HAL
open science

Validation de métriques de testabilité logicielle pour les programmes objets

Muhammad Rabee Shaheen

To cite this version:

Muhammad Rabee Shaheen. Validation de métriques de testabilité logicielle pour les programmes objets. Software Engineering [cs.SE]. Université Joseph-Fourier - Grenoble I, 2009. English. NNT : . tel-00978771

HAL Id: tel-00978771

https://theses.hal.science/tel-00978771v1

Submitted on 14 Apr 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Université Joseph Fourier - Grenoble I

THESE

pour obtenir le grade de

DOCTEUR DE
L'UNIVERSITÉ JOSEPH FOURIER

Discipline : INFORMATIQUE

préparée au Laboratoire d'Informatique de Grenoble

présentée et soutenue publiquement

par

Muhammad Rabee SHAHEEN

le 07 Octobre 2009

Validation de Métriques de Testabilité Logicielle pour Les Programmes Objets

JURY

M. Daniel DEVEAUX	*Rapporteur*
M. Yves le TRAON	*Rapporteur*
Mme. Chantal ROBACH	*Examinatrice*
Mme. Virginie WIELS	*Examinatrice*
M. Farid OUABDESSELAM	*Directeur*
Mme. Lydie du BOUSQUET	*Co-directrice*

To my mother
To the spirit of my father
To all my family

Acknowledgements

This thesis has been achieved in the Laboratoire d'Informatique de Grenoble. It is my pleasure to convey my gratitude to all the members in the LIG laboratory, especially the members of VASCO team that I joined since 2006.

I gratefully acknowledge:
Farid Oubedsslam and Lady du Bousquet, my supervisors for their guidance, advice, valuable discussion, patience and their constructive comments during the three years.

Yves Ledru, head of VASCO team, for his friendship over the past two years...

Roland Groz, Akram Idani, Catherine Oriat, Jean-Luc Richier and all other team members ...

Chantal Robach and Virginie Wiels, who accepted to judge this work.

Daniel Deveaux and Yves le Traon, who accepted to review this thesis.

administrative and technical team, for their great availability...

all my friends and everyone whom I spent with him a nice time...

Words fail me express my appreciation to my parents and all my family to all their supports and their love...

Finally, I would like to acknowledge everyone who was behind the successful realization of this thesis, as well as expressing my apology that I could not mention personally one by one...

Résumé

Pour les systèmes logiciels, la méthode de validation la plus utilisée est le test. Tester consiste en l'exécution du logiciel en sélectionnant des données et en observant/jugeant les sorties. C'est un processus souvent coûteux. L'effort de test est difficile à caractériser précisément. Il dépend de la complexité du logiciel, des objectifs en termes de validation, des outils et du processus développement.

La testabilité logicielle s'intéresse à caractériser et prédire l'effort de test. Cela est nécessaire pour estimer le travail de test, prévoir les coûts , planifier et organiser le travail. De nombreuses mesures ont été proposées dans la littérature comme indicateurs du coût du test. Ces mesures sont focalisées sur l'évaluation de certains attributs qui peuvent rendre le test difficile. D'autre approches proposent de repèrer des constructions difficiles à tester à l'aide de patrons (*testability antipatterns*) par exemple.

D'une façon générale, peu d'études ont été réalisées pour valider ces métriques ou patrons. Certaines de ces études donnent des résultats contradictoires. Or il est essentiel de fournir des informations non biaisées.

Notre travail de thèse porte en premier lieu sur la validation de certaines métriques de testabilité proposées pour la prédiction du coût du test de programmes objet. Notre approche s'appuie sur une mise en relation des métriques et des stratégies de test et vise à l'établissement de corrélation entre coût prédictive et coût effective. Ceci nous a conduit à raffiner certaines des métriques étudiées.

Dans un second temps, nous nous sommes intéressés à des patrons (*testability antipatterns*) visant à détecter des faiblesses dans le code vis à vis du test. Le but de cette étude est de comprendre à quels moments ces constructions sont introduites dans le code, afin de les repérer le plus efficacement possible.

Abstract

The most used validation method for software is testing. Testing process consists of executing the program by selecting a set of data and observing the outputs of the program. The testing process is a costly in terms of time and money. Estimating the effort of testing is important in order to be able to plan the testing phase. Therefore, some reliable indicators are required to predict the cost of testing, according to the selected testing strategy.

Software testability is a concept that characterizes the effort of testing. A variety of software metrics were proposed in the literature as indicators of software testability. All of them focus on measuring some software attributes that intend to make test difficult. A few studies have been carried out in order to validate these metrics. Some of these studies have controversial results about same metrics.

Our work in this thesis focuses on validating some testability metrics, and examining whether they could be really used as indicators of testability. Our approach in metrics validation considers both testability metrics and testing strategies, i.e. our methodology checks a specific metric against a specific testing criterion with respect to predefined hypotheses, and evaluates how much they are correlated. Additionally, we have defined new metrics which are a result of an adapting of some classical object-oriented metrics. The second part of our work concentrates on the testability antipatterns. The purpose of this part is checking some testability antipatterns and detecting at which point they are introduced during the software development phases.

Contents

1 Introduction **17**

2 Software Testability Notion **23**

2.1 Introduction 23
2.2 Testability from Hardware to Software 23
2.3 Software Testability Definitions 25
2.4 Binder's Testability Factors 26
2.5 Common Testability Factors 27
2.6 Conclusion 32

3 Testability Metrics and Metrics Validation **33**

3.1 Introduction 33
3.2 Scope-Oriented Testability Metrics 34
3.2.1 Methods 34
3.2.2 Classes 37
3.2.3 Stubs 37
3.2.4 Inheritance 40
3.3 General Complexity Metrics 40
3.3.1 Cohesion 41
3.3.2 Inheritance 43
3.3.3 Polymorphism 44
3.3.4 Encapsulation 44
3.4 Complexity Metrics Related to Observability and Controllability 45
3.4.1 Domain Testability 45
3.4.2 Observability and Controllability of Components 46
3.4.3 System Testability - STA 47
3.5 Testability Metrics Related to Error Likelihood 47

3.5.1 Propagation Infection Execution - PIE 47
3.5.2 Domain-Range Ratio - DRR and Visibility Component - VC . . . 48
3.6 Conclusion 50
3.7 Metrics Validation 50
3.7.1 Measure and Metric Definitions 52
3.7.2 Metric Construction Guide 53
3.7.3 Metrics Validation Approaches 54
3.7.4 Validation of Testability Metrics 56
3.7.5 Conclusion 57

4 Inheritance Testing: Adjusting Classical Testability Metrics 59
4.1 Introduction 59
4.2 Related Works 60
4.3 Inheritance Testing in the Context of Java Applications 61
4.3.1 Inheritance in Java 62
4.3.2 Dealing with Inheritance When Testing Java Systems 63
4.4 Cost of Inheritance Testing 65
4.4.1 Hypotheses 65
4.4.2 Cost of Testing 65
4.4.3 Cost of Testing to Achieve Method Coverage 66
4.4.4 Cost of Testing Strategies to Achieve Branch Coverage 67
4.5 Conclusion 69

5 Is DIT a Good Predictive Cost for Method Coverage Testing? 71
5.1 Introduction 71
5.2 Data Source 72
5.3 Data Analysis 74
5.3.1 Inherited Methods and DIT/DIT$_A$ 74
5.3.2 Defined Methods and DIT/DIT$_A$ 80
5.4 Conclusion 84

6 Is DIT a Good Predictive Cost for Branch Coverage Testing? 85
6.1 Introduction 85
6.2 Statistical analysis 86
6.2.1 Data Analysis 86
6.3 Limits of the Experiment 90

6.4 Conclusion 91

7 Predicting the Cost of JUnit Designing 93

7.1 Introduction 93
7.2 Basic Concepts in JUnit 94
7.3 Data Source 96
7.4 Data Analysis 97
7.5 Limit of the Work 100
7.6 Conclusion 100

8 Detecting Testability Antipatterns during the Development Process 103

8.1 Introduction 103
8.2 Testability Antipatterns 104
8.3 Simulation of the Development Phases 105
8.4 Data Analysis 107
8.5 Limits of the Work 111
8.6 Conclusions 111

Conclusion 112

A Basic Concepts in Statistics 115

A.1 Introduction 115
A.2 Types of Variables 116
A.3 Data Representation 116
A.3.1 Tables 117
A.3.2 Graphics 118
A.4 Correlation Coefficient 121
A.4.1 Pearson Correlation 122
A.4.2 Spearman's Correlation Coefficient 123
A.4.3 Kendall Rank Correlation Coefficient 126
A.4.4 Conclusion 126
A.5 Statistical Hypothesis Testing 127
A.6 Goodness of Fit Tests 129
A.6.1 Chi-square Test χ^2 130
A.6.2 Wilcoxon Tests 131
A.7 Regression Analysis 135
A.7.1 Simple Regression Model 135

A.7.2 Multiple Regression Model 137
A.8 Conclusion 138

B Metrics Calculator **139**

Glossary **143**

Bibliography **147**

List of Figures

2.1 OR gate - Controllability and observability 24
2.2 Testability Factors 26
2.3 Sys-B is more observable than Sys-A 28
2.4 Cycle 30

4.1 Sub tree of NanoXML inheritance tree 64

5.1 Number of inherited methods w.r.t. DIT 75
5.2 Number of inherited application methods w.r.t. DIT_A 77
5.3 Number of inherited (application) methods w.r.t. DIT (and DIT_A) 78
5.4 Number of defined methods w.r.t. DIT 81
5.5 Number of defined methods w.r.t. DIT_A 81
5.6 Number of defined methods w.r.t. DIT and DIT_A 83

6.1 MCC distribution with respect to DIT_A=0 to DIT_A=5 88
6.2 Scatter plot for WMC w.r.t. DIT_A for some applications 89
6.3 Scatter plot for WMH_A w.r.t. DIT_A for **SCOPE** and **Azureus** applications 90

7.1 Visualizing the relationship between WMC and WMC4Junit 98
7.2 Visualizing the relationship between WMC and WMC4Junit 99

8.1 Class interaction and Cycles 105
8.2 Number of cycles per application at each level 110

A.1 Scatter plot for data given in Table A.4 119
A.2 Scatter plot for data given in Table A.5 120
A.3 Frequency histogram for marks in Example 4 121
A.4 Simple regression for data given in Table A.20 137

B.1 Metrics Calculator Main Window 141

B.2 Adding Required Libraries 141
B.3 Result Window of Metrics Calculator 142

List of Tables

3.1 Testability Metrics and Factors 35
3.2 Metrics Tools 51
3.3 Effort estimation 51

4.1 Test or not to test the inherited methods 61
4.2 Appropriate metric w.r.t selected testing strategies 69

5.1 Data source 73
5.2 Distribution of the application classes w.r.t DIT_A and DIT 74
5.3 The Percentage of Number of Inherited Methods 76
5.4 Spearman's rank-order correlation between the number of methods and depth of total and application inheritance tree for each application 79
5.5 Average of Number of inherited (application) Methods w.r.t DIT and DIT_A 80
5.6 Number of classes w.r.t the ranges of number of inherited methods . . . 80
5.7 P-value of Wilcoxon test for the declared methods distribution at a fixed DIT 82
5.8 P-value of Wilcoxon test for the declared methods distribution at a fixed DIT_A 82

6.1 Spearman coefficients 86
6.2 P-value of Wilcoxon test for MCC at a fixed DIT_A 87

7.1 Data Source 96
7.2 Max and Min value for WMC and WMC4Junit 97
7.3 Spearman Correlations Analysis 100

8.1 Data source 105
8.2 The frequency of cycles of different sizes at the source code level 109
8.3 Max/Min cycle size and number of cycles at the source code level 109

8.4 Max/Min cycle size and number of cycles at the abstraction levels 110

A.1 Data Representation 117
A.2 Contingency Table For Example 1 118
A.3 Two Way Table 118
A.4 Sample data of Example 2 119
A.5 Distance/Gasoline Data - Example 3 120
A.6 Pearson Correlation for Example 6 124
A.7 Calculating Spearman Correlation For Example 7 125
A.8 Special Case for Spearman (Repeated values of X and/or Y) 125
A.9 Ranking and Calculating P value in Kendall Example 8 126
A.10 Correlation Methods 127
A.11 Chi-Square (male/ female) promoted example 130
A.12 Calculating Chi-Square for (male/female) promoted example 131
A.13 Partial table of critical values of Chi-square Distribution 131
A.14 Data for Example 10 132
A.15 Ranking data of Example 10 133
A.16 Marks of students - Example 11 133
A.17 Ranking all students marks Example 11 134
A.18 Sum and average of ranks of subgroup A and B 134
A.19 Price of gas per gallon 136
A.20 Calculating A and B values for simple regression - Example 12 136

Chapter 1

Introduction

Software development goes through several phases before arriving to a stable and reliable release. A general approach of software development starts by gathering information about the *requirements*, followed by *designing*, *implementing*, *testing* and *maintaining* phases. Each of these phases is essential and important in the development cycle.

Software testing is one of the most important and complex phases in the software development. It is considered as the only way to validate the software and the most used one. It aims at verifying the validity of the obtained results of a program execution by comparing them with the expected ones. The comparing process is called *test oracle*. If any difference between an obtained and expected result, an error is reported and a correction process begins.

A common definition of software testing is "Testing is the process of executing a software with the intent of finding errors" [89]. Another definition of software testing is given by Hetzel "Testing is any activity aimed at evaluating an attribute or capability of a program or system and determining that it meets its required results" [62]. The first definition does not focus only on the requirements of the software, it looks for any problem in the system, while the second one is more strict (from the requirements view), because it checks if the software conforms to its requirements.

Testing Techniques

The software testing activity consists of several successive steps. First, it begins with *unit testing*, which is a testing process that intends to test the software unit by unit[1]. The next step, is *integration testing*. It intends to test the combination of two or more

[1] A unit might be a method or a class.

units of the system together, in order to verify the functionality of these parts altogether. Finally, the *system testing* is testing the overall system with all its parts. It is necessary to achieve the three types of test, because errors that could appear during the integration testing for instance, might not appear through the unit testing.

Other types of testing can be carried out in order to achieve different goals such as *regression testing*, *acceptance testing*, *performance testing*, *security testing*, etc.

To test a software there is a variety of testing techniques (strategies) that could be applied according to the available data about the software under test. Testing strategies should focus especially on parts of the systems that are known to be fault proneness. Here are some testing strategies:

- *White-box*, *logic-driven* or *structural* testing is a technique that allows the tester to see and examine the internal structure of the software, and sometimes without considering the specification of the software. For that, this technique focuses on the details of the implementation, programming styles, control structures, methods, data design ...etc. There are several methodologies belong to White-box technique, such as *branch coverage, statement coverage, decision coverage, etc.*

- *Black-box*, *data driven* or *functional* testing is a technique that depends on the specification of the software. Neither the behavior nor the structure of the software is examined by the tester. Therefore, this technique focuses only on the inputs, then its outputs are verified for conformance to the specified objectives. There are also several methodologies belong to black-box technique, such as *boundary value analysis, equivalence partitions, etc.*

- *Gray-box* testing is a technique that uses a combination of black box testing and white box testing.

Testing Difficulties

Unfortunately, there is no way to guarantee that a system is free of bugs. Therefore all systems must be exercised (tested) as much as possible. That means spending more time and money on the software development. It often takes longer to test software than to implement it. Because reported errors during the testing add more time to the implementation phase, which in turn leads to a delay in the software release. In addition to that, the more complex the software is, the more time it needs to be tested. Therefore, the software testing is considered as a time consuming process. The difficulty of testing

increases with the absence of the software documentation or the poor one. Moreover, it is not possible to find all errors that could be found in software. Additionally, the software testing is a long process, since it involves several activities, such as *test plan*, *test analysis*, *test cases*, *test design*, etc. Furthermore, the selected testing strategy or the testing criterion which is required to be met, has also an influence on the cost of testing. Some testing strategies are cheaper than others, but they are less powerful.

In addition to these difficulties, other difficulties also may raise due to some programming language techniques, for instance, in object oriented languages some feature such as polymorphism and inheritance could significantly increase the difficulty of the test.

Therefore, the software testing is considered as a costly process in terms of time and money. It could represent 40% of the total development cost [17]. Roughly speaking, the cost of testing consists of two main parts, the cost of the machine time and the cost of personnel time to find and fix defects. Of course this implies the test planning and development in addition to the cost of running test cases and analyzing results. These costs are assumed to increase linearly with the amount of test effort spent to find and fix defects and length of the testing time [32, 98].

Software Testability

Several attempts were done to reduce the cost of testing. One approach is automating as much as possible. The automation of testing activities reduces significantly the cost of testing, and minimizes the human errors [8]. Another approach is "design for testing" or "design for testability" [21]. This approach relies on the observation that for a same problem, different solutions (with different designs) can be produced. Some of them are easier to test than others. "Design for testability" favors design solution(s) that would ease the test. These attempts and other tried to obtain systems which are easy to test. A system that is easy to test is called a *testable* system. In this context, a new notion has been introduced that is "Software Testability". Software testability is an adapted concept from the hardware systems that characterizes the testing difficulty level of the software. This proposition raised the question, how do we measure the testability? In other words, how to measure the difficulty level?

Metrics and Antipatterns

Different approaches have been in introduced in order to evaluate the software testability. All of them were proposed in order to obtain testable software or to measure the testability

of software. And all aim at detecting the locations which are difficult to test. One approach is based on defining testability metrics, which were proposed to estimate some software attribute, such as *number of lines of code*, *number of operators*, *cyclomatic complexity*, *etc.* Another approach is based on avoiding anti-testability patterns, which are known to increase the difficulty of testing.

A large number of metrics were proposed, but one common idea is behind them. The common idea is measuring an attribute that may cause or participate in increasing the difficulty of test, which in turn increases the cost of testing. One could classify these metrics into two main groups, one group is the metrics which help at evaluating the testability for non-object programs and one group for evaluating the testability for object programs. Since in this thesis, we are more interested in estimating the software testability for object oriented programs, our focus will be dedicated to study the metrics that are proposed to measure their testability.

Context and Motivation

Although there is a large set of metrics were defined to estimate the testability, a few works have been done to validate these propositions. Even though the goal is clear, it is not easy to validate how much it is accurate that these metrics are the required indicators of the testability. And how to validate the relationship between these proposed metrics and the testability. This is important issue for several reasons. First of all, a proposed metric may not be a *metric*, because of the absence of a metric definition, and for that a proposed metric might calculate a software attribute and not any more! Second, a proposed metric could be suitable for certain testing strategies, but it is not suitable for others. Therefore, it is important to decide which metric(s) one should use with respect to the testing method that will be applied. Third point is the impossibility of relying on one metric, either because each metric has its own limits, or because it could not characterize what is really required. And that raises the necessity of finding a suite of metrics, where each of them completes a part of the whole mission. Moreover, the proposed metrics have to be validated according to accurate, clear and well defined validation methods.

Therefore, we are interested in experimenting some metrics and antipatterns in order to validate them. Our research could be seen as a work with four axes:

A. Adjusting classical metrics: We have adjusted some classical metrics to be used in estimating the cost of inheritance testing. We focused on two main testing criteria,

i.e. *methods coverage* and *branch coverage* criteria. These adjusted metrics are dedicated to be used at the unit testing level or the integration testing level.

B. Validating adjusted metrics: We focused in this axis on validating the adjusted metrics with respect to related testing strategies. In order to validate them if they could be used to estimate the cost of testing. That was done by analyzing a set of `Java` applications and calculating the adjusted metrics, and then we studied empirically these metrics values to check if these metrics could be used as predictive metrics with respect to specific criteria.

C. Testability antipatterns analysis: In this part of our study, we concentrated on detecting some antipatterns. We are interested in discovering at which points they are introduced during the development process. Therefore, we studied a set of application classes at different levels of abstraction (that we extracted from application classes) and at source code level.

D. Developing "*Metrics Calculator*" tool: We developed a simple tool that can analyze `Java` applications in order to calculate the adjusted metrics and some other classical ones.

This work addresses two testing phases, i.e. *unit testing* and *integration testing*. More precisely, for the unit testing we focused on estimating the cost of inheritance testing which is often estimated by the depth of inheritance tree DIT. Therefore, we looked to know whether DIT could be used to predict different testing costs. We chose costs corresponding to classical testing approaches used in the industry. For integration testing phase, we focused on an analysis of some testability antipatterns. The approaches that we followed in this work to validate metrics or to analyze antipatterns are general. They could be applied to other metrics or antipatterns, and could be validated with respect to other testing strategies.

Thesis Outlines

The rest of this thesis is organized as following: *Chapter 2* defines the software testability concept, and presents the different factors that have an influence on the testability. *Chapter 3* summarizes a large set of testability metrics which were proposed in the literature and presents an overview about the different methodologies of metrics validation process. *Chapter 4* introduces our adaptation for the some metrics, and different hypotheses associated with the adapted metrics. *Chapter 5* and *6* present the analysis of

depth of inheritance tree as a predictive and effective cost of testing with respect to the made hypotheses. *Chapter 7* presents the analysis of detecting the testability antipatterns during at different levels of software development. *Chapter 8* an analysis to predict the cost JUnit by measuring cyclomatic complexity of the class. *Appendix A* summarizes basic concepts in statistics that we used during this study. *Appendix B* is dedicated to "Metrics Calculator" tool that we developed to calculate our adapted metrics and some other ones.

Chapter 2

Software Testability Notion

2.1 Introduction

Since software requires a high reliability level, and since the complexity of the software increases, then more time and money will be needed for the software testing in order to detect and identify *all* faults. Software testability is a system characteristic that could achieve a double goal. On one hand, it aims at reducing the cost of software testing. And on another hand, it helps to build more reliable software. The testability is an abstract concept that deals with the costs of testing. By improving the testability, it is expected that some part of these costs is being reduced, though not necessarily each individual cost. In other words, one testability improvement could affects at least two types of testing costs, such as *time, money, number of test cases, etc.*

The testability analysis is more related to the *testing* phase in the software development life-cycle, but considering this concept early in the development cycle could improve the testing process significantly. Adding the testability analysis to the development phases will increase the design time and costs, but it will reduce the costs of validation and maintenance.

In this chapter we discuss in more details the definitions of testability for software and the main factors that influence the testability.

2.2 Testability from Hardware to Software

Testability was first introduced for hardware systems. It was defined as a design characteristic that influences various costs associated with testing. A good testability level for

the hardware is necessary, because each produced piece (hardware component) must be tested in order to detect the faults that are introduced during the realization.

Several factors have an influence on the hardware testability. Controllability, observability and predictability are the three most important factors that determine the difficulty of deriving a test for a circuit. *Controllability* is the ability to establish a specific signal value at each node in a circuit by setting on the circuit's input. *Observability* is the ability to determine the signal value at any node in a circuit by controlling the circuit's inputs and observing its outputs. *Predictability* is the ability to obtain known output values in response to given input [4].

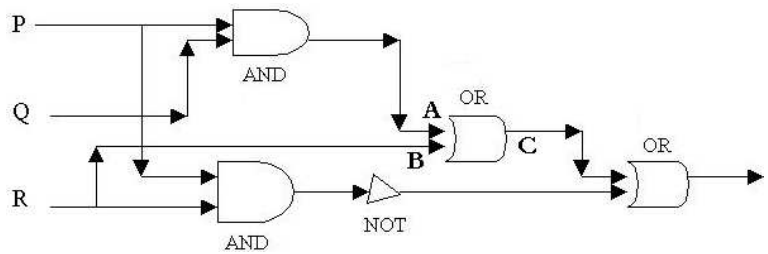


Figure 2.1: OR gate - Controllability and observability

For example, the `OR` gate (Figure 2.1) is controllable if one can *set* the input value of A and B. And it is observable, if one can *observe* the output value C.

The degree of a circuit's controllability and observability is often measured with respect to whether tests are generated randomly or deterministically using some algorithms [4]. Thus, testability is a relative measure of the effort or cost of testing a circuit. In general, it is based on the assumption that only primary inputs and primary outputs can be directly controlled and observed. Testability reflects the effort required to perform the main test operations of controlling internal signals from primary inputs and observing internal signal at primary outputs.

One approach to build a testable system is "*Design for testability*". The *Design For Testability (DFT) techniques* are design efforts specifically employed to ensure that a device is testable. In, general, DFT is used to reduce testing costs, enhance the quality (fault coverage) of tests, and hence reduce defect levels. It can also affect test length, tester memory, and test application time.

This notion has been applied to software for the same purpose in hardware, but this time for reducing the test generation costs and enhancing the quality in *software programs* [21]. Several testability definitions, for software, were proposed in the literature, in the following we state some of these definitions.

2.3 Software Testability Definitions

The software testability is considered as a system characteristic which estimates the effort of testing, or detecting the parts of software which are difficult to be tested. Several software testability definitions have been introduced. All of them are relevant to the cost of testing. Some definitions for instance consider the testability as a measure of the ability to select inputs that satisfy certain structural testing criteria, e.g. the ability to satisfy various code-based testing coverage's [123]. For example, if the goal is to select a set of inputs that execute every statement in the code at least once, and it is virtually impossible to find a set to do so, then the testability would be lower than if it was easy to create this set [119].

In the following we give some software testability definitions that were introduced in the literature:

- Binder defines testability as "Other things being equal, a more testable system will reduce the time and cost needed to meet reliability goals." A second definition by Binder is "A program's testability is a prediction of its ability to hide faults when the program is black-box tested with inputs selected randomly from a particular input distribution" [21].

- The `ISO` definition of testability is "Attributes of software that bear on the effort needed to validate the software product" [51].

- Bennitts defines "Testability is the ability to generate, evaluate, and apply tests to satisfy a number of predefined test objectives (for example fault coverage, fault isolation, runtime, time-to-profit) subject to the two fundamental constraints of time and money." [18].

- Voas has defined software testability as "The probability that a piece of software will fail on its next execution during testing (with a particular assumed input distribution) if the software includes a fault" [118].

- The `IEEE` standard glossary of software engineering terminology defined testability as "the degree to which a system or component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met" [3].

The ISO and Binder give a general definition of testability, while Voas focuses on the notion of probability of failing. The second definition by Binder focuses on the ability of hiding faults. On the other hand, the IEEE and Bennitts define the testability with respect to specific criteria. As we said previously, all these definitions look at the testability as a system feature relevant to the cost of testing. This characteristic is influenced by various factors, which we present in the following sections.

2.4 Binder's Testability Factors

Software testability for Binder is more than a characteristic. He considers it as a process which consists of several parts [21]. He identifies six primary testability factors, which could make test easier.

1. Representation
2. Implementation
3. Built-in Test
4. Test Suite
5. Test Tools
6. Test Process Capability

Figure 2.2 represents these six elements.

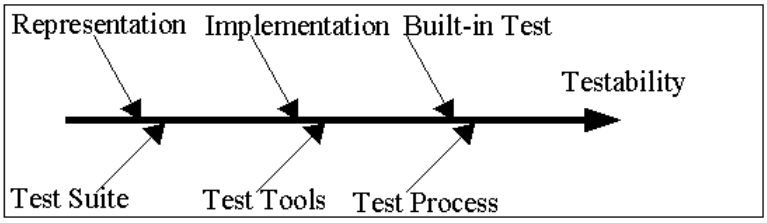


Figure 2.2: Testability Factors

Representation System representation starts from natural language statements about desired capabilities to detailed formal specifications. Various approaches could be used for representing object oriented systems, such as OOA and OOD. The system representation is considered as a factor of testability because it provides an explicit description of the

system behavior which aims at determining if a test has passed or not.

Implementation Respecting general principles of object oriented design or even of structural programming, could decrease various obstacles of testing. These principles such as complexity, inheritance and other, could be measured by some metrics.

Built-in Test - (BIT) provides explicit separation of test and application functionality. The built-in test aims at achieving an effective level of controllability and observability. For instance, BIT capabilities could include *set/reset* methods, *reporters* to observe the object state, and *assertions* to monitor some keys, such as pre/post conditions.

Test Suite is a collection of test cases and plans to use them. A test suite represents an asset acquired at considerable cost and should be treated accordingly.

Test Tools are needed to automate the test. Because without automation less testing will be done or greater test cost will be required to get a specific reliable goal. Test tools should have basic elements, such as runtime trace, static analyzer, script editor, code-base generator, input data generator, initializing the system, executing the test...etc.

Test Process Capability is the software process in which the testing is conducted. It focuses on the organizational structure, staff and resources supporting the testing process. Moreover, it implies different factors that can significantly improve the testing such as training, motivation, experience, etc.

2.5 Common Testability Factors

In the previous section we showed the factors that influence the software testability as they were proposed by Binder. In the following we present common factors that were widely considered as main keys of software testability. The factors presented here are more related to Binder's first and second factors, i.e. *representation* and *implementation*.

Factors of Hardware Origin

Some testability factors have been adapted from hardware origin, such as *Controllability*, *Observability*, *Information loss* and *Diagnosability*. The *Controllability* and *Observability*

have been considered as main factors that affect the testability.

Software controllability has been defined as "How easy it is to provide a program with the needed inputs, in terms of values, operations, and behaviors" [8]. According to Freedman, controllability is the ease of producing a specified output from a specified input [53]. While controllability, as defined by Pettichord, is the ability to apply inputs to software under test or place it in a specified state [96].

Software observability has been defined as "How easy it is to observe the behavior of a program in terms of its outputs, effects on the environment, and other hardware and software components" [8]. According to Freedman, observability refers to the ease of determining if specified inputs affect the outputs [53].

So a part of a system is *controllable* and *observable* if we can *activate* this part and *observe* its output respectively. Activating a part of system means it can be put in certain state, or under certain condition. The more one could observe, the more the part of system is observable Figure 2.3.

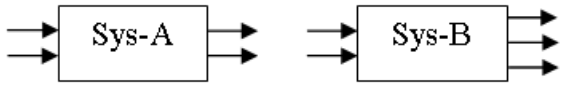


Figure 2.3: Sys-B is more observable than Sys-A

Example: The following class `SumExample` has an integer attribute b. The value of b is produced randomly in the constructor of the class. This way in generating the value of b, makes it somehow uncontrollable Listing. 2.1.

Listing 2.1: Uncontrollable Attribute Example

```
class SumExample
{
    int a,b,c;

    public SumExample()
    {
      Random r = new Random();
      b = r.nextInt(10);
    }

    void setA(int x)
    {
```

```
        a = x;
    }

    void sum()
    {
            c = a + b;
    }
}
```

A related factor to the observability has been introduced by Voas. It is *Information Loss*. The *Information Loss* is defined as a phenomenon that occurs during program execution that increases the likelihood that a fault will remain undetected [117]. It occurs when internal information computed by the program during execution is not communicated in the program's output. The Information loss could be explicit or implicit. It is explicit when the variables are not validated either during execution (by a self-test) or at execution termination as output. Therefore the inaccessibility to internal variable (local variable) leads to explicit information loss. A very simple example is a class with one (more) attribute(s), but it does not have any *get* methods that allows to monitor the value of this attribute(s).

While implicit loss information happens if two or more different input parameters produce same output. Let $f(t)$ be a function with one input parameter. Then if for two inputs x and y, while $x \neq y$, we get same output $z = f(x) = (y)$, we say the function f causes implicit information loss. e.g. $f(t) = (t)^2$.

The forth factor of hardware origin is *diagnosis*. It is defined as the minimal difference between the system and its model. The easiness of diagnostics can be seen as a criterion of testability [72]. There are some influencing factors between testability and diagnostics, such as *fuzziness*, *state characterization*, and *abstraction*. Each of these factors could have an influence on the testability [72].

Different metrics relevant to these four factors were proposed in the literature, next chapter shows a variety of such metrics.

Integration Level Factors

In the previous section, we showed some factors that came from hardware origin then they were adapted to be used with software systems. The factors presented here have an influence on the integration testing. *Dependency*, *Coupling* and *Cohesion* are three factors that address problems related to integration testing. The *dependency* between classes is

one of the elements that could make test difficult. For example, in unit testing we test every class in isolation from other classes. Therefore server[1] classes should be stubbed, which could be difficult in some cases, especially if it manifests in many different code locations [69]. The dependency is not necessarily between classes, it could be between class and a system resource; e.g. *data file, images, etc.* Also this kind of dependency could cause a difficulty in testing when these resources are unavailable or not suitable for testing and no other resources could be used instead.

