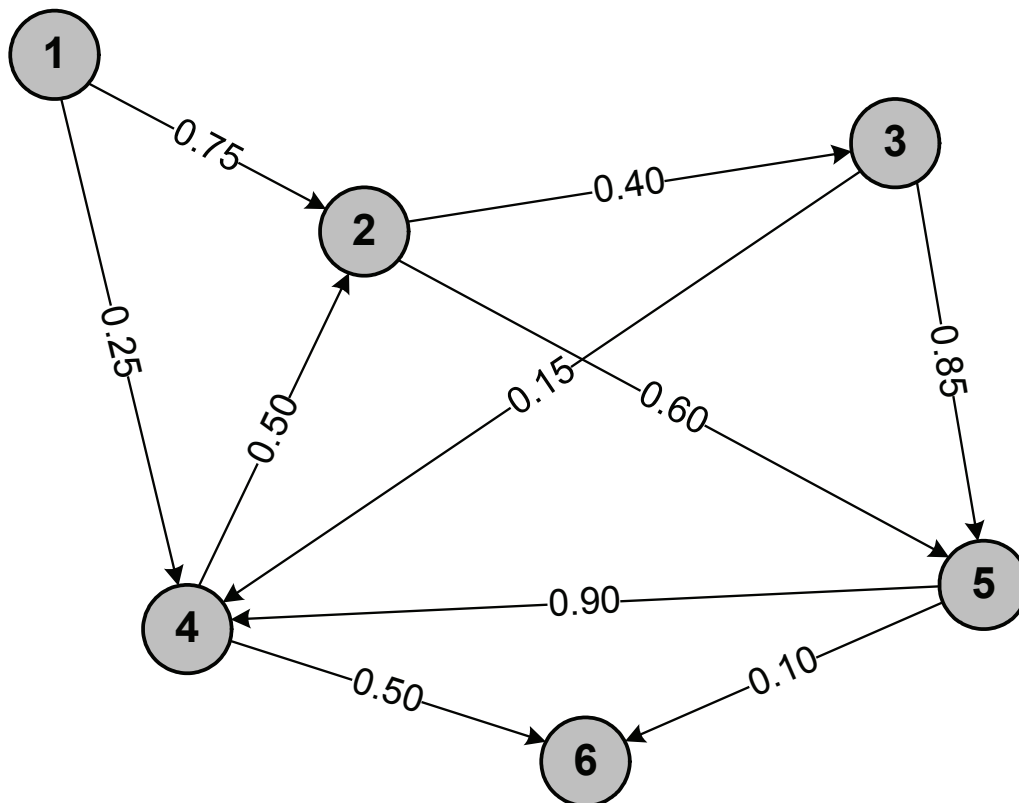


Figure 4.3  
Node representation of software

[JM94] puts an emphasis on how this coverage-based technique seems more effective and cost-effective compared to most existing reliability models, which perform random testing until additional failures have been discovered. Having computed the reliability estimates for the software, testers can decide whether sufficient testing has been done with respect to the entire system or specific modules. The approach is considered practical because of how achieved coverage directly estimates reliability and thus requires less extensive data collection. On the downside, they have no experiments verifying the suitability of the model. We also question whether the article actually refers to code coverage in the way that we define the term. The model assumes that node failures are independent of failures in other nodes, justified by the hypothesis that this is the case with large modules or modules of a

certain size. Thus, for smaller modules this independence might not be present. The fact that coverage is measured based on modules as opposed to other code elements might not have such an impact. However, it puzzles us how the share of tested code, or modules in this case, appears to be neglected. Instead they register the number of times a module is tested and use this information as a basis for reliability estimation. This strikes us as a measure of test intensity more than of code coverage.

#### 4.4.1.2 Logarithmic-exponential Model

A logarithmic-exponential approach is presented in [MLBK02] that models the relation between time spent during testing, code coverage and reliability. In accordance with what was written in [JM94], the article sees code coverage as a direct measure of how thoroughly a system has been exercised, in contrast to the traditional measure of test intensity. This fact, along with the emergence of tools that are capable of tracking coverage measures automatically, suggest that the relationship between code coverage and reliability deserves renewed attention. Additionally, developing a model that relates code coverage to reliability opens up the possibility of computing estimates with respect to defect density.

The proposed model employs the logarithmic growth model of Musa and Okumoto, and assumes that code coverage follows a logarithmic 2-parameter model which can be transformed to a 3-parameter model if feasible. It is based on the hypothesis that different parts of the code have different probabilities of being executed during testing – just as some faults and failures are less likely to be detected than others. In this way [MLBK02] wants to relate a measure of code coverage to a measure of defect coverage. The model takes into account that not all faults need to be found at 100% coverage, since *“full statement coverage can be reached before full branch-coverage because of the subsumption hierarchy”*. Test sets should not be randomly chosen from a distribution based on an operational profile, but rather selected with the purpose of driving testing towards input and program components with a greater likelihood of faults and failures being present. This approach is expected to speed up detection of failures and the underlying faults causing them. Having executed program code with input corresponding to a chosen distribution, a reliability growth model can be used to predict the effort needed to satisfy product reliability requirements [MLBK02]. However, candidate reliability models must be able to estimate reliability based on tests that do not employ a data subset of the operational profile.

The article highlights three factors that are considered crucial to the attained reliability. These factors are:

- ✓ Test strategy employed
- ✓ Amount of time spent during testing
- ✓ Testing of low-usage modules

As far as time spent during testing is concerned, this has to be measured as execution time instead of calendar time. Parallel testing and automated testing tools render possible the execution of considerably more tests and operations in the software than what would be the case in an operational phase. The last of the three factors is concerned with how thoroughly low-usage functionality has been tested. This typically includes code elements such as error-handling routines. Although hard to test, they are critical components of the software that require high reliability. Interestingly, the importance of testing low-usage modules and routines thoroughly seems to be a matter of great dissension. Whereas [MLBK02] emphasizes the significance of testing these areas properly, others seek to lower the priority of low-usage testing and hence focus on the parts of the application that are more frequently exercised.

The reason why a logarithmic growth model is chosen is the defect detection behavior when performing non-random testing – that is, selecting a test case with the intention of exercising untested functionality. Parameter interpretation is a challenge; however, logarithmic models have two advantages when used to describe testing efforts and enumerables covered [Mal+94]:

- ✓ Superior prediction of number of defects
- ✓ Accounts for 100% coverage achieved in finite time

The achieved coverage is, as mentioned earlier, not solely dependent on the number of tests applied. The distribution of testability values for several enumerables must be taken into account as well [Mal+94].

#### 4.4.1.3 Hyper-geometric Distribution Model

In the article titled “*Test Coverage Dependent Software Reliability Estimation by the HGD Model*” by Raymond Jacoby and Kaori Masuzawa, a software reliability growth model is presented which is capable of making “*estimations for various kind of real observed test and debug data*” [JM92]. A parameter referred to as the “ease of test” function plays a pivotal part in the HGD model. This function denotes the number of faults uncovered by a certain test instance. Apparently the model does not assume that faults found in previous test instances are removed or corrected prior to the execution of the next test instance. Thus, faults discovered by a particular instance may well have been detected earlier during testing.

The “ease of test” measure is estimated based on an estimate of the initial number of faults in the software and a function representing the effort spent



on testing. In [JM92] code coverage is then integrated into the function of estimating testing efforts. The following “ease of test” function is proposed:

$$W(i) = E[m] \cdot C_i \cdot tc(i)$$

In this equation  $E[m]$  is an estimate of the number of faults residing in the application, while  $tc(i)$  denotes a function showing a linear progression trend in code coverage, returning values in the interval 0 to 1. According to [JM92], other functions might yield a more realistic picture of the progress. The remaining parameter in the equation above,  $C_i$ , is a constant suggesting how good the tests are at discovering faults. As we already know, there is no guarantee that faults will be detected even though the code lines causing them are being executed. As a consequence, this constant is a measure of how well the tests have been designed to intercept existing faults, and hence  $C_i$  is assumed to be assigned values between 0 and 1. Taking a closer look at the “ease of test” expression, a test with 0% code coverage will, according to the function, end up not uncovering any faults, as expected. And even with 100% code coverage the number of discovered faults will be dependent on the constant  $C_i$ , which is also to be expected.

#### 4.4.1.4 Detectability Profile Model

A method described in [Mal+94] estimates several code coverage measures based on detectability profiles. The rationale for a detectability profile is that different code lines, functions, blocks of code – often referred to by the generic term enumerables – have different probabilities of being executed by randomly generated tests. The probability of each enumerable being exercised depends on the parts of code in which it is “wrapped”. Enumerables that are hidden in code implementing low-usage functionality according to an operational profile, will typically have low detectability. In many ways detectability relates closely to testability, as defined in chapter 4.1 earlier. However, detectability takes into account the probability of the code containing faults actually causing an observable failure during testing.

Detectability profiles can be employed as a means of estimating code coverage after a given number of tests have taken place. This would assume random generation of test sets based on an operational profile, along with an established detectability profile. Unfortunately, the creation of an accurate detectability profile requires substantial work. By conveying information on when given levels of code coverage will be achieved during testing, the concept of a detectability profile does not directly improve reliability estimation. On the other hand, one might argue an indirect effect on estimating software reliability if code coverage can be verified to improve predictions. The main features of this and other models outlined in this chapter are summarized in table 4.1.

<i>Model</i>	<i>Characteristics</i>	<i>Strengths &amp; weaknesses</i>
Node-based	<ul style="list-style-type: none"> <li>- estimation based on coverage history</li> <li>- flow graph representation</li> </ul>	<ul style="list-style-type: none"> <li>+ less extensive data collection</li> <li>÷ no verifying experiments</li> <li>÷ assumes independent node failures</li> </ul>
Logarithmic-exponential	<ul style="list-style-type: none"> <li>- Musa-Okumoto reliability growth model</li> <li>- relate a measure of code coverage to a measure of defect coverage</li> <li>- driving testing towards input and components likely to contain faults</li> </ul>	<ul style="list-style-type: none"> <li>+ superior defect prediction</li> <li>÷ difficult parameter interpretation</li> </ul>
Hyper-geometric	<ul style="list-style-type: none"> <li>- ease-of-test function based on testing efforts and initial number of faults</li> <li>- code coverage included in estimations of testing effort</li> </ul>	<ul style="list-style-type: none"> <li>÷ nontrivial determination of constants</li> </ul>
Detectability profile	<ul style="list-style-type: none"> <li>- detectability profile generation for various enumerables</li> <li>- use of operational profile</li> <li>- a means of code coverage estimation</li> </ul>	<ul style="list-style-type: none"> <li>÷ accurate detectability profile requires extensive processing</li> </ul>

Table 4.1: *Summary of models incorporating code coverage*

#### 4.4.1.5 Influential Factors to the Coverage – Reliability Relationship

Judging from existing literature on the relationship between code coverage and reliability, the findings with regards to impacting factors appear to be more suggestive than resolute. [KSMG96] poses the question of whether conclusions stemming from experiments conducted on smaller programs, are valid for greater and more complex applications. Hence, both size and complexity should be viewed as non-negligible factors until experiments conclude the opposite. The same article suggests that reliability estimates could end up imprecise and at worst misleading, if the number of people involved in the development process is relatively small. Then again, defining

what numbers are small and what numbers are not, is likely to vary by context and company.

Another factor frequently mentioned in articles is the application domain. [FGMP95] concludes that the domain does not appear to have any significant impact on the code coverage – reliability relationship, but instead assumes that factors such as code complexity, as already noted, might play a more pivotal role.

## 4.4.2 Experimental Results and Conclusions Reached

In this section we will look at what positive findings have been made based on experiments described in reliability literature. For now, the focus will be on conclusions, hypotheses and reasoning that are in favor of an existing relationship between code coverage and reliability. However, the center of attention will be the results obtained while giving a proper explanation as to how they came about, and not detailed figures of the multitude of experiments conducted.

### 4.4.2.1 Code Coverage – Reliability Correlation

[Gar94] is one of the articles that produced positive results with respect to the aforementioned relationship. In this experiment ten faults were injected into a software program with the purpose of investigating the sensitivity of reliability to operational profiles. In addition to concluding that two different operational profiles provided vastly contrasting results from testing the same program – as was mentioned in chapter 4.2.2 – reliability was seen to increase in step with code coverage. Although reliability had a monotonic increase as code coverage increased, the variance between different coverage measures was significant. Code coverage was also seen to increase the more faults were discovered in the sample application. Furthermore, both reliability and code coverage showed similar staircase growth curves with respect to remaining faults in the program, thus signaling that neither coverage, nor reliability will necessarily increase as more faults are found. This does not imply the view that the number of remaining faults in a program equals its reliability, but rather that this number is assumed to affect reliability to a certain extent. On the other hand, reliability was shown to increase when code coverage increases – a fact that contradicts the theoretical foundation of existing reliability models, namely that reliability grows with each fault uncovered. However, such a hypothesis needs confirmation by taking into account the various application domains and parameters such as fault density, fault distribution and error types of other software programs [Gar94]. Although the results obtained show a strong correlation between actual reliability and code coverage, there is no foundation for generalizing the results until further experiments have taken place.

The issue of generalizability of experimental results within the field is a point of concern in [KSMG96]. The authors ratify that several experiments have been carried out within the topic, however, they remain sceptical as to the external validity of these experiments. Typically, the programs selected have been small and relatively simple when comparing them to commercial software products. Development environment is also launched as a potential factor of impact on reliability estimates, in addition to the ones mentioned in chapter 4.4.1.5. Smaller programs are on average developed in smaller, less complex development environments than what is the case with commercially developed systems. Hence, [KSMG96] conducted an experiment with a C program containing an amount of code lines in the hundred-thousands range, and constructed in an environment consisting of several full-time developers.

Data analysis was performed with two primary intentions in mind; to establish the predictive accuracy of the Musa-Okumoto and Goel-Okumoto reliability models, and determine the correlation between errors in prediction and changes in coverage. [KSMG96] employs the term Mean Test Case To Failure – MTCTF, and seeks to predict MTCTF for future weeks based on failure data from previous weeks. This measure is computed by dividing the number of test cases performed in a given period, by the relative increase in failure count. Results obtained from the experiment indicated that an increase in at least one coverage measure was accompanied by an increase in the actual MTCTF. Also, the difference between estimated reliability stemming from the Musa-Okumoto model, and the actual reliability, was seen to vary significantly throughout the testing process. Errors in predicting MTCTF seemed to follow at least one of the coverage measures. Thus, if estimation errors increased, then so would coverage values, however, *”when the coverage measure does not increase or change significantly, the error decreases due to the data tracking ability of the Musa-Okumoto model”* [KSMG96]. Any lack of precision could, according to the authors, be a result of not using an operational profile in the experiment.

With reference to the latter experiment, the Musa-Okumoto reliability model does not seem to be capable of making MTCTF estimates sufficiently precise. This fact is likely due to the model’s lack of knowledge as to how thoroughly a system has been tested up until now, and hence, an increase in module coverage may result in a drastic increase in the number of failures, yet the reliability model will fail to predict this [KSMG96]. During reliability estimation, then, improved MTCTF estimates seem likely to result from considering coverage in addition to failure data. Statistical correlations between the error in MTCTF predictions and the chosen coverage measures – module coverage and branch coverage – were computed to 0.76 and 0.60 respectively. This leads to the hypothesis that high MTCTF estimates from reliability models, along with relatively low coverage, indicate an over-estimation which, in turn, could lead to a non-negligible risk for software requiring exceptional reliability.

#### 4.4.2.2 Fault Removal Behavior

Fabio Del Frate, Praerit Garg, Aditya Mathur and Alberto Pasquini seek to uncover possible connections between code coverage and reliability in their article "*On the Correlation between Code Coverage and Software Reliability*" [FGMP95]. An empirical investigation is performed based on a four-step methodology. In short, these four steps are:

- 1) Selecting software programs for use in the experiment
- 2) Generating operational profiles
- 3) Creating a set of faults and preparing fault seeding of the chosen programs
- 4) Generating data for code coverage and reliability, respectively

The first two steps are reasonably straight-forward to comprehend. In step 3 numerous faults are injected into the program source code based on previously created fault sets. The final stage of the methodology implies that the reliability of a program has to be measured for each operational profile, every time a fault is detected and subsequently removed. Furthermore, the current program version has to be executed "*on test data generated randomly from the selected operational profile until a failure occurs*" [FGMP95]. The fault responsible for the largest number of failures must be identified and then removed, thus resulting in a new "version" of the program.

