Tiago Miguel Laureano Alves **Benchmark-based Software Product Quality Evaluation**

UMinho | 2011

Universidade do Minho
Escola de Engenharia

Tiago Miguel Laureano Alves

Benchmark-based Software Product Quality Evaluation

Novembro de 2011

Universidade do Minho
Escola de Engenharia

Tiago Miguel Laureano Alves

Benchmark-based Software Product Quality Evaluation

Tese de Doutoramento
Doutoramento em Informática

Trabalho efectuado sob a orientação do
Professor Doutor José Nuno Oliveira
e do
Doutor Joost Visser

Novembro de 2011

Acknowledgments

These past four years of PhD were a truly fantastic experience, both professionally and personally. Not only I was granted the opportunity of doing research in exclusivity but also I received support by supervisors, organizations, colleagues, friends and family. It was due to their support that I have made it, and this means the world to me. Hence, I would like to thank all those who supported me and it is to them I dedicate this work.

First, I want to start thanking my supervisors: Joost Visser and José Nuno Oliveira. It was their encouragement and enthusiasm that lead me to, after a 5-year graduation, engage in MSc and afterwards in PhD studies. They were tireless providing directions and comments, while supporting my decisions and helping me to achieve my goals. I have to additionally thank Joost, my day-to-day supervisor, with whom I enjoyed working and learned a lot both professionally and personally.

I have to thank Fundação para a Ciência e a Tecnologia (FCT) for sponsoring me with grant SFRH/BD/30215/2006. This sponsorship was essential, supporting all my living costs in The Netherlands and my research costs. This work was further supported by the SSaaPP project, FCT contract no. PTDC/EIA-CCO/108613/2008.

I have to thank the Software Improvement Group (SIG) which hosted me during all my PhD. SIG provided me a work environment of excellency, where I was able to share ideas and cooperate with researchers, consultants, technical consultants (Lab) and sales. Non only SIG highly-educated professionals have challenged me every time, helping me to look at the problems from other angles, leading to a continuous improvement of my work, but also some of them directly contributed to my work. It was also while at SIG I had the chance to learn and practice consultancy and sales.

Thanks to Tobias Kuipers for accepting me at the company.

Thanks to the research team: José Pedro Correira, Xander Schrijen, Eric Bouwers, Dimitrios Athanasiou and Ariadi Nugroho. Together, we have done and discussed research, read and commented each other's papers, helped preparing each other's presentations, but also end up becoming friends, having daily lunch, socializing, traveling and partying together - and all that was challenging, inspiring and fun!

Thanks to Brigitte van der Vliet, Paula van den Heuvel, Femke van Velthoven, Mendy de Jonge and Sarina Petri for their support with all the administrative things - although their work does sometimes not show, it is extremely important.

Thanks to the Lab for all the technical support: Patrick Duin, Rob van der Leek, Cora Janse, Eric Bouwers, Reinier Vis, Wander Grevink, Kay Grosskop, Dennis Bijlma, Jeroen Heijmans, Marica Ljesar and Per John. I would like to additionally thank Patrick and Rob for always being the first providing me support about SIG tooling which they master, but also with whom I had the chance to spend many great social moments, and Eric and Cora for always making my day with their good mood.

Thanks to all consultants: Michel Kroon, Harro Stokman, Pieter Jan't Hoen, Ilja Heitlager, Cathal Boogerd, Yiannis Kanellopoulos, Magiel Bruntink, Sieuwert van Otterloo, Hans van Oosten, Marijn Dessens, Mark Hissink Muller, Sven Hazejager, Joost Schalken, Bas Cornelissen, Zeeger Lubsen and Christiaan Ypma. I have to thank Michel for mentoring me in the industrial projects and for his clinical sharpness and ironic sense that I admired. Thanks Harro for the hints on statistics and his stories (which I miss the most). Thanks Yiannis and Magiel for taking me on board in consultancy, to Sieuwert for the nasty questions (which allowed me to be better prepared) and to Hans, Marijn and Mark for care and support in my presentations.

Thanks the sales team, Jan Willem Klerkx, Bart Pruijn, Eddy Boorsma, Dennis Beets and Jan Hein, which also attended and asked questions in my presentations. In particular thanks to Jan Willem, from which I learned that sales is mostly a process, although following the right steps not always allow us to achieve our goals, and with whom I shared and enjoyed the taste for wine and great talks about classic cars.