A specific type of the dependency is *cycling.* A cycle is a loop in a directed graph, e.g. a module A uses module B, the module B uses C, and C uses A, see Figure 2.4. The more modules are in a cycle, the more difficult to test these modules [70], because a tester cannot test any of these modules without the presence of all other parts. For that a cycle must be tested as a group or decoupled with cycle-breaking stub [23].

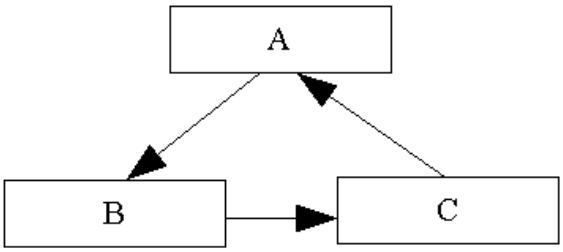


Figure 2.4: Cycle

Another common factor is *Coupling.* This factor provides summary information about the design and the structure of the software. It has a double effect on the testability: complexity and scope effects [21]. The coupling between two units measures the dependency relations between two units by reflecting the interconnections between units. Faults in one unit may affect the coupled unit [113]. Therefore, more interconnections are between units, more likelihood that a fault in one unit may affect others. Indeed, faults are found during integration testing exactly where coupling typically occurs [66]. Different levels of coupling have been ordered by Jones [94]. These levels have influence on a variety of software quality factors. And they are also used to estimate the complexity of software system design, in addition to number of faults [116].

The third common factor is *Cohesion.* It corresponds to the degree to which elements of a class belong together [41]. In other words, it refers to the internal consistency within parts of the design. Cohesion is considered as one of most important factors of quality

[1] A server or supplier class is a class that offers some services to other classes.

[86], and is considered to affect the complexity of testing [21]. Several types of cohesion were defined by Constantine in [113], such as *Coincidental, Logical, Temporal, etc.* These cohesion types are based on *how much a class actions are related to each other?, are the actions perform always a certain sequence of execution?, are the actions related by time?, etc.*

Binder has considered the lack of cohesion will lead to test more states in order to prove the absence of side effects among methods [21]. Therefore cohesion is considered as a factor of testability, and good testing results in better quality software [68, 8]

Object-Oriented Relevant Factors

Other factors that influence the testability are relevant to object-oriented paradigm, such as *Inheritance* and *Polymorphism.* The *Inheritance* is one of the main characteristic in object oriented programming. Inheritance is the ability of using features (data and functionality) defined already in another module (class) as if they were defined in the inheritor module (class). The inheritor module is called *child*, while the module that offers this ability is called *parent.* There are two types of inheritance:

1. "Single Inheritance" means a child can only have one direct parent.

2. "Multiple Inheritance" means a child could have more than one parent.

Inheritance is considered as a critical issue in object oriented testing. One reason behind this consideration is that an inheritance error may lead to subtle bugs, e.g. bugs due to overridden functionality [90], subclass bugs or side effects can cause failure in superclass methods. If a superclass is changed, all subclasses need to be tested...etc.

Another important factor related to object oriented programming is *Polymorphism.* The polymorphism allows instance variables to be bound to references of different types according to the structure of the inheritance hierarchy. Polymorphism is considered as a factor that makes testing process difficult. The compositional relationships of inheritance and aggregation, combined with the power of polymorphism, can make it harder to detect faults in the way components are integrated [7, 8]. Using polymorphism could also make code harder to understand and therefore fault-prone [100]. For instance the difficulty of understanding the different interactions between a message sender and a receiver including all possible bindings, another example is the problem caused by the difference between the pre/postconditions of an overriding method and the overridden one. In general, most of the problems that could happen due to inheritance could happen with polymorphism [22, 23, 40, 95].

2.6 Conclusion

In this chapter, we have presented several factors that have an influence on testability. Some factors from hardware origin, other are relevant to integration testing, etc. Each of these factors obligates us to take it into account during the process of testing software. Therefore we could imagine that testability is more than just a simple characteristic. It could be seen as a set of characteristics each of them makes a part of puzzle game. To complete this game one has to be sure that he has all the pieces (controllability, observability, no-information loss, etc). If any piece is missing, that means the software is not testable completely.

The question now can we estimate the testability of software? If yes, how could we measure it? Are all the factors, which are mentioned here, are really good indicators of testability? How to evaluate how much they are relevant to testability? In the following chapter, we will navigate through a large set of metrics of testability, which were proposed to evaluate these different factors.

Chapter 3

Testability Metrics and Metrics Validation

3.1 Introduction

In chapter 2, we have introduced different factors that could have some influences on testability. Each of these factors has its own characteristic that obligates us to take it into account during testability analysis process. Taking these factors in developer's consideration will make testing process easier, cheaper, and less time consuming. An important question is, how could we estimate each one of these factors? And how could we estimate the total testability of a system? Several approaches have been proposed to evaluate the testability of a system. Lots of metrics have been proposed to estimate testability.

In this chapter, we introduce a large set of metrics which have been proposed as testability indicator. Most of them can be evaluated only at the source code level. Even it is a bit late in the development cycle, evaluating testability at the source code level allows to identify low-testable parts of an application and to organize the testing work.

Despite of the availability of such set of metrics but not all of them have been validated. Later in this chapter, we will discuss two main approaches that were used to validate some metrics.

Testability metrics evaluate the scope and/or the complexity of testing [21]. The *scope* evaluates how many test cases have to be produced. The *complexity* indicates how much it is difficult to produce a test. For some cases, lots of test cases may be required, but it could be easy to identify them; or few tests may be required but it could be very difficult to design them.

Here we focus on source-code based metrics for *object-oriented* systems. We have collected more than 40 metrics that were proposed in the literature. They were declared to be testability relevant. The reason why there are so many metrics is that there are many strategies for test data selection/generation. Test data selection is based on the code or on the specification, at different phases (unit, integration and system testing), and with different purposes (among which achieving different coverage criteria). Moreover, there are different ways to understand what the *cost* of testing is. For instance, it can be the number of test cases, the size of test cases, the number of stubs, or the time spent to produce the tests.

Surprisingly, few metrics were really usable for testability evaluation. Some of them are hardly computable. Few are included in case tools. And few studies did formal or experimental validation to check if they are *really relevant* to software testability.

Table 3.1 associates some testability metrics that we present here, with the testability factors that we presented in the Chapter 2. Clearly, most of metrics are related to the complexity. Henderson identified the complexity as a factor affects the understandability, modifiability and testability [60]. The last three columns on the right of the Table 3.1 show at which level one could use these metrics to estimate the testability. Even one should notice that some metrics could be used at more than level, for instance *LOC* could be used to estimate to the number of lines per method, per class, all classes in packages, or the whole system. Additionally, certain metrics could be used to estimate the number of stubs that are needed to test method or class.

In the following, we first present scope-oriented metrics (section 3.2). Sections 3.3, 3.4 and 3.5 focus on complexity metrics. Section 3.4 is dedicated to observability and controllability related metrics, and metrics presented in section 3.5 are related to the probability to reveal the next error. Section 3.7 is dedicated to discuss some methodologies for validation the testability metrics.

3.2 Scope-Oriented Testability Metrics

3.2.1 Methods

Methods may be considered separately during unit-testing. Several sets of criteria defined for procedural programs can then be used. They are generally defined with respect to the control-flow graph or the data-flow graphs [89, 23, 55, 91, 52]. Classical control-flow graph

Metric \Factor	**Controllability**	**Observability**	**Information Loss**	**Dependency**	**Coupling**	**Cohesion**	**Inheritance**	**Polymorphism**	**Complexity**	**Scope**	**Estimating Stubs**	**Unit Testing**	**Integration Testing**	**System Testing**
LOC									✓			✓	✓	✓
Halstead									✓			✓	✓	✓
CC									✓	✓		✓		
DRR		✓	✓									✓		
Fan-in/Out				✓					✓	✓	✓	✓	✓	
PIE/EPIE		✓										✓		
Domain Testability	✓	✓							✓			✓		
WMC									✓	✓		✓		
NOC									✓	✓			✓	✓
DIT							✓			✓			✓	✓
NOM									✓			✓		
CBO				✓	✓				✓	✓	✓		✓	✓
RFC				✓						✓	✓		✓	
LCOM						✓						✓		
RCO		✓							✓			✓		
SCCr									✓			✓		
SCCp									✓			✓		
STA	✓	✓							✓			✓		
VC		✓	✓									✓		
MIF							✓							✓
PF								✓		✓		✓	✓	
EF									✓			✓		

Table 3.1: Testability Metrics and Factors

criteria are statement, branch or decision, condition, MC/DC (Modified Condition/Decision Coverage), path. For data-flow graph, classical criteria are all-paths, all-du-paths, all-uses, all-c-uses, all-defs-uses, all-p-uses.

In [11] and [130], two *families of metrics* have been proposed to evaluate the number of elements which has to be covered with respect to the control-flow and data-flow graph testing strategies : respectively all-paths, visit-each-loop-paths, simple paths, structured, branches, statements, and p-uses, defs, uses, d-u-paths and dominating paths. By definition, those metrics predict the scope of the associated testing strategies, i.e. the minimum number of required tests to reach the coverage criteria.

Similarly, the *Cyclomatic Complexity (CC)*, named also McCabe's complexity, [80, 81, 127] is equal to the number of decision statements (or individual conditions) plus one. Mathematical analysis has shown that CC gives the recommended number of tests needed to test every decision point in a program [127]. Thus, it predicts the scope of the branch coverage testing strategy.

In [12], two flow graph metrics were defined axiomatically: *Number of Trails* metric which represents the number of unique simple paths through a flowgraph (path with no repeated nodes), and *Mask [k=2]* metric, which stands for "MAximal Set of K-walks", where a k-walk is a walk through a flowgraph that visits no node of the flowgraph more than k times. Mask reflects a sequence of increasingly exhaustive loop-testing strategies. These two metrics measure the structural complexity of the code. One of the main benefits of defining these testability metrics axiomatically is that flowgraphs can be measured easily and efficiently with tools such as QUALMS [12].

Number of lines of code (LOC)[1] is one of the simplest metrics [5]. *LOC* has no standard definition. One definition of a *Line of Code* is given in [39]: "A Line of code is any line of program text that is not a comment or blank line, regardless of the number of statements or fragments of statements on the line. This is specifically includes all lines containing program headers, declaration, and executable and non-executable statements". *LOC* is considered an ambiguous metric, because there are many ways in which this metric could be calculated.

For example, consider the following code:

```
for (i=0; i<10; ++i) write("hello"); /* just print !!*/
```

The ambiguity appears here, what is the value of *LOC* in the previous code? Is it 1, 2 or 3 lines? Due to this ambiguity of calculation, *LOC* was not considered as reliable

[1]There are several acronyms for *LOC* such as *SLOC*, *KLOC* and *KSLOC*. The letter K stands for *Kilo* indicates that the scale is in thousands. The letter S stands for *Source*.

metric. However, since the bigger the program is, the more errors will be, so it could be used to estimate the cost of testing.

3.2.2 Classes

Since it is often difficult to test the class methods independently, they are considered as a minor granularity for the unit testing. Therefore, we focus in this section on the testability metrics that are relevant to the classes. The class is an essential concept in object-oriented programming. It groups attributes of an object and the operations on these attributes. It can also be considered during unit-testing.

Weighted Methods per Class (WMC) metric belongs to the Chidamber and Kemerer object-oriented metrics suite [36]. For a class C with methods $M_1, M_2...M_n$, let $c_1, c_2, .., c_n$ be the complexity of these methods.

$$WMC = \sum_{i=1}^{i=n} c_i$$

Complexity was deliberately not defined in the original paper in order to allow a general application of this metric. If all method complexities are considered to be unity, then $\mathrm{WMC}_1 = n$ represents the number of methods. WMC_1 can be used to evaluate the number of test cases to achieve the method coverage [21]. This criterion is one of the simplest object-oriented code-coverage coverage criteria (corresponding to function coverage). It requires each method to be executed at least once. A similar metric to WMC_1 is *Number Of Methods (NOM)* representing explicitly the number of methods of a class [21].

When cyclomatic complexity is used as complexity measure to compute WMC (WMC_{CC}), it evaluates the number of the test cases required to test all the methods of the class to reach the decision coverage criteria. Moreover, the more methods in a class, the greater potential impact on children, since children inherit all (public/protected) methods defined in the class.

3.2.3 Stubs

Stubs may be required during unit or integration testing, they are also useful to achieve more unitary test. A stub is an extra routine that is provided by the tester, to imitate another part of the system. Different metrics can be used to estimate the number of required stubs to carry out a unit testing. In the following we present some of these metrics:

The *Fan Out (FOUT)* of method A is the number of local flows from method A plus the number of data structures which A updates [61]. In other words FOUT estimates the number of *methods* to be stubbed, to carry out a unit testing of method A

Binder proposes to use *Response For Class (RFC)* metric to evaluate how many stubs has to be produced for unit testing at class level [21]. RFC is one of the Chidamber and Kemerer metrics suite [36]. It is defined as the count of the methods defined in a class, in addition to the methods that are called directly by a method of this class. The number of methods to be stubbed corresponds to the number of calls of methods defined outside the class/subsystem under test. Since RFC counts also the number of methods defined within the class, RFC only provides an approximation of the number of *methods* to be stubbed, to carry out a unit testing of a class.

$$RFC = |RS| \quad where \quad RS = \{M\} \cup_{\forall i} \{R_i\}$$

where $\{R_i\}$ is the set of methods called by method i, and $\{M\}$ is a set of all methods in the class.

A class is coupled to another class if one of them acts on the other, i.e. a method of a class uses methods or instance variables of the other. There are several definitions for the coupling between objects. One of them was proposed by Chidamber and Kemerer in [36]: *Coupling Between Objects (CBO)* of a class is the number of other classes to which it is coupled. CBO can be used to evaluate the number of *classes* to be stubbed, in order to carry out a unit testing of a class [21].
Eight different levels of coupling were ordered by Jones [94]. These levels influence the different quality factors of a unit such as reusability, maintainability, understandability...etc. An extension to these levels was made to become 12 levels [67] (see also Chapter 2). These levels evaluate the software system designs complexity, and a relationship between these levels and the number of faults [116].

Other coupling metrics were also proposed to measure the coupling, such as *Coupling Factor (COF)* which is based on estimating the number of relations between a client class and supplier(server) class [47, 42]. The larger number of relations between classes, the more complexity will be. In addition to that the larger number of relations will limit the understandability. COF is given by the following formula:

$$COF = \frac{\sum_{i=1}^{TC}[\sum_{j=1}^{TC} is_client(C_i, C_j)]}{TC^2 - TC}$$

where TC represents the total number of classes in the system under study, and *is_client* refers to the relation between two classes C_i, C_j, and is defined as following:

$$is_client(C_i, C_j) = \left\{ \begin{array}{c} 1 \text{ iff } (C_i \Rightarrow C_j) \wedge (C_i \neq C_j) \\ 0 \text{ otherwise} \end{array} \right\}$$

where $C_i \Rightarrow C_j$ means the class C_i contains at least one reference to the class C_j.

Other coupling metrics focus on the relationship between classes in different packages, such as *Afferent/Efferent Coupling (Ca & Ce)* [99]. The Ca metric is defined as the number of classes from other packages that depend on the classes within a package, while the Ce metric is defined as the number of classes from other packages that the classes within the package depend upon.

Certain coupling metrics were proposed to measure the complexity of message passing among classes, such as *Message Passing Coupling (MPC)* [26]. MPC counts only calls of methods of other classes, and do not consider the calls for the methods defined inside the class itself.

Class Fan Out (Class FOUT) represents the number of classes on which a given class depends [105]. This metric could be used to estimate the number of classes to be stubbed. When some strategies are used to schedule intelligently the test of the different classes in order to decrease the number of subs required to be produced, CBO or Class FOUT may be not relevant. In [74], authors show that 400 stubs would be required to test 122 classes individually (without any strategy) against 8 when an optimal test order is used. In [70], S. Jungmayr proposes a testability metric in the context of static dependencies within object-oriented systems. A dependency of a component A on a component B exists if A requires B to compile or to function correctly. If A inherits from B or if it uses method(s) or attribute(s) of B then A depends on B. Dependency relation is transitive. A dependency graph may contain cycles. Such cycles can be broken by removing some dependencies. The set of removed dependencies are called *Feedback Dependency Set*.

Number of Stubs needed to Break Cycles (NSBC) evaluates the number of stubs required to be built with an integration testing strategy.

$$NSBC = |C_{Fb}|$$

where C is the set of all components, C_{Fb} a feedback component set ($C_{Fb} \subset D$), and D is the set of all dependencies. Finding a smallest feedback component set is NP-complete. To identify a small feedback component set S. Jungmayr proposed an algorithm based on both Tarjan and greedy algorithms [70].

3.2.4 Inheritance

Inheritance is one of the main features of object-oriented programming paradigm. Since it has been demonstrated that inheritance may be abused in many ways [9], one may expect that several testing criteria would have been dedicated to inheritance testing. Surprisingly, few testing methods/criteria deal with inheritance [59, 95, 34, 50, 37]. Most of them restrict testing to validate changes in the inherited features (methods and attributes).

In [23], all inherited methods should be retested. In [59, 95, 34, 50], it is suggested to re-test only (modified) inherited features (attributes or methods). Since, the number of inherited methods is considered to be generally proportional to the *Depth of Inheritance Tree (DIT)* [36, 21], DIT is considered as a way to estimate the testing effort. But it does *not* provide an estimation of *how many* test cases have to be produced [108]. A class with a small inheritance tree may have more inherited methods than a class with a large inheritance tree.

When a class inherits the same property of an ancestor via multiple paths in the hierarchy, there is repeated inheritance. Repeated inheritance is not allowed in several object-oriented languages, such as Java, C#, and VB .Net. Overuse of repeated inheritance increases software error [37]. That's why C.-M. Chung *et al.* propose a testing method to search for errors caused by the repeated inheritance. Each class concerned by repeated inheritance has to be tested in the context of its inheritance sub-trees.
Authors introduce the notion of URI (Unit Repeated Inheritance) as a specific inheritance sub-graph, where the number of nodes equals the number of edges ($G = (V, E)$ where $|V| = |E|$). Repeated inheritance tree can be decomposed as a set of basic $URIs$. For an inheritance tree, let t be the number of terminal classes (classes with no out-edges) and U_i be the set of URI related to the terminal class i. The *complexity of the repeated inheritance* is defined as $| \cup_{i=1}^{t} U_i|$. This complexity corresponds to the number of repeated inheritance sub-tree to be examined. Here again, it does not predict how many test cases have to be produced for each of them.

3.3 General Complexity Metrics

In the previous section, we discussed scope metrics which could be used to estimate the number of test cases. In this section, we present complexity metrics which could be used

to estimate the difficulty of producing a test. As we have seen previously, the Cyclomatic Complexity (CC) gives the recommended number of tests needed to test every decision point in a program [127]. Thus, it predicts the scope of the branch coverage testing strategy. It is also considered as an indication of the complexity of testing. Indeed, a method with a CC greater than 50 is considered to be untestable [1].
By extension, WMC_{CC} could also be considered as an indication how difficult it is to test the class. However, it could be difficult to interpret: WMC_{CC} indicates that a class A with 60 very simple methods ($c_i = 1$) will require more testing than a class B with one method having a complexity of 50. When analyzing CC for each method, B will be more difficult to test (since a method with a complexity of 50 is supposed to be untestable) and 60 simple methods will require 60 simple tests. The authors in [85] proposes to use WMC_{CC} with three other metrics: *Mean Method Complexity (MMC)*, *Standard Deviation Method Complexity (SDMC)* and *Number of Trivial Methods (NTM)*. Using the 4 metrics as opposed to WMC_{CC} alone allows distinguishing between certain types of classes and therefore interprets the results accordingly.

Another complexity metrics were proposed by Halstead [84]. Halstead's metrics are based on the number of operators and operands per module. They evaluate the length, difficulty, effort and required time of programming a program.

3.3.1 Cohesion

Cohesion is an extension to the definition of *similarity*, which was proposed by Bunge [36]. The similarity $\sigma()$ of two things is the intersection of the sets of properties of the two things.

$$\sigma(X, Y) = P(X) \cap P(Y)$$

Cohesion has been considered to be a factor influence the cost of testing [36]. Having lack of cohesion in a class C means that this class has many functionalities which are not related to each other, and as a result the class C will behave in less predictable way than a class of a fewer functionalities.

Lack of Cohesion of a Method (LCOM) is the 6th measure of the Chidamber and Kemerer metrics suite [36]. LCOM measures the degree of similarity between the class's methods. The more methods share the same attributes, the larger is cohesion. LCOM

[1] `http://www.sei.cmu.edu/str/descriptions/cyclomatic_body.html`

metric has been redefined several times [25]. We give here the definition[1] of LCOM4, given by Hitz and Montazer. Let $G_x = (V, E)$ be a graph representing calls between class methods. V is the set of vertices, which represents the method names. E is the set of edges. $E = \{(m, n) \in V \times V | \exists i \in I_x : (m \text{ accesses } i) \wedge (n \text{ accesses } i) \wedge (m \text{ calls } n) \wedge (n \text{ calls } m)\}$ where I_x is the set of the attributes. $LCOM_4 = |E|$
LCOM has been proposed as a testability metric by Binder in [21], because high LCOM means more states that have to be tested to prove the absence of side effect among the methods.

Tight Class Cohesion (TCC) is the percentage of pairs of public methods of the class which are directly connected [20].

$$TCC = NDC/NP \quad where \quad NP = N * (N - 1)/2$$

where N the number of methods, NP represents the number of possible connections and NDC number of indirect connections.

Loose Class Cohesion (LCC) considers the pairs connected directly or indirectly [20].

$$LCC = (NDC + NIC)/NP$$

where NIC represents the number of indirect connections.

Information CoHesion (ICH) was proposed by Lee *et al.* [75]. It is based on the information flow. It considers the cohesion of a method m implemented in a class c as the number of the invocations to other non-inherited methods of class c, weighted by the number of the parameters of the invoked methods. Cohesion between calling and called methods is stronger if the latter has more parameters more information is passed.

Most Cohesive Component (MCC) is introduced in [106] as the most cohesive form if each class's method has interaction with all of instance variables (special methods such as *set* and *get* are excluded).

Ratio of Cohesive Interactions (RCI) is another cohesion metric that was defined by Briand *et al.* [27]. It is based on a concept called "Data declaration interaction $DD -$

[1] *LCOM1, LCOM2, etc* definitions differ from each other in that some definitions consider direct/indirect access to an attribute, include/exclude *set* and *get* methods, include/exclude inherited features, etc.

interaction". A $DD-interaction$ exists between two attributes, when any change/use of one of these two attributes will require a change/use of the another attribute. When the interaction happens between an attribute of the class and a local variable of a method, this interaction is called $DM - interaction$. The RCI for a class is given by:

$$RCI(c) = \frac{|CI(c)|}{Max(c)}$$

where CI represents all interactions $DD - interactions$ and $DM - interactions$, and $Max(c)$ is the maximal set of all possible cohesive interactions of the software part, that could be obtained by connecting every data declaration to every other data declaration and method with which it could interact.

3.3.2 Inheritance

Inheritance influences the scope of testing as previously indicated. Binder indicates that it also influences the complexity of testing. Like DIT, several other measures were proposed for inheritance tree complexity evaluation. *Class Fan-In (FIN)* is the number of parent classes of a subclass. It is applied only in multiple inheritance languages. A high Class FIN value increases the possibility of incorrect bindings [21].

Number Of Children (NOC) is the number of classes that inherit directly from a class [36]. It is the number of immediate subclasses subordinated to a class in the class hierarchy. It indicates how many derived classes will be affected by some modification in the parent class. So if a modification in the parent class affects the derived classes, it is required to retest the methods in the children. NOC is considered as a scope metric and a complexity metric. Because the higher NOC is, the more tests should be produced, and the greater the number of children, the greater likelihood of improper abstraction of the parent class [36].

In [42], Abreu *et al.* propose a set of measures called MOOD set, for object-oriented design quality evaluation. This set includes *Method Inheritance Factor (MIF)* and *Attribute Inheritance Factor (AIF)*. They compute the number of the inherited methods/attributes (M_i) in all classes divided by the number of available (inherited & defined) methods/attributes (M_a) for all classes. Those factors are defined to evaluate the inheritance complexity of the whole system. They could be adapted to evaluate MIF and the AIF for

each class (instead of for all classes together).

$$MIF = \frac{\sum_{j=1}^{TC} M_i(C_j)}{\sum_{j=1}^{TC} M_a(C_j)}$$

where TC is the total number of classes in the system under study. Having $MIF = 0$ indicates that either there are no inheritance, or all inherited methods are overridden.

3.3.3 Polymorphism

It is an important feature in object-oriented programming, defined as a characteristic of being able to have multiple forms. To test software that uses polymorphism, one should test all possible bindings of receiver classes and target methods at different call points. In [77], Lin and Huang define a *testability of polymorphism* metric in inheritance hierarchy, based on the descendant paths.
Also a *polymorphism factor (PF or POF)* was defined to represent the actual number of possible different polymorphic situations [44]. A very high POF value (above 10%) will reduce the benefits of polymorphism. The PF metric is given by:

$$PF = \frac{\sum_{j=1}^{TC} M_o(C_j)}{\sum_{j=1}^{TC} [M_n(C_j) * DC(C_j)]}$$

where TC is the total number of classes in the system. M_o is the number of overriding methods in the class C_j, and DC is the number of derived classes of the class C_j.

Other metrics have been proposed in [21] to measure the complexity as a result of the polymorphism such as *percent of Dynamic calls (DYN)*, *percent of non-overloaded calls (OVR)*, *number of yo-yo paths visible to CUT (Bounce-C)*, and *number of yo-yo paths in SUT (Bounce-S)*. Although these metrics have not been evaluated, they were proposed as an indicator of the opportunities for faults.

3.3.4 Encapsulation

Encapsulation is defined as *"a software development technique that consists of isolating a system function or a set of data and operations on those data within a module and providing precise specifications for the module"* [3]. Using the encapsulation makes private data inaccessible directly during testing. An *Encapsulation Factor (EF)* metric for a class was proposed to measure the encapsulation level of a class. EF was defined as a function of two parameters *privacy* and *unity*, where privacy is based on the private data members (the data visibility), and unity is the cohesion between the attributes and methods [102].

3.4 Complexity Metrics Related to Observability and Controllability

As said previously, testability was originally defined for hardware components. For hardware component context, testability is often characterized through observability and controllability. To test a component, one must be able to control its inputs and observe its outputs. When a component is embedded, its controllability and observability can be decreased. This partly depends on the architectural design. Observability and controllability were adapted to software systems based on components. In the following we present 4 metrics of testability based on the observability and controllability.

3.4.1 Domain Testability

Two *Domain Testability* metrics have been proposed by Freedman for non object-oriented software and based on these concepts [53]. A procedure F is observable if distinct outputs are generated from distinct inputs, and an expression procedure F is controllable if the set of all evaluations of F covers all values in the range (co-domain of F). Observability and controllability *extensions* are the input/output variables required to achieve the definitions of observability and controllability.
The *observability (Ob)* and the *controllability (Ct)* according to the domain size of the added inputs/outputs are given as following:

$$Ob = log_2(|ID_1| * .... * |ID_n|)$$

$$Ct = log_2(|OD_1| * .... * |OD_m|)$$

The Ob and Ct represent the number of extra binary inputs required to convert a function/procedure into observable and controllable form respectively, where ID_i is the domain of the ith added input and OD_j is the domain of the jth added output.
A limit of these metrics is that the cardinality of certain types cannot be calculated (i.e. vector, array, object, etc). Moreover, one important feature in object-oriented paradigm is that objects preserve states. So attributes could be used as implicit input/output for a method. Therefore, the notion of observability and controllability extensions have to be adapted to object-oriented features.

3.4.2 Observability and Controllability of Components

In [126], five metrics were proposed as reusability measure of software component. Two of them are related to observability and one to controllability. The *Rate of Component Observability (RCO)* represents the percentage of readable attributes in all fields implemented within the Façade class of a component. If the value of RCO is in [0.17, 0.42] (confidence interval) the height of the observability is supposed to be appropriate [126]. It is a variation of *percent Public And Protected (PAP)* defined in [21].

$$RCO(c) = P_r(c)/A(c) \;\; when(A(c) > 0), \;\; 0 \; otherwise$$

where $P_r(c)$ is the number of readable properties in the component c, and $A(c)$ is the number of fields in c's Façade class.

The *Self-Completeness of Component's Return value (SCCr)* is the percentage of business methods without any return value in all business methods implemented within a component c. It can be extended to non business methods and allows detecting the absence of return values, which decreases observability. However, observability is not limited to the existence of return values.

$$SCCr(c) = B_v(c)/B(c) \;\; when(B(c) > 0), \;\; 1 \; otherwise$$

where $B_v(c)$ is the number of business methods without return value in c, and $B(c)$ the number of all business methods in c.

The *Self completeness of component's parameter (SCCp)* is the percentage of business methods without any parameters in all business methods implemented within a component c. SCCp can be extended to other application layers (i.e. non business methods). Since it can be easier to test methods which have no input parameter (see Category and Partition [92]), classes with high SCCp may be easier to test than some with low SCCp. One limit of SCCp is that it is independent from the difficulty to test methods with parameters. Let A be a class of several methods each of them has one parameter, ($SCCp(A) = 0$). Let B be a class of two methods, one with several methods and the other without any parameter, ($SCCp(B) > 0$). A is probably easier to test than B. Moreover $SCCp$ does not capture the situation where methods have implicit parameters (attributes).

$$SCCp(c) = B_p(c)/B(c) \;\; when(B(c) > 0), \;\; 1 \; otherwise$$

where B_p is the number of business methods without parameters in c.

3.4.3 System Testability - STA

System Testability (STA) of an object-oriented software is defined as mathematical mean of all the objects testability obtained in the system [125].

$$STA = \frac{1}{n} \sum_{j=1}^{m} OTA_j$$

where OTA_j is the object testability defined as the product of its test controllability and observability. Controllability of an object is the ability to control all the basic control structures within the object. Observability of an object is the ability to indicate the values of any variables within the path(s) sensitised by the current test case.
STA is the average of its component testability values. This definition does not take into account the architecture of the system, which is quite unusual. It is usually expected that the architecture influences the observability and the controllability.

3.5 Testability Metrics Related to Error Likelihood

Testing aims at finding errors [89]. The easier it is to find errors, the easier testing is. One definition of software testability given by A. Bertolino is "the probability that a test of the program on an input drawn from a specified probability distribution of the input is rejected, given a specified oracle and given that the program is faulty" [19]. In other words, it is the probability to observe an error at the next execution if there is a fault in the program. Several metrics are related to this definition are presented hereafter.

3.5.1 Propagation Infection Execution - PIE

Propagation Infection Execution (PIE) analysis is a well-known metric proposed for testability. It has been proposed by Voas [120, 122]. PIE measure aims at computing the *sensitivity* of individual locations in a program.
The sensitivity of a program location refers to the minimum likelihood that a fault at that location will produce incorrect output, under a specified input distribution. This measure has its origin in the RELAY model used for error detection [87]. It relies on the fact that for discovering a fault in a program, three conditions should be met. First the statement that contains the fault should be executed. Secondly the state of the variable should be infected. And at last, it should be propagated to an output.

Testability of a software statement $T(s) = Re(s) * Ri(s) * Rp(s)$ where $Re(s)$ is the probability of the statement execution, $Ri(s)$ the probability of internal state infection and $Rp(s)$ the probability of error propagation.
PIE analysis determines the probability of each fault to be revealed, and requires sophisticated calculations. It does not cover object-oriented features such as encapsulation, inheritance, polymorphism, etc.

In [79], authors propose an adaptation of PIE analysis to compute the *testability of a class t(C)*. Based on t(C), the testability of the class is derived with respect to different factors: cohesion, communication and inheritance. The $t(C)$ is defined as the sum of the testability of the methods $t(M_i)$, where the testability of a method is defined as the product of the execution rate of a method $E(M)$ and the propagation rate of a method $P(M)$.

$EPIE$ an extension to PIE analysis was proposed to reduce the number of analyzed locations [65]. This method depends on two elements: implicit information loss and statement dependency. $EPIE$ method could be divided into three steps:

1. *Break a program into blocks*: a block consists of sequential statements. Loops and conditional statement should be dispatched into another block.

