

Université de Montréal

**Modelling Software Quality: A Multidimensional Approach**

par  
Stéphane Vaucher

Département d'informatique et de recherche opérationnelle  
Faculté des arts et des sciences

Thèse présentée à la Faculté des arts et des sciences  
en vue de l'obtention du grade de Philosophiæ Doctor (Ph.D.)  
en informatique

Août, 2010

© Stéphane Vaucher, 2010.



Université de Montréal  
Faculté des arts et des sciences

Cette thèse intitulée:

**Modelling Software Quality: A Multidimensional Approach**

présentée par:

Stéphane Vaucher

a été évaluée par un jury composé des personnes suivantes:

Jian-Yun Nie,  
président-rapporteur  
Houari Sahraoui,  
directeur de recherche  
Bruno Dufour,  
membre du jury  
Massimiliano Di Penta,  
examineur externe  
Khalid Benabdallah,  
représentant du doyen de la F.A.S.

Thèse acceptée le: 15 novembre 2010



## RÉSUMÉ

Les sociétés modernes dépendent de plus en plus sur les systèmes informatiques et ainsi, il y a de plus en plus de pression sur les équipes de développement pour produire des logiciels de bonne qualité. Plusieurs compagnies utilisent des modèles de qualité, des suites de programmes qui analysent et évaluent la qualité d'autres programmes, mais la construction de modèles de qualité est difficile parce qu'il existe plusieurs questions qui n'ont pas été répondues dans la littérature. Nous avons étudié les pratiques de modélisation de la qualité auprès d'une grande entreprise et avons identifié les trois dimensions où une recherche additionnelle est désirable : Le support de la subjectivité de la qualité, les techniques pour faire le suivi de la qualité lors de l'évolution des logiciels, et la composition de la qualité entre différents niveaux d'abstraction.

Concernant la subjectivité, nous avons proposé l'utilisation de modèles bayésiens parce qu'ils sont capables de traiter des données ambiguës. Nous avons appliqué nos modèles au problème de la détection des défauts de conception. Dans une étude de deux logiciels libres, nous avons trouvé que notre approche est supérieure aux techniques décrites dans l'état de l'art, qui sont basées sur des règles.

Pour supporter l'évolution des logiciels, nous avons considéré que les scores produits par un modèle de qualité sont des signaux qui peuvent être analysés en utilisant des techniques d'exploration de données pour identifier des patrons d'évolution de la qualité. Nous avons étudié comment les défauts de conception apparaissent et disparaissent des logiciels.

Un logiciel est typiquement conçu comme une hiérarchie de composants, mais les modèles de qualité ne tiennent pas compte de cette organisation. Dans la dernière partie de la dissertation, nous présentons un modèle de qualité à deux niveaux. Ces modèles ont trois parties : un modèle au niveau du composant, un modèle qui évalue l'importance de chacun des composants, et un autre qui évalue la qualité d'un composé en combinant la qualité de ses composants. L'approche a été testée sur la prédiction de classes à fort changement à partir de la qualité des méthodes. Nous avons trouvé que nos modèles à deux niveaux permettent une meilleure identification des classes à fort changement.

Pour terminer, nous avons appliqué nos modèles à deux niveaux pour l'évaluation de la navigabilité des sites web à partir de la qualité des pages. Nos modèles étaient capables de distinguer entre des sites de très bonne qualité et des sites choisis aléatoirement.

Au cours de la dissertation, nous présentons non seulement des problèmes théoriques et leurs solutions, mais nous avons également mené des expériences pour démontrer les avantages et les limitations de nos solutions. Nos résultats indiquent qu'on peut espérer améliorer l'état de l'art dans les trois dimensions présentées. En particulier, notre travail sur la composition de la qualité et la modélisation de l'importance est le premier à cibler ce problème. Nous croyons que nos modèles à deux niveaux sont un point de départ intéressant pour des travaux de recherche plus approfondis.

**Mots clés: Génie logiciel, qualité du logiciel, modèles de qualité, études empiriques, réseaux bayésiens, évolution du logiciel, composition de la qualité.**

## ABSTRACT

As society becomes ever more dependent on computer systems, there is more and more pressure on development teams to produce high-quality software. Many companies therefore rely on quality models, program suites that analyse and evaluate the quality of other programs, but building good quality models is hard as there are many questions concerning quality modelling that have yet to be adequately addressed in the literature. We analysed quality modelling practices in a large organisation and identified three dimensions where research is needed: proper support of the subjective notion of quality, techniques to track the quality of evolving software, and the composition of quality judgments from different abstraction levels.

To tackle subjectivity, we propose using Bayesian models as these can deal with uncertain data. We applied our models to the problem of anti-pattern detection. In a study of two open-source systems, we found that our approach was superior to state of the art rule-based techniques.

To support software evolution, we consider scores produced by quality models as signals and the use of signal data-mining techniques to identify patterns in the evolution of quality. We studied how anti-patterns are introduced and removed from systems.

Software is typically written using a hierarchy of components, yet quality models do not explicitly consider this hierarchy. As the last part of our dissertation, we present two level quality models. These are composed of three parts: a component-level model, a second model to evaluate the importance of each component, and a container-level model to combine the contribution of components with container attributes. This approach was tested on the prediction of class-level changes based on the quality and importance of its components: methods. It was shown to be more useful than single-level, traditional approaches.

To finish, we reapplied this two-level methodology to the problem of assessing web site navigability. Our models could successfully distinguish award-winning sites from average sites picked at random.

Throughout the dissertation, we present not only theoretical problems and solutions,

but we performed experiments to illustrate the pros and cons of our solutions. Our results show that there are considerable improvements to be had in all three proposed dimensions. In particular, our work on quality composition and importance modelling is the first that focuses on this particular problem. We believe that our general two-level models are only a starting point for more in-depth research.

**Keywords:** Software engineering, software quality, quality models, empirical studies, Bayesian models, software evolution, quality composition.



## CONTENTS

<b>RÉSUMÉ</b> . . . . .	<b>v</b>
<b>ABSTRACT</b> . . . . .	<b>vii</b>
<b>CONTENTS</b> . . . . .	<b>ix</b>
<b>LIST OF TABLES</b> . . . . .	<b>xv</b>
<b>LIST OF FIGURES</b> . . . . .	<b>xvii</b>
<b>LIST OF APPENDICES</b> . . . . .	<b>xix</b>
<b>ACKNOWLEDGMENTS</b> . . . . .	<b>xxi</b>
<b>CHAPTER 1: INTRODUCTION</b> . . . . .	<b>1</b>
1.1 A Historical Perspective . . . . .	1
1.2 Defining Quality . . . . .	3
1.3 Ensuring Quality . . . . .	5
1.4 Problem Statement . . . . .	7
1.4.1 Manipulation of Higher-order Concepts . . . . .	7
1.4.2 Tracking the Evolution of Quality . . . . .	8
1.4.3 Quality Composition . . . . .	8
1.5 Dissertation Organisation . . . . .	8
<b>CHAPTER 2: BACKGROUND</b> . . . . .	<b>11</b>
2.1 Quality Models . . . . .	11
2.1.1 Definitional Models . . . . .	11
2.1.2 Operational Quality Models . . . . .	13
2.2 Model Building . . . . .	15
2.2.1 What are Software Metrics? . . . . .	16

2.2.2	What can we Measure? . . . . .	16
2.2.3	Quality Indicators . . . . .	17
2.2.4	Notable Code Metrics . . . . .	19
2.3	Model Building Techniques . . . . .	21
2.3.1	Regression Techniques . . . . .	21
2.3.2	Machine-Learning Techniques . . . . .	23
2.4	Open Issues in Quality Modelling . . . . .	25
2.4.1	Incompatibility of Measurement Tools . . . . .	25
2.4.2	Unwieldy Distributions . . . . .	26
2.4.3	Validity of Metrics . . . . .	27
2.4.4	The Search for Causality . . . . .	28
2.5	Conclusion . . . . .	29
<b>CHAPTER 3: STATE OF PRACTICE IN QUALITY MODELLING . . .</b>		<b>31</b>
3.1	Quality Modelling Practices: an Industrial Case Study . . . . .	31
3.1.1	Quality Evaluation Process . . . . .	33
3.1.2	Issues . . . . .	35
3.2	Initial Exploration of Concerns . . . . .	37
3.2.1	Systems Studied . . . . .	38
3.2.2	Metrics Studied . . . . .	39
3.2.3	Analysis Techniques . . . . .	40
3.2.4	Issue 1: Usefulness of Metrics . . . . .	40
3.2.5	Issue 2: Combination Strategies . . . . .	41
3.3	Research Perspectives . . . . .	44
3.4	Conclusion . . . . .	45
<b>CHAPTER 4: DEALING WITH SUBJECTIVITY . . . . .</b>		<b>47</b>
4.1	Existing Techniques to Model Subjectivity . . . . .	48
4.1.1	Fuzzy Logic . . . . .	48
4.1.2	Bayesian Inference . . . . .	50
4.2	Evaluating the Quality of Design . . . . .	51

4.2.1	Industrial Interest in Anti-pattern Detection . . . . .	51
4.2.2	The Subjective Nature of Pattern Detection . . . . .	54
4.2.3	Detection Methodology . . . . .	54
4.2.4	Evaluating our Bayesian Methodology . . . . .	58
4.2.5	Scenario 1: General Detection Model . . . . .	61
4.2.6	Scenario 2: Locally Calibrated Model . . . . .	64
4.3	Discussion of the Results . . . . .	66
4.3.1	Threats to Validity . . . . .	66
4.3.2	Dealing with Good Programs . . . . .	66
4.3.3	Estimating the Number of Anti-Patterns . . . . .	67
4.3.4	Alternative Code Representation . . . . .	68
4.4	Conclusion . . . . .	68
<b>CHAPTER 5: ANALYSING THE EVOLUTION OF QUALITY . . . . .</b>		<b>69</b>
5.1	Industrial Practice . . . . .	69
5.2	State of the Research . . . . .	70
5.3	Identifying and Tracking Quality . . . . .	72
5.3.1	Dealing with Temporal Data . . . . .	72
5.3.2	Quality Trend Analysis . . . . .	73
5.3.3	Finding Quality Trends . . . . .	74
5.4	Tracking Design Problems . . . . .	74
5.4.1	Global View on the Evolution of Blobs . . . . .	75
5.4.2	Evolution Trends Identified . . . . .	76
5.5	Discussion . . . . .	81
5.5.1	How to Track Blobs? . . . . .	82
5.5.2	How to Remove Blobs? . . . . .	82
5.6	Conclusion . . . . .	83
<b>CHAPTER 6: HIERARCHICAL MODELS FOR QUALITY AGGREGA-</b>		
<b>TION . . . . .</b>		<b>85</b>
6.1	A Multi-level Composition Approach . . . . .	86

6.2	Modelling Code Changeability . . . . .	88
6.2.1	Change Models . . . . .	89
6.2.2	Determining Method Importance . . . . .	92
6.3	Comparing Aggregation Strategies . . . . .	99
6.3.1	Study Definition and Design . . . . .	99
6.3.2	Data Analysed . . . . .	99
6.3.3	Variables Studied . . . . .	101
6.3.4	Operationalising the Change Models . . . . .	102
6.3.5	Results . . . . .	105
6.3.6	Discussion . . . . .	110
6.4	Related Work . . . . .	113
6.5	Conclusion . . . . .	115
<b>CHAPTER 7: COMPOSITION OF QUALITY FOR WEB SITES . . . . .</b>		<b>117</b>
7.1	Problem Statement . . . . .	117
7.1.1	Finding Information on a Web Site . . . . .	118
7.1.2	Assessing Site Navigability . . . . .	119
7.2	Related Work . . . . .	120
7.3	A Multi-level Model to Assess Web Site Navigability . . . . .	121
7.3.1	Bayesian Belief Networks . . . . .	121
7.3.2	Assessing a Web Page . . . . .	122
7.3.3	Navigation Model . . . . .	126
7.3.4	Assessing a Web Site . . . . .	128
7.4	Case Study . . . . .	129
7.4.1	Study Setup . . . . .	129
7.4.2	Navigability Evaluation Results . . . . .	130
7.4.3	Discussion . . . . .	130
7.5	Conclusion . . . . .	133
<b>CHAPTER 8: CONCLUSION . . . . .</b>		<b>135</b>
8.1	Contributions . . . . .	136

8.1.1	Handling Subjectivity . . . . .	136
8.1.2	Supporting Evolution . . . . .	137
8.1.3	Composing Quality Judgements . . . . .	138
8.1.4	Application to Web Applications . . . . .	138
8.2	Future Research Avenues . . . . .	139
8.2.1	Crowd-sourcing . . . . .	139
8.2.2	Using Complex Structures instead of Metrics . . . . .	139
8.2.3	Activity Modelling . . . . .	141
8.3	Other Research Paths Explored . . . . .	141
8.4	Closing Words . . . . .	144
	<b>BIBLIOGRAPHY . . . . .</b>	<b>145</b>



## LIST OF TABLES

3.I	Method-level metrics studied . . . . .	39
3.II	Class-level metrics studied . . . . .	39
3.III	Descriptive statistics for the NASA data set . . . . .	40
3.IV	Rank correlations between method metrics and bug count . . . . .	41
3.V	Rank correlations between class metrics and bug count . . . . .	42
3.VI	Classification rates for class fault-proneness . . . . .	43
4.I	GQM applied to Blobs . . . . .	55
4.II	GQM applied to Spaghetti Code . . . . .	55
4.III	GQM applied to Functional Decomposition . . . . .	56
4.IV	Program Statistics . . . . .	59
4.V	Salient symptom identification . . . . .	60
4.VI	JHotDraw: inspection sizes . . . . .	67
5.I	Degradation growth rates in Xerces . . . . .	80
5.II	Refactorings identified in Xerces for the correction of Blobs . . . . .	81
6.I	Relative weights for each importance function. . . . .	98
6.II	Descriptive statistics for replication data . . . . .	100
6.III	Correlations: number of changes and class metrics/model output . . . . .	106
6.IV	Correlations: number of changes and scores (type 2 models) . . . . .	107
6.V	Correlations: number of changes and scores (type 3 models) . . . . .	109
7.I	Inputs to the navigability model . . . . .	124
7.II	Page navigability CPT . . . . .	126
7.III	Independent samples test . . . . .	130





## LIST OF FIGURES

2.1	ISO9126 quality decomposition . . . . .	12
2.2	Distribution of cyclomatic complexity . . . . .	26
3.1	Fault distribution per development phase . . . . .	32
3.2	Stakeholders in a quality evaluation process . . . . .	32
3.3	Quality evaluation process of our partner . . . . .	34
3.4	Example of industrial quality models . . . . .	34
3.5	Example of a maintainability model . . . . .	35
3.6	Fault-proneness model using method-level metrics . . . . .	44
3.7	Fault-proneness model using class-level metrics . . . . .	44
4.1	Fuzzification process of a metric value . . . . .	49
4.2	Functional decomposition detection rules . . . . .	52
4.3	GQM applied to the Blob . . . . .	56
4.4	Probability interpolation for metrics . . . . .	57
4.5	Inspection process for Bayesian inference . . . . .	62
4.6	Inter-project validation . . . . .	63
4.7	Local calibration: average precision and recall . . . . .	65
5.1	Quality signal . . . . .	73
5.2	Dendrogram of quality signals . . . . .	75
5.3	Blob Ratios vs. Total Classes . . . . .	76
5.4	Evolution Trend Classification . . . . .	77
5.5	Evolution Trends Distribution of Blobs . . . . .	78
6.1	Logical decomposition of a system . . . . .	86
6.2	General composition model . . . . .	87
6.3	Class-method composition model . . . . .	88
6.4	Single-level change model . . . . .	90
6.5	Single-level change model explained . . . . .	90

6.6	Sample method-level model . . . . .	91
6.7	Modified type 1 change model to support method-level quality models . . . . .	91
6.8	Aggregation models for method-level evaluations . . . . .	92
6.9	Simple call graph between methods in their classes . . . . .	95
6.10	Simple call graph between methods in their classes . . . . .	98
6.11	Method change model . . . . .	103
6.12	Inspection efficiency for different sized inspections using class- level information . . . . .	107
6.13	Inspection efficiency for different standard aggregation strategies .	108
6.14	Inspection efficiency for different sized inspections by combining method-level information . . . . .	110
6.15	Inspection efficiency (in terms of size) . . . . .	111
7.1	Navigability Evaluation Process . . . . .	119
7.2	The original page-level navigability model: site-level metrics are identified in light gray . . . . .	123
7.3	The modified page-level navigability model . . . . .	123
7.4	Binary input classification . . . . .	125
7.5	Site-level quality model . . . . .	129
7.6	Navigability scores for good and random sites . . . . .	131
7.7	Initial model . . . . .	132
7.8	Potential improvements to the site . . . . .	132

**LIST OF APPENDICES**

**Appendix I: Reformulating Class-level Metrics . . . . .xxiii**



## ACKNOWLEDGMENTS

I thank my advisor, Houari Sahraoui for introducing me to the problems with empirical software engineering. He taught me that good experimentation is difficult and needs to be done in a rigorous manner. He also helped me become independent by assisting in the supervision of other students in my research group.

I would like to thank my collaborators. I thank Foutse Khomh who was a great research partner, for our great discussions that lead to multiple publications. I thank Prof. Yann-Gaël Guéhéneuc for his great toolkit (Ptidej), scientific discussions, and moral support. I also thank Marouane Kessentini and Prof. Naouel Moha, for our productive partnerships concerning anti-patterns. Finally, I would like to thank Simon Allier and Prof. Bruno Dufour for their help understanding how call graphs work.

Most importantly, my Ph.D. would not have been possible without the support of my wife, my “grande pitoune”, Catherine and the love of my two-year-old “petite pitoune”, Fiona. I would also thank my pop, Jean Vaucher, for his help focusing my ideas and for rereading my dissertation.



## CHAPTER 1

### INTRODUCTION

At the start of the 21st century, software is at the heart of many important aspects of society. Not only are these software systems omnipresent, but their complexity is an order of magnitude greater than anything produced ten years ago. These systems were built over the course of many years during which developers come and leave. The consequence is that keeping an old system up to date with new requirements is very expensive. Conservative studies show that well over 50% of effort is spent on maintenance [Bel00, HM00]. In this context, it is important to have tools and techniques to regularly check the quality of these systems to ensure that, with time, a system does not become a burden.

#### 1.1 A Historical Perspective

In the sixties, IBM decided to advance the state of the art with an ambitious operating system called OS/360. At the time, this OS was one of the most complex software ever undertaken. Although eventually a commercial success, its development proved to be almost too difficult to manage, going billions over budget: the estimate was 675 million dollars, but it cost 5 billion dollars [IBM08]. At its peak, over 50,000 employees were working on the project. The cost of the project was such that it was famously coined the 5 billion dollar gamble by Fortune magazine, for if the project failed, IBM would have gone bankrupt. Many important lessons were learned from this ordeal [FPB78], and these contributed to the emergence of the field of software engineering as its managerial failure was much discussed in the NATO Software engineering conferences [NR69, BR70].

The NATO conferences had the objective of shaping the field of software engineering. It was for this purpose that different participants described issues with producing good software. One of the most important issues raised was how to scale production up

to the new large systems and limit costs. The main method was to improve the process of “manufacturing code”. They argued that process improvement would improve software *quality*, a term mostly reserved to describe software that conforms to its specification and that contains few bugs. The advantage of having “clean” code was expressed only once, by M.D. McIlroy (p.53 [NR69]). He stated that some effort should be spent to evaluate the “quality of work” of a programmer, by assessing its *readability*.

Since the early days of software engineering, the notion of quality has had to evolve. It was typically thought that a system would be developed, and as soon as all the bugs were found and corrected, it would then be used with little or no modification. Eventually, the system would be retired and replaced by a newer system. This was not the case. The construction of these early monster systems was not cost-efficient, and therefore management moved towards an evolution model hoping to reuse previously coded artefacts. In this context, the seemingly unimportant factor of readability becomes very important. The predominant methodologies like the waterfall model [Roy70] focused on the production of a final, functional system; these were not adapted to the reality of modifying large-scale systems [McC04].

Another important change to software systems is their increasing level of interactivity. In the sixties and seventies, programs were batch-oriented and there was little to no interaction. Ever since the advent of user interfaces, user expectations in terms of usability have constantly increased. Currently, trends on the Web even allow users to interact with each other with user-generated content. Consequently, for modern software to be successful, it should not only provide bug-free functionality, but it should do so with an adequate user-experience. Concerning the issues of user perceived quality, Kan [Kan95] states that it is an ambiguous, multi-dimensional concept. In particular, he distinguishes between the user view of quality which can be experienced but not defined, and the professional view which can be measured and controlled. Of course, the view of professionals is difficult to assess objectively because it is determined by their specific quality-related needs.



## 1.2 Defining Quality

According to the Oxford Dictionary, quality is the *degree of excellence* of a thing. A quality software system is one that works well and gives satisfaction. With the majority of effort being spent to maintain systems [Bel00, HM00], we can assume that quality should mean that the system will continue to give satisfaction in the future.

For a broader understanding of software quality, Kitchenham and Phleeger [KP96] refer to the work of David Garvin [Gar84]. Garvin presents case studies of quality management from different industries, mostly manufacturing. He reports five views of quality:

- The transcendental view: quality can be recognized, but not defined. This can be seen as the idealised view of quality presented by software evangelists;
- The user view: quality is how well a product suits a user's needs. This view considers, but is not limited to, the notion of crashes as it relates to user experience;
- The manufacturing view: quality can be guaranteed by following a well-controlled series of steps. This perspective focuses on improving the development process. It follows the philosophy that by following a good process, we should be able to produce good software;
- The product view: the quality of software depends on its internal characteristics. This is the view of most metrics advocates. If a product is well written, then an external perception of quality should in turn be good;
- The value view: quality depends on what someone would pay for it. This is typically a viewpoint of investors and upper management.

Each of these views corresponds to that of different types of stakeholders. Ideally, the best product would satisfy all these stakeholders, but this is rarely the case. For example, a development team might consider their code base unwieldy and would like to restructure (product view), but management does not want to spend more on the product as the restructuring might not produce enough value (value view) [BAB<sup>+</sup>05].

Kitchenham and Phleeger then used these views to organise the results of a survey. The majority of respondents said that the manufacturing and user views were the most important. Ironically, other research suggests that users often do not know what they would like unless they experience it first [Gla09]. Without this experience, users will focus on an idealised view of what they want (*e.g.* multitude of features and few bugs). However, his idealised view rarely corresponds to their actual appreciation of the system [Sch03b].

It is impossible to find a universally accepted notion of quality as it depends both on the application domain and on the needs of the different stakeholders. For example, the aeronautics industry needs for their systems to be as reliable as possible by avoiding failures. A text editor would obviously not have the same reliability requirement; instead, it might have a requirement to be user-friendly. The perception of quality also depends very much on the stakeholder. Managers might be interested in limiting the cost of maintaining old systems while a user would have other criteria to evaluate quality.

The abstract nature of quality is a problem as quality control is an important part of all fields of engineering. In software engineering, we therefore use concrete, useful approximations of quality (*e.g.*, absence of bugs) and try to exploit this data to improve the quality of software to enable various stakeholders to make enlightened decisions. For example, we might like to know when a system is reliable enough to be released.

The objective of quality modelling is to describe relationships between different explicative factors and quality. Until now, we have focused on quality as an abstract concept. However, for it to be understood and controlled, it first needs to be measured, and the exact measurement to hard to define. Furthermore, this measure can be difficult to collect because it cannot be measured directly on the system. Rather, it is observed in a specific context that is domain and stakeholder dependent. For example, to measure reliability, it is possible to use the average up time of a system. This measure would obviously require the system to be executed.

### 1.3 Ensuring Quality

There are many approaches proposed to produce and ensure high-quality software. One approach considers the *process* used to build software. By setting up a process that includes quality-assurance activities, the perceived quality of a system should increase. In fact, over the past decades, there have been significant process-level improvements which in turn have improved the software produced. There are many existing methodologies that focus on different perceptions of quality. For example, the Cleanroom process [MDL87] is focused on defect prevention. On the other hand, agile methodologies like eXtreme programming [Bec99] try to keep software as flexible as possible in order to quickly get feedback from its users.

To ensure quality, these development methodologies rely on activities like formal methods, testing, and code inspections to judge if a system is ready to be released. *Formal methods* use proofs to show that code actually conforms to a specification. Formal methods are very expensive because they require well-defined specifications, and consequently are usually only used for important parts of mission critical applications [Men08].

*Testing* is generally considered the most important quality assurance activity. It consists of comparing the state and behaviour of a system to expectations; it can focus on different aspects like reliability, performance, and conformity to specification. However, it is limited to finding given specific parameters (*e.g.* finding faults within a specific execution context). It therefore cannot guarantee the absence of problems. From a management standpoint, it is difficult to assess when enough testing has been completed before releasing software [oST02]. Testing is a relatively costly task: studies show that testing accounts for over 30% of forward development costs [Bei90], and many mission-critical systems even have more testing code than actual system code. Therefore, the amount of testing done is typically limited by the resources available. Testing can ensure that there are few bugs in a system and that the specification is followed, but it cannot predict future problems like high maintenance costs.

Finally, *code inspection* techniques require developers to read the code of others and

evaluate its quality. Code inspections are very useful as they can uncover a great variety of problems, some are functional [BS87, SBB<sup>+</sup>02], but many are concerned with abstract notions of software quality like its capacity to be changed for future needs [ML08]. The basic idea is that *well written code* is good quality code, which needs minimal testing and which should be easy to maintain. Typically, inspectors will evaluate different aspects of the code, like its understandability, the presence undue complexity, and the use of known robust patterns. This notion of quality is difficult to test as it depends not only on the satisfaction of current needs, but of future needs as well. To benefit from inspections, developers need to read, understand, and comment code, which requires time. Consequently, this approach cannot be applied to the totality of a large system.

The idea motivating quality modeling is that code inspections are good, but require human intervention and are, therefore, too costly to be applied on the totality of a system. A quality model tries to automate the inspection process. As inputs, the model uses metrics which describe characteristics a human inspector would use, and tries to predict an inspector's quality assessment. There are limits to what a model can do automatically. Although it tries to mimic an expert's judgment, it can only analyse the surface structure of the system. For example, it can check for the presence of comments, check formatting, and even consider the size and links between different modules, but it cannot analyse elements of good design like *meaningful* comments, reuse of tested architectures and algorithms. A model may try to simulate a human process by using AI techniques.

Quality models can find different niches in different development processes. In agile development processes, they can be included in a continuous integration process [DMG07], and executed every day, informing developer of potential problems. In a more process-heavy methodology, these can be used by independent validation and verification teams who perform quality audits [MGF07].

As a quality model can evaluate the totality of a system, it can be used for two purposes. First, it can guide testing efforts and more elaborate inspections to certain parts of a system exhibiting bad properties. Second, knowing the quality of the different parts of a system, a model could provide an aggregate measure of the system. From a management perspective, a model can thus be used to guide decisions. They can,

for example, enforce quality-level obligations or estimate the cost of ownership of their systems.

## 1.4 Problem Statement

The current state of research focuses on proposing models that directly mimic a human inspection process. Models typically use metrics describing structural properties (like size, coupling, and cohesion) of the system extracted either from design artefacts or directly from the source code. As outputs, they produce an estimated value of an indicator of quality like bugs or effort [LH93b, MK92, BBM96, HKAH96, KM90, OW02, NBZ06]. These two quality indicators are arguably the most unambiguous measures of quality as few can deny that a system that crashes should be fixed, or that a system that is expensive to maintain might need to be restructured.

There are however some problems with the application of state of the art techniques. At the start of our research, a large company asked us to evaluate their quality modeling efforts. They had performed a literature review and applied existing standards to build quality models. They wanted independent and knowledgeable experts to audit their approach and suggest improvements. Our investigations of their modeling efforts identified the following problems in current research on quality modeling.

### 1.4.1 Manipulation of Higher-order Concepts

The first problem comes from the theory that underpins traditional fault-prediction models: *that good design leads to good quality*. The problem is that it is not obvious what good design is. Current practices rely on code-level metrics to decide if a module is well designed. Such approaches ignore the fact that good design conveys meaning and semantics that are not easily measurable. Therefore, we need to manipulate higher-order concepts (*e.g.*, design patterns and practices), to improve the state of quality modelling. Unfortunately, there are no exact techniques to identify these semantically rich concepts. Furthermore, these concepts are useful only if they are applied to specific contexts. For example, a design pattern is a reusable solution for a specific type of problem. Unfortu-

nately, different developers might disagree on whether pattern is appropriate for a given context; this is because **judging the quality of design is a subjective activity**.

### 1.4.2 Tracking the Evolution of Quality

Good design allows developers to modify and improve a system. Recent research has included aspects of this when modelling quality. Typically, this is done by including changes in a quality model [MPS08, NB05b]. Yet, changes, like design, have strong semantic meanings. For example, changes can be done to add functionality or to correct a bug. There also exists work that tries to classify changes [AAD<sup>+</sup>08, MV00], but what is lacking are **techniques to identify changes in code that have a significant impact on the quality of a system**.

### 1.4.3 Quality Composition

A final problem concerns the **absence of quality composition models**. Typically, a quality model evaluates quality on one specific type of software artefact, more often than not, the level for which we have access to quality indicators. In procedural code, the evaluated artefacts are either files or procedures; in object-oriented systems, the focus is generally on classes. The quality of a system is a function of the quality of its constituents [BD02], but to the best of our knowledge, no research seeks to determine which types of aggregation functions are appropriate. For a decision-maker who wants a bird-eyes view of the state of their systems, the lack of a proper aggregation mechanism is a serious problem. For example, our partner uses an aggregated quality score to monitor outsourced maintenance activities. This score is then used to accept or reject changes to the system. They consequently need for this score to represent the actual quality of the system.

## 1.5 Dissertation Organisation

This dissertation is organised as follows:

1. Chapter 2 covers quality models. It describes input variables, output variables and model building techniques. It also describes open issues in quality modelling.
2. Chapter 3 covers industrial practice. We examined the practices at a large company and observed problems that influenced the directions of our research. We reproduced some of the issues raised on available data sets.
3. Chapter 4, addresses subjectivity in quality assessment. We proposed the use Bayesian models to support subjectivity. We successfully applied these models to the problem of anti-pattern detection.
4. Chapter 5 explains how to use quality models to track quality over time. We presented a signal analysis technique to identify quality change patterns.
5. Chapter 6, we present issues with quality composition, and propose a general model to enhance quality models to support composition relationships. We introduced the complex notion of importance when combining low-level data. Experimentally, we show that our enhanced models can produce superior results to traditional single-level models.
6. Chapter 7 shows how our composition techniques can be applied to a different domain (web sites vs. executable code). In a study of 40 web sites, we found that our models could correctly distinguish good site from randomly selected ones.
7. We conclude the dissertation in Chapter 8, where we describe the lessons we learned, the discarded research paths, and discuss potential future work.





## CHAPTER 2

### BACKGROUND

This chapter presents the background material required to understand the evaluation of quality in software systems. In particular, we focus on quality models, and software metrics. We also provide an overview of existing techniques to build these models. Finally, we raise open issues pertaining to quality. Specific related work required to understand a specific chapter will be present at that moment.

#### 2.1 Quality Models

The goal of modelling quality is to establish a relationship between measurable aspects of a system and its quality. There are two types of quality models: *definitional* models and *operational* models. Definitional models decompose a high-level, abstract notion of quality into more concrete contributing factors. Operational models are sometimes based on these definitional models, but the main difference is that they can be implemented and executed to produce quality scores.

##### 2.1.1 Definitional Models

Many definitional models have been proposed: the McCall [MRW77], Boehm [Boe78], FURPS [GC87] and ISO 9126 [ISO91] models, to name a few. The first three models were created in specific industrial contexts while the last is an ISO standard. All models organise quality in a tree-like structure where quality characteristics are either measured directly or composed using other characteristics (or subcharacteristics). Figure 2.1 presents this progressive decomposition of quality into metrics. These models are interesting because they point out different non-functional aspects of software to consider when evaluating quality.

The first criticism of these models is that they are not operational. While they express the need for measurement, they do not define which exact metrics to use and how to

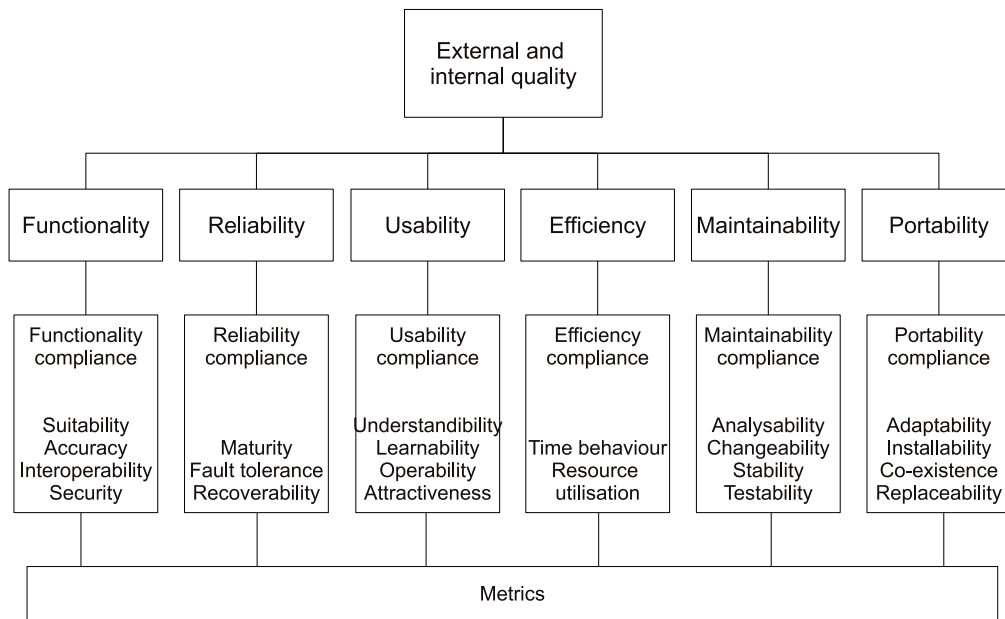


Figure 2.1: ISO9126 quality decomposition

combine information from one level to another. For example, the McCall and FURPS models consist of checklists of concepts to evaluate, but do not specify what to measure and how to obtain quality values [AKCK05]. Second, these standards are not based on empirically observed relationships, but rather on qualitative, sometimes anecdotal evidence [PFP94, JKC04]. Other criticism comes from the fact that these models have a limited view of quality which is based on a developer perspective instead of a managerial, more value-based perspective [Pfl01, BAB<sup>+</sup>05].

Another family of models is the result of the Dromey model [Dro95] building framework. The idea proposed is that abstract quality attributes cannot be measured directly. There are however quality-carrying properties that can be measured on the entities used to build the system. These properties can impact the abstract quality characteristics. Good properties should therefore be built into software to insure its good quality while bad properties should be avoided. As this is a model building framework, Dromey does not present how to implement and validate a model.

## 2.1.2 Operational Quality Models

The step between having a theoretical model and putting one into operation is complex. The literature describes two approaches that correspond to two different perspectives on quality. The first is a *top-down* managerial approach because management wants to verify or attain certain managerial goals. The second is a *bottom-up* approach corresponding to the view of an independent quality team that conducts audits to locate quality problems in a code base.

A pragmatic reason why these approaches are different is the issue of *data ownership* [Wes05]. If you want to evaluate a system, you need to have access to data, and the access to this data depends directly on who controls its production. Let us consider a scenario with a manager, a development team, and users. The manager knows a project's budget and schedules. A developer knows the time spent on its development tasks. Users know its operational problems. In order to study these different factors, we would need for these different actors to report the data needed, in a relatively systematic process. A top-down management decision can impose the collection of certain metrics because it allocates the budget of some of these actors. On the other hand, an auditing team generally has to make do with what is available, or easily extracted because its audits should not directly interfere with the activities of development teams.

### 2.1.2.1 Top-down Models: a Goal-oriented Approach

Basili and Weiss [BW84] described a systematic measurement process called GQM to evaluate and guide quality assurance in an industrial setting. GQM stands for Goal-Question-Metric and proposes a top-down approach to modelling. First, management should define a clear quality goal like minimizing downtime of a system. Then, relevant questions should be defined that verify if the goal has been attained. Finally, specific metrics should be chosen to answer the questions asked and verify objectively if the goals have been attained. A key aspect to this methodology is that it can be extended to track quality and suggest improvements [BR88].

In [BMB02], Briand *et al.* describe a methodology to define the variables in a quality

model. They separate the process in four steps:

1. Setting up the empirical study: derive measurement goals from corporate objectives. These goals should define a set of hypotheses that relate attributes of some entities to others. For example, the use of product size to predict effort.
2. Definition of measures for *independent* attributes: this is the conversion of an abstract attribute to a concrete measurement applied to *input variables*. This mapping should be done after a literature review and a refinement process. Example: size measured as lines of code (LOCs).
3. Definition of measures for *dependent variables*: same process, but for the *quality metric*. For example, effort measured as number of staff months.
4. Refinement and verification of the proposed hypothesis. This is when the exact relationship is refined. For example, we can assume that there is a linear function between LOCs and number of staff months. This assumption can be tested.

As we mentioned, these steps are only applicable when it is possible to select the metrics used. Otherwise, we need to consider a bottom-up, data driven approach.

### **2.1.2.2 Bottom-up Models: Locating Problems**

The second perspective concerns *independent verification and validation* (IV&V) teams. Some large companies, like our partner, use IV&V teams to perform quality audits. Their *raison d'être* is to provide an in-house expertise in quality control. They act as internal consultants to ensure a certain level of quality in their applications using different techniques like inspections and testing. Their main focus is to efficiently locate problems and transmit this information back to the development teams for correction.

These teams have a difficult time getting metrics because they tend to deal with a large number of development teams, each with its own particularities. A former research chair with the IV&V team at NASA, Tim Menzies [Men08] makes the case for using simple static (*i.e.* non-executed) code metrics in quality models. His justification is

that only those metrics can be collected in a consistent format over a long period of time. Other metrics, like dynamic (executed) metrics, could possibly provide interesting information to consider, but their collection is not cost-effective because an IV&V team would need to manage, understand, and run different representative execution profiles. Consequently, the models used by NASA only rely on static code metrics to predict where bugs might lie and focus black-box testing to those areas.

In this dissertation, we share the NASA view on metrics: we only consider data that is measurable on the product and evaluate its influence on quality indicators. The reasons stated by Tim Menzies also affect our partner: other types of data are either unavailable, in ad hoc formats, or outdated [Par94]. This is also similar to the situation of university research teams that study open-source systems where there is no controlled metric collection process.

## 2.2 Model Building

The point of a model is to establish a relationship between two sets of variables: explicative factors, called independent variables, and observed quality assessment, called dependent variable. Essentially, we are looking for a function  $f$  of the form:

$$quality \approx f(attributes)$$

where *quality* is our quality score, and *attributes* are metrics describing the system.

In general, *quality* is a metric that is an indicator of quality like the absence of faults. The *attributes* are typically code metrics that describe the structure of the system. Finally, the function depends on the type of relation that links the structure of a system to its quality.