For each of the selected programs, reliability and code coverage values were measured after each fault had been removed. Results obtained indicate that code coverage may decrease, increase and even remain unchanged upon fault removal. One plausible reason for code coverage to decrease having removed a fault from the source code, is a phenomenon referred to as fault masking. Fault masking implies one fault in the source code preventing a second fault from being detected. Once the first fault, in this case, is removed, a marginally smaller part of code will be executed, hence resulting in lower code coverage. [FGMP95] takes the following approach when explaining fault masking, where  $P$  represents a random program: "*There is no test case in the input domain of  $P$  that will reveal  $f_2$ , and there is at least one test case in the input domain of  $P$  that will reveal  $f_2$  after  $f_1$  has been removed*". Another explanation for decreasing coverage could be that the fault removal requires additional code to be made. If the existing test cases fail to exercise newly added code portions, a coverage reduction will be the outcome. This will have a greater impact on coverage percentages if the number of code lines added are many when compared to the total size of the program.

Analysis also showed that reliability can go in all directions upon removing a fault, as was the case with code coverage. Decreasing reliability may also be a consequence of fault masking. By removing a fault, a subsequent fault might

get exposed and thus lead to the program failing on a more frequent basis. However, based on data observations [FGMP95] concludes that *"an increase in code coverage is always accompanied by an increased or unchanged reliability"*. Furthermore, observations of code coverage and reliability measures increasing, decreasing or remaining unchanged when faults are removed, along with an increase in code coverage resulting in unchanged or increased reliability, are claimed to be independent of software complexity measures.

Finally, the statistical correlation between code coverage and reliability was computed to be in the range of 0.89 to 0.99 for larger programs. However, the statistical correlation varies significantly for smaller programs, whereas larger, more comprehensive software applications produce higher and more stable correlation values. In spite of intensive program executions, [FGMP95] reports that none of the coverage measures employed reached their maximum levels.

### 4.4.3 Critics and Experimental Weaknesses

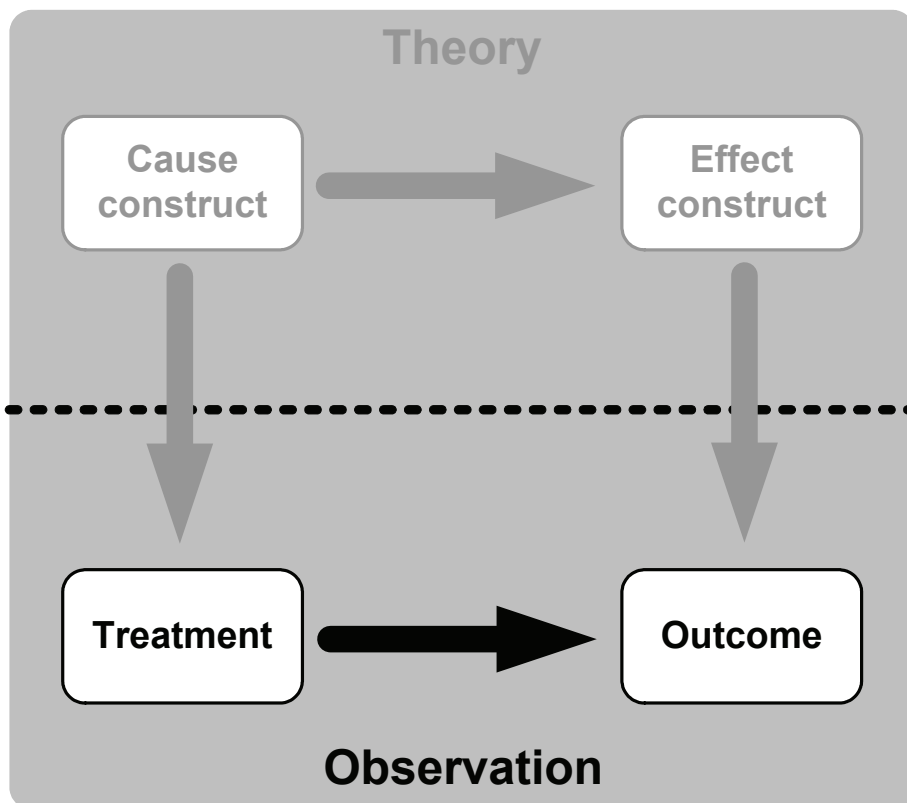
Not all published articles report positive findings as far as the code coverage – reliability relationship is concerned. This section will take the view of the ones who are critical to the validity and value of some of the experiments performed, and brings up a few questions that are seemingly left unanswered. The two main issues revolve around a lack of prioritization as to what parts of the code are important and less important, and how much of achieved reliability is down to code coverage, and not test intensity solely.

#### 4.4.3.1 The Effect of Test Intensity on Reliability

High code coverage is claimed to achieve high defect coverage, implying that more defects are found in software during testing. This, in turn, is believed to improve the quality of testing efforts and contribute to better end product quality. [BP00] discusses the matter of whether the claimed relationship between code coverage and defect coverage is genuine and existing. The question is whether there are other variables or factors that might impact the aforementioned measures, that have not been taken into account in empirical investigations and experiments performed. [BP00] correctly points out that code coverage as well as defect coverage typically increase with time and effort spent on testing the software, and hence it is only logical that empirical experiments confirm the existence of such a relationship. Focus is therefore directed to the internal validity of these experiments and whether a third factor unaccounted for, is interfering with the actual outcome. In the book *"Experimentation in Software Engineering: An Introduction"* by Claes Wohlin et al. [Woh+00], the issue of internal validity is explained this way: *"If a relationship is observed between the treatment and the outcome, we must make sure that it is a causal relationship, and that it is not a result of a factor*

of which we have no control or have not measured". The role of internal validity in an experiment is depicted in figure 4.4 below.

In order to prove the correlation between code coverage and defect coverage, test intensity has to be accounted for in experiments, or put differently – one has to determine whether the combined effect of test intensity and code coverage is capable of explaining defect coverage variations better than test intensity alone [BP00]. A procedure is presented that assumes testing to not be coverage-driven, since that would affect the design of test cases and thus make it infeasible to separate test intensity from coverage. The procedure requires a sample of projects where defect coverage and code coverage data exist that correspond to several test intensity values, such as number of test cases and testing efforts. One should keep in mind that similar test intensity values are not necessarily comparable across software systems of different sizes. If code coverage plays a considerable role, simulated samples are expected to, on average, show a poorer statistical correlation than what is the case for the actual sample, where the effects of both test intensity and code coverage should be visible on the resulting defect coverage.



*Figure 4.4*  
*Internal validity represented as the black arrow between treatment and outcome*

In [BP00] an experiment was conducted where code coverage and defect coverage was measured in three testing phases. For each of these phases, the same level of test intensity was applied and each and every software application was exposed to the same set of test cases. The conclusion of the experiment was that results obtained did not support the hypothesis that code coverage carries an additional and pivotal effect on defect coverage, when test intensity is already accounted for. Instead, [BP00] finds it reasonable to assume that both code coverage and defect coverage are driven by test intensity only, but opens up the possibility of factors such as defect distribution, defect types and differing environments impacting the end result.

#### 4.4.3.2 Absence of Operational Profile Leads to Unweighted Testing

[Ham94] takes a different approach in criticizing the claimed effect of code coverage on reliability, starting off by explaining two primary ways of developing a test set. Firstly, a best practice for performing code coverage testing is described that guides the creation of test sets. Initially, test sets should be generated based on specifications, while later on expanding the test set to include tests deemed necessary by studying the source code. The code coverage of a given test set can then be measured to get an idea as to the quality of the tests designed. It is recommended that tests be included in the test set that are capable of covering several requirements in the specification, since complex operations tend to be more effective at revealing faults in the code. The alternative and contrasting way of generating test sets, according to [Ham94], is to employ uniformly distributed random tests within functional classes.

Mutation coverage, see chapter 2.2.2, can be used as a technique for measuring the quality of test sets. The theory behind mutation coverage is that the causes of most faults are rather commonplace and uncomplicated. Hence, faults that none of the tests are able to uncover might get detected by the mutation technique, typically including faults caused by the use of unintended logical operators. By producing mutated versions of the software and test each of them against the current test set, one can determine which mutations of the program that were detected by the test set. Hence, we can observe that faults may remain in the code even after all tests have been run. It is not unusual for functional test sets to be incapable of detecting an amount of seeded faults, and the majority of people in charge of testing activities are aware of the difficulty in satisfying the mutation criteria. As a result, testers are tempted to create additional tests that are tailored to each mutation, thus disputing the recommended best practice for coverage testing. As [Ham94] underlines, if fault seeding represents genuine faults in a good way, the hit ratio can be looked upon as a quality indicator of testing; however, it is meaningless to improve coverage so that the seeded faults remaining can be found. After all, the 100% hit ratio will be achieved because of existing knowledge of seeded faults.



The quality of testing nowadays is based on the failure detection probability of the tests – their ability to uncover failures and their underlying faults in software. [Ham94] points out that for random testing based on an operational profile, the failure detection probability takes on the meaning of failure intensity – the reciprocal of Mean Time To Failure, MTTF. In other words, by using the system one will encounter failures that are likely to be found during actual system usage. This seems pretty inevitable, since MTTF is a widespread measure denoting the reliability of software systems. Coverage testing contributes to discovering a multitude of failures and hence boasts an impressive failure detection probability. This fact is down to code coverage directing focus to broader parts of the code and consequently covering parts of the source code neglected by alternative testing techniques. The formidable failure detection results from avoiding to sample the operational profile, and instead sampling according to classes that emphasize failures [Ham94]. The main objection to this strategy is that these classes carry no relation to the operational profile. As we all know, certain parts of a program are likely to be used more than others, and coverage testing may detect failures and subsequently remove faults in low-usage functionality and for that reason show relatively less interest in high-usage scenarios, compared to testing driven by an operational profile. A possible outcome is encountering problems in a production phase because of failures hiding in frequently used functionality of the software. With this in mind, [Ham94] concludes that coverage testing is, at best, no more significant than random testing. Coverage testing does not pay attention to the usage profile of a system and would hence lead to less intensive testing of its most important parts. As a result, software reliability is likely to get affected.

Based on the arguments and reasoning above, coverage testing could have a negative impact on reliability even if the test set attains high coverage scores. Reliability predictions may consequently be overestimated because of code coverage viewing all parts of the code as equally important. According to [Ham94], there is little theoretical basis and few experiments that would suggest there exists a relationship between code coverage and reliability, while at the same time underlining the sharp contrast between the difficulties of performing the experiments and their modest results. However, coverage testing is thought to uncover more failures than random testing, but that does not automatically imply that code coverage contributes to increased reliability.

Although critical of the code coverage – reliability relationship, [Ham94] sees code coverage possibly relating to a notion of trustworthiness as defined in chapter 4.1. Software quality ought to be measured based on results as opposed to time or effort spent on achieving that level of quality. Thus, hours spent during testing and failure detection probabilities are inappropriate candidate measures as far as software quality is concerned. The reason for this is that neither of these measures will provide an answer as to whether the software is trustworthy. As was mentioned in the opening section of this

chapter, failures occurring in low-usage functionality will affect reliability, but leave trustworthiness practically unchanged. In the words of [Ham94]: *“Catastrophic failures occur because no one has any conception of the situations that lead to them”*. It seems feasible to make measures of trustworthiness independent of any operational profile, thus suggesting that code coverage may have a stronger connection to the quality attribute of trustworthiness, rather than reliability.

#### 4.4.3.3 Inconclusive Results

Not all investigations and experiments performed yield conclusive or even suggestive results. An automated coverage tool named ATAC is employed in a series of tests to determine the correlation between code coverage and the number of faults found in *“A Coverage Analysis Tool for the Effectiveness of Software Testing”* by Michael Lyu, Joseph Horgan and Saul London [LHL94]. The stated hypothesis is that more faults will be discovered as code coverage increases. A number of groups of developers were formed, with each individual group responsible for creating a module facilitating automated landing of commercial air planes. The resulting modules were tested and code coverage, along with the number of detected faults, were measured. The findings were deemed inconclusive: *“We did not see strong correlations between the total faults detected in the program versions and their coverage measures during various testing conditions”* [LHL94]. However, the hypothesis is not completely rejected as each version was found to have a different fault distribution from the outset, thus rendering code coverage as an improper measure of the total number of faults residing in the code. Additionally, the statistical precision was believed to be reduced since the number of discovered faults in each version was relatively modest.

*"The only real mistake is the one from  
which we learn nothing."  
John Powell*

## CHAPTER 5:

# The Musa-Okumoto Model with Data Pre-processing

This chapter aims to give a thorough presentation of the reliability model of our choice in proposing a feasible approach to integrating reliability estimation into an automated tool for code coverage assessment. This software tool is an open-source product named JCoverage, and its operation is described in the subsequent chapter. Our suggested approach combines the use of a well-known reliability model with a technique that employs code coverage data to improve the estimates. To be more concrete, the logarithmic Poisson execution model by John D. Musa and Kazuhira Okumoto – from this point on referred to as the Musa-Okumoto model – is selected, along with the technique proposed by [CLW01] and presented in chapter 4. We will now consider each of them in turn, starting with the Musa-Okumoto model.

## 5.1 The Musa-Okumoto Model

The general elements of reliability estimation and the main features of reliability models were touched upon in chapter 3. As for the Musa-Okumoto model it consists of two components: An execution time component and a calendar time component. [MIO87] claims that the model has high predicative validity, achieved early during the system test phase. The required parameters need to be estimated prior to model usage since they do not relate to pre-execution characteristics of software and development environment. Thus, the task of predicting any parameters seems virtually impossible.

The Musa-Okumoto model employs a two-part approach in characterizing failure behavior. Initially, execution time is used in estimation, while later this notion of time is converted to calendar time to be more understandable to testers. This conversion takes place in the calendar time component, which characterizes how human and computer resources are utilized in the project. Execution time is said to be a preferable measure of time because of its superior ability to characterize *"the failure-inducing stress placed on software"* [MOI87].

### 5.1.1 Execution Time Component

The execution time component of the model is based on failures appearing as described by a non-homogeneous Poisson process. In chapter 3, non-homogeneous referred to a probability distribution changing with time. This is captured in a failure intensity function  $\lambda$  which, accordingly, changes with time. The failure intensity describes how the average number of failures experienced changes at different points in time. The Musa-Okumoto model defines the failure intensity function as:

$$\lambda(\mu) = \lambda_0 \exp(-\theta\mu)$$

In the equation above,  $\mu$  represents the cumulative number of failures experienced at a given time.  $\lambda_0$  denotes the initial failure intensity – the failure intensity at the beginning of execution, whereas  $\theta$  is a failure intensity decay parameter. The purpose of the latter parameter is to describe the relative change in failure intensity per failure experienced. When the reliability model is put to use the failure intensity is predicted to slowly decrease after a certain time. The reason for this, according to [MOI87], is that the parts of the code hiding faults even at a late stage of testing are unlikely to be exercised particularly often. Typically, these pieces of code have a low probability of being tested because of conditions – requiring atypical user input or an uncommon environment – that have to be satisfied in order for that code to be exercised.

One of the advantages of the Musa-Okumoto Model is its ability to tackle non-uniform operational profiles considerably better than some of the other reliability models available [MOI87]. A non-uniform operational profile results from dealing with a system where the operational profile fails to mirror actual system usage. The main reason as to why this particular model handles non-uniform operational profiles well, is said to be the Poisson probability distribution. This distribution appears to provide a better fit to actual failure detection than probability distributions found in a number of competing models. Another noteworthy feature of the Musa-Okumoto model is its ignorance of the quality of fault repairs, thus allowing new faults to be introduced in the code. The following function is used to denote mean failures experienced,  $\mu$ , versus execution time,  $\tau$ :

$$\mu(\tau) = \frac{1}{\theta} \ln(\lambda_0 \theta \tau + 1)$$

In this case  $\lambda_0$  represents the initial failure intensity as execution begins, while  $\theta$  is the failure intensity decay parameter. This function – mean failures experienced versus execution time that is – is infinite at infinite time. An expression also exists for determining failure intensity based on execution time, with the symbols carrying the same meaning as in previous definitions:

$$\lambda(\tau) = \frac{\lambda_0}{\lambda_0 \theta \tau + 1}$$

The Musa-Okumoto model allows us to compute Mean Time To Failure – MTTF – given that the failure intensity decay parameter  $\theta$  is less than 1:  $\theta < 1$ . If this condition is satisfied MTTF can be determined by means of the following expression:

$$\Theta(\tau) = \frac{\theta}{1-\theta} (\lambda_0 \theta \tau + 1)^{1-\frac{1}{\theta}} \quad (\theta < 1)$$

Since MTTF may not always be possible to determine, there exists an alternative expression for computing system reliability. In the function below,  $\tau'$  represents execution time measured from the present whereas  $\tau$  denotes the total execution time. As before,  $\theta$  is the failure intensity decay parameter while  $\lambda_0$  corresponds to the initial failure intensity.

$$R(\tau' | \tau) = \left[ \frac{\lambda_0 \theta \tau + 1}{\lambda_0 \theta (\tau' + \tau) + 1} \right]^{1/\theta}$$

Making use of the expressions presented thus far makes it possible to quantify software reliability of a given product, albeit in different ways. In addition, the Musa-Okumoto model proposes further mathematical functions that convey useful information for testing purposes. For instance, there are expressions which estimate the amount of time required to achieve a fixed reliability goal, or the number of detected failures required to reach that same target. We will not go into greater detail with respect to these expressions.