2. *Divide block into groups*: each block is transfered into dependency graphs. Each node has to be checked to see whether it has implicit information loss. In case of finding information loss in some node, it has to be analyzed alone, finally the dependency graph is broken into sub-graphs (groups).

3. *Mark target statements*: In each group one statement will be marked. A marked statement is a location that will be executed first in the group.

Since $EPIE$ marks only one statement in each group, the number of analyzed locations is decreased. The reason of choosing the first statement in each group, is that in propagation analysis, wrong data status generated by statement may be cancelled by the following statements, in such situation the probability of propagation will be decreased.

3.5.2 Domain-Range Ratio - DRR and Visibility Component - VC

In the previous section we focused on the metrics that are based on the sensitivity analysis. Since it is complicated to calculate the sensitivity metrics, such as PIE, a simplification

of sensitivity analysis has been proposed such as *Domain-Range Ratio (DRR)*. DRR of a specification is the ratio between the cardinality of the domain to the cardinality of the range. DRR depends only on the number of values in the domain and the range, not on the relative probabilities that individual elements may appear in these sets [121].
For a program, when the input domain is larger than the output domain, information about the internal states may not be communicated in the outputs. This information may have included evidence that internal states were incorrect. This loss suggests a lower testability.
For instance, let us consider two functions $F(x) = \text{x } mod\ 2$ and $G(x) = 2 * x$. F has a domain on all R and a range on {0,1}. For such function, it is difficult to predict if the result is really the one that corresponds exactly to the given input: it has an unlimited and infinite set of inputs produce the same result (output). G has a domain on all R, and the range also on all R. Any two different inputs will produce two different outputs. Detecting an error for $G(x)$ is easier than detecting an error on $F(x)$.
DRR evaluates how much an application is supposed to hide faults. It is a priori information, which can be considered as a rough approximation of testability.

Three categories of DRR were introduced in [121]:

1. *Variable Domain/Fixed Range - VDFR*: this category includes function(s) which its domain is an infinite set of values, and its range is over a finite set.

2. *Variable Domain/Variable Range - VDVR*: this category includes function(s) which its domain and range are infinite set of values.

3. *Fixed Domain/Fixed Range - FDFR*: this category includes function(s) which its domain and range are finite set of values.

Depending on these categories, the exhaustive testing is theoretically possible on the $FDFR$ category, therefore the functions belong to $FDFR$ category are considered to be the highest testability. Functions belong to $VDVR$ are considered to be less testable than the ones of $FDFR$ category and more testable than the ones of $VDFR$ category. $VDFR$ is the category with the lowest testability, because it represents the largest amount of internal state collapse among these categories.

J. McGregor and S. Srinivas proposed an extension of DRR for object-oriented programs called *Visibility Component (VC)* [82]. VC is the cardinality of possible outputs (including the exceptions) divided by the cardinality of possible inputs. Two types of

parameters for a method are considered: implicit and explicit ones. The *Visibility Component* is given by the formula:

$$VC = \frac{cardinality\ of\ possible\ outputs}{cardinality\ of\ possible\ inputs}$$

Like domain testability measures, cardinality of certain types cannot be calculated (i.e. vector, array, object, etc) by DRR or VC. This is major drawback makes impossible to use DRR or VC practically.

3.6 Conclusion

Up to this point, we showed a large set of metrics that were proposed in the literature as testability metrics. Some tools also are available for certain metrics, see Table 3.2 and Table 3.3. In front of such large set of metrics one could be asked, are these metrics accurate? Do they *really* estimate the testability of method, class and system? Actually, very few works have been done to validate some metrics, even there is no clear guide about *when* or *when not* to use some metric. Moreover, certain metrics are not calculable. Therefore, we are interested in validating some metrics that have controversial impressions in different studies.

Different validation methodologies for the metrics were defined. We look at them in some details in the following sections. Some validation methodologies depend on statistical concepts that we describe in details in the Appendix A.

3.7 Metrics Validation

As we mentioned previously, most of proposed metrics have not been proved in formal way nor empirically. Moreover, some of them do not satisfy the metric definition. Therefore, a lot of metrics have been criticized in the literature. In [71], C. Kaner and W.P. Bond question the *construct validity* of software engineering metrics, as several other authors before them [73, 10, 104]. The general question is "*how do we know that we are measuring the attribute that we think we are measuring?*" And especially for the metrics presented before, how do we know that they really correspond to testability evaluation?

Several frameworks and methodologies were proposed in the literature for measurement and metrics validation [2, 73, 104, 83, 58, 128]. Among them, the standard IEEE 1061 [2] proposes a methodology to define metrics where the fifth step is Validation.

	Tool name	Reference	Calculated metrics
1	CKJM	http://www.spinellis.gr/sw/ckjm/	WMC, DIT, NOC, CBO, RFC, LCOM,...
2	NetBeans Metrics Module	http://metrics.netbeans.org/	WMC, CBO, RFC, DIT, NOC,...
3	Eclipse Metrics plug-in	http://metrics.sourceforge.net/	LCOM, WMC, CC, DIT,...
4	JStyle	http://www.mmsindia.com/jstyle.html	RFC, LCOM, Fan-In, Fan-Out, WMC, DIT
5	Understand for Java	http://www.scitools.com	LCOM, DIT, CBO, NOC, RFC,...

Table 3.2: Metrics Tools

	Metrics	Effort estimation	Available tools
1	Control-flow graph metric suite [11]	# of TC for control-flow graph coverage	no
2	Data-flow graph metric suite [130]	# of TC for data-flow graph coverage	no
3	CC [80, 81, 127]	# of TC for branch coverage (decision point)	yes
4	WMC_1 [36, 21]	# of TC to achieve the method coverage	yes
5	NOM [21]	# of TC to achieve the method coverage	yes
6	WMC_{CC} [36, 21]	# of TC to achieve the branch coverage at the class level	yes
7	RFC [36, 21]	# of methods to be stubbed	yes
8	CBO [36, 21]	# of classes to be stubbed (no integration strategy)	yes
9	FOUT [61]	# of methods to be stubbed (no integration strategy)	yes
10	NSBC [70]	# of classes to be stubbed (with an integration strategy)	no
11	DIT [36, 21]	proportional to # of TC (when inheritance is re-tested)	yes
12	Complexity of the repeated inheritance [37]	# of repeated inheritance sub-tree to examine	no

Table 3.3: Effort estimation

Predictive metric results are compared to the direct metric results to determine whether the predictive metrics accurately measure their associated quality factor. Moreover, the standard lays out six validation criteria: correlation, tracking, consistency, predictability, discriminative power and reliability. Another draft standard for testability and diagnosability characteristics and metrics was developed by (D&MC), a subcommittee of IEEE [111]. The purpose of this standard was to provide formal and unambiguous definition of testability, where this definition should be independent of specific test, diagnosis process and system under test.

In the following sections, we introduce the formal metric definition, different methods of metric validation and the properties of metric.

3.7.1 Measure and Metric Definitions

In simple words, measuring means associating a number, that represents an attribute, with a physical object. This association is called *mapping* or *function* in mathematics. Formally, a measure is defined as following:

Let A be a set of physical or empirical objects. Let B be a set of formal objects, such as numbers. A ***measure*** *μ is defined to be a one-to-one mapping $\mu : A \rightarrow B$.*

Since a measure is a one-to-one mapping, it guarantees that every object has one and only one measure [49, 48, 71]. Some uses the term "metric" instead of "measure" which is not the same. The "measure" is the common term, while "metric" is a special case of it. A "metric" is a way of measuring the distance between two entities. Formally a metric is defined as following:

Let A be a set of objects, let $\Re$ be the set of real numbers, and let m be a measure $m : A \rightarrow \Re$. m is called ***metric*** *if and only if it satisfies the following three properties:*

1- Identity of indiscernibles:	$m(x, y) = 0$ for $x = y$
2- Symmetry:	$m(x, y) = m(y, x)\ \forall\ x, y$
3- Subadditivity:	$m(x, z) \leq m(x, y) + m(y, z)\ \forall\ x, y, z$

Although the term *metric* is often used when defining a testability measure, but it is not the accurate term, since no testability measure has considered the three properties of a *metric*. Additionally, the term *metric* is used to define the distance between two entities in a set, which is the not the case of many testability metrics.

3.7.2 Metric Construction Guide

Several attempts have been done to describe necessary properties of software metrics [128, 45, 101, 63, 97]. For instance, authors in [45, 101, 104] suggested certain number of characteristics such as, a metric must be intuitive, computable easily, calculated in reliable manner, robust, able to capture real differences between the measured values, related to a specific quality factor and return useful information that could be used to improve the design.

Weyuker proposed nine properties that were supposed to allow identifying the weaknesses of a measure in concrete way [128]. We list briefly here these properties:

- **Property 1:** A measure that rates *all* programs as equally complex is not a measure. There are distinct programs P, Q, such as $\exists P$, $\exists Q$ where $|P| \neq |Q|$.

- **Property 2:** Let c be a nonnegative number. Then there are only finitely many programs of complexity c.

- **Property 3:** There are distinct programs P, Q, such as $|P| = |Q|$.

- **Property 4:** $\exists P$, $\exists Q$ where $P \equiv Q$ and $|P| \neq |Q|$. That is to say, having two programs that produce same results does not necessitate to have same complexity.

- **Property 5:** $\forall P$, $\forall Q$ where $|P| \leq |P;Q|$ and $|Q| \leq |P;Q|$. This property states that the complexity of any program P is less than the complexity of P combined with any other program.

- **Property 6:** $\exists P$, $\exists Q$, $\exists R$ where $|P| = |Q|$ and $|P;R| \neq |Q;R|$. With the assumption that P, R use some common variables, but Q, R are disjoint.

- **Property 7:** There are program bodies P, Q, such that Q is formed by permuting the order of the statement of P and $|P| \neq |Q|$

- **Property 8:** If P is renaming of Q, then $|P| = |Q|$. That is to say, the name of a program does not influence its complexity.

- **Property 9:** $\exists P$, $\exists Q$ where $|P| + |Q| < |P;Q|$. It is possible to have two programs where the sum of their complexities is smaller than the complexity of the combination of these two programs.

These properties are considered as axioms that metrics should satisfy, but several metrics does not satisfy these properties. For example, cyclomatic number do not satisfy property 7, because the cyclomatic complexity of a program is completely independent of the placement. Another example is the *effort* measure proposed by Halstead, which does not satisfy the property 5, complete proof is given in [128].

It is important to notice that these properties are necessary but are not sufficient to determine if a metric is a *metric* [49]. One reason for that is, these properties are dedicated to complexity metrics, which *may* not be appropriate to be used with other metrics. Therefore, the axioms should be defined without regarding a specific metric or specific context. Generally, a good software metric has to possess well defined axioms, that should be met by any proposed metric, and it should be defined with respecting to a formal metric/measure definition.

3.7.3 Metrics Validation Approaches

In general, there are two main approaches for validating the software metrics, *standard* and *empirical* approaches. The standard approach relies on validating the metrics formally, while the empirical approach looks for a correlation between a quality factor and the metric value. In the following we discuss both approaches.

Standard Approach

The goal of metrics validation is identifying both product and process metrics that could predict specified quality factor values. A *process metric* is a metric that is used to measure characteristics of the methods, techniques, and tools employed in developing, implementing, and maintaining the software system. A *product metric* is a metric used to measure the characteristics of any intermediate or final product of the software development process.

The IEEE has proposed a methodology for software quality metrics. This methodology has been defined as "a systematic approach to establishing quality requirements and identifying, implementing, analyzing, and validating the process and product software quality metrics for a software system" [2].

Measuring quality factor values could be done at a certain point in the life cycle, either early or lately in the development process. If it cannot be applied early then predictive measuring could be done.

The IEEE methodology consists of the following steps:

1. Establish software quality requirements.
2. Identify software quality metrics.
3. Implement the software quality metrics.
4. Analyze the software metrics results.
5. Validate the software quality metrics.

The validation of metrics refers to validating the relationship between a set of metrics and a quality factor for a given application. In other words, the validation does not mean a universal validation of the metrics for all applications [2].

The IEEE proposed to apply a validity criterion to validate a predictive metric. This criterion consists of the following points:

- Square of the linear correlation coefficient.
- Rank correlation coefficient.
- Prediction error.
- Confidence level.
- Success rate.

These points imply several characteristics: ***Correlation*** which is used to estimate whether there is a sufficiently strong linear association between a quality factor and a metric. If there is not such association then it is not feasible to use that metric. This characteristic is often used during the empirical validation of metrics. ***Tracking*** this characteristic ensures that for any change in the quality factor a corresponding change will occur to the metric value. ***Consistency*** this criterion assesses whether there is a consistency between the ranks of quality factors values and the ranks of metrics values for same quality factors. ***Predictability*** that means the metric is capable to estimate the value of the quality factor with the required accuracy. ***Discriminative power*** this characteristic reflects that the metric could be used to identify critical values, in order to separate high quality software components from low ones. ***Reliability*** that means that a metric has passed a validity test over a sufficient number or percentage of applications, therefore there will be a confidence this metric could predict an accurate value.

In the Appendix A, we discuss in more details the points of this criterion, and we introduce more concepts which are related to the validation process.

Empirical Validation

A common approach for validating a metric empirically begins by *collecting* a large set of data, followed by *calculating* the required metric(s), then stating *research hypotheses* about the measured attributes, finally applying some *correlation methods* to determine if there is a correlation between the measured data and the metric(s) [60].

Generally, the empirical validation does not facilitate or resolve the problem of validating. In other words it does not help to find the desired metrics, which could be used as a predictive metrics. One reason behind that is the contradictions between the results of these studies, for instance, some studies show the importance for certain metrics, such as *DIT, CC*, other studies dismiss this importance.

Different studies to validate the software metrics empirically have been carried out. Most of them tried to establish the relation between object oriented metrics and fault-proneness of classes [13, 30, 114, 28, 31, 46, 131, 57, 132].

3.7.4 Validation of Testability Metrics

Several formal and informal methods have been used to validate software metrics. This severity reflects the difficulty of this process, in addition to the ambiguity of the characteristic that we want to predict. The absence of both the formal testability definition and formal metric definition causes this difficulty of validating. Furthermore, the difficulty of the validation of the testability relies on the fact that the *effort* to test is subjective and depends on the point of view. Therefore, the validation of testability metrics is considered as a hard work, especially for the complexity metrics.

For instance, let us consider the *Rate of Component Observability* (RCO) [126] and the *percent Public And Protected* (PAP) [21]. They both represent the percentage of readable attributes, but are used differently with respect to the testability analysis: if a high RCO is supposed to ease testing because observability is increased [126], a high PAP is supposed to increase the difficulty of testing, because there are more opportunities for side effects. A challenge is thus to propose some definition(s) for what could be the effort to test.

The validation of scope metrics seems to be easier. By definition, scope metrics predict the number of tests to produce with respect to a testing approach. To validate those metrics empirically, one can compare the expected number of test cases given by the metric and the effective one when applying the testing method. The possible differences

can be due to the impossibility to reach the associated coverage criteria because of the infeasible execution paths for instance. If empirically, a scope metric prediction is very different from the effective number of test cases, it may be error-prone and thus should not be used.

We highlight here some studies have been done to validate certain object oriented metrics, which are considered to be testability metrics. In [88], Mouchawrab *et al.* introduced a generic measurement framework for object oriented software testability, which is based on a theory expressed as a set of operational hypotheses. They identified 20 hypotheses, 4 of these hypotheses are related to inheritance concept (i.e. inherited features, operation rule...etc), which could be identified, more or less, as a result of the metric DIT. Other hypotheses are related to number of paths, coupling, dependency cycles, cohesion, complexity of pre/post conditions and invariants, etc. This study focuses on the analysis and design phases of software development. However, these hypotheses summarize different testability metrics, and could be seen as an extension to the work of Binder presented in [21].

Other studies focus on classical metrics, such as WMC, LCOM, DIT, RFC, CBO, etc. But they were not correlated to testability. In [33], Bruntink *et al.* have evaluated the correlation between a set of object oriented source code metrics (among which DIT) and their capabilities to predict the effort needed for testing, expressed as dLOCC (*Lines Of Code for Class*) and dNOTC (*Number of Test Cases*). Somewhat surprisingly, DIT was not correlated to dNOTC. This was explained by the fact that inherited methods were probably not systematically re-tested. However, if all inherited methods are re-tested, it was expected that the number of test cases should increase with respect to DIT, as it has been shown after in [108].

3.7.5 Conclusion

The validation of software metrics is a critical process. And it is more critical and important for the validation of testability metrics. Formal and informal attempts have been achieved in order to find the desired metric. Despite this variety of the methodologies of validation, both the formal ones and the informal ones, it did not help to find the appropriate testability metrics. Therefore, we are interested in validating different testability metrics with respect to specific testing criteria, and we carried out a validation process that is similar to the standard approaches presented in this chapter, and supported by an empirical analysis.

In this chapter, we introduced some metrics validation approaches. We presented certain points that we considered as important factors to be considered during the metric validation process. In the following chapters, we show our empirical validation of adapted metrics with respect to specified testing strategies.

Chapter 4

Inheritance Testing: Adjusting Classical Testability Metrics

4.1 Introduction

In chapter 3, we presented a large set of metrics which were proposed to estimate some factors that influence the software testability. Some of these metrics are dedicated to evaluate different object-oriented features, such as polymorphism, encapsulation, inheritance, etc.

In this chapter, we focus on a main feature in object-oriented programming, that is *inheritance*. As we have seen in chapter 3, several metrics were defined to characterize the inheritance tree complexity, among which *Number Of Root classes (NOR)*, *Fan In (FIN)*, *Number Of Children (NOC)* and *Depth of Inheritance Tree (DIT)*.

Depth of Inheritance Tree (DIT) is our main interest. It belongs to the Chidamber and Kemerer metrics suite [36], which has been proposed in early nineties. The depth of a class within the inheritance hierarchy is the maximum length from the class node to the root of the tree, measured by the number of ancestor classes. The deeper a class within the hierarchy, the greater the number of methods it is likely to inherit [36, 59].

A variety of testing strategies have been dedicated to inheritance testing. Basically, two types of strategies were proposed. It is either suggested to test all new methods and to retest all inherited methods (for instance, see Binder's book [23].) Or, it is suggested restricting testing to validate only changes in the inherited features (methods and attributes) [50, 34, 59].

In both cases, the number of methods to be tested is supposed to be proportional to DIT [36, 21]. That is why DIT is generally considered as a way to estimate the

testing effort. Additionally, DIT is often considered to be predictive metric, because the inheritance tree is often defined during the design phase. Moreover, as we said previously, DIT is one of the metrics that have controversial impression.

Examining some tests of Java applications, led us to refine the definition of DIT, which in turn led to introduce an adaptation for some classical testability metrics that aim at estimating the cost of inheritance testing. This chapter is dedicated to show our adaptation of these metrics.

In the following, section 2 shows some related works, section 3 describes the inheritance testing in the context of object-oriented systems. Section 4 is dedicated to the cost of inheritance testing.

4.2 Related Works

Lots of work focused on the understanding of software systems in terms of objects and their properties. For that, several set of metrics have been proposed such as the Chidamber and Kemerer set [36, 43]. The understanding of these metrics and especially their relevance in improving the outcomes of software developments led us to a large set of researches focusing on the validation of the different metrics [29, 13, 30, 114, 28, 31, 46, 131, 57, 132, 33, 44, 124, 24, 93].

We focused during this research on the object oriented paradigm, and we concentrated on the understanding of how the Depth of Inheritance Tree (DIT) can be used to predict the cost of unit testing. Since it has been demonstrated that inheritance may be abused in many ways [9], several testing strategies have been dedicated to inheritance testing. While other studies [13, 30, 114, 28, 31, 46, 131, 57, 132] did not consider DIT as a significant factor of fault-proneness of classes. A class B that inherits class A, means it will inherit all public properties and methods defined in A. Therefore, more test cases will be required to test the methods not only the defined ones in the class B, but also the inherited ones from A. Additionally, it will be required to test the different possible interactions between the parent class A and the child one B. Consequently, some authors have suggested to consider the inheritance complexity during the testing prediction. While other considered that it is not necessary to retest already inherited tested features because it is already tested, or they did not find any signification relation between the inheritance and the fault-proneness of classes.

In [33], M. Bruntink *et al.* have evaluated the correlation between a set of OO source code metrics (among which DIT) and their capabilities to predict the effort needed for

testing. In this study, test cases were all written as JUnit class [1]. The effort for testing was expressed as the number of Lines Of Code for the JUnit Class (dLOCC) and the Number of Test Cases (dNOTC). Five large open-source applications were studied. They were available with their JUnit tests. One difficulty here was the fact that the method used for the production of the test was not under control. Coverage criteria were different from one application to another, and sometimes different from one package to another from the same application. The result that the testing effort was not related to the DIT is explained by the fact that inherited methods were probably not systematically re-tested.

4.3 Inheritance Testing in the Context of Java Applications

The cost of inheritance testing is influenced by the selected testing strategies. Some of these strategies require to test all (defined and inherited) methods of the class, while other strategies require to test only the inherited features that have been changed.

`class parentClass`
`{`
`  public void method1()`
`  {`
`  // do something...`
`  }`
 	
`  public int method2()`
`  {`
`  // do something else ...`
`  }`
`}`
	`class childClass`
	`  extends parentClass`
	`{`
	`  public void method3()`
	`  {`
	`  // do whatever ...`
	`  }`
	
	`  public void method4()`
	`  {`
	`  // do ...`
	`  }`
	`}`

Table 4.1: Test or not to test the inherited methods

For instance, to test the class *childClass* (Table 4.1) we could choose one of two strategies, the first one is to test all methods; that means testing *method1, method2, method3* and *method4*. Or to choose the second strategy that is to test just the defined methods in the class *childClass*; that means only we have to test *method3, method4*.

One could ask, why should one retest an inherited method of a class in the following contexts:

- if there is no modifications on inherited method?
- if it does not make any reference to modified attributes?
- if it does not call modified or overridden methods?

That might be true in some contexts, but not always. The inherited methods from a super class were tested in the context of the super class, which might not be the same context of the sub class(es). In other words, testing some methods in some specific context does not guarantee that they still correct in other contexts.

4.3.1 Inheritance in Java

Object-oriented programming allows classes to inherit commonly used state and behavior from other classes. Inheritance is the capability of a class to use the properties and methods of another class while adding its own functionality. The deeper a class within the hierarchy, the greater the number of methods it is likely to inherit [36, 59]. In the Java programming language, each class is allowed to have one direct superclass, and each superclass has the potential for an unlimited number of subclasses. The `Object` class is the superclass of any class written in Java. All other classes are subclasses of the `Object` class. The `Object` class includes 11 methods among which `clone()`, `finalize()`, `toString()`, and `equals(Object src)`.

A superclass contains elements and properties common to all of the subclasses. Often, a superclass will be set up as an abstract class which does not allow objects of its prototype to be created. Abstract methods are methods with no implementation. Subclass of an abstract class must provide the implementation of the abstract methods or should be declared as an abstract class.
Java does not allow multiple inheritance for classes (i.e. a subclass being the extension of more than one superclass). To tie elements of different classes together Java uses an interface. Interfaces are similar to abstract classes but all methods are abstract and all properties are static final. Interfaces can be inherited (i.e. you can have a sub-interface). It is not possible to create object from interfaces, and thus they cannot be tested. Moreover, interfaces do not inherit of the `Object` class.

4.3.2 Dealing with Inheritance When Testing Java Systems

Inheritance mechanism leads one to suppose that well-known super classes can be reused with confidence in subclasses. However, Binder underlines that *even if a superclass has been shown to be reliable, it is not guarantee equal reliability its subclasses*: reliable superclass methods can fail in the context of the subclasses [23].
For testing classes in the context of inheritance, Binder distinguishes two cases [23]. Either the class under test inherits from a trusted development environment (Java Development Kit - JDK - in our case), or it inherits from a class that does not warrant this level of confidence. In the first case, testing inherited methods is a problem of integration. Otherwise, it is suggested to retest the superclass methods in the context of the subclass.

Therefore, it is important differentiate between the complete inheritance tree (including the JDK classes) and the inheritance tree restricted to the application. This distinguishing will lead to more accurate estimation of the cost of the inheritance testing. In the following we formalize these two definitions of DIT.

Depth of Inheritance Tree

An object-oriented system consists of a set of classes, C. For every class $c \in C$ we have $Ancestors(c) \subset C$ the set of classes from which c inherits either directly or indirectly. The following formal definition of Depth of Inheritance Tree was given in [33].

$$DIT(c) = |Ancestors(c)|$$

This definition of DIT relies on the assumption that the considered object-oriented programming language allows each class to have at most one parent class. Only then the number of ancestors of c will correspond to the depth of c in the inheritance tree. Java complies with this requirement.
This definition of DIT corresponds to the complete inheritance tree, i.e. it includes standard Java classes JDK. Let us introduce DIT_A as the Depth of Inheritance Tree restricted to the Application classes. Let JC be a set of all standard Java classes and AC be a set of all application classes, $JC \cap AC = \emptyset$.

$$\text{If } c \in AC,\ DIT_A(c) = |Ancestors(c) \backslash JC|$$

For instance, let us consider NanoXML, a small XML parser for Java [103]. Figure 4.1 shows a sub-part of NanoXML inheritance tree. The classes `XMLValidationException` and `XMLException` are both application classes. The three remaining classes are standard

Java classes. The DIT for `XMLValidationException` class is 4 if we take into account the complete inheritance tree. While it is only 1 if we take into account only the application's classes.

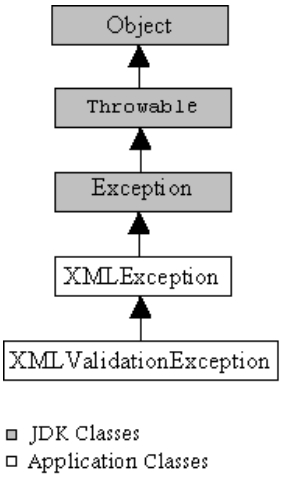


Figure 4.1: Sub tree of NanoXML inheritance tree

The differentiation between the entire inheritance tree and the application one, leads to redefine certain number of metrics.
Let us now consider the methods. Let $M_D(c)$ be the set of methods that c *newly declares*, and let $M_{In}(c)$ be the set of methods that c *inherits*. Let $M(c)$ denote the set of (defined and inherited) methods of c:

$$M(c) = M_D(c) \cup M_{In}(c)$$

The metric NOM represents the number of methods defined in class the c (see Chapter 3). It is given by the following formula:

$$NOM = |M_D(c)|$$

Let us consider NOH to be the number of inherited methods:

$$NOH = |M_{In}|$$

Those definitions include standard Java classes. To restrict the study to the application methods, we define $M_{IA}(c)$ as the set of methods that c inherits, and which are defined in $Ancestors(c) \backslash JC$. We also consider the set of application methods of c as:

$$M_A(c) = M_D(c) \cup M_{IA}(c)$$

Since the NOH represents all inherited methods that belong to standard classes and application classes, we introduce here an adapted definition to consider only the inherited methods from application classes:

$$NOH_A = |M_{IA}|$$

the NOH_A represents the number of inherited methods from application classes.
In the following, we discuss the cost of inheritance testing with respect to the metrics NOM, NOH and NOH$_A$.

4.4 Cost of Inheritance Testing

Since we are interesting in validating testability metrics, we focus in the following sections on one of the controversial metrics, that is *Depth of Inheritance Tree DIT*. We want to know if DIT could be used to as a predictive metric of testability.

4.4.1 Hypotheses

According to the previous discussion, let us call T_s the set of strategies that focus only on testing the methods that are newly defined in a class, and let us call T_t the set of strategies that suggest that all the inherited methods should be re-tested. Then the two hypotheses are:

A. For testing strategies that do not consider inheritance (T_s), the **cost of testing** is not influenced by DIT.

B. For testing strategies that consider inheritance (T_t), the **cost of testing** is influenced by DIT.

4.4.2 Cost of Testing

The difficulty of estimating the cost of testing, resides in specifying what the cost of testing is. Since there are several testing methods, as we have shown in Chapter 1, there are also numerous methods of estimating the cost of testing.

In the following, we have chosen to estimate the cost of testing in terms of the *number of required tests* for achieving the coverage of a given criterion. Estimating the cost in such way, is compliant with the testability definitions given by Bennitts, IEEE and others [18, 3].

Applying this choice allows to refine the precedent hypotheses, to become:

$\dot{A}$. For testing strategies that do not consider inheritance (T_s), the number of required tests for covering a given criterion is not influenced by DIT.

$\dot{B}$. For testing strategies that consider inheritance (T_t), the number of required tests for covering a given criterion is influenced by DIT.

Clearly, the number of required tests for covering a criterion depends on the criterion, which itself depends upon the selected testing strategy.

In the following, we study two coverage criteria, the first criterion is *method coverage* (studied in chapter 5), the second one is *branch coverage* (studied in chapter 6).

4.4.3 Cost of Testing to Achieve Method Coverage

For achieving the methods coverage: (1) With considering the testing strategies T_s, the cost of testing is estimated by the number of methods to test which is given by the simple metric *Number Of Methods (NOM)*. (2) But when considering the testing strategies T_t, then either we test all the inherited methods by the class, in this case the number of methods to be tested is given by NOH, or we test only inherited methods from application classes, and in this case the number of methods to be tested is given by NOH_A.

Therefore, we refine the previous hypotheses as following:

$\ddot{A}$. For testing strategies that do not consider inheritance (T_s), the number of defined methods in a class (NOM) is not influenced by DIT.

$\ddot{B}$. For testing strategies that consider inheritance (T_t), the number of inherited methods in a class is influenced by DIT.

With respect to the definition 4.3.2, we can rewrite the previous hypotheses as following:

$\ddot{A}$-1 For testing strategies that <u>do not consider inheritance</u> (T_s), the number of defined methods in a class <u>is not influenced by</u> DIT.

$\ddot{A}$-2 For testing strategies that <u>do not consider inheritance</u> (T_s), the number of defined methods in a class <u>is not influenced by</u> DIT_A.

$\ddot{B}$-1 For testing strategies that <u>consider inheritance</u> (T_t), the number of inherited methods in a class <u>is influenced by</u> DIT.

$\ddot{B}$-2 For testing strategies that consider inheritance (T_t), the number of inherited methods in a class is influenced by DIT_A.

Since the number of methods to be tested is considered as a metric to estimate the predictive cost, we chose also another metric that could be used to estimate the effective cost of testing, that allows to have more accurate estimation for the cost of testing. In the following, we discuss the previous hypotheses but from the branch coverage point of view.

4.4.4 Cost of Testing Strategies to Achieve Branch Coverage

The second selected testing strategy is the branch coverage. The branch coverage (or Decision Coverage) exercises each possible branch in flow control structures. In other words, branch coverage means that each branch direction must be traversed at least once [89]. It is accepted as a "minimum mandatory testing requirement" in many industries for the unit test phase of software [17].

It requires that each branch alternative in a program is exercised at least once [89]. It is used to evaluate testing thoroughness regardless of the strategy followed to select the test cases and to decide whether testing can be stopped. The number of tests needed to achieve branch coverage can be regarded as a measure of the "minimum mandatory effort" to test a given program.

CC was defined by McCabe [80]. It is built on the number of basis paths through the program. Cyclomatic Complexity CC is derived from a flowgraph and is mathematically computed using graph theory. More simply stated, it is found by determining the number of decision statements in a program and is calculated as: $CC = \textit{number of decision statements} + 1$. CC is a lower bound for the number of test cases which are necessary to achieve a complete branch coverage.

CC is also considered as an indication of the difficulty of testing (complexity in the sense of Binder in [21]). The following set of threshold values is generally given[1]. When CC is between 1-10, the method or program is considered to be simple to test. With a CC between 11-20, the method is more complex with a moderate risk. Between 21 and 50, the method is complex to test with a high risk. If CC is greater than 50, the method is supposed to be untestable.

Achieving the branch coverage of the class A means achieving the branch coverage of the methods of A. A lower bound for the number of test cases that are necessary to

[1] http://www.sei.cmu.edu/str/descriptions/cyclomatic_body.html

achieve a complete branch coverage of the class can be approximated by the sum of the CC numbers of the methods. This can be computed with the Weighted Methods Per Class (WMC), proposed by Chidamber and Kemerer [36] The original definition is the following.