The models can serve two purposes. They can either focus on predicting quality or on explaining the underlying relationships. Prediction only requires that the function  $f$  be accurate and produce a good estimate of quality. To use a function to understand the phenomenon, then the model needs to be able to assess individual contributions of independent variables to quality.

### 2.2.1 What are Software Metrics?

Measurement is at the heart of a quality assurance process. The act of measurement is the assignment of a value to the attribute of an entity [ISO91]. This permits a description, and a subsequent quantitative analysis of a phenomenon. From a management perspective, what is measured should provide information to guide decision-making.

### 2.2.2 What can we Measure?

Many aspects of a software development process can be measured. What is measured will determine the usefulness of a quality model. Fenton [Fen91] lists three types of entities that can be measured:

1. The **product**: This category includes the *artefacts* produced during all development cycles, *e.g.* there are the design documents, the source code, *etc.*
2. The **process**: This category includes activities, methods, practices and transformations applied in developing a product, *e.g.* the development methodology, testing procedures, *etc.*
3. The **resources**: The category describes the *environment* whether software, material, or personnel.

These entities have both *internal* attributes which can be measured directly and *external* attributes, measured within a specific context (*e.g.* operational environment). For example, program size can be measured directly (*e.g.* using a Unix `wc` command), but there is no direct measure of how much effort is required to change a program because this would require knowledge of external factors like the skills of a developer. What is measured typically depends on the possibility, the cost, and the benefit of extracting a metric. Typically, internal metrics are cheaper to compute than external metrics because they can be quantified unambiguously, even automated to a certain extent [MGF07]. In fact, internal product metrics are so cheap to extract, that at Microsoft, developers have tools regularly extracting code metrics installed in their work environments [NB05a].

External metrics (*e.g.* the opinion of a developer) are more expensive to collect, but contain more pertinent information.

To get an appreciation of the quality of an entity, we use certain external attributes, called *quality indicators*. The most common indicators concern bugs (measured using a count or by density). This corresponds to the intuitive notion that “good” software should not contain errors. It is an external measure because it requires the system to be executed for an error to appear. This quality indicator is expensive to gather because it requires a relatively systematic process where development maintenance teams archive their testing activities.

There is an abundant (overwhelming) literature on this subject; Zuse [Zus97] lists hundreds of articles proposing over 1500 metrics. Here, we concentrate on the important metrics that have been most studied.

### **2.2.3 Quality Indicators**

As stated in the previous section, the only source of coherent and consistent metrics is the source code. However, these internal metrics do not indicate the quality of a system. We therefore need to have access to additional data: quality indicators. For an overview of the quality indicators studied in the literature, we refer to a survey conducted by Briand and Weiss [BW02] in 2002. They surveyed almost 50 studies with respect to the metrics used and their analysis techniques. Their survey indicates that the most studied indicator is the presence of faults. Since 2002, these types of studies have multiplied (*e.g.*, [EZS<sup>+</sup>08, ZPZ07, MPF08]). Most other studies focused on either effort or change (*e.g.*, [KT05, KL07, DCGA08, KDG09]).

#### **2.2.3.1 Software Faults**

The study of faults is an obvious choice as a crash will obviously impact every stakeholder. Finding a clean source of bug data is, however, a difficult task because they can be discovered at many different stages in a development life-cycle: development, testing, or operation. This means that there are different owners of this data [Wes05]. Developers

rarely keep track of the mistakes they make while coding. We therefore need to get this data from testers and users. Data from testers can be gathered relatively systematically, but data from users is much more difficult to deal with. A typical approach is that users will file bug reports and we want to extract bug counts from these reports. However, bug reports tend to lack important information to perform a study. The most obvious problem is that of *traceability*: given a bug description, what is the most likely part of the software responsible? Depending on the organisational structure of a company, there might be a paper-trail that can be mined to figure out what code modification corrected which bug, and where. Current state of the art techniques use heuristics to match information contained in bug reports to different software entities. These typically combine information from mining version control system logs and from bug-repositories [EZS<sup>+</sup>08, DZ07]. They sometimes produce noisy results because the heuristics used (*e.g.* searching for bug identifiers in commit logs) depend on the presence of patterns in the data analysed.

The study of bug repositories shows that although they are an essential source of information, they also present significant problems with data coherence. First, there are incorrectly classified bug reports [AMAD07, AAD<sup>+</sup>08, BBA<sup>+</sup>09]. Misclassification might mean that what is thought to be a bug is actually an enhancement request. Second, some development teams know that bug reports can affect their performance reviews. Consequently, development teams may try to hide severe bugs for political reasons [OWB04]. These problems are a reason why researchers tend to limit the use of bug data to determining if a software entity is fault-prone or not ( $> 0$  bugs) [Men08].

### 2.2.3.2 Effort

The second most studied indicator is *effort*. Effort is even more difficult to measure because software engineers tend to rarely indicate the time spent on specific activities. There are two solutions to circumvent this problem. First, managers have effort data, but on a macro-scale (project-level). This could allow a high-level study of quality, but this information would be difficult to map to specific software components. Second, for a fine-grained analysis, researchers generally use changes as a surrogate measure. Changes in code can be measured in terms of size, type and scope [NB05b]. However,



unlike faults, changes are not necessarily a sign of something bad; good software is likely to be modified to provide additional functionality. Moreover, we would expect a developer to add a lot of code if he adds lot of new functionalities.

As with faults, changes are often treated as binary activities: components are considered change-prone or not [DCGA08, KDG09] because the total development effort of modifying code is not only related to the size of a change, but also to the subsequent activities in the development process like re-testing.

### 2.2.3.3 Other Quality Indicators

There are few other studied quality indicators like reusability [MSL98] and perceived cohesion [EDL98], but assessing reusability and cohesion requires a developer to review a whole system, a requirement that would not scale to large systems.

## 2.2.4 Notable Code Metrics

Over the past decades, researchers have proposed a multitude of metrics to characterise different aspects of software artefacts. These metrics generally focus on notions like size, complexity, coupling and cohesion [YC79]. We present here what could be considered the standard set of metrics. These are widely used in research and are included in commercial tools such as McCabeIQ<sup>1</sup> and Borland TogetherSoft<sup>2</sup>. Furthermore, the metrics described here are those used by NASA and by our industrial partner. For an extensive comparison of standard OO metrics, we refer to [JRA97]; for an exhaustive list of coupling metrics, we refer to [BDM97, BDW99]; for a list cohesion metrics, we refer to [BDW98].

### 2.2.4.1 Traditional Metrics

The first well-studied metric was the number of lines of code (LOCs), used to describe the *size* of a system. LOCs were used by Wolverton [Wol74] to measure the pro-

---

<sup>1</sup><http://www.mccabe.com/iq.htm>

<sup>2</sup><http://www.borland.com/us/products/together/index.html>

ductivity of developers. The widespread acceptance of this metrics is due to its simplicity and the ease of its interpretation. It has however been criticised for several reasons. The major concerns are that it does not take into account simple formatting rules (*e.g.* use of hanging brackets) and is language-dependent because a programming languages can be more verbose than another. We can still assume that LOCs is a reasonable measure of size within a specific context (*e.g.*, same language and coding rules).

More robust measures of size were proposed in the seventies by McCabe [McC76] and Halstead [Hal77]. The McCabe metric suite analyses the control flow graph of procedures to measure different types of *complexity*. The most popular metric is cyclomatic complexity which counts the number of independent paths through this graph. The Halstead complexity metrics are based on the number of different operands and operators defined in a program.

#### 2.2.4.2 Object-oriented Metrics

For object-oriented (OO) programs, a metric suite was proposed by Chidamder and Kemerer [CK91]. They proposed six metrics to characterise different aspects of classes. The metrics cover *coupling*, *cohesion*, *complexity* and added the use of *inheritance*. Although their metric suite, also called the CK metrics, has its fair share of weaknesses, the most notable being the lack of empirical validation at time of proposal, this suite is still in use almost 20 years later. For a given class, their metrics are defined as follows:

- **Weighted Method Complexity (WMC)**: summation of the complexity of the methods in the class;
- **Number of Children (NOC)**: the number of direct children of a class;
- **Depth of the Inheritance Tree (DIT)**: the maximum distance to the root of the inheritance tree;
- **Response For Class (RFC)**: the number of methods that can be invoked in response to a message received by an object of the class;

- **Lack of COhesion in Methods (LCOM)**: the disjoint set of local methods with regards to attributes;
- **Coupling Between Objects (CBO)**: the number of other classes that use and that are used by the class.

The exact interpretation of WMC is open to interpretation because the complexity of a method is left undefined. Metric extraction tools often simply assign a value of one per method [LH93a].

## 2.3 Model Building Techniques

The standard approach for building models is to apply statistical regression techniques or machine learning to a set of data previously collected. Next, we discuss both techniques.

### 2.3.1 Regression Techniques

There is a multitude of regression techniques used in quality evaluation. We typically select the regression analysis that corresponds to the type of relationship ( $f$ ) we believe exists between independent and dependent variables. Then, there are methods that try to find the optimal parameters to fit the data to this function. Most tools like SPSS and R have many predefined regression analyses implemented.

We reformulated the quality function for regression in Equation 2.1. In this equation, *quality* describes the desired output, the function  $f$  is of a form defined by the regression type, *code\_metrics* are the metrics used, and  $\beta$  is the set of parameters that are estimated by the regression.

$$quality \approx f(\text{code\_metrics}, \beta) \quad (2.1)$$

Different types of regression exist ( $f$ ). There is work on applying linear regression [MK92, KMBR92], logistic regression [BTH93, KAJH05], Poisson regression [KGS01], and negative-binomial [OWB04] to quality evaluation. These de-

fine not only the function type, but the specific type of parameters  $\beta$  considered and the type of variables that can be used.

All these regression types provide ways to ensure that 1) the relationship found is significant (not likely due to chance, *e.g.*, using an F-test), and 2) that the model correctly explains the relationship (goodness of fit, *e.g.*, using R-square). These two types of information are both important for the interpretation of the model [KDHS07]. A model could very well be significant, but explain very little of the phenomenon. Likewise, it might provide a good explanation, but there might not be a sufficient number of elements examined to assert that it is significant. We can also examine the parameters estimated,  $\beta$  to assess the importance of different independent variables in the model. We can verify both the statistical significance of parameters and its relative importance in the regression function.

*Linear regression* assumes that there is a linear relationship between metrics and a quality indicator. This is arguably the most intuitive regression technique; we therefore present an illustrative example. Let us consider that size, as measured in LOCs has a linear relationship with the number of faults. The regression function is expressed in Equation 2.2. The analysis (*e.g.* least ordinary square) estimates the slope  $\beta_1$  and the intercept  $\beta_0$  of the function given the data available. If the relationship is significant, the value of the slope ( $\beta_1$ ) can be interpreted. For every additional line of code, we would expect the number of faults would typically increase by  $\beta_1$ .

$$faults \approx \beta_0 + LOCs \times \beta_1 \quad (2.2)$$

The research using this technique dates back almost 20 years [MK92, KMBR92]. The authors of this work did not actually verify if the assumption of linearity made sense. As it turns out, recently the relationship between size and faults was explored by Koru *et al.* [KZEL09] who showed that it is not linear: as modules get bigger, the density of faults decreases. There exist recent fault models based on multi-variate (many independent variables) linear regression. These will apply transformations (*e.g.* Box-Cox) to the dependent variable [SK03] to make sure that the relationship is coherent.

*Logistic regression* is arguably the most popular technique as it is robust and is easily interpreted. It tries to explain the influence of discrete or continuous independent variables on a *binary* output such as fault-proneness (contains at least a fault). This technique was used in fault-proneness models [GFS05, BTH93, KAJH05] as well as change-proneness models [KDG09]

*Poisson regression* assumes that the dependent variable has a Poisson distribution. It thus functions on count data (non-negative integers). If we consider that faults are introduced as events, then this regression technique might be appropriate. There is a problem with the overdispersion of data. In a Poisson distribution, the mean and variance are the same. However, faults, changes, and other indicators follow a power-law distribution where the mean and variance might not even be finite (this is discussed later in this chapter). Researchers like Khoshgoftaar *et al.* [KGS01] and Ostrand *et al.* [OWB04] have thus used variations on this regression to account for this dispersion. They have respectively explored, the use of zero-inflated Poisson regression (which treats zero values separately), and negative-binomial regression, which includes a dispersion parameter. A *negative-binomial* regression is particularly interesting because it estimates a parameter to handle cases when variance far exceeds the mean.

A common problem to all regression techniques is called *multicollinearity*. This problem manifests itself when dependent variables are highly correlated with one another. In this case, models cannot evaluate the contribution of individual metrics as they compete with one another for importance in the model. Consequently, researchers tend to address the problem of multicollinearity by either eliminating variables or by performing pre-treatments like a principal component analysis to reproject the inputs onto another set of orthogonal dimensions [KZEL09, NWO<sup>+</sup>05].

### 2.3.2 Machine-Learning Techniques

In the early nineties, researchers like Porter and Selby introduced machine-learning techniques to build quality models [PS90]. Applying these techniques to the problem of quality evaluation has some advantages and disadvantages. Many techniques are non-parametric (do not rely on the particularities of the underlying distributions of variables),

and can model relationships that regressions cannot. On the other hand, few machine-learning techniques provide meaningful information as to the relative importance of metrics in the model.

Two main strategies exist: *lazy techniques* and *eager techniques*. A lazy technique keeps a set of observations on hand and uses these observations directly to decide how to predict the quality of a new observation. Eager techniques are very similar to statistical regression as they attempt to derive a relationship between a training set of observations and a quality indicator. This function is then reused to classify all new observations.

In the literature, there is a multitude of papers, each proposing the application of a given machine-learning technique: neural networks [HKAH96, KAHA96], decision trees [KPB06, KAB<sup>+</sup>96], k-nearest neighbours [KGA<sup>+</sup>97, GSV02, EBGR01], and boosting [KGNB02, Zhe10, BSK02]. The problem with the majority of this previous work is that the authors merely show how a machine-learning technique can be applied, but the results do not indicate significant improvements.

Nowadays, modern machine-learning libraries like Weka [WF05] allow for a multitude of algorithms to be executed on the same data set without needing any knowledge of the specific techniques used. Recent articles usually present the results of multiple machine-learning techniques [MGF07] to compare their relative performance. In their research, Lessmann *et al.* [LBMP08], show that the exact algorithm used for building models on publicly available NASA data is unimportant: most algorithms will converge and produce an equivalent function. The usefulness of a given model would depend on how easy it can be used (*e.g.*, if it can be interpreted).

Using machine-learning techniques to validate models is different than using statistical techniques. Machine-learning algorithms do not provide measures of how well they fit the data (unlike regression R-square). Instead, the models need to be tested empirically. Typically, a data set is split up in two parts. The first part is used to train the model, and the other is used to test it. Generally, if the problem is a classification problem like deciding if a module is fault-prone or not, we can use the correct classification rate. If the result is a numeric value, then we can either consider using correlations or average squared differences between predicted and known values.

## 2.4 Open Issues in Quality Modelling

In this section, we present some open issues pertaining to quality modelling. In the dissertation, we will not try to solve these problems. However, they affect our choices and will shape our discussions in later chapters.

### 2.4.1 Incompatibility of Measurement Tools

From a practical perspective, any relationship or model derived using metrics from one tool might not hold when using the “same” metrics, but extracted using another tool. Lincke *et al.* [LLL08] challenged the assumption that all metric extraction tools tend to use the same definitions to extract the same set of metrics. They ran different tools on data sets and found that the differences can be significant. There are many factors that explain these inconsistencies:

1. Some tools like MASU and CKJM [MYH<sup>+</sup>09, Spi06] analysing source code perform only limited type resolution. Therefore, any non-trivial class resolution will incur imprecision.
2. Some tools analyse byte-code and this representation contains less information than the source. For example, local variables in Java are not typed in the byte-code. Consequently, to recover and consider these types, tool developers would need to analyse the byte-code as in [GHM00]. However, this is often not the case.
3. Some classes are filtered from the set of considered classes. In Java, different subsets of the standard library are often ignored. For example, when measuring coupling, CKJM will ignore classes from the *java.\** pages.

Therefore, teams that want to set up coding guidelines using thresholds cannot simply reuse results from the literature unless they are sure the tools are the same.

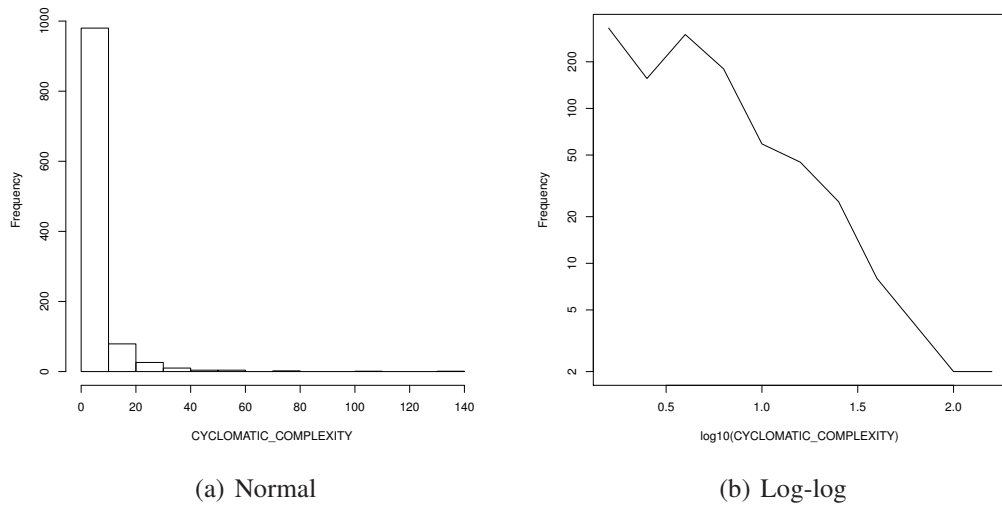


Figure 2.2: Distribution of cyclomatic complexity

### 2.4.2 Unwieldy Distributions

The type of distribution of the metrics used in models is a subject that has attracted little attention from researchers in software engineering. Model builders often implicitly assume that various parameters have simple bell-shaped distributions, but this may not be the case when dealing with real software systems. The first important article comes from a theoretical physicist, Christopher Myers [Mye03] who analysed the distribution of metrics. Both Myers and Louridas *et al.* [LSV08] found that a great number of metrics like LOCs and all C&K metrics (excluding inheritance metrics) follow a power-law distribution, a heavily skewed distribution.

A power-law distribution is characterised by the fact that the probability of observing a value,  $p(x)$ , decreases as  $x$  increases following the general form:

$$p(x) \propto \frac{k}{x^\alpha}$$

where  $\alpha > 1$ , and  $k$  can either be a constant or a slowly decreasing function (that depends on  $x$ ).

In Figure 2.2(a), we present the distribution of a complexity metric for the procedures



taken from an industrial system<sup>3</sup>. We can see that the vast majority of values tend to be around 0, but the mean is determined by a minority of high value instances (highly complex methods in this example).

Such skewed distributions are typically displayed using a log-log plot (Figure 2.2(b)). In a log-log plot, distributions decrease almost linearly. The corresponding slope corresponds to the value of  $\alpha$ . Theoretically, the value of  $\alpha$  is very important because a distribution has a finite mean only if  $\alpha \geq 2$ . Otherwise, the observed average value of a sample would increase with the sample size. Moreover, if  $\alpha < 3$ , the variance is infinite. If the variance of a distribution is not finite, then we cannot manipulate distributions as if they were normal [Ric94]. For any quality model like that of Bansiya and Davis [BD02], which uses an average of class-level metrics values to assess systems, this is a serious problem.

The  $\alpha$  values of the coupling data studied in [LSV08] indicates that there exist average values for coupling, but that variance would tend towards infinity as more data is obtained. Consequently, the average value is not a measure of central tendency. To avoid this problem, researchers typically apply a logarithm function to these metrics, which limits the impact of extreme values [AB06, MGF07]. These distributions are not well understood and are still actively researched [CSN09].

### 2.4.3 Validity of Metrics

A metric is only useful if it correctly characterises the attribute that it should stand for. For this reason, the validity of one of the standard CK metrics, LCOM, has been challenged, and this lead to this (lack of) cohesion metric to be redefined 4 times (LCOM2 to LCOM5). This is a serious problem and a possible reason why cohesion metrics are often left out of quality models [BW02, SK03],

Though the metric is flawed, attributes like *cohesion* and *coupling* are important indicators of quality. Good code is typically built using a divide and conquer strategy where a software system is made up of small independent components. A good decomposition ensures that these components are relatively independent from one another and

---

<sup>3</sup>Data is from the NASA IV&V dataset (PC1), which is studied futher in the thesis

that each component deals with one facet of the application. The independence between these components is what is called coupling and is typically measured by the number of links to/from components. A cohesive component is one that only deals with a specific facet of the application. This is an important concept and should be monitored, but it is semantic in nature. Since this concept requires interpretation, it should be considered an external metric. This is why it is hard to find a representative surrogate using an internal metric.

Recently, there are new metrics based on information retrieval that try to assess cohesion by estimating how many domain concepts are manipulated by a component [MPF08]. These techniques go beyond traditional metrics that only consider code structure. However, they are still limited by choice of terms using the source code. What can be retained is that both linguistic and structural information could be used in assessing cohesion, but they are often not sufficient because they cannot replace human judgment.

#### 2.4.4 The Search for Causality

Before basing action on the results of quality models, one should distinguish between correlation and causality. Correlation indicates that there are systematic changes on a variable whenever the other is changed. Causation [WRH<sup>+</sup>00, Pea00, FKN02] on the other hand, is the relationship between two events where the first *causes* the second. Discovering this type of relationship is very important when a model is used to guide improvements. In order to establish causality, there needs to be strong qualitative evidence suggesting that our quantitative observation is causal.

Operational quality models describe the relationship between metrics and quality indicators. However, even if the relationship is significant, we can only state that there is a correlation, but correlation does not imply causality. To assert that there is a causal relationship, there first needs to be a logical reason for the metrics to be the cause of quality. This could be addressed by selecting metrics using a methodology like GQM [BW84]. Second, there needs to be no alternative explanation, from a *confounding factor* [LK03]. A confounding factor is a variable that is not considered in the model, but that is correlated both to the metrics considered and to the quality indicator. A model that only

considers code metrics obviously misses information from the development context, *e.g.* process and resources. For example, if the model was built using regression data from a system written by C programmers with no OO background, it might identify that the use of inheritance is a potential cause of maintenance problems. An alternative cause could very well be the lack of experience of the developers using the OO paradigm. If the model does not explicitly take into account developer experience (the confounding factor), the only relationship visible is that inheritance is bad. Consequently, someone might believe that the way to improve quality is to eliminate inheritance in the system (instead of providing training to the developers). These confounding factors can reduce the usefulness of a model to guide changes.

Even when we only consider the product, there are confounding factors; the most important is the notion of *size*. A large number of existing studies have shown that size is correlated to faults. Furthermore, a great number of metrics in the literature are also highly correlated with size, thus any relationship identified using these metrics could be attributed to size. El Emam *et al.* [EBGR01] discuss this problem. On a large commercial system, they showed that CK metrics of complexity and coupling are highly correlated to size. When they did not control for size, these metrics were all useful predictors of faults. After control, this relationship disappeared indicating that the predictive relationship was in great part due to size. In a rebuttal, Evancho [Eva03] mentions that this type of problem is typical of many studies from social sciences. Trying to control size is only a logical step for model building only if size is available when the model would be used. Since the CK metrics can be extracted early towards the end of the design phase, it makes no sense to consider that size is a true confounding factor.

## 2.5 Conclusion

In this section, we presented on the different concepts that will be used throughout the dissertation. We focused on *metrics and quality models*. Furthermore, we indicated how these concepts are used in modern research. Finally, we concluded with a presentation of open-issues that will be used to discuss our methodologies and results. As our focus

is not on theoretical models, in the next chapter, we present our work with an industrial partner, as well as the problems they identified when building and applying their quality models.

## CHAPTER 3

### STATE OF PRACTICE IN QUALITY MODELLING

During the course of our research, we had the opportunity to observe quality modelling practices in the context of a large organisation. Our team was asked by a large company in the transportation sector to evaluate its quality assessment process. After a comprehensive review of the literature, this company had set up a state of the art quality assurance process and acquired the latest code measurement tools; yet, it had concrete problems in applying the process. In this chapter, we describe its quality modelling approach and the different issues that it faced. Finally, we present a study of some of these issues and discuss why we believe that these are interesting ideas from a research perspective.

#### 3.1 Quality Modelling Practices: an Industrial Case Study

Our industrial partner is the IT department of a very large company. This department has over 800 employees and generates over \$200M USD in sales. Following deregulation in the transportation sector, it faced more and more international competition. In order to remain competitive, our partner reviewed its IT development processes. The most startling discovery it made was that the number of faults discovered late in the development process was far superior to the number of faults discovered earlier on. The fault distribution is illustrated in Figure 3.1. The percentage of faults (31%) discovered after the systems were released was far greater than those discovered in any other development phase, including integration testing (25%).

This observation prompted management to create an independent verification and validation (IV&V) team to verify the quality of the systems produced. This IV&V team offers an auditing service to both internal and external teams who develop and maintain the 200+ live systems the company operates. Between 10-25% of projects are regularly examined by the IV&V team.

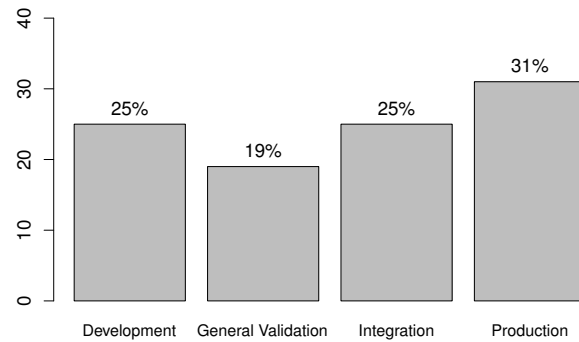


Figure 3.1: Fault distribution per development phase

There are three interested stakeholders in the quality auditing process (illustrated in Figure 3.2):

- **Quality engineers** evaluate code-level risks, and provide information to managers as well as corrections to developers;
- **Managers** evaluate system-level risks, and review performance of developers;
- **Developers** see their performance reviewed and need to write code that passes quality checks.

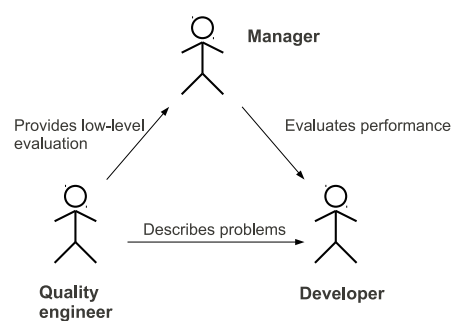


Figure 3.2: Stakeholders in a quality evaluation process

The IV&V team adapted the ISO9126 standard to describe their quality needs and applied the GQM methodology to define metrics to assess the quality of their systems.

However, they needed advice as to how to validate their models; this prompted them to contact us. There were three axes to our partnership: reviewing their quality evaluation model, improving their identification of design problems, and providing visualisation methods to assist with audits. I served as the contact between the quality engineers and the researchers. My primary research focus was on quality evaluation, but I also investigated design problems. To perform this review, we were provided with documents detailing their requirements, processes, and tools. Furthermore, we conducted interviews with the members of the IV&V team to better understand the situation. In the following sections, we describe these issues.

### **3.1.1 Quality Evaluation Process**

At the heart of the company's quality measurement effort is a software suite that automates the evaluation of systems. Upon the receipt of a system to audit, the IV&V team enters the system into the suite, which first extracts metrics from methods and classes using commercial tools. These metrics are then combined to produce scores for different quality characteristics. Finally, the quality engineers manually review the findings and often re-adjust some aspects of the quality model. This process is presented in Figure 3.3.

Our interests lie in the three middle steps: metric extraction, quality evaluation, and manual adjustments. We first looked at the specific metrics extracted and how they are used to produce a quality score. The general form of their quality models is illustrated in Figure 3.4. As we can see, their models are very similar to the ISO9126 product quality model. The model describes how to aggregate and combine information from metrics to quality evaluations. First, the models define combination strategies: metrics are transformed to subcharacteristics using rules and thresholds, and the sub-characteristics are combined to produce the quality score using a weighted (arithmetic) mean. As in the example, subcharacteristics are defined either at the method, class, or system-level, but these are all eventually used to assess a system-level quality characteristic. These automatic evaluations can contain errors and so the IV&V team reviews part of the code to re-adjust the scores. Our interest in this step concerns the amount of manual intervention

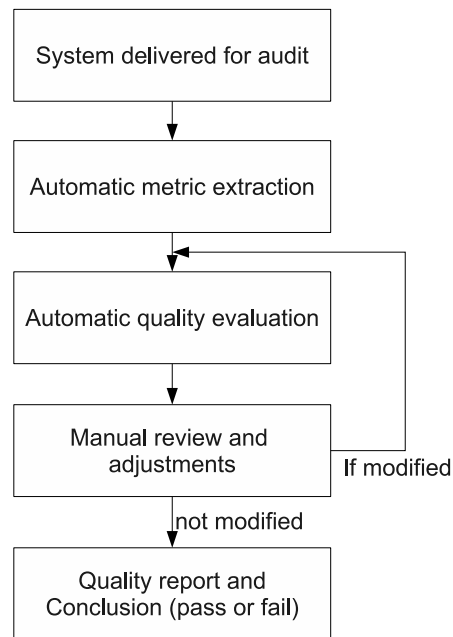


Figure 3.3: Quality evaluation process of our partner

required to finalise an audit.

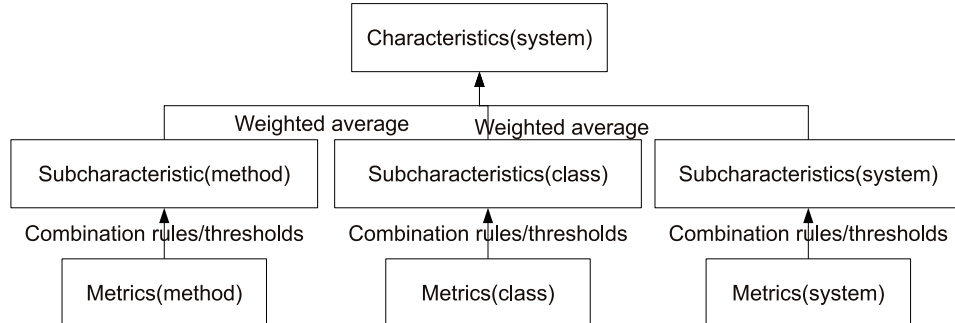


Figure 3.4: Example of industrial quality models

To illustrate the execution of the model, we present a fictitious, but representative quality model to assess the maintainability of a system in Figure 3.5. The characteristic of maintainability is decomposed into three subcharacteristics: the comprehensibility of methods, the cohesion of classes, and the modularity of the system. Every method is judged according to rules to see if its level of comprehensibility is good, acceptable, or bad which are respectively assigned the values of 1, 2 and 3. These individual values



are aggregated to produce a score for the system-wide comprehensibility using the mean value. The same process is repeated for the level of cohesion of classes. Finally, the mean values of comprehensibility, cohesion, and modularisation are combined linearly to produce a maintainability score between 1 and 3.

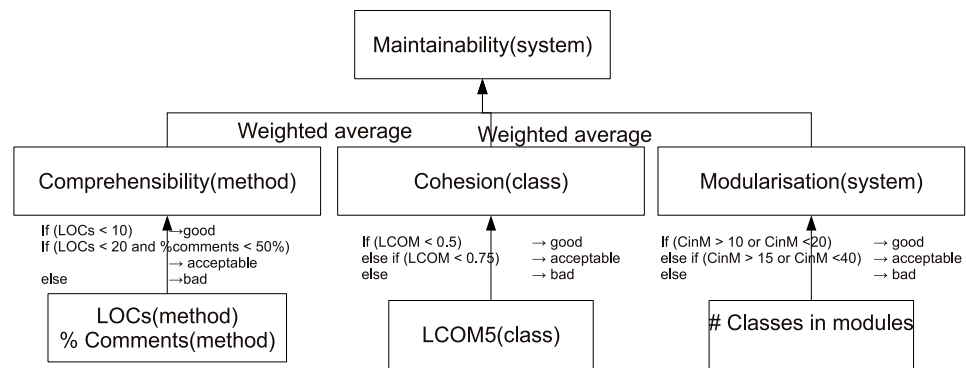


Figure 3.5: Example of a maintainability model

This model directly provides information to two types of stakeholders: the IV&V team and managers. The IV&V team validates low-level information by verifying if classes and methods with bad scores (subcharacteristic level) are indeed problematic. If there is a problem, it is transmitted to the development team for correction. Otherwise, the quality rating of the subcharacteristic is corrected and the model is re-executed. Managers use this corrected, system-level quality assessment to assist in decision-making. In extreme cases, they can reject a project that does not pass predefined quality thresholds. Note that our partner's model explicitly uses low-level quality evaluations to assess a high-level quality measure. Although this is an intuitive notion, this has not been studied explicitly in the literature.

### 3.1.2 Issues

We identified several problems with our partner's models that hamper their effectiveness. Some are theoretical, others are practical. Practical problems mostly come from the fact that these models are used to judge the performance of developers who have difficulty accepting that their performance is primarily judged using metrics. Apart from

these industrial concerns, there are also several fundamental problems in the way the models compute scores. Below, we give more details on the issues we identified. In the next section, we show that the same issues recur in publicly available systems.

- **Bad scores do not necessarily imply bad quality.** The metrics and rules used in the models are only symptoms that something is good or bad. It is possible for a long method with few comments to be understood while a shorter method might be totally unreadable. The problem is that corrections requested by the IV&V team emphasize the importance of these symptoms. For example, a correction request might state that method *X* is hard to understand because it has *Y* lines of code and only *Z%* comments. From the perspective of development teams, they see that they get a slap on the wrist when their code is structured in a certain way. Instead of addressing the general problem, they will hack their code bases to fit a model's specific parameters. For the particular case of comprehensibility, some (external) development teams started to use comment-generating templates in their development environments. However, the comments in the templates are useless. The fundamental problem is that the IV&V team *presents symptoms as if they were logical causes of quality problems*.
- **Thresholds are a source of conflict.** The use of thresholds causes problems at many different levels. First, they are used to discriminate between good and bad code. For this use, the choice of appropriate thresholds is a source of contention between the IV&V and the development teams. Development teams would like permissive thresholds and IV&V teams would like thresholds to be more conservative. Finding a balance is a delicate task that is complicated by the general nature of the models: they are used to assess different types of systems each with their particularities. Second, these same thresholds hide important information concerning the magnitude of a problem. A model using thresholds cannot differentiate between marginally bad code and catastrophic code that requires serious attention. This is also important when tracking the evolution of a system. If a quality model uses a threshold of 100 to discriminate between methods of accept-

able and bad complexity, then any method with a complexity over 100 would be considered equally bad. These thresholds thus create blind spots where the IV&V team cannot evaluate if the development team is improving or degrading the quality of the system. The final problem is that numeric values are transformed to an ordinal scale  $bad = 1, acceptable = 2, good = 3$  before being combined using an arithmetic mean. Means are not a measure of central tendency for metrics on an ordinal scale.

*The use of thresholds has a negative impact on a quality evaluation process: thresholds are not universal; they hide information for the IV&V team, and lead to inappropriate data manipulation.*

- **The model does not adequately rank problems** The models produce detailed reports of the status of every method, and class each possibly containing multiple problems. In theory, IV&V would need to manually inspect the thousands of problems detected for large systems. Given that the IV&V team has limited resources, it needs to prioritise its effort on detecting the most critical problems. The model, through its use of thresholds and averages cannot prioritise the effort of the IV&V team.

*We believe that the model should be able to rank the risk of every component to allow for an efficient inspection.*

In the following section, we investigate these issues in publicly available data sets.

### 3.2 Initial Exploration of Concerns

The quality models of our partners are built to be general: they are applied on multiple systems, written in various in different programming languages. We wanted to perform an evaluation of the importance of the different issues identified. Specifically, we focused on the following questions concerning our partner's quality model:

- **Q1** - Can we use metrics to predict quality problems, and if so, can we hope to find general quality models?

- **Q2** - What is the impact of using averages to combine metric values? How does this compare to other strategies like sums and maxima?

Q1 investigates the impact of judging work solely using metrics. If the quality model is inappropriate for the data it judges, then it cannot correctly identify high-risk code, then an IV&V team needs to spend more time reviewing its results. Q2 examines the choice of using means in our partner's quality models and explores alternatives.

For political reasons, our partner could not provide us with enough data to perform a study on their systems. Instead, we turned to publicly available data sets. We analysed data sets published by NASA, which contains the quality indicators required for a quantitative study. This data contains only a few object-oriented systems, so we present only an exploration of these problems.

The quality indicators available in the NASA data pertain to the reliability of a system: there are bug counts and density for different types of software modules. For OO systems, data is available for the class and method levels. For procedural code, the data is available for procedures. The different software entities are described only by a set of software metrics (the source is unavailable) and by quality information as entered by a development team.

### 3.2.1 Systems Studied

We analysed the two publically available object-oriented systems<sup>1</sup>: KC1 and KC3, as well as one procedural system, PC1 (for a procedure-level analysis). These systems are relatively small. KC1 is a 43 KLOCs C++ system that is a subsystem in a ground system. KC3 is an 18 KLOCs Java system that collects, processes and delivers satellite metadata. PC1 is flight software from an earth orbiting satellite that is no longer operational. It consists of 40 KLOCs of C code. The choice of systems is motivated by the assumption that a same set of metrics is useful across language boundaries. Our interests lie in OO systems, thus, we only performed a simple analysis of PC1. All systems have bugs assigned to method/procedures.

---

<sup>1</sup><http://mdp.ivv.nasa.gov/>

Metric	Definition	Description
size		Number of lines of code
comment density		Number of lines of comments/Number of lines of code
vg	McCabe [McC76]	the cyclomatic complexity of a method: the number of independent paths through the control flow graph
evg	McCabe [McC76]	the essential complexity of a method: the number of independent paths through a simplified control flow graph
ivg	McCabe [McC76]	the design complexity of a method: the number of complexity of the control flow graph that was simplified to include only paths containing invocations of external procedure;
Unique operands	Halstead [Hal77]	Counts the number of operands in a method
Halstead effort	Halstead [Hal77]	Computes the effort to develop the method based on number of operands and operators

Table 3.I: Method-level metrics studied

Metric	Definition	Description
CBO	C&K [CK91]	Coupling between objects (Sect. 2.2.4.1)
RFC	C&K [CK91]	Response set (Sect. 2.2.4.1)
WMC	C&K [CK91]	Class complexity (Sect. 2.2.4.1)
NOC	C&K [CK91]	Number of children (Sect 2.2.4.1)
DIT	C&K [CK91]	Depth of inheritance tree (Sect. 2.2.4.1)
LCOM	C&K [CK91]	Lack of Cohesion (Sect. 2.2.4.1)
Fan-in		The number of unique incoming calls from other classes

Table 3.II: Class-level metrics studied

### 3.2.2 Metrics Studied

The metrics available in the NASA data set were extracted by McCabeIQ<sup>2</sup>. The method-level and class-level metrics that we studied are presented in Tables 3.I and 3.II respectively. For our analysis, we considered only a subset of metrics in the NASA dataset for two reasons. First, our partner uses a different version of this tool, and there are differences in some metric definitions. Second, many metrics are simple compositions of other metrics. For example, Halstead defined the number of operands and operators (respectively  $n_1$  and  $n_2$ ) in a procedure/method. The Halstead “vocabulary size” metric is  $n_1 + n_2$ . We ignored the composed metrics unless used by our partner.