When a software program is put to use without any variables changing, reliability models are no longer considered to be non-homogeneous Poisson processes. Instead, models are reduced to homogeneous Poisson, with the number of failures in a fixed time interval corresponding to a Poisson distribution. The relationship between reliability,  $R$ , and the failure intensity,  $\lambda$ , can be expressed as a function of the execution time,  $\tau$ :

$$R(\tau) = \exp(-\lambda \tau)$$

Model characteristics include the use of the two parameters  $\lambda$  and  $\theta$ , symbolizing initial failure intensity and failure intensity decay respectively. As mentioned earlier in this section, these parameters must be estimated before the model can be used. There will always be a certain amount of uncertainty in connection with parameter estimation, and this uncertainty is set to transmit to the different estimated measures of reliability. Having said that, estimation is generally more accurate than what would be the case with

prediction, the latter not even facilitated by the Musa-Okumoto model [MOI87]. Estimation can be carried out with a statistical method based on observed failure times. The use of maximum likelihood estimation to determine which parameter values fit observed data the best, could serve as an example.

### 5.1.2 Calendar Time Component

The calendar time component relates execution time and calendar time at each point in time to a ratio between the two notions of time. During periods when the system is neither modified nor repaired, this ratio is viewed as a constant. The component bases itself on a debugging process with factors of limitation affecting its performance. Resources and the utilization of these are assumed to be constant during the period in which the model is used. The available resources and the resource needs of the process are assessed and possible bottlenecks are identified. A planned resource can be measured in quantities such as the size of the test team and the number of personnel assigned to fault removal tasks. The resource needs of the process are found and quantified as resources required per hour of execution time and/or per failure experienced. Hence, the ratio between the time units is determined by factors that impact and limit testing. [MOI87] lists the following candidate factors:

- ✓ Failure identification or test crew
- ✓ Failure correction or debugging personnel
- ✓ Available computer time
- ✓ Other limited resources

Out of the four factors stated above, the first three control the rate of testing, with failure correction personnel typically having the most significant effect on calendar time prediction. For projects where failure identification and failure correction is performed by the same individuals, these two tasks can be merged. The resource consumption,  $\chi_r$ , for each individual resource in the process can be computed as follows:

$$\chi_r = \theta_r \tau + \mu_r \mu$$

In this formula the resource consumption is a function of both the number of failures detected and the amount of CPU time used. The variable  $\theta_r$  refers to the resource consumption per CPU hour, while  $\mu_r$  denotes the resource consumption per failure – both of which require necessary adaptation and tailoring to each project and resource.

The ratio between execution time and calendar time can be found by using the expression below:

$$\frac{dt_r}{d\tau} = \frac{1}{P_r \rho_r} \left( \theta_r + \mu_r \frac{\lambda_0}{\lambda_0 \theta \tau + 1} \right)$$

$P_r$  refers to resource quantity and  $\rho_r$  denotes resource utilization.  $\theta_r$  is assigned resource consumption per CPU hour, whereas  $\mu_r$  represents the resource consumption per failure. These measures are explained in the following section. The remaining variables – execution time, failure intensity decay parameter and initial failure intensity – are the same as for previous expressions stated in this chapter.

### 5.1.2.1 Description of Resource Measures

Resource quantity is a measure of the amount of resources available. As far as computer time is concerned, this measure is in the form of "prescribed work periods" [MIO87]. The same source proposes an example for the purpose of comprehension: If there are 80 computer hours available per week and the prescribed working week consists of 40 hours, this will yield a resource quantity for available computer time of 2. For other resources it is a matter of the number of people, and the fact that some may not work full days is not taken into account since resource quantity is a measure of available resources as opposed to employed resources.

Resource utilization is a measure of the amount of available resources that are put to use. "Resource utilizations are generally estimated from formula or practical experience" [MOI87]. Maximum values must be found for the period in which the current resource is the limiting factor. The two resource usage parameters –  $\theta_r$  and  $\mu_r$  – represent average values of resource expenditure and are affected by factors such as application domain and the level of experience among software developers. By collecting data on resource usage along with relevant data obtained during testing, the following model can be adapted to the data observed:

$$\frac{\Delta\chi_r}{\Delta\tau} = \theta_r + \mu_r \frac{\Delta m(\tau)}{\Delta\tau}$$

Additionally, overhead factors such as holidays, vacations, absence, courses and administrative duties have to be taken into consideration. By now all parameters necessary for making the move from execution time to calendar time ought to be presented.

## 5.2 Code Coverage Pre-process Model

In chapter 4 we introduced a technique developed by Mei-Hwa Chen, Michael R. Lyu and Eric Wong and described in detail in [CLW01], which reduces the relative weight of tests in reliability estimation that do not increase code coverage nor detect additional failures. In this section we will try to adopt a more technical perspective with regards to this technique than what was the case in chapter 4.

### 5.2.1 Model Rationale

The technique, presented as two different versions in [CLW96] and [CLW01] respectively, is based on reliability models using the time between failures as a measure of reliability. The line of action is to measure cumulative code coverage and number of failures after execution of each test case to assess whether it qualifies as an effective testing effort. If a test case turns out to be non-effective in this sense, a function is used to reduce the execution time employed by that particular test case. This consequently leads to a reduction in the reliability estimate.

The heart of both versions is the compression ratio  $\rho_i$  which is computed for all tests considered to be non-effective. This is a value which is multiplied by the execution time of the non-effective test case, thus resulting in a time reduction before data is relayed to the reliability model. The compression ratio is computed slightly differently in the two versions, as will be explained below; however, they both make use of changes in execution time and cumulative code coverage during calculation and are hence comparable. As is to be expected, test cases deemed effective will yield a compression ratio of 1 and consequently leave execution time unchanged.

### 5.2.2 Compression Ratio through Smoothing Parameters

The initial version of the technique presented in [CLW96], proposes two parameters  $\alpha$  and  $\beta$  in computing the all-important compression ratio. These parameters, referred to as smoothing parameters, are dependent on the software itself and the reliability model of choice, for instance Goel-Okumoto or Musa-Okumoto. Unfortunately, information was not obtainable on how a given reliability model would impact the parameters. The equation below is used to compute the compression ratio by employing the smoothing parameters:

$$\rho_i = \frac{\alpha \delta t_i^2 \cdot \beta \delta c_i^2 + \alpha \delta t_i^2}{\alpha \delta t_i^2 + \beta \delta c_i^2 + (\alpha \delta t_i^2 \cdot \beta \delta c_i^2)}$$

In the above expression  $t_i$  and  $c_i$  refers to execution time and cumulative code coverage respectively, upon termination of test case  $i$ . Subscript is used to indicate which test case is referred to. The  $\alpha$  and  $\beta$  parameters are the smoothing parameters mentioned earlier, while  $\delta t_i$  and  $\delta c_i$  relate to the change in execution time and cumulative code coverage before and after execution of the  $i$ th test case.



### 5.2.3 Compression Ratio through Scaled Parameters

The second version of the technique, elaborated in [CLW01], is based merely on changes in code coverage and the number of failures discovered. In practice the parameters  $c$  and  $t$  are scaled so that their respective numerical values are virtually equal. The claimed intention is to stabilize the computation of the compression ratio  $\rho$  and can be done at various points in time. According to [CLW01], code coverage measures are generally scaled so as to match the average execution time of test cases. The scaled values of  $c$  and  $t$  are then used in computing  $\rho$ . In the estimation itself the original values of  $t$  are used, but these are adjusted by employment of the estimated compression ratio. "Scaling of the execution time does not affect the reliability estimation process" [CLW01]. The expression for calculating the compression ratio in this version is as follows:

$$\rho_i = \frac{\delta t_{i-1}^2 + \delta t_{i-1}^2 \cdot \delta c_{i-1}^2}{\delta t_{i-1}^2 + \delta c_{i-1}^2 + (\delta t_{i-1}^2 \cdot \delta c_{i-1}^2)}$$

In correspondence with the compression ratio equation of the initial version,  $t_i$  and  $c_i$  denotes execution time and cumulative code coverage respectively, upon termination of test case  $i$ . Once again, subscript is used to indicate the test case of interest.  $\delta t_i$  and  $\delta c_i$  refers to the change in execution time and cumulative code coverage before and after execution of the  $i$ th test case.

For our approach we have chosen the latter of the two versions as the pre-process technique to be used in conjunction with the Musa-Okumoto software reliability model.

*"Vision is the art of seeing the invisible."*

Jonathan Swift

## CHAPTER 6:

# A Software Implementation of Automated Code Coverage Analysis: JCoverage

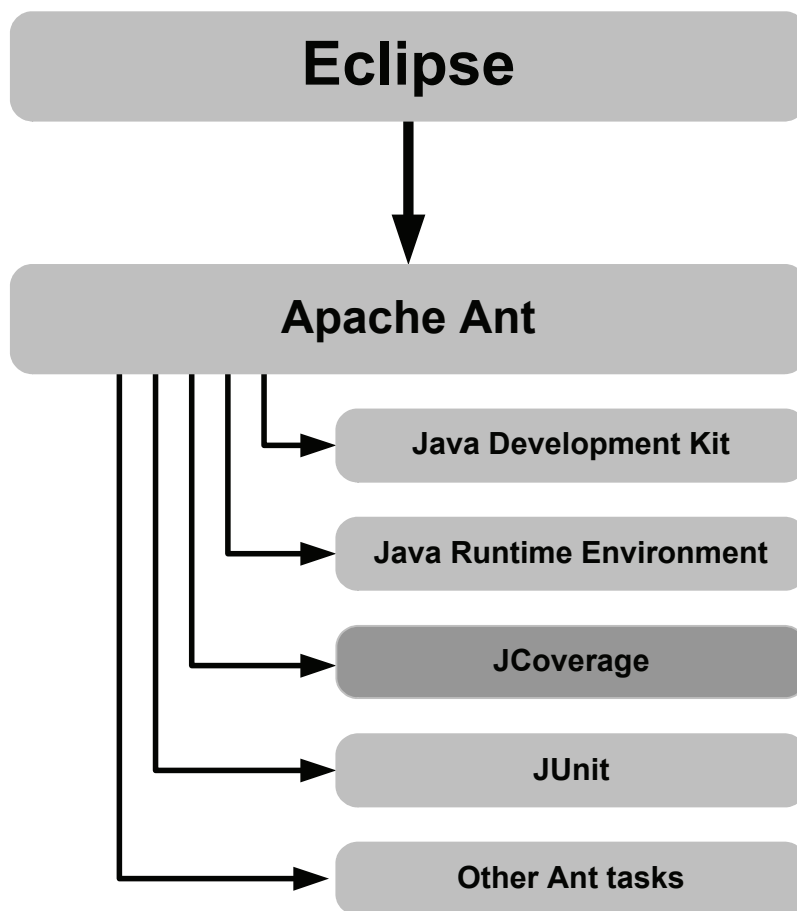
This chapter is mainly dedicated to JCoverage – a software tool for automated code coverage analysis – and its operation and implementation of code coverage principles. We start off by describing the context and environment of JCoverage before diving into its operation and means of computing code coverage measures. Having covered the main functionality and scope of JCoverage we move on to pinpoint possible improvements for future versions. Finally, we round off this chapter by considering the possibility and uncovering the main challenges of integrating the Musa-Okumoto model and the code coverage pre-processing technique presented in the previous chapter, to yield a reliability measure.

## 6.1 Characteristics and Environment

In order to compute code coverage for a system, a tool is needed for registering which parts of the code are being executed. JCoverage facilitates code coverage measurements for programs developed in Java, and appears in three different versions with associated licenses. The version that we have been examining has a GNU Public License and has not undergone improvements found in the other two versions. One of the versions is said to scale better, as well as offering support for Java Remote Method Invocation. JCoverage is, however, not the only available solution on the market for measuring code coverage. The main reason why we ended up focusing on JCoverage was its open-source nature, thus providing us with access to the source code. This characteristic, in turn, made it possible to adopt a more detailed approach in studying how various code coverage measures were implemented.

When putting JCoverage to use it might be practical to employ Apache Ant and Eclipse simultaneously. Eclipse is an IDE – Integrated Development Environment – developed in Java and used to develop other Java applications. One of its features is the capability to make XML-based build files which specify how the Java application should be compiled. This file can then be interpreted by Apache Ant, which is a build tool for automating tasks that are performed several times during development. Typical Ant usage includes code compilation and related tasks. The tool incorporates a number of possible “tasks”, known as small programs intended to solve simple problems. These

tasks can be combined to constitute macros in the form of XML files which may then be read and executed by Ant. Examples of tasks include the generation of Java Archive files, the creation of folders or directories and deletion of files. Ant distinguishes itself from other build tools in the sense that it is a Java-developed product that can be executed on a number of operating systems. With JCoverage being a utility for testing Java code, possibly developed on a variety of platforms, it seems only natural that JCoverage can be integrated into the development process in the form of an Ant task. The environment of JCoverage and its interaction with Ant and Eclipse is modeled in figure 6.1 below.



*Figure 6.1*  
*The respective roles of*  
*JCoverage, Ant and Eclipse*

Technically speaking, coverage data are extracted from a Java program during testing in one of three ways. These implementation approaches were presented in chapter 2.5 but are briefly reiterated here for the sake of convenience:

- ✓ Source code instrumentation
- ✓ Byte code instrumentation
- ✓ Execution through a modified Java Virtual Machine

By opting for instrumentation of the source code, additional lines of code are added prior to byte code compilation. In the case of byte code instrumentation one is not dependent on having source code available, whereas the third and final alternative requires modifications to be made to JVM so that information is logged during program execution.

## 6.2 JCoverage Operation

Having introduced the main concepts of JCoverage it is time to look at how it goes about to convey coverage information to users. We find it feasible to divide its operation into two different phases. The first of these deals with the process of instrumentation, while the second deals with the computation of coverage measures and the ensuing presentation of coverage information of interest.

### 6.2.1 Instrumentation

When code coverage is to be measured the first thing JCoverage does is to properly instrument the code to be tested. This implies inserting additional code that will enable information to be logged to a file, which will later provide users with information as to what parts of the code have been exercised. JCoverage performs code instrumentation by modifying the byte code, or more specifically the class files. Technically speaking, code instrumentation is started once a separate instrument task is called in Ant which initiates the entire instrumentation process.

```
<instrument todir="instr/">
  <fileset dir="build/">
    <include name="**/*.class"/>
  </fileset>
</instrument>
```

Extracted parts of an XML file is stated above where Ant is asked to instrument all class files residing in the `build` directory and all its sub-directories. The build directory is placed in a project directory, while all files successfully instrumented are stored in a directory named `instr`, also within the project directory.

When JCoverage is set to instrument a project, all class files of the current project are examined. Class files that have already been instrumented or merely contain an interface with no executable code are exempted from instrumentation. This is solved by attaching an empty interface named `HasBeenInstrumented` to each class file previously instrumented. Class files may thus be checked to see if they implement the interface mentioned above prior to deciding whether a particular class will be instrumented or not.

The source code, which in turn is made into instructions interpreted by the Java Virtual Machine, is found in the methods of each class. Consequently, all methods belonging to a class are examined when instrumenting that particular class. Certain lines of code do not contribute with instructions executed in the JVM, for instance lines of comments, blank lines for improved visibility and “} else {”. The else case referred to affects the syntax or sequence of instructions as they appear in the program, but does not explicitly add instructions itself. For this reason JCoverage neglects the execution of these lines when computing coverage measures. It also provides users with the option of defining elements to be ignored, but we consider this particular functionality to be of no great concern at this point.