Consider a class C, with methods $M_1, ..., M_n$ that are *defined* in the class. Let c_i be the complexity of the method M_i. Then WMC(C) $= \sum_{i=1}^{n} c_i$. The original definition of WMC does not fix the definition of the c_i. It can be either equal to one or equal to CC. Here, we consider this last definition.

To predict the cost of testing strategies that requires achieving branch coverage of methods defined and inherited in the class, we introduce Weighted Methods for the Hierarchy of the Class (WMH). We define WMH(C) to be the sum of the WMC of each class in the hierarchy of C.

$$WMH(C) = WMC(C) + \sum_{C_i \in Ancestors(C)} WMC(C_i)$$

This definition is general: the whole hierarchy is considered. As we see previously, in the context of Java, this definition includes standard classes. Therefore, we also introduce Weighted Methods for the Hierarchy of the Class restricted to the application classes (WMH$_A$).

$$WMH_A(C) = WMC(C) + \sum_{c_i \in Ancestors(C) \setminus JC} WMC(c_i)$$

Hypotheses

According to the previous discussion, we could state different hypotheses which are related to evaluate the difficulty to test a class. In the following hypotheses, we consider the number of test cases that are necessary to achieve the branch coverage of the class methods is expressed by Weighted Methods by Class (WMC), Weighted Methods for the Hierarchy (WMH$_A$) and the difficulty of testing is expressed by the Maximum Cyclomatic Complexity of the methods within the class (MCC).

We define MCC of a class as following: Let C be a class with methods $M_1, ..., M_n$ that are *defined* in the class. Let c_i be the cyclomatic complexity of the method M_i.

$$MCC(C) = \max_{i=1}^{n} c_i$$

A. For branch testing strategies that <u>do not consider inheritance</u> (T_s testing strategies), the difficulty to produce the test (estimated by MCC) is not correlated with DIT$_A$.

Metric	Testing strategy
NOM	Testing only defined methods
NOH	Testing all inherited methods (including standard)
NOH_A	Testing inherited methods from application only
WMC	Branch coverage for class (defined methods)
WMH	Branch coverage for complete class tree
WMH_A	Branch coverage for application class tree

Table 4.2: Appropriate metric w.r.t selected testing strategies

B. For branch testing strategies that do not consider inheritance (T_s testing strategies), the number of tests to be produced (estimated by WMC) is not correlated with DIT_A.

C. For branch testing strategies that consider inheritance (T_t testing strategies), the number of tests to be produced (estimated by WMH_A) is correlated with DIT_A.

4.5 Conclusion

The cost of testing is varying according to the selected testing strategy. Different software metrics were proposed to estimate the cost of testing. Choosing the *right* metric to estimate the testing cost is considered an important issue. In this chapter, we presented an adaptation to three metrics NOM, DIT and WMC, which are the total number of inherited methods *NOH*, the number of inherited methods from application classes NOH_A, the depth of inheritance tree restricted to the application DIT_A, the Weighted Methods for the Hierarchy *WMH* and the Weighted Methods for the Hierarchy restricted to the application WMH_A. Table 4.2 summarizes the testing strategies and the associated metrics that we considered in this chapter.

These metrics are related to some testing strategies that consider the inherited methods during the testing process. In the next chapter, we present an experiment that we carried out on a set of open source Java applications in order to validate the presented hypotheses with respect to the adapted metrics presented in this chapter.

Chapter 5

Is DIT a Good Predictive Cost for Method Coverage Testing?

5.1 Introduction

In this chapter, we carry out an experiment that allows us to analyze our adapted metrics presented in the previous chapter and to validate certain hypotheses about estimating the cost of testing.

In the following, we analyze the **cost of testing** by the estimating the number of methods to be tested. Method coverage criterion is one of the weakest coverage criteria. However, since this criterion could be calculated early in the development life cycle, it could be used as a cost estimation of the testing strategies at the specification level. Contrary to other criteria, such as *instructions* and *branch* coverage which could be used only after the implementation phase, which is considered to be late to evaluate the software testability.

We remind that we focus on two main groups of testing strategies: (1) A testing strategies in T_s will require to test $|M_D(c)|$ methods for class c. (2) A testing strategies in T_t will require to test $|M(c)|$ or $|M_A(c)|$ methods for class c, depending if one considers the complete inheritance tree or the inheritance tree restricted to the application classes. Therefore, we stated some hypotheses that we would like to check them in this chapter.

$\ddot{A}$-1 For testing strategies that <u>do not consider inheritance</u> (T_s), the number of defined methods in a class <u>is not influenced by</u> DIT.

$\ddot{A}$-2 For testing strategies that <u>do not consider inheritance</u> (T_s), the number of defined methods in a class <u>is not influenced by</u> DIT_A.

$\ddot{B}$-1 For testing strategies that consider inheritance (T_t), the number of inherited methods in a class is influenced by DIT.

$\ddot{B}$-2 For testing strategies that consider inheritance (T_t), the number of inherited methods in a class is influenced by DIT_A.

This chapter is organized as following, Section 2 presents the data source that we used in this experiments. Section 3 shows the different steps of the data analysis.

5.2 Data Source

This study based on data collected from 25 real open-source applications. They were chosen arbitrarily and downloaded from several web sites (mainly Sourceforge). They represent 1 247 packages and more than 15 038 classes and interfaces. Table 5.1 gives a brief description of these applications.

Data collection

In order to collect data from subject systems, we have produced a Java program[1] that takes a path of the application, and navigates through the different packages and classes. It loads each class of the application dynamically, navigating from one class to another in the inheritance tree in order to reach to the `Object` class. During this navigation the program collects the following data:

- the total number of inherited methods NOH
- the total number of inherited methods from the application classes NOH_A
- the total number of inherited methods from standard Java classes $NOH - NOH_A$,
- the number of declared methods in the class NOM
- the classical DIT, starting from the root (`Object` class),
- the DIT_A, starting from the first parent application class.

This program focuses only on the inheritance of classes and ignores the inheritance of interfaces. The calculations produced by the program are stored in an excel file.

[1]More details about our tool can be found in Appendix B.

Application	Description	# packages	# classes	max DIT_A
1-NanoXML(1)	small XML parser for Java	1	19	1
2-EMMA(*) (version 2.1)	for measuring Java code coverage	28	255	5
3-AspectJ(2) (2007/02/28)	aspect-oriented extension of Java	10	67	3
4-Chemical Evaluation Framework(*)	software to assist in hazard assessment	3	127	0
5-Java Groups-2.5.0(*)	group communication based on IP multicast	23	918	4
6-Ant project(3) (version 1.7)	Java-based build tool	71	1022	6
7-Jsxe(*)	Java simple XML editor	25	453	2
8-XMLMath(*)	XML-based expression evaluator	2	80	3
9-HTMLCleaner(*)	Transforms HTML a into a well-formed XML	1	27	2
10-Azureus-v3.0.5.0(*)	Java bitTorrent Client	483	4827	7
11-KoLmafia-v12.3(*)	A interfacing tool with online adventure game	27	740	5
12-FreeCol-v0.7.3(*)	Civilization-like game	28	744	3
13-MegaMek-v0.32.2(*)	A networked Java clone of BattleTech	32	811	4
14-Robocode-v1.5.4(*)	A Java programming game	29	277	6
15-Freemind-v0.8.1(*)	A mind mapper and hierarchical editor	29	593	4
16-SweetHome3D-1.2.1(*)	An interior design for choosing placing furniture	6	560	4
17-Hibernate-3.2(*)	Relational Persistence for Idiomatic Java	267	2035	7
18-MyDoggy Docking Framework(*)	Manage secondary windows within main window	51	565	2
19-PDF Split and Merge 1.0.0-b2(*)	A Tool to merge and split PDF documents	51	179	4
20-QuickDownloader(*)	Accelerating downloads	19	90	0
21-Weirdx-1.0.32(*)	A pure Java X Window System server	3	108	4
22-Java Gui Builder0.6.5a(*)	Decouple GUI code from the rest of application	19	163	3
23-Scope/generic HMVC Framework(*)	A framework for component based development	27	189	3
24-Bluepad v0.1(*)	Turns cellphone to a remote PC controller	5	11	0
25-Jaxe(*)	Java XML editor	7	178	2
All	-	1247	15038	

(1) `http://nanoxml.cyberelf.be`; (2) `http://www.eclipse.org/aspectj/`; (3) `http://ant.apache.org/`;

(*) `http://www.sourceforge.net/`;

Table 5.1: Data source

5.3 Data Analysis

We analyzed the *application* classes in all the packages. Since an interface does not include any implementation for any method and objects cannot be instantiated from interfaces, they were excluded of this analysis. So the number of classes that we analyzed is 14 125.

Table 5.2 displays the distribution of these classes with respect to DIT and DIT_A values. For the classical Depth of Inheritance Tree, there is no *application* class with DIT=0 since all application classes inherits at least from `Object` class. For the Depth of Inheritance Tree restricted to application classes, DIT_A is 0 if its superclass is a standard JDK class. The number of classes with $DIT_A = 0$ is not equal to the number of classes with DIT=1 since an application class can inherit from a standard JDK class, which in its turn inherits from another JDK class and so on (cf. the `XMLException` class Fig. 4.1).

Inheritance Depth	#classes w.r.t DIT_A	#classes w.r.t DIT
0	8095	0
1	3432	6653
2	1342	3449
3	797	1623
4	306	973
5	99	689
6	35	366
7	19	254
8	0	84
9	0	26
10	0	8
Total	14125	14125

Table 5.2: Distribution of the application classes w.r.t DIT_A and DIT

5.3.1 Inherited Methods and DIT/DIT_A

In this section, we carried out a first analysis to evaluate the influence of the number of inherited methods with respect to the depth of inheritance.

Correlation analysis

This analysis bases on three statistical concepts: scatter plot, Spearman's rank and Wilcoxon test. From a scatter plot graph, one could easily visualize if there is any correlation between two variables, in our case for instance, DIT and NOM. Using

Spearman's rank allows to have more accurate vision about the correlation. We uses Spearman correlation measurement (and not the more common Pearson correlation), since rs can be applied independent of the underlying data distribution, and independent of the nature of the relationship (which need not be linear). Finally, Wilcoxon test will allow us to know if some distributions (in our case for instance, the methods distribution w.r.t DIT) follow the same statistical law. More details about these concepts are found in Appendix A.

It is expected that the deeper a class is within the hierarchy, the greater the number of methods it is likely to inherit. To do that, we express the following two questions that correspond to the hypotheses $\ddot{B} - 1$ and $\ddot{B} - 2$ (see 4.4):

1. Is the number of inherited methods ($M_{In}(c)$) influenced by DIT?

2. Is the number of inherited application methods ($M_{IA}(c)$) influenced by DIT_A?

A first informal verification consists in drawing the scatter plot diagrams between the number of inherited methods and the DIT. Fig 5.1 displays the number of inherited methods with respect to the DIT. Fig. 5.2 displays the number of inherited methods *from the application classes* with respect to the DIT_A. The scatter plot diagrams show that there is positive relation between the number of inherited methods and the DIT.

In order to carry out a more formal verification, we calculate Spearman's[1] rank-order correlation coefficient, rs, for the number of inherited methods ($|M_{In}(c)|$) (resp. ($|M_{IA}(c)|$) and DIT (resp. DIT_A). We use $rs(n, d)$ to denote Spearman's rank-order correlation between the number of methods n and depth of inheritance d.

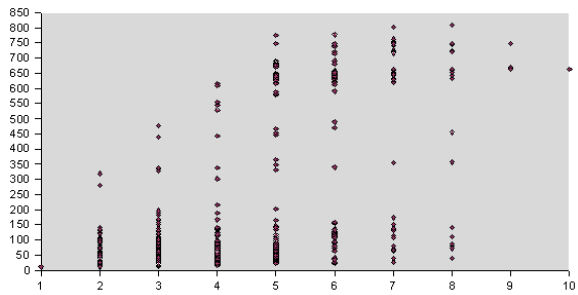


Figure 5.1: Number of inherited methods w.r.t. DIT

[1]Spearman's and Kendall usually produce very similar results and there is no strong reason for preferring one over the other [38].

Application Name	Total Inherited Methods	Total Inherited from Java	Total Inherited from Application	%of Java	%of Application
NanoXML	288	272	16	94,44	5,56
EMMA	3994	3448	546	86,33	13,67
AspectJ	1108	843	265	76,08	23,92
CEF	7440	7440	0	100	0
JGroup	37605	27291	10314	72,57	27,43
Ant	59981	18561	41420	30,94	69,06
Jsxe	48265	47480	785	98,37	1,63
XMLmath	1293	935	358	72,31	27,69
HTMLcleaner	423	342	81	80,85	19,15
Azureus	93868	63864	30004	68,04	31,96
KoLmafia	204932	195701	9231	95,5	4,5
FreeCol	101491	96001	5490	94,59	5,41
MegaMek	96814	88581	8233	91,5	8,5
Robocode	35037	34146	891	97,46	2,54
Freemind	43560	34034	9526	78,13	21,87
SweetHome3D	46094	44324	1770	96,16	3,84
Hibernate	68542	23220	45322	33,88	66,12
MyDoggy	39641	38921	720	98,18	1,82
PDFSplit&Merge	18294	17690	604	96,7	3,3
QuickDownloader	11525	11525	0	100	0
Weirdx	4526	3711	815	81,99	18,01
JavaGuiBuilder	6601	2993	3608	45,34	54,66
Scope	8424	7280	1144	86,42	13,58
Bluepad	851	851	0	100	0
Jaxe	26049	24295	1754	93,27	6,73
Total	966646	793749	172897	82,11	17,89

Table 5.3: The Percentage of Number of Inherited Methods

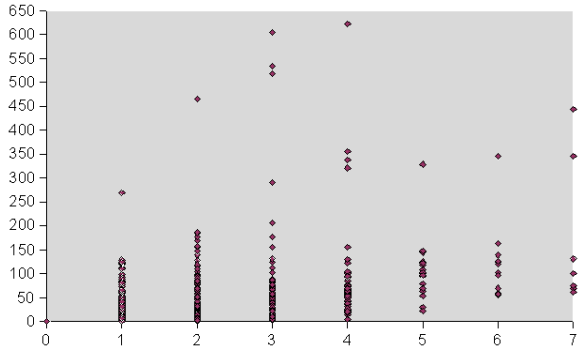


Figure 5.2: Number of inherited application methods w.r.t. DIT_A

We have calculated rs for each application. Table 5.4 shows the rs values observed for each application. As expected, for all cases, rs is positive and goes between 0.79 and 1 for DIT, and between 0.87 and 1 for DIT_A. This indicates a strong positive correlation. This means that the number of inherited methods increase with the DIT or DIT_A.

One can notice from Table 5.3 that the number of inherited methods from standard classes represents a high percentage (more than 80%) of the total inherited methods. While the number of inherited methods of the application classes is less than 20% of the total inherited methods. That allow us to say, including the inherited methods from standard Java classes will increase the number of test cases significantly.

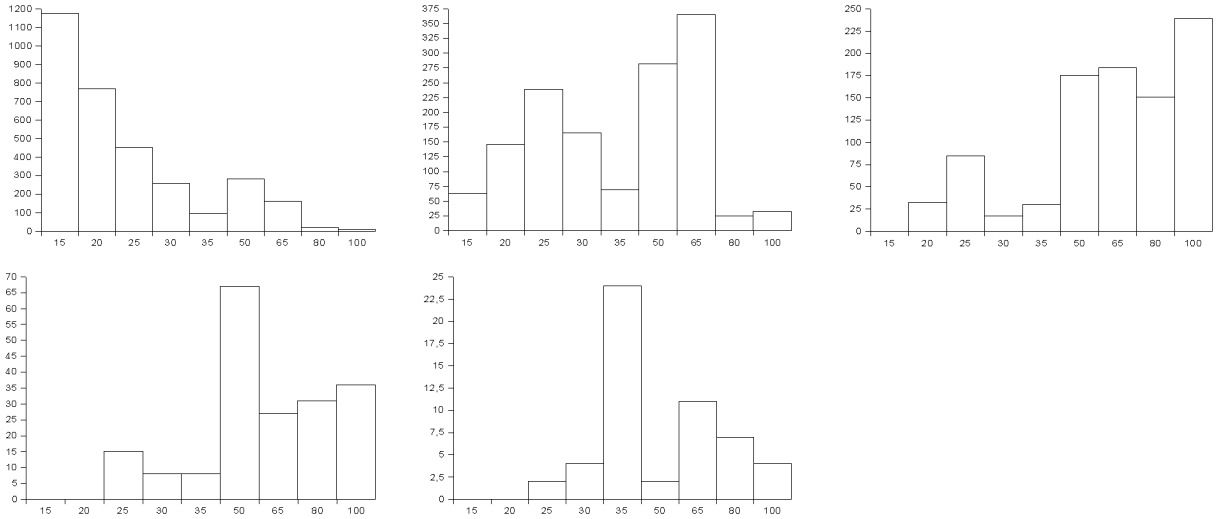
We have also computed the average number of inherited methods for each of those distributions. Table 5.5 shows the average number of inherited (application) methods for DIT and DIT_A. One can notice from Table 5.5 that the average number of inherited methods increases significantly from the level four (DIT=4).

Distribution analysis

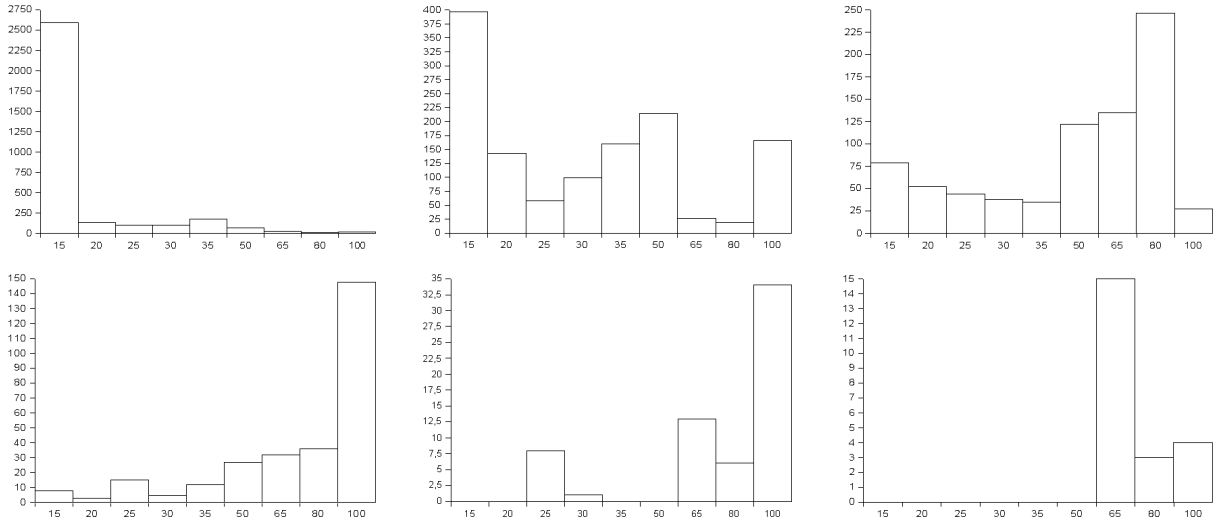
So as expected, the number of inherited methods is correlated to the depth of inheritance tree. The next question is "how does the number of inherited methods increase with the DIT?"

To have a first idea, we have drawn the histograms of the number of inherited methods distribution for a fixed DIT (resp. DITA). Fig. 5.3(a-e) draws these histograms for DIT = 2 to 6 and Fig. 5.3(f-k) draws them for DIT_A = 1 to 6.

For better understanding of these histograms, we present the frequencies in Table 5.6 and which correspond to the histograms Fig. 5.3(a-e). From this table, one could see that the number of classes that inherit less than 15 methods decreases on going from one



(a-e) Number of inherited methods distribution for DIT = 2, 3, 4, 5 and 6



(f-k) Number of inherited application methods distribution for DIT_A = 1, 2, 3, 4, 5 and 6

Figure 5.3: Number of inherited (application) methods w.r.t. DIT (and DIT_A)

Application Name	$rs(\|M_{In}(c)\|$, DIT)	$rs(\|M_{IA}(c)\|$, DIT$_A$)
NanoXML	1	1
EMMA	0,98	0,98
AspectJ	0,98	0,99
CEF	0,91	1
JGroup	0,95	0,98
Ant	0,92	0,95
Jsxe	0,98	0,99
XMLmath	0,96	0,98
HTMLcleaner	0,99	0,99
Azureus	0,98	0,98
KoLmafia	0,95	0,92
FreeCol	0,79	0,98
MegaMek	0,97	0,98
Robocode	0,98	0,99
Freemind	0,79	0,87
SweetHome3D	0,97	0,99
Hibernate	0,96	0,97
MyDoggy	0,98	0,99
PDFSplitAndMerge	0,91	0,97
QuickDownloader	0,99	1
Weirdx	0,92	0,95
JavaGuiBuilder	0,93	0,98
Scope	0,9	0,91
Bluepad	0,91	1
Jaxe	0,9	0,99

Table 5.4: Spearman's rank-order correlation between the number of methods and depth of total and application inheritance tree for each application

inheritance level to a deeper one, that is clear from both Table 5.6 (row 1). In general, this observation is still true for other inherited methods ranges, except for some cases, such as the range 51-65. This observation assures that number of inherited methods increases when going deeper in the hierarchy, therefore, the number of classes which inherit certain range of methods decreases. While the exceptions of this observation, such as the range 51-65, is interpreted that at this range and at DIT=3 the number of classes increases (from 164 to 365 classes) since there are methods that have been inherited in previous ranges. On the other hand, the number of classes decreases (from 365 to 184) because not all classes have the same depth, in other words, not all the classes that inherit from the range 51-65 at DIT=3 have children.

Same observation of the histograms Fig. 5.3(a-e) can be noticed on the histograms

DIT	Average Number of Inherited Methods	DIT_A	Average Number of Inherited Application Methods
1	11	0	0
2	24.68	1	11.83
3	52.23	2	27.86
4	80.69	3	54.69
5	340.9	4	80.61
6	443.84	5	93.26
7	527.5	6	95.74

Table 5.5: Average of Number of inherited (application) Methods w.r.t DIT and DIT_A

Fig. 5.3(f-k). However, it is important to mention that the variation of number of classes and the number of inherited methods on the different levels of the hierarchy do not follow same distribution law, which means calculating the number of inherited methods (or a range) at certain depth does not allow assessing the exact number of inherited methods at the next depth.

#inherited methods ranges	DIT=2	DIT=3	DIT=4	DIT=5	DIT=6
11-15	1177	63	0	0	0
16-20	768	146	32	0	0
21-25	451	239	85	15	2
26-30	259	165	17	8	4
31-35	97	69	30	8	24
36-50	283	282	175	67	2
51-65	164	365	184	27	11
66-80	20	25	151	31	7
81-100	11	33	239	36	4

Table 5.6: Number of classes w.r.t the ranges of number of inherited methods

Note: For DIT = 1, all classes inherit from Object and thus have 11 inherited methods. For $DIT_A = 0$, application classes do not inherit from another application class, thus they have 0 inherited application methods.

5.3.2 Defined Methods and DIT/DIT_A

Previous analysis has confirmed that the number of inherited methods was increasing with the DIT, both considering the total inheritance tree or tree restricted to the application

classes. Another question is: does the number of defined methods increase with DIT? This question is related to the hypotheses $\ddot{A} - 1$ and $\ddot{A} - 2$ (see 4.4).

One can expect that the number of methods defined in a class is increasing with the depth of inheritance tree since the behavior of the object is expected to be more complex (or at least, more detailed). Or, one can expect that fewer methods are required since most of the behavior has been previously defined. And finally, one can expect that there is no relation between them.

Again, as a first informal verification, we have drawn the scatter plot diagrams between the number of defined methods and the DIT. Fig 5.4 and 5.5 display the number of defined methods with respect to the DIT and DIT_A. The scatter plot diagrams show that there is no evident relation between the number of defined methods and the DIT or DIT_A.

We have calculated the Spearman's rank order for the classes considered all together. We have obtained rs($|M_D(c)|$,DIT) = 0.05 and $rs(|M_D(c)|,\text{DIT}_A) = 0.06$. This indicates that there is no correlation between the number of declared methods and the depth of inheritance.

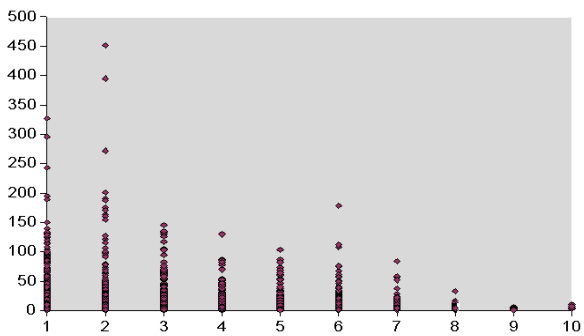


Figure 5.4: Number of defined methods w.r.t. DIT

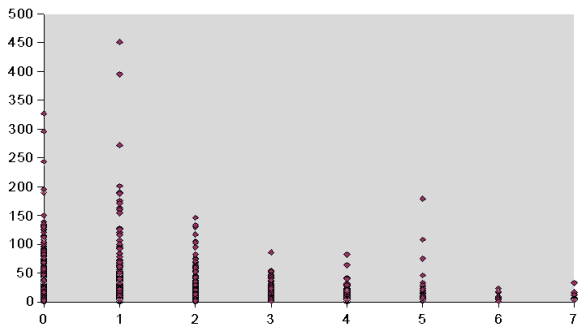


Figure 5.5: Number of defined methods w.r.t. DIT_A

	1	2	3	4	5
1	-	$2.28\ 10^{-12}$	0.404	0.282	0.763
2	-	-	$2.51\ 10^{-8}$	0.016	0.002
3	-	-	-	0.181	0.834
4	-	-	-	-	0.326

Table 5.7: P-value of Wilcoxon test for the declared methods distribution at a fixed DIT

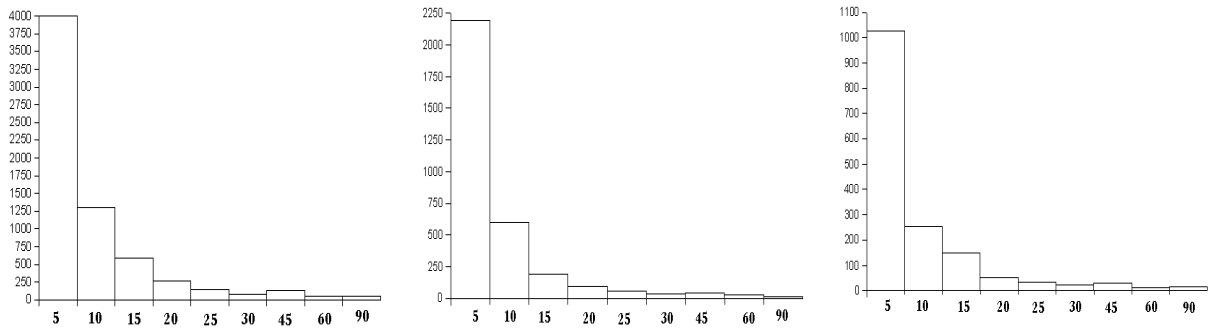
	0	1	2	3	4
0	-	0.046	$3.23\ 10^{-7}$	0.0003	0.21
1	-	-	$4.14\ 10^{-10}$	8.710^{-6}	0.05
2	-	-	-	0.48	0.68
3	-	-	-	-	0.53

Table 5.8: P-value of Wilcoxon test for the declared methods distribution at a fixed DIT_A

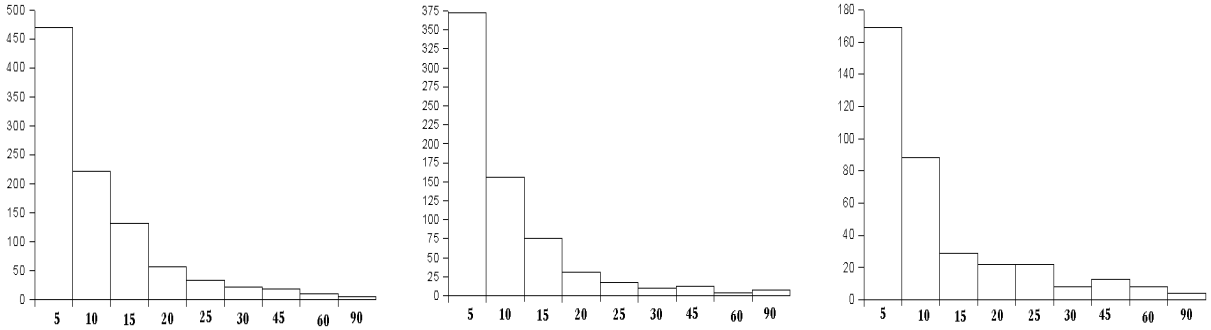
To visualize the distribution law, we have drawn the histograms of the number of declared methods distribution for a fixed DIT and a fixed DIT_A (see Fig. 5.6). These histograms give the impression that the different distributions follow the same probabilistic law. To check if the different distributions follow the same law, we have used the Wilcoxon test (see A.6.2). This is a nonparametric test that compares two paired groups. It has an associated null hypothesis. Generally, one rejects the null hypothesis if the p-value is smaller than or equal to the significance level. If the level is 0.05, then the results are only 5% likely to be as extraordinary as just seen, given that the null hypothesis is true.

Results for DIT and DIT_A are given in Tables 5.7 and 5.8. For $\mathrm{DIT}_A = 2$, 3 and 4, it is reasonable to accept that the samples come from the same law. It is also reasonable to accept that samples from $\mathrm{DIT}_A = 0$ and 1 come from the same law, which is different from the previous ones. For DIT, it is reasonable to accept that all the sample come from the same law, expect the second sample (DIT=2), which is different.

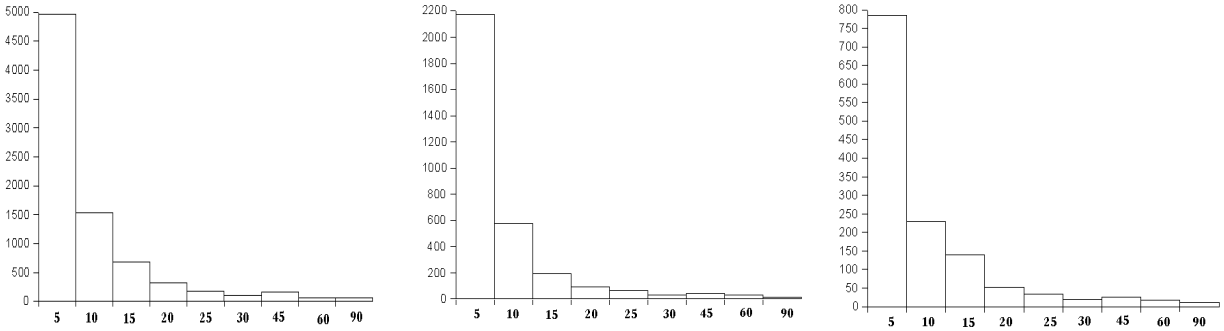
As a consequence of these results, it might be interesting to have a predictive tool that predicts the number of defined methods for a sub-class. This prediction is based on the number of defined methods of the parent class and its depth of inheritance. Such a predictive tool may give an idea about how much the sub-classes (new children) are.