Some descriptive statistics are presented in Table 3.III. We can see that for most metrics, the mean is not a measure of central tendency. It tends to be closer to the 3<sup>rd</sup> quartile than the median. This suggests that the metrics follow a power-law distribution.

<sup>2</sup><http://www.mccabe.com/iq.htm>

Metric	Min.	1st Qu.	Median	3rd Qu.	Max.	Mean
LOC	1	3	9	24	288	20.39
%COMMENTS	0%	0%	0%	0%	78%	21%
vg	1	1	1	3	45	2.84
evg	1	1	1	1	26	1.675
ivg	1	1	1	3	45	2.548
unique operands	0	1	5	13	120	9.545
hal. effort	0	12	214	2278	324800	5247
CBO	0	3	8	14	24	8.32
RFC	0	10	28	44	222	34.38
WMC	0	8	12	22	100	17.42
NOC	0	0	0	0	5	0.21
DIT	1	1	2	2	7	2
LCOM	0	58	84	96	100	68.72
Fan-in	0	0	1	1	3	0.65
errors	0	0	0	0	7	0.2492

Table 3.III: Descriptive statistics for the NASA data set

### 3.2.3 Analysis Techniques

In this study, we used two types of analyses: correlations for univariate analysis<sup>3</sup>, and machine-learning predictions. The use of correlations is guided by our partner’s code inspection techniques where they choose the metric (or computed quality) to inspect the code (in order of “badness”). A rank-order (Spearman) correlation seeks to evaluate if the order of exploration is generally efficient.

### 3.2.4 Issue 1: Usefulness of Metrics

For this first issue, we investigated whether or not these general models can be applied while ensuring an acceptable performance. For this, we looked at the metrics used within these models and how well they could individually contribute to a quality model.

To evaluate the exact choice of metrics, we performed a correlation analysis at the method level. The rank correlation values between method-level metrics and the number of bugs are presented in Table 3.IV. In bold are the correlations for the best metrics. There are no strong correlations (all are  $< 0.5$ ). This indicates that a model cannot rely on a single metric to identify buggy code. However, there are many moderately correlated metrics (0.3 to 0.4) that can be useful and included in a quality model.

When comparing the usefulness of the metrics, there are no clear winners. Size

---

<sup>3</sup>we used R version 2.9.2

System	size	comments (%)	vg	evg	ivg	unique operands	Halstead effort
KC1	0.15	0.20	<b>0.47</b>	0.41	<b>0.49</b>	0.37	0.40
KC3	<b>0.44</b>	0.20	<b>0.39</b>	0.28	-0.13	0.23	0.11
PC1	<b>0.19</b>	0.12	0.13	0.06	0.15	<b>0.22</b>	0.16
All	0.06	0.13	<b>0.18</b>	0.15	<b>0.21</b>	0.11	0.08

Table 3.IV: Rank correlations between method metrics and bug count

performs well for KC3 and PC1 (meaning that the  $n^{th}$  largest class tends to be the class with the  $n^{th}$  most bugs), but size is the worst metric for KC1. Surprisingly, there is even a complexity metric (ivg) that is negatively, though weakly, correlated to the number of bugs. We assume that it is due to the programming language used. Another complexity measure, essential complexity (or evg), calculates the “unstructureness” of methods, but Java has only a limited number of constructs that can affect this metric. High values of evg will be less present than in C programs. When we combine the data from all systems (identified as All) and identify the best metrics this way, the best metrics are vg and ivg. But, as mentioned before, both these metrics have their weaknesses. This observation concurs with the assessment of Menzies *et al.* [MGF07] that different individual metrics are good for different models.

Any general metric-based model should be built by first identifying whether or not the metric is useful. Furthermore, since we noticed that different metrics are useful in different contexts, we believe that there needs to be an effort to adapt general models to new systems. In the context of our partner’s practices, this means that development teams might be correct when they demand that different thresholds and metrics be used to judge their systems.

### 3.2.5 Issue 2: Combination Strategies

We presented our partner’s problem with using weighed means to compute system-level quality estimates. We cannot test this aggregation strategy at the system-level; instead, we focused on converting method-level metrics to class-level metrics. Currently, our partner’s models use arithmetic means to combine information. However, we mentioned in Section 2.4.2 that a mean is not a measure of central tendency. For our fi-

nal investigation, we combined the seven method-level metrics using averages, maxima and sums to get a class-level metrics, as did Zimmermann *et al.* [ZPZ07] and Koru and Liu [KL05], for a total of 21 combinations. Our objective is two-fold. First, to identify what are the best metrics to predict faults. We did so by analysing correlations between all metrics and # of faults in classes. Second, we wanted to know whether method-level metrics or class-level metrics are more interesting, thus we built a fault-proneness model using method metrics, class metrics, and both, and compared their performance.

Metric	Correlation
sumHALSTEAD_EFFORT	0.63
maxNUM_UNIQUE_OPERANDS	0.63
sumNUM_UNIQUE_OPERANDS	0.61
maxHALSTEAD_EFFORT	0.61
sumLOC_TOTAL	0.60
maxLOC_TOTAL	0.58
COUPLING_BETWEEN_OBJECTS	0.58
sumCYCLOMATIC_COMPLEXITY	0.58
sumDESIGN_COMPLEXITY	0.57
maxDESIGN_COMPLEXITY	0.56
maxCOMMENTS	0.55
sumESSENTIAL_COMPLEXITY	0.55
avHALSTEAD_EFFORT	0.52
sumCOMMENTS	0.55
maxCYCLOMATIC_COMPLEXITY	0.55
avNUM_UNIQUE_OPERANDS	0.54
maxESSENTIAL_COMPLEXITY	0.51
avLOC_TOTAL	0.50
avDESIGN_COMPLEXITY	0.48
avCYCLOMATIC_COMPLEXITY	0.47
avESSENTIAL_COMPLEXITY	0.47
avCOMMENTS	0.46
WEIGHTED_METHODS_PER_CLASS	0.38
RESPONSE_FOR_CLASS	0.28
FAN_IN	0.11
DEP_ON_CHILD	0.10
PERCENT_PUB_DATA	0.07
DEPTH_INHERITANCE	0.05
LACK_OF_COHESION_OF_METHODS	0.01
NUM_OF_CHILDREN	-0.18

Table 3.V: Rank correlations between class metrics and bug count

Table 3.V presents the metrics considered and their correlations with the number of faults in KC1. We can see that class-level metrics (highlighted in gray) are clearly inferior to method-level metrics on the system. Sums and maxima have higher correlations than averages. Their importance is explainable by the notion of size: the bigger the class, the more bugs it contains.

The effect of size also explains the cases when averages present good correlations.



As most metrics follow a power distribution, the maximum value of a metric has a heavily influence on the average. If the maximum value is important (an outlier), then it pushes up the value of the mean. We can observe this phenomenon when the maximum value of a metric is less important than its sum. When this occurs, the impact of outliers on the average is less present (*e.g.*, like with cyclomatic and essential complexity). In these cases, the averages are less correlated to faults, indicating that average values are often biased because of outlier values.

Finally, we compared the performance of a fault-proneness model built using method-level metrics, class-level metrics, and both. The results are presented in Table 3.VI. To evaluate the combined effect of metrics, we used a machine learning technique called the RIPPER (for Repeated Incremental Pruning to Produce Error Reduction) rule inducer [Coh95] as implemented in Weka [WF05]. This technique learns rules from continuous or discrete input variables; the rules predict a discrete output. In our case, we classify code according to whether or not it is fault-prone (containing  $> 0$  *faults*). We chose to use RIPPER to reflect the rule-based approach of our partner.

Configuration	classification rate
Method-level only	74%
Class-level only	70%
Both	74%

Table 3.VI: Classification rates for class fault-proneness

Method-level metrics produced the best classifier; furthermore, when both type of metrics are combined, the RIPPER algorithm never used OO metrics in the rules it creates. Thus, the method-only and combination models produced are the same. The method-level model (rules presented in Figure 3.6) is sensitive to the notion of size. In fact, the only metric used is the number of unique operands. The rules search either for large methods, or for a single large method within a class otherwise containing mostly small methods. The class-level model (Figure 3.7) uses only the class-level coupling.

What we can conclude from this study is that, for this system, method-level metrics provide better information than class-level metrics for the prediction of faults. Finally, averages seem to be heavily influenced by the tail of the distribution, making them more

Rule 1: (maxNUM_UNIQUE_OPERANDS >= 22) => Fault-prone Rule 2: (maxNUM_UNIQUE_OPERANDS >= 13) and (avNUM_UNIQUE_OPERANDS <= 5.65) => Fault-prone Rule 3: default => sumERROR_COUNT= safe
---

Figure 3.6: Fault-proneness model using method-level metrics

Rule 1: (COUPLING_BETWEEN_OBJECTS >= 7) => Fault-prone Rule 2: default => sumERROR_COUNT=safe
--

Figure 3.7: Fault-proneness model using class-level metrics

a measure of dispersion than a useful measure of central tendency.

### 3.3 Research Perspectives

The qualitative and quantitative investigation of our partner’s models helped us identify a few promising research perspectives. These perspectives guided the work presented in later chapters:

**The subjective nature of quality assessment.** Our interviews raised the issue that developers and quality engineers disagree on specific metrics thresholds to use in the quality models. Our quantitative study showed that the exact metrics to include is also debatable. As we mentioned before, metrics do not measure something that directly causes faults, rather they characterise zones of excessive complexity. Some complexity can be an unavoidable as it is part of the problem; other complexity can come from inadequate solutions, but our models cannot distinguish between them. We believe we can improve quality models by focusing on how to include the quality judgments of individuals because these different judgments implicitly include their capacity to understand and manipulate complex code. In Chapter 4, we present a methodology to build such models. Our approach is evaluated empirically on the problem of anti-pattern detections.

**The composition of information from one level of granularity to another is non-trivial.** We also showed that information from methods seems more interesting than class-level metrics to assess the quality of classes. However, there are different possible ways to combine method-level information. We investigated and compared different combination strategies. Our research is presented in Chapter 6.

### **3.4 Conclusion**

In this chapter, we presented industrial problems with the evaluation of the quality of software. We based our observations on a particular case study of our partner's quality modelling efforts. We showed that an industrial partner had problems producing trouble-free quality models even though it followed state of the art practices. Some problems were identified in interviews with members of the IV&V team. At the heart of these problems are disagreements between the IV&V and development teams on how the models should be built and used. Other problems were identified in a quantitative study of NASA data. These mostly pertain to the universal nature of the models and the choice of metrics and rules. These issues serve to illustrate different barriers that affect the acceptance of quality models in industry. In the following chapters, we present our contributions to the state of practice.



## CHAPTER 4

### DEALING WITH SUBJECTIVITY

The evaluation of quality is ultimately a subjective activity. Different individuals looking at the same code might have totally different opinions, depending on their needs and expectations. However, in the context of an IV&V team, they want tools to indicate which parts of the code need to be tested and inspected. For this, an organisation needs to determine what metrics can be collected and used to build and validate its quality models. It is however difficult to define and collect metrics that fully describe a complex subjective notion of quality <sup>1</sup>.

Many organisations track only “obvious” quality indicators, like the number of faults. These metrics do not provide a complete view of quality, but most would agree that buggy code is of bad quality. The value and simplicity of fault counts explains the plethora of articles presenting fault-proneness quality models (*e.g.*, [GFS05, BTH93, KAJH05]). However, even in this narrower context (*i.e.*, reliability), there are other important aspects like fault containment and recovery that should be considered to assess software *reliability* [Mos09]. To consider these other aspects, we need to understand how a system reacts to failures, a task that is difficult to automate.

Independently of what a system does, there is a notion of *good code*: code that is structured into readable, independent modules. Many believe that this good code should impact the quality of the end product. Studies consistently show that developer productivity can vary by factors of up to 30 to 1 [FPB78, Boe81, SEG68]. To achieve this level of productivity, developers must find ways to write code that is relatively bug-free and maintainable. We therefore hypothesize that good code should lead to fewer faults and good maintainability, symptoms of good quality.

Defining what constitutes good code is also difficult: the adequacy of overall design to the nature of the task at hand requires the correct choice of objects and opera-

---

<sup>1</sup>The content of this chapter was my contribution to an article accepted for publication in the Journal of Software Systems [KVGS10]. The article in question is the extension of article from the International Conference on Software Quality 2009 [KVGS09].

tions. All this requires intelligence and experience. Even the evaluation of code quality is subjective. Different individuals might have different ideas of what good code is. Furthermore, even with a consensual definition, their judgments might vary in magnitude (*e.g.*, an A- vs. B+ grade). Tempered judgments might be more acceptable to individuals who object to binary thresholds.

In this chapter, we present different techniques to model subjectivity and propose an approach to predict subjective developer opinions. Our approach uses Bayesian models to describe an expert's evaluation process and can be trained on subjective data. Finally, we present the application of this approach to the problem of anti-pattern detection.

## **4.1 Existing Techniques to Model Subjectivity**

To support subjectivity in a quality evaluation process, we must deal with uncertain decisions: for a given system, the quality might be judged good or bad depending on who performs the evaluation. Existing research has focused on two main categories of techniques to deal with uncertain reasoning: those based on fuzzy logic and those based on Bayesian inference. Fuzzy logic assesses the degree that a system conforms to quality rules. Bayesian inference uses probabilities to model uncertainty. Given a characterisation of a module, it estimates what is the probability the system is good/bad. The problem is consequently one of classification, and therefore, there is no explicit notion of magnitude. In the following sections, we discuss these techniques and the cases when they are appropriate.

### **4.1.1 Fuzzy Logic**

In the literature, many heuristics have been proposed to recognize good quality software [Rie96]. These are typically formulated as rules, but tend to be difficult to identify automatically. An example is the well-known design heuristic for software modules: low coupling and high cohesion. This heuristic is difficult to apply because the notions of high and low are vague: going from a precise measurement to a judgment is non-trivial. Fuzzy logic can be used to model this vagueness. In traditional logic, we assign crisp

values of truth (*false, true*) to different observations (*e.g., coupling(module) = high*). Fuzzy logic goes beyond traditional logic by assigning “degrees of truth” to propositions. In our example, a module could be considered more or less part of the high coupling set (*e.g., high module\_coupling = 75%* and *low module\_coupling = 25%*). By combining fuzzy measures (*e.g., metrics defined by degrees of low and high*), we could evaluate a degree of goodness/badness of a system [EL07]. If we were to apply the fuzzy set operator *AND*, the quality of a module might be:

$$\min(\text{low module\_coupling}, \text{high module\_cohesion})$$

The first step in a fuzzy quality evaluation is the transformation of numeric values to their membership in fuzzy sets; this process is called fuzzification. This allows continuous data to be converted to an ordinal scale, *e.g.* low, medium, or high. This process is illustrated in Figure 4.1. For the second step, the evaluation process would apply a series of operations on this data [KY95] using fuzzy operators like *ANDs* or *ORs*. The result would be a quality score represented as a fuzzy membership (*e.g., quality(module) = good, 25%* and *quality(module) = bad, 75%*). This membership could be defuzzified if we need a crisp value.

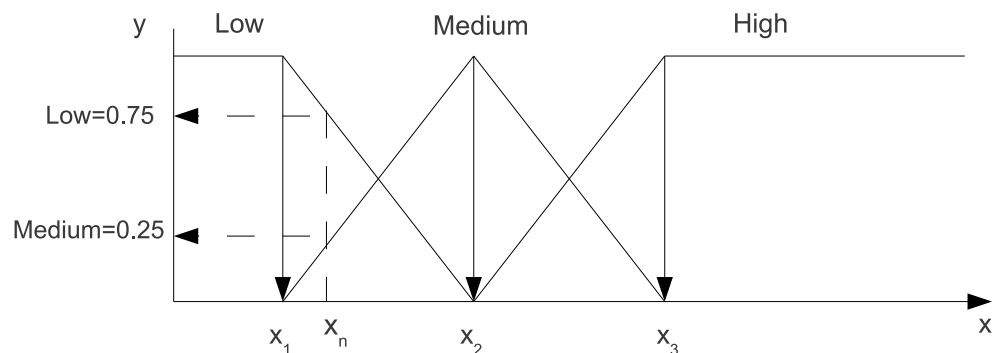


Figure 4.1: Fuzzification process of a metric value  $x_n$ . The value of  $x_n$  is calculated by its distance to its nearest fuzzy thresholds ( $x_1$  and  $x_2$ )

Fuzzy logic has been used to assess software quality in [SCK02, MB09]. We believe that fuzzy logic can represent vague data and a vague evaluation process. There are two

problems with fuzzy logic for quality assessment. First, it supposes that the heuristics are accurate. Second, even though the output quality score is vague, it is assumed that it is universal (higher value = better code). As mentioned before, quality evaluations are not universal, and depend on individuals. This reason indicates that fuzzy logic is not appropriate to handle subjectivity. It is however interesting for the manipulation of metrics because fuzzification supports the transformation of continuous data to an ordinal scale.

#### 4.1.2 Bayesian Inference

Bayesian inference depends on a description of quality in probabilistic terms. The quality of a system ( $Q$ ) is generally represented as a conditional probability given a probabilistic description of its attributes ( $A$ ), or  $P(Q|A)$ ; in this representation, all uncertainty is represented using probabilities. We could restate our previous example as  $P(\text{quality} = \text{high} | \text{Coupling}, \text{Cohesion})$ .

In order to use Bayesian inference, we need to transform metrics describing these attributes to probability distributions, *e.g.*  $P(\text{coupling} = \text{low})$  and  $P(\text{cohesion} = \text{high})$  given a metric value, and describe the effect of combining these probabilities on quality. This technique is known to be more robust than fuzzy logic when we have an idea of the metrics' distributions or when we have access to a large number of observations (historical data) [CNC<sup>+</sup>99]. When this is the case, we can accurately describe the probabilistic relationship between measurable attributes and quality. Previous work used Bayesian theory to support uncertainty in quality related decision making [PSR05, FN99], but did not specifically address subjectivity.

To support subjectivity, we consider that probabilities should model the odds that a stakeholder would agree with a quality assessment. This means that if we have  $P(\text{quality} = \text{high}) = 90\%$ , nine out of ten stakeholders would agree that quality is high. Where fuzzy logic tries to assess a universal notion of quality (although vague), a probabilistic approach focuses on the probability that a specific quality evaluation is consensual. Therefore, a higher probability does not necessarily mean that a module is of better quality; only that more people would agree that a module is of good quality. The difficulty of



using Bayesian inference lies in converting metrics to probability distributions and their subsequent manipulation. Many metrics follow power-law distributions and existing tool support generally only supports normally distributed data. We consequently need to find an adequate way to handle these metrics. However, our goal is to deal with diverging opinions, and a Bayesian approach is, in our opinion, best suited for this task.

## 4.2 Evaluating the Quality of Design

Software engineers constantly make decisions that are reasonable solutions to the problems they are trying to solve, and there are many viable alternative solutions. However, a bad solution can cause a lot of grief to a development team. One such area where this is particularly true is software design. Identifying design problems early is important, but requires developers to evaluate if a design is inappropriate for the problem they are trying to solve.

In this section, we present the problem of *anti-pattern* detection, which we consider an evaluation of the quality of the design of classes. Anti-patterns are recurring design solutions that are considered harmful to the quality of a software system (mostly concerning its maintainability). These harmful solutions are called anti-patterns because they are typically presented following a template [BMB<sup>+</sup>98, Fow99] describing the problem they pose, the symptoms that can be used to detect the problem, as well as standard correction strategies. The symptoms can be used to identify metrics that can be used to automate a detection process. We propose a Bayesian approach to support this detection to assess the probability that a developer would consider that class is an anti-pattern. This approach is tested and compared empirically to a state of the art rule-based detection technique.

### 4.2.1 Industrial Interest in Anti-pattern Detection

A part of the partner's IV&V process focuses on the detection of design flaws. In their quality model, the IV&V team included rules to detect these flaws, using an approach similar to that of Marinescu [Mar04]). These detection models proved to be a

burden as development teams argued that both thresholds and the results were inaccurate. The underlying problem was that the IV&V team encoded what they thought were good, universal rules, but did not consider the possibility that others might disagree. In fact, there are many possible solutions and ways to organise code in a successful program, some contradict the rules implemented in the detection models.

Through our collaboration, our partner had access to DECOR, a state-of-the-art anti-pattern detection tool that is also based on rules [MGDM10]. The rules define sets of classes sharing similar characteristics. Each set corresponds to one or many symptoms that are indication that a class might be an anti-pattern. These rules describe either linguistic (class and method names) or structural (metrics and associations) characteristics. These are combined using Boolean set operations and a special operation to describe association relationships to combine information from different classes. The result is a set of candidate classes. An example of detection rules is shown in Figure 4.2.

```

RULE\_CARD : FunctionalDecomposition {
  RULE : FunctionalDecomposition { UNION FunctionMethodClass
    FunctionalAssociation};
  RULE : FunctionMethodClass {(SEMANTIC: METHODNAME, {end, check,
    validate, traverse, prepare, report, configure, init, create...}) } ;
  RULE : FunctionaAssociation {ASSOC: associated FROM: MainClass ONE TO: aClass MANY };
  RULE : MainClass { INTER NoPolymorphism NoInheritance };
  RULE : NoInheritance {(METRIC: DIT, SUP\_EQ, 1, 0) };
  RULE : NoPolymorphism { (STRUCT: DIFFERENT\_PARAMETER) };
  RULE : MainClass { INTER NoPolymorphism NoInheritance };
  RULE : aClass {INTER ClassOneMethod FieldPrivate};
  RULE : ClassOneMethod {(STRUCT: ONE\_METHOD)};
  RULE : FieldPrivate {(STRUCT: PRIVATE\_FIELD, 100) };
};

```

Figure 4.2: Functional decomposition detection rules

Briefly, in this example, a class is a *Functional Decomposition* anti-pattern if 1) one of its methods contains terms like end, check, etc (FunctionMethodClass rule), or 2) if it uses no inheritance (MainClass rule) and is associated to at least a class defining only one method and which contains only private fields. The anti-patterns correspond to heuristics described by Brown *et al.* [BMB<sup>+</sup>98].

Our partner found that the tool was not suited to their environment for the following reasons:

- **The number of false-positives was too important for the team to efficiently**

**identify real problems.** Moha *et al.*[MGDM10] focused on finding all possible defects in a system (maximising recall). When DECOR was applied to Eclipse: they found 2412 candidates on a system with over 9000 classes. The effort of reviewing this list is too important for the approach to be usable in an industrial setting. This problem is not limited to open-source systems as this was also observed by our partner when applying DECOR to their systems.

- **The rules defined were not appropriate for modern systems.** The rules are relatively imprecise and do not correspond to modern, industrial programs. For example, to detect functional decompositions (Figure 4.2), a key rule is identified by the label `FunctionMethodClass`: any class that declares a method containing the term “create” is flagged as a defect. Such a rule labels many proper design patterns like Factory classes, as problems, yet, design patterns are generally examples of “good design”.
- **The anti-patterns are not presented in order of importance.** Having a large number of results is less of a problem if the results are ranked because the IV&V team could then inspect and validate classes in order of risk. The use of simple unions and intersections does not allow for such a process. Consequently, a class that presents all of the symptoms described would have the same importance as a class presenting the minimum necessary conditions.

A fundamental problem with DECOR is that they use hard thresholds, and consequently, the use of fuzzy logic, as in [AS06], could have addressed some of the problems identified by our partner. However, this would not have solved the problems concerning the conflicting points of view between the quality and the development teams. Since we had access to anti-pattern data, indicating the opinions of different developers on two systems, we opted for a Bayesian approach.

In previous work, we adapted DECOR to use a Bayesian approach [KVGS09]. We defined equivalencies to its rules and applied machine learning techniques to train the detection model. In this section, we present our later work: a methodology to build the models directly from a catalogue of anti-patterns [BMB<sup>+</sup>98].

### 4.2.2 The Subjective Nature of Pattern Detection

Our partners were not the first to notice the many problems with detecting anti-patterns. In [DSP08], Dhambri *et al.* performed an experiment to test visualisation techniques to locate anti-patterns. Reusing DECOR's manually annotated replication data as a baseline, they found that their subjects (and themselves) disagreed with many evaluations.

The DECOR corpus contains multiple types of anti-patterns. They had asked five groups to tag instances in different open-source systems, Xerces and Gantt. When 3/5 groups identified the same class, it was retained as an anti-pattern. We asked two students with industrial experience and good knowledge of anti-patterns to independently review the corpus. These students only agreed with the original corpus in 30% of cases, and had a similar rate of disagreement with each other. The reason for these disagreements mostly concerned two aspects: the students used different (implicit) symptoms to tag anti-patterns and the students had different ideas on alternate design, *i.e.* would they have designed the system differently. When there were agreements there was a strong structural justification, mostly pertaining to size. We concluded that the problems were due to the interpretation of the intentions of developers (semantics), not actual structural symptoms.

### 4.2.3 Detection Methodology

We decided to apply a goal oriented approach to the detection of anti-patterns, following the GQM approach [BW84]. Following the work of Moha, we determined how three types of anti-patterns could be identified. These were: *Blobs*, large classes that do too much; *Spaghetti Code*, large, badly structured classes; and *Functional Decomposition*: classes that are only built to provide functions. For every anti-pattern, our goal is its detection. The questions correspond to the different symptoms that are described in the anti-pattern reference by Brown *et al.* [BMB<sup>+</sup>98]. Finally, we selected metrics to evaluate every question. The choice of metrics was guided by two sources of information. First, if Brown *et al.* described a specific metric, then it was chosen. If not, we

<b>Goal: Identify Blobs</b>		
<b>Definition: Class that knows or does too much</b>		
<b>Question</b>		<b>Metric</b>
<b>B1</b>	Is the class a large class?	Number of methods and attributes declared
<b>B2</b>	Is the class a controller class	Presence of names indicative of control: Process, Manage... (terms used in [MGDM10])
<b>B3</b>	Is the class not cohesive?	LCOM5 is high
<b>B4</b>	Does the class use data classes	Number of classes used that contain 90% accessors

Table 4.I: GQM applied to Blobs

<b>Goal: Identify Spaghetti Code</b>		
<b>Definition: Code that does not use appropriate structural mechanisms</b>		
<b>Question</b>		<b>Metric</b>
<b>S1</b>	Does the class have long methods?	maximum LOCs in methods is high
<b>S2</b>	Does the class have methods with no parameters?	# methods with no parameters is high
<b>S3</b>	Does the class use global variables?	# global variable access is high
<b>S4</b>	Does the class not use polymorphism?	% of non-polymorphic method calls is high
<b>S5</b>	Are the names of the class indicative of procedural programming?	presence of names <i>Process, Init, Exec, Handle, Calculate, Make...</i> (terms used in [MGDM10])

Table 4.II: GQM applied to Spaghetti Code

reused metrics used by Moha *et al.* in their rules. If they ignored the symptom, then we defined our own metric. The result of our methodology is presented in Tables 4.I, 4.II and 4.III.

#### 4.2.3.1 Operationalising the Model

In Figure 4.3, we present the detection model corresponding to the Blob. There are three levels to the detection process. The bottom corresponds to the metrics extracted; the middle, to the questions, and the top to the goal. Between every level, there is an operation that is defined. To pass from a metric to a symptom, we need to transform a metric to a probability distribution. To pass from a symptom to a quality assessment, we apply Bayesian inference (*i.e.*,  $P(\text{Blob}|\text{Symptoms})$ ). From the perspective of a quality engineer or a developer, he would likely not want to know if a class is a Blob as does DECOR, but rather what are those that present the highest risk of being a Blob..

Goal: Identify Functional Decompositions		
Definition: Object-oriented code that is structured as function calls		
Question	Metric	
F1	Does the class use functional names?	same as Q5, in Table 4.II
F2	Does the class use object-oriented mechanisms (no inheritance, no polymorphism)	# overridden methods > 0 or <i>DIT</i> > 1
F3	Does the class use classes with functional names?	% of invocations to classes/methods with functional names (like Q1)
F4	Does the class declare a single method?	# methods declared = 1
F5	Are all the class attributes private?	% of private attributes = 100%

Table 4.III: GQM applied to Functional Decomposition

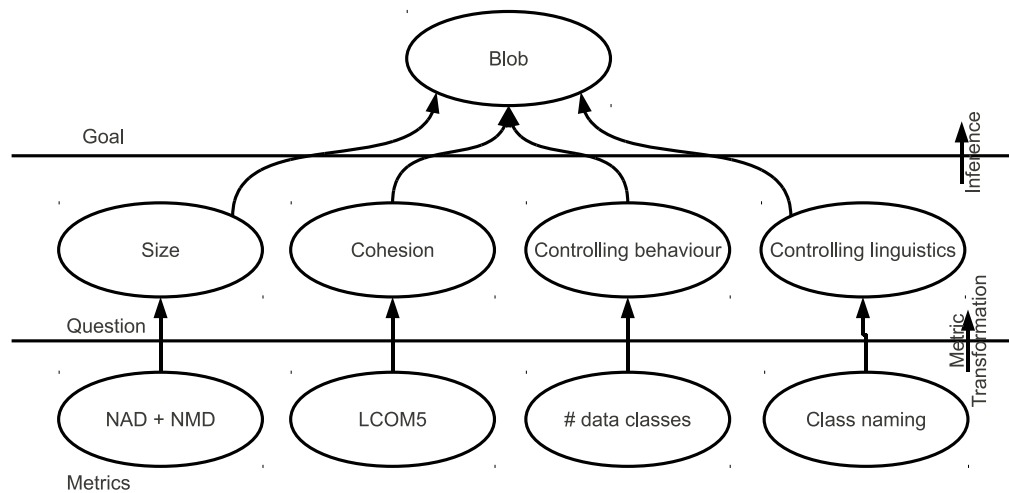


Figure 4.3: GQM applied to the Blob

**Converting Metrics to Distributions** To compute the probability distributions of the symptoms, we first discretised metrics values into three different levels: “low”, “medium”, and “high”. We used a box-plot to perform the discretisation. A box-plot, also known as a box-and-whisker plot, is used to single out the statistical particularities of a distribution and allows for a simple identification of abnormally high or low values. It identifies any value outside  $[Q1 - 1.5 \times IQ, Q3 + 1.5 \times IQ]$ , where  $Q1$  and  $Q3$  are respectively the first and third quartile, and  $IQ$  is the inter-quartile range ( $Q3 - Q1$ ). Figure 4.4 illustrates the box-plot and the thresholds that it defines:  $LQ$  and  $UQ$  correspond respectively to the lower and upper quartiles that define thresholds for outliers ( $LOut$  and  $UOut$ ).

For each class in a system, and each symptom, the probability that the class presents

the symptom is computed as follows:

- For symptoms captured by metric values, the probabilities are calculated as follows: we use three groups (“low”, “medium”, “high”) and estimate the probability that a quality analyst would consider the metric values as belonging to each group. Limiting the number of groups to three simplifies the interpretation of the detection results. For each metric value, the probability is derived by calculating the relative distance between the value and its surrounding thresholds like with a fuzzification process. The probability is interpolated linearly as presented in Figure 4.4. In this Figure, we present the probability density function corresponding to our statistical analysis of metric values.

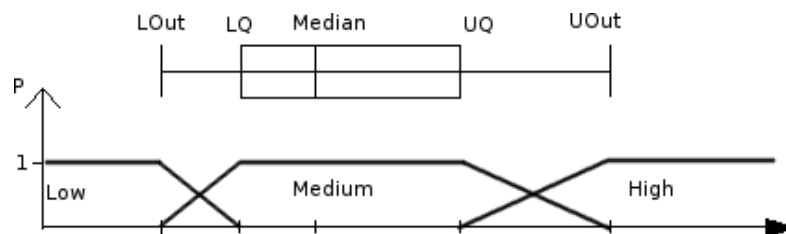


Figure 4.4: Probability interpolation for metrics

- For symptoms describing class names, probabilities are either 0 or 1, whether the name contains a term or not. For method names, we treated the number of methods containing the term as a metric and used the box-plot to interpolate a probability. According to this rule, the probability that a class named MakeFoo has a functional name is 1 ( $P(\text{FunctionalName} = \text{True}) = 1$ );
- For symptoms that determine the strength of relationships, the probabilities are calculated using the numbers of such relations, (*e.g.* the number of data classes with which a class is associated). The more a class is associated to data classes, the more likely it is a Blob. In our example, we consider that classes with over 90% of their methods that are accessors (return or set the values of an attribute), are data classes. To convert this count to a probability distribution, its value is interpolated between 0 and  $N$  where  $N$  is the upper outlier value observed in the

program. If the upper outlier value of # of data classes in a system were 11, then a class associated to 6 data classes would have a probability of 50% to be considered a controller ( $P(\text{ControllingBehaviour} = T) = 50\%$ ).

**Evaluating Anti-pattern Probabilities** The probability of a class being an anti-pattern is inferred from the probabilities of symptoms using Bayes' theorem. Every output node has a conditional probability table to describe the decision given a set of inputs. In our example, the probability of a class being a Blob depends on four symptoms. We can use previously tagged data to fill a conditional probability table describing all possible combination of symptoms, *i.e.*  $P(\text{Blob} | \text{Size} = \text{high}, \text{Cohesion} = \text{low}, \text{ContrBehav} = \text{high}, \text{ContrLing} = T)$ . When executing the model, the actual probability distribution of all symptoms will be used to evaluate the probability of a class being a Blob (4.1) as we assume the independence of all symptoms.

$$P(\text{Blob}) = \sum_{\text{Symptoms} \in \text{all combinations}} P(\text{Blob} | \text{Symptoms}) \times P(\text{Symptoms}) \quad (4.1)$$

#### 4.2.4 Evaluating our Bayesian Methodology

To empirically validate our approach, we followed a three step approach. First, we built our corpus in such a way as to support multiple, contradictory opinions. Then, we performed an analysis of the different symptoms presented earlier and evaluated their usefulness. Finally, we executed our detection models to compare their performance to DECOR.

##### 4.2.4.1 Building an Uncertain Corpus

To test our approach, we used the corpus of DECOR. This corpus contains a set of classes that are instances of every anti-pattern type (as judged by 3/5 groups). Additionally, we had the opinions of 2 additional developers concerning those classes. To obtain a corpus that supports conflicting opinions, we considered that every opinion is a vote



(with data in DECOR counting as 3 positive votes). Thus, if a class tagged by DECOR was also found to be an anti-pattern by one of our developers, then it would have 4 votes for, and one vote against. All classes that were not part of the corpus had 3 negative votes as our developers did not review them.

We could not encode this information directly in our model, thus we proposed the use of “bootstrapping”, a technique that multiplies existing data in a corpus. Our class with 4 positive votes would thus be included 4 times as a yes, and once as a no. If that class were the only class considered in the corpus, its symptoms would consequently have a probability of 80% of being evaluated as an anti-pattern. Finally, since we cannot be sure that our prior knowledge ( $P(Q)$ , unconditionally) is similar across projects, we balanced our corpus to have an equal number of positive and negative classes by randomly sampling our data set.

#### 4.2.4.2 Systems Analysed

Programs	# Classes	KLOCs
GanttProject v1.10.2	188	31
Xerces v2.7.0	589	240
<b>Total</b>	<i>777</i>	<i>271</i>

Table 4.IV: Program Statistics

We used two open-source Java programs to perform our experiments: GanttProject v1.10.2, Xerces v2.7.0, presented in Table 4.IV. GanttProject<sup>2</sup> is a tool for creating project schedules by means of Gantt charts and resource-load charts. Xerces<sup>3</sup> is a family of software packages for parsing and manipulating XML.

We chose GanttProject and Xerces because DECOR had annotated the corpus. Furthermore, they were small enough so that students could understand the general software architecture in order review the annotations. We used the POM framework [GSF04] to extract metrics.

---

<sup>2</sup><http://ganttproject.biz/index.php>

<sup>3</sup><http://xerces.apache.org/>

Anti-patterns	Symptoms	% in GanttProject		p-values	% in Xerces		p-values
		AP	Not AP		AP	Not AP	
Blob	B1	100%	4%	✓	94%	3%	✓
	B2	0%	7%	×	5%	1%	53%
	B3	0%	26%	×	0%	19%	×
	B4	72%	8%	✓	63%	48%	✓
S.C.	S1	100%	6%	✓	89%	6%	✓
	S2	49%	3%	✓	19%	10%	70%
	S3	38%	4%	✓	12%	2%	19%
	S4	0%	44%	×	27%	49%	×
	S5	0%	2%	×	36%	18%	✓
F.D.	F1	75%	47%	✓	87%	18%	✓
	F2	70%	39%	✓	60%	19%	✓
	F3	6%	8%	93%	67%	26%	✓
	F4	63%	17%	✓	7%	8%	92%
	F5	6%	4%	87%	0	3%	×
	F6	75%	35%	✓	33%	57%	7%

Table 4.V: Intra project validation, salient symptom identification

#### 4.2.4.3 Salient Symptom Identification

As mentioned before, some symptoms used by DECOR are inadequate in an industrial context. Figuring out which symptoms are useful can allow an IV&V team to tweak their detection models. In this section, we identify which symptoms are useful in a detection process.

We decided to use a simple univariate proportion test to identify the interesting symptoms for anti-patterns in both systems. The symptoms tested correspond to those presented in Tables 4.I, 4.II and 4.III. The test evaluates if the difference in the proportion of anti-patterns and non anti-patterns is statistically significant (not due to chance). When the difference is significant and the symptom is more present in anti-patterns, then it is a useful detector for that system.

We applied this approach to both systems and the results are presented in Table 4.V. The table contains the different proportions measured and the significance. When a symptom was significant ( $\leq 0.01$  level), it is presented by a ✓; when it was not significant, the exact value is shown; and when the relationship is the contrary of what is expected, we use a × symbol. The first thing we notice is that there are many inappropriate symptoms. There are two possible explanations: the measure used might be incorrect, or the symptom itself might not apply. In the case of symptom B3 for Blobs, we are interested in using cohesion of a class, but the LCOM5 measure is not useful.

As mentioned in Section 2.4.3, although there are many alternative measures, none in the literature are shown to be a better measure of cohesion. The second thing we notice is that there are a large number of symptoms that are useful only for one system; their importance is context-dependent. Naming convention is such a case (S5 and F3). The terms used are only useful for predictions on Xerces. Obviously, the development teams followed different coding practices. Finally, we observed that a maximum of two symptoms are useful for every model. These symptoms are thus the least context-dependent symptoms. The conclusion of the analysis of symptoms is that simply encoding heuristics found in the literature is not a good idea to produce a general purpose detection model.

This conclusion leads us to our empirical validation. We tested two scenarios. First, we tested a general approach for which we built our models using data on one system and tested it on the other. This corresponds to the approach of an IV&V team that uses general knowledge to detect anti-patterns in a new system. The second scenario evaluates the importance of local data. We built a model for a system using only symptoms that affect that system, and training on the local data (we used 3-fold cross-validation). This scenario corresponds to an IV&V team tracking a system over a long period of time during which they could adapt their prediction models.