One line of source code can be represented as one or more elementary instructions in the resulting byte code. In order to be able to decide whether a line of code has been exercised, JCoverage inserts code ahead of the first instruction representing that specific line. As a matter of fact, JCoverage adds code before each line that has byte code instructions associated with it. An example of code that instruments byte code is stated below.

```
/**
 * The core instrumentation. This sequence of instructions is
 * emitted into the instrumented class on every line of
 * original Java code.
 *
 * NOTE THAT THIS EMITTED CODE IS ALSO LICENSED UNDER THE GNU
 * GENERAL PUBLIC LICENSE. NON GPL INSTRUMENTED APPLICATIONS
 * MUST BE LICENSED UNDER SEPARATE AGREEMENT. FOR FURTHER
 * DETAILS, PLEASE VISIT http://jcoverage.com/license.html.
 */
```

```
InstructionList emitGetInstrumentationAndTouchLine(
    LineNumberGen lng) {

    InstructionList il=new InstructionList();

/**
 * Obtain an instance of InstrumentationFactory, via a static
 * call to InstrumentationFactory.
 */

    il.append(classGenHelper.createInvokeStatic(
        InstrumentationFactory.class,
        "getInstance", InstrumentationFactory.class));
```

In the case above, JCoverage adds code for the purpose of accessing an instance of the `InstrumentationFactory` class. This class loads existing test data from a file and saves new data upon terminating program execution.

```
/**
 * Create a new instance of Instrumentation (or reuse an
 * existing instance, if one is already present in the
 * factory), for the class that we have instrumented.
 */
```

```

il.append(new LDC(←
    classGenHelper.getConstantPool().addString(←
    classGenHelper.getClassGen().getClassName()));

il.append(classGenHelper.createInvokeVirtual(←
    InstrumentationFactory.class, "newInstrumentation", ←
    Instrumentation.class, String.class));

```

Moving on, JCoverage adds a `String` object to the stack with the name of the class being instrumented. A method of the `InstrumentationFactory` class is called next that returns an already existing instance or alternatively creates and returns a new instance of the `Instrumentation` class. The method invoked requires a `String` object as an input parameter, and the `String` object currently residing on the stack will now become the `String` argument which the method receives.

```

/**
 * Update the coverage counters for this line of source code,
 * by "touching" its instrumentation.
 */

il.append(InstructionListHelper.push(←
    classGenHelper.getConstantPool(), lng.getSourceLine()));

```

In the code above, an integer, corresponding to the line number of the source code that led to the generated instructions, is pushed onto the stack. This integer comes to use in the following code:

```

il.append(classGenHelper.createInvokeInterface(←
    Instrumentation.class, "touch", void.class, int.class));

return il;
}

```

The integer that was pushed onto the stack in the previous code sequence is this time used as an argument when calling a method of the `Instrumentation` class. This method increments a counter used to indicate the number of times that this specific line has been exercised during testing. The line number is used as an index in a data structure for keeping track of the number of executions for each line contained in the class.

In order for JCoverage to be able to compute branch coverage, all instructions are studied. In the case of `if` instructions the line number where the `if` construct occurred is registered along with the line number of the first line of code immediately succeeding the `if` clause belonging to the `if` construct. All data that are generated through the execution of an instrumented Java

application are stored as serialized Java objects in a file named `jcovrage.ser`. In the case of successive executions the data are aggregated.

## 6.2.2 Code Coverage Presentation and Computation

Instrumenting the code is necessary to provide access to and information about which parts of the code are being executed. All data must further be processed to enable computation and presentation of coverage measures of interest. JCoverage offers the possibility of generating coverage reports in both HTML and XML format to present line coverage and branch coverage scores on different levels.

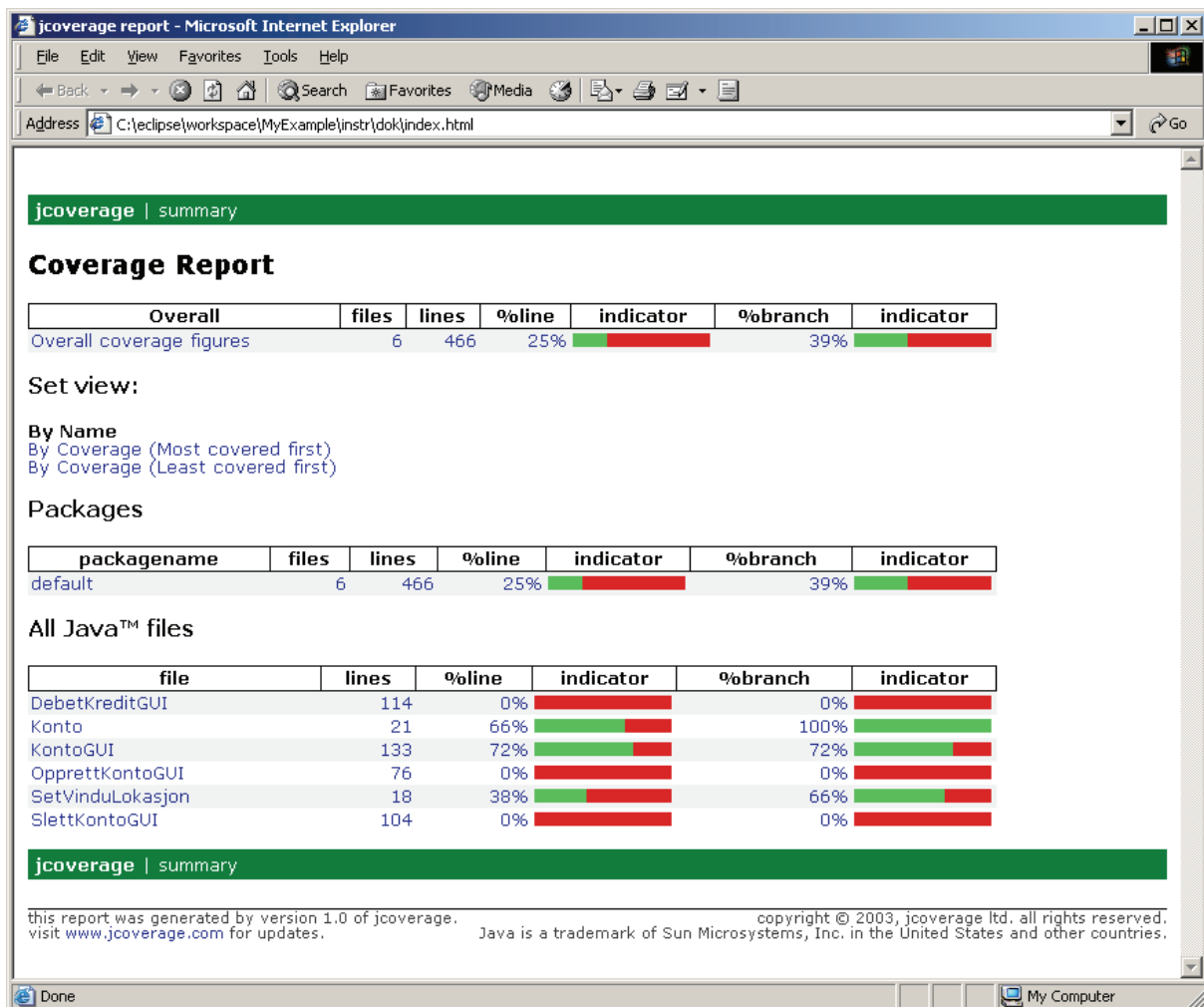
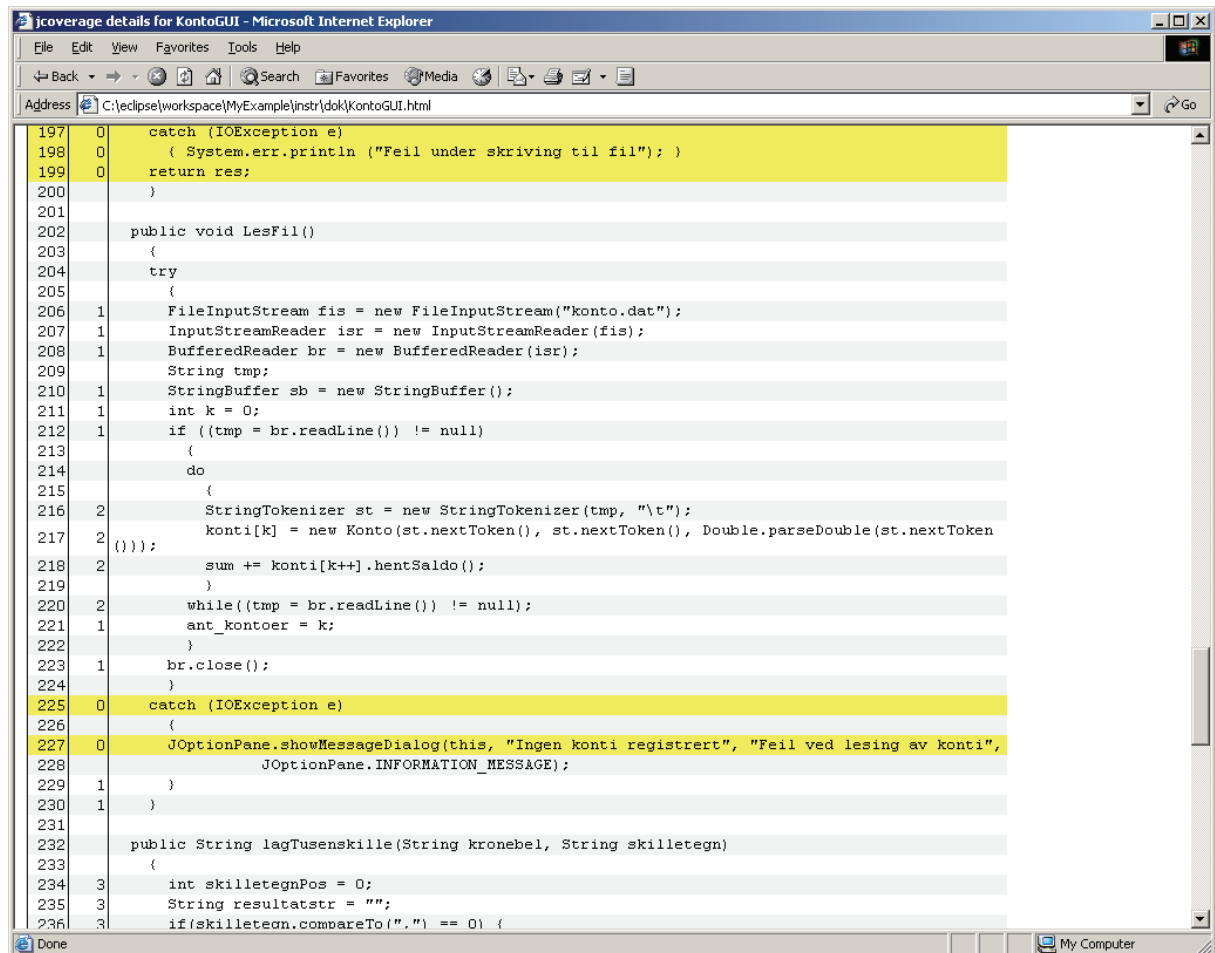


Figure 6.2  
 HTML page for presenting coverage  
 measures for all packages and classes being tested

### 6.2.2.1 Coverage Reports

The generation of reports can be done by employing an Ant task. This will lead to a procedure generating reports based on data stored in the aforementioned file `jcoverage.ser`. In the case of an HTML representation of the report, a series of HTML files are created or alternatively modified if a report already exists. Code coverage measures are presented for the individual classes or Java files, the packages, and in total for all code under test, as illustrated in figure 6.2 above. The report lets users navigate through existing classes and packages. Each class has a dedicated HTML page where the source code is presented. The parts of the code not exercised by the test set are highlighted as shown in figure 6.3 below.



*Figure 6.3*  
HTML page generated by JCoverage for presenting the source code of a tested class – lines of code left unexercised are highlighted

JCoverage can also produce an XML version of the coverage report. This consists of a simple XML file which lists all registered data, as well as computed line coverage and branch coverage for each class and its associated



methods. This file is useful when exchanging coverage data with other applications or when designing a separate and tailored presentation. Having said that, it turns out that the XML file generated by JCoverage does not satisfy the XML standard, thus running the risk of encountering an error message when employed. The coverage report is shown in XML format in figure 6.4.

```

<?xml version="1.0" ?>
- <coverage src="C:\eclipse\workspace\MyExample">
- <class name="KontoGUI$4">
  <file name="/KontoGUI.java" />
  <line rate="0.5" />
  <branch rate="1.0" />
  - <methods>
    - <method nameAndSignature="init(LKontoGUI;)V">
      <line rate="1.0" />
      <branch rate="1.0" />
    </method>
    - <method nameAndSignature="actionPerformed(Ljava/awt/event/ActionEvent;)V">
      <line rate="0.0" />
      <branch rate="1.0" />
    </method>
  </methods>
  <valid lines="81, 82" />
  <line number="81" hits="1" />
  <line number="82" hits="0" />
</class>
- <class name="OpprettKontoGUI$1">
  <file name="/OpprettKontoGUI.java" />
  <line rate="0.0" />
  <branch rate="1.0" />
  - <methods>
    - <method nameAndSignature="init(LOpprettKontoGUI;)V">
      <line rate="0.0" />
      <branch rate="1.0" />
    </method>
    - <method nameAndSignature="actionPerformed(Ljava/awt/event/ActionEvent;)V">
      <line rate="0.0" />
      <branch rate="1.0" />
    </method>
  </methods>
  <valid lines="49, 50" />
  <line number="49" hits="0" />
  <line number="50" hits="0" />
</class>

```

Figure 6.4  
XML version of the coverage report

### 6.2.2.2 Computation of Coverage Measures

JCoverage restricts itself to the computation of two coverage measures, namely line coverage and branch coverage. Line coverage denotes the percentage of lines of code executed with respect to the total number of lines and can be formulated as follows, with  $C_l$  representing line coverage,  $n_c$  the number of lines covered and  $n$  the total number of lines:

$$C_l = \frac{n_c}{n} \cdot 100\%$$

Thus, to calculate the line coverage of a given method, the total number of code lines must be assessed along with how many of these lines have been executed at least once. The same procedure is employed, irrespective of whether the line coverage is computed for a class, a package or for the entire application under test. Figure 6.5 reports a line coverage of 69% for a simple example program.