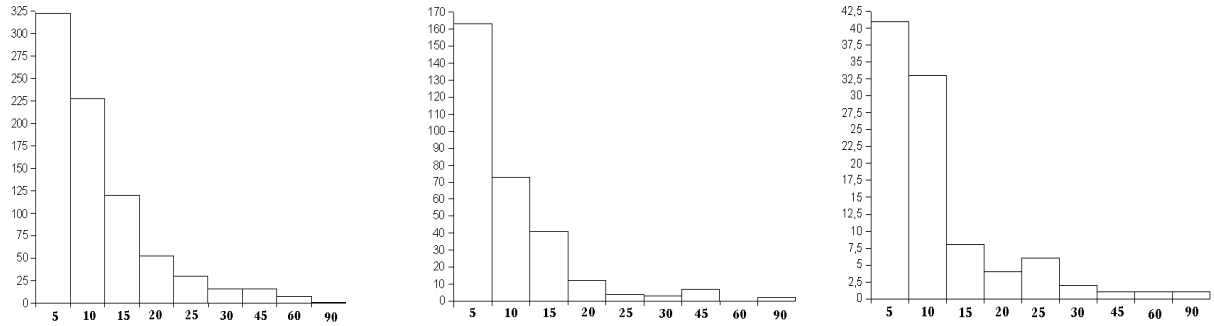
(a-c) Number of defined methods distribution for DIT = 1, 2 and 3



(d-f) Number of defined methods distribution for DIT = 4, 5 and 6



(g-i) Number of defined application methods distribution for $\mathrm{DIT}_A = 0$, 1 and 2



(j-l) Number of defined application methods distribution for $\mathrm{DIT}_A = 3$, 4 and 5

Figure 5.6: Number of defined methods w.r.t. DIT and DIT_A

5.4 Conclusion

The goal of this case study was understanding how the Depth of Inheritance Tree (DIT) can be used to predict the cost of unit testing. Several previous studied have showed the influence of inheritance on the quality of the developed systems. During this case study, we wanted to estimate effective testing cost to achieve methods coverage criterion by estimating predictive testing cost. We expressed the effective cost of unit testing as the number of methods to be tested, and the predictive cost as the depth of inheritance tree. Additionally, since we concentrated on object oriented language, we focused on an important feature i.e. *inheritance* therefore, we have distinguished two types of testing strategies: those which consider that inherited methods should be re-tested and those which only consider defined methods to be tested.

We carried out this study with respect to the adapted definition of DIT i.e. DIT_A that we defined in the previous chapter, and with respect to the different hypotheses that we made in chapter 4. As a result of this analysis, one can use $\mathrm{DIT}/\mathrm{DIT}_A$ for predicting the cost of testing if it is intended to retest the inherited methods at the unit level. Otherwise, $\mathrm{DIT}/\mathrm{DIT}_A$ is not an adequate predictor for the cost of testing.

The number of methods to be tested is not considered as an accurate estimation to estimate effective testing cost, that why we are going to consider more accurate effective cost in the following chapter. In the next chapter we focus on branch coverage criterion, for which we estimate the effective cost as the number of branches to be passed.

Chapter 6

Is DIT a Good Predictive Cost for Branch Coverage Testing?

6.1 Introduction

In the previous chapter, we considered during our experimentation the cost of testing as the number of methods to be tested, with respect to different testing strategies T_s and T_t. Here, we present similar experiment but with respect to a different cost of testing. As we mentioned in chapter 4, the number of methods to test represents a rough way to estimate the cost of testing. Therefore, we consider here more accurate estimation that is the cost of achieving branch coverage.

This study is carried on the same set of applications which are shown in Table 5.1, and with respect to the following hypotheses that we detailed in chapter 4:

A. For branch testing strategies that <u>do not consider inheritance</u> (T_s testing strategies), the difficulty to produce the test (estimated by MCC) is not correlated with DIT_A.

B. For branch testing strategies that <u>do not consider inheritance</u> (T_s testing strategies), the number of tests to be produced (estimated by WMC) is not correlated with DIT_A.

C. For branch testing strategies that <u>consider inheritance</u> (T_t testing strategies), the number of tests to be produced (estimated by WMH_A) is correlated with DIT_A.

6.2 Statistical analysis

Application	MCC	rs(MCC,DIT$_A$)	rs(WMC,DIT$_A$)	rs(WMH$_A$,DIT$_A$)
NanoXML	24	-0,270	-0.376561	0.156694
EMMA	55	-0,030	0.055409	0.428051
AspectJ	22	-0,018	0.002037	0.508643
CEF	70	0	0	0
Jgroup	91	0,039	0.230362	0.363079
Ant	109	0,158	0.140267	0.424910
JSXE	169	-0,057	-0.086228	0.127950
XMLMath	10	-0,227	-0.282045	0.615053
HTMLCleaner	49	0,154	0.124086	0.344639
Azureus	190	0,0469	-0.049650	0.082084
KoLmafia	341	0,037	0.055853	0.349263
FreeCol	129	0,056	0.165906	0.622566
MegaMek	627	0,127	0.135182	0.408373
RoboCode	54	-0,250	-0.101887	0.241725
Freemind	137	0,097	0.277715	0.619646
SweetHome3D	69	0,133	0.184145	0.433074
Hibernate	102	0,038	-0.077916	0.264031
MyDoggy	50	-0,024	0.063910	0.257330
PDF Split	32	0,025	0.184899	0.441028
Quick Downloader	20	0	0	0
Weirdx	97	0,221	0.059432	0.557433
Java GUI Builder	12	0,268	0.456639	0.439395
Scope	23	-0,127	-0.379926	0.056544
Bluepad	13	0	0	0
JAXE	143	0,073	0.159762	0.453179

Table 6.1: Spearman coefficients

6.2.1 Data Analysis

In the following subsections we are looking for validating the different assumptions about the cost of testing and the relevant metrics to cyclomatic complexity.

Checking the hypothesis A.

We first analysed MCC and calculated Spearman's rank-order correlation coefficient, rs, for MCC with respect to DIT$_A$. The computation was done for each application. Table

6.1 displays the results. As it can be noticed, the rs values are on average very small. This seems to indicate that the correlation between MCC and DIT_A is a weak correlation.

In a second step, we have drawn the histograms of MCC distributions for each DIT_A. We have chosen the intervals classically used for the analysis of CC (see sect. 4.1). Figures 6.1 display the histograms for $\mathrm{DIT}_A = 0$ to DIT_A=5 all applications together.

These histograms give the impression that the different distributions follow the same probabilistic law. To check this, we have used the Wilcoxon test.

	0	1	2	3	4	5
0	-	0.95	1	1	0.14	0.98
1	-	-	1	0.95	0.05	0.96
2	-	-	-	0.72	10^{-7}	0.31
3	-	-	-	-	10^{-6}	0.22
4	-	-	-	-	-	0.99

Table 6.2: P-value of Wilcoxon test for MCC at a fixed DIT_A

The p-value of the Wilcoxon tests are given in Table 6.2. For DIT_A =0, 1, 2, 3 and 5, it is reasonable to accept that the samples come from the same law. The reason why MCC distribution for DIT_A=4 is different from the other is not clear yet. However, one should keep in mind that the samples of data for DIT_A=4 and 5 are not as large as the others. For this reason, observations may be not significant. This analysis confirms that the difficulty to test a class is independent of the depth of inheritance tree.

Checking the hypothesis B.

We carried out the same analysis for Weighted Methods per Class (WMC). Table 6.1 (column 4) displays Spearman's rank-order correlation coefficients. For WMC, coefficients are generally quite small, except for **NanoXML**, **SCOPE**, **XMLMath**, **Java GUI builder** and **Freemind**. These small values tend to show that there is no correlation between WMC and DIT_A, that is to say, for these applications, the number of tests to write does not increase with DIT_A.

For the five applications: one should note that **NanoXML** has 17 classes out of 19 that have a DIT_A equals to 0. The two last 2 classes have a DIT_A equals to 1. It is therefore not really relevant from a statistical point of view. For all the other, the spearman rank-order coefficients are quite consistent with the observation of the scatter plot (see Figure 6.2). For instance, the **SCOPE** application has 11 classes of DIT_A=2

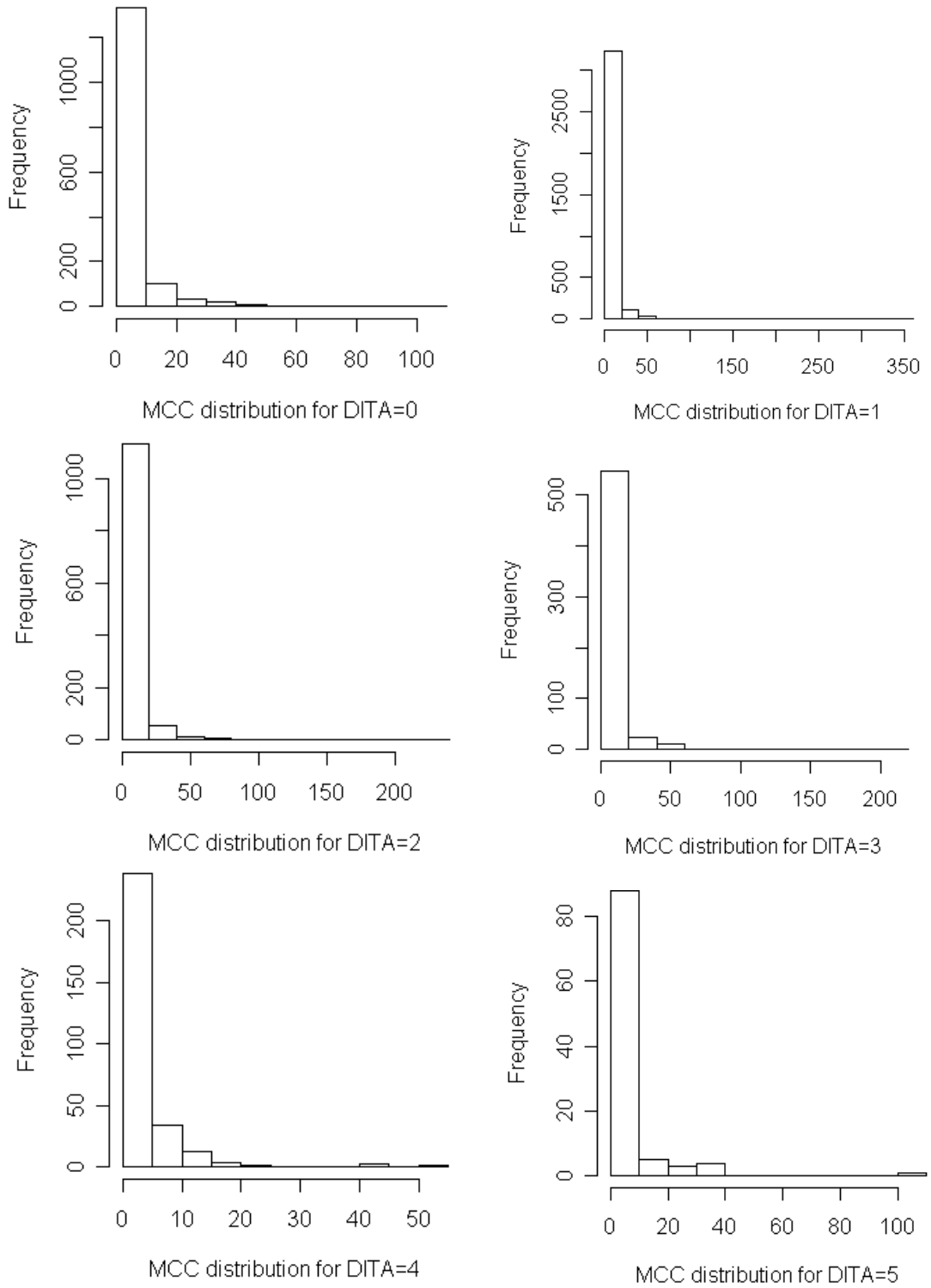


Figure 6.1: MCC distribution with respect to DIT_A=0 to DIT_A=5

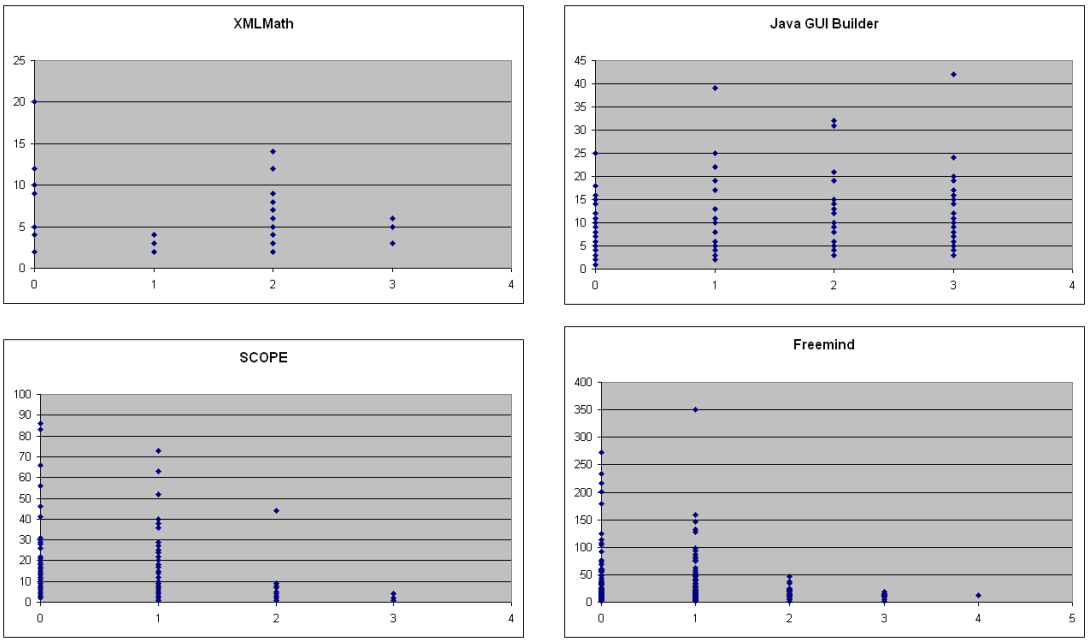


Figure 6.2: Scatter plot for WMC w.r.t. DIT_A for some applications

and 9 classes of DIT_A=3. Even if the sample is also not really relevant, classes at these levels seems to require less tests than at level $\mathrm{DIT}_A = 0$ or 1. (WMC is between 1 and 4 at level 3, with an average of 2, between 1 and 44 at level 2 with an average of 9, between 1 and 110 at level 1 with an average of 19 and between 1 and 86 at level 0 with an average of 15). Similar observation can be done for the **XMLMath** application.

For **Java GUI builder** on the contrary, WMC tend to increase with DIT_A. (WMC is between 1 and 25 at level 0 with an average of 6, between 2 and 39 at level 1 with an average of 27, between 3 and 32 at level 2 with an average of 10, and between 3 and 42 at level 3 with an average of 12). Similar observation can be done for **Freemind** application.

That led us to say, generally there is no correlation between the depth of inheritance of the application DIT_A and the weighted methods per class WMC, which means the number of tests to be produced with a T_s strategy is not correlated with DIT_A.

Checking the hypothesis C.

For Weighted Methods for Hierarchy (WMH_A), it is naturally expected that it would increase with DIT_A, since for one inheritance tree, a class at a certain level has a WMH_A superior than its ancestor(s), by definition. For this reason, one could expect positive

values closed to one for the Spearman's rank-order correlation coefficients. On Table 6.1, one can notice that the values are not so high.

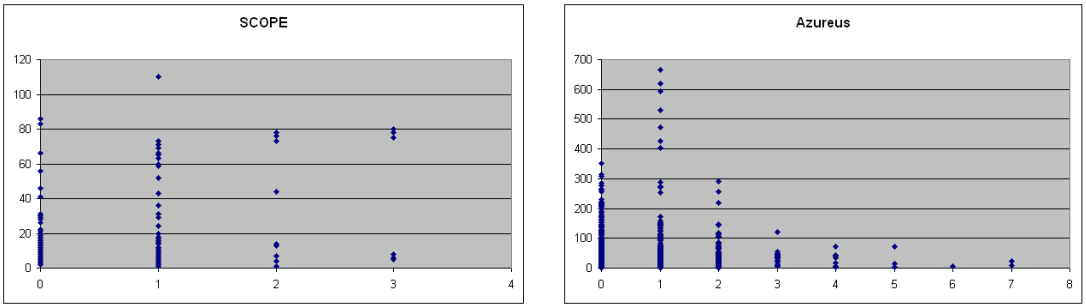


Figure 6.3: Scatter plot for WMH_A w.r.t. DIT_A for **SCOPE** and **Azureus** applications

Two elements can explain this result. Firstly, there are lots of classes at level 0 that do not have children (i.e. they are not in an inheritance tree). Those classes can have a WMC very high (sometimes higher than the WMH_A of the classes at the last level(s)). A second element is that data are compared all together. In one inheritance tree, WMH_A increases. All classes together, the inheritance trees are mixed and classes from one tree can be compared with classes from other ones. An illustration is given Figure 6.3 with the scatter plot of WMH_A with respect to DIT_A for **Azureus** and **AspectJ** applications.

These results allow us to state that DITA could not be considered as a good indicator for estimating the cost of structural testing.

6.3 Limits of the Experiment

In order to carry out our experiment, we have arbitrarily chosen 25 open-source Java applications from different web sites (mainly sourceforge). It is quite surprising to see that inheritance is not used as often as it could be expected: for the whole applications, 65% of the classes are defined at DIT_A=0 (see Table 5.2). Classes at levels 5, 6, and 7 represent approximately 1% of the classes. The choice of the application and especially the type of applications (open-source) is possibly not representative of any Java software applications.

Another important point is that the Cyclomatic Complexity is usually used to approximate the upper bound of the number of tests to be produced. It does not take into account the infeasible "paths". Therefore, fewer tests can be required in reality. So it could be misleading to estimate the effective cost of testing with CC.

6.4 Conclusion

In chapter 5, we have carried out an experimentation in which we have distinguished two types of testing strategies: those which consider that inherited methods should be re-tested and those which only consider defined methods to be tested. The cost of unit testing was expressed as the *number of methods* to be tested. This estimation of cost was considered to be too inaccurate.

In this chapter, we wanted to analyze DIT_A as a predictive evaluation of the testing cost with respect to a more precise effective testing cost. For this reason, we have chosen to focus on branch coverage, which is considered by the practitioners to be "a minimum mandatory testing requirement". We expressed the cost of test as the *number of test cases that are necessary to achieve the branch coverage* of the class methods expressed by Weighted Methods by Class and Weighted Methods for the Hierarchy (WMC and WMH_A) and the difficulty of testing expressed by the Maximum Cyclomatic Complexity of the methods within the class (MCC).

We have observed that the complexity of testing classes (MCC) was not correlated with DIT_A. We also observed that WMC is often not correlated with DIT_A and that WMH_A is often correlated with DIT_A but not as much as expected. This means that when one tests only the defined methods in a class, the DIT_A does not influence the number of tests to produce in general. On the contrary, when one tests the defined methods in the class and retests all inherited method, the number of tests to produce increase with DIT_A, but not in a predictable way. Therefore, *DIT_A is not a good predictive metric at a design level.* It is too abstract to be really relevant. This work has to be published in [110].

The comparison between an effective cost and predictive cost as we showed till now may not appear realistic, therefore we decided to lead another experiment in which we focus on real test cases which are written in JUnit. In the following chapter we show our analysis that is based on the relationship between the complexity of program and the complexity of generating JUnit test with respect to branch coverage criterion.

Chapter 7

Predicting the Cost of JUnit Designing

7.1 Introduction

In the previous chapters, we chose the number of required tests to be generated as an indicator of the cost of testing. This indication does not consider the difficulty of generating the test cases. Another method of assessing the testing cost is the difficulty of the produced (generated) testing programs. Intuitively, the more a program is complex, the more time it needs to be built and validated. Therefore and in order to complete our previous experiments that we presented in precedent chapters, we carried out an additional experiment which looks for predicting the complexity of testing programs. Here we focus on some particular context where the testing programs are written using JUnit.

JUnit is a library that could be used at unit testing level for testing methods and classes. It has been proposed for the purpose of writing and running tests in `Java`. It is not an automated tool, so the programmer needs to write the test cases by hand. Previously, we chose WMC as a predictor of the number of required tests, here we wanted to estimate more precisely some testing effort, therefore, we chose the effort of JUnit test design. Thus, if we consider the WMC of JUnit tests as a predictor to estimate the effort of designing the tests, then we could check if the number of required tests is related to the effort of designing the tests. In other words, we would like to check two assumptions:

A. For branch testing strategies, the number of required tests to be produced is correlated with the effort of generating the testing programs.

B. For branch testing strategies, the number of required tests to be produced is not correlated with the effort of generating the testing programs.

We would like to mention here, that we do not have control on the JUnit tests which we chose for the applications under study, but all these JUnit tests do not consider the inheritance tree, i.e. they do not retest inherited feature. That means the testing strategies of the selected applications corresponds to the testing strategies T_s that we discussed in Chapter 4. Our study that we introduce in this chapter, aims to estimate the cost of JUnit designing through the evaluating of the cost of a class under test. We use the metric *WMC* as a predictor of the cost. Previously, we discussed the metric *WMC*, therefore we just remind here the definition of *WMC*. It is the sum of the complexity of each method in a class. The complexity of a method is expressed by the cyclomatic number.

7.2 Basic Concepts in JUnit

JUnit is a library which is used to test classes and methods. It can run tests files by executing a tool called *test runner*. The JUnit is not an automatic process, therefore one must write the tests manually. The tests JUnit could be written for achieving different goals, i.e. a specific criterion of test. For instance, the goal of writing JUnit tests could be achieving some coverage criteria, verifying the boundary conditions, etc.

To illustrate the basic concepts of JUnit, let us have the following simple example that returns the absolute value of an integer value. Then we are going to write the JUnit test of this class.

```
package trivialExample;

public class AbsoluteValue {
 private int value;
 private int absoluteValue;

 public AbsoluteValue(){ }

 public AbsoluteValue(int value){ this.value = value; }

 public void setValue(int value){ this.value = value; }
```

```
 public int getValue(){ return value; }

 public int getAbsoluteValue(){
  if (value >= 0){ absoluteValue = value; }
  else {absoluteValue = value;}//This must be absoluteValue=-1*value;

  return absoluteValue;
 }
}
```

To start using JUnit, first we need to install the JUnit.jar file[1]. After that we can write the JUnit test that starts with importing the required packages. The JUnit test class of our simple example could be something similar to the following one:

```
package trivialExample;
import org.junit.*;
import org.junit.Assert;

public class AbsoluteValueTest {

 @Test
 public void getAbsoluteValueTest(){
  Assert.assertEquals(2,(new AbsoluteValue(2)).getAbsoluteValue());
  Assert.assertEquals(3,(new AbsoluteValue(-3)).getAbsoluteValue());
 }
}
```

The class `AbsoluteValueTest` represents the test cases for our class `AbsoluteValue`. The class `AbsoluteValueTest` may have different scenarios to test `AbsoluteValue` class's methods. In our case we just want to test the method `getAbsoluteValue()`. The corresponding testing method for that one is `getAbsoluteValueTest()`. The method `assertEquals()` evaluates whether a returned value equals the expected one. If there is inequality then JUnit will show that there is an error.

Assert.assertEquals(ExpectedValue, ReturnedValue)

In our example, we test the method `getAbsoluteValue()` against two values 2 and -3. The expected values must be 2 and 3 respectively. Since we have introduced an error in the method `getAbsoluteValue()`, then the returned value for (-3) will be (-3). For

[1]The JUnit library can be downloaded from `http://junit.org`, we use the version 4.1

that JUnit will show the failure, by indicating that the expected value does not match the returned one.

That is all what we need about JUnit to go ahead in our following statistical study. Of course, there are other important notion related to JUnit, such as *oneTimeSetUp*, *oneTimeTearDown*, *setUp*, *etc*, but here we do not need to explore all the features of JUnit. In the following, we are going to analyze some open source applications, where our goal is comparing the complexity of a class with the complexity of its JUnit test class. For example, the weighted methods per class *WMC* of the class `AbsoluteValue` is 6. While the *WMC* of test class `AbsoluteValueTest` is 1. In other words, this study is an attempt to predict the cost of writing JUnit tests by measuring the *WMC* of the class under test.

7.3 Data Source

Our study was carried out on 10 open-source applications[1]. They were chosen arbitrarily and downloaded from sourceforge.net (except Ant, which is downloaded from ant.apache.org). Table 7.1 gives small description for these applications, and the number of JUnits that were written for some classes in the application. We would like to mention here that the number of JUnit classes shown in Table 7.1 equals to the number of classes for which these JUnits were written. In other words, we ignored other JUnit tests which are written to run a sequence of JUnit tests.

Application	**Description**	**#JUnit**
Ant	Java-based build tool	198
SweetHome3D	An interior design for choosing placing furniture	11
JGroup	Group communication based on IP multicast	50
JavaGuiBuilder	Decouple GUI code from the rest of application	50
FreeCol	Civilization-like game	11
Hibernate	Relational Persistence for Idiomatic Java	49
Dozer	A Java Bean to Java Bean mapper	48
OpenSaml	A cross-platform SAML implementation	7
DrJava	Java design environment to foster test-driven software development	89
Sahi	An automation and testing tool for web applications	36
Total	-	549

Table 7.1: Data Source

[1]In this study, it was not possible to apply the analysis on all the applications that we analyzed in previous chpaters, since not all of them have the associated JUnit tests

7.4 Data Analysis

We have analyzed these applications in order to calculate the *WMC* for each class in addition to its associated JUnit. We refer in the following to the *WMC* of a JUnit class as *WMC4Junit*. Table 7.2 shows the maximum and the minimum values of *WMC* and *WMC4Junit* for each application. We would like to mention here that these values concern only the classes which are associated with JUnit classes.

Application	Max(WMC)	Min(WMC)	Max(WMC4Junit)	Min(WMC4Junit)
Ant	263	1	121	2
SweetHome3D	288	12	28	3
JGroup	434	1	102	3
JavaGuiBuilder	39	3	42	3
FreeCol	947	15	33	2
Hibernate	142	1	98	3
Dozer	205	1	40	2
OpenSaml	75	1	12	3
DrJava	1067	1	80	2
Sahi	147	2	27	2

Table 7.2: Max and Min value for WMC and WMC4Junit

In order to predict the relationship between the *WMC* and *WMC4Junit*, first we drew the scatter plot graph for the *WMC* with respect to *WMC4Junit*. Additionally, we tried to express this relationship by using simple regression model (linear regression), in order to see if one could predict the value of *WMC4Junit* by depending upon the value of *WMC*. Then we used more formal correlation analysis. More precisely, we used the spearman rank-order correlation coefficient.

From the figures 7.1 and 7.2, one could notice that the cyclomatic complexity of JUnit class (WMC4Junit) *slightly* increases with the cyclomatic complexity of the associated class (WMC). Three applications *OpenSAML* and *FreeCol* do not match this result, one reason behind that is there is not probably enough information (JUnit classes) for these two applications.

This result was confirmed by another analysis using the Spearman correlation, where we found a positive correlation between the two variables *WMC* and *WMC4Junit*. Table 7.3 shows the values of Spearman correlation analysis. As a consequence, reducing the complexity of a class, may lead to reducing the complexity and difficulty of writing tests at the unit testing level. The advantage of considering this result into account is the easiness of its applying, because the *WMC* can be estimated instantly during the coding

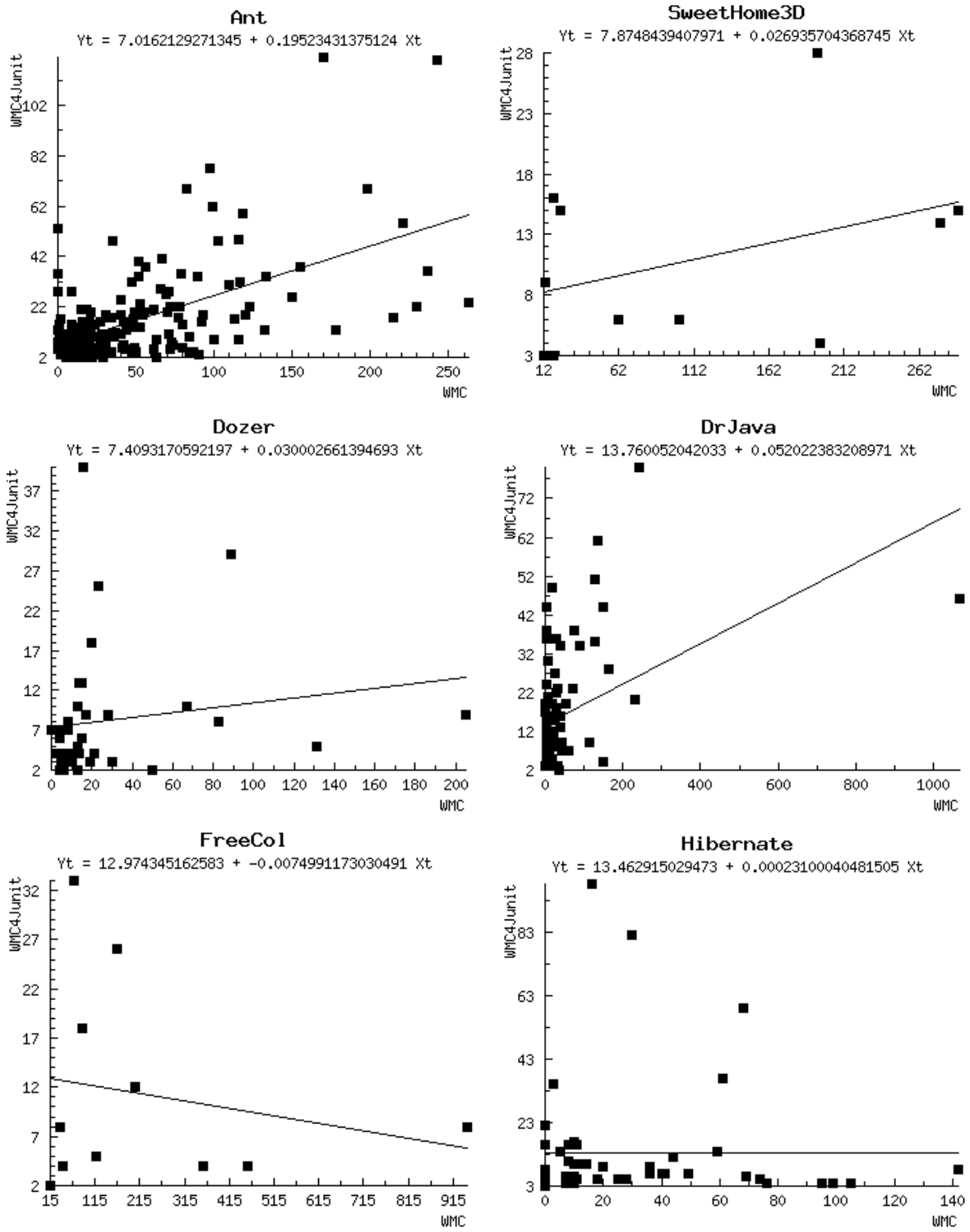


Figure 7.1: Visualizing the relationship between WMC and WMC4Junit

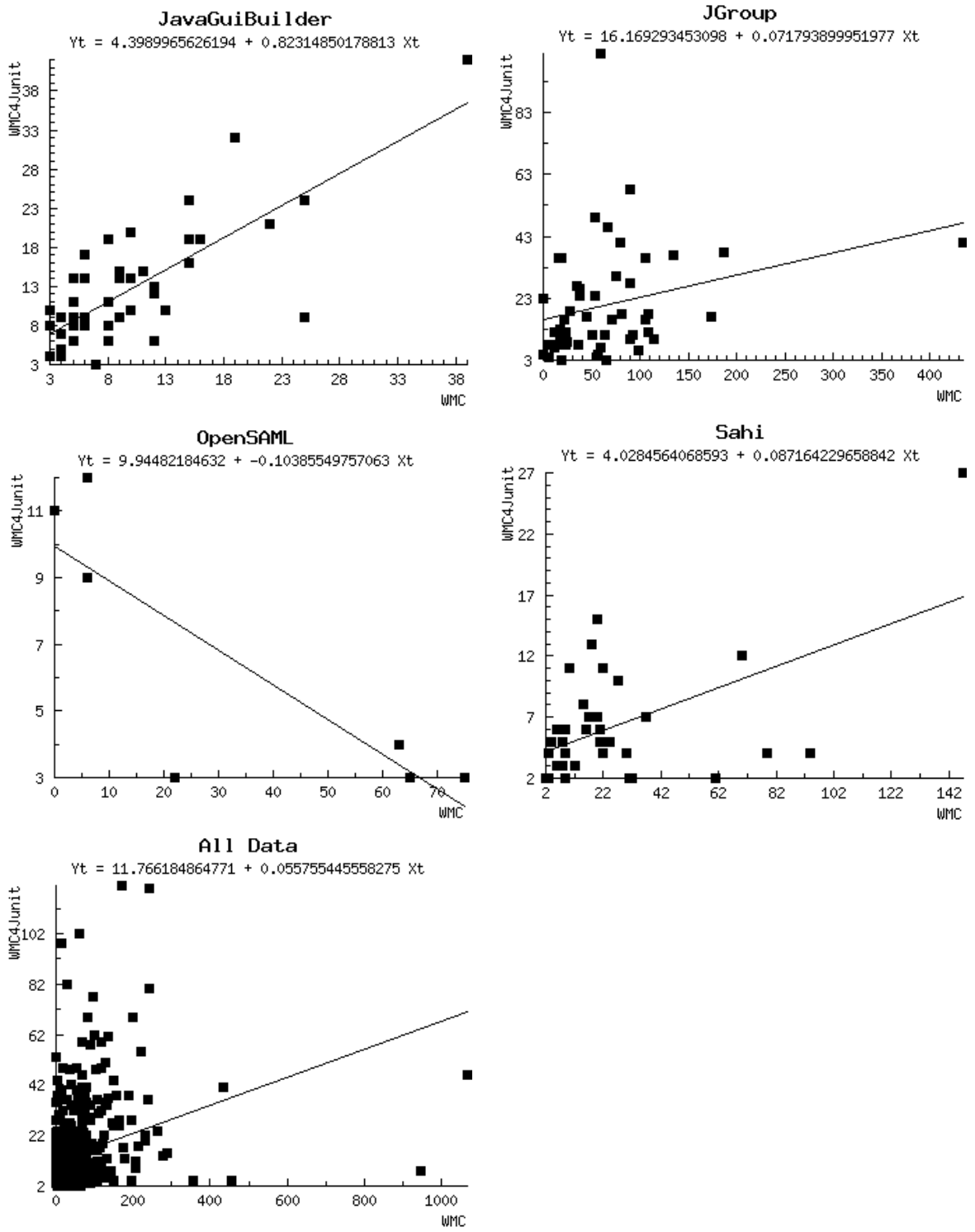


Figure 7.2: Visualizing the relationship between WMC and WMC4Junit

phase. Once the programmer notices a high level of complexity, he could rewrite/modify the method/class responsible of that complexity.