#### **4.2.5 Scenario 1: General Detection Model**

For this scenario, we assumed that a quality analyst is reviewing a system for the first time, and only has access to historical data from another program. He would therefore use this data to build the model using this and apply the model to the other program. This scenario is similar to that of DECOR and consequently our results are compared to theirs.

A quality engineer needs to try to find the most anti-patterns possible with a limited amount of resources. Consequently, we are interested in evaluating the performance of our approach according to how useful it is given this constraint. We used two metrics to assess the quality of our models: precision and recall. Precision is the rate of true positives returned in a candidate set ( $\# \text{ anti-patterns in candidate list} \div \# \text{ candidates}$ ); it is

Class	P(S.C.)
org.apache.xerces.dom.CoreDocumentImpl	99.89%
org.apache.xerces.dom.AttrImpl	99.88%
org.apache.xerces.impl.XMLEntityScanner	99.31%
...	
org.w3c.dom.xpath.XPathException	0.11%




Figure 4.5: Inspection process for Bayesian inference

a measure of how efficiently quality engineers spend their time. Recall is the proportion of anti-patterns returned in the candidate set ( $\# \text{ anti-patterns in candidate list} \div \# \text{ anti-patterns in the system}$ ); it is a measure of how exhaustive is a model. We also included the harmonic average of precision of recall called the balanced F1 measure.

Our approach does not directly return a set of candidates; instead, it returns a list of all classes with their associated probability of being considered an anti-pattern. A quality engineer would consequently investigate the most probable anti-patterns first, and continue until he decides that he has seen enough (see Figure 4.5). DECOR, on the other hand returns a fixed sized set. Our analysis is focused on precision as it indicates how efficient is the model at optimising the resources of an IV&V team. Moreover, we are interested in the first candidates as these are the classes that are the most likely to be examined by a quality engineer.

For our first scenario, the results are presented in Figure 4.6. On the X axis, we have the number of classes inspected by a quality engineer, and on the Y axis, we indicated the corresponding recall and precision for the detection of unambiguous instances of anti-patterns (5 votes) as these would be the results that would most likely satisfy any quality analyst. The exception is for Functional Decompositions on Xerces because it had no instances with 5 votes. For that program, we computed the precision and recall for the instances with 3 or more votes.

In five cases, our system was superior to DECOR: the precision was better for the first inspected classes. Furthermore, all faulty classes were found more efficiently, requiring quality engineers to inspect a set half the size of that of DECOR. This is also reflect in the F1 measure, which indicates that, in general, our approach also provides better balanced result sets. We observed the same phenomenon with the detection of Spaghetti Code on

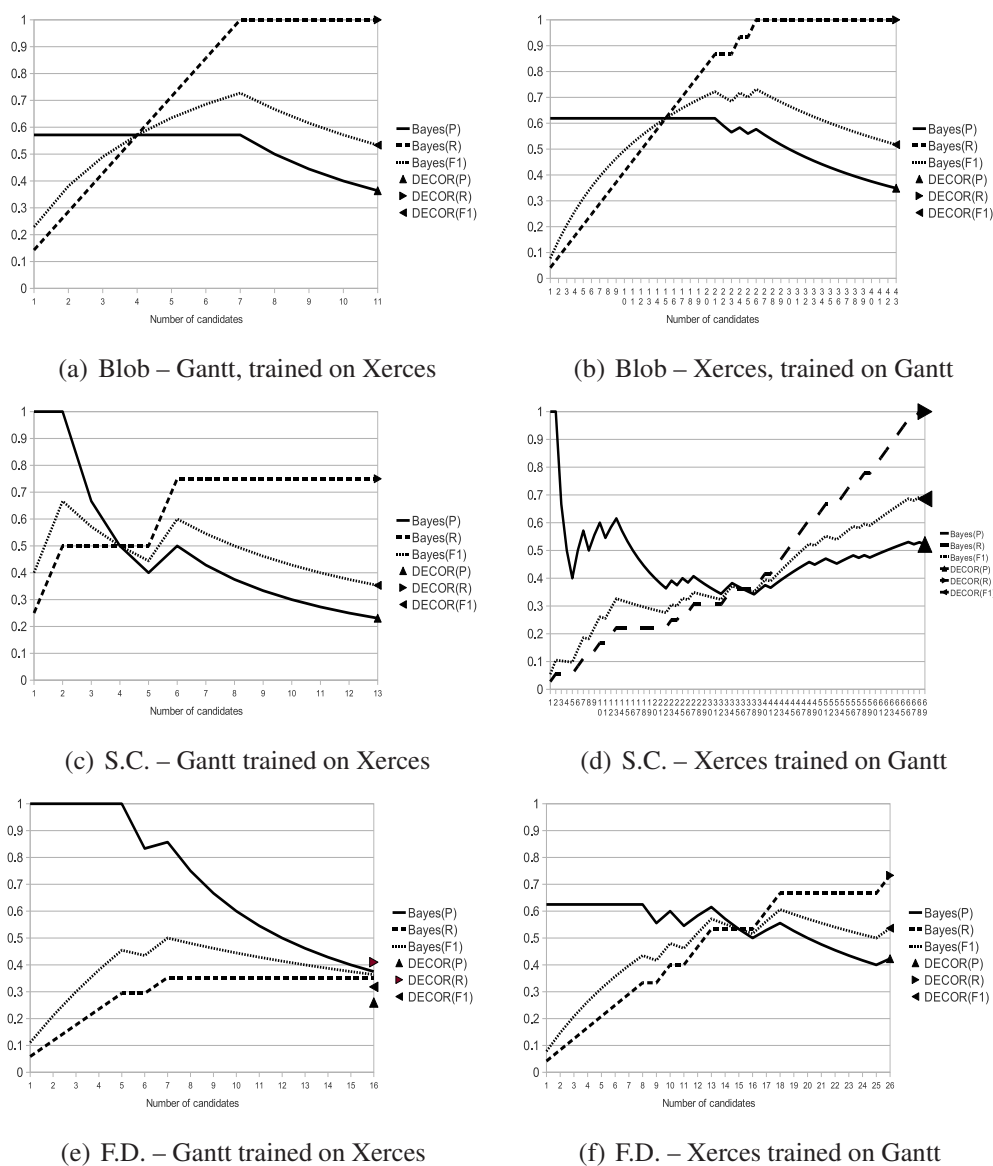


Figure 4.6: Inter-project validation

Gantt. This model did not fare so well on Xerces. The symptom set used was inadequate as our precision is consistently inferior to that of DECOR. However, for the same number of candidate classes, our precision/recall is equivalent. Finally, the Functional Decomposition model produced mixed results. For Xerces, the models provided good precision and recall. For GanttProject, the first five candidates were correctly classified.

These are promising results because they suggest that even in the absence of histori-

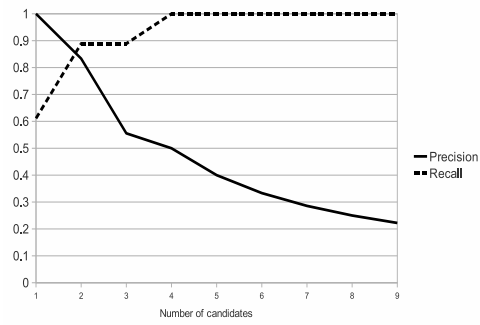
cal data for a specific program, a quality analyst can use a Bayesian approach calibrated on different programs and obtain an acceptable precision and recall, and outperform DECOR. These results also show that a model could be built using data external to a company and then be adapted and applied in this company successfully. These models could also be enriched with locally salient symptoms to further improve their accuracy.

#### 4.2.6 Scenario 2: Locally Calibrated Model

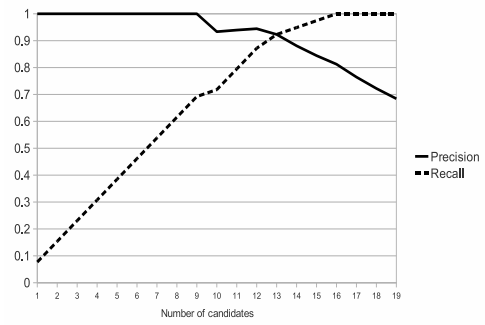
This scenario occurs when we know which symptoms are important. It is actually rare that an IV&V team audits a new project. Most of the time, audits are recurrent as they are part of an iterative development cycle. In this context, it is realistic to have a model built using local data. Therefore, we used local data to build and test the detection models. In this scenario, we compare these results to those produced in Scenario 1. This scenario should represent the best possible setup given the symptoms we selected while the previous scenario corresponds to a base line.

We built these local models using 3-fold cross-validation. Cross-validation is a technique to avoid testing on the same data as we use to calibrate our models, and avoid the problem of overfitting models. It requires each model to be built and tested multiple times using different combinations of our data set. In our case, the data set was split into three equal-sized groups. For every run, the model used a different pair of groups for training and the third group was used for testing. The precisions and recalls presented here are the averages of the three runs.

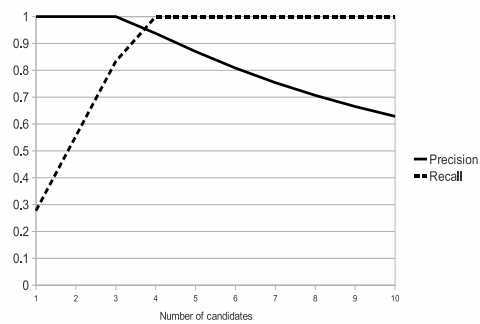
All the results for the local project validation are presented in Figure 4.7. Each sub-figure shows the average precision/recall curves for each combination of anti-pattern for each program. The results indicate that local knowledge should be used whenever possible as the performance of every detection models is significantly increased compared to Scenario 1. The identification of Blobs and Spaghetti Code is very accurate and includes only a few false positives (all precisions of over 65%). We assume that this precision is due to the notions of size, which are very important symptoms. Size is one of the few concepts that can be measured directly unlike heuristics used to identify the developers' intent (*e.g.*, words that could indicate procedural thinking). The lack of semantics in the



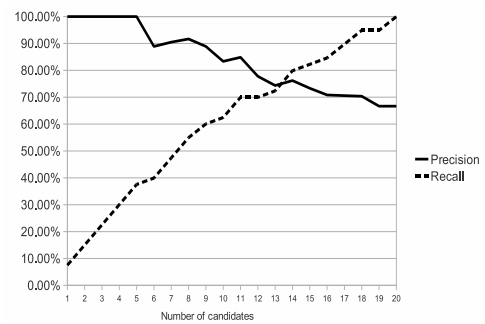
(a) Blob: Gantt



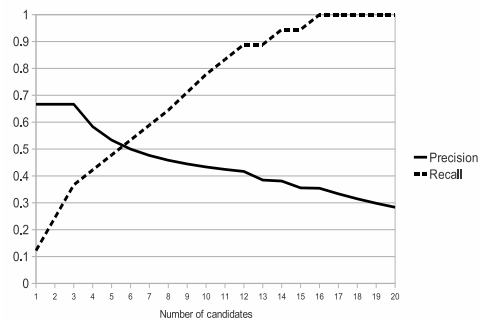
(b) Blob: Xerces



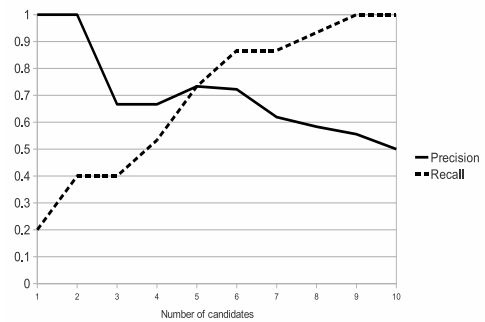
(c) Spaghetti Code: Gantt



(d) Spaghetti Code: Xerces



(e) Functional Decomposition: Gantt



(f) Functional Decomposition: Xerces

Figure 4.7: Local calibration: average precision and recall

metrics used to detect Functional Decompositions explains the general poorness of the models, even when using the best symptoms.

### 4.3 Discussion of the Results

We now discuss the experiments and the use of a Bayesian approach by quality analysts based on the results of the experiments.

#### 4.3.1 Threats to Validity

We showed that our models are able to efficiently prioritise candidate classes that should be inspected by a quality analyst. The models, built using global data, can successfully identify occurrences of anti-patterns. This is regardless of the nature of the program: Xerces is a XML parser library, and Gantt is a full-fledged GUI; both were developed by different development teams. Although these systems are open-sourced, we have no reason to believe that closed-source systems would behave any differently. Quality analysts in an another context should be able to build a repository of anti-patterns on a set of programs and use this data to calibrate models on another context. The customisation that we performed in the experiments was minimal, but improved detection significantly. In an industrial setting, where audits are performed regularly, this customisation could be more important.

#### 4.3.2 Dealing with Good Programs

Another issue with our detection models concerns their application to “good” programs. To test how many false positives are returned when applied to a healthy system, we applied our models to JHotDraw. JHotDraw<sup>4</sup> is a GUI framework for structured graphics (176 classes). It was designed as a design exercise by object-oriented experts. It is a good reference of a program that most likely does not contain any occurrence of anti-patterns.

Table 4.VI presents the number of classes that present all symptoms of the three anti-patterns. These numbers indicate that only a few candidate classes need to be inspected by quality analysts.

---

<sup>4</sup><http://www.jhotdraw.org>



Anti-Patterns	‡ Inspected Classes
Blob	1
S.C.	2
F.D.	0

Table 4.VI: JHotDraw: inspection sizes

The reason why the models return these false positives is the discretisation methods used, which uses relative sizes (based on quartiles). For the detection of Blobs and Spaghetti Code, size is an important symptom. Even in a program with generally small classes (like JHotDraw), the models identify those that are relatively large.

It is possible for an organisation to impose global thresholds in the detection process. These thresholds could be extracted from an analysis of a large number of systems. If we had used the relative thresholds values measured on Xerces and GanttProject to identify occurrences of the anti-patterns in JHotDraw, there would have been no candidate classes corresponding to the highest level of probability.

### 4.3.3 Estimating the Number of Anti-Patterns

The problem with presenting a large list of candidates rather than a set is that it is up to the quality engineer to decide when to stop looking. This concern affects all techniques that rank results. Typically, we could sum the expectation of probabilities that each class is considered an anti-pattern. The models presented tend to overestimate the number of anti-patterns by 75-200%. This is due to the treatment of the metrics. To correctly estimate the number of anti-patterns we would need to consider the precise distribution of the metrics (mostly power-laws), instead of using a discrete representation. There are two problems with this treatment. First, most tools that implement Bayesian inference do not handle power-laws. Second, the resulting model would be hard to interpret by a quality analyst unlike conditional probability tables. In future work, we consider moving towards using continuous probabilistic variables to describe our metrics.

#### 4.3.4 Alternative Code Representation

In collaboration with a teammate from the Université de Montréal, Marouane Kessentini, we have started to explore how to go beyond simple metrics to represent code. Instead, we represent code as sets of predicates to describe complex code structures. In [KVS10], we showed that these structures contain more semantic information than metrics. In a study of both Xerxes and Gantt, we could more accurately identify all three types of anti-pattern ( $> 90\%$  precision). Of course, these techniques require more computation, but the results are encouraging and we believe that this is an interesting research avenue to pursue.

#### 4.4 Conclusion

In this chapter, we presented the problem of dealing with diverging opinions. We proposed a way to combine these opinions using Bayesian inference. In particular, we indicated how a corpus could be built to represent this subjectivity. We presented a methodology to build detection models of anti-patterns. Anti-pattern detection is a very subjective problem (only 30% of agreement), our Bayesian approach was shown to be an appropriate solution. With no local knowledge, our models outperformed DECOR, a state-of-the-art detection technique. When we included local data, we noticed significant improvements.

The reason why we are interested in identifying anti-patterns is because these are known design faults that impede maintainability of classes. However, there is the issue of using our models to guide maintenance activities. In the next chapter, we present how we can use our models to track the evolution of the quality of a system, in order to detect when classes are degrading or improving.

## CHAPTER 5

### ANALYSING THE EVOLUTION OF QUALITY

Until now, we presented quality models (and detection models) in an isolated context: they were executed on specific versions of a system. This is a limited, static view of quality management that is not representative of industrial practice. In such a setting, models are used by IV&V teams to regularly track the quality of the systems developed. Consequently, an IV&V team that performs these evaluations has quality information spanning multiple releases. This historical information could be exploited to guide maintenance efforts, but not without difficulty. Considering the changes in quality literally adds another dimension of complexity for quality engineers, and can easily overload them with too much information. Be that as it may, this additional complexity can be worthwhile if it can prevent future developments problems<sup>1</sup>.

We are interested in two types of events. Sometimes changes to a system can degrade its quality; other times, they can improve quality. Identifying potentially recurring changes that degrade quality could serve to establish coding guidelines for developers and lead to an early detection. Additionally, there might be cases when quality problems are solved in an elegant manner. These solutions might be re-usable and proposed to development teams faced with similar problems.

#### 5.1 Industrial Practice

In our partner's process, the models are executed every time a development team wants to release a new version of a system. This historical information is used by managers to get a general appreciation of the work of a development team. When quality scores drop below a specific threshold, then a QA engineer manually inspects the code. The use of thresholds can hide information in such a context: a high quality system could slowly degrade over a long period of time until a minor modification becomes the straw

---

<sup>1</sup>The content of this chapter was the subject of an article published at the 15th Working Conference on Reverse Engineering (2009) [VKMG09]

that breaks the camel's back. Only at that point would a QA engineer notice that the quality of code had been worsening without his knowledge.

To get a historical view of quality, past evaluations are typically presented using a time series chart presenting scores over time. This approach is relatively standard as many tools offer this view (*e.g.*, Sonar<sup>2</sup>). However, there is still the problem with deciding which parts of the software to visualize as it is impractical to simultaneously display quality affecting thousands of software entities. This difficulty is one of the reasons our partners' quality engineers seldom use visualisation tools. In fact, they ignore information from all but the previous quality evaluation. Consequently, there are certain quality trends that might be ignored.

## 5.2 State of the Research

Much research has been done on the improvement of software maintenance. An important period that brought attention to the problem in modifying legacy system preceded the year 2000 when many large, legacy systems needed to be modified to support four digit dates. This incident created a surge of interest in tools and techniques to support change.

Recent work focuses on the detection and analysis of change patterns: recurring ways in which software changes. The best known patterns are *refactorings*. A refactoring is a behaviour preserving change to improve structure [Fow99]. An example is "pushing methods" up to superclasses which takes methods from a set of classes and moves them to a common superclass. These are typically implemented in development tools to ensure that a change is complete, and that it does not alter behaviour. Some research has tried to identify refactoring in code bases, but most techniques have problems scaling [DCMJ06].

In a series of papers, Xing and Stroulia [XS04, XS05] proposed an approach to analyse the evolution of the logical design of systems and recover distinct evolutionary phases. This research tries to mine knowledge to improve the developer's knowledge

---

<sup>2</sup><http://www.sonarsource.org/>

of the systems they maintain. The result of this research only indirectly affects quality: this research finds common ways that the software product evolves. Other work focuses on identifying other types of patterns like co-changing files [ZWDZ04], and code clone evolution [HT99, BFG07]. In the past, we also investigated the presence of change patterns in software using clustering techniques. In the systems analysed, we found that developers structure their systems in ways to minimise future maintenance [VSV08].

Another direction of research provides help to understand system evolution through visualisation techniques. For example, Eick *et al.* [ESM<sup>+</sup>02] developed tools to visualise the evolution of software measures and change data, including size and effort. Langelier *et al.* [LSP08] map static metrics of a system to characteristics of boxes (size, twist and colour), and animate the evolution of these characteristics.

Finally, there is work trying to combine change information with quality measures. Change metrics are now included in many quality models (*e.g.*, [AB06, MPS08, NB05b]). In these models, changes are included as inputs alongside structure metrics. Moser *et al.* [MPS08] show that change metrics are better predictors than structure metrics for faults. We might suspect that this relationship is obvious as bugs are introduced by changes, but Arisholm *et al.* [AB06] studied another system and found structure to be a better predictor than changes.

There are two main research axes that deal with the quality of evolving software. The first is concerned with finding patterns in software to help developers understand how systems change. The second is concerned in quality evaluation by including change metrics with structure metrics as inputs in quality models. Our approach is an attempt at bridging these two categories by finding and exploiting patterns in the variations of quality by analysing its evolution in a time series.

Getting a clear grasp of the different trends present in a system can help identify maintenance problems and improve the health of the system. By analysing the parts of a system that have shown significant improvement, we could hope to find practices that developers employ to keep their systems fit. These positive trends could be provided by an IV&V team to a development team when concrete examples of quality improvements are needed.

### 5.3 Identifying and Tracking Quality

In this section, we present our approach for finding quality trends in evolving software. We start by discussing issues pertaining to the analysis of temporal data. Then, we present an approach to analyse the evolution of quality.

#### 5.3.1 Dealing with Temporal Data

When trying to study evolution, a decision must be made on how to represent the passing of time. Generally, we use either calendar time or the version numbers of software releases. An analysis using calendar time implies that a project is audited at regular intervals (*e.g.*, every month, or even nightly in a continuous integration process). Extracting information regularly can produce some misleading results because these dates might not correspond to an actual release. Many projects tend to be developed actively until they become relatively stable. At that point, the development slows down. This is the case with open-source projects like Xerces-J, which went from having releases every month to every year. If weeks go by without any changes to the software, a quality audit might be a waste of time.

Analysing the evolution of software according to release dates ignores specific timestamps and has the added advantage of consistently capturing a project in the same development phase. Furthermore, this approach can indicate what type of maintenance was performed (*e.g.*, bug-fix versions indicate that developers did not add functionality, and mostly removed bugs). This choice can have an adverse effect on models when releases become rare. Xerces-J is now maintained by a small number of developers who have limited knowledge of the inner workings of most of the project. Typically, the quality of software tends to degrade when key knowledge of a system is lost due to turnover [HBVW08]. As releases become infrequent, there should be even more attention paid to a system when it is changed. If the time between changes and releases is long, then the feedback from an IV&V team might arrive too late to be useful.

A final problem concerns the tracking of new classes. We might assume that “new” classes go through a period of debugging before becoming stable. But, not all new

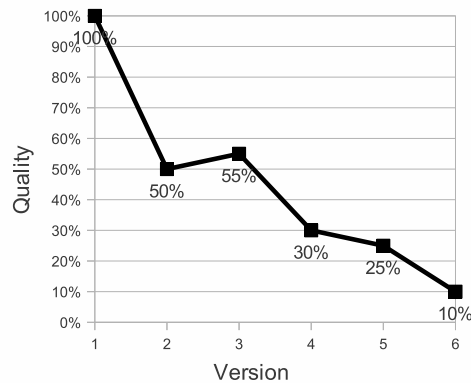


Figure 5.1: Quality signal: the quality of the module degrades over six versions

classes exist in the first version of the software; some are introduced much later. We would need a technique that is able to adapt to the notion of class-centered time as opposed to project-centered time.

We consider that the quality of every class is represented by a time series. These series can be of different lengths and have different rates of change. The technique used to identify trends should be able to support the fact that classes evolves somewhat independently from each other, and can appear and disappear.

### 5.3.2 Quality Trend Analysis

The evolution of a class with respect to its level of quality can be represented as a signal  $Q = (q_1, q_2, q_3, \dots, q_n)$  where  $q_i$  is the quality score of a class at the version  $i$ . Version 1 represents a version in which a class appears in the system and  $n$  is the total number of versions in which the class exists and is analysed. Such a signal is illustrated in Figure 5.1.

Our approach consists of two steps. First, we seek to find different trends given a set of signals representing how all classes have evolved. Then, we want to classify a specific signal  $Q$  according to these trends to guide a subsequent, in depth-study of evolution. Both these steps require a way to compare these signals given the aforementioned issues with temporal data. We chose to use dynamic time warping.

### 5.3.2.1 Dynamic Time Warping

Dynamic time-warping (DTW) is a well known signal analysis algorithm. DTW was first presented by Kruskal and Lieberman [KL83] to compute a distance measure between pairs of signals independent of timescale. This technique finds a “topological” distance between two signals by modifying the time axis of each one. For example, it can align two signals by matching peaks that occur at different times. It is commonly used in speech recognition software to handle different speaking speeds. This ability to modify time is important in the context of change pattern detection because changes tend to happen irregularly, often independently of the system versions.

### 5.3.3 Finding Quality Trends

Identifying quality trends can be done completely manually or semi-automatically using data mining techniques from a set of quality signals. We opted for a semi-automatic process. By using mining techniques, it is possible to process all known quality signals to find regular patterns, and present this information to an analyst for review.

To find candidate trends, we used a technique called *clustering*. Clustering is an unsupervised classification technique that takes a set of items and tries to group items together using a notion of similarity. To find possible trends, we applied agglomerative hierarchical clustering using the DTW distance metric as a measure of dissimilarity. This clustering algorithm starts by considering every item as a cluster and iteratively merges similar clusters together. The specific algorithm is described in [BGA06]. The result is a dendrogram representing all possible groupings as shown in Figure 5.2. A manual analysis of these groupings provides a set of possible trends. A quality engineer can select a set of clusters that makes sense.

## 5.4 Tracking Design Problems

We applied our approach to the problem of anti-pattern detection. We chose this problem because there is substantial literature that discusses why anti-patterns appear and how they should be removed. However, there is little hard, quantitative evidence that



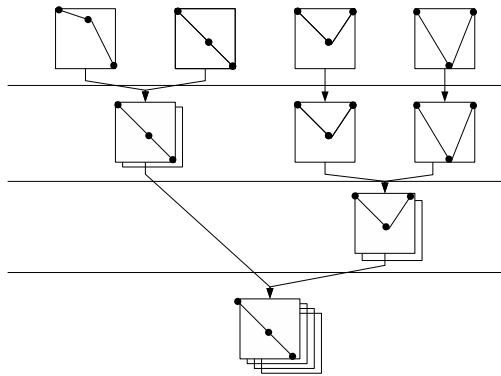


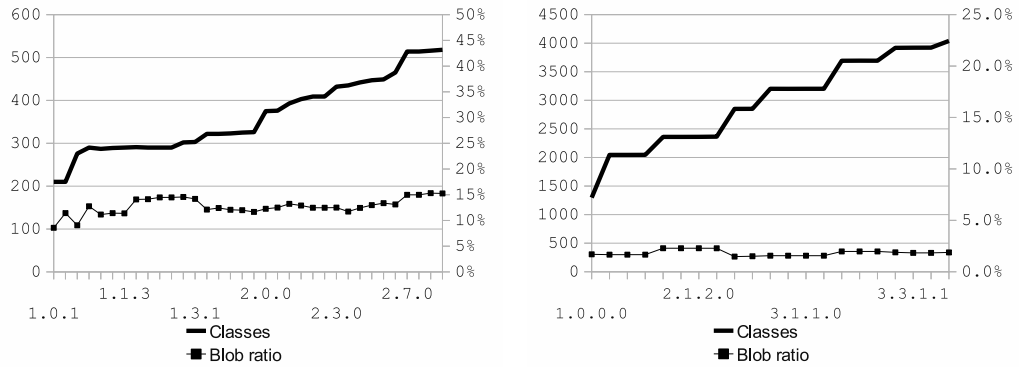
Figure 5.2: Dendrogram of quality signals. For the first step (on top), there are four different clusters, one per signal. Every step, the two nearest clusters are merged. Eventually (bottom), we find a cluster that contains all the signals.

describes how anti-patterns are introduced and removed from systems. In this section, we study the presence of Blobs in two systems: Xerces and EclipseJDT. Blobs are anti-patterns that correspond to a large class that centralises functionality.

For this study, we reused a variant of the Blob detection model from Chapter 4. This variant directly encoded the rules of DECOR and was presented in [KVGS09]. The quality signals are vectors containing the probabilities  $P(\text{Blob}|\text{Symptoms})$  for every version of a class. Our assumption is that classes that exhibit the highest probability of being Blobs at some point in their existence would be those tracked by quality engineers. We limit our study to these classes. Finally, we analysed released versions because this corresponds to how our partners perform their audits.

#### 5.4.1 Global View on the Evolution of Blobs

The first step in the study consisted of evaluating the number of Blobs that an IV&V would track for every version of a system. Figure 5.3 presents the ratio of Blobs (right axis) as well as the growth of the system (left axis, in number of classes) for every version of the systems. The figure shows that the growth of both systems is relatively linear. There are some different plateaus that correspond to minor versions during which few new classes/Blobs are added. The proportion of Blobs is relatively stable in Eclipse (2%) but increases significantly in Xerces (from 10 to 15%). This indicates that a sizeable part



(a) Number of Blobs in Xerces from 1.0.1 to 2.9.0 (b) Number of Blobs in Eclipse JDT from 1.0.0 to 3.4.0

Figure 5.3: Blob Ratios vs. Total Classes

of the code base would need to be reviewed.

Globally, Xerces and Eclipse JDT have respectively 138 and 144 classes that were considered to be Blobs at some point in their existence. The majority of these Blobs were Blobs right from the start: 70% for Xerces and 61% for Eclipse JDT. This indicates that these classes have a large number of responsibilities because of the original design, not due to maintenance work. Consequently, an IV&V team might want to focus its effort on inspecting large new classes. Furthermore, we noticed that a sizeable portion of these classes were eventually eliminated. They were deleted in the proportion of 30% in Xerces (41) and 19% in Eclipse JDT (29).

#### 5.4.2 Evolution Trends Identified

We found seven typical trends as shown in Figure 5.4. These seven trends are identified from *a* to *g*. The different trends are defined by two or three point configurations where every point can either have a low, medium, or high value. Low corresponds to the lowest observed value of the signal (probability). Likewise, high corresponds to the highest observed probability, and medium is  $(low + high)/2$ . Only three points are needed because the DTW can stretch the signal as much as needed.

The *Constant* stereotype corresponds to a stable signal where the class is always tagged as a Blob. *Gradual improvement* corresponds to a class that starts with a high

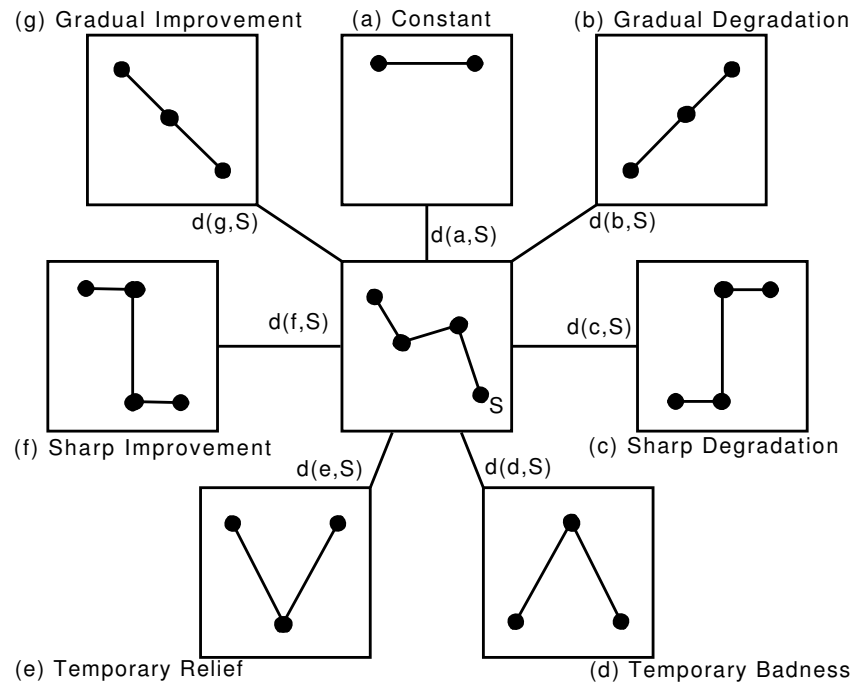


Figure 5.4: Evolution Trend Classification

probability of being a Blob, probability which drops to a medium level before becoming low. *Sharp improvement* is similar but the transition is abrupt: the signal level drops from high to low in a single version. *Gradual degradation* and *Sharp degradation* show the same phenomenon concerning degradation. Finally, *Temporary relief* and *Temporary badness* are stereotypes of classes that are only temporarily Blobs. To classify an evolution signal  $S$ , we compute the distance with the different stereotypes. The DTW algorithm finds  $d(x,S)$  as the minimal distance between  $x$  and  $S$ .

In the middle, we included an example of a subsequent classification: the quality signal<sup>3</sup>  $S$ .  $S$  is the signal to be classified, *i.e.* the probability that the class is a Blob, and  $d(x,S)$  corresponds to the distance calculated by the DTW algorithm between the signal  $S$  and a trend  $x$ . A signal is classified into the group that minimises the distance. The example was classified as a gradual improvement (trend g) as its probability of being a Blob decreased over time.

The distribution of different trends is presented in Figure 5.5. In Xerces, out of the

<sup>3</sup>the signal corresponds to `org.apache.xerces.impl.XMLVersionDetector`

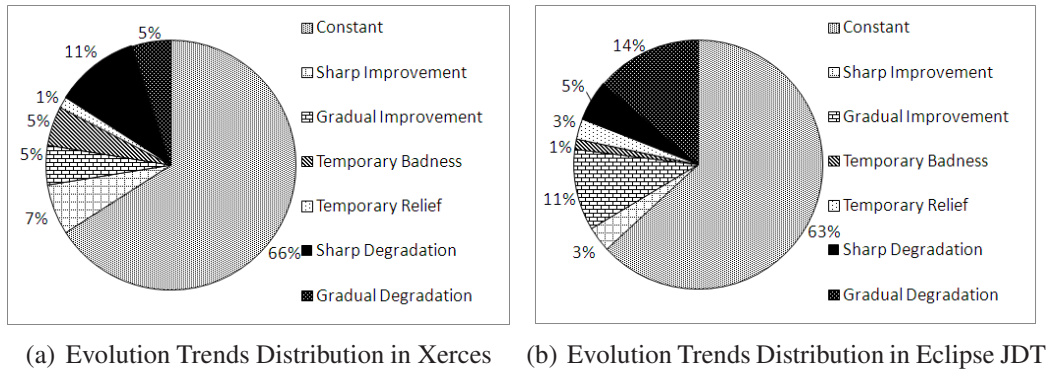


Figure 5.5: Evolution Trends Distribution of Blobs

138 classes that were Blobs at some point in their existence, 91 (66%) showed no significant variations and followed the constant stereotype. 16 Blobs (12%) were corrected by developers. This is significantly fewer than the number of Blobs deleted by developers (30%). We found 22 (16%) presenting different degradation symptoms.

Similarly, Eclipse JDT, a system containing 144 Blobs had a large number of stable Blobs, 96 (63%). In this system, almost as many Blobs were corrected, 21 (14%) as were deleted (19%). Finally, 27 (19%) classes saw their quality degrade. We consider that the classes corresponding to Temporary badness and relief are instances of both an improvement and a degradation.

This classification process highlights three main types of evolution trends of interest: improved, degraded, and constant Blobs. Analysing each group can provide key insights into the nature of these complex classes, why they still exist or become more complex, and how they are improved. We now analyse in greater detail these three evolution trends.

#### 5.4.2.1 Constant Trend

By far, this is largest group. It contains a high number of Blobs that are introduced at the very beginning in systems and that remain Blobs throughout the existence of the systems. We investigated the motivation of experienced developers to create “bad classes”. We contacted the primary developer of Xerces and presented to him a list of “confirmed”

Blobs. His answer was that these classes were as complex as the problems addressed; there was no simpler alternative.

Independently, we verified the use of design patterns [GHJV94] in these classes as it could indicate a clear intention by developers to write clean code; the structure of the classes is no accident. We found that 82% of the classes from this group were playing roles in at least one of the following design patterns: Abstract Factory, Adapter, Observer, and Prototype.

#### 5.4.2.2 Degradation Trend

The Blobs observed in this category have two different reasons for their degradation: either they gained new responsibilities (and grew in size), or they gained new data classes. In EclipseJDT, for the most part, these Blobs are very large classes from their introduction. The main reason why their observed quality degrades is due to the addition of data classes. This can be explained by the particular use of data classes in Eclipse: often, data classes are used to communicate data between different application layers. One typical case concerned a data class<sup>4</sup> that describes a certain range in an indexed text store. This data class is in fact a value object used to transmit information from one system layer to another. One large class<sup>5</sup> uses it for that purpose. Although there seems to be a justification for such design, any large class that interacts with a large number of these value objects, centralises behaviour, a symptom of Blobs.

For Xerces, 11 of 15 sharp degradations are due to a similar situation: the quality of a class degrades because it is already large and developers add new data classes. Gradual changes, however, were all incurred by additional code. Table 5.I summarises the changes occurring in the degraded classes. In this table, two different growth rates are presented: the average relative size increases in instructions and in methods. The number of versions indicates the duration of the gradual degradation. Sharp degradations show an average increase of 150% in instruction size and 86% in method size between a pair of versions. In the case of gradual degradations, the average change rate per version is

---

<sup>4</sup>`org.eclipse.jface.text.Region`

<sup>5</sup>`org.eclipse.jdt.internal.debug.ui.snippeteditor.JavaSnippetEditor`

65% for instructions and 41% for methods. Not presented in the table, the total changes in gradual degraded classes tend to be equivalent to that of the sharp degradations. Other metrics like cohesion were not significantly impacted.

Table 5.I: Degradation growth rates in Xerces

Degradation trend	Growth rate instructions/version	Growth rate methods/version	Nb of versions
Sharp	363.64%	162.50%	1
Sharp	138.44%	283.33%	1
Sharp	214.98%	113.64%	1
Sharp	513.33%	100.00%	1
Gradual	40.99%	16.67%	2
Gradual	142.90%	6.67%	3
Gradual	9.46%	15.38%	5
Gradual	65.04%	6.25%	2
Gradual	34.28%	155.56%	3
Gradual	63.46%	42.86%	2
Gradual	103.33%	45.00%	4

The results presented in Table 5.I seem to indicate that, when performing a modification on a class, developers should pay attention to the size of the changes made, as they may induce a degradation of the quality of the class. We only observed one instance of gradual degradation that would have escaped detection (9.5% average over 5 versions).

### 5.4.2.3 Improvement Trend

Two symptoms that can help detect Blobs are the size of a class and its use of data classes. Fowler [Fow99] suggested refactorings to correct both large classes and data classes. To correct large classes, these refactorings include *Extract Class*, *Extract Subclass*, and *Extract Interface*. When correcting a data class, the main concern is to limit access to its public attributes using the *Encapsulate Field* refactoring and then to add functionalities using the *Move Method* and *Extract Method* refactorings.

In our investigation of the improved Blobs, we analysed the different classes to identify if and what refactorings were applied. The results are presented in Table 5.II. The vast majority of refactorings found in Xerces were not refactorings “by the book”. In fact, four times, developers of Xerces extracted new super classes (indicated by \*), a refactoring not explicitly mentioned by Fowler as a solution to Blobs. Furthermore, in

three cases out of five, the extracted classes became new Blobs. We observed thus that the correction of Blobs may induce the creation of new Blobs.