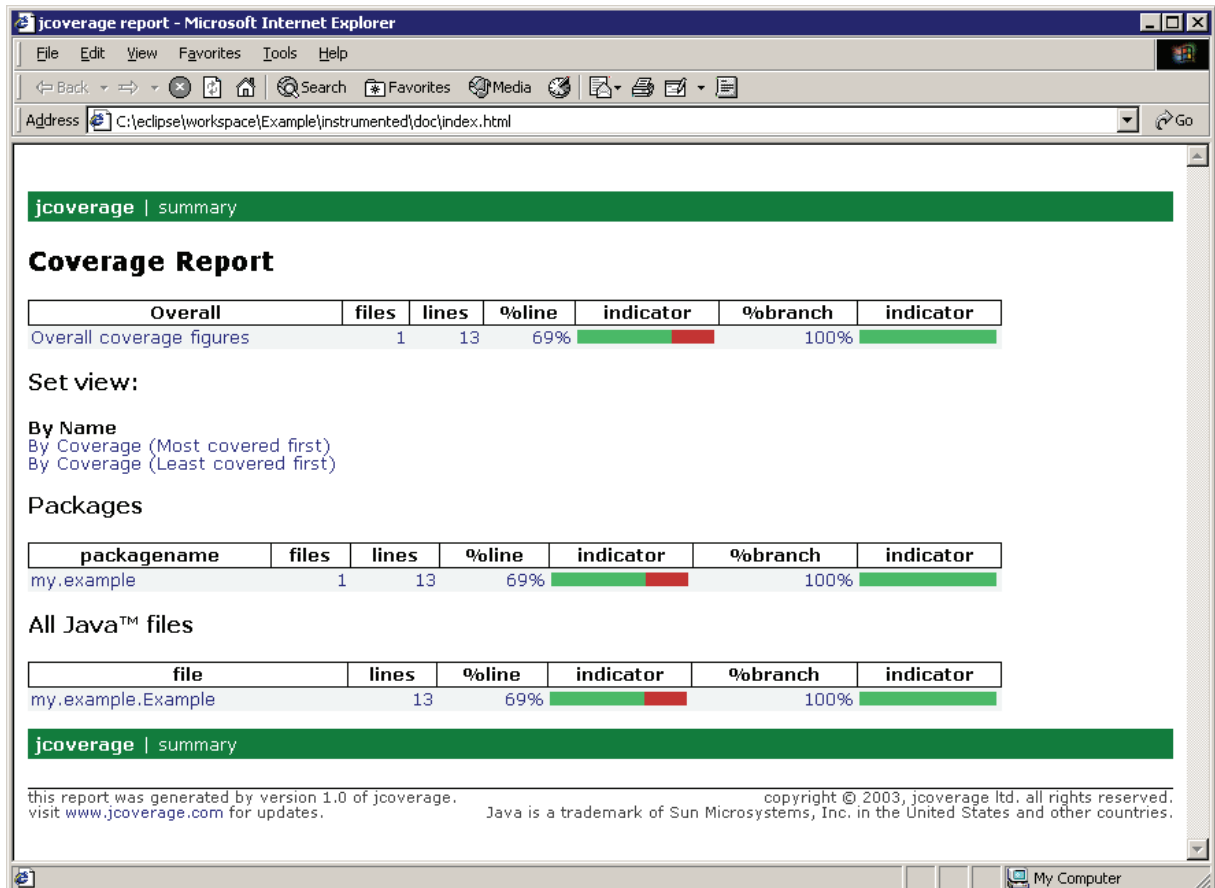
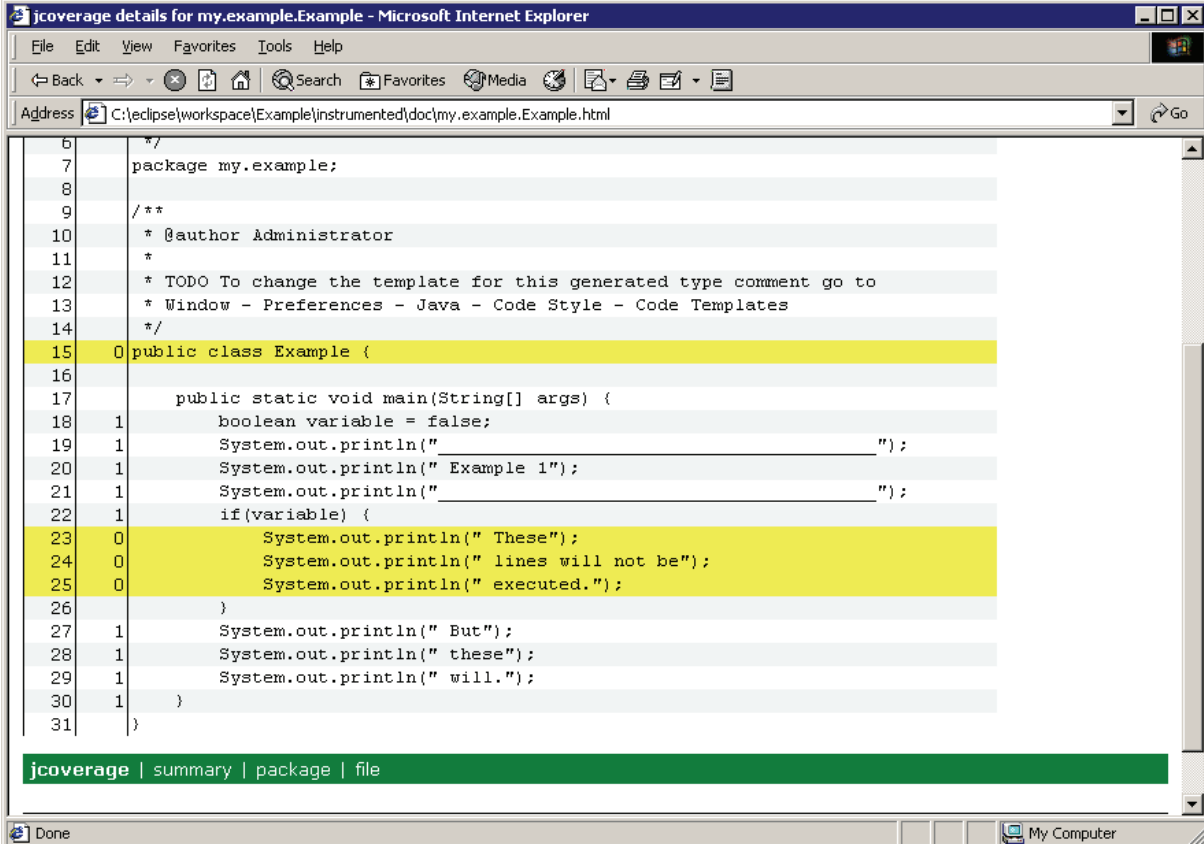


Figure 6.5  
Coverage report for testing of a sample program

For the purpose of computing branch coverage, JCoverage makes use of the data structure created during the instrumentation phase, where the line number of the `if` construct was stored, along with the line number succeeding the block of code that, if the expression evaluates to `true`, is executed. This pair of lines is then used when calculating the branch coverage. The tool examines which of the lines that have been visited during testing, and the relationship between the number of visited lines and the total number of lines in the data structure is finally returned as the branch coverage. Hence, the computation of branch coverage follows the same basic procedure as the one used for line coverage, but with a more restricted set of code lines as input.

Having evaluated and studied JCoverage's implementation of branch coverage, we find that it does not seem to correspond to our perception of how code coverage literature defines the measure. The current implementation does not check whether the code inside an `if` construct is executed if the expression evaluates to `true`. As mentioned earlier, JCoverage merely verifies that the line where the `if` construct appears is exercised, along with the first line succeeding the `if` clause. As a consequence branch coverage can be reported at 100% even when one or more `if` statements have never evaluated to `true`. This situation is illustrated below by means of a constructed and straightforward example.



```

6      */
7      package my.example;
8
9      /**
10     * @author Administrator
11     *
12     * TODO To change the template for this generated type comment go to
13     * Window - Preferences - Java - Code Style - Code Templates
14     */
15     0 public class Example {
16
17         public static void main(String[] args) {
18             1     boolean variable = false;
19             1     System.out.println("_____");
20             1     System.out.println(" Example 1");
21             1     System.out.println("_____");
22             1     if(variable) {
23                 0     System.out.println(" These");
24                 0     System.out.println(" lines will not be");
25                 0     System.out.println(" executed.");
26             }
27             1     System.out.println(" But");
28             1     System.out.println(" these");
29             1     System.out.println(" will.");
30         }
31     }

```

At the bottom of the browser window, there is a navigation bar with links: [jcoverage](#) | [summary](#) | [package](#) | [file](#)

Figure 6.6

*An example highlighting the branch coverage dilemma*

Figure 6.5 on the previous page displayed a coverage report for the sample program used in figure 6.6 right above. Line coverage was computed to be 69% with branch coverage reaching 100%. As before the yellow lines represent lines of code that have not been exercised by the tests performed. Figure 6.6 reveals that the entire block of code inside the `if` statement has not been tested, since the boolean variable used to evaluate the expression is set to `false`. It is possible to attain complete branch coverage without having achieved complete line coverage, exemplified by line 15 which is not yet executed. However, it should not be possible, according to our interpretation

of code coverage literature and the branch coverage measure in particular, to achieve complete branch coverage without having tested all existing `if` clauses.

Even if JCoverage had been modified to test whether the `if` clause was executed and hence if the expression evaluates to `true`, we still question if this would be sufficient to qualify as branch coverage. According to our understanding of branch coverage defined in chapter 2.2.1.2, the measure of branch/decision coverage is based on whether conditions evaluate to both `true` and `false`. This requires all `if` statements to be combined with matching `else` clauses, as shown below.

```
11: ..
12: if(isAscii) {
13:     in = new AsciiReader();
14: }
15: else {
16:     in = new UnicodeReader();
17: }
18: return in;
19: ..
```

The piece of code above serves as an example to illustrate how all `if` statements must be properly formatted in order to detect that an expression evaluates to `false`. In this specific case, line 15 will be marked as the first line succeeding the `if` construct. This line is then examined to find out whether it has been executed when computing the branch coverage. If, however, the code stated below is tested, then 100% branch coverage could be achieved even if the `if` expression has never evaluated to `false`.

```
87: ..
88: if(customerNum < 100) {
89:     price = price * 0.9;
90:     customerNum++;
91: }
92: getCreditCredentials();
93: ..
```

In this case line 92 will be verified when computing the branch coverage, since this is the line immediately following the `if` clause. Line 92 will be executed irrespective of whether the `if` expression evaluates to `true` or `false`; it is not part of an `else` clause. Since JCoverage ignores whether the logical expression of an `if` statement returns both `true` and `false`, its notion of branch coverage appears to resemble a measure known as basic block coverage. Basic block coverage, described in chapter 2.2.1.1, is meant to

disregard or overlook the size of the different blocks and rather give a picture of how well the blocks of code are tested. During testing only the largest blocks might be tested, thus resulting in a coverage measure such as statement coverage approaching 100%, whereas basic block coverage would detect that smaller blocks of code have been ignored. However, we have to make a reservation that the naming conventions of the various code coverage measures may differ and that a considerable part of this could be down to name confusions. This may always be a factor when a defined, agreed-upon standard or framework is non-existent.

In addition to questioning the notion of branch coverage employed by this tool, we find it necessary to comment upon a few aspects of the line coverage measure. The code line below raises one point of concern:

```
44: ..
45: if(debug) Logger.println("Input file not available.");
46: ..
```

When the entire `if` statement is restricted to a single line of code even line coverage may reach 100% although not all parts of the code have been executed. A similar situation occurs when the `if` statement is integrated into “different” code on the same line, as exemplified by line 79 below.

```
77: ..
78: out = "Loading "
79: out += numFiles + " file" + (numFiles < 1 ? "s" : "");
80: ..
```

Thus, JCoverage does not necessarily paint a correct picture of the situation it measures. On the one hand, lines can be said to have been executed since certain instructions belonging to them have been exercised and hence, line coverage reports these lines as covered. On the other hand, we believe that all instructions on a line of code have to be executed in order for sufficient line coverage to be reported. If the latter is used as a requirement we know of an additional case where parts of a line are left untested, namely the evaluation of boolean expressions with short-circuiting support, as explained in chapter 2.2.1.2. This is mainly an issue when dealing with measures such as condition coverage and multiple condition coverage. Since JCoverage implements neither of these measures we find it infeasible to take the evaluation of boolean expressions into account when assessing whether or not the code is covered.

## 6.3 Suggested Improvements

As can be seen from the previous sections of this chapter, JCoverage presents a few challenges, most of which center around the chosen implementation of code coverage measures. In this section we try to explain the efforts needed in order to establish a correspondence between the coverage measure definitions of this report and the tool implementation of the same measures.

For JCoverage to be able to determine whether an `if` expression evaluates to `true`, regardless of whether the code to be executed in the case of `true` is a block of code, a single line of code or merely a part of one, the code requires additional instrumentation. As of now, instrumentation code is only added at the start of each line, whereas the scenario mentioned above would involve the insertion of instrumentation code ahead of the first instruction to be executed given an evaluation of `true`. This raises yet another question: How should the added lines of instrumentation code count in the computation of line coverage and branch coverage respectively? If they count in the same way as the original instrumentations did, certain lines might be counted twice and impact line coverage accordingly. It hence seems inevitable that data collected from this added instrumentation should only be used in computing the branch coverage. Alternatively, the instrumentation for line coverage and the instrumentations within `if` clauses can be combined so that a particular line is not reported as executed until both instrumentations have been called by JCoverage with their respective parameters.

In order for JCoverage to test whether `if` expressions evaluate to `false` it can instrument `if` statements as illustrated through the source code excerpt below:

```
77: ..
78: if(originalExpression || instr.touchIf(78)) {
79: ..
```

For this particular case `|| instr.touchIf(78)` represents the necessary instrumentation. This expression will be a method call that returns the value `false`, under any circumstances, with the intention of not interfering with the original evaluation of the `if` statement, while at the same time indicating that evaluation has taken place. A call to the method is assumed to be made if `originalExpression` evaluates to `false` only, since the Java excerpt above short-circuits.

## 6.4 An Approach to Implementing Reliability Estimation in JCoverage

During the course of this project we have, among other things, studied different techniques for estimating software reliability through the use of code coverage information. A data pre-processing technique and the Musa-Okumoto model were described in chapter 5 as a combined means of utilizing code coverage data to come up with realistic and useful reliability estimates. In this section we consider the feasibility of integrating the aforementioned approach into JCoverage as an added reliability module.

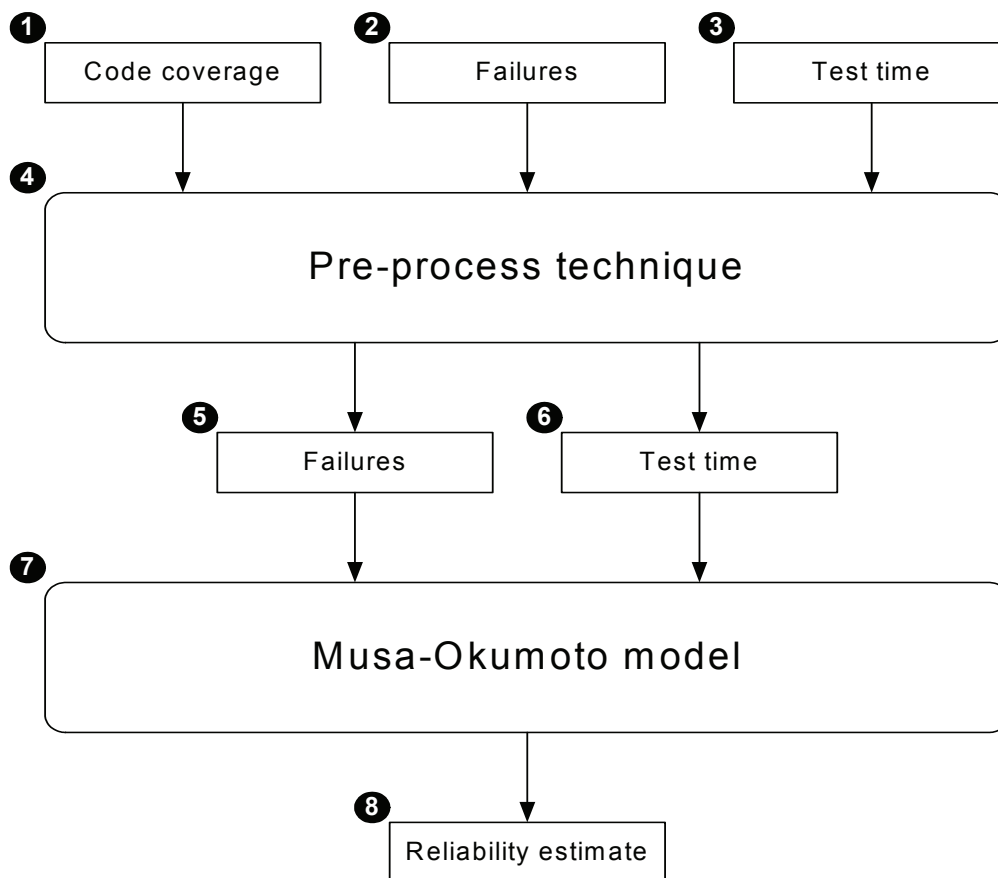


Figure 6.7

*Illustration of the proposed reliability estimation approach*

What makes this an attractive strategy, in theory, is that an automated tool for tracking code coverage provides coverage information that could be directly employed by the pre-processing technique described in chapters 4 and 5, before relaying processed data to the Musa-Okumoto model for estimation. Implementing such an approach in JCoverage is, however, far from straightforward. We emphasize the fact that the notion of automatically generating reliability estimates through tool usage is a highly relative one. Since reliability might be affected by external factors related to phases of system development as well as the application domain, we are fully aware of

the fact that reliability estimation requires substantial human intuition, consideration and effort. Instead, the approach is intended to simplify the more administrative parts of the effort, particularly with regards to the pre-process technique.

### 6.4.1 The Road to Obtaining Reliability Estimates

Throughout this section we will refer to figure 6.7 which depicts the proposed course of generating a reliability estimate. The numbers from 1 to 8 in the subsequent text refer to corresponding elements in the figure above.