Application	Spearman Correlation Value
Ant	0.46
SweetHome3D	0.3
JGroup	0.3
JavaGuiBuilder	0.67
FreeCol	0.08
Hibernate	0.03
Dozer	0.23
OpenSaml	-0.7
DrJava	0.3
Sahi	0.26
All Data	0.32

Table 7.3: Spearman Correlations Analysis

7.5 Limit of the Work

This work relies on a set of JUnit tests collected with some open source applications. It is not so often that tests are delivered with open source code. The lack of data is a first limit of this study. Moreover, the test files are not documented. In particular, the way the tests were designed is not described. So it is possible that selected JUnit test files were not designed to achieve same coverage type. The fact that the data may have different nature is the second limit of the work. For a deeper analysis, it could be more interesting to identify what kind of coverage was expected to be achieved by the JUnit files.

7.6 Conclusion

Since in previous chapters we concentrated more on comparing the effective and predictive testing cost and in order to get more reliable results we decided to lead an analysis based on real test cases which are written in JUnit.

In this chapter, we tried to associate the number of required tests with the effort of designing the tests. We estimated the number of tests as WMC, while the effort of testing is estimated as the WMC of JUnit tests. The results of this analysis show that there is a moderate correlation between the number of tests required to achieve branch coverage in a class (WMC) and the complexity of JUnit tests. Therefore, WMC is not only an indicator of the number of tests, but also is an indicator of the complexity of JUnit test design.

The following chapter is dedicated to show the results of our analysis on integration testing level. For this analysis we focus on detecting antitestability patterns that could appear during the software development cycle.

Chapter 8

Detecting Testability Antipatterns during the Development Process

8.1 Introduction

In chapter 5, 6 and 7, we focused on some metrics that can be used to estimate the testability at the unit testing level. Here in this chapter, we are interested in evaluating the testability at the integration testing level. More precisely, we concentrate on detecting some testability antipatterns and determining at which point they are introduced during the development. We are motivated by several questions:

- Is it frequent to find cycles in a model?
- What is the percent of/How many cycles are introduced during the coding?
- At which point the cycles are introduced?
- How do the cycles sizes evolve in the development?
- Are there any elements that favor the occurrence of cycles? Which ones?

A testability antipattern is a factor that could affect negatively the testability of software. It is a design solution known to make test difficult (and/or known to increase the number of test to carry out) [14]. Two testability weaknesses have been described in [16], ***self usage*** and ***interactions***. Both of them characterize dependency cycles in classes. As we showed in chapter 3, lots of testability metrics have been proposed [35, 36, 21, 82, 46, 56], but only few studies have been carried out to identify the different weaknesses of a design patterns [14, 70].

The antipatterns were proposed to be metrics that could be used at system or integration testing level. Difficulty of testing usually depends on dependencies among classes, which require an order in the integration.

In this chapter we compare the antipatterns at source code level and at different abstraction levels, in order to understand at which point they are introduced during the development. We studied a number of open-source Java applications to detect the self usage and interactions antipatterns occurrences at the source code and at different levels of abstraction. The objective was to observe how frequent, how complex the cycles in those applications and at which level they are introduced. We especially wanted to see if the cycles are mainly introduced during the modeling or the coding phase.

8.2 Testability Antipatterns

Design patterns represent solutions to problems that arise when software is being developed in a particular context [14]. A testability antipattern "describes undesirable configurations in the class diagram". They could affect testability efforts and make tests ineffective [64, 15].

An object oriented system is a set of classes that communicate with each other. Some of these classes depend on others. Different studies showed that dependency is the main cause behind the occurrence of antipatterns. In [70] two types of dependencies were defined. The first type is **physical dependency**. The physical dependency exists between two classes A and B if A cannot be ***compiled*** without B (e.g. inheritance). And the second type is **logical dependency**. The logical dependency exists between A and B if a change happens to a B would require a change to A.

A dependency relationship could be either direct or indirect. It is indirect if the path between A and B includes other classes, otherwise it is direct. Figure 8.1(a) shows a dependency (interaction) between the class C and D, which could be interpreted either directly, or indirectly through the inheritance tree. Figure 8.1(b) represents the notion of class cycle, for example, one cycle exists between D and E, another one exists also between B and D through C.

The complexity of a dependency cycle increases when there are more than one path to reach from one class to another, which in turns increase the potential usages. That means more test cases or at least potential hidden errors. Also more classes in a cycle will increase the difficulty of the test, one reason for that is the need for more instances to be instantiated.

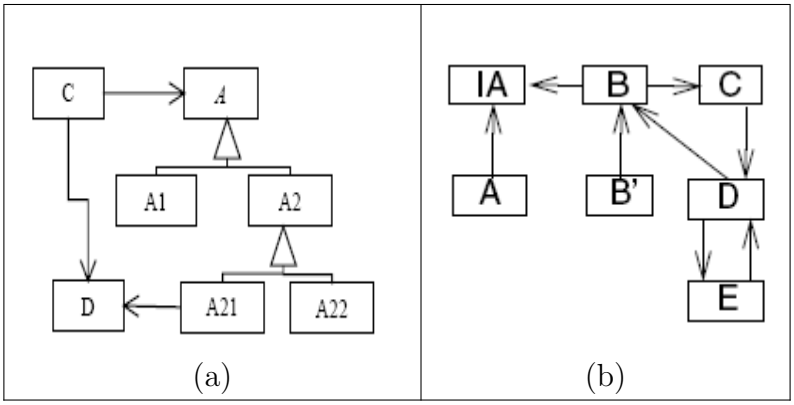


Figure 8.1: Class interaction and Cycles

These types of dependency (interaction through more than one path and class cycle) should be avoided either by refactoring or by using class interfaces [70, 14]. Several studies [19, 129] focused on detecting and assessing the dependency. Jungmayr in [70] introduced a set of metrics to measure the dependability and its influence on the testability.

As we previously said, we are interesting here in detecting at which point the cycles are introduced in the development phases. In the following we introduce our method in simulating a software development approach.

Application	Description	# packages	# classes
1-NanoXML(1)	small XML parser for Java	1	19
2-Chemical Evaluation Framework 1.001(*)	software to assist in hazard assessment	3	127
3-Jsxe(*)	Java simple XML editor	25	453
4-XMLMath(*)	XML-based expression evaluator	2	80
5-HTMLCleaner(*)	Transforms HTML a into a well-formed XML	1	27
6-MegaMek-v0.32.2(*)	A networked Java clone of BattleTech	32	811
7-weirdx-1.0.32(*)	A pure Java X Window System server	3	108
8-Java Gui Builder0.6.5a(*)	Decouple GUI code from the rest of application	19	163
9-Bluepad v0.1(*)	Turns cellphone to a remote PC controller	5	11
10-Jaxe(*)	Java XML editor	7	178
11-JDom1-1(2)	Access to XML data	7	68
12-JFreeChart-1.0.9(*)	Create Charts	41	475
13-KoLmafia-v12.3(*)	A interfacing tool with online adventure game	27	740
All	-	173	3260

(1) `http://nanoxml.cyberelf.be`; (2) `http://www.jdom.org/`; (*) `http://www.sourceforge.net/` ;

Table 8.1: Data source

8.3 Simulation of the Development Phases

There are various methods of software development. Some of these methods propose *top-down* approach [112], which begins with describing the system at high level of abstraction

(called model or high design level), then this description is refined progressively until getting the code (the final implementation).

In the following we define three abstraction levels. These levels intuitively correspond to an approach that starts by highlighting the principle classes in the application, then it identifies their attributes, afterwards it defines their methods (signatures), then it defines the inner classes[1], and finally the complete coding.

Obtaining the models using this method has two properties. First, all applications are consistent with their models. Second, all models are expressed at the same level of abstraction. As a consequence, this method makes the comparison consistent.

Level 1 considers the attributes of the application classes, but neither considers the methods nor the attributes of an inner class type. In other words, it just considers the attributes of types such as numbers, array, lists, string,...etc and the attributes whose their types are classes which are not inner classes.

```
public class SampleExample{
 /* Attributes */
  int attribute_1;
  String attribute_2;
  byte[] attribute_array;
  Non_innerClassType attribute_complex;
}
```

Level 2 considers the attributes and the methods signatures of the application classes, but not those of inner class types, nor the methods which have one or more parameters of inner class types, nor the methods that return a value of an inner class type.

```
public class SampleExample{
 /* Attributes */
  int attribute_1;
  String attribute_2;
  byte[] attribute_array;
  Non_innerClassType attribute_complex;
 /* Methods */
  public int getAttribute_1(){
   return attribute_1;
  }
  public void setAttribute_1(int para0){}
```

[1]An inner class is a class that is defined in another class, it could be declared also inside the body of a method (either with name or without name).

```
  public void init(){}
}
```

Level 3 considers the attributes, methods signatures and inner classes. It also considers the attributes of inner class types, and the methods which have parameter(s) of inner class types, in addition to the methods that return a value of an inner class type. Since the inner classes do not appear early in the development, therefore we did not include them in the first and second levels of the abstraction levels.

```
public class SampleExample{
 /* Attributes */
  int attribute_1;
  String attribute_2;
  byte[] attribute_array;
  Non_innerClassType attribute_complex;
  InnerClassEx attribute_3;

 /* Inner Class Type */
  InnerClassEx(){
    // some attributes and methods
  }

 /* Methods */
  public int getAttribute_1(){
   return attribute_1;
  }
  public void setAttribute_1(int para0){}
  public void init(){}
  public void calc(int para0,
     InnerClassEx para1){}
}
```

Level 4 or Source code level represents the complete implementation of the class.

8.4 Data Analysis

This work aims at studying several open source applications in order to detect and analyze the testability antipattern occurrences, and especially the class cycles, with the intention to understand at which point the cycles are introduced during the development. To

achieve this goal we analyzed 13 open-source Java applications. Table 8.1 gives small description for these applications, and their sizes expressed by the number of packages and classes. They represent 173 packages and more than 3260 classes and interfaces. They were chosen arbitrarily on the web mainly from sourceforge.

It would be interesting to collect data during the real development of the application. However, this is really difficult because the models which are used to build an application are not distributed with the application. Therefore, we chose to simulate the top-down approach using the reverse engineering.

To get the first three levels of abstraction, we developed a program based on Java reflection. It can extract the three levels from the byte code. We have compiled each model in order to get a model that is free of compilation errors. Then we have used Classycle's Analyser[1], an Eclipse plugin, to analyze the class dependencies in Java applications at the source code level and at the abstraction levels. From this plugin we can get the number of cycles and their size for each application.

The first goal of this analysis is identifying the class cycles that appear in each application. Table 8.2 shows the frequency of different cycle sizes, expressed also by a percentage. From this table one can observe that the number of cycles of size 2 and 3 represent 49.05% and 20.95% respectively of the total number of cycles of all sizes, while the cycles that have a size greater than 21 smaller than 30 represent 2.38% of the total number of cycles.

Although the cycles of size n are less complex than cycles of size m, where $m > n$, but the total complexity caused by the cycles could be reduced if we can refactor the cycles of size 2 and 3 which represent about 70% of all classes. Therefore, one should avoid and refactor the cycles of all sizes, and that is *not* limited only to the big cycles.

In order to answer the questions "*Are there any elements that favor the occurrence of cycles? Which ones?* ", we studied the structure of the classes that make a part of the cycles. We found that the main cause behind the previous observation is the occurrence of inner classes. Table 8.2 shows the maximum cycle size, and the minimum cycle size, the total number of classes related to all cycles in an application, the number of inner classes, and the percentage of inner classes in a cycle to the total classes in it. From the Table 8.2 one can notice that more than 50% of classes that belong to cycles are inner classes.

[1]http://classycle.sourceforge.net/

Therefore avoiding or limiting the use of inner classes will reduce the number of potential usages, that means less number of test cases will be required.

The second goal of our analysis is detecting at which point the cycles are introduced during the development. Therefore we carried out this analysis at four levels, the three abstraction levels in addition to the source cod level. The different levels of abstraction allow us to know how many cycles are introduced at each level. Although it is clear that the source code level will have the maximum number of cycles, and the *level 1* will have the minimum number of cycles, but we do not know at which level the number of cycles begins to increase significantly, and at which level the cycle's complexity increases.

Application/Cycle size	2	3	4	5-10	11-20	21-30	>30
CEF	1	4	1	3	0	1	1
Bluepad	0	0	0	1	0	0	0
HtmlCleaner	0	0	0	0	0	1	0
JavaGUIBuilder	6	4	1	1	0	0	0
Jaxe	6	2	0	0	0	0	1
Jdom	2	4	1	0	0	1	0
JfreeChart	20	5	1	2	1	0	1
JSXE	12	2	2	5	1	1	4
KoLmafia	13	4	4	3	0	1	1
MegaMek	38	18	3	11	3	0	4
Weirdx	2	1	0	0	0	0	1
NanoXML	2	0	0	0	0	0	0
XmlMath	1	0	1	0	0	0	0
Total	103	44	14	26	5	5	13
Percent	49,05	20,95	6,67	12,38	2,38	2,38	6,19

Table 8.2: The frequency of cycles of different sizes at the source code level

Application/Cycle size	#Max	#Min	#Cycles	#Classes in all cycles	#Inner Classes	%Inner classes
NanoXML	2	2	2	2	1	50
CEF	38	2	11	105	91	86.67
JSXE	129	2	27	361	209	57.89
XmlMath	4	2	2	6	1	16.67
HtmlCleaner	21	21	1	21	0	0
MegaMek	101	2	77	571	321	56.22
Weirdx	58	2	4	65	10	15.38
JavaGUIBuilder	6	2	12	34	22	64.71
Bluepad	10	10	1	10	0	0
Jaxe	132	2	9	150	84	56
JDom	24	2	8	44	13	29.55
JFreeChart	36	2	30	121	22	18.18
KoLmafia	598	2	26	703	396	56.33
Total	-	-	210	2193	1170	53.35

Table 8.3: Max/Min cycle size and number of cycles at the source code level

Table 8.4 shows, for each abstraction level, the maximum cycle size, minimum cycle size and number of cycles. From both Table 8.3 and 8.4, we can observe that the number of cycles increases progressively from level one to level four (source code level).

	Level 1			Level 2			Level 3		
Application/Cycle size	#Max	#Min	#Cycles	#Max	#Min	#Cycles	#Max	#Min	#Cycles
1-NanoXML	0	0	0	0	0	0	2	2	1
2-CEF	0	0	0	3	3	1	3	3	3
3-Jsxe	2	2	2	8	2	6	46	2	43
4-XMLMath	0	0	0	0	0	0	0	0	0
5-HTMLCleaner	0	0	0	8	8	1	10	10	1
6-MegaMek	13	2	5	39	2	15	45	2	47
7-weirdx	13	2	4	27	2	2	27	2	9
8-Java Gui Builder	0	0	0	0	0	0	3	2	7
9-Bluepad	0	0	0	0	0	0	0	0	0
10-Jaxe	4	3	3	9	3	3	24	2	19
11-JDom	3	3	3	8	8	1	10	2	7
12-JFreeChart	2	2	3	31	2	9	31	2	24
13-Kolmafia	2	2	1	29	2	4	29	2	99
Total			21			42			260

Table 8.4: Max/Min cycle size and number of cycles at the abstraction levels

Additionally, from Figure 8.2 one can notice that the number of cycles for the 3rd, 7th, 10th and 13th application at *level 3* is greater than the number of cycles at the source level. To interpret this observation we study the structure of the cycles, and we found that some cycles at the source level could be formed from a combination between two or more cycles found at *level 3*. This also could be seen by comparing the maximum number of classes that form the cycles at the source level and the level 3 (see Table 8.3 and 8.4).

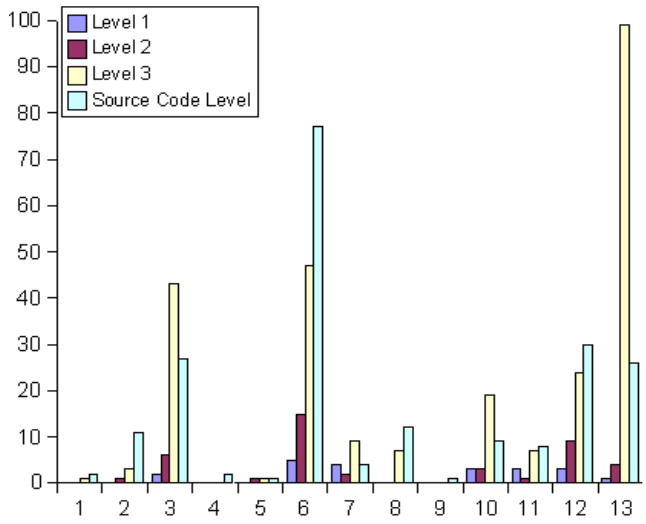


Figure 8.2: Number of cycles per application at each level

This analysis leads us to two conclusions. First, it is important to detect the cycles at the model level and to refactor them in order to eliminate them as many as possible and as soon as possible. Because the more there are cycles in the model, the more cycles will

be in the coding phase, or the more complex they will be. As a consequence the difficulty of testing increases.

Second, it is not sufficient to analyze cycles at the model level, since new cycles would be introduced during the coding phase. This means analyzing and refactoring of the code *may* be required to ease the test. Moreover, the integration tests may not be built only from the model since it may not describe existing cycles at the source code.

8.5 Limits of the Work

In this study, we focused only on the relationships between classes, more precisely, on detecting the cycles that are introduced during the development phases. In this analysis we have simulated the development by defining three levels of abstraction and collecting data using reverse engineering. But these levels do not represent real development case, which is not possible due to the difficulty of obtaining the *real* models that are used for building the applications under study. This study could be extended to detect other types of antipatterns that could be introduced during the development. Also, it is important to analyze the consequences of refactoring the cycles and the inner classes, in order to evaluate how much the testability is influenced by this modification. And whether it is really merit the cost of the modifications.

8.6 Conclusions

In this chapter, we presented the notion of antipatterns, which are weaknesses that reduce the testability of software. The antipatterns could be detected early in the life cycle of software development. Additionally, we presented our quantitative analysis, that we carried out to find if there is any relationship between the occurrence of antipatterns and the characteristics of the code.

The results of this analysis show that the cycles are frequent to be found in the different levels. According to our defined levels, the number of cycles increases significantly at the level 3 and 4. Most cycles are of size 2 and 3, they represent about 70% of the total cycles. Also, we found that inner classes participate strongly in forming the cycles.

Moreover, we found that detecting the cycles at the model level is important, but it is not sufficient to avoid all problems caused by the cycles. One reason behind that either the number of cycles introduced during any level of abstraction increases at the next refined level or the cycles may become more complex. The results of this work have

been published in [107, 109] Therefore, analyzing the cycles at each level of development is essential, and it is interesting to have some tools associated with the development environments that could detect the cycles during the different phases of development.

Conclusion and Perspectives

Software testing is an expensive activity. It is the most reliable and used method to validate the software programs. Different approaches were proposed to reduce the cost of testing. One approach is the software testability. A large work has been done to define metrics that are proposed to estimate the testability of the software. But a few works have been carried out to validate those metrics.

In this thesis, we showed a large set of software testability metrics, and we focused on validating some classical metrics. Some of these metrics are subject to controversial opinions. In our validation process, we differentiate between two testing levels, the unit testing level and the integration testing level. At the unit level, we focused on validating testability metrics against specific testing criteria. Also, we have introduced new metrics that were adapted from classical software metrics, these new metrics can be used to estimate the cost of testing at the unit level. For the integration testing, our focus was on the testability antipatterns, we were interested in detecting the antipatterns, and checking when they are introduced during the development life cycle.

Our validation based essentially on empirical analyses that we carried out on different open-source `Java` applications of different sizes. Finally, we developed "Metrics Calculator", that is a simple tool to calculate our adapted metrics and some other classical ones.

As a perspective to this work, it would be interesting to extend this methodology to validate other testability metrics with respect to other testing strategies. That may lead to well-defined guide that permits the developers and testers to determine which metrics have to be used to evaluate the testability with respect to the testing methods which will be used.

Another perspective is introducing a testability transformation approach that aims at detecting predefined and customized testability antipatterns in order to transform low testable software to more testable one. This approach may be developed to be used at different development phases.

Appendix A

Basic Concepts in Statistics

A.1 Introduction

The role of scientists is not simply to try to describe phenomena, but rather to explain them. From available knowledge, they develop scientific theories of why something happens and then make appropriate observations to check them. In turn, these observations can lead to modifications of the theory, which must also be checked, and so on [78]. Scientific research involves a continual interaction between theory and empirical observations. An observation is characterized by several characteristics: type of variables, role of variables, accuracy and precision.

To start a statistical study, one should clarify what we shall actually observe or measure. Useful information can occur in many contexts. It may be obtained at first hand by measuring the height of each person for example, or at second hand, by asking the people how tall they are. Then this information should be shown by using some data representation methods, in order to be used later to understand the nature of this information, and to discover if there is any relationship that relates this data.

In our case, the information that we need to study is the proposed metrics of software testability. Therefore, we have calculated certain number of testability metrics for several applications in order to study the correlations (if there is any) between these metrics.

In this appendix, we present general basic concepts in statistics that have been used in our empirical validation (chapters 5, 6, 7 and 8). Section 2 describes the different types of statistical variables. Section 3 presents some methods for representing data. Section 4 shows how to measure the correlation between variables. Section 5 is dedicated to describe different notions which are used in testing statistical hypotheses process. Section

6 describes the goodness of fit tests. Section 7 focuses on analyzing the relationship between variables by using a regression model.

A.2 Types of Variables

A variable is a symbol that represents any value of a specified set of values. The variables refer to things that could be measured, controlled or manipulated in a research problem. The variables differ according to their role, they could be independent or dependent. In the following we list the different types of variables.

1. **Nominal**: when the names of the variables have no relationship of order or magnitude among them, then the variable is called nominal, e.g.: sex, nationality or religion.

2. **Ordinal**: if a variable can have three or more distinct values and these have a rank order, without measure of distance among them, in this case the variable of ordinal type, e.g.: letters grades (A, B, C...), appreciation of something (like, dislike, indifferent).

3. **Integral**: in case of values that are counts, e.g. number of children.

4. **Continuous**: when the values are measures, and these variables have two types (ratio and interval), e.g.: length, weight.

5. **Discrete**: the discrete variables are those variables which are not continuous, e.g.: age, income, temperature.

A.3 Data Representation

Collected data should be shown in a representative and objective manner, in such way one could get useful information from that representation. Showing data can be done in different ways depending on the type of involved variable. Two main forms to represent data are tables and graphics [78]. These two forms are important in descriptive statistics, which is concerned with describing the number of relationships between them.

A.3.1 Tables

After determining the variable that one wants to observe, one should determine how to register it. An initial table of any study will contains observed values of all the variables. This table called ***raw data***, in which each column corresponds to a different variable.

Individual	**Sex**	**Age**	**Opinion**
1	M	20	Agree
2	F	65	Disagree
3	F	45	Indifferent
4	M	71	Disagree
5	M	45	Agree

(a) Raw Data

	Frequency
M	3
F	2
Total = #individuals	5

(b) Frequency Table

Table A.1: Data Representation

To summarize raw data we could use the notion of frequencies, where for each category we calculate the number of individuals observed of a variable. For example in the Table A.1a, we have 3 males and 2 females. The corresponding **frequency table** is Table A.1b, which represents the number of individuals observed in each category. Here one should notice that the total size of the sample (number of individuals) is equal to sum of all the frequencies of a characteristic.

$$\text{Number individuals} = \text{Number of males} + \text{Number of females}$$

A simple frequency table does not allow showing the relationship among variables in case of several variables. To show the relationship ***cross-classified frequency table*** or ***contingency table*** can be used.

Example 1 Let us consider a group of 30 students, 13 out of 30 are males. This group has the choice to follow an algorithm course. Some of them (18 students, within 10 are males) has followed it, while the others not. Table A.2 uses cross-classified frequency table to summarize these data, the right column represents the frequency of the row; 13 students are male, and 17 students are female. The last line represents the frequency of column; 18 students have followed the course, and 12 have not.

The Table A.2 is a case for the frequencies of two variables, which called two-way table. Table A.3 represents a general form for two-way table of two variable Y and Z, where n_{ij} corresponds to the frequency of Y_i with considering Z_j, and the marginal totals are obtained by the following equations:

	Followed the course		
Sex	**Yes**	**No**	**Total**
M	10	3	13
F	8	9	17
Total	18	12	30

Table A.2: Contingency Table For Example 1

	$Z=1$	$Z=2$	$\cdots$	$Z=J$
$Y=1$	n_{11}	n_{12}	$\cdots$	n_{1J}
$Y=2$	n_{21}	n_{22}	$\cdots$	n_{2J}
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
$Y=I$	n_{I1}	n_{I2}	$\cdots$	n_{IJ}

Table A.3: Two Way Table

$$n_{i+} = \sum_{j=1}^{J} n_{ij}, i = 1, 2, ..., I(rows)$$

$$n_{+j} = \sum_{i=1}^{I} n_{ij}, j = 1, 2, ..., J(columns)$$

$$n_{++} = \sum_{i=1}^{I} \sum_{j=1}^{J} n_{ij} = n(Total)$$

A.3.2 Graphics

There are several types of diagrams for representing data such as scatter plot, stem plots, leaf plots, box plots and whisker plots. In particular, we use the scatter plot and histogram diagrams that we present here.

Scatter Plot

Scatter plot (scatter graph) could be used to take a first look on the behavior of data understudy. It uses the Cartesian coordinates to visualize the values of X and Y. From this graph, we can also determine the correlation (see also A.4) between the two variables as following:

- Positive correlation if the data points go from lower left to upper right (Figure A.1).

- Negative correlation if the data points go from upper left to lower right (Figure A.2).

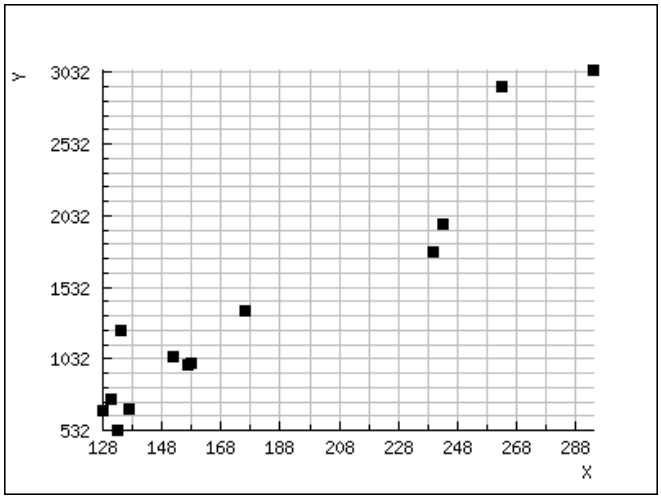


Figure A.1: Scatter plot for data given in Table A.4

Example 2 Let us consider two variables X and Y, their values are given in the Table A.4. To visualize the type of the correlation between these two variables, we represent the data of X and Y using scatter plot. Figure A.1 represents the correlation between the both variables. From this graph, we can notice that data points go from the lower left to the upper right. Therefore we say there is a positive correlation between X and Y.

X	152	134	157	176	133	158	137	128	131	294	240	243	263
Y	1041	1230	984	1366	532	997	683	672	754	3048	1780	1970	2938

Table A.4: Sample data of Example 2

Example 3 Let us consider two variables X and Y, their values are given in the Table A.5. X represents the distance traveled by a car, and Y represents the gasoline remained in the car's tank[1]. For visualizing the relationship between the both variables we use scatter plot graph. Figure A.2 represents the correlation between X and Y. From this graph, we can notice that data points go from the upper left to the lower right. Therefore we say there is a negative correlation between X and Y.

[1]We suppose that the car's driver does not refill the tank until it goes empty.

X:Distance	0	50	150	280	350	420	540	680	700	750
Y:Gasoline	15	14	12	9,5	8	6,5	4,2	1,4	1	0

Table A.5: Distance/Gasoline Data - Example 3

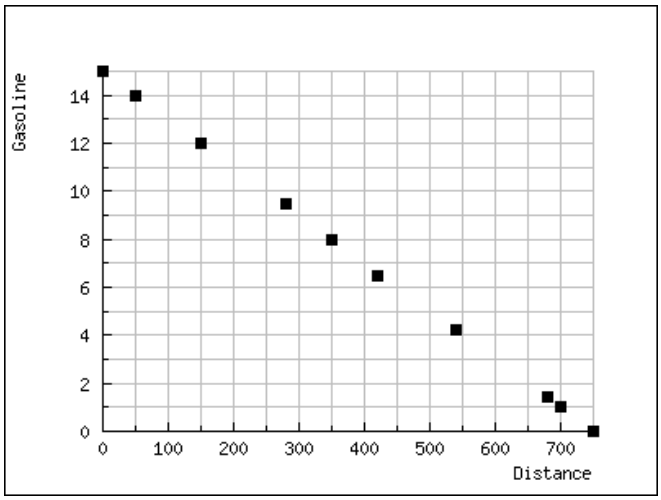


Figure A.2: Scatter plot for data given in Table A.5

Frequency Histograms

The histogram is a summary graph that represents the count of data points which fall in various ranges. In other words, a frequency histogram is a way to represent how many items are in each numerical category.

Example 4 Let us consider the marks of students in maths (marks of 100): *Mark: 47, 24, 67, 20, 38, 24, 76, 98, 31, 1, 38, 28, 6, 21, 2, 49, 38, 70, 19, 29, 62, 54, 38, 45, 44, 70, 92, 26, 35, 46, 44, 11, 49, 45, 6, 22, 2, 53, 33, 64.*

We want to know how these marks spread according to five interval: [0,25], [26,45], [46,55], [56,70], [70,100]. To do that, frequency histogram can be used as shown in Figure A.3.