Table 5.II: Refactorings identified in Xerces for the correction of Blobs

Improvement	Refactoring	Nb (%)
Sharp	Move Method to data class	5 (31%)
Gradual	Move Method from Blob	2 (13%)
Sharp	Extract Superclass* from Blob	4 (25%)
Sharp	Extract Class from Blob	1 (6%)

In Eclipse, the vast majority of corrections involved the addition of new methods to the data classes. These typically included validation methods to ensure that the data transmitted was coherent, and methods to provide multiple views of the data. The widespread use of data classes in this system exists for organisational and architectural reasons, and might not be indicative of defective design as per the work of Brown [BMB<sup>+</sup>98] who states that data classes are a sign that developer might not know how to program in an OO language. This is obviously not the case of EclipseJDT because it was designed by Erich Gamma, an expert of software patterns and author of one of the most important books on software design [GHJV94]. In large-scale, modern software system, the use of data classes (mostly JavaBeans) is relatively common to transmit information from a subsystem to another. To apply a detection model to Eclipse, it might be best to adapt the detection model to treat these classes appropriately as mentioned in Section 4.2.4.3.

## 5.5 Discussion

We believe that our study of both systems is representative of an actual industrial application of our technique. The systems analysed are successful systems: they have been used by thousands of developers and have been actively developed for almost ten years. Although, in the past, some design decisions might have been bad, the developers were able to keep the systems up to date with evolving specifications. Furthermore, these projects are key components in commercial offering of large companies, in par-

ticular IBM. We therefore believe that not only could our signal analysis approach be followed by an IV&V team, but also that the results of our study contribute to the state of knowledge in quality management.

### **5.5.1 How to Track Blobs?**

The most important aspect of our study in analysing the evolution of Blobs concerns their introduction. There is a common belief that Blobs are the result of slow degradation in the code base. However, this was not so. In our study, large complex classes tend to be introduced as such. It is thus important to inspect classes as soon as they are introduced in the system. Degradation is only responsible for 15-20% of Blobs. This degradation could be identified by including a change metrics in the detection model.

### **5.5.2 How to Remove Blobs?**

Our study identified two ways used by developers to “correct” Blobs: some were modified, but more often than not, they were removed from the system. These classes were removed when the program specification changed significantly (*e.g.*, when Xerces would implement new W3C standards). In general, when this occurred, we noticed that developers removed not only the Blobs, but the rest of the classes in the packages as well. Blobs tend to be heavily coupled with other classes; they can be difficult to change in an isolated manner. Sometimes, replacing classes can be easier than modifying them if these packages are well modularised. If a development team wants to avoid problems with Blobs, they can proceed by isolating the Blob from its associated (data) classes, then swap in a suitable replacement.

We noted that few “proper” refactorings were found. There are two plausible explanations for this. First, the systems were developed in the early 2000s. In those years, there were few refactoring tools that could automate these changes. Second, we might have had problems identifying these refactorings. In open-source systems, developers tend to follow agile practices, which dictate that refactoring should be a continuous process included with regular development activities. Correctly identifying these refac-



torings amongst the different changes is very difficult. Even though we did not observe this, refactoring activities can move functionality from a Blob to the data classes.

## **5.6 Conclusion**

In this chapter, we proposed a methodology to identify trends describing how quality evolves. We applied data mining technique using time-independent signal analysis techniques to find recurring types of evolution in two mature open-source systems. Using a Blob detection model, we found that these anti-patterns tend to not be the result of a slow degradation of the code base. They are instead added to the system as such. Furthermore, these classes are rarely corrected; instead, they are often replaced. We found that potential Blobs tend to play roles in design patterns. Finally, we discussed how the results of our study could be exploited by an IV&V team.

In future work, we want to evaluate if classes exhibiting the symptoms of a Blob from their outset have a higher probability of being Blobs than those that degraded. Also, we plan on exploring the possibility of evaluating when a team should consider replacement vs. correction, and the issues with both solutions. Finally, it would be interesting to extend this work to other systems, possibly systems that are considered failures. A major problem with an analysis of “bad” project is that these projects, if open-source, tend to die off before producing a sufficient number of versions to analyse.



## CHAPTER 6

### HIERARCHICAL MODELS FOR QUALITY AGGREGATION

Large software systems are designed and implemented as multi-level hierarchies of code entities. Thus, a system is composed of a number of sub-systems; each made up of packages containing classes, which in turn, contain methods made up of statements and variables. Unfortunately, most quality models do not handle hierarchies well. They were developed to judge quality at a specific level of granularity, for example computing class quality from class metrics. We call these *type 1* models. Most research has concentrated on the lower levels: quality models for classes or methods/procedures.

Although low-level models are useful for developers and testers, other stakeholders need quality measures for higher-level entities. For example, a manager might want a general view of the quality of the system illustrated in Figure 6.1. In practice, this is done by simple aggregation of lower level values, *e.g.*, using the average score of classes [BD02], or by counting the number of error-prone classes (a standard reporting practice in bug tracking systems). We call these *type 2* models. However, not all parts of a system are equally important: some parts corresponding to the system's kernel are always being used while others, like a function for an optional feature, might not. One would expect that more accurate system modelling could be achieved by weighing quality measures by a notion of *importance*.

This leads us to propose a general method to evaluate, at each level, the quality of an entity from that of their components. This method involves three parts: a) a component-level quality model, b) an aggregation model that weighs the relative importance of each component and produces an aggregate quality score, and c) a container-level quality model that uses this aggregate score as well as container-level metrics. We shall call models based on this approach, *type 3* models.

The usefulness of this approach was tested on the problem of identifying frequently changing classes. Here, the aggregate is the class and the components are its methods. We tested various approaches to determine *importance* and compared our importance-

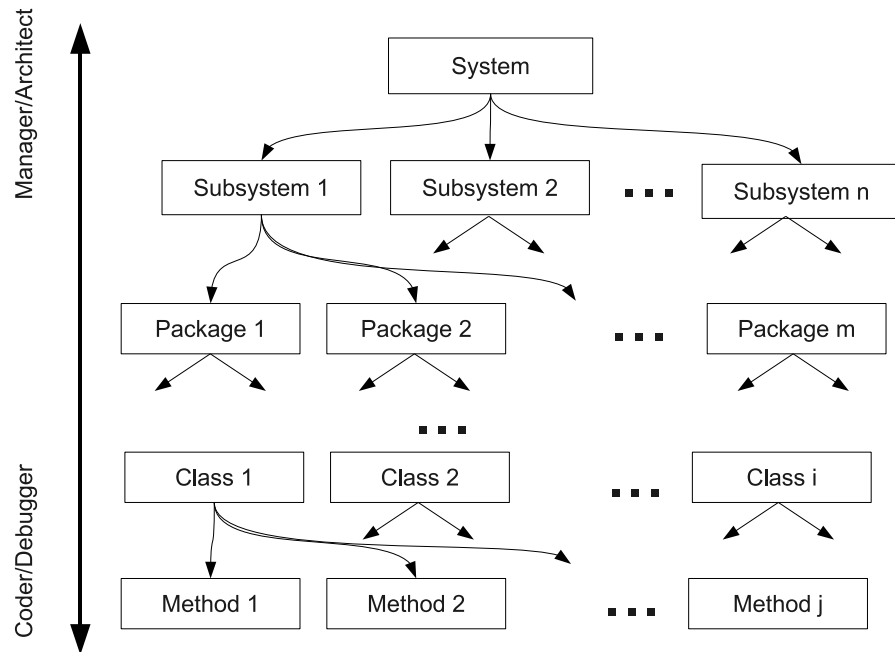


Figure 6.1: Logical decomposition of a system (right) vs. quality stakeholders (left)

weighing approach to traditional aggregation techniques (*e.g.*, sums and averages). We found that by using graph analysis techniques to determine importance, we could improve the identification of the highest-changed classes. This improvement is sufficient enough to justify the extra work of gathering quality data at multiple levels.

## 6.1 A Multi-level Composition Approach

To the best of our knowledge, there is no work that tries to explicitly combine quality information from different levels of a system in an intelligent way. In general, existing approaches like [BD02, KL05, KL07, ZNG<sup>+</sup>09] combine metrics from the method level to aid in a higher-level quality evaluation, using simplistic strategies where all components are considered of equal weight. Implicitly, this assumes that the quality of a container depends on the average quality of its components or on the sum of complexities of its components. Our general model explicitly models the quality of components, the importance of components, and the container's quality. Our approach is shown in

Figure 6.2. In this figure, we have three distinct models:

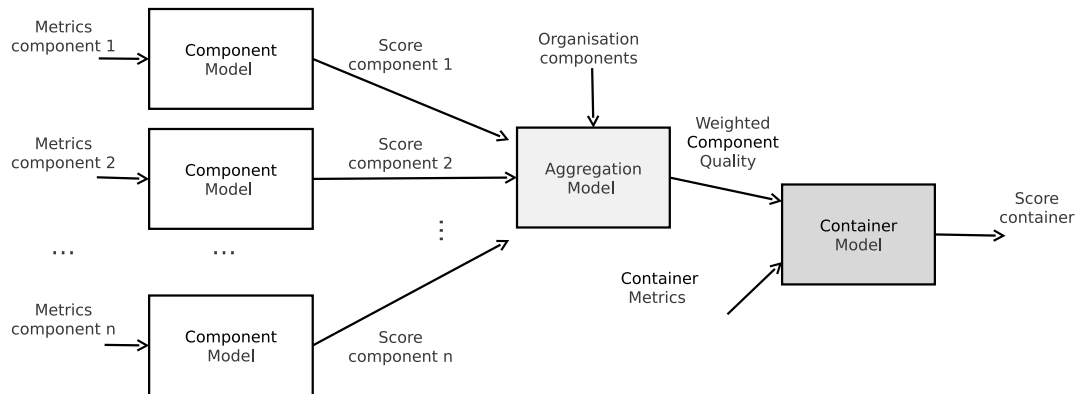


Figure 6.2: General composition model

- **Component model:** this model is a traditional quality model that uses metrics to assess the quality of a component.
- **Aggregation model:** this model evaluates the relative importance of the quality of a component on its container. It uses two types of inputs to produce an aggregate quality score: the individual quality scores of all components as judged by the component model and data describing how these components are inter-related.
- **Container model:** this model evaluates the quality of the container (*e.g.*, a class in a class-level mode). It integrates container specific attributes with the quality evaluation of its components.

To apply our approach, we propose three steps. First, we either build or reuse a component model. Ideally, it should be self-contained. Then, we define an aggregation model that quantifies the impact of each component on its container. Finally, the container model needs to be built and calibrated using information coming from the aggregation model.

An important hurdle faced when using a multi-level strategy is the scarcity of available data. It is already difficult and expensive to collect clean data at the code-level, where bugs are typically reported. It is even more difficult to find good package/system-level quality data. Consequently, we believe that we can build the models separately

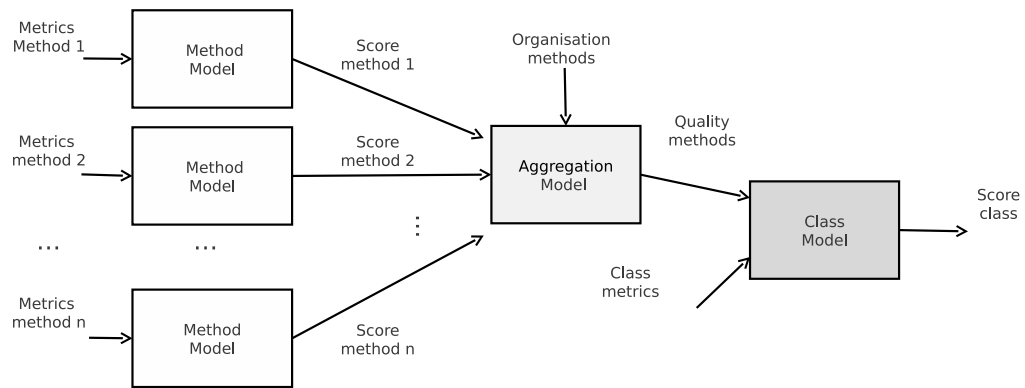


Figure 6.3: Class-method composition model

with the data at hand, and then back-fit the container model to make sure that the final models are as accurate as possible. In our study, presented later in this chapter, we will detail how this can be done in practice.

## 6.2 Modelling Code Changeability

We applied our general quality modelling approach to the problem of code changeability in order to test it. The majority of work on quality modelling focuses on identifying high-risk components. One way to identify problem components is to identify the ones that are in constant change because high-change components are moving targets for IV&V teams. These high-change components could in fact correspond to good practices like evolution patterns, but regardless of the reason for a change, any change requires development and testing effort and should be identified.

Our approach applied to class-level changeability is illustrated in Figure 6.3. Our component model is a method-level model that estimates the presence of changes in a method. Such a model can be built using the different techniques presented in Chapter 2. The scores of all methods in a class are aggregated using a composition model to produce a class-level metric. This aggregated metric is included as an input to a class-level model. In this section, we present three different types of change models. We then present our main contribution, the notion of importance.

## 6.2.1 Change Models

Three types of models can be applied to the identification of high-change classes:

1. **Type 1:** simple single level models. These are class-level models that use class attributes to predict class quality;
2. **Type 2:** models based that use method-level models, which consider all methods to be equally important;
3. **Type 3:** an extension of type 2 models that includes an aggregation model to assess the importance of individual methods.

A type 1 model corresponds to what is currently being produced and exploited by our partner. For an IV&V team to build models of type 2, it would need to collect method-level metrics. Finally, models of type 3 requires for the IV&V to define importance functions, something that is currently inexistent in the literature.

### 6.2.1.1 Classic Class-level Models (type 1)

The first model type uses only class-level information. Our baseline change model is based on the model shown in Figure 6.4 that Li and Henry [LH93b] used to predict class-level changes. In the figure, we can see that high-change classes are identified using the six metrics of the CK metrics suite<sup>1</sup>. We can see that no method-level information is explicitly used to evaluate classes.

In this model, we can notice that some metrics belong to another level. In fact, out of the six CK metrics, three (CBO, RFC, and WMC) can be calculated using a lower-level model. WMC is defined as the sum of method complexities, where method complexity is undefined. Obviously the complexity of a class could be computed using a method-level model that estimates the complexity using a set of method-level metrics. Coupling metrics like CBO and RFC try to assess the level of coupling of a class by considering the coupling of its methods (and attributes for CBO). A formal description of how these

---

<sup>1</sup>Li and Henri actually used 5 out of 6 CK metrics. The model included two versions of WMC; they replaced CBO with other coupling metrics.

can be expressed at different levels is presented in Appendix I. We believe that these metrics should be replaced by metrics provided by a method-level model.

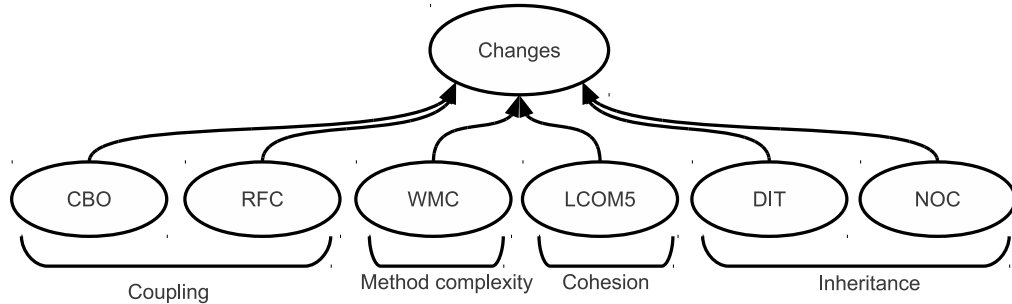


Figure 6.4: Reference class change model

As a result, we can identify and remove these metrics from our class-level model as illustrated in Figure 6.5, and replace them with a method-level model. We consider that this is a general problem affecting most quality models (*e.g.*, [ZPZ07, BD02]) that assess entities that can be broken down into sub-entities.

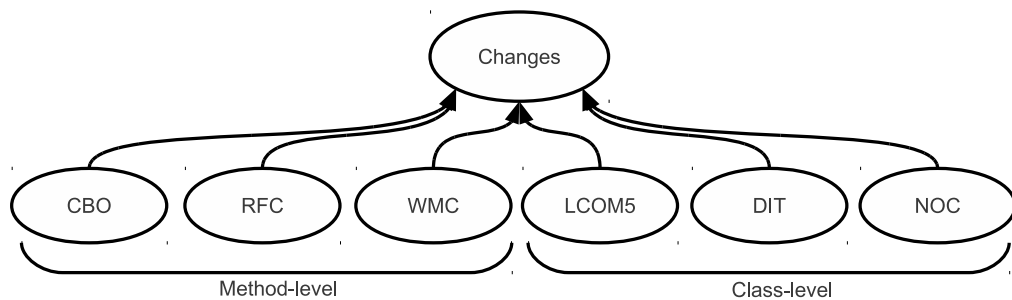


Figure 6.5: Reference class change model: the metrics on the left correspond to method characteristics; on the right, the metrics correspond to class characteristics.

### 6.2.1.2 Type 2: Adding Method-level Models

Research has shown that method-level metrics are useful; thus, researchers try to use these metrics by using simple aggregation techniques. For example, Koru *et al.* in [KL05, KL07] used the CK metrics as well as the maxima, sums, and averages of method-level metrics to predict class-level changes. Type 2 models differ from this



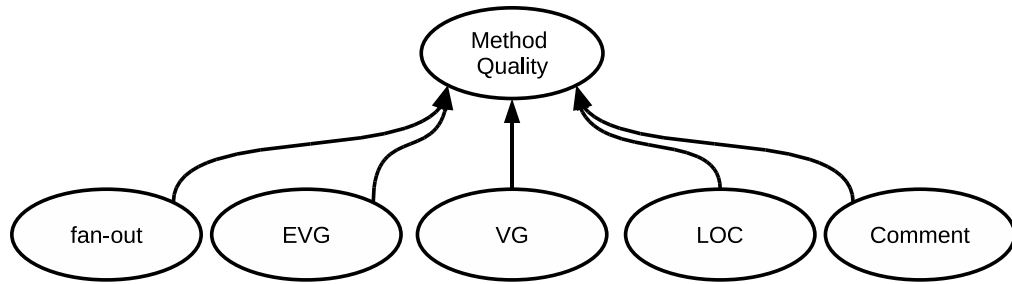


Figure 6.6: Sample method-level model

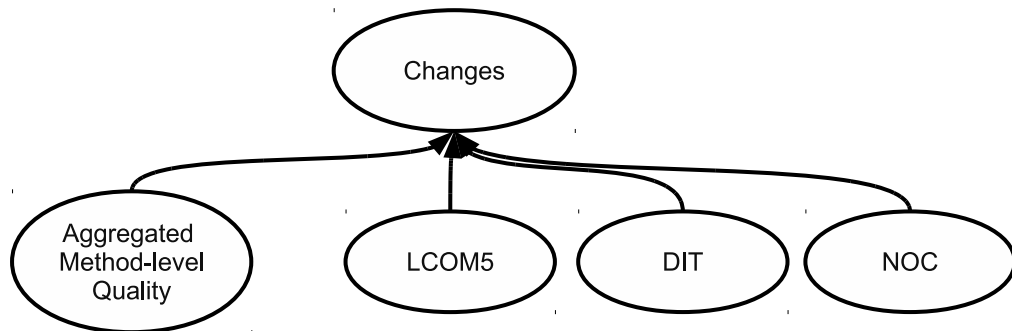


Figure 6.7: Modified type 1 change model to support method-level quality models

approach by including a method-level model to assess the quality of methods. Instead of considering the average method size, a model would consider the average quality score of a method.

To build a type 2 model, we need to have access to a tagged data set describing the quality of methods. Therefore, there are additional costs implied by such an approach: an IV&V team would need to collect and maintain a method-level corpus. As far as we know, only the NASA data set contains clean method-level data. The exact model would depend on the available data and can be built using standard model-building techniques such as those used in Section 3.2.3. For example, Figure 6.6 illustrates a model that evaluates the quality of a method using the NASA data set<sup>2</sup>.

In Figure 6.7, we present a type 2 model adapted from the Li and Henri model. We can see that three CK metrics, CBO, WMC, and RFC, were removed from the model and were replaced by an aggregate measure of method quality. Semantically, this aggregate

<sup>2</sup>Note that EVG, VG, *etc* are metrics explained in Section 3.2.2

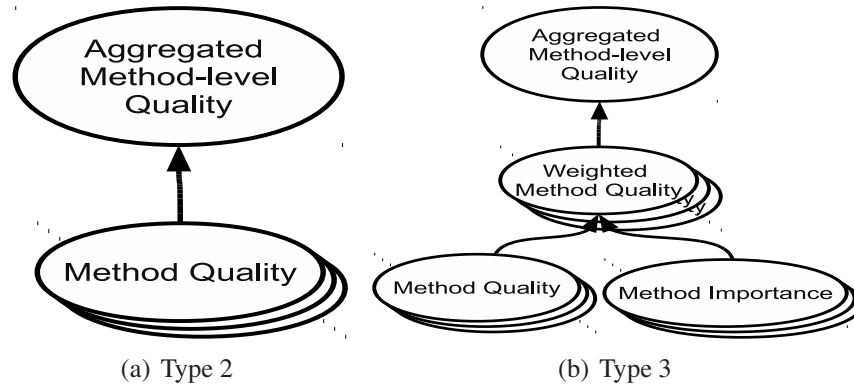


Figure 6.8: Aggregation models for method-level evaluations

measure represent the quality of contained methods: we judge whether or not the class is risky given that its methods are good. In a type 2 model, we consider that all methods are equally important. The aggregation strategy of individual method scores is consequently similar to what is done in [KL05, KL07, ZNG<sup>+</sup>09, BD02]: we can use sums, maxima, means, and medians.

### 6.2.1.3 Type 3: Adding an Importance Function

What distinguishes a type 2 model from a type 3 model is the addition of an importance function. Instead of treating every method as equal, it is weighed by a notion of importance. The difference is illustrated in Figure 6.8. To improve on a type 2 model, an IV&V team needs to understand how the importance of individual methods would affect the quality of a class.

## 6.2.2 Determining Method Importance

Many existing class-level models combine information from methods, but these do not assign importance to individual methods. Instead, they use standard aggregation techniques like a sum or an average, which consider that every method is equally important. We believe that it is possible and advantageous to assess the importance of methods. For example, a method that is executed frequently or is invoked from many different locations might be considered important.

### 6.2.2.1 Naive Aggregation Functions

In most quality models [KL05, KL07], method-level data is aggregated without a notion of importance. They use standard aggregation techniques that consider method-level metrics as a bag of values and generally describe the central tendency of these values or produce a measure a cumulative impact. These importance functions are easy to compute, but contain hidden assumptions that might or might not be acceptable depending on the context of our assessment. Here are the four most common techniques:

- Means and medians: we assume that all methods have an equal weight and that the average value of a metric of a set of methods is what would affect the quality of a class. These are used as measures of central tendency to reflect a “standard” use of any method within a class.
- Sums: we assume that quality, (or conversely risk) is cumulative. As more good methods are present, the goodness of a class would increase. Unless the metric is normalised, this strategy will favour classes containing many methods. Conversely, the same can be said of bad methods.
- Maxima (or minima): in general, software entities contain some extreme metric values. This strategy stresses the importance of these “worst/best” parts of a class. It will disregard any contribution of other methods.

When choosing a strategy over another, it is possible that the aggregation strategy might provide an artificial view on method-level quality. For example, the complexity metrics of methods follow a power-law distribution. An average quality measure might overemphasize the importance of an outlier even though we are interested in a measure of central tendency.

### 6.2.2.2 Importance Functions

Instead of using uninformed strategies, we believe that aggregation techniques should be chosen according to our quality goal. It is logical that the quality of a method that

is always executed might be more important than the quality of a method that is unreachable, or rarely invoked. Depending on how methods call one another, we could estimate the probability that it should be executed or inspected and use that as a measure of importance.

Ideally, we would use runtime data to estimate the probabilities. The problem with such an approach is that it can be difficult to find a sufficient number of executions to get a good approximation of “normal” executions. An alternate approach is to build a *call graph* through a static analysis of the code. A call-graph represents the structure of a program in terms of possible calling relationships. Call graphs are typically used by compilers to optimise code execution. For the compiler to avoid introducing errors, this graph should describe all possible execution paths through a program. In our case, we will analyse call graphs to see which methods are more likely to be executed.

Figure 6.9 presents a simple call graph of four classes. The arrows indicate calls between methods. If the method *main* corresponds to the entry point of the application, then any method that is not reachable from this method should never be executed. This is the case for method *m2* as there is no path connecting *main* to *m2*. This method would not likely affect the quality of a class. However, if we are analysing a code library, we need to make assumptions on which methods are possible entry points. For example, we could use all public methods. In our example, we assume that *m2* is also an entry point, as it would otherwise not be included in the call graph.

Metrics like fan-in and fan-out are based on this type of representation of code and have been used in quality research (for example, [OA96, KM90]). These measure respectively the number of incoming (calling methods) and outgoing edges (called methods) in a graph. These metrics only measure direct coupling; consequently, the importance measured is only local. These graphs however show characteristics that allow for a more sophisticated analysis.

### 6.2.2.3 Characteristics of Call Graphs

The general topology of call graphs has been studied and they exhibit several characteristics similar to other types of graphs observed in biology, social networks, etc [Mye03].

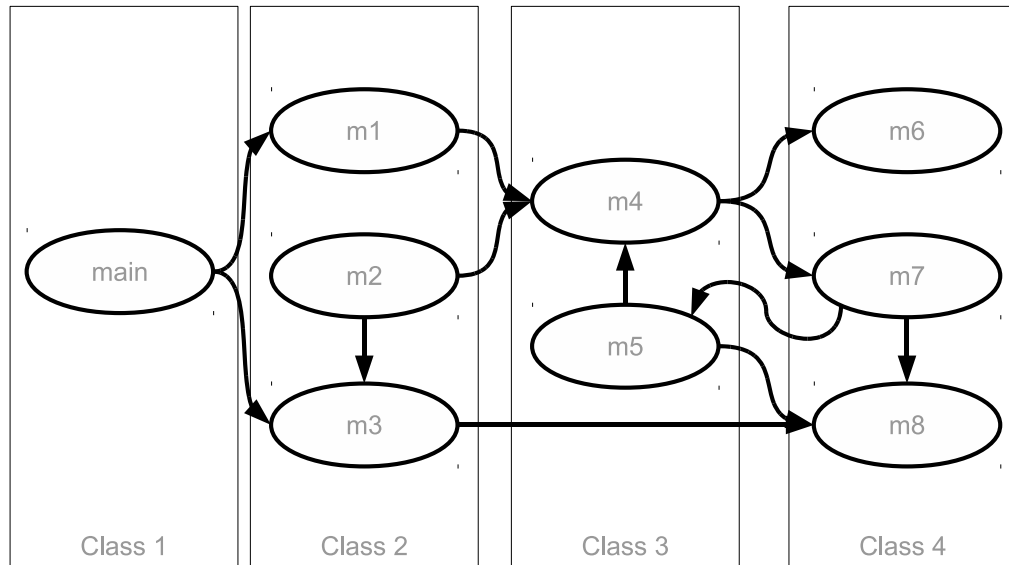


Figure 6.9: Simple call graph between methods in their classes

First of all, they are *scale-free*. Scale-free networks are networks that have a scale-free degree distribution, meaning that there is no “average” number of incoming/outgoing edges. Instead, the probability of a node having a degree of  $k$  follows a power-law distribution. Most nodes in such a graph are coupled to very few others (*e.g.*,  $< 5$ ), but there are nodes that have huge number of couplings. Second, there is a small world phenomenon that indicates that there are islands of tightly coupled methods [JKYR06], with a few key methods interconnecting these groups. To identify important methods, there are two different types of measures: those that are based on the *graph topology*, which identify key “linking” methods, and those that focus on *graph exploration*, seen as a stochastic process, which identify potentially high-execution methods. A systematic survey and comparison of graph-based importance measures was reported in [WS03].

The first important study of scale-free graphs came from Freeman [Fre79]. In a study of graph topology, he stated that the simplest notion of importance in a network is the notion of *degree*. The degree of a node is the number of nodes it is directly connected to. As this notion of importance is local, he mentioned two other notions: *closeness* and *betweenness*. Betweenness is based on the frequency of a node appearing on the shortest-paths separating pairs of points. *Closeness* is the inverse distance separating a node to

all others in the graph. The notion of *closeness* is the basis of the notion of *degrees of separation* in human networks, defined as the minimal distance connecting a person to any other in a network.

The second category of measures represents the activities that are done on a graph. The advent of the web contributed to this type of analysis. On the web, users navigate a large network where every page is a node. Measures like the PageRank [BP98] model the behaviour of users clicking different sites as a Markov chain. The measure estimates the probability that a user would be on a page. These measures could be used on a call graph where the resulting probability would be the probability that a method is being executed. In the field of software engineering, these types of measures have been explored in the context of identifying fault-prone binaries using dependency graphs between programs [ZN08].

#### 6.2.2.4 Call Graph Analysis Algorithms

The specific importance function used should depend on the activity we consider important. Our method-level change model decides whether or not a method will change frequently. The importance function should in turn indicate if changing this method will cause many class-level changes. We believe that we should look at central methods because a method in the middle of an execution trace could likely impact both the expected result of all calling methods as well as all called methods. This strategy would downplay both entry points and utility methods. In a call graph, the starting point represents the entry point of an application, typically the `main` function. Methods that do not call any others are often utility methods like `equals()` or `toString()`. We consider that four different measures could be used to determine method importance:

- The degree of a method [Fre79]. This is the simplest measure to calculate: it is the sum of calling and called methods. It should measure the impact of having a high-change method on its immediate neighbourhood.
- The betweenness of a method [Fre79]. The betweenness of a method  $m$  is calculated by measuring the proportion of shortest paths connecting every pair of methods

which contain  $m$ .

- The probability of execution as measured by PageRank [PBMW99]. This algorithm applied to a call graph approximates a random execution of a program. Methods that are executed are useful and therefore will likely change as a user's needs will evolve.
- The centrality of a method can be measured by the Markov centrality algorithm [WS03]. This algorithm calculates the inverse of the mean number of steps required to reach a method  $m$  from any other (in a Markov chain). This indicates how likely a method will be called by others in the system.

We normalised the value of these metrics so that the sum of importance of all methods in a system is one (1.0). A detailed description and comparison of the different algorithms applied to social networks is presented in [WS03].

The example shown in Figure 6.10 will illustrate the various measures of importance. We can assume that methods *main* and *m2* are starting points in the call graph, and that conversely, *m6* and *m8* are sinks. We would like for an importance function to identify central methods like *m4*. In Table 6.I, we show the results of these different weighing functions on the example code. Using the degree of a method accurately identifies *m4* as an important method (absolute degree of 5); it also gives significant weight to *m8*, a sink. Betweenness and centrality on the other hand, assign little, if no importance, to starting or sink methods. Finally, PageRank identifies *m8* as the most important method because most simulated execution paths will eventually reach the method; *m4* is a close second.

In our importance models (type 3), the aggregation model calculates a simple weighted importance of its methods:

$$method\_input = \sum importance(m) \times quality(m)$$

This input is bounded between 0 and 1.

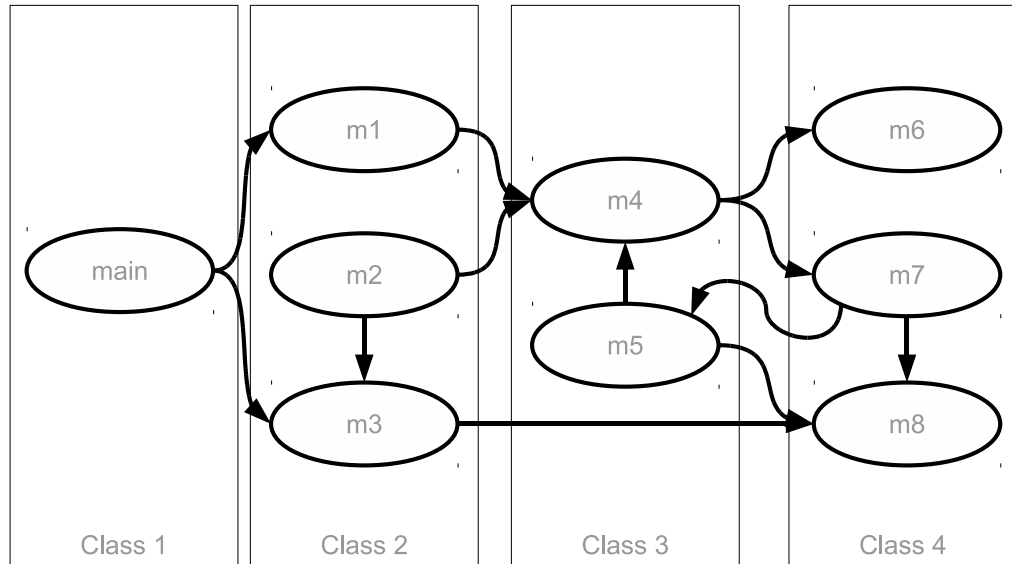


Figure 6.10: Simple call graph between methods in their classes

Table 6.I: Relative weights for each importance function. The most important method is identified in bold.

Method	Degree	Betweenness	Centrality	PageRank
main	0.08	0	0.01	0.04
m1	0.08	0.15	0.03	0.06
m2	0.08	0	0.02	0.04
m3	0.13	0.07	0.03	0.08
m4	<b>0.21</b>	<b>0.48</b>	<b>0.61</b>	0.17
m5	0.13	0.07	0.05	0.10
m6	0.04	0	0.08	0.12
m7	0.13	0.22	0.08	0.13
m8	0.13	0	0.09	<b>0.24</b>



## 6.3 Comparing Aggregation Strategies

In the previous section, we presented three types of class-level change models. The multi-level, sophisticated models could be useful, but they require significant investments to build and maintain. In this section, we present a study to show if these models are worthwhile. We build instance of all three models and tested them on the problem of identifying high-change classes.

### 6.3.1 Study Definition and Design

Our goal is to study the ability of the three different change models to predict the changes in classes from the perspective of a software maintainer trying to assess maintenance risks. Our specific research questions are as follows:

1. **RQ1:** How good are models of type 1 at identifying high-change classes?
2. **RQ2:** Are models that aggregate information from a method-level model (type 2) better than type 1 models?
3. **RQ3:** Are models that include importance functions (type 3) better than those without (types 1 and 2)?

For RQ1, we evaluated the capacity of the Li and Henry model (from Section 6.2.1.1), to identify high-change classes. Secondly, we tested simpler models using single CK metrics as input. For RQ2, we examined the performance of method-level metrics combined using the standard aggregation techniques described in Section 6.2.2.1 (*e.g.*, maximum and mean). Finally, for RQ3, we investigated the performance of models using the various importance functions described in Section 6.2.2.2.

### 6.3.2 Data Analysed

To build our different models, we needed tagged data. We could not find data on a single system describing both class and method-level changes. We consequently reused two separated data sets. The first data set describes classes in open-source systems.

Each class is characterised by OO metrics and is associated to a number of changes. The second data set is the NASA data set used in Chapter 3, which provides metrics and change data at the method level.

### 6.3.2.1 Class Change Data

Our main data set is the replication data used by a teammate, which is described in [KGA09]. This data was used to build our class-level quality models. It describes classes from six different open-source Java systems using metrics<sup>3</sup>. Also, classes are tagged with the number of changes between the selected version and January 1<sup>st</sup> 2009. Changes were identified by mining the version control system logs; all changes were considered. Since we had full access to the source code of these systems, we could add missing metrics to build our models.

The names, the versions, the size and the type of the systems analysed are presented in Table 6.II. We used ArgoUML as our test system and used the other systems to build and train the class-level model.

Table 6.II: Descriptive statistics for replication data

System	version	# classes	type
ArgoUML	0.18.1	1237	application
Azureus	2.1.0.0	1232	application
JDT Core	2.1.2	669	library
JHotdraw	5.4b2	413	library
Xalan	2.7.0	734	library
Xerces	1.4.4	306	library

A few CK metrics used to describe these classes are implemented in a particular manner that deserves additional details. WMC (or sum of method complexities) sums the number of method invocations/field assignments in each method. It is consequently very highly correlated to a notion of size. CBO counts the number of declared types (including interfaces) coupled to a class. It does not consider constructor calls for coupling.

<sup>3</sup>The metric were extracted using the PTIDEJ tool suite and the metric definitions can be found at <http://www.ptidej.net>.

We used the most recent version of the LCOM metric (LCOM5). Finally, as the corpus did not contain RFC, we calculated RFC independently using MASU<sup>4</sup>.

### 6.3.2.2 Method Change Data

To build the method-level model, we reused the three systems from the NASA data set described in Section 3.2.2. For these systems, we did not have access to the source code and consequently dealt with this data as a black box. The data set does not explicitly mention changes, so we considered that any method that had an issue requiring a correction was changed. Additionally, many metrics were very highly correlated with one another, so we included computed relative versions of the complexity metrics (*e.g.*,  $vg / LOC$ ).

As our test system is ArgoUML, we needed to be able to apply method-level models on this system. We therefore reimplemented the majority of procedural metrics (*e.g.*, LOC, fan-out,  $vg$ ) used in the NASA data set for Java systems.

Before going further, we would like to note that the construction of both the class and method-level models required external data. There is therefore no guarantee that the models built will properly fit the particularities of our test system. This was shown to have an effect both in Chapters 3 and 4. This can have a severe, negative influence on our results, but we leveraged data that was available.

### 6.3.3 Variables Studied

A class in this study is represented as a vector of variables:

$$class = (iv_1, iv_2, \dots, iv_n, dv)$$

where  $iv_i$  is an independent variable and  $dv$  is the dependent variable.

**Dependent variable** - The dependent variable used for all research questions is whether or not the class is changed significantly more than the others. We used a box-plot as used previously in Section 4.2.3.1 to determine if a class is high-change (HC) or

---

<sup>4</sup><http://masu.sourceforge.net>

not (NHC). On ArgoUML, any class changed over 18 times was a high-change class.

**Independent variables** - For RQ1, we considered only the use of the CK metric suite. These metrics were analysed separately and combined in the reference quality model. For RQ2 and RQ3, we used our general class-level change model. We consequently had four variables, the three class-level metrics as well as the aggregated scores produced by the method aggregation model. In RQ2, we applied standard aggregation techniques whereas in RQ3, we applied our importance functions.

### 6.3.4 Operationalising the Change Models

We built models corresponding to the three types of change models using Bayesian models following the approach presented in Section 4. All metrics were discretised using a box-plot. The quality models evaluate probability that a class/method has many changes given a set of metrics ( $P(HC|iv)$ ).

#### 6.3.4.1 Class and Method Changeability Models

The method model was trained on NASA procedure/method data used in Section 3.2. Training on this data, we found that the best model used only two inputs: a metric that was a standard size metric and the other was a relative complexity metric (complexity divided by size). We selected only one unnormalised metric because all the metrics are highly correlated with one another. The combination of metrics that produced the best results was the size of a method (in LOCs), and its relative essential complexity (Figure 6.11).

To evaluate the quality of classes, we started by building models using the CK metrics as those built by Li and Henri, but we found that two class-level metrics, RFC and LCOM5 had adverse effect on the predictive capacity of the models. The inclusion of LCOM5 would have added noise to the model, and we found RFC to be too highly correlated to WMC (as observed independently in other studies as [BD04, WH98]) for both to be considered in the same model.