#### 6.4.1.1 Code Coverage (1)

JCoverage can be used to assess the achieved code coverage once a test case has finished execution. An experiment is referred to in [CLW01] where block coverage is used with the pre-process technique. However, the authors have

Line	Hits	Source
1		package my.example;
2		
3		/**
4		* @author Administrator
5		* Example2.java
6		*/
7		public class Member {
8		
9		private String name;
10		private String phone;
11		private String email;
12		private String idNum;
13		
14	1	public Member(String name,String phone,String email,String idNum) {
15	1	this.name = name;
16	1	this.phone = phone;
17	1	this.email = email;
18	1	this.idNum = idNum;
19	1	}
20		
21		public void setName(String name) {
22	1	this.name = name;
23	1	}
24		
25		public void setPhone(String phone) {
26	1	this.phone = phone;
27	1	}
28		
29		public String getPhone() {
30	1	return phone;
31		}
32		
33		public void setEmail(String email) {
34	0	this.email = email;
35	0	}
36		

Figure 6.8  
The cumulative frequency of test executions per line of code



deferred empirical experiments with other coverage measures such as branch coverage, thus ending up on a list of future work. The Musa-Okumoto model does not make use of code coverage in its estimation of reliability, but an assumption which is made, unravels new challenges as to how code coverage is measured. The Musa-Okumoto model assumes, as do several other reliability models, that faults are removed instantly – that is, immediately following detection. In the words of [MIO87]: *“In actuality, there is always some delay, but the effects of the delay are not serious and can easily be accounted for”*.

It is virtually impossible to completely avoid minor modifications to the program code during testing. This poses a number of challenges in JCoverage, with the main issue being the way that JCoverage registers code coverage. Each line is instrumented prior to test execution, and each time that a line is exercised its line number is reported by the tool. A problem occurs when wanting to reuse this information after the code has been modified. Figure 6.8 on the preceding page helps to illustrate the problem at hand, displaying the number of times each line of code in a class has been executed. Let us assume that a fault is detected in the current class, hence requiring added code or modifications of the existing code before testing and JCoverage operation may resume. Now the previously registered data will most likely end up with incorrect coverage values.

Line Number	Execution Count	Code Snippet
16	3	this.phone = phone;
17	3	this.email = email;
18	3	this.idNum = idNum;
19	3	}
20		
21		public void setName(String name) {
22	3	this.name = name;
23	3	}
24		
25		public String getName() {
26	2	return name;
27	2	}
28		
29		public void setPhone(String phone) {
30	3	this.phone = phone;
31	1	}
32		
33		public String getPhone() {
34	1	return phone;
35	0	}
36		
37		public void setEmail(String email) {
38	6	this.email = email;
39	0	}
40		

Figure 6.9  
Consequences of adding or modifying code

Figure 6.9 above is similar to figure 6.8, except for the introduction of a new method `getName()` on lines 25 through 27. Since this method replaces pre-existing code it will not be visible to JCoverage that the new method has never been executed. The code previously occupying those same lines, the method `setPhone()` in this particular instance, may have been exercised during earlier phases of testing, thus causing the lines to be marked as executed. Deleting all information of which lines have been tested would require a re-run of all previous test cases – an impractical outcome both with respect to time and resources. A possible solution could be to modify JCoverage so that it would be capable of dealing with changes in the code under test. We believe this can be done by registering changes in the code, such as the displacement of line numbers, the code being removed and the code being added. These pieces of information may then be used to update previously registered data so as to mirror the actual code. Alternatively, JCoverage would have to be modified to accommodate a different way of identifying which lines have been executed.

#### 6.4.1.2 Failures (2)

In order to decide whether a test case should be deemed effective or not, the pre-process technique utilizes knowledge of whether that particular test case detected one or more failures. Additionally, knowledge of the total number of failures is used by the Musa-Okumoto model. It is therefore an important need for collecting failure information. An interface must be developed accordingly for reporting – for each test case – whether a failure was discovered, either based on manual feeding of data from a tester or an interface towards a software tool for automatically performing functional testing. As was pointed out earlier in this chapter, the Musa-Okumoto model assumes that the faults triggering the observable failures are removed at the time of discovery.

#### 6.4.1.3 Test Time (3)

The time spent on the execution of a test case is measured in execution time – the amount of time for which the software uses the central processing unit. In order for JCoverage to measure execution time we suggest further code instrumentations. Code executing without interruptions can be instrumented in such a way that time is measured at execution start and termination respectively. The elapsed time can then be easily computed. By employing this approach all classes must be analyzed to find blocks of code that execute uninterrupted. There is no guarantee that the code will run without interruptions, since applications typically share CPU usage with the system being tested. An ideal scenario would be to execute tests with as few external processes as possible competing for processor time. Assuming that CPU usage of other applications is modest and constant would solve this problem.

#### 6.4.1.4 Pre-process Technique (4)

All the data labeled (1), (2) and (3) in figure 6.7 must be managed so that their respective values are observable after execution of each test case. A more detailed description was given in chapter 5 as to how the pre-process technique reduces the measured execution time of a test case if it failed to increase code coverage or the number of failures detected. If a test case is considered effective its execution time will remain unchanged. The initial test case will always be considered effective for obvious reasons and hence the actual execution time will be relayed from the pre-process (4) to the Musa-Okumoto model (7). The pre-process technique will add up the computed execution time of all test cases to yield a total time consumption, which will then be used by the Musa-Okumoto reliability model. The latter model also needs information as to the number of failures detected during testing (5).

#### 6.4.1.5 Musa-Okumoto (7)

Before the Musa-Okumoto model can be put to use it needs to be tailored to the software program under testing. This is done by means of two parameters of the model referred to as initial failure intensity  $\lambda$  and failure decay parameter  $\theta$ . These must be estimated for the projects to be tested. [MIO87] suggests that values be estimated by seeking those that match the observed failure times from the first stage of testing reasonably well. It does not seem natural to include estimation of these values in JCoverage or any other tool for that matter, since these parameters ought to be assessed by personnel that is capable of interpreting necessary data and making intelligent, thought-through decisions. We believe that this is best accomplished by one or more human capacities involved with the testing process. In [MIO87] several statistical methods are presented for estimating parameter values.

There is one particular aspect we would like to bring up with respect to the estimation of parameters and the use of the pre-process technique (4). When describing the Musa-Okumoto model in the previous chapter we said something along the lines of estimation possibly being performed with a statistical method based on observed failure times. In other words, time is used as a factor in estimating the model parameters, and we have now decided to control time through the pre-process technique. The question now is what will happen when the pre-process technique is employed. For instance, the model-specific parameters can be estimated to fit an observed curve without the use of the technique and the observed times are then pre-processed prior to being fed to the Musa-Okumoto model. As a result, we assume that the estimated failure time will no longer follow the same curve. The pre-process technique is a general technique also proposed to be used in conjunction with the Goel-Okumoto model. Thus, there are no indications as to how the technique will affect the parameters of the Musa-Okumoto model. In our

opinion it seems natural that the observed data will be pre-processed through the use of the technique to ensure equal treatment to all time measures.

In order to move from execution time to calendar time, knowledge is needed concerning the limiting factors of testing activities, with factors represented as a lack of competent testing personnel or shortage of programmers in charge of fault removal and repair. Since these limiting factors will vary during the course of the project, the relationship between execution time and calendar time is also set to vary. Information concerning this relationship must be obtained to be able to assess the amount of time required for achieving a given measure of reliability. The relationship between the two notions of time can also be found in periods when no code repair is taking place and the failure intensity remains constant, hence keeping the aforementioned relationship constant. This is destined to happen when, for instance, a software application moves from testing to normal, real-life operation. In this case it is the pattern of usage, ideally matching the operational profile of the program, that will determine the size of the relationship. Even though a system is operational for an entire day does not necessarily imply a correspondingly substantial use of execution time, since a number of programs – particularly those that require some kind of human interaction – typically remain idle for longer periods of time. For example, using Microsoft Word for eight hours may correspond to a mere hour of CPU processing.

#### 6.4.1.6 Reliability Estimates (8)

The Musa-Okumoto model provides a means of computing useful quantities. Determining a threshold for desired reliability renders possible an estimated measure of the number of failures that need to be detected in order to achieve the quality goal set forth. It is also possible to estimate the amount of time required to reach the same quality goal. Given that the time consumption is measured in terms of execution time, an overview is needed as to the overall resource consumption during testing in order to translate this measure into a number of working days. From this measure it is possible to derive an estimated date for when sufficient testing has been performed to yield the desired level of reliability. Yet another measure – Mean Time To Failure – may also be calculated through use of the Musa-Okumoto model. However, because the various measures result from employing a model that uses estimated parameters in its operation, there will always be uncertainty involved in the values produced. In other words, the uncertainty connected to parameter estimation transmits to the resulting values, with the uncertainty expressed in terms of confidence intervals.

#### 6.4.2 Solving the Calendar Time Issue

In describing the calendar time component [MIO87] emphasizes three factors that hamper testing in some way and hence form the basis for the relationship

between execution time and calendar time, namely failure identification personnel, failure correction personnel and computer time. As was explained in chapter 5, the need for the limited resources mentioned above is likely to vary from project to project. Thus, there is no fixed or constant relationship between the two notions of time during the different phases of testing. The use of automated testing utilities may contribute to reducing the impact of resource limitations of the first type – the number of failure identification personnel; especially if the tested system is a pre-existing application with ready-to-run tests, thus not requiring new test cases to be generated. We underline the use of the word *may* in the previous sentence since it might be tempting to cut down on personnel to save money, which would bring us back to where we were prior to introducing test automation.

In our opinion it does not appear to be imperative to implement the calendar time component unless there is a specific desire to compute calendar time for quantities during fault repair and testing. A simple overview of the current testing situation or status can be obtained even without employing a specific unit of time. This refers to the measure which indicates the required number of detected failures in order to attain a desired reliability target. Instead of estimating a ratio between execution time and calendar time at any time during testing, a constant ratio can be computed based on the expected profile of usage for the software during normal operation. This would suffice in converting measures such as MTTF from execution time to calendar time for the programs to be used. This proposed solution will not be capable of estimating the time consumption, measured in calendar time, required for testing purposes to achieve the desired reliability. It will, on the other hand, be able to verify the reliability of the end system, once fully developed and adequately tested.

Employing JCoverage in the realization of the proposed solution looks challenging, at best. As was pointed out earlier in this chapter there were several aspects of this tool that did not live up to our expectations. The line coverage measure was satisfactory and could possibly be used in a prospective realization. However, there are still problems related to re-use of test data once faults have been removed from the program under test. Had it not been for the fact that execution time needs to be measured for the tested application, then no major modifications to JCoverage would seem necessary. In this imaginary scenario the XML reports generated by JCoverage itself could have been input to the estimation module.

## 6.5 An Alternative Instrumentation Technique and Code Coverage Tool

Since the current version of JCoverage appears to be infeasible for our implementation approach, an alternative instrumentation technique as well as a competing coverage tool were briefly surveyed. The purpose of this section is to indicate the presence of more than one fixed way of dealing with code instrumentation, and also to show that automated code coverage utilities are no longer as scarce as once was the case.

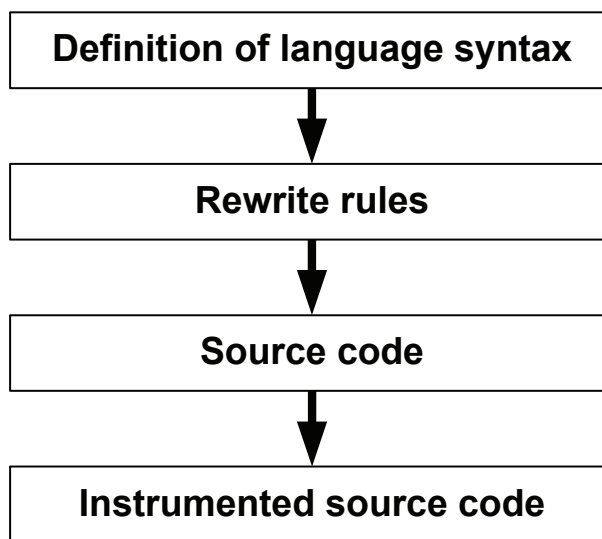
### 6.5.1 A General Technique for Source Code Instrumentation

A possible alternative to the instrumentation technique used in JCoverage is presented by Ira. D Baxter in his article titled *"Branch Coverage for Arbitrary Languages Made Easy"* [Bax98]. The article explains how programs that add instrumentation to the source code can easily be created for the purpose of code coverage measurement. The uniqueness of the presented technique is its generality, implying applicability to a wide range of programming languages for which code coverage measurement is generally not facilitated. Hence, solutions can be made that do not restrict support to popular languages only, such as Java, but rather offer support to an array of languages. The fundamental idea behind the approach described is to identify basic blocks, meaning parts of code which execute coherently as atomic units, and subsequently add instrumentation to each of these blocks. The coverage measure to be computed is branch coverage, although the notion of branch coverage stated in [Bax98] would come closer to qualify as basic block coverage according to our definition. The main challenge ahead is to identify basic blocks across different programming languages. In this case, a basic block can be explained as a piece or collection of code which is executed without any transfer of control to another part of the code being made.

The solution portrayed in [Bax98] makes use of so-called strength transformation systems that accept source-to-source rewrite rules – rules which can be defined for the transformation from source code to instrumented source code. Parsers that accompany a number of compiler toolkits will typically restrict operation to a given class of programming languages. This is the main reason why [Bax98] wants to employ industrial strength transformation systems. In order to use a particular programming language, its syntax must be defined. Since the industrial strength transformation systems are highly configurable it is a rather straightforward matter to define the syntax of the languages. Rewrite rules may, for instance, specify patterns to be replaced by a different pattern if a certain condition evaluates to `true`. According to [Bax98] it is easy to establish rewrite rules for procedural languages since these indicate all points of control transfer with the help of explicit syntax.

We believe that this technique might be used to track modifications to the source code and hence store code coverage data for the parts of the code left unchanged. Before the source code is transformed by means of rewrite rules, these rules are parsed and modified to adapt to the language to be transformed. Figure 6.10 depicted to the right illustrates the sequence of changes. In addition to the instrumentation itself, routines are required that initialize the

data structure and store code coverage information upon completion of testing. Functionality for presenting code coverage data obtained from the performed tests has to be in place as well. One alternative could be to add the instrumentation to the code ourselves, thus gaining full control of the instrumentation and being able to ensure that identification of the different code elements remain unchanged even when modifications are made to the code. This strategy, although time- and resource-consuming compared to other solutions, has the prospect of yielding better results. A number of things would have to be taken explicit care of, such as the assignment of unique identifiers to points of instrumentation.



*Figure 6.10*  
*Sequence of modifications to code*

### 6.5.2 A Brief Presentation of Clover

Several alternatives to JCoverage are available on the market, some of which are commercial products. We opted for an on-the-surface look at a tool belonging to the latter category. Clover is developed by Cenqua Pty Ltd. and is a Java code coverage analysis tool which offers one or two coverage measures not supported by JCoverage. The developer has also made available a version supporting code coverage measurement for applications developed through Microsoft .NET. The following code coverage measures are supported in Clover:

- ✓ Method coverage
- ✓ Branch coverage
- ✓ Statement coverage

These three are measured for projects, packages, files and classes and are subsequently presented in reports formatted as either HTML, XML, PDF or

plain text. Clover “accurately measures per statement coverage, rather than per line coverage” [Cen05]. Historical reports – mentioned as a desirable feature of automated code coverage tools in chapter 2.5.2 – can be generated to illustrate the development of each coverage measure, as well as other project metrics, during the course of the project. Another useful facet of Clover is the existence of several plug-ins with support for integrated development environments (IDE) used in Java development. These allow developers to keep an eye on coverage measures of different parts of the code without leaving the IDE.