Furthermore, suppose that for each course, a histogram represents how the marks spread. To check these histograms if they follow the same distribution, or they belong to the same distribution law, one could use *Wilcoxon test method (see A.6.2).*

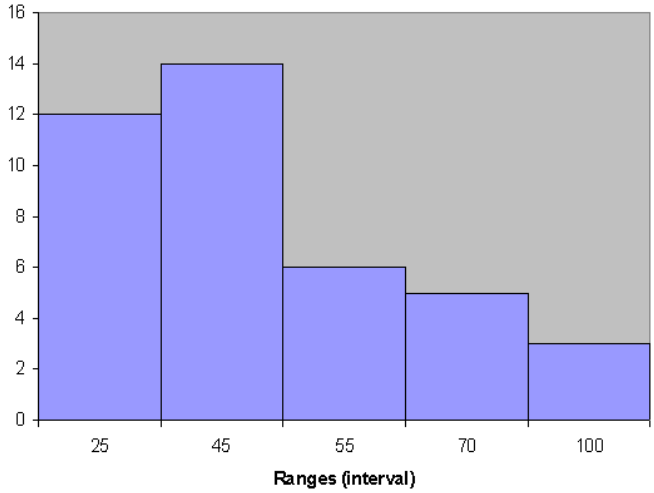


Figure A.3: Frequency histogram for marks in Example 4

A.4 Correlation Coefficient

The correlation coefficient measures the strength of relationship between two variables. Let us consider two variables X, Y that are covarying, then there are two possibilities of this covariance, either X and Y are increasing/decreasing together, or one of them increases and the other decreases. In the first case, we say a positive correlation exists between X and Y, and in the second case, we say a negative correlation exists between X and Y.

Example 5 Consider X to be man's age between 1 and 20 years, and Y to be his/her height. One could state that X and Y are both increasing, so man's age and his height are correlated positively. Another example, consider X to be the distance that a car travels, and Y to be the gasoline remained in the tank. In this example, X increases while Y decreases which indicates a negative correlation between the two variables (see Example 3 and Figure A.2).

There are different methods to measure the correlation between variables. Some of these methods depend on the relationship between the variables (if it is linear or not). A relationship between two variables X and Y is **linear** if Y could be written as a function of $X(Y = \mathrm{F}(X))$, where the degree of X in $\mathrm{F}(X)$ is at most one. In other words, Y could be written in the form $Y = \mathrm{a}X + \mathrm{c}$, where a and c are constant. Otherwise, the relationship is **non-linear**.

There are two categories of statistical methods: parametric and non-parametric. A parametric statistical method could be used only if we know form which statistical dis-

tribution the data sample is drawn. Otherwise, if the distribution is unknown then we must use a non-parametric method to evaluate the correlation. In general a non-parametric method could be used instead of a parametric method, but not the vice versa. Choosing a parametric or non-parametric method is a matter of judgment. In general, a non-parametric method protects against some violations of assumptions and not others. Because it does not require any assumption about normality, equal variances, and independence, which are required by some parametric methods. On the other hand, it is not preferred to use non-parametric all the time, because parametric methods are robust and have great power efficiency (relative to sample size).

We discuss in the following some of these methods:

1. Pearson Correlation
2. Spearman's Correlation
3. Kendall Rank Correlation

A.4.1 Pearson Correlation

Pearson Correlation Coefficient is a statistical parametric method that requires a linear relation between X and Y, and they should be drawn from *Normal* distribution. Pearson correlation r is calculated from the following equation:

$$r = \frac{1}{n-1} \sum \left(\frac{X_i - \bar{X}}{s_x}\right)\left(\frac{Y_i - \bar{Y}}{s_y}\right)$$

where $\bar{X}$, $\bar{Y}$ are the sample averages, s_x, s_y are standard deviations[1] and n is the sample size.

The value of Pearson r ranges between -1 and +1. If the value of r is zero then there is no relationship between X and Y. If the value of r belongs to]0,1[then X and Y are **correlated positively**. If the value of r belongs to]-1,0[then X and Y are **correlated negatively**. In case of r equals one, that means all data are lying on same line, and both X and Y are increasing, in this case X and Y have **perfect positive correlation**. In case of r equals -1, that means the value of Y increases while the value of X decreases, and in this case X and Y have **perfect negative correlation**.

[1]Let X be a random variable and let $\bar{X}$ be the average value of X, then the standard deviation of X is given by: $\sqrt{E(X-\bar{X})^2}$, where E represents the mathematical expectation.

Example 6 If we go back to the sample data given in the Table A.4 and calculate the Pearson correlation value[1], we get $r = 0.9539$ which indicates a positive correlation between the two variables X and Y. In the following we show how to calculate the Pearson correlation:

Step 1: we start by calculating the average of X and Y:

$$\bar{X} = \frac{1}{n}\sum_{i=1}^{n} X_i = 180,46$$

$$\bar{Y} = \frac{1}{n}\sum_{i=1}^{n} Y_i = 1384,23$$

where n is the sample size.

Step 2: we need the standard deviation for X and Y. The standard deviation for sample data is calculated from the following formula:

$$s_x = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(X_i - \bar{X})^2}$$

$$s_x = 55,85$$

$$s_y = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(Y_i - \bar{Y})^2}$$

$$s_y = 796,95$$

Step 3: Now, we can apply the formula of Pearson, where the all required values are presented in the Table A.6. Finally, we get $r = 0,95$

A.4.2 Spearman's Correlation Coefficient

Spearman correlation does not require a linear relationship between the two variables. In addition to that, it is non-parametric method. So it could be used whatever the distribution of X and Y. But we should be aware that *this method is used if there are no tied pairs in the sample.* Spearman's Correlation is interpreted in the same way as Pearson Correlation.

Spearman correlation ρ is calculated as following:

[1]Several tools could be used to compute the Pearson correlation value, e.g. Microsoft Excel.

X	**Y**	$(X-\bar{X})^2$	$(Y-\bar{Y})^2$	$(X-\bar{X})/s_x$	$(Y-\bar{Y})/s_y$	$(X-\bar{X})(Y-\bar{Y})/s_x.s_y$
152	1041	810,06	117807,36	-0,51	-0,43	0,22
134	1230	2158,67	23787,13	-0,83	-0,19	0,16
157	984	550,44	160184,67	-0,42	-0,50	0,21
176	1366	19,91	332,36	-0,08	-0,02	0,00
133	532	2252,60	726297,28	-0,85	-1,07	0,91
158	997	504,52	149947,67	-0,40	-0,49	0,20
137	683	1888,91	491724,59	-0,78	-0,88	0,68
128	672	2752,21	507272,67	-0,94	-0,89	0,84
131	754	2446,44	397190,82	-0,89	-0,79	0,70
294	3048	12890,98	2768128,05	2,03	2,09	4,24
240	1780	3544,83	156633,28	1,07	0,50	0,53
243	1970	3911,06	343125,59	1,12	0,74	0,82
263	2938	6812,60	2414198,82	1,48	1,95	2,88

Table A.6: Pearson Correlation for Example 6

$$\rho = 1 - \frac{6\sum d_i^2}{n(n^2-1)}$$

where d_i is the difference between the ranks of corresponding values of X and Y, and n is the number of pairs.

Example 7 Suppose that X and Y are given in the Table A.4. To calculate ρ we follow the following steps:

1. Order the value of X.
2. Ranking each value of X and Y.
3. Calculate the difference between rank X and the corresponding rank Y.
4. Applying the formula of ρ.

$$\sum d_i^2 = 38 \Rightarrow \rho = 0,895604$$

According to the calculated value of Spearman Correlation for X and Y, we can say that X and Y are correlated positively, which corresponds also to the Figure A.1.

Note In the ranking step, if we have two (or more) equal values of the variables, then the associated rank will be the average of the corresponding ranks (see Table A.8). The ranks that will be used in the calculation of Spearman are shown in the column "Rank X" in of the Table A.8.

X	**Y**	**Rank X**	**Rank Y**	d_i	d_i^2
128	672	1	2	-1	1
131	754	2	4	-2	4
133	532	3	1	2	4
134	1230	4	8	-4	16
137	683	5	3	2	4
152	1041	6	7	-1	1
157	984	7	5	2	4
158	997	8	6	2	4
176	1366	9	9	0	0
240	1780	10	10	0	0
243	1970	11	11	0	0
263	2938	12	12	0	0
294	3048	13	13	0	0

Table A.7: Calculating Spearman Correlation For Example 7

X	**Initial Rank X**	**Rank X**
128	1	1
131	2	2
133	3	3
134	4	4
137	5	$(5+6)/2 = 5.5$
137	6	$(5+6)/2 = 5.5$
152	7	7
157	8	8
158	9	9
176	10	$(10+11+12)/3 = 11$
176	11	$(10+11+12)/3 = 11$
176	12	$(10+11+12)/3 = 11$
240	13	13
243	14	14
263	15	15
294	16	16

Table A.8: Special Case for Spearman (Repeated values of X and/or Y)

A.4.3 Kendall Rank Correlation Coefficient

Kendall correlation τ is also a non-parametric method that evaluates the degree of similarity between two sets of objects ranks. The following formula shows how to calculate τ value:

$$\tau = \frac{4P}{n(n-1)} - 1$$

where n is the number of items, and P is the sum of items ranked after the given item by both rankings.

Example 8 Suppose that X and Y are given in the Table A.4. To calculate Kendall correlation we apply the following steps:

1. Order the value of X.
2. Ranking each value of X and Y.
3. Calculating P as sum of t_i, where t_i is the number of items that have rank higher than the rank value of ith item.
4. Applying the formula of τ.

X	1	2	3	4	5	6	7	8	9	10	11	12	13
Y	2	4	1	8	3	7	5	6	9	10	11	12	13
t_i	11	9	10	5	8	5	6	5	4	3	2	1	0

Table A.9: Ranking and Calculating P value in Kendall Example 8

$$P = \sum t_i = 69$$

$$\tau = \frac{4 * 69}{13(13-1)} - 1 = 0.769 \text{ where n=13}$$

Kendall τ has three versions, called τa, τb and τc, the first version does not make any adjustment for ties, while the second and the third versions do. τb is used when data tables are square (2-by-2 table) otherwise τc is used.

A.4.4 Conclusion

To analyze accurately the correlation between two variables, we should choose the more appropriate correlation method. Choosing the right method depends on some assumptions about the variables. Table A.10 represents a simple guide to choose the appropriate correlation method.

Correlation methods	Assumptions about the variables
Pearson Correlation	- A linear relationship between the both variables. - Both variables are interval. - Both variables are well approximated by normal distribution.
Spearman Corrlation	- Do not require a linear relationship between variables. - Do not require the variable to be interval. - Do not require any assumption about the frequency distribution.
Kendall Correlation	- Same as Spearman.

Table A.10: Correlation Methods

A.5 Statistical Hypothesis Testing

A hypothesis is either a suggested explanation for an observable phenomenon or a reasoned proposal predicting a possible causal correlation among multiple phenomena. If the hypothesis is stated in terms of population parameters such as mean and variance, the hypothesis is called a **statistical hypothesis** [54]. A **statistical hypothesis test** is a procedure that enables us to agree or disagree with the statistical hypothesis using sample data. For instance, a quality test organization carries out a test to evaluate the quality of an appliance if it is correspond to the standard quality.

Hypothesis testing enables one to quantify the degree of uncertainty in sampling variation, which may account for the results that deviate from the hypothesized values in a particular study. There are two types of hypothesis, first a research hypothesis that represents a general idea about the nature of the question in the population, while the goal of the second type, statistical hypothesis, is establishing the basis for tests of significance.

Statistical hypothesis testing starts by making a set of two statements about the parameter(s) in question. One statement must be true, while the other must be false. The first statement is called the **null hypothesis** and is denoted by H_0, and the second is called the **alternative hypothesis** and is denoted by H_1. The null hypothesis is a generally the opposite of the research hypothesis. For instance, H_0 assumes that there is no association between the predictor and the outcome variables in the population under study, while H_1 assumes the existence of this association.

Statistical hypothesis testing depends on calculating the probability of observing the difference between two values. This probability is called the **p-value**. According to the p-value, if it is low enough, one can conclude that H_0 is not true, and there is really a difference between the two values.

Several interpretation of p-value are often made:

- p-value is viewed as the probability that the results obtained were due to chance.
- 1-p is considered the reliability of the result; that is, the probability of getting the same result if the experiments were repeated.
- Alternatively p-value can be treated as the probability that the null hypothesis is true.

To establish a statistical hypothesis test, one should follow the fives steps:

1. Formulate the null hypothesis H_0, about some phenomenon or parameter, and the alternative hypothesis H_1.
2. Compute the statistical test for the given conditions.
3. Calculate the p-value.
4. Accept/reject the null hypothesis:
   - accept H_0: if the p-value is greater than a significance level (typically 0.05).
   - reject H_0: if the p-value is less than or equal a significance level (typically 0.05).
5. Interpret the results according to the hypotheses H_0 and H_1.

The results of a hypothesis test may be subject to two distinctly different errors. These errors called *Type I* and *Type II* errors. **Type I error** happens if H_0 is rejected incorrectly, that is, when H_0 is true, but for the sample (under study)-based inference procedure rejects it. **Type II error** happens if H_0 is incorrectly failed to be rejected, that is, if H_0 is not true, but inference procedure fails to detect this fact. Clearly, Type I and II errors cannot be commited at the same time, and we cannot know if we have commited one of these two errors.

Example 9 An elementary school has 300 students. The principal of the school thinks that the average IQ of students is at least 110. To prove her point, she administers an IQ test to 20 randomly selected students. Among the sampled students, the average IQ is 108 with a standard deviation of 10. Based on these results, one can test if the principal should accept or reject the original hypothesis. Let us consider the significance level of 0.01.

Step 1 Stating the null hypothesis and the alternative one:

$$\text{Null hypothesis } H_0\text{: } \mu >= 110$$

$$\text{Alternative hypothesis } H_1\text{: } \mu < 110$$

where μ is the hypothesized population mean of IQ.

The null hypothesis will be rejected if the sample mean is too small, according to the significance level.

Step 2 We then calculate the standard error SE:

$$SE = \frac{s}{\sqrt{n}} = \frac{10}{\sqrt{20}} = 2.236$$

where s is the given standard deviation and n is the sample size.

Step 3 We will apply the Student t-test t that corresponds to the Student distribution[1] (with the assumption the population is normally distributed):

$$t = \frac{(\bar{x} - \mu)}{SE} = \frac{(108 - 110)}{2.236} = -0.894$$

where $\bar{x}$ is the sample mean.

Step 4 Since the sample size is 20, the the degree of freedom is 20-1=19. We use the previous information to calculate the probability `P(t < -0.894)`. We used an online statistical table[2].

$$P(t < -0.894) = 0.19$$

$$\Rightarrow p - value = 0.19$$

Since obtained p-value is greater than the significance level, we cannot reject the null hypothesis.

A.6 Goodness of Fit Tests

One could have a sample data, and a function that supposed to describe the data. From hypothesis testing point of view, one could ask the question: does this function really provide an adequate description of the way the data behave? In other words, are there any

[1] Choosing the statistical test is done with respect to the population distribution, for instance if it is Fisher distribution, then the corresponding test is F-test.

[2] `http://stattrek.com/Tables/t.aspx`

significance and incompatible differences? To show that there are real differences between the data and the function, one has to prove that the probability of these differences is small, then the null hypothesis is rejected, the function and the date are supposed to disagree, and the fit is not "good". **Goodness of fit tests** are formal hypothesis tests that can be used to determine whether a set of observed values fit some specified distribution [78]. There are several tests could be used to do that, such as χ^2 and Wilcoxon tests.

A.6.1 Chi-square Test χ^2

The χ^2 is the squared difference between the observed values and their theoretical predications, suitably weighted by the errors of measurement.

$$\chi^2 = \sum_{i=1}^{N} (x_i^{observed} - x_i^{ideal})^2 / expected\ error$$

where the *expected error* is the standard deviation of x. The chi-square test is used if we have one categorical independent variable and one categorical dependent variable. For example, the independent variable might be gender (male and female) and the dependent variable whether or not employees are promoted within two years of appointment. The two variables are cast into a contingency Table A.11. The chi-square test tells us whether the variable on the row is independent of the variable on the column.

	Promoted	**Not promoted**	**Total**
Male	36	14	50
Female	30	25	55
Total	66	39	105

Table A.11: Chi-Square (male/ female) promoted example

To apply the formula of χ^2 we need to calculate the expected value for each cell. The expected value of a cell c_{ij} equals to the total value of the row i times the total value of the column j divided by the grand total. For example the expected value that corresponds to the value 36 is $50 \times 66/105 = 31.428$, in similar way we can calculate the other expected values for the others, Table A.12.

Now, we can get the value of χ^2, which represents the sum of the values in the left column:

$$\chi^2 = 0.665 + 1.125 + 0.604 + 1.023 = 3.417$$

$x^{observed}$	$x^{expected}$	$y = x^{observed} - x^{expected}$	$y^2/x^{expected}$
36	31.428	4.572	0.665
14	18.571	-4.571	1.125
30	34.571	4.571	0.604
25	20.428	4.572	1.023

Table A.12: Calculating Chi-Square for (male/female) promoted example

To interpret the calculated value of Chi-Square, we need to determine the degree of freedom, and we need a table of the critical values of the Chi-Square distribution. The degree of freedom *df* is given by the number of cells minus one. For our example *df = 3*. The Table A.13 is a partial table[1] of critical values of χ^2.

	Level of Significance				
df	0.10	0.05	0.025	0.01	0.001
1	2.706	3.841	5.024	6.635	10.828
2	4.605	5.991	7.378	9.210	13.816
3	6.251	7.815	9.348	11.345	16.266
4	7.779	9.488	11.143	13.277	18.467
5	9.236	11.070	12.833	15.086	20.515

Table A.13: Partial table of critical values of Chi-square Distribution

If we consider a significance level of 0.05, then our calculated χ^2 is smaller than the corresponding critical value 7.815 for a degree of freedom df=3. Therefore, there is no dependency between the row and column variables of Table A.11.

A.6.2 Wilcoxon Tests

There are two types of Wilcoxon tests: Wilcoxon Mann-Whitney Test and Wilcoxon Signed Ranks Test. **Wilcoxon Mann-Whitney Test** is a powerful non-parametric test, which is used for *comparing two populations*. It is used to test the null hypothesis that two populations have identical distribution functions against the alternative hypothesis that the two distribution functions differ only with respect to location (median). This test does not require the assumption that the differences between the two samples are normally

[1] Chi-Square table could be found on `http://www.itl.nist.gov/div898/handbook/eda/section3/eda3674.htm`

distributed [76]. **Wilcoxon Signed Ranks Test** is used to test that *a distribution is symmetric about hypothesized value.* In other words, it is designed to test a hypothesis about the location (median) of a population distribution. This test also does not require the assumption that the population is normally distributed.

Example 10 (Wilcoxon Signed Ranks Test): Let us suppose that we want to make a report that shows if the students are really interested in some topics. We will inquire 16 students to give a mark of 100 for two topics A and B that represents their interest in these two topics. Table A.14 shows their responses:

Student	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Topic A	78	24	64	45	64	52	30	50	64	50	78	22	84	40	90	72
Topic B	78	24	62	48	68	56	25	44	56	40	68	36	68	20	58	32

Table A.14: Data for Example 10

To apply Wilcoxon test, first we start by calculating the difference between X that represents the marks of topic A, and Y that represents the marks of topic B. Second, we calculate the absolute value of this difference. After calculating the difference, we rank[1] these resulting values (absolute and signed) Table A.15.

The next step is calculating the value of W for the Wilcoxon test, which is the sum of the signed ranks.

$$W = 1 - 2 - 3.5 - 3.5 + 5 + 6 + 7 + 8.5 + 8.5 - 10 + 11 + 12 + 13 = 67$$

Note: The sum of positive ranks is $+W = n(n+1)/2$, where n is the number of ranks without regarding the zero values, in the previous example n is 14. This sum represents the maximum possible positive value. While $-W = -1 \times n(n+1)/2$ is the minimum possible negative value.

In our example $+W= 105$ and $-W= -105$, therefore a preponderance of positive signs among the signed ranks would refer to rate the topic A as a higher probability than the topic B. Otherwise, if there is a preponderance of negative signs, that would refer that the topic A has a lower probability than the topic B. While in case of W *is close to 0* that refers the probability of each topic A and B is approximately equal, then there is no preference between these topics for the students. For our example, since there is a preponderance of positive signs, then the students prefer the topic A more than the topic B.

[1]If the difference between X and Y is zero, they are excluded from the ranking.

X	Y	X-Y	Abs(X-Y)	Rank absolute	Rank signed
78	78	0	0	-	-
24	24	0	0	-	-
64	62	-2	2	1	1
45	48	3	3	2	-2
64	68	4	4	3.5	-3.5
52	56	4	4	3.5	-3.5
30	25	-5	5	5	5
50	44	-6	6	6	6
64	56	-8	8	7	7
50	40	-10	10	8.5	8.5
78	68	-10	10	8.5	8.5
22	36	14	14	10	-10
84	68	-16	16	11	11
40	20	-20	20	12	12
90	58	-32	32	13	13
72	32	-40	40	14	14

Table A.15: Ranking data of Example 10

Example 11 (Wilcoxon Mann-Whitney Example): Let us consider a group of 21 students. We form two subgroups A of 11 students, and B of 10 students. A hypothesis is supposed about the subgroup A that its students are more active than the students of B. A teacher has examined the two subgroups, with giving a mark of 10 for each student. The Table A.16 represents the marks given by the teacher.

Subgroup A	4.6	4.7	4.9	5.1	5.2	5.5	5.8	6.1	6.5	6.5	7.2
Subgroup B	5.2	5.3	5.4	5.6	6.2	6.3	6.8	7.7	8.0	8.1	

Table A.16: Marks of students - Example 11

The average of marks of the subgroup A is 5.6, and the average of the subgroup B is 6.5. From regarding the average of each subgroup, one could say that the subgroup A does not reflect the assumption that A is more active than B. To test whether the calculated average difference is significant, we will apply Wilcoxon Mann-Whitney test. First we are going to rank the marks of all students, Table A.17.

Then we calculate the sum and the average of ranks of each subgroup Table A.18.

Student's Mark	4.6	4.7	4.9	5.1	5.2	5.2	5.3	5.4	5.5	5.6	5.8
Rank	1	2	3	4	5.5	5.5	7	8	9	10	11
Belongs to	A	A	A	A	A	B	B	B	A	B	A

Student's Mark	6.1	6.2	6.3	6.5	6.5	6.8	7.2	7.7	8.0	8.1
Rank	12	13	14	15.5	15.5	17	18	19	20	21
Belongs to	A	B	B	A	A	B	A	B	B	B

Table A.17: Ranking all students marks Example 11

The total sum of all ranks T_{AB} is *N(N+1)/2*, where N is the total number of students. Therefore T_{AB} is 231. We could also find the average of all ranks simply by the formula *(N+1)/2*. In our example the total average is 11.

	Ranks											$\sum$Ranks	Ranks Ave.
Ranks of A	1	2	3	4	5.5	9	11	12	15.5	15.5	18	$T_A = 96.5$	8.8
Ranks of B	5.5	7	8	10	13	14	17	19	20	21	-	$T_B = 134.5$	13.5

Table A.18: Sum and average of ranks of subgroup A and B

Let us assume that the null hypothesis for our example is, "there is no difference between the subgroup A and B". The null hypothesis could be true if the average of the A ranks and the B ranks approximate the overall average value of (N+1)/2=11. In other words the T_A and T_B (that are given in Table A.18) will approximate the values:

$$T_A^{'} = n_A(N+1)/2 = 11(21+1)/2 = 121 \text{ where } n_A \text{ is size of } A$$

$$T_B^{'} = n_B(N+1)/2 = 10(21+1)/2 = 110 \text{ where } n_B \text{ is size of } B$$

We calculate the standard deviation of T_A and T_B, which is given by:

$$\sigma_T = \sqrt{\frac{n_A n_B (N+1)}{12}} = \pm 14.2$$

Next step is calculating the standard score or what is called z-ratio. The standard score indicates how many standard deviations of an observation is above or below the average. Z-ratio is given by the following formula:

$$Z = \frac{(T-M) \pm 0.5}{\sigma_T}$$

where T is any of the observed values T_A or T_B and M is the corresponding average of T.

$$Z_A = \frac{(96.5 - 121) + 0.5}{14.2} = -1.69$$
$$Z_B = \frac{(134.5 - 110) - 0.5}{14.2} = +1.69$$

It is not by chance that the absolute value of Z_A and Z_B are equal. For all instance of Z_A and Z_B they will have the same absolute values. Therefore no difference if the test of statistical significance is done with Z_A or Z_B. For our example, since observed value T_A smaller than its null-hypothesis value of $T_A^{'}$ and the value of Z_A is negative, that will lead to the assumption that the subgroup B is more active than A.

A.7 Regression Analysis

Regression analysis is a statistical method for analyzing a relationship between two or more variables in such a manner that one variable can be predicted by using information on the others [6, 76, 115]. It could be applied to predict the future values of variables. The purpose of a regression analysis is to observe sample measurements taken on different variables, called independent variables (factors), and to examine the relationship between these variables and a dependent variable (response).

In the following we describe two regression methods that could be used to find a relationship between variables.

A.7.1 Simple Regression Model

A regression analysis starts with an estimate of the population mean(s) using a function, which explains the relationship between the independent variable(s) and the dependent variable. This function is called the regression function (regression model). In the simplest case, this function can be described geometrically by a line if there is only one independent variable or a multidimensional plane if there are several. In the simple case, the regression function (model) has the following form:

$$y = A + Bx + \epsilon$$

This form represents the linear regression model. Where y is the independent variable and x is the dependent one. A and B are called the regression coefficients. The ϵ represents the error term.

Example 12 Let us consider the data shown in the Table A.19, which represents the price of gas per gallon and the quantity sold over a period of 15 years:

Year	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Price	1.62	1.667	1.69	1.70	1.72	1.73	1.736	1.74	1.75	1.755	1.756	1.77	1.767	1.756	1.77
Quantity	159	160	163	166	167	167	168	167	167.9	168.9	169	169	170	171	172

Table A.19: Price of gas per gallon

We want to represent these data as a regression model. For simplifying we will consider a linear form $y = A + Bx$. Now we have to find the value of A and B. For that purpose we use the *Least Square method*. The idea behind this method is minimizing the error of prediction. A and B are calculated from the formula:

$$B = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^{n}(x_i - \bar{x})^2}$$

$$A = \bar{y} - B\bar{x}$$

Year	**x=Price**	**y=Quantity**	$(x - \bar{x})$	$(y - \bar{y})$	$(x - \bar{x}) * (y - \bar{y})$	$(x - \bar{x})^2$
1	1,62	159	-0,10847	-7,98667	0,86629	0,01177
2	1,67	160	-0,06147	-6,98667	0,42945	0,00378
3	1,69	163	-0,03847	-3,98667	0,15335	0,00148
4	1,7	166	-0,02847	-0,98667	0,02809	0,00081
5	1,72	167	-0,00847	0,01333	-0,00011	0,00007
6	1,73	167	0,00153	0,01333	0,00002	0,00000
7	1,74	168	0,00753	1,01333	0,00763	0,00006
8	1,74	167	0,01153	0,01333	0,00015	0,00013
9	1,75	167,9	0,02153	0,91333	0,01967	0,00046
10	1,76	168,9	0,02653	1,91333	0,05077	0,00070
11	1,76	169	0,02753	2,01333	0,05543	0,00076
12	1,77	169	0,04153	2,01333	0,08362	0,00173
13	1,77	170	0,03853	3,01333	0,11611	0,00148
14	1,76	171	0,02753	4,01333	0,11050	0,00076
15	1,77	172	0,04153	5,01333	0,20822	0,00173

Table A.20: Calculating A and B values for simple regression - Example 12

According to the Table A.20 the A and B could be calculated:

$$\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y}) = 2,12919$$

$$\sum_{i=1}^{n}(x_i - \bar{x})^2 = 0,02572$$

then:

$$B = 2,12919/0,02572 = 82,7834$$

$$A = 166,9866 - (82,7834 * 1,72847) = 23,8979$$

So the regression model (function) is:

$$y = A + Bx = 23,8979 + 82,7834x$$

Form this regression model, one can predict future values of y that represent the quantity of gas.

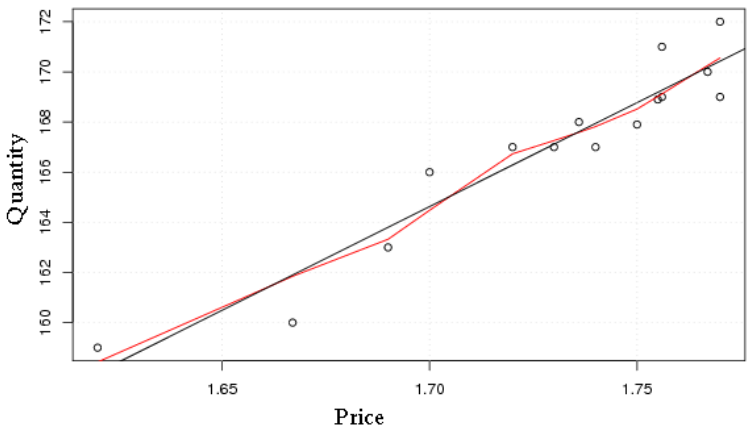


Figure A.4: Simple regression for data given in Table A.20

A.7.2 Multiple Regression Model

In the previous section, we saw how to find the regression model (function) using the simple form $y = A + Bx$. Where the dependent variable y depends only on one variable x, which is not always the case. In general, the dependent variable could have dependency on other variables, and here the regression model will have the following form:

$$y = A + \sum_{i=1}^{p} B_i X_i$$

Finding the constants B_i requires a sample of n observations of X where each observation satisfies the regression model:

$$y_i = \beta_0 + \sum_{i=p}^{i=1} \beta_i x_i + u_i$$

where u_i is the error term, and β_i are constants.

The constant β_i can be calculated using the least squares procedure, which minimizes the sum of squares of errors:

$$S = \sum_{i=1}^{i=n} u_i^2 = \sum_{i=1}^{i=n} (y_i - \beta_0 - \beta_1 x_1 - \beta_2 x_2 - \ \ldots \ - \beta_p x_p)^2$$

This minimization of the sum of squares leads to the following equations, from which the values of β_i can be found:

$$\sum_{i=1}^{i=n} y_i = n\beta_0 + \beta_1 \sum_{i=1}^{i=n} x_{i1} + \beta_2 \sum_{i=1}^{i=n} x_{i2} + \ \ldots \ + \beta_p \sum_{i=1}^{i=n} x_{ip}$$

$$\sum_{i=1}^{i=n} x_{i1} y_i = \beta_0 \sum_{i=1}^{i=n} x_{i1} + \beta_1 \sum_{i=1}^{i=n} x_{i1}^2 + \beta_2 \sum_{i=1}^{i=n} x_{i1} x_{i2} + \ \ldots \ + \beta_p \sum_{i=1}^{i=n} x_{i1} x_{ip}$$

$$\sum_{i=1}^{i=n} x_{i2} y_i = \beta_0 \sum_{i=1}^{i=n} x_{i2} + \beta_1 \sum_{i=1}^{i=n} x_{i1} x_{i2} + \beta_2 \sum_{i=1}^{i=n} x_{i2}^i + \ \ldots \ + \beta_p \sum_{i=1}^{i=n} x_{i2} x_{ip}$$

$$\ldots.$$

$$\sum_{i=1}^{i=n} x_{ip} y_i = \beta_0 \sum_{i=1}^{i=n} x_{ip} + \beta_1 \sum_{i=1}^{i=n} x_{i1} x_{ip} + \beta_2 \sum_{i=1}^{i=n} x_{i2} x_{ip} + \ \ldots \ + \beta_p \sum_{i=1}^{i=n} x_{ip}^2$$

There are various tools that can calculate these constant such as *TANAGRA*[1].

A.8 Conclusion

In this appendix we have presented several basic concepts in statistics such as representing data using graphics and table, measuring the correlation between variables using different methods, in addition to an overview on the steps of leading a statistical hypotheses test. The topics presented here present common concepts that are used in several empirical studies. Therefore, we used them also during the analysis study that we carried out.

[1]TANAGRA is a free data mining software, that has several methods for exploratory data analysis. It can be downloaded from `http://eric.univ-lyon2.fr/~ricco/tanagra/en/tanagra.html`

Appendix B

Metrics Calculator

Metrics Calculator is a simple tool that we developed to calculate some metrics, among which the adapted metrics that we introduced in this thesis. It can calculate the following metrics:

- DIT, depth of inheritance tree.

$$DIT(C) = |Ancestors(C)|$$

- DIT$_A$, depth of inheritance tree restricted to the application tree.

$$DIT_A(C) = |Ancestors(C) \backslash JC|$$

  where JC is the set of all `Java` classes

- WMC, weighted method per class.

$$WMC(C) = \sum_{i=1}^{n} c_i$$

  where c_i is the cyclomatic complexity of the method i in the class C.