To construct our models of types 2 and 3, we reused the type 1 model. The idea is

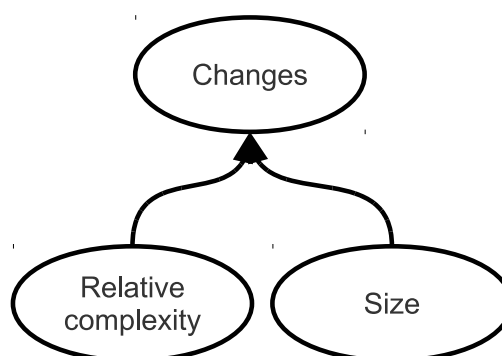


Figure 6.11: Method change model

that we build a model that considers global method quality (*e.g.*, low or high quality) using CBO and WMC. Then, we replace the way it assesses method quality by with a method model. To do this, we isolated the contributions of CBO and WMC using an independent probabilistic variable  $P(\text{method\_quality} = \text{high} | \text{WMC}, \text{CBO})$ , and learned the probabilities that  $P(\text{Classchange} = \text{high} | \text{method\_quality}, \text{LCOM}, \text{DIT}, \text{NOC})$ . When testing RQ2 and RQ3, we directly used a probability distribution of method-level change ( $P(\text{method\_quality} = H)$ ) as interpolated from our aggregate method quality.

### 6.3.4.2 Aggregation Model and Call Graph Creation

For RQ3, we tested the four importance function presented in Section 6.2.2.4 on statically constructed call graphs. In OO systems where runtime object types can determine exact method calls, building a precise call graph is an undecidable problem. Consequently the algorithms that are commonly used try to find approximations. For this study, we used a points-to analysis (0-CFA) as implemented within SOOT<sup>5</sup>. This algorithm requires for the whole program to be provided for analysis (including all library dependencies), and uses an entry point (the main method) to know the types of objects in the system. In a previous study, we analysed the use of static call graphs to measure coupling [AVDS10], and found that dynamic class-loading is a valid concern. We therefore used SOOT’s support for basic dynamic class-loading (`Class.forName()`)

<sup>5</sup>SOOT implements multiple type analysis algorithms (*e.g.*, Class hierarchy analysis (CHA) and Rapid type analysis (RTA)) that are used to build call graphs. These algorithms analyse which methods redefined in subclasses can be invoked. SOOT is available at <http://www.sable.mcgill.ca/soot/>

and `Class.newInstance()`.

### 6.3.4.3 Analyses Used

Our models are Bayesian models that produce a probability that a class is HC given the set of dependent variables ( $HC|iv$ ). As in Chapter 4, we consider that a quality engineer would inspect these classes according to its evaluated risk. We propose two ways to evaluate the performance.

The first seeks to verify if the ranked results are good; we do this using the rank correlation (Spearman) between the evaluated probability and the number of actual changes. The rank correlation indicates whether or not the inspection would be done in the best order possible, *i.e.*, in order of decreasing # of changes. This correlation penalises every incorrectly positioned class equally. This way to measure the quality of rankings does not necessarily correspond to the opinion of quality engineers as their interest lies in the first results. We therefore also measured Pearson's correlation, which considers the actual number of changes even though it is typically not used on non-normally distributed data.

The second way to evaluate our models is to evaluate the efficiency of our approach given an inspection task. Specifically, we are interested in evaluating how many changes would be accounted for given the inspection of  $n$  classes. For this, we define the efficiency of an inspection order of  $n$  elements as the ratio of number of changes found for the first  $n$  classes ranked according to our model compared to the changes found in  $n$  classes ranked optimally. Formally,

$$efficiency(n) = \frac{NumberChangesFound(n)}{NumberOptimalChanges(n)}$$

Since we have an oracle providing the exact number of future changes for every class ( $NumberOptimalChanges$ ), we can determine what would have been the optimal set of  $n$  classes and their corresponding changes. Both analyses are used for our univariate analyses and model-based analyses.

### 6.3.5 Results

In this section, we present the results for our three different research questions. RQ1 was concerned with the use of OO metrics. RQ2 considered the contribution of method-level metrics within a method-level quality model. The result of every method-level evaluation is combined using standard aggregation strategies. Finally, for RQ3 we explored different sophisticated aggregation strategies.

#### 6.3.5.1 RQ1: predicting class changes from class-level data

The correlations between the different OO metrics (including the scores produced by our model) and the number of changes in each class are presented in Table 6.III. The first thing we can observe is that the Li and Henry model is relatively weak, while WMC is the best predictor of changes. This implies that the training data did not adequately characterise our test system (ArgoUML).

**Rank results** The rank correlations indicate whether or not the order of the results is similar to the rankings of an optimal inspection. The best strategy in this context is to use the number of statements (as measured by this implementation of WMC). This measure of size is fine-grained and can discern very small differences in size. However, considering only the differences in rank can be misleading. For example, there are over 400 more statements in the largest class than in second largest class (6133 vs. 5720). On the other hand, the three classes at the 100<sup>th</sup> rank all have the same number of statements (377). This capacity to differentiate between small and very small classes helps to order classes with few changes, but it is not very important for a quality engineer whose objective is to locate dangerous classes. The correlations of CBO and our Li & Henri model are both moderate (correlations between 30-50%). Cohesion, and inheritance metrics are all weakly, but significantly correlated to the number of changes.

The Pearson correlation is included to indicate the strength of the relationship observed. This type of correlation tends to overemphasize extreme values, but since a code inspection would try to find the classes with an extremely large number of changes, this

Table 6.III: Correlations: number of changes and class metrics/model output

Metric	Correlation (rank)	Correlation (Pearson)
Li&Henry (baseline)	31%	25%
WMC	51%	30%
CBO	33%	25%
LCOM	23%	20%
NOC	19%	14%
DIT	16%	10%

measure indicates if these outliers are near the top of an inspection list. Unlike rank correlation, we only have metrics that are weakly to moderately correlated to changes (20-30%).

The general tendency observed is that there is a bias towards size metrics (*i.e.*, WMC and CBO). This indicates that the extremely large classes tend to change more frequently than small classes. This follows from the hypothesis that every line of code is subject to change; therefore, the bigger the code, the bigger the chance of change. The Li and Henry model is built on systems where the notion of size is considered; therefore, its results (in Pearson correlation) are comparable.

**Efficiency results** In Figure 6.12, we plotted the efficiency of an inspection process based on the same models. For inspections of fewer than 20 classes, all models are inefficient: around 15% of changes are found. This is a serious problem for an IV&V team as they would generally reject a model that wastes their time. In fact, the “best” metric, WMC, performs poorly and every inspection would be costly as it returns very large classes as false positives. As expected, on the long run, WMC performs well. An unexpected result is that, NOC is the best ranker for the top 20 classes. The baseline model gives average performance with 50% efficiencies at > 30 classes. It is better than most individual metrics, but surprisingly, WMC is still the best. We can assume that the model is too specific: the data we used to build and train the model is not a good fit for ArgoUML.



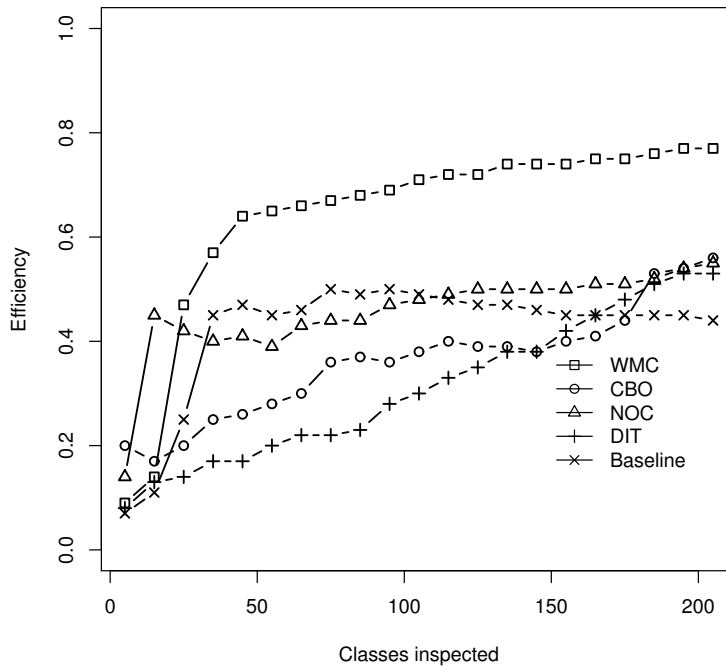


Figure 6.12: Inspection efficiency for different sized inspections using class-level information

### 6.3.5.2 RQ2: including a Method-level Model

Table 6.IV: Correlations: number of changes and scores (type 2 models)

Metric	Correlation (rank)	Correlation (Pearson)
WMC	51%	30%
Max	32%	34%
Sum	30%	31%
Mean	22%	17%
Median	-8%	2%

In RQ2, we studied the effect of using a method-level model whose results were combined using naive aggregation techniques. In Table 6.IV and Figure 6.13, we present the results for aggregation models using the maximum, sum, average and median values of method-level models, as well as WMC for comparison purposes. We can observe that both the maximum and the sum of method-level quality estimations show similar Pear-

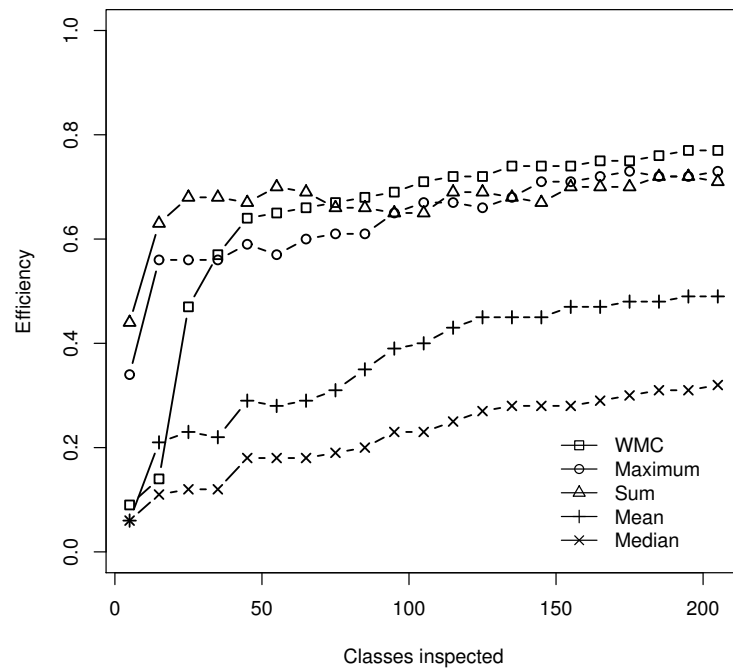


Figure 6.13: Inspection efficiency for different standard aggregation strategies

son correlations to WMC. They are, however, significantly better strategies at identify the top high-change classes: for the first 75 classes, they are more efficient than WMC. Afterwards, the max, sum, and WMC models are roughly equivalent. On the other hand, using measures of central tendency does not produce interesting results. This is additional evidence that our partner's extensive use of average is not a good idea.

We believe that the sum/max strategies are better predictors in an industrial context than models of type 1 as few IV&V teams would adopt a process that cannot identify the worst classes in a system. This finding supports our intuition that using a composition model can improve a quality evaluation process.

### 6.3.5.3 RQ3: using Importance Functions

For our final research question, we investigated the usefulness of our intelligent aggregation strategies. By using alternative ways to measure the importance of methods,

we found that it is possible to improve the performance of our baseline, Li & Henry model. First, we can see in Table 6.V that by weighing methods by their centrality in a call-graph, we can identify better combination strategies. In fact, the Pearson correlation between the Centrality and Page Rank output and the number of future changes is relatively strong (near 50%). Using a local notion of centrality, measured by the degree metric, does not produce interesting results even though it is superior to type 1 models.

Table 6.V: Correlations: number of changes and scores (type 3 models)

Metric	Correlation (rank)	Correlation (Pearson)
Page Rank	33%	47%
Centrality	31%	49%
Betweenness	26%	26%
Degree	25%	21%

In Figure 6.14, we present the inspection efficiency for our different strategies as well as WMC as it was generally the best metric for inspection. The best correlated strategies, Page Rank and Centrality, lead to the most efficient inspections as they identify highly-changing classes: efficiency quickly stabilises at 70%. Furthermore, even for small inspections, the PageRank is very efficient, meaning that it is able to identify the most-changed classes in the system. After the first hundred classes (towards 10% of the classes in the system), WMC becomes the best ranker.

The capacity of our composition-based models to predict change is actually quite surprising as they were not specifically trained on ArgoUML, and we substituted the training metrics for our more complex combination strategies. We believe that the call graphs are able to eliminate noise, thus enhancing our results. Furthermore, the concepts they measure seem to be system-independent as the results are superior to other configurations.

Our observations are consequently that the more a method is central to an execution path, the more likely it will be modified. This makes sense because 1) users would rarely ask for a change to a part of a system that is never executed, and 2) failures are more likely to be observed when code is executed, the changes measured could be corrections.

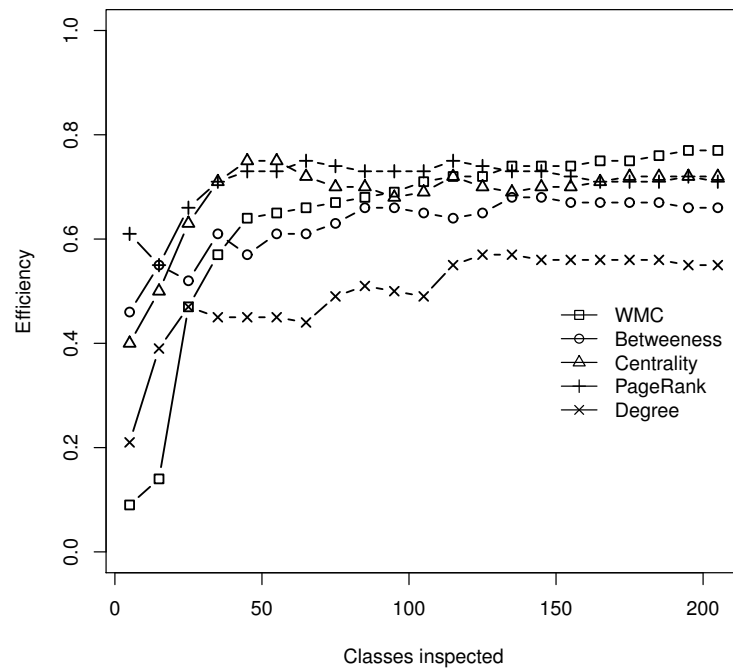


Figure 6.14: Inspection efficiency for different sized inspections by combining method-level information

### 6.3.6 Discussion

In this section, we discuss different issues with the proposed approach in the context of this study, as well as the results of the study itself.

#### 6.3.6.1 Cost-effective Inspections

We saw that size was a very important factor in a change identification process. For an IV&V that focuses on testing software, focussing on large classes first can be a reasonable idea. However, there are other quality assurance activities like Fagan code inspections that require developers to read and understand the risky code identified. In our evaluation of efficiency, we considered that every class was equal, regardless of its size, and this showed WMC rivalling our best type 3 models. In industry, this is not the case [AB06]: an IV&V team member can inspect 8-20 lines/minute [Men08]. When

considering the effort required for checking, size becomes a very important factor. In this section, we compare the inspection strategies using an alternative measure of efficiency. We consider the average number of changes per line of code found for an inspection set (change discovery rate). For the whole system, the change rate is 3.9%, or 39 changes per KLOC.

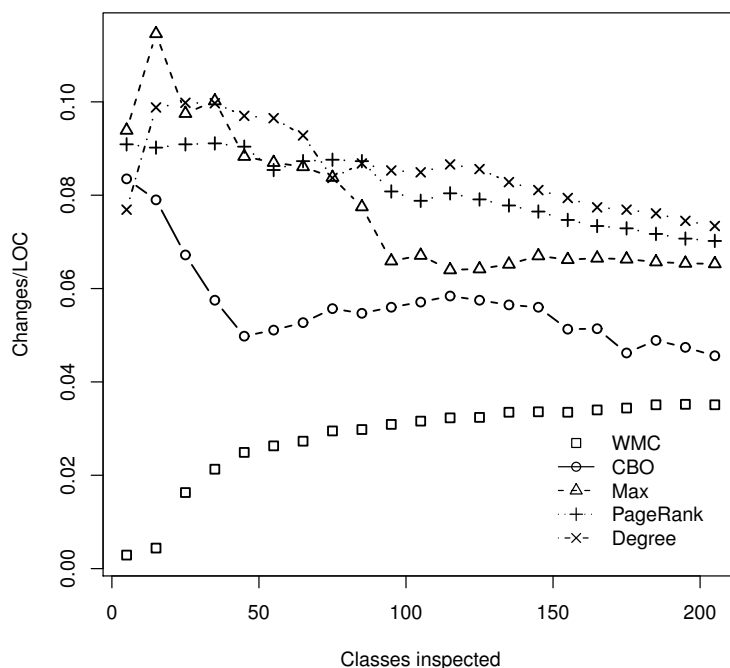


Figure 6.15: Inspection efficiency (in terms of size)

In Figure 6.15, we present the change discovery rates for five of the previous experimental configurations. We can see that an inspection strategy based on size as measured by WMC performs very poorly: the change discovery density is lower than 3%. This indicates that the larger the class, the less relative changes it will likely have. On the other hand, models of type 2 and 3 produce a far superior ranking. Until an inspection size of 80 classes, the change discovery is between 8 and 10%. Using the maximum aggregation strategy, type 2 model, produces the best rankings for the top 10 classes. After these top classes, the quality of its rankings drops significantly below that of type

3 models. Finally, of our different composition strategies, we can see that the degree of a method is consistently a good strategy to guide inspections, and outperforms the PageRank, which is correlated to size (35%, vs. 19% for degree).

Our results indicate that a team developing a quality model should know what it will be used for. To locate high-change classes for tasks like testing where size might not be an important factor, simple size-based strategies can be interesting. In other situations that are dependent on the size of the class inspected, size should not be used. The type 2 and 3 models produce acceptable results for both situations.

#### **6.3.6.2 Threats to Validity**

Although we studied open-source systems, we believe that our approach would be applicable in an industrial context. We have no reason to believe that closed source systems would be structured any differently from the systems we analysed. The real issue lies in the difficulty of gathering data. Already, it is difficult to gather one-level data; to improve a simple type 1 model, metrics need to be extracted at different levels. In our study, however, we used a general training data set completely independent of our target systems. The models achieved interesting results. In the absence of local data, a generic corpus could be a starting point for an industrial quality model effort. Furthermore, we showed that it is possible to modify the aggregation strategy to improve the performance of a generic quality model.

#### **6.3.6.3 Call Graph Generation**

There are two issues with call graph generations that have a direct impact on the results of the quality models. In our opinion, a major limitation that would limit the acceptance of our approach in an industrial setting is the cost required to build the call graphs. The 0-CFA algorithm used in this study can require days of computation for large programs, which can be unacceptable in certain contexts. Another important issue of the 0-CFA algorithm is that it analyses whole programs. Unless we can accept the additional imprecision of an approach as [DH08], we are limited to the study of full-

fledged applications (no libraries and frameworks). Depending on the precision required of a call graph, we could use cheaper call graph building techniques like RTA (rapid type analysis) and CHA (class hierarchy analysis) that are extremely efficient, but less precise. Furthermore, these techniques can handle incomplete applications at the cost of adding more imprecision. In a study of call graphs [AVDS10], we showed that RTA could be an interesting alternative to sophisticated analysis techniques.

Another issue with the use of statically constructed call graphs is the support of dynamic class-loading. Most modern applications use reflection. We noted that as it was used in a non-trivial manner in ArgoUML. In our previous study [AVDS10], we found that nearly 2% of classes use reflection; certain parts of the application are unreachable without the use of this reflection. The limited support in existing tools encourages us to investigate alternative ways to support dynamic features in modern languages. Existing techniques include combining static and dynamic analysis as [BSSM10].

#### **6.3.6.4 The Generalisability of the Results**

Our study demonstrates that combining information from different levels can improve the identification of high-change classes. We must note that our results are based on the observation of only one system, and we therefore cannot generalise beyond this system. However, we believe that we have found an interesting research avenue as we explicitly treat method calls as complex graphs. Furthermore, the fact that we merely replaced part of a quality model by a more sophisticated sub-model and that our results improved is an indication that it is possible to improve on state of the art practices basing our quality models on a purely static analysis of code.

### **6.4 Related Work**

In this chapter, we presented change models, but our primary purpose was not to conduct research specifically in that area. We used these models as a test bed to test our multi-level quality models. Here, we briefly describe existing work on change prediction and changeability evaluation.

There are different studies that focused on predicting software changeability. In one of the first articles assessing maintainability, Munson and Khoghofaar studied the use of procedural metrics to identify high-change modules [MK92]. When large OO systems started being produced, researchers performed similar studies on OO systems. Li and Henri conducted the first study of the maintainability of classes when they tried to predict changes in classes using two size metrics as well as the CK metrics suite [LH93b]. Arisholm *et al.* [ABF04] compared the performance of dynamic and static coupling metrics for the classification of change-prone classes. Koru and Lio identified used several metrics to identify and characterise high-change classes [KL07]. Finally, Koru and Tian investigated how well high values of metrics corresponded to high levels of change [KL05]. What these different studies share is that they use method-level metrics that are aggregated implicitly or explicitly to assess the changeability of classes. Our research challenges the assumptions that class-level metrics are sufficient to predict the changeability of classes.

Other studies evaluated the changeability of classes that compose design patterns and design defects [DCGA08, KGA09, KDG09]. Khomh *et al.* [KDG09] considered the effect of bad smells on the changeability of classes. They found that classes presenting more smells were more likely to change. We noted that some smells affect the methods (*e.g.*, Long parameter list), others affect classes. They considered that when at least one “smelly” method was found, the class presented the smell. From our point of view, we believe that these smells should have been studied at the method-level

Design patterns are another example where composition comes into play. In [KGA09], Khomh *et al.* investigated the relationship between the change frequency of a class and the roles it plays in a design pattern. They found that classes playing one or more roles tend to change more than those playing no roles. Furthermore, these classes tend to be more complex (as measured by many metrics). This work is similar to ours as they consider quality in the composition of classes. Classes are combined by role to form a design pattern. What they investigated was the effect on a class “being contained” within a pattern whereas, we consider the opposite relationship: a class “containing” methods. In [DCGA08], Di Penta *et al.* showed how specific roles in design patterns affect the



change-proneness of a class.

## 6.5 Conclusion

The majority of existing models predict quality using metrics extracted on a software entity as a whole. They do not explicitly consider the fact that most of these entities are in fact containers. Yet, many commonly used metrics are in fact aggregates of lower level data, but the aggregation mechanisms used are naive. In this chapter, we introduced the notions of quality composition using component quality models and that of importance functions.

Our approach was presented on the example of class changeability. We adapted an existing change model to include method-level information and tested it in an exploratory study. Our study showed that using component-level quality models increases the discovery of high-change classes. Furthermore, we show that using an intelligent importance function (Markov Centrality and Page Rank) produced the best rankings. The next logical step would extend this study to additional systems to ensure that the results found are generalisable.

Nevertheless, we consider our results to be very interesting as they open many research perspectives. In particular, we found that central methods tend to change more often than non-central methods, but there are many other aspects of software that could be used to assess importance. We believe that there is a lot of potential work to be done in assessing importance. For example, we could build cognitive models describing how quality engineers explore a code base instead of simply using execution possibilities. However, building such models is non-trivial as it requires data to analyse. This type of data is currently unavailable and would require controlled experimentation to be collected.



## CHAPTER 7

### COMPOSITION OF QUALITY FOR WEB SITES

Although, we focused on the quality of object-oriented software in previous chapters, we also explored the use of our type 3 models in other contexts. In this chapter, we present a second application of our composition strategy: we applied the strategy to the problem of evaluating web site navigability<sup>1</sup>. We decided to go outside of the realm of traditional software development to show how composition models can bridge the gap between a code-level view and a user-level view. This type of model could help a development team focus its programming effort on adding value to users. We believe that since web sites are mostly used by non-experts there is potentially much more high-level data to exploit in quality models. Our site navigation model combines low-level information from individual pages as well as the presence of site-wide navigation mechanisms.

Arguably the most important problem with evaluating navigability is that it depends on the exploration strategy of a site by a user [Pal02]. As noted before, this is the case with all high-level notions of quality, it is subjective; different users will use different ways to navigate a site and have different opinions of their experience doing so. This chapter consequently uses the same Bayesian modelling as in Chapter 4.

#### 7.1 Problem Statement

In this section, we present how users navigate web sites and describe our adaptation of our composition model to this problem.

---

<sup>1</sup>The content of this chapter is the subject of a publication at the 12th IEEE International Symposium on Web Systems Evolution [VS10].

### 7.1.1 Finding Information on a Web Site

A user typically has two options to find information on a web site. He can either explore the site, going from page to page by following links or, if available, he can use other features like a search engine to access pages directly. Consequently, the general navigability of a web site needs to take into consideration both ways to navigate the site. First, it needs to evaluate the impact of visiting every page on the navigability of the web site. This means that the model should evaluate how easy it is to find the appropriate link to follow on every page and combine this to the probability that a user will be on that page. Thus two models are involved: a page quality model and a navigation model. In addition, our method must take into account the alternate navigation mechanisms that are provided by the web site.

From an exploration perspective, a web site can be viewed as a directed graph; formally,  $G \Rightarrow \langle V, E \rangle$  where  $V$  and  $E$  are respectively the set of vertices representing the pages and the set of directed edges representing links between pages. An edge  $(u, v)$  represents a link in the page  $u$  to the page  $v$ . Vertex  $u$ , is called the head of the link and  $v$ , the tail. For vertex  $u$ , the *out-links* is the set of links with  $u$  as the head, representing the links to the other pages, and *in-links* is the set with  $u$  as the tail, representing the links to  $u$  from the other pages.

A user requiring information located at page  $p_{dest}$  needs to find a path  $(p_1, p_2, \dots, p_{dest})$  in  $G$  that takes him from his origin  $p_1$  to his destination  $p_{dest}$ . In terms of the graph, this is a greedy path-finding problem where at any given page a user needs to figure out which out-link leads him closer to his destination. A site with good navigability should ensure that few steps are required to reach any destination. Some potential navigation difficulties arise due to pages with inadequate link identification (*e.g.*, no titles and bad anchor text) or to pages that overwhelm him with too much information (*e.g.*, the user needs to scroll down to find the correct link). There are consequently two sources of information that influence this view of navigability: the quality of individual pages, and the presence of these pages on an exploration path.

Another option available to a user is to use a search engine. By using the search

engine, the user jumps directly to another page. Exploration is pertinent even if the site has been indexed by a search engine because, lacking the correct keywords, a user may not find the page he needs from the index. The two methods of navigation are thus complementary.

### 7.1.2 Assessing Site Navigability

Figure 7.1 shows our composition model adapted for the evaluation of web site navigability. The three sub-models handle respectively three kinds of decisions: the navigability at individual pages (white boxes), the importance of each page in the site to weight the contribution of its navigability (light-gray box), and the navigability at the site level (dark-gray box).

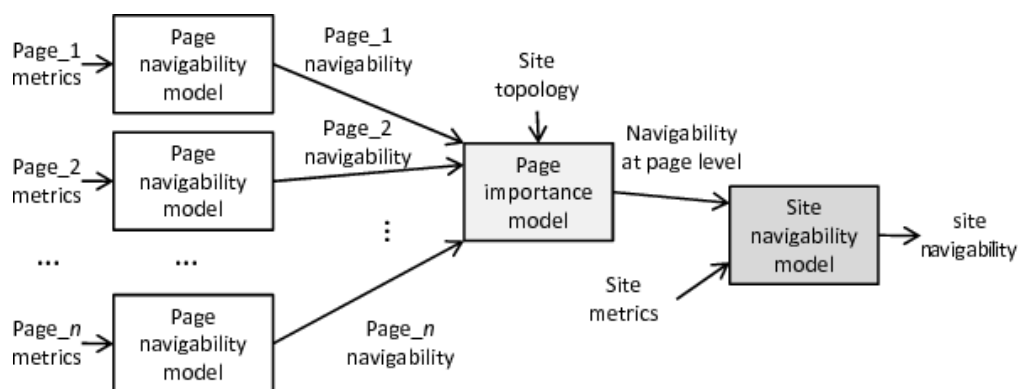


Figure 7.1: Navigability Evaluation Process

The different models describe the following aspects of navigability:

- **The Page-level model** describes the ability of a user to find relevant navigation information on a given web page. This information can either be to find the correct link to follow, or whether or not the page follows standard navigation practices that allow users to go to the site's home page or simply go back.
- **The Composition model** describes how likely a user will land up on a given web page, and need to interact with the page.

- **The Site-level model** uses both the result of the composition model and a set of site-level metrics. These site-level metrics describe the navigation mechanisms available site-wide. For example, the presence of a search engine is a site-wide mechanism.

## 7.2 Related Work

There is an abundance of information describing how to build usable Web sites (*e.g.*, <http://usability.gov>). This information is typically provided by practitioners. However, unlike usability, the problem of building navigable sites is mostly the subject of research articles. Zhang *et al.* [ZZG04] proposed complexity metrics to evaluate navigability. Newman and Landay considered it as one of three aspects affecting the quality of the interface design of Web applications [NL00]. Olsina *et al.* [OLR01] decompose quality hierarchically and navigability is a factor affecting the suitability quality sub-characteristics. Zhou *et al.* [ZLW07] proposed a navigation model that abstracts the user Web surfing behaviour as a Markov model. This model is used to quantify the navigability. Cachero *et al.*

[CCMG<sup>+</sup>07] used a model-driven approach to define a model for the measurement of navigability and a process for evolving this model. Finally, Ricca and Tonella [RT01] propose using the UML to represent Web pages. Using this representation, they present TestWeb, a tool to generate test cases.

Other methodologies consider additional characteristics to assess quality of web applications. For instance, Olsina *et al.* [OLR01] define WebQEM (Web Quality Evaluation Methodology). Albuquerque *et al.* [AB02] suggest FMSQE (Fuzzy Model for Software Quality Evaluation) model. The model uses fuzzy logic and presents a quality tree for e-commerce applications. It takes into account problems related to uncertainty during quality evaluation. Shubert *et al.* [Sch03a] develop EWAM (Extended Web Assessment Method). The method is based on Fishbein's behavioural model and Davis' technology acceptance model. It is applied to e-commerce web sites and is supported by a tool. Recently, Mavromoustakos *et al.* [MA07] use importance-based criteria for eval-

uating requirements in their quality model WAQE (Web Application Quality Evaluation model). Regarding the use of probabilistic approaches for quality assessment, Malak *et al.* [MSBB06] propose a method for building web application quality models using Bayesian networks. The approach of Malak *et al.* was used by Caro *et al.* [CCdSP07] for the particular case of web portal data quality.

Finally, there is work done to model the behaviour of a user navigating a web site to find specific information [CRS<sup>+</sup>03]. This work is very similar to the use of call graphs (described in Section 6.3) that are used to identify what methods can be invoked for a given execution. In our quality models, we are not interested in understanding the behaviour of users conducting specific activities (*e.g.*, finding information  $X$  located on page  $Y$ ), but rather we want an estimate of the general acceptability of a site.

### 7.3 A Multi-level Model to Assess Web Site Navigability

In this section, we present the details of our multi-level navigability model. We start by presenting the special type of Bayesian models used called Bayesian Belief Networks. Then, we present the page model, the composition model, and finally, we finish with the site-level model.

#### 7.3.1 Bayesian Belief Networks

The models proposed are based on Bayesian Belief Networks (BBNs) [Pea88], a specialised version of Bayesian models. BBNs are useful when there are too many inputs for a conventional Bayesian model. Probabilistic modelling is based on Bayes' conditional probability theorem, which combines the inherent probability of an output ( $A$ ) with its dependence on inputs ( $B$ ) as well as the probability of ( $B$ ) occurring. This is expressed by the following equation:

$$P(A|B) = P(B|A) \times P(A)/P(B) \quad (7.1)$$

A problem with traditional Bayesian models is that we need to consider the effect of every combination of inputs on an output. To consider the effect of  $n$  binary in-

puts, we would need to build conditional probability tables (CPTs) describing what the model should predict given  $2^{(n-1)}$  combinations. BBNs use a graph structure to organise conditional dependencies and limit the size of the problem, thus simplifying the problem [Hec95].

A BBN [Pea88] is a directed, acyclic graph that organises the dependencies of a set of random variables ( $X$ ). Every vertex of the graph corresponds to a variable and every edge connecting two vertices indicates a probabilistic dependency from the head, called parent, to the tail, called child. The random variables represented in the graph are only conditionally dependent on their parents. Each node  $X_i$  in the network is associated with a conditional probability table that specifies the probability distribution of all of its possible values, for every possible combination of values of its parent nodes.

When building a BBN, each vertex should correspond to either a concept that is observable (and measurable) or to a decision point given inputs defined by parents. The edges should be used to represent causal relations between vertices and allow a developer to interpret the results of an evaluation (*e.g.* the output is caused by this input). This structuring can be done either automatically using heuristics found in the literature [Pea00] or manually to correspond to a specific decision process. Since there is abundant information on how to evaluate the quality of web sites, we chose the latter. The structure only determines the dependencies between variables, the exact joint distribution needs to be defined in the form of a CPT. These tables can be learnt using historical data, or entered by an expert.

### 7.3.2 Assessing a Web Page

To assess the navigability of a web page, we adapted a BBN from [MSBB10]. This model is presented in Figure 7.2. In their study, the authors presented and validated a model that evaluates the navigability of a page with the possibility of using site-level mechanisms. This model was obtained using a GQM approach [BCR94] to refine navigability characteristics collected from ten sources (model proposals, standards, guidelines, etc.) In our proposed approach, as we separate page and site influences, navigation mechanisms are moved to the site-level model. Figure 7.3 presents this modified BBN.



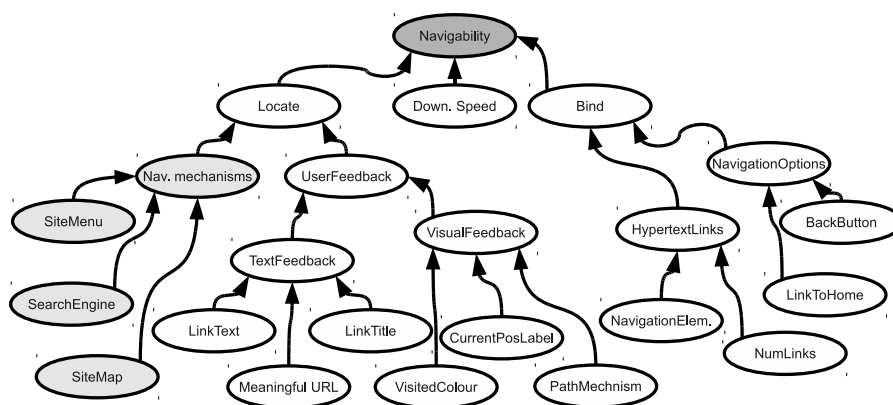


Figure 7.2: The original page-level navigability model: site-level metrics are identified in light gray

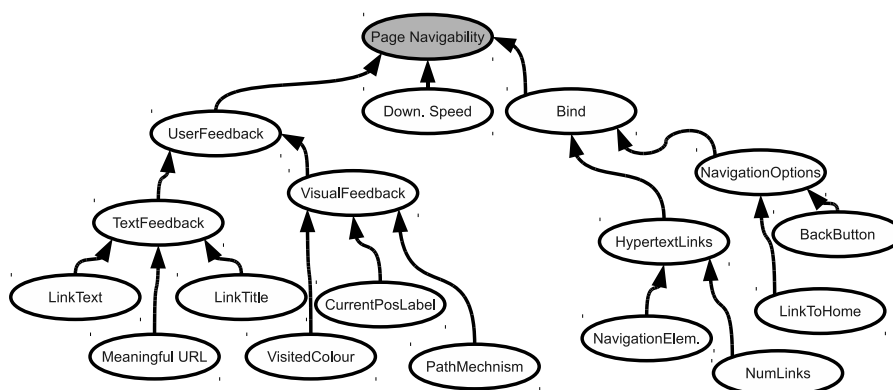


Figure 7.3: The modified page-level navigability model

We conserved the CPTs from this original BBN.

The decision of whether or not a page is easily navigable (navigability node) directly depends on three sub-characteristics: the ability of a user to identify the correct link to follow (*UserFeedback*), his access to available navigation mechanisms (*Bind*), and the size of a downloaded page (*PageSize*). Both the *UserFeedback* and the *Bind* nodes are intermediate decision nodes, which depend on other sub-characteristics that are then decomposed into metrics. This model has 10 inputs that are described in Table 7.I.

Table 7.I: Inputs to the navigability model

Metric	Node	Type
Page Size	PageSize	measure (count)
Ratio of links with titles	LinkTitle	measure ([0,1])
Ratio of links with text	LinkText	measure ([0,1])
Significance of page URL	Meaningful URL	binary
Indication of location in web site	CurrentPosLabel	binary
Visited links change colour	VisitedColour	binary
Presence of breadcrumbs	PathMechanism	binary
Number of links in page	NumLinks	measure (count)
Link to home (Home)	LinkToHome	binary
Support for Back Button (BB)	BackButton	binary

### 7.3.2.1 Input Metrics

In our model, every input needs to be converted to a random *discrete* variable. Binary metrics with two values:  $\{T, F\}$  are handled directly; but numeric metrics must be converted to a set of discrete ordinal values, *e.g.*, *low, medium, high*. These values should reflect what a user might answer in a survey. For example, he might state that, for a particular page, there are breadcrumbs,  $P(PathMechanism = T) = 1$  and that there are many links  $P(LinkNumber = High) = 1$ . However, it is not only unfeasible to maintain a group of users on hand whenever a developer wants to evaluate his site, it is unlikely a user will be able evaluate every page in a large site. Consequently, this process needs to be automated as well.

From the automation perspective, binary metrics fall into two categories. For the first one, it is possible to determine automatically if the value of the metric is true or false. For example, by analysing automatically the HTML source, one can decide whether the visited links change colour ( $VisitedLinkColor = T$ ) or not ( $VisitedLinkColor = F$ ).

Metrics of the second category are more difficult to extract automatically. This is the case, for example, of *PathMechanism*. Indeed, we can decide whether or not there are breadcrumbs only by using a heuristic rule. The rule uses two symptoms: the name and identifiers of divisions and the names of CSS (presentation) styles. If either element contains terms like “breadcrumbs”, then it is likely that the page contains breadcrumbs. Deciding into which group a given metric should fall corresponds to a classification problem. As we are dealing with heuristic, rules are encoded as classification BBNs

where, given a set of symptoms  $S$ , the probability that the metric belongs to a class  $C$  ( $T$  or  $F$ ) is determined by  $P(C|S)$ . An example of a BBN implementing the rule for breadcrumbs is given in Figure 7.4. The CPTs corresponding to these rules were trained on a set of randomly selected pages.