Last, but far way from least, the SIG System Administrators: Leo Makkinje and Gerard Kok. Not that Mac computers have problems, of course not, but they made sure that everything was running smoothly, most of the times stopping whatever they were doing to help me and allow me to continue my work. Endless times I walked in their office for a small break and left with improved mood and strength to continue whatever I was doing.

I have to thank Dutch Space, in particular to Coen Claassens, Leon Bremer and Jose Fernandez-Alcon, for their support on the quality analysis of the EuroSim Simulator which lead to a published paper and later to one of this dissertation chapters. It was a great experience to work with them, both from the valuable feedback I got and the opportunity to learn from each other.

I have to thank the European Space Agency (ESA) for granting me a license to use their software for research, their support on their software, for teaching me their software engineering concerns and the important things you should not do. From the European Space Operations Center (ESOC) thanks to Nestor Peccia, Colin Haddow, Yves Doat, James Eggleston, Mauro Pecchioli, Eduardo Gomez and Angelika Slade, from the European Space Research and Technology Centre (ESTEC) thanks to Stefano Fiorilli and Quirien Wijnands and, from the European Space Astronomy Centre (ESAC) thanks to Serge Moulin.

I have to thank Paul Klint, Jurgen Vinju and Bas Basten from CWI, Arie van Deursen and Eelco Visser from University of Delft and Jurriaan Hage from University of Utrecht. Several times they invited me to give guest lectures and presentations allowing me the chance of discussing and improving my work. I would like to additionally thank Jurriaan for his work and co-authorship in the Code Querying paper. Thanks to Leon Moonen from Simula and Ira Baxter from Semantic Designs for their support in many conferences where we would regularly meet. In particular thanks to Ira which was one of the first to see my PhD proposal and with whom I had the chance to share my progress and learn from his wise and practical words and jokes.

I have to thank my friends, who have been with me through this journey and for the great time we spent together in Amsterdam: Bas Basten, Patrick Duin (and Anna Sanecka), Rob van der Leek, Xander Schrijven (and Nienke Zwennes), Leo Makkinje (and Yvonne Makkinje), Joost Visser (and Dina Ruano), Stephanie Kemper, José Pedro Correia, José Proença, Mário Duarte, Alexandra Silva, Levi Pires (and Stefanie Smit) and Ivo Rodrigues. Thanks to all those in Portugal which kept being great friends even with whom I only had a chance to meet a couple of times a year: Ricardo Castro, Arine Malheiro, Paulo Silva, Raquel Santos, Ana Cláudia Norte, Filipa Magalhães and Valente family. Thanks to all those I had the chance to meet around the world, in particular to Önder Gürcan, Violetta Kuvaeva and Jelena Vlasenko. Thanks to Andreia Esteves for being such important part of my life, support and care.

Finally, I want to thank my family for being an extremely important pillar of my life. Although I have spent most of my PhD away from them, sometimes missing important celebrations and moments, I have tried to be always present by phone, mail, or Skype. At all times, they supported me, cheered me up and encouraged me to go further and dream higher. Thanks to my grandmother Laida, which I miss dearly, that did not lived enough to see me become PhD. To my grandparents Dora and Teotónio which countless of times told me how much they missed me, but always respected what I wanted and no matter what supported me on that. Thanks to the Almeida and Moutinho families for this great family we have and for their support. Thanks to my aunt Lili that despite being in the background was always a front supporter, being there for me. To my sisters Ana, Sara and Paula Alves, which always made sure to remind me who I am. To my mom and dad that are the best persons I ever had the chance to met – they have done for me more than I am able to count or express with words (or pay back making sure the computers work). I love you and thank you for being there.

Benchmark-based Software Product Quality Evaluation

Two main problems have been hindering the adoption of source code metrics for quality evaluation in industry: *(i)* the difficulty in doing a qualitative interpretation of measurements; and *(ii)* the inability of summarizing measurements into a single meaningful value that captures quality at the level of overall system.

This dissertation proposes an approach based on two methods to solve these problems using thresholds derived from an industrial benchmark.

The first method categorizes measurements into different risk areas using *risk thresholds*. These thresholds are derived by aggregating different metric distributions while preserving their statistical properties.