- WMH, weighted method per class for the entire hierarchy.

$$WMH(C) = WMC(C) + \sum_{C_i \in Ancestors(C)} WMC(C_i)$$

- WMH$_A$, weighted method per class for the application hierarchy (does not include the `Java` classes such as *JDK*).

$$WMH_A(C) = WMC(C) + \sum_{c_i \in Ancestors(C) \backslash JC} WMC(c_i)$$

- NOM, number of methods including the class's constructors.

$$NOM = |M_D(C)|$$

where $M_D(C)$ is the set of methods which are defined in the class C.

- NOH, number of inherited methods.

$$NOH = |M_{In}|$$

where M_{In} is the set of all inherited methods.

- NOH$_A$, number of inherited methods of application classes.

$$NOH_A = |M_{IA}|$$

where M_{IA} is the set of all inherited methods except the inherited ones from `Java` classes.

- RFC, response for class.

$$RFC = |RS| \quad where \quad RS = \{M\} \cup_{\forall i} \{R_i\}$$

where $\{R_i\}$ is the set of methods called by method i, and $\{M\}$ is a set of all methods in the class.

- NOC, number of immediate children.

$$NOC(C) = |Descendants(C)|$$

- CC, McCabe cyclomatic complexity of a method.

$$CC = number\ of\ decision\ statements + 1$$

How to use

To use *Metrics Calculator*, we need to specify the path of the application that we want to analyze, see Figure B.1. If the specified application requires some jars library to be run, we can specify them from the menu *Settings>Class path*, see Figure B.2. The *Settings* window allows the use to add single, multiple or folder path. If repeated paths are common to be used in the analysis, they can be save to a text file and be loaded later from the same *Settings* window.

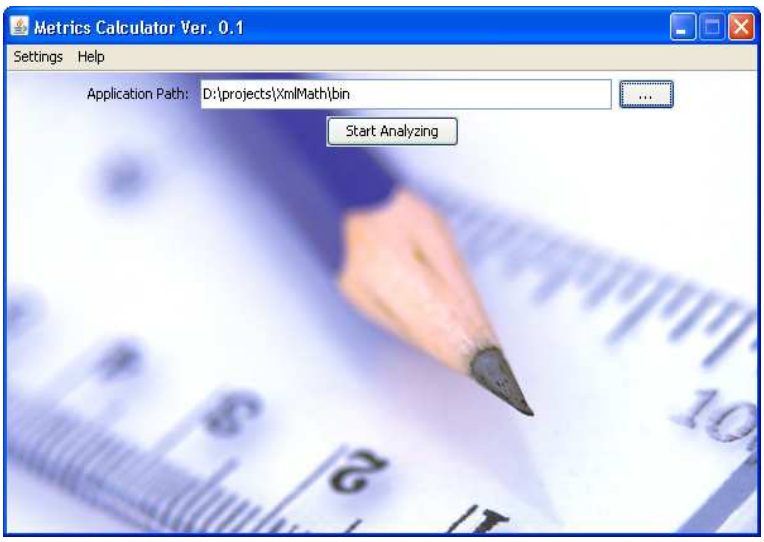


Figure B.1: Metrics Calculator Main Window

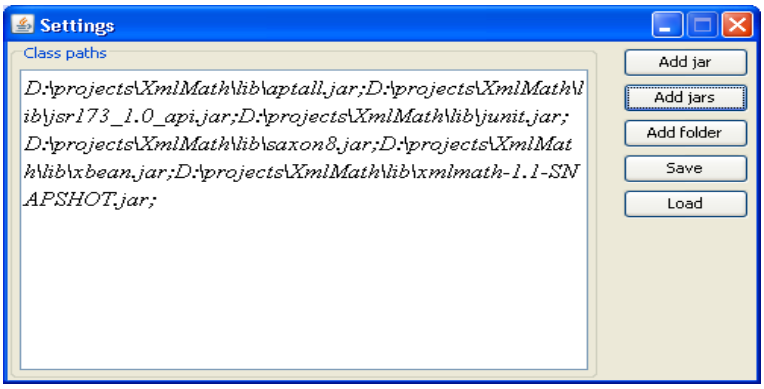


Figure B.2: Adding Required Libraries

After specifying the application path and its required libraries, we simply click on the button "Start Analyzing" to run the analysis process. Once the analysis process finishes a result window appears, see Figure B.3. The result window shows each class of the application classes the calculated metrics values in the same row.

In order to Visualize the cyclomatic complexity of any class, it is sufficient to click on the class name to list all class's methods including the constructors and their associated complexity.

Some tools, such *Metrics* plugin for eclipse, assign a unity to any methods if they are defined in an interface, defined as native or even as abstract method. Actually, the method in such cases does not have any implementation, therefore assigning a value of 1 to it may not be accurate. For that we have chosen to assign zero value to such methods.

Metrics Calculator Results

Class Name	DIT	DIT_A	WMC	WMH	WMH_A	NOM	NOH	NOH_A	RFC	NOC
org.xmlmath.operands.AbstractBooleanValue	2	1	2	22	9	3	16	4	5	11
org.xmlmath.operands.AbstractDoubleValue	3	2	2	24	11	3	19	7	5	14
org.xmlmath.operands.AbstractListValue	2	1	2	22	9	3	16	4	5	6
org.xmlmath.operands.AbstractLongValue	3	2	2	24	11	3	19	7	5	9
org.xmlmath.operands.AbstractNumberValue	2	1	2	22	9	3	16	4	5	10
org.xmlmath.operands.AbstractStringValue	2	1	2	22	9	3	16	4	5	6
org.xmlmath.operands.AbstractValue	1	0	9	13	0	4	12	0	53	7
org.xmlmath.operands.Add	3	2	8	24	11	3	19	7	20	0
org.xmlmath.operands.And	3	2	6	24	11	3	19	7	17	0
org.xmlmath.operands.Cos	4	3	3	26	13	3	22	10	10	0
org.xmlmath.operands.Count	4	3	3	26	13	3	22	10	9	0
org.xmlmath.operands.Declaration	1	0	5	13	0	5	12	0	12	1
org.xmlmath.operands.DivideByZeroException	5	1	2	54	4	2	39	4	4	0
org.xmlmath.operands.DoubleValue	4	3	3	26	13	3	22	10	6	0

Method Name	Cyclomatic Complexity
<init>	1
getName	1
getValue	1
build	1
parse	1

Figure B.3: Result Window of Metrics Calculator

Additionally, we differentiate among four levels of RFC metric:

- RFC_0: includes all kinds of call, even the ones that do not appear explicitly in the source code.
- RFC_1: excludes some special calls such as toString(), append() that could appear in both byte and source code.
- RFC_2: excludes the calls of constructors, even the ones that do not appear explicitly in the source code.
- RFC_3: excludes both calls of level one and two.

Choosing the required level for RFC is done from the menu *Settings*.

Glossary

AIF Attribute Inheritance Factor

Bounce-C number of yo-yo paths visible to CUT

Bounce-S number of yo-yo paths in SUT

Ca & Ce Afferent/Efferent Coupling

CBO Coupling

CC Cyclomatic Complexity

COF Coupling Factor

Ct Controlability

DIT Depth of Inheritance Tree

DIT$_A$ Depth of Inheritance Tree restricted to Application

DRR Domain Range Ratio

DYN percent of Dynamic Calls

EF Encapsulation Factor

EPIE Extened PIE

FDFR Fixed Domain / Fixed Range

FIN Fan-In

ICH Information CoHesion

KSLOC same as SLOC but K stands for Kilo

LCC Losse Class Cohesion

LCOM Lack Of Cohesion of a Method

LOC number of Lines Of Code

MC/DC Modified Condition / Decision Coverage

MCC Most Cohesive Component

MIF Method Inheritance Factor

MMC Mean Method Complexity

MPC Message Passing Coupling

NOC Number Of Children

NOH Number Of InHerited methods

NOH$_A$ Number Of Application inHerited methods

NOM Number Of Methods

NOR Number Of Root Classes

NOTC Number Of Test Cases

NSBC Number oof Stubs needed to Break Cycles

NTM Number of Trivial Methods

Ob Observability

OVR percent of non-Overloaded Calls

PAP Percent of Public and Protected attributes

PF/POF Polymorphism Factor

PIE Propogation Infection Execution

RCI Ratio of Cohesive Interactions

RCO Ratio of Component Observability

RFC Response For Class

SCCp Self-Completeness of Component's parameter

SCCr Self-Completeness of Component's Return value

SDMC Standard Deviation Method Complexity

SLOC number of Source Lines Of Code

STA System Testability

TCC Tight Class Cohesion

VC Visibility Component

VDFR Variable Domain / Fixed Range

VDVR Variable Domain / Variable Range

WMC Weighted Methods per Class

WMH Weighted Methods per Class for the Hierarchy

$\mathbf{WMH}_A$ Weighted Methods per Class for the Hierarchy restricted to the application

Bibliography

[1] JUnit. http://www.junit.org. 61

[2] IEEE standard for a software quality metrics methodology. *IEEE Std 1061-1992*, 12 Mar 1993. 50, 54, 55

[3] IEEE standard Glossary of Software Engineering Terminology. *IEEE Std 610.12-1990*, Dec 1990. 25, 44, 65

[4] R. Kent A. Kent, J. G. Williams. *Encyclopedia of microcomputers*. CRC Press, 1991. 24

[5] A. A. Abdul Ghani, K. T. Wei, G. M. Muketha, and W. P. Wen. Complexity metrics for measuring the understandability and maintainability of business process models using goal-question-metric (GQM). *IJCSNS International Journal of Computer Science and Network Security*, 8(5):219–225, 2008. 36

[6] A. Agresti. *An Introduction to Categorical Data Analysis*. Wiley, 1996. 135

[7] R. T. Alexander and A. J. Offutt. Criteria for testing polymorphic relationships. *Software Reliability Engineering, International Symposium on*, 0:15, 2000. 31

[8] P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008. 19, 28, 31

[9] J.M. Armstrong and R.J. Mitchell. Uses and abuses of inheritance. *Software Engineering Journal*, 9(1):19–26, january 1994. 40, 60

[10] L.M. Pickard B.A. Kitchenham and S.J. Linkman. An evaluation of some design metrics. *Softw. Eng. J.*, 5(1):50–58, 1990. 50

[11] R. Bache and M. Mullerburg. Measures of testability as a basis for quality assurance. *Software Engineering Journal*, 5(2):86–92, 1990. 36, 51

[12] J. Bainbridge. Defining testability metrics axiomatically. *Softw. Test., Verif. Reliab.*, 4(2):63–80, 1994. 36

[13] V. R. Basili, L. C. Briand, and W. L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Trans. Software Eng.*, 22(10):751–761, 1996. 56, 60

[14] B. Baudry, Y. Le Traon, G. Sunyé, and J.-M. Jézéquel. Measuring and improving design patterns testability. In *9th IEEE International Software Metrics Symposium (METRICS 2003)*, pages 50–, Sydney, Australia, September 2003. 103, 104, 105

[15] B. Baudry and Y. Le Traon. Measuring design testability of a uml class diagram. *Information & Software Technology*, 47(13):859–879, 2005. 104

[16] B. Baudry, Y. Le Traon, and G. Sunyé. Testability analysis of a uml class diagram. In *8th IEEE International Software Metrics Symposium (METRICS 2002)*, pages 54–, Ottawa, Canada, June 2002. 103

[17] B. Beizer. *Software Testing Techniques.* Van Nostrand Reinhold, 1990. 19, 67

[18] R. G. (Ben) Bennitts. Progress in design for test: A personal view. *IEEE Design & Test of Computers*, 11(1):53–59, 1994. 25, 65

[19] A. Bertolino and L. Strigini. On the use of testability measures for dependability assessment. *IEEE Trans. Software Eng.*, 22(2):97–108, 1996. 47, 105

[20] J. M. Bieman and B.-K. Kang. Cohesion and reuse in an object-oriented system. In *SSR*, pages 259–262, 1995. 42

[21] R. V. Binder. Design for testability in object-oriented systems. *Communications of the ACM*, 37(9):87–101, September 1994. 19, 24, 25, 26, 30, 31, 33, 37, 38, 40, 42, 43, 44, 46, 51, 56, 57, 59, 67, 103

[22] R. V. Binder. Testing object-oriented software: A survey. *Softw. Test., Verif. Reliab.*, 6(3/4):125–252, 1996. 31

[23] R. V. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools.* The Addison-Wesley Object Technology Series, 1999. 30, 31, 34, 40, 59, 63

[24] A. B. Binkley and S. R. Schach. Validation of the coupling dependency metric as a predictor of run-time failures and maintenance measures. In *In Proceedings of the 20 th International Conference on Software Engineering*, pages 452–455, 1998. 60

[25] L. C. Briand, J. W. Daly, and J. Wüst. A unified framework for cohesion measurement in object-oriented systems. *Empirical Software Engineering*, 3(1):65–117, 1998. 42

[26] L. C. Briand, J. W. Daly, and J. Wüst. A unified framework for coupling measurement in object-oriented systems. *IEEE Trans. Software Eng.*, 25(1):91–121, 1999. 39

[27] L. C. Briand, S. Morasca, and V. R. Basili. Defining and validating high-level design metrics. Technical report, College Park, MD, USA, 1994. 42

[28] L. C. Briand, J. Wüst, J. W. Daly, and D. V. Porter. Exploring the relationships between design measures and software quality in object-oriented systems. *Journal of Systems and Software*, 51(3):245–273, 2000. 56, 60

[29] L. C. Briand, J. Wüst, J. W. Daly, and V. Porter. A comprehensive empirical validation of design measures for object-oriented systems. In *5th IEEE International Software Metrics Symposium (METRICS 1998)*, pages 246–257, Bethesda, Maryland, USA, March 1998. 60

[30] L. C. Briand, J. Wüst, S. V. Ikonomovski, and H. Lounis. Investigating quality factors in object-oriented designs: An industrial case study. In *ICSE*, pages 345–354, 1999. 56, 60

[31] L. C. Briand, J. Wüst, and H. Lounis. Replicated case studies for investigating quality factors in object-oriented designs. *Empirical Software Engineering*, 6(1):11–58, 2001. 56, 60

[32] D.B. Brown, S. Maghsoodloo, and W.H. Deason. A cost model for determining the optimal number of software test cases. *IEEE Transactions on Software Engineering*, 15(2):218–221, 1989. 19

[33] M. Bruntink and A. van Deursen. An empirical study into class testability. *Journal of Systems and Software*, 79(9):1219–1232, September 2006. 57, 60, 63

[34] T. J. Cheatham and L. Mellinger. Testing object-oriented software systems. In *ACM Conference on Computer Science*, pages 161–165, 1990. 40, 59

[35] S. R. Chidamber and C. F. Kemerer. Towards a metrics suite for object oriented design. In *OOPSLA*, pages 197–211, 1991. 103

[36] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Software Eng.*, 20(6):476–493, 1994. 37, 38, 40, 41, 43, 51, 59, 60, 62, 68, 103

[37] C.-M. Chung, T. K. Shih, C.-C. Wang, and M.-C. Lee. Integration object-oriented software testing and metrics. *International Journal of Software Engineering and Knowledge Engineering (IJSEKE)*, 7(1):125–144, 1997. 40, 51

[38] D. J. Colwell and J. R. Gillett. Spearman versus kendall. *The Mathematical Gazette*, 66(438):307–309, 1982. 75

[39] S. D. Conte, H. E. Dunsmore, and V. Y. Shen. *Software engineering metrics and models*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1986. 36

[40] B. J. Cox. The need for specification and testing languages. *Journal of Object-Oriented Programming*, 1(2):44–47, 1988. 31

[41] B. du Bois, S. Demeyer, and J. Verelst. Refactoring-improving coupling and cohesion of existing code. *Reverse Engineering, Working Conference on*, 0:144–151, 2004. 30

[42] F. Brito e Abreu and R. Carapua. Object-oriented software engineering: Measuring and controlling the development process. In *4th International Conference on Software Quality (ASQC)*, McLean, VA, USA, October 1994. 38, 43

[43] F. Brito e Abreu and R. Carapuça. Candidate metrics for object-oriented software within a taxonomy framework. *Journal of Systems and Software*, 26(1):87–96, 1994. 60

[44] F. Brito e Abreu and W. L. Melo. Evaluating the impact of object-oriented design on software quality. In *IEEE METRICS*, pages 90–99, 1996. 44, 60

[45] L. O. Ejiogu. Five principles for the formal validation of models of software metrics. *SIGPLAN Not.*, 28(8):67–76, 1993. 53

[46] K. El Emam, S. Benlarbi, N. Goel, and S. N. Rai. The confounding effect of class size on the validity of object-oriented metrics. *IEEE Trans. Software Eng.*, 27(7):630–650, 2001. 56, 60, 103

[47] M. Goulão F. Brito e Abreu and R. Esteves. Toward the design quality evaluation of object-oriented software systems. pages 44–57, October 1995. 38

[48] N. Fenton. Software measurement: A necessary scientific basis. *IEEE Trans. Softw. Eng.*, 20(3):199–206, 1994. 52

[49] N. E. Fenton. When a software measure is not a measure. *Software Engineering Journal*, 7(5):357–62, September 1992. 52, 54

[50] S. P. Fiedler. Object-oriented unit testing. *Hewlett-Packard Journal*, 40(2):69–75, April 1989. 40, 59

[51] International Organization for Standardization. ISO 9126: Information technology software product evaluation quality characteristics and guidelines for their use. Technical report, ISO, Geneva, 1991. 25

[52] P. G. Frankl and E. J. Weyuker. An applicable family of data flow testing criteria. *IEEE Trans. Software Eng.*, 14(10):1483–1498, 1988. 34

[53] R. S. Freedman. Testability of software components. *IEEE Trans. Software Eng.*, 17(6):553–564, 1991. 28, 45

[54] R. J. Freund and W. J. Wilson. *Statistical Methods.* Academic Press, 2003. 127

[55] D. Gu, Y. Zhong, and S. Ali. On testing of classes in object-oriented programs. In *Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*, page 22, Toronto, Ontario, Canada, October 1994. IBM. 34

[56] S. C. Gupta and M. K. Sinha. Improving software testability by observability and controllabliity measures. In *IFIP 13th World Computer Congress*, pages 147–154, September 1994. 103

[57] T. Gyimóthy, R. Ferenc, and I. Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Trans. Software Eng.*, 31(10):897–910, 2005. 56, 60

[58] R. Harrison, S. J. Counsell, and R. V. Nithi. An evaluation of the mood set of object-oriented software metrics. *IEEE Trans. Softw. Eng.*, 24(6):491–496, 1998. 50

[59] M.J. Harrold, J.D. McGregor, and K.J. Fitzpatrick. Incremental testing of object-oriented class structures. In *International Conference on Software Engineering (ICSE)*, May 1992. 40, 59, 62

[60] B. Henderson-Sellers. *Object Oriented Metrics: Measures of complexity.* Prentice Hall, 1996. 34, 56

[61] S. Henry and D. Kafura. Software structure metrics based on information flow. *Software Engineering, IEEE Transactions on*, SE-7(5):510–518, Sept. 1981. 38, 51

[62] B. Hetzel. *The complete guide to software testing (2nd ed.).* QED Information Sciences, Inc., Wellesley, MA, USA, 1988. 17

[63] A. Iannino, J. D. Musa, K. Okumoto, and B. Littlewood. Criteria for software reliability model comparisons. *SIGSOFT Softw. Eng. Notes*, 8(3):12–16, 1983. 53

[64] C. Izurieta and J. M. Bieman. Testing consequences of grime buildup in object oriented design patterns. In *ICST '08: Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation*, pages 171–179, Washington, DC, USA, 2008. IEEE Computer Society. 104

[65] C. Y. Huang J. R. Chang and T. H. Tsai. Software testability analysis using extended pie method. In *ISSRE: CD-ROM Proceedings of the 18th IEEE International Symposium on Software Reliability Engineering (Student Travel Grant Award)*, Trollhättan, Sweden, Nov. 2007. 48

[66] Z. Jin and A. Jefferson Offutt. Coupling-based integration testing. In *ICECCS*, pages 10–17, 1996. 30

[67] Z. Jin and A. Jefferson Offutt. Coupling-based criteria for integration testing. *Softw. Test., Verif. Reliab.*, 8(3):133–154, 1998. 38

[68] P. C. Jorgensen. *Software Testing: A Craftsman's Approach.* Auerbach Publications, 2008. 31

[69] S. Jungmayr. Design for testability. In *Proceedings of CONQUEST 2002.*, pages 57–64, September 2002. 30

[70] S. Jungmayr. Identifying test-critical dependencies. In *International Conference on Software Maintenance (ICSM)*, pages 404–413, Montreal, Quebec, Canada, 2002. IEEE Computer Society. 30, 39, 51, 103, 104, 105

[71] C. Kaner and W. P. Bond. Software engineering metrics: What do they measure and how do we know? In *10th IEEE International Software Metrics Symposium (METRICS 2004)*, Chicago, USA, September 2004. IEEE. 50, 52

[72] K. Karoui, A. Ghedamsi, and R. Dssouli. A study of some influencing factors in testability and diagnostics based on fsms. In *ISCC*, pages 109–115. IEEE Computer Society, 1999. 29

[73] B. Kitchenham, S. L. Pfleeger, and N. E. Fenton. Towards a framework for software measurement validation. *IEEE Trans. Software Eng.*, 21(12):929–943, 1995. 50

[74] D. C. Kung, J. Gao, P. Hsia, Y. Toyoshima, and C. Chen. A test strategy for object-oriented programs. In *19th International Computer Software and Applications Conference (COMPSAC'95)*, pages 239–244, Dallas, Texas, USA, August 1995. IEEE Computer Society. 39

[75] Y. Lee, B. Liang, S. Wu, and F. Wang. Measuring the coupling and cohesion of an object-oriented program based on information flow. In *International Conference on software quality (ICSQ'95)*, pages 81–90, Maribor, Slovenia, November 1995. 42

[76] E.L. Lehmann and Romano J. P. *Testing Statistical Hypotheses.* Springer, 2005. 132, 135

[77] J.-C. Lin and Y.-L. Huang. A new method for estimating the testability of polymorphism in class hierarchy. *Int. Computer Symposium*, Dec 1998. 44

[78] J.K. Lindsey. *Introduction to applied statistics a modelling approach.* Oxford, 2004. 115, 116, 130

[79] B.W.N. Lo and Haifeng Shi. A preliminary testability model for object-oriented software. *Software Engineering: Education and Practice, 1998. Proceedings. 1998 International Conference*, pages 330–337, 26-29 Jan 1998. 48

[80] T. J. McCabe. A complexity measure. *IEEE Trans. Software Eng.*, 2(4):308–320, 1976. 36, 51, 67

[81] T. J. McCabe and C. W. Butler. Design complexity measurement and testing. *Communication of the ACM*, 32(12):1415–1425, 1989. 36, 51

[82] J. D. McGregor and S. Srinivas. A measure of testing effort. In *Second USENIX Conference on Object-Oriented Technologies (COOTS)*, 1996. 49, 103

[83] M. G. Mendonça and V. R. Basili. Validation of an approach for improving existing measurement frameworks. *IEEE Trans. Softw. Eng.*, 26(6):484–499, 2000. 50

[84] T. Menzies, J. S. Di Stefano, and M. Chapman. Learning early lifecycle IV&V quality indicators. In *METRICS '03: Proceedings of the 9th International Symposium on Software Metrics*, page 88, Washington, DC, USA, 2003. IEEE Computer Society. 41

[85] J. Michura and M.A.M. Capretz. Metrics suite for class complexity. In *International Conference on Information Technology: Coding and Computing (ITCC'05)*, volume 2, April 2005. 41

[86] V. B. Misic. Cohesion is structural, coherence is functional: Different views, different measures. In *IEEE METRICS*, pages 135–, 2001. 31

[87] L. J. Morell. A theory of fault-based testing. *IEEE Trans. Software Eng.*, 16(8):844–857, 1990. 47

[88] S. Mouchawrab, L. C. Briand, and Y. Labiche. A measurement framework for object-oriented software testability. *Information & Software Technology*, 47(15):979–997, 2005. 57

[89] G. Myers. *The Art Of Software Testing.* Wiley-Interscience, 1979. 17, 34, 47, 67

[90] A. Nadeem and R. Lyu Michael. A framework for inheritance testing from VDM++ specifications. In *12th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC 2006)*, pages 81–88, California, Riverside, USA, December 2006. IEEE Computer Society. 31

[91] S. Ntafos. A Comparison of Some Structural Testing Strategies. *IEEE Transactions on Software Engineering*, pages 868–874, june 1988. 34

[92] S. Ntafos. On random and partition testing. In *Proceedings of ACM SIGSOFT international symposium on Software testing and analysis*, pages 42–48. ACM Press, 1998. 46

[93] H. M. Olague, S. Gholston, and S. Quattlebaum. Empirical validation of three software metrics suites to predict fault-proneness of object-oriented classes developed using highly iterative or agile software development processes. *IEEE Trans. Softw. Eng.*, 33(6):402–419, 2007. Senior Member-Etzkorn,, Letha H. 60

[94] M. Page-Jones. *The practical guide to structured systems design: 2nd edition.* Yourdon Press, Upper Saddle River, NJ, USA, 1988. 30, 38

[95] D. E. Perry and G. E. Kaiser. Adequate testing and object-oriented programming. *Journal of Object Oriented Programming*, 2(5):13–19, 1990. 31, 40

[96] B. Pettichord. Design for testability. In *Pacific Northwest Software Quality Conference (PNSQC)*, Portland, Oregon, 2002. 28

[97] R. E. Prather. An axiomatic theory of software complexity measure. *The Computer Journal*, 27(4):340–347, Nov 1984. 53

[98] Rani and R.B. Misra. On determining the software testing cost to assure desired field reliability. *India Annual Conference, 2004. Proceedings of the IEEE INDICON 2004. First*, pages 517–520, Dec. 2004. 19

[99] R. Reißing. Towards a model for object-oriented design measurement. In *In ECOOP Workshop on Quantative Approaches in Object-Oriented Software Engineering*, pages 71–84, 2001. 39

[100] A. Rountev, A. Milanova, and B. G. Ryder. Fragment class analysis for testing of polymorphism in java software. *IEEE Trans. Software Eng.*, 30(6):372–387, 2004. 31

[101] H. E. Dunsmore S. D. Conte and V. Y. Shen. *Software engineering metrics and models.* Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1986. 53

[102] S. Saini and M. Aggarwal. Enhancing mood metrics using encapsulation. In *ICAI'07: Proceedings of the 8th Conference on 8th WSEAS International Conference on Automation and Information*, pages 252–257, Stevens Point, Wisconsin, USA, 2007. World Scientific and Engineering Academy and Society (WSEAS). 44

[103] M. De Scheemaecker. About nanoxml, 2007. `http://nanoxml.cyberelf.be`. 63

[104] N.F. Schneidewind. Methodology for validating software metrics. *Software Engineering, IEEE Transactions on*, 18(5):410–422, May 1992. 50, 53

[105] M. Schroeder. A practical guide to object-oriented metrics. *IT Professional*, 1:30–36, Nov.-Dec. 1999. 39

[106] H. Seok Chae and Y. Rae Kwon. A cohesion measure for classes in object-oriented systems. In *5th IEEE International Software Metrics Symposium (METRICS)*, pages 158–166, Bethesda, Maryland, USA, March 1998. IEEE Computer Society. 42

[107] M.-R. Shaheen and L. du Bousquet. Quantitative analysis of testability antipatterns on open source java applications. In *12th ECOOP Workshop on Quantitative Approaches on Object Oriented Software Engineering*, 2008. 112

[108] M.-R. Shaheen and L. du Bousquet. Relation between depth of inheritance tree and number of methods to test. In *1st International Conference on Software Testing, Verification and Validation*, Lillehammer, Norway, April 2008. IEEE. 40, 57

[109] M.-R. Shaheen and L. du Bousquet. Analysis of the introduction of testability antipatterns during the development process. In *Fourth International Conference on Software Engineering Advances ICSEA*, Porto, Portugal, September 2009. IEEE. 112

[110] M.-R. Shaheen and L. du Bousquet. Is depth of inheritance tree a good cost prediction for branch coverage testing? In *First International Conference on Advances in System Testing and Validation Lifecycle VALID*, Porto, Portugal, September 2009. IEEE. 91

[111] J.W. Sheppard and M. Kaufman. Formal specification of testability metrics in ieee p1522. *AUTOTESTCON Proceedings, 2001. IEEE Systems Readiness Technology Conference*, pages 71–82, 2001. 52

[112] I. Sommerville. *Software Engineering*. Addison Wesley, 2004. 105

[113] W. Stevens, G. Myers, and L. Constantine. Structured design. *Classics in software engineering*, pages 205–232, 1979. 30, 31

[114] M.-H. Tang, M.-H. Kao, and M.-H. Chen. An empirical study on object-oriented metrics. In *6th IEEE International Software Metrics Symposium (METRICS'99)*, pages 242–249, Boca Raton, FL, USA, November 1999. IEEE Computer Society. 56, 60

[115] P. Tassi. *Méthodes Statistiques*. Economica, 2004. 135

[116] D. A. Troy and S. H. Zweben. Measuring the quality of structured designs. *Software engineering metrics I: measures and validations*, pages 214–226, 1993. 30, 38

[117] J. M. Voas. Factors that affect software testability. Technical report, 1991. 29

[118] J. M. Voas and K. W. Miller. Software testability: The new verification. *IEEE Software*, 12(3):17–28, 1995. 25

[119] J. M. Voas, J. Payne, R. Mills, and J. McManus. Software testability: An experiment in measuring simulation reusability. In *SSR*, pages 247–255, 1995. 25

[120] J.M. Voas. PIE: A dynamic Failure-Based Technique. *IEEE Transaction on Software Engineering*, 18(8):41–48, August 1992. 47

[121] J.M. Voas and K. Miller. Semantic Metrics for Software Testability. *J. Systems Software*, 20:207–216, 1993. 49

[122] J.M. Voas and K.W. Miller. Software Testability: the new Verification. *IEEE Software*, 3:17–28, May 1995. 47

[123] J.M. Voas and K.W. Miller. Substituting voas's testability measure for musa's fault exposure ratio. volume 1, pages 230–234 vol.1, June 1996. 25

[124] M. A. Branstad W. Richards Adrion and J. C. Cherniavsky. Validation, verification, and testing of computer software. *ACM Comput. Surv.*, 14(2):159–192, 1982. 60

[125] Y. Wang, G. King, I. Court, M. Ross, and G. Staples. On testable object-oriented programming. *SIGSOFT Softw. Eng. Notes*, 22(4):84–90, 1997. 47

[126] H. Washizaki, H. Yamamoto, and Y. Fukazawa. A metrics suite for measuring reusability of software components. In *9th IEEE International Software Metrics Symposium (METRICS'03)*, pages 211–, Sydney, Australia, September 2003. IEEE Computer Society. 46, 56

[127] A. H. Watson and T. J. McCabe. Structured testing: A testing methodology using the cyclomatic complexity metric. NIST Special Publication 500-235, National Institute of Standards and Technology, August 1996. 36, 41, 51

[128] E. J. Weyuker. Evaluating software complexity measures. *IEEE Trans. Softw. Eng.*, 14(9):1357–1365, 1988. 50, 53, 54

[129] H. Y. Yang, E. D. Tempero, and R. Berrigan. Detecting indirect coupling. In *Australian Software Engineering Conference*, pages 212–221. IEEE Computer Society, 2005. 105

[130] P.-L. Yeh and J.-C. Lin. Software testability measurements derived from data flow analysis. In *2nd Euromicro Conference on Software Maintenance and Reengineering (CSMR)*, pages 96–103, Florence, Italy, March 1998. IEEE Computer Society. 36, 51

[131] P. Yu, T. Systä, and H. A. Müller. Predicting fault-proneness using oo metrics: An industrial case study. In *6th European Conference on Software Maintenance and Reengineering (CSMR 2002)*, pages 99–107, Budapest, Hungary, March 2002. IEEE Computer Society. 56, 60

[132] Y. Zhou and H. Leung. Empirical analysis of object-oriented design metrics for predicting high and low severity faults. *IEEE Trans. Software Eng.*, 32(10):771–789, 2006. 56, 60