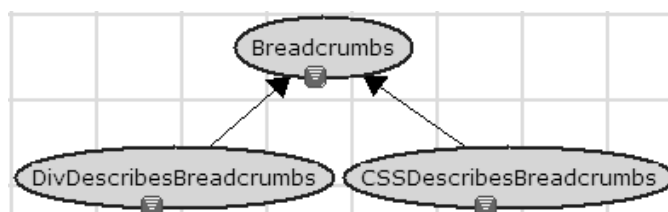


Figure 7.4: Binary input classification

To transform the numeric metrics into probability distributions, we calculated the probability that the metric would be classified, for example, as “low”, “medium” or “high” given a metric value. The exact number of classes depends on the attribute. The process followed is described in [SBC<sup>+</sup>02]. First, we extracted metrics from a set of randomly downloaded pages (over 1000). Second, for every metric, we derive the classes’ membership functions using fuzzy clustering. We used fuzzy kMeans clustering to find  $k$  centroids corresponding to the  $k$  classes (three in the case of *low*, *medium*, *high*). In this classification algorithm,  $k$  could be specified (usually two or three) or could be decided using Dunn partition coefficient [Tra88], which measures the quality of a classification ( $k$  that produces the best classification). Finally, the probability of a metric is calculated on its relative distance to the value with its surrounding centroids as we did in Section 4 when we used box-plots.

### 7.3.2.2 Executing the Model

The navigability of a page is assessed by evaluating the probability that the navigability node is true ( $P(Navigability = T | inputs)$ ) given the BBN structure and parameters (CPTs). Navigability only depends on its three parents nodes, two of which need to be recursively evaluated until an input node is reached. The value of input nodes needs to be computed on the evaluated web page. The precise CPT is shown in Table 7.II. The prob-

ability of having a good navigability ( $Nav$ ) would be calculated by evaluating the effect of all possible values ( $d$ ) of the parent nodes (shortened to Bind, Size and Feed) on  $Nav$  (illustrated in Equation 7.2). The individual parent nodes would need to be recursively evaluated.

$$P(Nav = T) = \sum_{Bind, Size, Feed \in \{d\}} P(Nav|Bind, Size, Feed)P(Bind, Size, Feed) \quad (7.2)$$

Table 7.II: Page navigability CPT

Bind	PageSize	UserFeedback	True	False
True	High	True	99%	1%
True	High	False	55%	45%
True	Low	True	80%	20%
True	Low	False	35%	65%
False	High	True	55%	45%
False	High	False	10%	90%
False	Low	True	45%	55%
False	Low	False	1%	99%

### 7.3.3 Navigation Model

The role of the navigation model is to evaluate the *importance* of each page, *i.e.*, the probability that a user will transit by that page to reach the desired page. This information is used to weight the navigability scores obtained by the page navigability model in order to produce a unique input to the web site navigability model.

Many existing algorithms decide on the importance of a page given the topology of the site. One of the best known algorithms is the PageRank [PBMW99]. It determines the probability that a user randomly clicking links will reach a given page. This particular algorithm discriminates against pages with large numbers of outgoing links and few incoming links. Yet, with regards to navigation, these pages are relatively important since they provide a way for users to reach many possible destinations.

Our algorithm to compute importance is also based on random walks, but differs

in the sense that it assigns more importance to transit nodes (like the home page). For every given navigation depth, it calculates the probability a user will reach a page given that at every page  $p$  he has a uniform probability of following any outgoing link ( $1/outlinks(p)$ ). The precise algorithm used is based on a breadth-first search and is presented in the following Algorithm:

**Algorithm:** Visit probability

**Inputs** : home: the start page

**Inputs** : outlinks: a vector of outlink for a page

**Outputs:** weight: a vector describing the relative weight of a page

$Q \leftarrow \text{empty-queue}$

$mark[home] \leftarrow \text{visited}$

$weight[home\_page] \leftarrow 1$

$Clicks \leftarrow 0$

enqueue home into Q

**repeat**

    dequeue page from Q

**foreach**  $Link(page, v) \in outlinks(page)$  **do**

$weight[v] \leftarrow weight[v] + weight[page]/|outlinks|$

**if**  $mark[v] \neq \text{visited}$  **then**

$mark[v] \leftarrow \text{visited}$

            enqueue v into Q

**end**

$Clicks \leftarrow Clicks + 1$

**end**

**until** Q is not empty ;

**forall**  $v \in weight$  **do**

$v \leftarrow v/Clicks$

**end**

**return** Visits

The way of modelling navigation will lend more weight to the home page, especially in shallow sites, which is normal when we consider that most users need to transit through there. As with the breadth-first search, the algorithm assumes that a user will only visit a page at most once. This is understandable since a user that is trying to locate information enabling him to return to a previously visited page, will likely go back and try another link instead of continuing on. The number of total possible clicks is used to calculate the relative weight of a page between  $[0, 1]$ .

The contribution of a page to the navigability of the site is determined by the function *weighted\_importance* (Equation 7.3) where *prob(page)* is the probability that a user will visit *page* and *nav(page)* is its navigability score as judged by the page-level navigability model.

$$weighted\_importance(page) = prob(page) \times nav(page) \quad (7.3)$$

The total page navigation score of a site is calculated by the function *total\_page\_nav* (Equation 7.4). As expected, when all of the pages are of the same quality *q*, this function also returns *q*, no matter the contribution of the navigation model. Furthermore, a common situation is when there are many pages at a same depth that follow the same template. This is sometimes due to dynamic generation of pages. In this case, the influence of the template will be relative to the number of pages generated.

$$total\_page\_nav(site) = \frac{\sum_{p \in site} weighted\_importance(p)}{|site|} \quad (7.4)$$

#### 7.3.4 Assessing a Web Site

The navigability of a site depends both on the ability of a user to navigate pages to find the desired page and the presence of site-level mechanisms. Three such mechanisms are the presence of a menu, a site map and a search engine. All three mechanisms allows for quick moves around different parts of the site.

A standard navigation menu in a site allows users to quickly switch to different parts of the site. Even though the menu provides links, which are taken into account when calculating the navigation path, semantically, a menu has a specific meaning for users, which improves a site's navigability. The same can be said of a site index, which links to many other pages, but essentially provides a sense to how the site is organised and how to navigate. Finally, a search engine allows access to parts of the site that might not be explicitly linked.

The proposed model is illustrated in Figure 7.5. It combines the concept of navigation mechanisms (a sub-graph) and page navigability (the other sub-graph). For the

sub-graph NavigationMechanisms, we reused the CPT given at the page level of the work in [MSBB10]. The input PageNavigability takes the probabilities obtained from Equation 7.4.

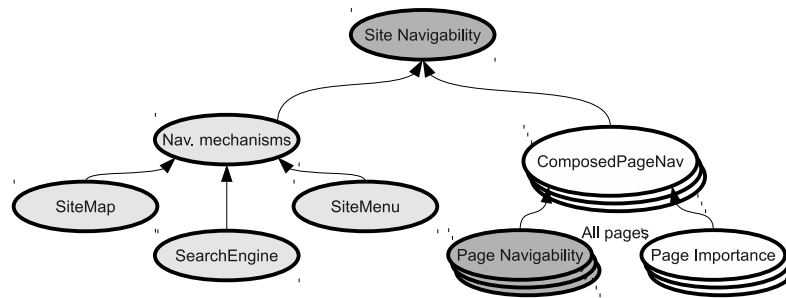


Figure 7.5: Site-level quality model

## 7.4 Case Study

In this section, we present a study that serves to establish whether or not the model correctly simulates the judgement of a user. Then, we show how the model can be used to guide maintenance efforts on a site.

### 7.4.1 Study Setup

Two groups of web sites were analysed: a group of “good” sites and a group of randomly sampled sites. The good sites are either Webby award winners or nominees<sup>2</sup>. In the judging process, different experts evaluate the sites according to six criteria including their navigability. The randomly selected sites are pages linked by <http://www.randomwebsite.com>; they include both personal sites as well as large businesses. The set of good and random sites is disjoint. In this study, we test whether or not the model finds a significant difference in navigability between the “good” sites and the random ones. To this end, we perform a mean- difference test. *t*-test is used if the data is

<sup>2</sup><http://www.webbyawards.com/webbys/>

normally distributed. If not, a Mann-Whitney test is used instead. Normality distribution is checked using a Kolmogorov-Smirnov test. The tested groups contain respectively 9 good sites and 14 randomly selected sites.

To download and evaluate the sites, we built a web crawler based on HtmlUnit, a web-testing library. HtmlUnit<sup>3</sup> supports JavaScript heavy pages. It allows our crawler to treat the majority of sites. We also used this library to extract metrics. The two quality models were built and executed using BNJ<sup>4</sup>, a library for probabilistic reasoning.

### 7.4.2 Navigability Evaluation Results

The results of our experiments are shown in Figure 7.6. Our approach clearly differentiates the sites with good navigability from those that were selected randomly. Good sites had an average navigability score of 0.74 vs. 0.51 for random sites. The scores were normally distributed as indicated by a Kolmogorov-Smirnov test. Using a t-test (Table 7.III), we found that the difference is statistically significant with a p-value of 0.00. We can then reject the null hypothesis that there is no difference between the two groups in terms of navigability as evaluated by our model. We alternatively confirm that the navigability model is able to correctly discriminate between web sites.

Table 7.III: Independent samples test

t	df	Sig. (2-tailed)	Mean difference	Std. error diff.
4.871	11.328	.000	.22365	.04592

### 7.4.3 Discussion

In addition to confirming statistically that our model is able to discriminate between web sites in terms of navigability, our results allow us to look closely to three additional points of interest: (1) can our model handle pages generated from templates? (2) is it necessary to evaluate exhaustively all the pages of a site?, and (3) how could we use the navigability model to improve a web site?.

<sup>3</sup><http://htmlunit.sourceforge.net>

<sup>4</sup><http://bnj.sourceforge.net>



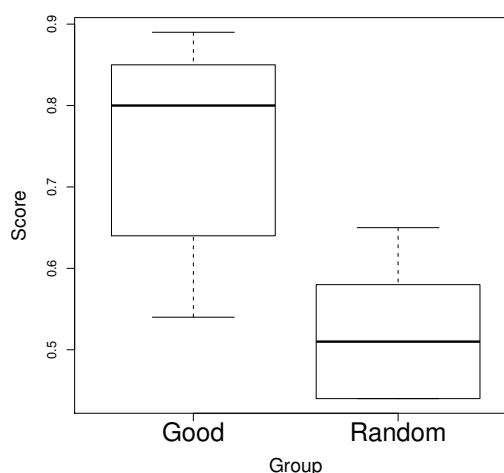


Figure 7.6: Navigability scores for good and random sites

For the first point, we noted that the “good” sites were much larger than the average site and make heavy use of templates. As we conjecture in Section 7.3.3, our navigation model correctly manages these sites. Our work could be enhanced by the inclusion of wrapper (template) inference techniques [CMM02] that can automatically identify these templates.

The second point of interest is that we did not see any significantly different score in navigability between analysing a complete site (thousands of pages) and a portion of the site (a hundred of pages). This is good news since sampling pages from a large site is sufficient for its evaluation. However, what is the determining factor is actually the depth of the search since it is at different levels that page quality tends to vary.

A final important point that is worth discussing is how to use our model to improve navigability. This is important because a recurrent concern of industry is the relationship between quality evaluation and quality improvement [SGM00]. Figure 7.7 shows, for example, the result of the execution of our model on a good site. The navigability of this site is considered good with a probability of 77.5%. The developers/managers could consider that this probability is not high enough and that they could improve the site to increase its navigability. Given the fact that there is a site menu (NavigationEle-

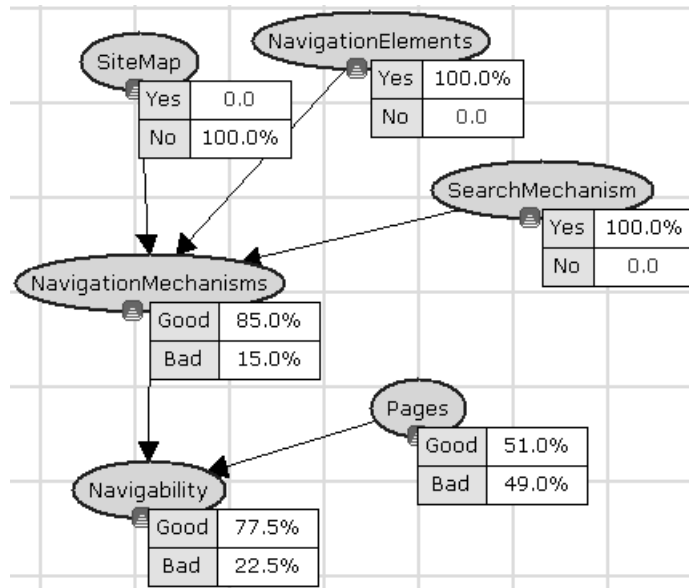


Figure 7.7: Initial model

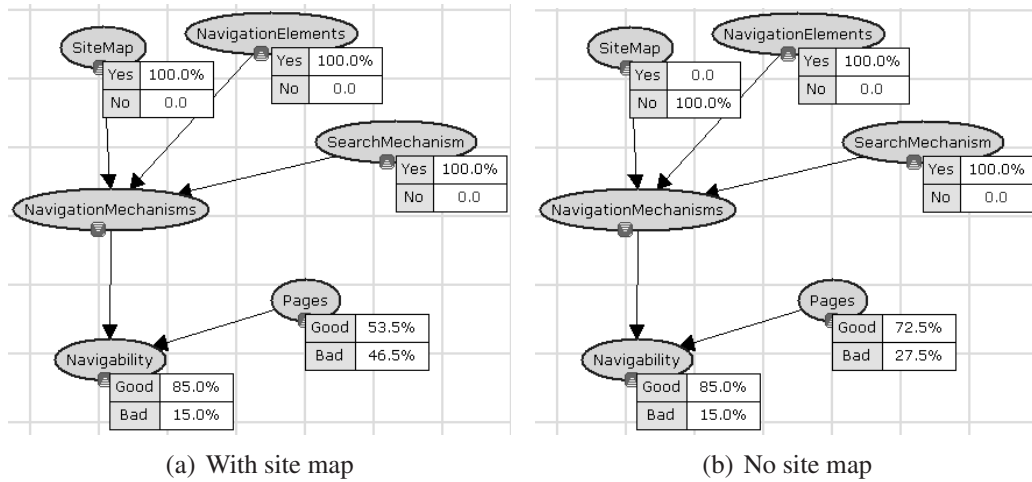


Figure 7.8: Potential improvements to the site

ments) and a search engine, there are two possible ways to improve the site: improve the navigability of the pages, and/or add a site map. Already the site-wide mechanisms are judged to be relatively good with  $P(\text{NavigationMechanisms} = \text{good}) = 85\%$ , but the page navigability is relatively bad ( $P(\text{Pages} = \text{good}) = 51\%$ ).

An important property of BBNs is that we can set the probabilities of any node (including the Navigability node) and standard algorithms can update the probability distribution of other nodes accordingly. Let's set a managerial objective of increasing general navigability to a level corresponding to a probability of 85%. A manager could set the output node's value to 85% and try two configurations: keep the site without a map ( $P(\text{SiteMap} = \text{Yes}) = 0$ ) and add a site map ( $P(\text{SiteMap} = \text{Yes}) = 1$ ). With a site map, page-level navigability needs to increase to 54% (Figure 7.8(a)) to reach the managerial objective of 85% (almost equal to the current value of 51%). Without the site map, page-level navigability needs to be increased to 72% (Figure 7.8(b)). The development team could then estimate the cost of adding the map and the cost of modifying the pages and find the cheapest solution. For the second solution, as we know the individual navigability of the pages as well as their respective importance, some pages could be targeted. For a particular page, we can repeat the output-probability setting to determine the improvement option at the page level.

## 7.5 Conclusion

We applied the composition strategy presented in Chapter 6 to the problem of evaluating the navigability of web sites. We did so for four reasons. First, it shows that the proposed approach can be applied to more than one problem. Second, it bridges the gap with Chapter 4, as navigability is a subjective notion. Third, it shows how an aggregation mechanism can be used to provide information from one stakeholder (HTML programmer) to a higher-level stakeholder, the site owner. Finally, we showed how to use the model to guide improvements, an issue raised in Chapter 5.

There exists related work proposing models to evaluate different characteristics of web sites. However, these models target either the page or the site level, and do not

provide explicit mechanisms to integrate both. Furthermore, most models are guidelines, which have not been validated on real data. In this chapter, we proposed a simple way to aggregate page-level information to evaluate the quality of sites based on a random walk strategy. To evaluate the proposed model, we conducted a study on a sample of real web sites. Our results show that our multi-level navigability model is able to correctly discriminate between sites considered excellent and randomly selected sites.

The study presented here was a proof of concept that shows that it is possible to build and use a model to support all three model-improvement dimensions. This study could serve as a stepping-stone for a more extensive study, which would require a larger-scale validation. Furthermore, it could be interesting to extend some of our previous work [VBSH09] where we proposed a method to recommend improvements to a web page on the basis of a quality model. Given a model, a set of possible transformations, and an estimate of available resources, that method proposed an optimised sequence of transformations to apply to a page. Considering the problem complexity, meta-heuristics were used to find this sequence of transformation. We believe that such an approach could be adapted to work in a multi-level model.

## CHAPTER 8

### CONCLUSION

As society becomes ever more dependent on computer systems, there is more and more pressure on development teams to produce high-quality software. Clients now expect modern systems to be delivered at little cost, and be flexible enough to satisfy not only current needs, but their future needs as well. These expectations are such that in order to ensure a certain level of quality in their systems, companies have invested significant resources developing auditing tools like quality models.

Ideally, one would like to develop a system that could analyze a program and compute its quality. Unfortunately, proper evaluation requires intelligence, just like evaluating the literary merits of a novel. In our case, a "quality analyzer" would have to understand how a program works and see how this corresponds to the objectives of the designers as well as best practices. Current models are much cruder: they try to predict quality attributes like fault density or maintenance effort, based on low level source code statistics, like code size or coupling between modules. Research in this area can be frustrating because we know, a priori, that the relation between measured inputs parameters and "quality" is tenuous. At best, current models try to identify "problematic" bits of code based on extreme metric values. Modelling often relies on machine learning, training on historical data. Even here we run into problems because clean, representative data is hard to find. However, quality is important and any progress in state of practice has high pay-off.

We worked with a large industrial partner to find operational problems that were not adequately addressed in the literature and see what could be done to improve the situation.

- Industrial quality models similar to those built by our partner lack proper empirical validation;
- The thresholds found in these models are often arbitrary;

- The use of thresholds does not allow for graded judgment;
- Models use standard aggregation mechanisms without verifying if these mechanisms are correct;
- Models are not prescriptive. They do not suggest how best to correct problems found;
- Models are not management-oriented. The information they produce tends to be only useable by coders/quality engineers.

Even though our partner performed a literature review and tried to follow what it thought were best practices, we could still observe fundamental problems in its models. We do not believe this is an isolated case as many industry reports describe similar problems.

## **8.1 Contributions**

In this dissertation, we presented these problems in a study of our industrial partner’s quality modelling approach and proposed ways to improve the state of the art of quality modelling practices. We explored three dimensions to improve quality modelling practices: subjectivity, evolution, and composition.

During the course of this dissertation, we made the following contributions:

### **8.1.1 Handling Subjectivity**

We believe that part of the problem with using thresholds and hard rules is that different people have varying opinions on what is good code. We proposed a *method to identify high-level, subjective quality indicators and tested it on the problem of efficiently detecting the presence of anti-patterns*. Existing, state of the art techniques are based on detection rules written by experts and classify code as either as “clean” or “unclean”. This binary classification was shown to be inadequate because there were only a few cases where a class was unanimously considered an anti-pattern in the corpus studied.

Instead, we built Bayesian models to predict the probability that a quality engineer would consider that the design of a class is a defect. In these models, subjectivity is encoding using Bayes' theorem.

The models built using our method produced results equal or superior to existing rule-based models, and allowed for a flexible browsing of the candidate classes. Additionally, we showed that by returning ranked results, we can improve the efficiency of a manual validation of the set of candidates.

### 8.1.2 Supporting Evolution

Suggesting improvements is a key aspect of what IV&V teams do. It is therefore essential to get a better understanding of what type of quality evolution patterns exist in software systems. Using an anti-pattern detection model, we presented a *technique to track the quality of evolving software entities*. The vast majority of existing quality models evaluates the quality of a snapshot of a system. There are many cases when a team wants a dynamic view of the system under development in order to determine if development is improving or degrading its quality. For example, our partner is interested in identifying if the quality of certain parts of a system has degraded for contractual reasons, or to evaluate if a development team is adequately correcting previously identified problems. We proposed a technique to use a quality model to find interesting patterns in the evolution of the quality of classes. We consider sequences of quality scores as a signal and applied signal-based data-mining techniques. We were able to find groups of classes sharing common evolution habits.

In a study of two open-source systems, we found that vast majority of classes exhibiting the symptoms of Blobs are added to the program with these symptoms already present. This type of anti-pattern is consequently not the result of a slow degradation as sometimes expressed in the literature. Furthermore, many of these classes that seem to implement too many responsibilities play roles in design patterns, a structure that is typically thought to be a sign of good quality. In the systems studied, we also found that code correction is rare, more often than not, bad classes are replaced.

### 8.1.3 Composing Quality Judgements

Software is developed using different levels of abstraction, yet quality models generally do not explicitly consider these abstractions. Our partner however expressed the need for quality models at various levels. In the company, IV&V team members want detailed information concerning what parts of the system need additional testing/inspection, yet managers want to use a model to get an aggregate view of quality. In order to estimate a higher-level view of quality, we performed a *detailed study of different aggregation strategies* that combine quality information from a level of granularity to another.

We performed a study to identify high-change classes using both method and class-level metrics. We tested both traditional aggregation techniques and a *new importance-based approach*. We found that building a simple model relying only on size (measured by the number of statements) produces the best global inspection efficiency. This model is however outperformed by our importance-based approach when trying to identify the most changed classes. Since an IV&V team does not inspect the totality of results, and focuses on the riskiest classes, we showed that our approach, using the PageRank algorithm on statically generated call-graphs outperforms existing aggregation techniques. We would like to note that our use of statically constructed call-graphs to approximate runtime behaviour in quality models is new.

### 8.1.4 Application to Web Applications

Finally, *we applied our importance-based approach to the assessment of the navigability of web sites*. Many modern applications are now web-based, and thus we tested our importance models to evaluate whether or not a web site can be easily navigated by a user. We adapted an existing, low-level quality model and used an aggregation strategy to evaluate the site as a whole. Our technique could successfully differentiate between random sites and good sites (winners of Webby awards). This work is one of the few empirically validated studies in the field. This work served to connect all three previous contributions. The mechanisms used support subjectivity; we showed how the model



can be used to recommend improvements, and finally, our aggregation model produces managerial-level quality estimates from code-level data.

## 8.2 Future Research Avenues

This dissertation started out as an exploration of why some software is good, and other software bad. We would have liked to find some obvious laws of goodness, but we quickly realised that metrics describing program structure are not necessarily a good indicator of goodness. What we should be collecting and analysing is higher-level, semantic information. This data however is not available. We consequently had to focus on doing the best we could with the data at hand, a practice common in industry. We identified certain research perspectives that we deem of interest for future work.

### 8.2.1 Crowd-sourcing

In many cases, it is nearly impossible to collect the opinions of experts to study. An alternate way of getting opinions is by using *crowd-sourcing*. For simple activities, instead of relying on highly trained “experts”, you pay individuals very small amounts of money (in the dollar range) to perform simple tasks in their spare time. In fields like translation [CB09], crowd-sourcing has produced better results than experts, while being cheaper. A way to perform crowd-sourcing is by using mechanical Turks like those provided by Amazon <sup>1</sup>. These Turks might not be usable to evaluate software, but could be used to conduct relatively inexpensive controlled experiments to evaluate Web sites. This data could help improve the navigability model presented in Chapter 7.

### 8.2.2 Using Complex Structures instead of Metrics

Most research studies the influence of code-level metrics on quality indicators, but what we would like to study is higher-order semantic data. We believe that the simplification of our representation of code to a collection of metric values reduces the efficiency

---

<sup>1</sup><https://www.mturk.com/mturk/welcome>

of quality models. A quality model should be able to reason using the relation of different code-level entities to assess the quality of a system. From an industrial perspective metrics are useful because they can be extracted by easy to use commercial tools, and included in different types of models, but from a research perspective, metrics are of limited value. We believe that code should be treated using alternative representations such as the following:

### **8.2.2.1 Predicates**

In our work with Marouane Kessentini [KVS10], we represented code as predicates encoding the interactions between methods, attributes and classes, and found that this was worthwhile for the problem of anti-pattern detection. We believe that these results could be generalised to general quality models, and that a richer representation of code should improve the general performance of quality models. Of course, these analyses are done at a significant cost in terms of performance.

### **8.2.2.2 Graph Representations**

Another representation of code is as a graph. Graphs are commonly used to describe both static and dynamic characteristics of systems. Call graphs are used to describe executions, and abstract syntax trees are indications of the structure of code. In our study of different composition strategies to combine the quality of methods on the quality of classes, we performed a graph analysis of a statically constructed call-graph to approximate the runtime importance of methods. This graph representation allowed for a more complex analysis of the flows of information than existing metrics (*e.g.*, fan-in and fan-out). In fact, traditional metrics measure attributes locally, but by analysing a representation of a whole program, it is possible to get a more accurate view of these attributes.

### 8.2.3 Activity Modelling

In Chapter 6, we presented standard importance functions. Since these were applied to call graphs, we considered that the importance of a method depends on its execution. For quality characteristics like reliability, this assumption certainly holds true. However, if we are interested in assessing the understandability of the code, we might want to determine importance using a code exploration model that is independent of the notion of execution. We believe that an aggregation model should be mapped to an activity, like the exploration model we used for web sites. There is, however, research to be done in understanding and modelling what types of activities are done on software.

## 8.3 Other Research Paths Explored

During the five years we spent completing this dissertation, we explored other aspects of quality modelling that we do not detail for lack of space and sake of consistency. This section serves to briefly summarize our findings and warn others who wish to pursue these paths of the types of problems they can expect.

**Software design as an enabler** From the beginning of our research, we have always believed that good software design does not necessarily imply good quality. Rather, it is an enabler: a well structured/designed system should support the activities that are performed by a development team. Following this idea, we need two things: 1) an idea to know how developers plan on modifying the system, and 2) higher-order patterns (like design patterns) to indicate the adequacy of a structure to support certain changes. We did some preliminary work on this subject [VS07, VS08], but decided to move on to other research dimensions because it was almost impossible to build an adequate data set describing the activities performed by developers.

**The influence of changes on fault-proneness** The first two years of our work were spent building change models [VSV08]. We built a corpus comprising over 10 systems over multiple versions. This required a lot of grunt work as we had to download these

versions either from source-code repositories, or from software archive sites. In many cases, the code could not compile, or wouldn't run after compilation. This work required that we recover the set of dependencies required of every version of the system; some dependencies were not available on-line anymore. We also had a problem determining the origin of different software entities [GZ05]. In successive versions of the system, we had to identify whether “new” software entities were simply old ones that were renamed. We built tools to assist with this classification, but ultimately, we needed to validate the results manually.

The effort required to build such a corpus was not worth the effort from a research perspective as an impact of changes on quality indicators was either minor, or not statistically significant. Consequently, the majority of the corpus created was never exploited. We must however admit that by reading the code of these systems and by building the tools to performs code analyses, we developed a clear understanding of the problems we were trying to model and of the complexity of maintaining software. In our opinion, the key problem with our approach is that we studied *good* open-source projects. Open-source projects tend to be abandoned/forked when maintenance costs become too high. Consequently, the data we collected corresponded mostly to programs with good structures and few maintenance problems, the phenomenon we wanted to study.

Our recommendation to future researchers wishing to perform these sorts of analyses is to take time to find projects with quality problems. These will likely 1) be very large as there needs to be substantial motivation to keep a project alive when it is hard to maintain, or 2) closed-source where clients would pay for this difficult maintenance. In either case, researchers should expect spending a lot of time building, understanding and analysing their corpus.

**Recovering clean bug data from version control systems** We were interested in finding objective quality indicators, and the standard indicator of “bad quality” is the presence of bugs. In two open-source systems we analysed, we tried to manually locate bugs from version control system logs and annotate classes with this data. We were confronted with two problems. First, the notion of a bug depends on who looks at the code. A bug

for a user often did not correspond to a bug for the developer. In the systems analysed, often these bugs pertained to parts of a specification that were not implemented yet. In open-source projects, developers will generally deliver an “unfinished” product as soon as the important parts are ready to get feedback from the community. In this context, what a user might consider bug fix is in fact a new feature for the developer. In our efforts to build a clean corpus, we found that it was almost impossible, as an outsider, to differentiate an improvement to the code from a bug-fix without an explicit indication of the intention of the developers (*e.g.*, explicit mention of bug fixes in the versioning system).

Second, the number of obvious bugs identified can be very low. In one of the systems we inspected (JFreeChart), we looked at all the CVS logs (and relevant code) for several years of development. We found that only a few (obvious) bugs had been released over the course of a few years of active development. Therefore, the noise to signal ratio was very high. The errors we found were also trivially corrected. For example, we found badly encoded RGB colour schemes, an error introduced by a developer who incorrectly entered a constant value that only caused minor issues for users.

**Moral** Obtaining large corpora of clean, representative software upon which to train and test models is of primary importance to any prospective researcher in this area. And he must necessarily spend an inordinate amount of time gathering, understanding and analyzing data; the quality of the data will ultimately determine his research perspectives. If he ignores the type of data he has and tries to go forward anyways, his results will be built on shaky foundations.

Our recommendation is to reuse existing data sources, participate with other researchers to build/maintain clean corpora, or work with a development team willing to invest its time to providing insight as to what they did to a system and why they did it. The latter option is the best way to gather high-level metrics. This also requires significant investments for researchers who should build tools and integrate them within developers’ tool sets. If metric gathering is difficult or expensive for a development team, it will not be done properly.

## 8.4 Closing Words

Our work has been well received by the research community. We published 10 articles in software engineering. We have two articles describing anti-pattern detection. The first was presented at the International Conference on Quality Software (QSIC) [KVGS09] and was selected for a journal extension. The other will be presented at Automated Software Engineering (ASE) later this year [KVS10]. We had two articles on software evolution presented at the Working Conference on Reverse Engineering (WCRE) [VSV08, VKMG09], a top conference in reverse engineering. Work describing a semi-automatic technique to improve web site quality was presented at the WISE conference, an important web engineering conference. Our work has been cited in [CHH<sup>+</sup>10, ZK10], indicating that the community has accepted our work and is using it as a building block for future research.

Previously, we commented on the inordinate amount of work we invested in acquiring data, building metric tools and analysis. We estimate that we studied almost 200 person-years' worth of development<sup>2</sup>. This exposure to different development practices has made us aware of the alternatives and complexities facing software developers. In the future, our endeavours, whether in academia or industry, will be shaped by the work we did for this dissertation.

---

<sup>2</sup>Based on a COCOMO estimation provided by <http://www.ohloh.net>

## BIBLIOGRAPHY

- [AAD<sup>+</sup>08] Giuliano Antoniol, Kamel Ayari, Massimiliano Di Penta, Foutse Khomh, and Yann-Gaël Guéhéneuc. Is it a bug or an enhancement?: a text-based approach to classify change requests. In Mark Vigder and Marsha Chechik, editors, *Conference of the center for advanced studies on collaborative research (CASCON)*, pages 304–318, New York, NY, USA, 2008. ACM.
- [AB02] Adriano Bessa Albuquerque and Arnaldo Dias Belchior. E-Commerce Websites: a Qualitative Evaluation. In Irwin King, editor, *Posters of the 11<sup>th</sup> World Wide Web Conference (WWW)*, 2002.
- [AB06] Erik Arisholm and Lionel C. Briand. Predicting fault-prone components in a Java legacy system. In Guilherme Horta Travassos, José Carlos Maldonado, and Claes Wohlin, editors, *International Symposium on Empirical Software Engineering (ISESE)*, pages 8–17, New York, NY, USA, 2006. ACM.
- [ABF04] Erik Arisholm, Lionel C. Briand, and Audun Foyen. Dynamic coupling measurement for object-oriented software. *IEEE Transactions on Software Engineering*, 30(8):491–506, 2004.
- [AKCK05] Hiyam Al-Kilidar, Karl Cox, and Barbara Kitchenham. The use and usefulness of the ISO/IEC 9126 quality standard. In Ross Jeffrey, June Verner, and Guilherme H. Travassos, editors, *International Symposium on Empirical Software Engineering (ISESE)*, volume 0, pages 38–47, Los Alamitos, CA, USA, 2005. IEEE Computer Society.
- [AMAD07] Kamel Ayari, Peyman Meshkinfam, Giuliano Antoniol, and Massimiliano Di Penta. Threats on building models from CVS and Bugzilla repositories: the Mozilla case study. In Kelly A. Lyons and Christian Couturier, editors,

*Conference of the center for advanced studies on Collaborative research (CASCON)*, pages 215–228, New York, NY, USA, 2007. ACM.

- [AS06] El Hachemi Alikacem and Houari Sahraoui. Détection d’anomalies utilisant un langage de description de règle de qualité. In Roger Rousseau, editor, *actes du 12<sup>e</sup> colloque Langages et Modèles à Objets*, pages 185–200. Hermès Science Publications, March 2006.
- [AVDS10] Simon Allier, Stéphane Vaucher, Bruno Dufour, and Houari Sahraoui. Deriving Coupling Metrics from Call Graphs . In Cristina Marinescu and Jurgen J. Vinju, editors, *10<sup>th</sup> International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE Computer Society Press, 2010. [to appear].
- [BAB<sup>+</sup>05] Stefan Biffl, Aybüke Aurum, Barry Boehm, Hakan Erdogmus, and Paul Grünbacher. *Value-Based Software Engineering*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.
- [BBA<sup>+</sup>09] Christian Bird, Adrian Bachmann, Eirik Aune, John Duffy, Abraham Bernstein, Vladimir Filkov, and Premkumar T. Devanbu. Fair and balanced?: bias in bug-fix datasets. In Hans van Vliet and Valérie Issarny, editors, *ESEC/SIGSOFT FSE*, pages 121–130. ACM, 2009.
- [BBM96] Victor R. Basili, Lionel C. Briand, and Walcelio L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, 22(10):751–761, 1996.
- [BCR94] Victor R. Basili, G. Caldiera, and H.D. Rombach. *Encyclopedia of Software Engineering*, chapter The Goal Question Metric Approach, pages 528–532. John Wiley and Sons, 1994.
- [BD02] Jagdish Bansiya and Carl G. Davis. A hierarchical model for object-oriented design quality assessment. *IEEE Transactions on Software Engineering*, 28(1):4–17, 2002.



- [BD04] Magiel Bruntink and Arie van Deursen. Predicting class testability using object-oriented metrics. In Thomas Dean, Rainer Koschke, and Michael Van De Vanter, editors, *4<sup>th</sup> International Workshop on Source Code Analysis and Manipulation (SCAM)*, pages 136–145, Washington, DC, USA, 2004. IEEE Computer Society.
- [BDM97] Lionel C. Briand, Premkumar T. Devanbu, and Walcelio L. Melo. An investigation into coupling measures for C++. In W. Richards Adrion, editor, *International Conference on Software Engineering (ICSE)*, pages 412–421, 1997.
- [BDW98] Lionel C. Briand, John W. Daly, and Jurgen Wuest. A unified framework for cohesion measurement in object-oriented systems. *Empirical Software Engineering*, 3(1):65–117, 1998.
- [BDW99] Lionel C. Briand, John W. Daly, and Jurgen K. Wuest. A unified framework for coupling measurement in object-oriented systems. *IEEE Transactions on Software Engineering*, 25(1):91–121, 1999.
- [Bec99] Kent Beck. Embracing change with extreme programming. *IEEE Computer*, 32(10):70–77, 1999.
- [Bei90] Boris Beizer. *Software testing techniques (2nd ed.)*. Van Nostrand Reinhold Co., New York, NY, USA, 1990.
- [Bel00] Douglas Bell. *Software Engineering: A Programming Approach*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.
- [BFG07] Tibor Bakota, Rudolf Ferenc, and Tibor Gyimóthy. Clone smells in software evolution. In Ladan Tahvildari and Gerardo Canfora, editors, *23<sup>rd</sup> International Conference on Software Maintenance (ICSM)*, pages 24–33. IEEE Computer Society Press, October 2007.

- [BGA06] Salah Bouktif, Yann-Gael Guéhéneuc, and Giuliano Antoniol. Extracting change-patterns from CVS repositories. In Susan Elliott Sim and Massimiliano Di Penta, editors, *13<sup>th</sup> Working Conference on Reverse Engineering*, pages 221–230, Washington, DC, USA, 2006. IEEE Computer Society.
- [BMB<sup>+</sup>98] William J. Brown, Raphael C. Malveau, William H. Brown, Hays W. McCormick III, and Thomas J. Mowbray. *Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley and Sons, March 1998.
- [BMB02] Lionel C. Briand, Sandro Morasca, and Victor R. Basili. An operational process for goal-driven definition of measures. *IEEE Transactions on Software Engineering*, 28(12):1106–1125, 2002.
- [Boe78] B.W. Boehm. *Characteristics of Software Quality*. North Holland, Amsterdam, 1978.
- [Boe81] Barry W. Boehm. *Software Engineering Economics*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1981.
- [BP98] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks*, 30(1-7):107–117, 1998.
- [BR70] J. N. Buxton and B. Randell, editors. *Software Engineering Techniques: Report of a conference sponsored by the NATO Science Committee, Rome, Italy, 27-31 Oct. 1969, Brussels*. Scientific Affairs Division, NATO, 1970.
- [BR88] Victor R. Basili and H. Dieter Rombach. The tame project: Towards improvement-oriented software environments. *IEEE Transactions on Software Engineering*, 14(6):758–773, 1988.
- [BS87] Victor R. Basili and Richard W. Selby. Comparing the effectiveness of software testing strategies. *IEEE Transactions on Software Engineering*, 13(12):1278–1296, 1987.