The second method enables the assignment of ratings to systems, for a given scale, using *rating thresholds*. These thresholds are calibrated such that it is possible to distinguish systems based on their metric distribution. For each rating, these thresholds set the maximum amount of code that is allowed in all risk categories.

Empirical and industrial studies provide evidence of the usefulness of the approach. The empirical study shows that ratings for a new test adequacy metric can be used to predict bug solving efficiency. The industrial case details the quality analysis and evaluation of two space-domain simulators.

Avaliação da Qualidade de Produto de Software baseada em Benchmarks

A adoção na indústria do uso de métricas de código fonte para a avaliação de qualidade tem sido dificultada por dois problemas: *(i)* pela dificuldade em interpretar métricas de forma qualitativa; e *(ii)* pela impossibilidade de agregar métricas num valor único que capture de forma fiel a qualidade do sistema como um todo.

Esta dissertação propõe uma solução para estes problemas utilizando dois métodos que usam valores-limite derivados de um benchmark industrial.

O primeiro método caracteriza medições em diferentes áreas de risco através de *valores-limite de risco*. Estes valores-limite são derivados através da agregação das distribuições de métricas preservando as suas propriedades estatísticas.

O segundo método, dada uma escala, permite atribuir uma classificação a sistemas de software, usando *valores-limite de classificação*. Estes valores-limite são calibrados para permitir diferenciar sistemas baseada na distribuição de métricas definindo, para cada classificação, a quantidade máxima de código permissível nas categorias de risco.

Dois estudos evidenciam os resultados desta abordagem. No estudo empírico mostra-se que as classificações atribuídas para uma nova métrica de teste podem ser usadas para prever a eficiência na resolução de erros. No estudo industrial detalha-se a avaliação e análise de qualidade de dois simuladores usados para missões no espaço.

Contents

List of Figures

List of Tables

Acronyms

ESA	European Space Agency
ECSS	European Cooperation for Space Standardization
FCT	Fundação para a Ciência e a Tecnologia
GUI	Graphical User Interface
IEC	International Electrotechnical Commission
ISBSG	International Software Benchmarking Standards Group
ISO	International Organization for Standardization
ITS	Issue Tracking System
MI	Maintainability Index
OO	Object-Oriented
OSS	Open-Source Software
P2P	Peer-to-Peer
SAT	Software Analysis Toolkit
SETC	Static Estimation of Test Coverage
SIG	Software Improvement Group

SLOC	Source Lines of Code
TÜViT	Technischer Überwachungs-Verein Informationstechnik – German Technical Inspection Association, Information Technology

“Count what is countable, measure what is measurable, and what is not measurable, make measurable.” *Galileo Galilei*

“When you can measure what you are speaking about, and express it in numbers, you know something about it.” *Lord Kelvin*

“The wonder is, not that the field of the stars is so vast, but that man has measured it.” *Anatole France*

“Not everything that counts can be counted, and not everything that can be counted counts.” *Albert Einstein*

Chapter 1

Introduction

Software has grown in importance to a level that has become ubiquitous, supporting our society. With such role, it is critical that software is regarded as a product, where its lifecycle (planning, development and maintenance) is supported by rigorous process with tools and methodologies. An important activity in this lifecycle is the software quality assurance which is responsible to make all process manageable and predictable.

Due to the ubiquity of software, software quality assurance, which was used to be seen as an expensive and secondary activity, it is now being seen as a primary need. This increase of importance poses a stronger demand for better methodologies and techniques to enable more proficiently software quality assurance, hence creating plenty opportunities for innovation (e.g. evaluating software quality).

Software quality evaluation is done through measurement. Although measurement already takes place in different software-engineering activities (e.g. risk control, improvement and estimation) the focus in this dissertation is on evaluation, i.e., to achieve an objective representation of quality.

Measuring quality allows one to objectively speak about it, to express it in numbers and to gain knowledge about it. This enables a well-founded decision-making process. Understanding and communicating about the overall quality, allows decision makers to rationalize it and decide, for instance, whether there is need for improvement.

Software quality assurance can act upon four complementary areas: process, project, people and product. For all these areas except product, software quality assurance is well defined and widely accepted in industry. Software product assurance, on the other hand, which is responsible for the *concrete* thing that is being developed or modified, is commonly achieved only by testing. This dissertation focus on the use of general measurements, derived from the software *product*, to provide a complementary quality view to support software quality assurance.