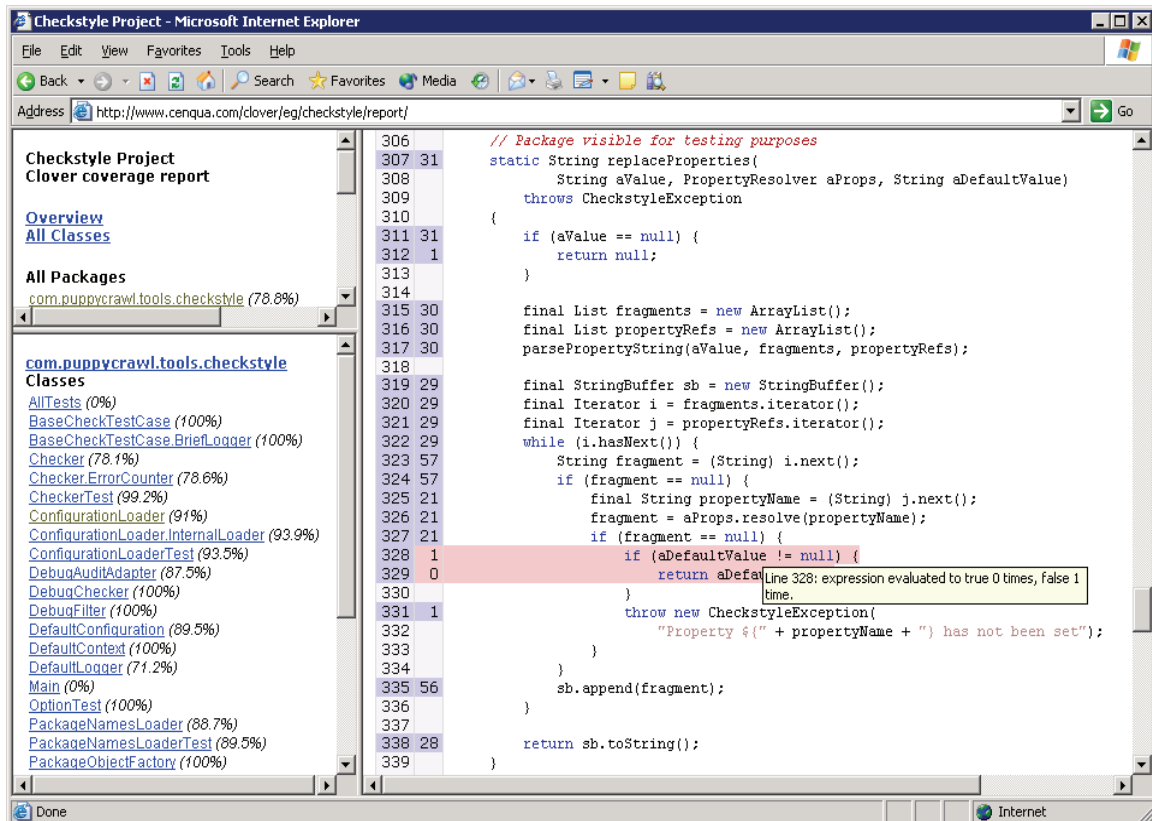


Figure 6.11  
HTML report generated by Clover

Clover is developed in Java and measures code coverage for applications written in Java, as was the case with JCoverage. The basic operation of the two tools is similar, following the steps of code instrumentation, test execution and report generation. There are, however, a few noteworthy differences. Most significantly, JCoverage, being an open-source product, can be attained by means of a GNU Public License, whereas Clover is a fully commercial product. Further, JCoverage does not instrument lines that call *log4j* – a logger which can optionally be employed for low-level debugging. This is said to be an advantage, since the instrumentation of the aforementioned lines would run the risk of impacting code coverage metrics by making calls to the logger. Finally, the two tools differ somewhat in how



they define branch coverage, with Clover's implementation of the measure corresponding significantly better with our definition. The HTML version of the generated report, shown in figure 6.11 on the preceding page, informs that the `if` statement on line 328 has been executed once, at which time it evaluated to `false`. The graphical interface for navigation and presenting information is comparable to the one found in the API specifications of Java or in javadoc-generated documentation of individual projects.

*"Great is the art of beginning, but  
greater is the art of ending."*

Lazurus Long

## CHAPTER 7:

# Conclusion

Having embarked on a journey that has seen us through intermediate destinations such as code coverage measures, software reliability models and estimation, debates revolving the relationship between code coverage and reliability, not to mention the implementation of code coverage basics into software utilities, it is now time to bring the expedition to a close, ponder on the experiences gained and where to look next.

## 7.1 Summary

**W**ith more and more actors entering the software market, competition is getting fiercer by the day. As additional products become available customers are left with the daunting task of selecting the right one, thus indirectly putting pressure on software businesses to develop first-class products. Hence it is far from surprising that modern-day software development methodologies, including the likes of Rational Unified Process and eXtreme Programming, pay significant attention to the role of testing in achieving reliable, high-quality applications. There is, however, always room for improvement. The hunt for means of increasing software reliability is still on, with code coverage playing a non-negligible role.

The rationale for employing code coverage in testing efforts is apparent. Although there is no guarantee that all existing faults will be uncovered in spite of complete code coverage being reported, faults will definitely not be found in parts of the code which have been left unexercised. Code coverage can improve test set quality, reveal flaws in test implementation and increase our understanding of existing tests. The presence of different coverage measures provides developers and testers with an array of options as to what set of measures to employ for various projects. Although the latter may appear to be a blessing, it is, however, just as much a challenge in disguise. The lack of a standard culminates into an issue of measure definitions. As a result, each publication on the topic must first elaborate on an exhaustive definition of each coverage measure used at a later stage, before moving on to the core content. During the course of this project we have encountered instances where a definition of a particular coverage measure has matched that of a different measure, coming from an alternative source.

Automation of reliability models and their estimation is far from straightforward. The main challenge rests in the context-dependent parameters which require substantial human experience, comprehension and interpretation. The uncertainty involved in these parameters will evidently impact the accuracy of resulting reliability estimates. Thus, great caution must be taken in the employment of such estimates, preferably by considering them as merely estimates and not some fixed, definite quantities of product quality. The difficulties in estimating software reliability makes it tempting to assume that code coverage may not only contribute to increased reliability, but also be utilized as a predictor of reliability.

The proposed relationship between code coverage and reliability has in many ways been the core theme of this assignment. Having examined a two-digit number of articles on the aforementioned relationship, a definite and agreed-upon conclusion with respect to the matter seems far away. The vast majority of experiments referred to in relevant literature reports positive findings, more or less, as to the existence of a claimed relationship. However, the big question is if these results are overthrown by theoretical and critical remarks made by others, having reservations about the internal validity of the experiments performed. After all, test intensity increases as code coverage increases, thus making it highly questionable whether code coverage contributions amount to anything beyond a negligible increase in reliability, once test intensity is taken into account. Also, the absence of an operational profile in a code coverage-driven test strategy is deemed to result in even, unweighted testing, hence possibly dedicating less attention to high-usage functionality, relative to an operational profile, at the expense of functionality less frequently employed. As a matter of fact, the level of testing required for low-usage areas of a software application strikes us a matter of great dissension among authors.

Finally, the report was rounded off by examining the internals of a tool for automated code coverage analysis. The previously noted problem of non-uniform definitions of coverage measures manifested itself in the implementation of JCoverage. In addition to suggesting imminent improvements we also discovered obstacles on our way to proposing an approach for integrating reliability estimation into JCoverage. We realize that questions can be raised as to the implementability of one or more propositions made, but pinpoint the fact that we have mainly focused on unearthing opportunities rather than deepen into existing constraints. On a final note, we believe that the emergence of code coverage tools, commercial as well as open-source, suggests that code coverage is destined to remain on the horizon for some time to come.

## 7.2 Further Work

**D**uring the course of this project we have stumbled upon ideas and aspects related to both code coverage and reliability that we have had to neglect because of lacking relevance or shortage of time. One of these ideas would be to go through with the implementation of a reliability estimation module, possibly integrated into an appropriate tool for tracking code coverage. Upon realization of such an estimation tool or module an experiment could be performed to determine or evaluate its usability. The resulting observations may then be used to assess improvements or decide whether such an implementation was feasible in the first place.

The reliability of software applications is affected by factors of the system development process, with the level or quality of performed testing being one such factor. In order to improve reliability estimation, propositions are made to quantify these factors and subsequently use them in the estimation process. However, through the work of this project we have come to learn that 100% code coverage by no means implies a code free of faults. We have also seen how mutation coverage can be employed to demonstrate the incapability of tests in discovering all existing faults. Thus, by combining these observations it might prove interesting to uncover how an amalgamation of mutation coverage and reliability models would change reliability estimates. A measure of mutation coverage should then be combined with the computed reliability estimate, so that the varying failure-detecting abilities of tests used during estimation can be taken into account.

In the early stages of this report we underlined the danger in designing tests with the explicit purpose of achieving complete coverage as soon as possible. We also commented on the fact that tests created within agile development methodologies such as eXtreme Programming, stand the risk of reaching 100% coverage. With this in mind, it would be interesting to acquire more knowledge as to the relationship between reliability and development methodology when reliability is estimated by means of code coverage. We also touched upon the position of code coverage after the introduction of agile, test-driven methodologies. This issue may deserve more attention, but will most likely require tedious research.

## APPENDIX A:

## ☑ References &amp; Bibliography

- [Apa05] Apache Maven Project  
– *What is Maven?*  
<http://maven.apache.org/about/whatisraven.html>
- [Bax98] Ira D. Baxter  
– *Branch Coverage for Arbitrary Languages Made Easy*  
<http://www.semdesigns.com/Company/Publications/TestCoverage.pdf>
- [Bei90] Boris Beizer  
– *Software Testing Techniques, second edition*  
ISBN 0-442-20672-0
- [BP00] Lionel C. Briand & Dietmar Pfahl  
– *Using Simulation for Assessing the Real Impact of Test-Coverage on Defect-Coverage*  
<http://www.sce.carleton.ca/faculty/briand/pubs/isern-99-05.pdf>
- [Cen05] Cenqua Pty Ltd.  
– *Cenqua Clover Features*  
<http://www.cenqua.com/clover/featurelist.html>
- [Cor04] Steve Cornett, Bullseye Testing Technology  
– *Code Coverage Analysis*  
<http://www.bullseye.com/coverage.html>
- [CLW96] Mei-Hwa Chen, Michael R. Lyu & Eric Wong  
– *An Empirical Study of the Correlation between Code Coverage and Reliability Estimation*  
IEEE 0-8186-7364-8/96
- [CLW01] Mei-Hwa Chen, Michael R. Lyu & Eric Wong  
– *Effect of Code Coverage on Software Reliability Measurement*  
[http://www.cse.cuhk.edu.hk/~lyu/paper\\_pdf/paper20.pdf](http://www.cse.cuhk.edu.hk/~lyu/paper_pdf/paper20.pdf)
- [Dam04] Craig E. Damon  
– *Software Engineering Course Notes*  
<http://www.cs.uvm.edu/%7Ecdamon/cs205/notes.pdf>

- [FGMP95] Fabio Del Frate, Praerit Garg, Aditya Mathur & Alberto Pasquini  
– *On the Correlation between Code Coverage and Software Reliability*  
<http://ieeexplore.ieee.org/iel3/3555/10649/00497650.pdf?arnumber=497650>
- [Gar94] Praerit Garg, Purdue University  
– *Investigating Coverage-Reliability Relationship and Sensitivity of Reliability to Errors in the Operational Profile*  
<http://portal.acm.org/citation.cfm?id=782204>
- [Ham94] Dick Hamlet, Portland State University  
– *Connecting Test Coverage to Software Dependability*  
<http://ieeexplore.ieee.org/iel4/1008/7986/00341368.pdf?arnumber=341368>
- [JM92] Raymond Jacoby & Kaori Masuzawa  
– *Test Coverage Dependent Software Reliability Estimation by the HGD Model*  
<http://ieeexplore.ieee.org/iel2/434/7090/00285845.pdf?arnumber=285845>
- [JM94] Pankaj Jalote & Y. R. Muralidhara  
– *A Coverage Based Model for Software Reliability Estimation*  
<http://ieeexplore.ieee.org/xpl/tocresult.jsp?isnumber=11601>
- [Kos04] Lasse Koskela, Accenture Technology Solutions  
– *Introduction to Code Coverage*  
<http://www.javaranch.com/newsletter/200401/IntroToCodeCoverage.html>
- [KSMG96] Richard M. Karcich, Robert Skibbe, Aditya P. Mathur & Praerit Garg  
– *On Software Reliability and Code Coverage*  
<http://ieeexplore.ieee.org/iel3/3554/10648/00499668.pdf?arnumber=499668>
- [LHL94] Michael R. Lyu, J. R. Horgan & Saul London, Bell Communications Research  
– *A Coverage Analysis Tool for the Effectiveness of Software Testing*  
<http://ieeexplore.ieee.org/xpl/tocresult.jsp?isNumber=13562>

- [Lyu05] Michael Rung-Tsong Lyu  
– *Software Reliability Engineering*  
<http://appsrv.cse.cuhk.edu.hk/~csc6001/SRE2005.ppt>
- [Mal+94] Yashwant Malaiya, Naixin Li & Jim Bieman, Colorado State University – Rick Karcich & Bob Skibbe, StorageTek  
– *The Relationship Between Test Coverage and Reliability*  
Reliability Engineering, Nov. 1994, pp. 186-195.
- [Mar99] Brian Marick, Testing Foundations  
– *How to Misuse Code Coverage*  
<http://www.testing.com/writings/coverage.pdf>
- [MIO87] John D. Musa, Anthony Iannino & Kazuhira Okumoto  
– *Software Reliability: Measurement, Prediction, Application*  
ISBN 0-07-044093-X
- [MLBK02] Yashwant Malaiya, Michael Naixin Li, James Bieman & Rick Karcich  
– *Software Reliability Growth With Test Coverage*  
<http://www.cs.colostate.edu/~bieman/Pubs/Malaiya-et alPublished02.pdf>
- [MR91] Marvin Rausand  
– *Risikoanalyse – Veiledning til NS5814*  
ISBN 8251909708
- [MR96] Aditya P. Mathur & Vernon J. Rego  
– *White-box Models for the Estimation of Software Reliability*  
<http://www.cs.purdue.edu/AnnualReports/95/AR95Book-108.html>
- [Mus04] John D. Musa  
– *Software Reliability Engineering: More Reliable Software Faster And Cheaper*  
ISBN 1418493872
- [NJH03] Hung Q. Nguyen, Bob Johnson, Michael Hackett  
– *Testing Applications on the Web, Second Edition*  
ISBN 0-471-20100-6
- [Pai02] Ganesh J. Pai, University of Virginia  
– *A Survey of Software Reliability Models*  
[www.ece.virginia.edu/~gjp5j/professional/coursework/gjp-cs651-SRMsurvey.pdf](http://www.ece.virginia.edu/~gjp5j/professional/coursework/gjp-cs651-SRMsurvey.pdf)

- [TS04] The Testing Standards Working Party  
– *Reliability Guidelines*  
[http://www.testingstandards.co.uk/reliability\\_guidelines.htm](http://www.testingstandards.co.uk/reliability_guidelines.htm)
- [Wik05] Wikipedia  
– *The Free Encyclopedia*  
[http://en.wikipedia.org/wiki/Main\\_Page](http://en.wikipedia.org/wiki/Main_Page)
- [Woh+00] Claes Wohlin, Per Runeson, Martin Höst, Magnus Ohlsson,  
Björn Regnell & Anders Wesslén, Lund University Sweden  
– *Experimentation in Software Engineering: An Introduction*  
ISBN 0-7923-8682-5



## APPENDIX B:

**☑ Glossary****✓ Application Programming Interface**

*Definitions of inter-program communication, typically used for abstractions between applications on higher and lower level.*

**✓ Black box testing**

*Also known as functional testing. Black box testing involves testing the system from a user's perspective, exposing it to different types of input and checking whether or not the resulting output is in accordance with the specification [Bei90].*

**✓ Enumerable**

*Collective term denoting different quantifiable units of program code related to code coverage measures, including statements, methods, blocks, etcetera.*

**✓ Error**

*Incorrect behavior resulting from a fault [Bei90].*

**✓ Error handling**

*See exception handling.*

**✓ Exception handling**

*Also known as error handling. Exception handling consists of code that is called upon in the case of system errors requiring treatment. The code generally involves operations for restoring system services or storing data and notifying users prior to system shutdown.*

**✓ External validity**

*Validity type that deals with the extent to which results from an empirical experiment can be generalized across contexts.*

**✓ Failure**

*Incorrect behavior of a component [Bei90]. The lack of ability of a component, equipment, subsystem or system to perform its intended function as designed [Wik05].*

**✓ Fault**

*Incorrect program or data object – a bug [Bei90]. An abnormal condition or defect at the component, equipment or subsystem level, which may lead to a failure.*

**✓ GNU Public License**

*License type that provides everyone with the right to use, copy, modify and re-distribute the product as long as the rights specified in the license are passed on. Software which is distributed with this license associated with it, is referred to as open-source.*

**✓ HTML**

*Hypertext Markup Language. Web page format for documents.*

**✓ Industrial Strength Transformation Systems**

*Tools employed for large scale reengineering, software quality analysis and enhancement reverse engineering [Bax98].*

**✓ Integrated Development Environment**

*Application for software development where tools such as source code editor, compiler, interpreter, automation tool and version control system are combined and accessible to developers by means of a graphical user interface.*