- [BSK02] Salah Bouktif, Houari Sahraoui, and Balazs Kegl. Combining software quality predictive models: An evolutionary approach. In Giuliano Antoniol and Ira D. Baxter, editors, *International Conference on Software Maintenance (ICSM)*, pages 385–392, Los Alamitos, CA, USA, 2002. IEEE Computer Society.
- [BSSM10] Eric Bodden, Andreas Sewe, Jan Sinschek, and Mira Mezini. Taming reflection: Static analysis in the presence of reflection and custom class loaders. Technical Report TUD-CS-2010-0066, CASED, TU Darmstadt, March 2010.
- [BTH93] Lionel C. Briand, William M. Thomas, and Christopher J. Hetmanski. Modeling and managing risk early in software development. In *15<sup>th</sup> International Conference on Software Engineering (ICSE)*, pages 55–65, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.
- [BW84] Victor R. Basili and David M. Weiss. A methodology for collecting valid software engineering data. *IEEE Transactions on Software Engineering*, 10(6):728–738, 1984.
- [BW02] Lionel Claude Briand and J Wuest. Empirical studies of quality models in object-oriented systems. *Advances in Computers*, 56(0), 2002. Edited by M. Zelkowitz.
- [CB09] C. Callison-Burch. Fast, cheap, and creative: evaluating translation quality using Amazon’s Mechanical Turk. In *Conference on Empirical Methods in Natural Language Processing: Volume 1*, pages 286–295. Association for Computational Linguistics, 2009.
- [CCdSP07] Angelica Caro, Coral Calero, Juan Enriquez de Salamanca, and Mario Piattini. Refinement of a tool to assess the data quality in Web portals. In Aditya Mathur and W. Eric Wong, editors, *7<sup>th</sup> International Confer-*

- ence on Quality Software (QSIC)*, pages 238–243. IEEE Computer Society, 2007.
- [CCMG<sup>+</sup>07] Cristina Cachero Castro, Santiago Melia, Marcela Genero, Geert Poels, and Coral Calero. Towards improving the navigability of web applications: a model-driven approach. *European Journal of Information Systems*, 16(4):420–447, 2007.
- [CHH<sup>+</sup>10] S. Counsell, R. M. Hierons, H. Hamza, S. Black, and M. Durrand. Is a strategy for code smell assessment long overdue? In *ICSE Workshop on Emerging Trends in Software Metrics (WETSoM)*, pages 32–38, New York, NY, USA, 2010. ACM.
- [CK91] Shyam R. Chidamber and Chris F. Kemerer. Towards a metrics suite for object oriented design. In Alan Snyder, editor, *Object-oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 197–211, New York, NY, USA, 1991. ACM Press.
- [CMM02] Valter Crescenzi, Giansalvatore Mecca, and Paolo Merialdo. Roadrunner: automatic data extraction from data-intensive web sites. In *ACM SIGMOD International Conference on Management of Data*, pages 624–624, New York, NY, USA, 2002. ACM.
- [CNC<sup>+</sup>99] S. Chen, E. Nikolaidis, H.H. Cudney, R. Rosca, and R.T. Haftka. Comparison of probabilistic and fuzzy set methods for designing under uncertainty. In *40<sup>th</sup> AIAA Structures, Structural Dynamics, and Materials Conference and Exhibit*, pages 2860–2874, 1999.
- [Coh95] William W. Cohen. Fast effective rule induction. In *12<sup>th</sup> International Conference on Machine Learning (ICML)*, pages 115–123. Morgan Kaufmann, 1995.
- [CRS<sup>+</sup>03] Ed H. Chi, Adam Rosien, Gesara Supattanasiri, Amanda Williams, Christian Royer, Celia Chow, Erica Robles, Brinda Dalal, Julie Chen, and

- Steve Cousins. The bloodhound project: automating discovery of web usability issues using the infoscent simulator. In *Conference on Human Factors in Computing Systems (CHI)*, pages 505–512, New York, NY, USA, 2003. ACM.
- [CSN09] Aaron Clauset, Cosma Rohilla Shalizi, and M. E. J. Newman. Power-law distributions in empirical data. *SIAM Review*, 51(4):661–703, 2009.
- [DCGA08] Massimiliano Di Penta, Luigi Cerulo, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. An empirical study of the relationships between design pattern roles and class change proneness. In Hong Mei and Kenny Wong, editors, *International Conference on Software Maintenance*, pages 217–226, Piscataway, NJ, USA, 2008. IEEE Computer Society.
- [DCMJ06] Danny Dig, Can Comertoglu, Darko Marinov, and Ralph Johnson. Automated detection of refactorings in evolving components. In Dave Thomas, editor, *20<sup>th</sup> European Conference on Object-Oriented Programming (ECOOP)*, pages 404–428, 2006.
- [DH08] Barthélémy Dagenais and Laurie Hendren. Enabling static analysis for partial java programs. *SIGPLAN Not.*, 43:313–328, October 2008.
- [DMG07] Paul Duvall, Stephen M. Matyas, and Andrew Glover. *Continuous Integration: Improving Software Quality and Reducing Risk (The Addison-Wesley Signature Series)*. Addison-Wesley Professional, 2007.
- [Dro95] R. Geoff Dromey. A model for software product quality. *IEEE Transactions on Software Engineering*, 21(2):146–162, 1995.
- [DSP08] Karim Dhambri, Houari Sahraoui, and Pierre Poulin. Visual detection of design anomalies. In Christos Tjortjis and Andreas Winter, editors, *12<sup>th</sup> European Conference on Software Maintenance and Reengineering*, pages 279–283. IEEE Computer Society, April 2008.

- [DZ07] Valentin Dallmeier and Thomas Zimmermann. Extraction of bug localization benchmarks from history. In Alexander Egyed and Bernd Fischer, editors, *22<sup>nd</sup> IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 433–436, New York, NY, USA, 2007. ACM.
- [EBGR01] Khaled El Emam, Saida Benlarbi, Nishith Goel, and Shesh N. Rai. Comparing case-based reasoning classifiers for predicting high risk software components. *Journal of Systems and Software*, 55(3):301–320, 2001.
- [EDL98] Letha Etzkorn, Carl Davis, and Wei Li. A practical look at the lack of cohesion in methods metric. *Journal of Object-Oriented Programming*, 11(5):27–34, 1998.
- [EL07] Avner Engel and Mark Last. Modeling software testing costs and risks using fuzzy logic paradigm. *Journal of Systems and Software*, 80(6):817–835, 2007.
- [ESM<sup>+</sup>02] Stephen G. Eick, Paul Schuster, Audris Mockus, Todd L. Graves, and Alan F. Karr. Visualizing software changes. *IEEE Transactions on Software Engineering*, 28(4):396–412, 2002.
- [Eva03] William M. Evancho. Comments on "the confounding effect of class size on the validity of object-oriented metrics". *IEEE Transactions on Software Engineering*, 29:670–672, 2003.
- [EZS<sup>+</sup>08] Marc Eaddy, Thomas Zimmermann, Kaitlin D. Sherwood, Vibhav Garg, Gail C. Murphy, Nachiappan Nagappan, and Alfred V. Aho. Do crosscutting concerns cause defects? *IEEE Transactions on Software Engineering*, 34(4):497–515, 2008.
- [Fen91] Norman E. Fenton. *Software Metrics: A Rigorous Approach*. Chapman & Hall, Ltd., London, UK, UK, 1991.
- [FKN02] Norman Fenton, Paul Krause, and Martin Neil. Software measurement: Uncertainty and causal modeling. *IEEE Software*, 19(4):116–122, 2002.

- [FN99] Norman E. Fenton and Martin Neil. A critique of software defect prediction models. *IEEE Transactions on Software Engineering*, 25(5):675–689, 1999.
- [Fow99] Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.
- [FPB78] Jr. Frederick P. Brooks. *The Mythical Man-Month: Essays on Softw.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1978.
- [Fre79] L.C. Freeman. Centrality in social networks conceptual clarification. *Social networks*, 1(3):215–239, 1979.
- [Gar84] David Garvin. What does product quality really mean? *Sloan Management Review*, pages 25–45, 1984.
- [GC87] Robert B. Grady and Deborah L. Caswell. *Software metrics: establishing a company-wide program*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1987.
- [GFS05] Tibor Gyimothy, Rudolf Ferenc, and Istvan Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software Engineering*, 31(10):897–910, 2005.
- [GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1<sup>st</sup> edition, 1994.
- [GHM00] Etienne Gagnon, Laurie J. Hendren, and Guillaume Marceau. Efficient inference of static types for Java bytecode. In *SAS '00: Proceedings of the 7th International Symposium on Static Analysis*, pages 199–219, London, UK, 2000. Springer-Verlag.
- [Gla09] Malcolm Gladwell. *What the Dog Saw: And Other Adventures*. Little, Brown and Company, 2009.

- [GSF04] Yann-Gaël Guéhéneuc, Houari Sahraoui, and Farouk Zaidi. Fingerprinting design patterns. In Eleni Stroulia and Andrea de Lucia, editors, *11<sup>th</sup> Working Conference on Reverse Engineering (WCRE)*, pages 172–181. IEEE Computer Society Press, November 2004. 10 pages.
- [GSV02] David Grosser, Houari A. Sahraoui, and Petko Valtchev. Predicting software stability using case-based reasoning. In Wolfgang Emmerich and David Wile, editors, *17<sup>th</sup> International Conference on Automated Software Engineering (ASE)*, pages 295–298, Los Alamitos, CA, USA, 2002. IEEE Computer Society.
- [GZ05] Michael W. Godfrey and Lijie Zou. Using origin analysis to detect merging and splitting of source code entities. *IEEE Transactions on Software Engineering*, 31(2):166–181, 2005.
- [Hal77] M. H. Halstead. *Elements of Software Science*. Elsevier, 1977.
- [HBVW08] Tracy Hall, Sarah Beecham, June Verner, and David Wilson. The impact of staff turnover on software projects: the importance of understanding what makes software practitioners tick. In *2008 ACM SIGMIS CPR conference on Computer personnel doctoral consortium and research*, pages 30–39, New York, NY, USA, 2008. ACM.
- [Hec95] David Heckerman. A Tutorial on Learning With Bayesian Networks. Technical Report MSR-TR-95-06, Microsoft Research, Redmond, WA, USA, March 1995.
- [HKAH96] Robert Hochman, Taghi M. Khoshgoftaar, Edward B. Allen, and John P. Hudepohl. Using the genetic algorithm to build optimal neural networks for fault-prone module detection. In Michael R. Lyu, editor, *International Symposium on Software Reliability Engineering (ISSRE)*, pages 152–162, White Plains, NY, oct 1996. IEEECS.



- [HM00] Dick Hamlet and Joe Maybee. *The Engineering of Software: A Technical Guide for the Individual*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.
- [HT99] Andrew Hunt and David Thomas. *The pragmatic programmer: from journeyman to master*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.
- [IBM08] IBM. IBM: z/VSE Operating System - History - 1960s. <http://www-03.ibm.com/systems/z/os/zvse/about/history1960s.html>, 2008.
- [ISO91] ISO/IEC. *Information Technology – Software Product Evaluation – Quality Characteristics and Guidelines for their Use*, December 1991. ISO/IEC 9126:1991(E).
- [JKC04] Ho-Won Jung, Seung-Gweon Kim, and Chang-Shin Chung. Measuring software product quality: A survey of ISO/IEC 9126. *IEEE Software*, 21(5):88–92, 2004.
- [JKYR06] Liu Jing, He Keqing, Ma Yutao, and Peng Rong. Scale free in software metrics. In Johnny Wong and Aditya Mathur, editors, *30<sup>th</sup> Annual International Computer Software and Applications Conference (COMPSAC)*, pages 229–235, Washington, DC, USA, 2006. IEEE Computer Society.
- [JRA97] David Alex Lamb Joe Raymond Abounader. A data model for object-oriented metrics. Technical Report 409, Department of Computing and Information Science, Queen’s University, October 1997.
- [KAB<sup>+</sup>96] T. M. Khoshgoftaar, E. B. Allen, L. A. Bullard, R. Halstead, and G. P. Trio. A tree-based classification model for analysis of a military software system. In *High-Assurance Systems Engineering Workshop (HASE)*, page 244, Washington, DC, USA, 1996. IEEE Computer Society.

- [KAHA96] Taghi M. Khoshgoftaar, Edward B. Allen, John P. Hudepohl, and Stephen J. Aud. Software metric-based neural network classification models of a very large telecommunications system. In Steven K. Rogers and Dennis W. Ruck, editors, *Applications and Science of Artificial Neural Networks II*, volume 2760, pages 634–645, Orlando, FL, 1996.
- [KAJH05] Taghi M. Khoshgoftaar, Edward B. Allen, Wendell D. Jones, and John P. Hudepohl. Accuracy of software quality models over multiple releases. *Annals of Software Engineering*, 9(1):103–116, May 2005.
- [Kan95] Stephen H. Kan. *Metrics and Models In Software Quality Engineering*. Addison Wesley, 1995.
- [KDG09] Foutse Khomh, Massimiliano Di Penta, and Yann-Gaël Guéhéneuc. An exploratory study of the impact of code smells on software change-proneness. In Andy Zaidman, Giuliano Antoniol, and Stéphane Ducasse, editors, *16<sup>th</sup> Working Conference on Reverse Engineering (WCRE)*, pages 75–84, Washington, DC, USA, 2009. IEEE Computer Society.
- [KDHS07] Vigdis By Kampenes, Tore Dybå, Jo Erskine Hannay, and Dag I. K Sjøberg. A systematic review of effect size in software engineering experiments. *Information and Software Technology*, 49(11-12):1073–1086, 2007.
- [KGA<sup>+</sup>97] Taghi M. Khoshgoftaar, K. Ganesan, Edward B. Allen, Fletcher D. Ross, Rama Munikoti, Nishith Goel, and Amit Nandi. Predicting fault-prone modules with case-based reasoning. In Bob Horgan and Yashwant K. Malaiya, editors, *8<sup>th</sup> International Symposium on Software Reliability Engineering (ISSRE)*, pages 27 – 35, Washington, DC, USA, 1997. IEEE Computer Society.
- [KGA09] Foutse Khomh, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. Playing roles in design patterns: An empirical descriptive and analytic study.

- In 25<sup>th</sup> *International Conference on Software Maintenance (ICSM)*, volume 0, pages 83–92, Los Alamitos, CA, USA, 2009. IEEE Computer Society.
- [KGNB02] Taghi M. Khoshgoftaar, Erik Geleyn, Laurent Nguyen, and Lofton Bullard. Cost-sensitive boosting in software quality modeling. *High-Assurance Systems Engineering, IEEE International Symposium on*, 0:51, 2002.
- [KGS01] Taghi M. Khoshgoftaar, Kehan Gao, and Robert M. Szabo. An application of zero-inflated poisson regression for software fault prediction. In *ISSRE '01: Proceedings of the 12th International Symposium on Software Reliability Engineering*, page 66, Washington, DC, USA, 2001. IEEE Computer Society.
- [KL83] Joseph B. Kruskal and Mark Liberman. The symmetric time-warping problem: from continuous to discrete. In David Sankoff and Joseph B. Kruskal, editors, *Time Warps String Edits and Macromolecules: The Theory and Practice of Sequence Comparison*. Addison-Wesley, 1983.
- [KL05] A. Güneş Koru and Hongfang Liu. An investigation of the effect of module size on defect prediction using static measures. In *Workshop on Predictor models in software engineering (PROMISE)*, pages 1–5, New York, NY, USA, 2005. ACM.
- [KL07] A. Güneş Koru and Hongfang Liu. Identifying and characterizing change-prone classes in two large-scale open-source products. *Journal of Systems and Software*, 80(1):63–73, 2007.
- [KM90] Taghi M. Khoshgoftaar and John C. Munson. Predicting software development errors using software complexity metrics. *IEEE Journal on Selected Areas in Communications*, 8(2):253–261, 1990.

- [KMBR92] Taghi M. Khoshgoftaar, John C. Munson, Bibhuti B. Bhattacharya, and Gary D. Richardson. Predictive modeling techniques of software quality from software measures. *IEEE Transactions on Software Engineering*, 18(11):979–987, 1992.
- [KP96] Barbara Kitchenham and Shari Lawrence Pfleeger. Software quality: The elusive target. *IEEE Software*, 13:12–21, 1996.
- [KPB06] Patrick Knab, Martin Pinzger, and Abraham Bernstein. Predicting defect densities in source code files with decision tree learners. In *MSR '06: Proceedings of the 2006 international workshop on Mining software repositories*, pages 119–125, New York, NY, USA, 2006. ACM Press.
- [KT05] A. Güneş Koru and Jeff Tian. Comparing high-change modules and modules with the highest measurement values in two large-scale open-source products. *IEEE Transactions of Software Engineering*, 31(8):625–642, 2005.
- [KVGS09] Foutse Khomh, Stéphane Vaucher, Yann-Gaë Guéhéneuc, and Houari Sahraoui. A Bayesian Approach for the Detection of Code and Design Smells. In Doo-Hwan Bae and Byoungju Choi, editors, *Proceedings of the 9<sup>th</sup> International Conference on Quality Software*. IEEE Computer Society Press, August 2009.
- [KVGS10] Foutse Khomh, Stéphane Vaucher, Yann-Gaël Guéhéneuc, and Houari Sahraoui. BDTEX: A GQM-based Bayesian approach for the detection of antipatterns. *Journal of Systems and Software*, 2010. Accepted with revisions.
- [KVS10] Marouane Kessentini, Stéphane Vaucher, and Houari Sahraoui. Deviance from perfection is a better criterion than closeness to evil when identifying risky code. In Elisabetta Di Nitto and Jamie Andrews, editors, *25<sup>th</sup>*

*IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE/ACM, 2010. [to appear].

- [KY95] George J. Klir and Bo Yuan. *Fuzzy sets and fuzzy logic: theory and applications*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1995.
- [KZEL09] A. Güneş Koru, Dongsong Zhang, Khaled El Emam, and Hongfang Liu. An investigation into the functional form of the size-defect relationship for software modules. *IEEE Transactions on Software Engineering*, 35(2):293–304, 2009.
- [LBMP08] Stefan Lessmann, Bart Baesens, Christophe Mues, and Swantje Pietsch. Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *IEEE Transactions on Software Engineering*, 34:485–496, 2008.
- [LH93a] W. Li and S. Henry. Maintenance metrics for the object oriented paradigm. In *1<sup>st</sup> International Software Metrics Symposium (METRICS)*, pages 52–60. IEEE, may 1993.
- [LH93b] Wei Li and Sallie Henry. Object-oriented metrics that predict maintainability. *Journal of Systems and Software*, 23(2):111–122, 1993.
- [LK03] Jim Lawler and Barbara Kitchenham. Measurement modeling technology. *IEEE Software*, 20(3):68–75, 2003.
- [LLL08] Rüdiger Lincke, Jonas Lundberg, and Welf Löwe. Comparing software metrics tools. In Barbara G. Ryder and Andreas Zeller, editors, *International Symposium on Software Testing and Analysis (ISSTA)*, pages 131–142, New York, NY, USA, 2008. ACM.
- [LSP08] Guillaume Langelier, Houari A. Sahraoui, and Pierre Poulin. Exploring the evolution of software quality with animated visualization. In Paolo Bottoni and Mary Beth Rosson, editors, *Symposium on Visual Languages and Human-Centric Computing (VL-HCC)*, pages 13–20, 2008.

- [LSV08] Panagiotis Louridas, Diomidis Spinellis, and Vasileios Vlachos. Power laws in software. *ACM Transactions on Software Engineering and Methodology*, 18(1):1–26, September 2008. Article 2.
- [MA07] S. Mavromoustakos and A. S. Andreou. WAQE: a Web Application Quality Evaluation Model. *International Journal of Web Engineering Technology*, 3(1):96–120, 2007.
- [Mar04] Radu Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. In Mark Harman and Bogdan Korel, editors, *20<sup>th</sup> IEEE International Conference on Software Maintenance (ICSM)*, pages 350–359, Los Alamitos, CA, USA, 2004. IEEE Computer Society.
- [MB09] Harish Mittal and Pradeep Bhatia. Software maintainability assessment based on fuzzy logic technique. *SIGSOFT Software Engineering Notes*, 34(3):1–5, 2009.
- [McC76] McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2:308–320, 1976.
- [McC04] Steve McConnell. *Code Complete, Second Edition*. Microsoft Press, Redmond, WA, USA, 2004.
- [MDL87] H.D. Mills, M. Dyer, and R.C. Linger. Cleanroom software engineering. *IEEE Software*, 4(5):19 – 25, 1987.
- [Men08] Tim Menzies. Learning defect predictors: Lessons from the trenches. <http://www.youtube.com/watch?v=vrvRsZsoMp8&url=http://menzies.us/>, September 2008.
- [MGDM10] Naouel Moha, Yann-Gaël Guéhéneuc, Laurence Duchien, and Anne-Françoise Le Meur. DECOR: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering*, 36(1):20–36, 2010.

- [MGF07] Tim Menzies, Jeremy Greenwald, and Art Frank. Data mining static code attributes to learn defect predictors. *IEEE Transactions on Software Engineering*, 33(1):2–13, 2007.
- [MK92] John C. Munson and Taghi M. Khoshgoftaar. The detection of fault-prone programs. *IEEE Transactions on Software Engineering*, 18(5):423–433, 1992.
- [ML08] Mika V. Mantyla and Casper Lassenius. What types of defects are really discovered in code reviews? *IEEE Transactions on Software Engineering*, 99(RapidPosts):430–448, 2008.
- [Mos09] John Moses. Should we try to measure software quality attributes directly? *Software Quality Control*, 17(2):203–213, 2009.
- [MPF08] Andrian Marcus, Denys Poshyvanyk, and Rudolf Ferenc. Using the conceptual cohesion of classes for fault prediction in object-oriented systems. *IEEE Transactions on Software Engineering*, 34(2):287–300, 2008.
- [MPS08] Raimund Moser, Witold Pedrycz, and Giancarlo Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In Matthew Dwyer and Volker Gruhn, editors, *30<sup>th</sup> International Conference on Software Engineering (ICSE)*, pages 181 – 190, may 2008.
- [MRW77] J. A. McCall, P. K. Richards, and G. F. Walters. Factors in software quality. Technical Report RADC-TR-77-369, Rome Laboratory, Griffis AFB, NY, 13441, 1977.
- [MSBB06] Ghazwa Malak, Houari A. Sahraoui, Linda Badri, and Mourad Badri. Modeling Web-Based Applications Quality: A Probabilistic Approach. In *International Conference on Web Information Systems Engineering*, volume 4255, pages 398–404, 2006.

- [MSBB10] Ghazwa Malak, Houari Sahraoui, Linda Badri, and Mourad Badri. Modeling web quality using a probabilistic approach: An empirical validation. *ACM Transactions on the Web (TWEB)*, 4(3):1–31, 2010.
- [MSL98] Y. Mao, H. Sahraoui, and H. Lounis. Reusability hypothesis verification using machine learning techniques: A case study. In David F. Redmiles and Bashar Nuseibeh, editors, *13<sup>th</sup> International Conference on Automated Software Engineering (ASE)*, page 84, Washington, DC, USA, 1998. IEEE Computer Society.
- [MV00] Audris Mockus and Lawrence G. Votta. Identifying reasons for software changes using historic databases. In *16<sup>th</sup> International Conference on Software Maintenance (ICSM)*, pages 120–130. IEEE Computer Society, 2000.
- [Mye03] Christopher R. Myers. Software systems as complex networks: Structure, function, and evolvability of software collaboration graphs. *Physical Review E*, 68(4):046116, Oct 2003.
- [MYH<sup>+</sup>09] Tatsuya Miyake, Yoshiki, Higo, Shinji Kusumoto, and Katsuro Inoue. MASU: A Metrics Measurement Framework for Multiple Programming Languages. *IEICE Transactions on Information and Systems*, J92-D(9):1518–1531, 2009.
- [NB05a] Nachiappan Nagappan and Thomas Ball. Static analysis tools as early indicators of pre-release defect density. In Gruia-Catalin Roman, William G. Griswold, and Bashar Nuseibeh, editors, *Proceedings of the 27th International Conference on Software Engineering*, pages 580–586, New York, NY, USA, 2005. ACM.
- [NB05b] Nachiappan Nagappan and Thomas Ball. Use of relative code churn measures to predict system defect density. In Gruia-Catalin Roman,



- William G. Griswold, and Bashar Nuseibeh, editors, *27<sup>th</sup> International Conference on Software Engineering (ICSE)*, pages 284–292, 2005.
- [NBZ06] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. Mining metrics to predict component failures. In Leon J. Osterweil, H. Dieter Rombach, and Mary Lou Soffa, editors, *28<sup>th</sup> International Conference on Software Engineering*, pages 452–461, New York, NY, USA, 2006. ACM.
- [NL00] Mark W. Newman and James A. Landay. Sitemaps, storyboards, and specifications: a sketch of web site design practice. In *3<sup>rd</sup> conf. on Designing interactive systems*, pages 263–274, 2000.
- [NR69] P. Naur and B. Randell, editors. *Software Engineering: Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 Oct. 1968, Brussels*. Scientific Affairs Division, NATO, 1969.
- [NWO<sup>+</sup>05] Nachiappan Nagappan, Laurie Williams, Jason Osborne, Mladen Vouk, and Pekka Abrahamsson. Providing test quality feedback using static source code and automatic test suite metrics. In W. Eric Wong and Mark E. Segal, editors, *16<sup>th</sup> IEEE International Symposium on Software Reliability Engineering*, pages 85–94, Washington, DC, USA, 2005. IEEE Computer Society.
- [OA96] Niclas Ohlsson and Hans Alberg. Predicting fault-prone software modules in telephone switches. *IEEE Transactions on Software Engineering*, 22(12):886–894, 1996.
- [OLR01] Luis Olsina, Guillermo Lafuente, and Gustavo Rossi. Specifying quality characteristics and attributes for websites. In San Murugesan and Yogesh Deshpande, editors, *Web Engineering*, volume 2016 of *Lecture Notes in Computer Science*, pages 266–278. Springer Berlin / Heidelberg, 2001.
- [oST02] National Institute of Standards & Technology. The economic impacts of

- inadequate infrastructure for software testing. Technical Report 02-3., US Department of Commerce, 2002.
- [OW02] Thomas J. Ostrand and Elaine J. Weyuker. The distribution of faults in a large industrial software system. In Phyllis G. Frankl, editor, *International Symposium on Software Testing and Analysis (ISSTA)*, pages 55–64, New York, NY, USA, 2002. ACM.
- [OWB04] Thomas J. Ostrand, Elaine J. Weyuker, and Robert M. Bell. Where the bugs are. In Gregg Rothermel, editor, *International Symposium on Software Testing and Analysis (ISSTA)*, pages 86–96, New York, NY, USA, 2004. ACM Press.
- [Pal02] Jonathan W. Palmer. Web site usability, design, and performance metrics. *Info. Sys. Research*, 13(2):151–167, 2002.
- [Par94] David Lorge Parnas. Software aging. In *16<sup>th</sup> International Conference on Software Engineering (ICSE)*, pages 279–287, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.
- [PBMW99] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Tech. Report 1999-66, Stanford InfoLab, 1999.
- [Pea88] Judea Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, 1 edition, September 1988.
- [Pea00] Judea Pearl. *Causality: models, reasoning, and inference*. Cambridge University Press, New York, NY, USA, 2000.
- [Pfl01] Shari Lawrence Pfleeger. *Software Engineering: Theory and Practice*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.
- [PFP94] Shari Lawrence Pfleeger, Norman Fenton, and Stella Page. Evaluating software engineering standards. *Computer*, 27(9):71–79, 1994.

- [PS90] A.A. Porter and R.W. Selby. Empirically guided software development using metric-based classification trees. *IEEE Software*, 7(2):46 – 54, mar 1990.
- [PSR05] Parag C. Pendharkar, Girish H. Subramanian, and James A. Rodger. A probabilistic model for predicting software development effort. *IEEE Transactions on Software Engineering*, 31(7):615–624, 2005.
- [Ric94] John A Rice. *Mathematical Statistics and Data Analysis, Second Edition*. Duxbury Press, 1994. ISBN 0-534-20934-3.
- [Rie96] Arthur J. Riel. *Object-Oriented Design Heuristics*. Addison-Wesley, 1996.
- [Roy70] Wiliam W. Royce. Managing the development of large software systems. In *Proceedings of the IEEE Wescon*, pages 1–9, Washington, DC, USA, 1970. IEEE Computer Society.
- [RT01] Filippo Ricca and Paolo Tonella. Analysis and testing of web applications. *International Conference on Software Engineering*, 0:0025, 2001.
- [SBB<sup>+</sup>02] Forrest Shull, Vic Basili, Barry Boehm, Winsor A. Brown, Patricia Costa, Mikael Lindvall, Dan Port, Ioana Rus, Roseanne Tesoriero, and Marvin Zelkowitz. What we have learned about fighting defects. In *8<sup>th</sup> International Symposium on Software Metrics (METRICS)*, page 249, Washington, DC, USA, 2002. IEEE Computer Society.
- [SBC<sup>+</sup>02] Houari A. Sahraoui, Mounir A. Boukadoum, Hassan M. Chawiche, Gang Mai, and Mohamed Serhani. A fuzzy logic framework to improve the performance and interpretation of rule-based quality prediction models for OO software. In *Proc. of the 26th Int. Computer Software and Applications Conf.*, pages 131–138, 2002.

- [Sch03a] Petra Schubert. Extended Web Assessment Method (EWAM): Evaluation of Electronic Commerce Applications from the Customer's Viewpoint. *International Journal of Electronic Commerce*, 7(2):51–80, 2003.
- [Sch03b] Barry Schwartz. *The Paradox of Choice: Why More Is Less*. Ecco, 2003.
- [SCK02] Sun Sup So, Sung Deok Cha, and Yong Rae Kwon. Empirical evaluation of a fuzzy logic-based software quality prediction model. *Fuzzy Sets Syst.*, 127(2):199–208, 2002.
- [SEG68] H. Sackman, W. J. Erikson, and E. E. Grant. Exploratory experimental studies comparing online and offline programming performance. *Communications of the ACM*, 11(1):3–11, 1968.
- [SGM00] Houari A. Sahraoui, Robert Godin, and Thierry Miceli. Can metrics help to bridge the gap between the improvement of oo design quality and its automation? In Lionel Briand and Jeffrey M. Voas, editors, *16<sup>th</sup> International Conference on Software Maintenance (ICSM)*, pages 154–162. IEEE Computer Society, 2000.
- [SK03] Ramanath Subramanyam and M. S. Krishnan. Empirical analysis of CK metrics for object-oriented design complexity: Implications for software defects. *IEEE Transactions on Software Engineering*, 29(4):297–310, 2003.
- [Spi06] Diomidis Spinellis. *Code Quality: The Open Source Perspective (Effective Software Development Series)*. Addison-Wesley Professional, 2006.
- [Tra88] E. Trauwaert. On the meaning of Dunn's partition coefficient for fuzzy clusters. *Fuzzy Sets Syst.*, 25(2):217–242, 1988.
- [VBSH09] Stéphane Vaucher, Samuel Boclinville, Houari Sahraoui, and Naji Habra. Recommending improvements to web applications using quality-driven heuristic search. In Darrell Long, Gottfried Vossen, and Jeffrey Xu Yu,

- editors, *10<sup>th</sup> International Conference On Web Information Systems Engineering*, pages 321–334. Springer, 2009.
- [VKMG09] Stéphane Vaucher, Foutse Khomh, Naouel Moha, and Yann-Gaël Guéhéneuc. Prevention and cure of software defects: Lessons from the study of god classes. In Giuliano Antoniol and Andy Zaidman, editors, *16<sup>th</sup> Working Conference on Reverse Engineering (WCRE)*, pages 145–154. IEEE Computer Society Press, 2009.
- [VS07] Stéphane Vaucher and Houari Sahraoui. Do software libraries evolve differently than applications?: an empirical investigation. In *Symposium on Library-Centric Software Design (LCSD)*, pages 88–96, New York, NY, USA, 2007. ACM.
- [VS08] Stéphane Vaucher and Houari Sahraoui. Étude de la changeabilité des systèmes orientés-objet. In *Actes des journées Langages et Modèles à Objets*. Éditions Cepaduès, 2008.
- [VS10] Stéphane Vaucher and Houari Sahraoui. Multi-level evaluation of web sites. In *12<sup>th</sup> IEEE International Symposium on Web System Evolution (WSE)*. IEEE Computer Society, 2010. [to appear].
- [VSV08] Stéphane Vaucher, Houari Sahraoui, and Jean Vaucher. Discovering New Change Patterns in Object-Oriented Systems. In Andy Zaidman, Massimiliano Di Penta, and Ahmed Hassan, editors, *15<sup>th</sup> Working Conference on Reverse Engineering (WCRE)*, pages 37–41, October 2008.
- [Wes05] Linda Westfall. 12 steps to useful software metrics. [http://www.westfallteam.com/12\\_steps\\_paper.pdf](http://www.westfallteam.com/12_steps_paper.pdf), 2005.
- [WF05] Ian H. Witten and Eibe Frank. *Data Mining: Practical Machine Learning Tools and Techniques, Second Edition (Morgan Kaufmann Series in Data Management Systems)*. Morgan Kaufmann, June 2005.

- [WH98] FG Wilkie and B. Hylands. Measuring complexity in C++ application software. *Software: Practice and Experience*, 28(5):513–546, 1998.
- [Wol74] R.W. Wolverton. The cost of developing large-scale software. *Computers, IEEE Transactions on*, C-23(0018-9450), 1974.
- [WRH<sup>+</sup>00] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in software engineering: an introduction*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.
- [WS03] Scott White and Padhraic Smyth. Algorithms for estimating relative importance in networks. In *9<sup>th</sup> International Conference on Knowledge Discovery and Data Mining*, pages 266–275, New York, NY, USA, 2003. ACM.
- [XS04] Zhenchang Xing and Eleni Stroulia. Understanding class evolution in object-oriented software. In *12<sup>th</sup> International Workshop on Program Comprehension (IWPC)*, page 34, Los Alamitos, CA, USA, 2004. IEEE Computer Society.
- [XS05] Zhenchang Xing and Eleni Stroulia. Analyzing the evolutionary history of the logical design of object-oriented software. *IEEE Transactions on Software Engineering*, 31(10):850–868, 2005.
- [YC79] Edward Yourdon and Larry L. Constantine. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1979.
- [Zhe10] Jun Zheng. Cost-sensitive boosting neural networks for software defect prediction. *Expert Systems with Applications*, 37(6):4537 – 4543, 2010.
- [ZK10] Hongyu Zhang and Sunghun Kim. Monitoring software quality evolution for defects. *IEEE Software*, 27(4):58–64, 2010.

- [ZLW07] Yuming Zhou, Hareton Leung, and Pinata Winoto. Mnav: A markov model-based web site navigability measure. *IEEE Transactions on Software Engineering*, 33(12):869–890, 2007.
- [ZN08] Thomas Zimmermann and Nachiappan Nagappan. Predicting defects using network analysis on dependency graphs. In Matthew Dwyer and Volker Gruhn, editors, *30<sup>th</sup> International Conference on Software Engineering (ICSE)*, pages 531–540, New York, NY, USA, 2008. ACM.
- [ZNG<sup>+</sup>09] Thomas Zimmermann, Nachiappan Nagappan, Harald Gall, Emanuel Giger, and Brendan Murphy. Cross-project defect prediction. In Hans van Vliet and Valérie Issarny, editors, *7<sup>th</sup> joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC-FSE)*. ACM, August 2009.
- [ZPZ07] Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. Predicting defects for Eclipse. In *3<sup>rd</sup> International Workshop on Predictor Models in Software Engineering (PROMISE)*, May 2007.
- [Zus97] Horst Zuse. *A Framework of Software Measurement*. Walter de Gruyter & Co., Hawthorne, NJ, USA, 1997.
- [ZWDZ04] Thomas Zimmermann, Peter Weisgerber, Stephan Diehl, and Andreas Zeller. Mining version histories to guide software changes. In Jacky Estublier and David S. Rosenblum, editors, *26<sup>th</sup> International Conference on Software Engineering (ICSE)*, pages 563–572. IEEE Computer Society, 2004.
- [ZZG04] Yanlong Zhang, Hong Zhu, and Sue Greenwood. Website complexity metrics for measuring navigability. In Hans-Dieter Ehrich and Klaus-Dieter Schewe, editors, *4<sup>th</sup> International Conference Quality Software (QSIC)*, pages 172–179, Washington, DC, USA, 2004. IEEE Computer Society.





## Appendix I

### Reformulating Class-level Metrics

In this appendix, we show how metrics from the CK metric suite [CK91] can be formulated using lower-level entities. In fact, the class-level metric value is obtained by using a naive combination strategy.

#### Coupling Between Object (CBO)

In [CK91], Chidamber and Kemerer defined the CBO of a class as the number of other classes to which it is coupled. A class  $c$  is *coupled* to a class  $d$  if  $c$  uses  $d$  or  $d$  uses  $c$ . A class  $c$  uses  $d$  if one of its methods invokes a method or accesses a field in  $d$ . Briand *et al.* formally defined CBO as follows:

$$CBO(c) = |d \in C - \{c\} | uses(c, d) \vee uses(d, c) |$$

where  $C$  is a set of classes and  $c \in C$ , and

$$uses(c, d) = (\exists m \in M_I(c) : \exists m' \in M_I(d) : m' \in PIM(m))$$

$$\vee (\exists m \in M_I(c) : \exists a \in A_I(d) : a \in AR(m))$$

where  $m$  and  $m'$  are methods,  $M_I(c)$  is the set of implemented methods in class  $c$ ,  $PIM(m)$  is the set of polymorphically invoked methods of  $m$ ,  $A_I(c)$  is the set of implemented attributes in class  $c$ , and  $AR(m)$  is the set of referenced attributes in the method  $m$ . An attribute  $a$  is in  $AR(m)$  if  $a$  is read or written in the body of the method  $m$ .

From our perspective, CBO is an aggregate measure that combines the coupling of methods and attributes. We can reformulate this metrics by stating that CBO measures the size of the set defined as by the union of *method\_couplings* and *attribute\_couplings*. This can be expressed formally as:

$$CBO(c) = |method\_coupling(c) \cup attribute\_coupling(c)|$$

$$method\_coupling(c) = \cup_{m \in c} coupling(m)$$

$$attribute\_coupling(c) = \cup_{a \in c} coupling(a)$$

$$coupling(m) = d \in C | \exists m_d \in d : uses(m, m_d) \vee uses(m_d, m)$$

$$coupling(a) = d \in C | \exists m_d \in d : reads/writes(m_d, a)$$

It is important to note that when using coupling measured at the level of a class, we lose important information as to which part of a class is responsible for this coupling. Therefore, we believe that measuring coupling at the method-level is more meaningful than at the class-level for our problem of identifying and locating high-change code.

### Response For Class (RFC)

Chidamber and Kemerer defined RFC for a given class as the number of unique methods that can be invoked in response to a message to an object of that class. Briand *et al.* formalised this definition as follows:

$RFC(c) = |M(c) \cup_{m \in M(c)} PIM(m)|$  where  $M(c)$  is the set of all methods in class  $c$ , and  $PIM(m)$  is the set of polymorphically invoked methods of  $m$ .

Yet again, the RFC value of a class can be expressed as the union of the response set (RS) of its methods.

$$RFC(c) = |\cup_{m \in c} RS(m)|$$

$$RS(m) = m \cup PIM(m)$$

As with CBO, we believe RFC should be measured at the method-level if we are interested in high-change code.

## Weighted Method Complexity(WMC)

WMC was directly defined by Chidamber and Kemerer as the sum of the complexities of its methods:

$$WMC(c) = \sum_{m \in c} complexity(m)$$

where complexity is left undefined, but has been implemented as unity or by using a method-level metric like cyclomatic complexity. Since the metric is directly defined for methods, we believe it should be used in a method-level model.

## Discussion

By deconstructing these metrics, we wanted to show how they could be expressed using lower-level models evaluating the same characteristics, and an aggregation model to combine individual evaluations. We believe that a better understanding of how to combine these method-level metrics can add flexibility to class-level analyses.

First, there are advantages in perfecting a method-level model. The most important advantage is that the model could be built using historical data to reflect a customised definition of method quality for a development team. This model could hopefully use combinations of metrics to identify the semantics of methods. There are cases when the standard version of the metrics explored produce counter-intuitive results. Let us consider a class defining no methods and a public attribute. WMC would consider this class as less complex than a clean, refactored version of the same class that has its attribute encapsulated using accessor methods. A well-trained lower-level model might identify the method as an accessor that should not affect the quality of the class.