For software product quality measurement to be of relevance it should go beyond a theoretical exercise, be applied in practice and yield results. The fulfillment of these three requirements has been demanded by many practitioners before adopting any advance made in software quality research into practice [8]. This was the reason that the work presented in this PhD dissertation required to be done in an industrial environment, to be close to the problems that matter and to have a better feedback about what actually works in practice.

The Software Improvement Group (SIG) provided such industrial environment. Established around the year 2000, SIG provides consultancy services to help their clients to achieve a more controlled and efficient software lifecycle. The uniqueness of their services is the use of a fact-based approach, which lies on extracting metrics from software products using tools. Source code metrics and other analyses (facts) are then interpreted by SIG consultants and turned into actionable recommendations that clients can follow to meet their goals. Not only SIG has been innovative in their approach, but also has been a continuous contributor to science with their research. This duality between consultancy and research offered, on one hand, access the developed technology, industrial projects and client feedback and, on the other hand, a challenging and demanding research environment.

The research in this dissertation will be introduced as follows. Section 1.1 first discusses software product evaluation under the umbrella of software quality assurance. Section 1.2, reviews the ISO/IEC 9126 International Standard for Software Engineering – Product Quality [44] as applicable framework for this purpose. Section 1.3 intro-

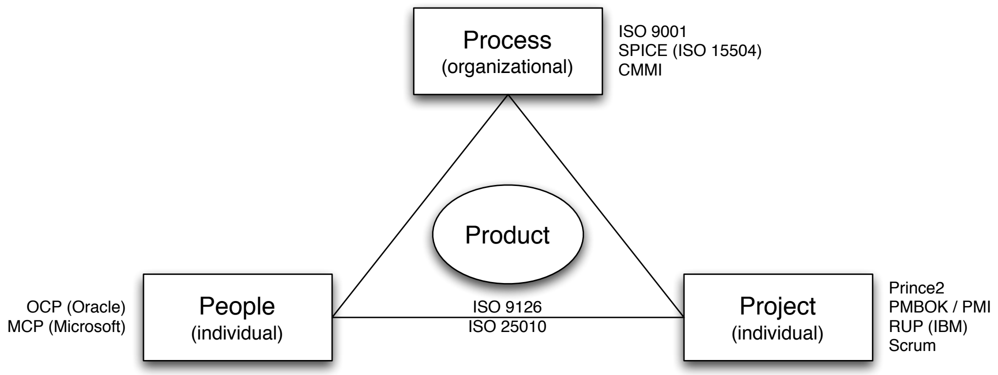


Figure 1.1: Bermuda triangle of software quality. The corners of the triangle indicate the main areas of focus of the software quality product assurance. In the center of the triangle is the software product which is an area commonly overlooked. Next to each area are examples of applicable standards.

duces the SIG quality model for maintainability as starting point for software product measurement. Section 1.4 proposes the use of software benchmarks to support the interpretation of measurements. Section 1.5 introduces the problem statement and the research questions. Section 1.6 presents the structure of the dissertation and the origin of the chapters. Finally, Section 1.7 lists other contributions made in the context of this PhD.

1.1 Software quality assurance

The traditional approach to software quality focuses on three main areas: process, people and project. Quality assurance on the product itself, i.e., the *concrete thing* that is being developed or maintained, is however often disregarded. This situation is depicted in Figure 1.1 as the “Bermuda Triangle of Software Quality”, introduced by Visser [88].

For process, the most known frameworks are ISO 9001 [47], SPICE [46] (or ISO/IEC 15504) and CMMI [83]. These frameworks define generic practices, requirements and guidelines to help organizations to meet their goals in a more structured and efficient way. They promote specification, control and procedures allowing the organization to improve the way they work. For CMMI and SPICE, organizations are appraised to a certain compliance level (called maturity level in CMMI and capability level in SPICE) defining the extent to which the organization follows the defined guidelines. For ISO 9001, organizations can be certified via a certification body.

For people, most software vendors provide their own professional certification services. The Microsoft Certified Professional (MCP) and the Oracle Certification Program (OCP) are well known examples of vendor-specific professional certifications. These certifications cover a broad range of technologies, from programming languages to the optimization or tailoring of specific products. There are also several levels each identifying a specific degree of knowledge.