**✓ Internal validity**

*Validity type that deals with the extent to which observations from empirical experiments can actually be said to be caused by the factor accounted for – that there is a causal relationship between treatment and outcome.*

**✓ Java Virtual Machine - JVM**

*Software program available to several platforms. JVM emulates a computer where byte code can be executed and given access to computer resources, thus facilitating execution of Java applications on any platform running a JVM implementation.*

**✓ log4j**

*Tool for logging information to file and used as low-tech method for debugging.*

**✓ Maven**

*A software project management and comprehension tool. Based on the concept of a project object model, Maven can manage a project's build, reporting and documentation from a central piece of information [Apa05].*

**✓ Object code**

*Intermediate representation of code generated by a compiler after it processes a source code file [Wik05].*

**✓ Object-oriented testing**

*Testing techniques geared towards applications developed with object-oriented development methodologies, taking into account distinctive characteristics of object-orientation such as polymorphism, encapsulation and inheritance.*

**✓ Open-source**

*Software whose source code is made available, hence giving users the right and opportunity to create tailored versions of the program.*

**✓ Package**

*Intended as a collection of closely related Java classes that solve a certain type of problems or deal with a specific, coherent set of activities.*

**✓ Parser**

*Software program which analyzes grammatical expressions of program input, based on a formal grammar [Wik05].*

**✓ PDF**

*Abbreviation for Portable Document Format – a file format developed by Adobe for documents independent of software, hardware and operating system. The open-standard format combines text, graphics and images.*

**✓ Polymorphism**

*A software property enabling the existence of several implementations of methods in object-oriented programming languages. This is the case when classes inherit from the same class and implement new functionality in the methods of this class.*

**✓ Quality attribute**

*A measurable part of the system that is used to quantify its quality.*

**✓ Race coverage**

*A coverage measure that reports whether two or more threads execute the same piece of code simultaneously. Race coverage can be employed to detect failures in synchronizing access to resources [Cor04].*

**✓ Random testing**

*Testing technique for random generation of test cases, implying that new tests are selected independently of previously executed tests.*

**✓ Robustness**

*The resilience of the system, especially when under stress or when confronted with invalid input [Wik05].*

**✓ Source-to-source rewrite rule**

*Description of how a text pattern will be transformed into a different text pattern upon the satisfaction of a particular condition.*

**✓ Structural testing**

*See white box testing.*

**✓ Test-driven development methodologies**

*Methodologies for developing software where test cases are created prior to the code attempting to satisfy it, with the purpose of controlling a given functional requirement.*

**✓ Test set**

*A set of input data and expected output data used to test a system.*

**✓ Trustworthiness**

*Software property denoting dependable, stable and fault-tolerant operation while yielding predictable results, typically in one or more functions deemed critical.*

**✓ White box testing**

*Performing tests on functions that are not directly available for the users of the final product.*

**✓ Wrapper class**

*A software class which wraps an inner class and forms the interface of the class it wraps. All other classes must now communicate via the wrapper class.*

**✓ XML**

*Extensible Markup Language. A W3C recommendation for creating special-purpose markup languages [Wik05].*

**APPENDIX C:****☑ Index**

- actual system usage, 41, 42, 58, 61
- actual usage, 21
- all-uses coverage, 11
- AND operator, 9
- Ant, 20, 67, 68, 69, 73
- Apache Ant, 67
- API, 90
- application domain, 52, 64, 80
- ATAC, 59
- automated code coverage
  - utilities, 87
- automated coverage tool, 59
- automated testing, 49, 86
- automated testing utilities, 86
- automated tools, 4
- average failure rate, 46
- basic block, 7, 8, 77, 87
- basic block coverage, 7, 8, 77, 87
- Basic instrumentation, 19
- Basic Musa, IV, 38, 39
- basic path coverage, 8
- Bayesian software reliability
  - growth model, 38
- blank lines, 70
- block coverage, 8, 14, 77, 81
- block of code, 8, 75, 76, 79
- branch coverage, 8, 14, 15, 16, 53, 71, 72, 73, 74, 75, 76, 77, 78, 79, 82, 87, 90
- branches, 9, 10, 11, 13
- build tools, 68
- Byte code, 68
- byte code instrumentation, 69
- calendar time, 28, 30, 49, 60, 63, 64, 85, 86
- calendar time component, 60, 63, 85, 86
- catastrophic failure, 40
- central processing unit, 29, 83
- Clover, V, 88, 89, 94
- code complexity, 52
- Code coverage, 1, 2, 16, 18, 52, 73, 91
- code coverage analysis, I, 2, 3, 4, 5, 6, 15, 16, 17, 18, 20, 21, 40, 67, 88, 92
- code coverage analysis tool, 3, 88
- code coverage measures, 2, 6, 7, 23, 50, 66, 67, 78, 79, 88, 91
- compiler toolkits, 87
- component failures, 27
- components, 3, 25, 27, 28, 48, 49, 51, 60
- compression ratio, 65, 66
- computational-use, 11
- computer systems, I, 5, 24
- computer time, 63, 64, 86
- Condition coverage, 9, 10, 13
- constructional flaws, 27
- control flow, 9, 11
- control structures, 8, 13
- coverage data, 56, 60, 68, 74, 80, 88
- coverage history, 46, 51
- coverage measure, I, 2, 3, 5, 7, 10, 11, 12, 13, 16, 17, 20, 48, 52, 53, 55, 59, 67, 69, 70, 72, 74, 77, 78, 79, 82, 87, 88, 89, 91, 92
- coverage measures, I, 2, 3, 5, 7, 11, 12, 13, 16, 17, 20, 48, 52, 53, 55, 59, 69, 70, 72, 74, 82, 88, 89, 91, 92
- coverage reduction, 54
- coverage report, 14, 20, 72, 73, 74, 76
- coverage tool, 16, 19, 20, 87, 89, 92

- CPU, 29, 30, 63, 64, 83, 85
- cumulative code coverage, 65, 66
- data collection, I, 42, 46, 47, 51
- data flow coverage, 11
- data types, 14
- data-centric systems, 15
- debug data, 49
- debugging process, 63
- decision coverage, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 77
- decision points, 1, 15
- defect coverage, 3, 7, 46, 48, 51, 55, 56, 57
- defect distribution, 57
- defect types, 57
- defects, 3, 7, 17, 49, 55
- def-use pair, 11
- detectability, 50, 51
- detectability profile, 50, 51
- development environment, 22, 53, 60, 89
- development methodologies, 4, 6, 20, 23, 91, 93
- development process, 15, 22, 35, 36, 37, 51, 68, 93
- development techniques, 47
- domain, 2, 24, 41, 44, 52, 54
- ease of test, 49, 50
- Eclipse, 67, 68
- effective testing efforts, 45
- empirical experiments, 29, 55, 82
- empirical investigation, 1, 3, 46, 54, 55
- empirical investigations, 1, 3, 55
- encapsulation, 22
- end product, 1, 2, 7, 18, 22, 41, 55
- enumerable, 50
- environments, 31, 57
- equivalence partitioning, 34
- error types, 52
- error-handling, 49
- estimating parameter values, 84
- exception handlers, 8
- execution control, 11
- execution environment, 33
- execution time, 28, 30, 37, 38, 39, 45, 49, 60, 61, 62, 63, 64, 65, 66, 83, 84, 85, 86
- experiments, 16, 29, 47, 51, 52, 53, 55, 56, 58, 59, 92
- external validity, 53
- failure behavior, 32, 36, 60
- failure correction personnel, 63, 86
- failure data, 28, 36, 38, 53
- failure detection probability, 58
- failure identification personnel, 86
- failure information, 83
- failure intensity, 36, 42, 58, 61, 62, 64, 85
- failure intensity decay
  - parameter, 61, 62, 64
- failure intensity function, 61
- failure intervals, 36
- failure times, 35, 63, 84
- failures experienced, 30, 61
- fault density, 52
- fault detectability, 45
- fault detection, 18, 35
- fault distribution, 52, 59
- fault masking, 54
- fault repair, 86
- fault seeding, 13, 54, 57
- fault tree, 25
- flow graph, 46, 51
- functional testing, 15, 43, 44, 83, 98
- GNU Public License, 67, 89
- Goel and Okumoto, IV, 38
- Goel-Okumoto, 1, 38, 39, 45, 53, 65, 84
- hardware, 27, 35
- hardware component, 27
- HGD model, 49
- Historical reports, 20, 89
- HTML, 72, 73, 88, 89, 90, 98
- human interaction, 85
- hypothesis, 12, 47, 48, 52, 53, 57, 59

- IDE, 67, 89
- idle time, 29, 30
- IEEE, II, 40, 94
- if clause, 71, 76, 77, 79
- if construct, 71, 75, 76, 77
- if expression, 9, 77, 79
- if instructions, 71
- if statement, 8, 9, 10, 76, 77, 78, 79, 90
- industrial strength
  - transformation systems, 87
- ineffective test cases, 45
- initial failure intensity, 61, 62, 64, 84
- input parameter, 29, 30, 71
- input parameters, 29
- input space, 33, 34
- input space coverage, 33, 35
- instrumentation, 19, 68, 69, 70, 71, 75, 79, 87, 88, 89
- instrumentation technique, 87
- Integrated Development Environment, 67
- internal validity, 55, 92
- interrupt handlers, 8
- invalid states, 33
- Java Virtual Machine, 19, 68, 70
- JCoverage, IV, V, 5, 19, 38, 60, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 86, 87, 88, 89, 92
- Jelinski and Moranda, IV, 38
- jump coverage, 11
- JVM, 19, 69, 70
- limited time, 17
- limiting factor, 64, 85
- limiting factors, 85
- line coverage, 7, 72, 73, 74, 75, 76, 78, 79, 86, 89
- line of code, 70, 71, 78, 79, 81, 82
- line of source code, 17, 70, 71
- linear code sequence, 11
- linear progression, 50
- Littlewood-Verrall, IV, 38
- log4j, 89
- logarithmic Poisson execution model, 60
- logical expressions, 17
- logical operators, 8, 57
- loops, 8, 10
- low-usage functionality, 40, 43, 49, 50, 58, 59
- low-usage modules, 49
- Maven, 20, 94
- maximum likelihood estimation, 63
- method, 8, 10, 11, 12, 50, 63, 71, 75, 79, 83, 84
- Method coverage, 88
- module coverage, 53
- modules, 3, 12, 17, 28, 37, 46, 47, 59
- MTCTF, 53
- MTTF, 58, 62, 86
- multiple condition coverage, 9, 10, 15, 78
- multithreaded applications, 13
- Musa-Okumoto, IV, V, 1, 5, 38, 39, 42, 45, 51, 53, 60, 61, 62, 63, 65, 66, 67, 80, 82, 83, 84, 85
- mutation coverage, 13, 43, 57, 93
- mutation criteria, 57
- node, 46, 47, 51
- non-homogenous Poisson process, 32
- non-random testing, 49
- notion of time, 29, 30, 60
- number of failures, 28, 30, 31, 32, 35, 36, 37, 38, 39, 53, 54, 61, 62, 63, 65, 66, 83, 84, 85
- number of faults, 33, 38, 44, 49, 50, 51, 59
- object code, 7, 13
- observed data, 63, 85
- off-by-one errors, 12
- open-source, I, 60, 67, 89, 92
- operating systems, 68
- operational profile, 17, 21, 33, 34, 35, 37, 38, 40, 41, 42, 43, 44, 46, 48, 50, 51, 52, 53, 54, 58, 59, 61, 85, 92

- OR operator, 9  
 overestimate, 3, 43  
 overhead factors, 64  
 packages, 72, 73, 88  
 parameter estimation, 62, 85  
 parameters, 36, 38, 47, 52, 60, 62, 64, 65, 66, 79, 84, 85, 92  
 path, 4, 6, 10, 11, 15, 46  
 path coverage, 10, 11, 15  
 path reliability, 46  
 path testing, 6, 10  
 PDF, 88  
 Poisson distributions, 32  
 polymorphism, 22, 23  
 predicate coverage, 15  
 predicate-use, 11  
 Prediction, 24, 36, 96  
 pre-process, 1, 5, 20, 44, 45, 66, 67, 80, 81, 83, 84  
 pre-processing, 1, 20, 67, 80  
 pre-processing technique, 67, 80  
 prescribed work periods, 64  
 probability distribution, 31, 35, 36, 39, 61  
 procedural languages, 87  
 production phase, 58  
 profile precision, 42  
 programming languages, 16, 23, 87  
 programming-centric, 21  
 quality assurance, 2, 27  
 quality attributes, 3  
 quality measure, 6  
 quality of fault repairs, 61  
 quality of repair, 35  
 race coverage, 13  
 random processes, 31, 35  
 random testing, 47, 58  
 random tests, 57  
 random variables, 31  
 redundant components, 25  
 redundant test cases, 6  
 reliability, I, II, 1, 2, 3, 4, 5, 21, 24, 25, 27, 28, 29, 30, 31, 33, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 57, 58, 60, 61, 62, 64, 65, 66, 67, 80, 81, 82, 84, 85, 86, 91, 92, 93, 97  
 reliability estimates, 1, 2, 27, 29, 35, 36, 37, 42, 43, 44, 47, 51, 53, 80, 92, 93  
 reliability estimation, I, II, 1, 3, 4, 5, 21, 28, 45, 48, 50, 53, 60, 64, 66, 80, 81, 92, 93  
 reliability goal, 62  
 reliability growth models, 3, 29, 30, 40, 43, 44, 45  
 reliability models, 1, 4, 29, 35, 36, 37, 38, 39, 41, 42, 43, 44, 45, 47, 48, 52, 53, 60, 61, 62, 65, 82, 92, 93  
 requirements specification, 15  
 resource consumption, 37, 63, 64, 85  
 resource limitations, 33, 86  
 resource utilization, 64  
 robustness, 33  
 saturation effect, 3, 40, 43, 44  
 Shooman, IV, 38  
 short-circuited operators, 9  
 short-circuits, 79  
 smoothing parameters, 65  
 software developers, 16, 27, 64  
 software modules, 27  
 Software reliability, 39  
 software reliability growth models, 24, 28  
 software reliability models, 5, 28, 35, 91  
 software tools, I, 2, 6  
 software vendors, I, 1  
 sound assumptions, 36  
 source code, 6, 7, 11, 13, 16, 18, 19, 20, 29, 35, 40, 54, 57, 58, 67, 69, 70, 71, 73, 79, 87, 88  
 source-to-source rewrite rules, 87  
 statement coverage, 7, 8, 9, 16, 48, 78, 89  
 statistical precision, 59  
 structural testing, 6, 15, 43



- sub-expressions, 9, 10, 13
- subsume, 2, 14, 15
- subsumption hierarchy, 14
- sufficient line coverage, 78
- sufficient testing, 18, 21, 47, 85
- switch statements, 8
- syntax, 70, 87
- system failure, 25, 27, 34
- system reliability, 27, 37, 46, 62
- system usage, 29, 33
- tasks, 63, 67
- test case, 3, 6, 9, 12, 14, 16, 17, 42, 45, 49, 53, 54, 56, 57, 65, 66, 81, 83, 84, 86
- test coverage, 6, 7
- test data, 14, 33, 44, 54, 70, 86
- test effort, 1
- test intensity, 3, 46, 48, 55, 56, 57, 92
- test phase, 60
- test set, 7, 11, 16, 17, 18, 21, 45, 50, 57, 58, 73, 91
- test team, 16, 20, 21, 63
- testability, 40, 49, 50
- test-driven development
  - methodologies, 5, 21
  - test-driven methodologies, 4, 22, 93
- testing, 1, 2, 3, 6, 7, 10, 12, 15, 16, 17, 18, 21, 22, 23, 24, 28, 30, 31, 32, 33, 35, 36, 38, 39, 40, 41, 42, 43, 44, 45, 46, 48, 49, 50, 51, 52, 53, 55, 56, 57, 58, 59, 61, 62, 63, 64, 65, 68, 71, 75, 78, 82, 83, 84, 85, 86, 88, 91, 92, 93, 96, 98
- testing activities, 17, 57, 85
- testing phases, 57
- testing process, 53, 84
- testing strategy, 1, 3, 33
- time measures, 85
- time-based models, 36
- transformation, 87
- trustworthiness, 40, 46, 58
- uncertainty, 31, 36, 62, 85, 92
- uniformly distributed, 57
- unit of time, 35, 86
- user needs, 21
- user perspective, 21
- White box models, 37
- wrapper classes, 20
- XML, 67, 69, 72, 73, 74, 86, 88, 98