For project, the most known standards are Prince2 [69] (PRojects IN Controlled Environment) by the UK Government, PMBOK [43] (A Guide to Project Management Body of Knowledge) from the Project Management Institute, RUP [55] (Rational Unified Process) from IBM and Scrum [76] by Ken Schwaber. In general they all provide guidelines to start, organize, execute (and control) and finalize temporary activities to achieve a defined project goal. For some of these standards software tools are available to support the project management activities.

There are several commonalities in quality assurance standards for process, people and project. They are all supported by specific certifications provided by third-parties and these certifications must be periodically renewed in order to remain valid. They all provide general guidelines leaving the specific definition of the actions to the people implementing them. The implementation of these standards offer confidence that a goal is going to be achieved although they are not bullet-proof against failure. Finally, these standards are well accepted by industry since both organizations and professionals recognize their importance and value.

The value of using standards for quality assurance comes basically from two scenarios: improvement and capability determination.

The value of using standards for improvement comes from efficiency increase. Process and project standards offer organizations and individuals a framework they can use to measure and evaluate their practices, leading for instance to a better usage of resources, time or money. This improvement scenario then offers the possibility of increasing the internal value, i.e., the value within the organization or project. Similarly, professional certification also offers internal value, offering the professional capabilities to develop certain activities more efficiently.

The value of using standards for capability determination comes from demonstrating to other that the organizations or individuals have advantages over their competitors. By following process and project standards, organizations or individuals, provide evidence that they are capable of delivering a service or a product, hence increasing their perceived or external value to others. The same is also applicable to personal certification, where the individuals can provide evidence that they are capable of delivering on a specific area.

A key aspect to the success of these standards and frameworks is the fact that they provide foundations for measurement, comparison and evaluation. Making these operational is key, providing added value to those who use them. The motivation and benefits that arise from the use of quality assurance standards/frameworks for quality assurance of process, people and projects are clear. They are wide-spread and well-accepted in industry.

Software product assurance, except for testing, has been given less importance when compared to other areas of quality assurance. For *long time*, reliability (as measured in number of failures) has been the single criteria for gauging software product quality [44]. Also, the mutual influence between product and process is well known, i.e., product quality affects process quality and vice-versa. The recognition of the need of a more well-defined criteria for software product quality lead to the development of the ISO/IEC 9126 [44] International Standard for Software Engineering – Product Quality. As of March 2011, ISO/IEC 9126 is being replaced by ISO/IEC 25010 [49], but since it is still very recent we will focus only on the former. Although ISO/IEC 9126 is well known it does not have the same acceptance as the other above mentioned quality assurance standards. Also, another fundamental problem is its operationalization, i.e., the implementation of this framework in practice such that it can be used to

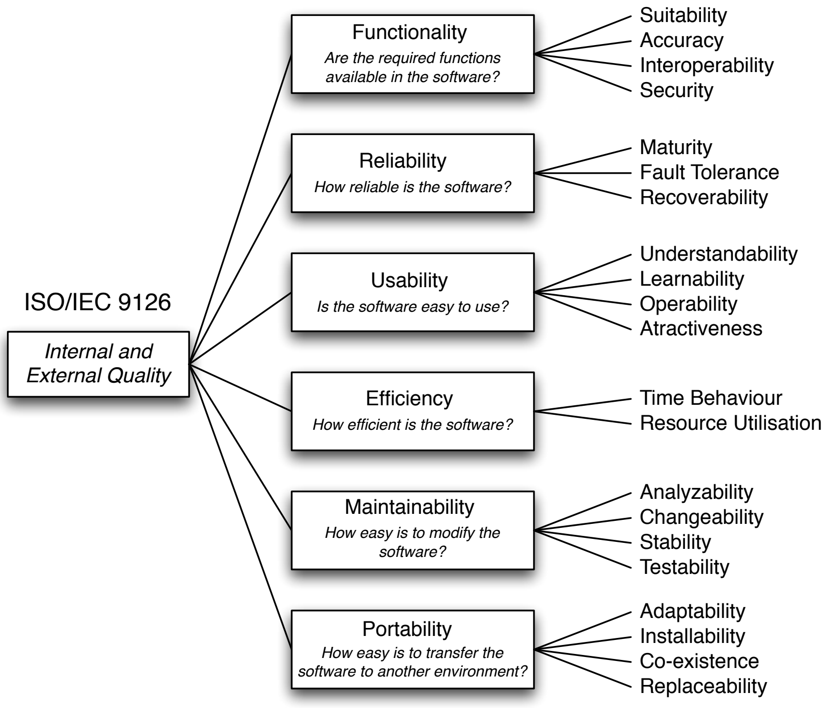


Figure 1.2: Quality Characteristics, sub-characteristics and attributes of the ISO/IEC 9126.

measure, compare and evaluate a software product [1].

1.2 ISO/IEC 9126 for Software Product Quality

The ISO/IEC 9126 [44] International Standard for Software Engineering – Product Quality, defines a model to support the definition of quality requirements (by setting quality goals) and the evaluation of quality of software products (by verifying if the goals are met). This model is meant to be applicable to every kind of software (e.g. source code and data) and is hierarchically decomposed into characteristics and sub-characteristics, covering all aspects of software product quality. Figure 1.2 shows at the highest level, on the left-hand side, the quality characteristics and its relation with the sub-characteristics of the ISO/IEC 9126 quality model. The lowest level of this model, on the right-hand side, consists of the software quality attributes that are going to be measured. Quality evaluation should be performed by specifying appropriate metrics (to measure software quality attributes) and acceptable ranges. The acceptable ranges are intended to verify if the quality for the attributes are met, hence validating the requirements for the quality sub-characteristics and then for the quality characteristics.

Whenever quality is a requirement, it is commonly specified in terms of external quality. External quality is the capability of the software product as perceived by the user during the execution of the system in a specific environment. Since software products are meant to be executed, external quality is clearly the final goal. External quality can be captured by measuring software product behavior, by testing, operating and observing the running system. The ISO/IEC 9126 defines that "before acquiring or using a software product it should be evaluated using metrics based on business objectives related to the use, exploitation and management of the product in a specified organizational and technical environment." This means that external quality can only be measured and evaluated after the product is ready. Also, this means that if only external quality is used as criteria, there will be a quality assurance gap between start of development/maintenance and delivery for external quality evaluation. Naturally this will lead to higher risks of the project not meeting the imposed requirements and higher costs to fix problems found during execution.

In addition to external quality, the ISO/IEC 9126 considers internal quality. Internal quality measures intrinsic properties of software products, including those derived from simulated behaviors, indicating external attributes of a software product. This is achieved by the analysis of the static properties of intermediate or deliverable software products. The majority of source code metrics, e.g. McCabe or Source Lines of Code (SLOC), and dependency analyses, e.g. coupling and cohesion metrics, provide measurements about static properties. The main advantage of using internal measurements is that they can be used at early stages of the development. At the earlier stages only resources and process can be measured. However, as soon as there are intermediate products or product deliverables (e.g. documentation, specifications or source code) it is possible to start using internal measurements and thus validating targets at various stages of development. Internal measurements allow users, evaluators, testers, and developers to benefit from the evaluation of the software product quality and identify quality issues early before the software product is finalized. Hence, internal metrics complement external metrics and together they cover the quality assurance of the whole software life-cycle.

It is important to stress, as stated by the ISO/IEC 9126, that the main goal of internal metrics is to predict/achieve external quality. For this reason, the internal attributes should be used to predict values of external metrics and hence used as indicators of external quality. However, ISO/IEC 9126 recognizes the challenge that "it is generally difficult to design a rigorous theoretical model which provides a strong relationship between internal and external metrics" and that "one attribute may influence one or more characteristic, and a characteristic may be influenced by more than one attribute." Although much research has been done in software metrics validation, this is still an open challenge. Moreover, it is partly due to this lack of evidence, that internal metrics can be predictors of external quality, that it is difficult to implement metrics programs within organizations.

This dissertation, will focus on internal metrics (metrics that are statically derived). Although a core set of metrics is used in this research, the goal is to provide a solution to solve some of the challenges of using metrics, in general, for software analysis and evaluation. In line with the ISO/IEC 9126, it is a concern that these internal metrics be predictors of external quality.

1.3 SIG quality model for maintainability

The ISO/IEC 9126 standard provides a framework for software product assurance. However, as noted by many authors [1, 20, 41, 52], the defined framework guidelines are not precise enough, leaving room for different interpretations which can lead to inconsistent evaluations of software product quality. Also, ISO/IEC 9126 does not foresee automatization when evaluating software product quality. Evidence of this lack of automatization can be found in the proposed internal metrics [45] which strongly de-

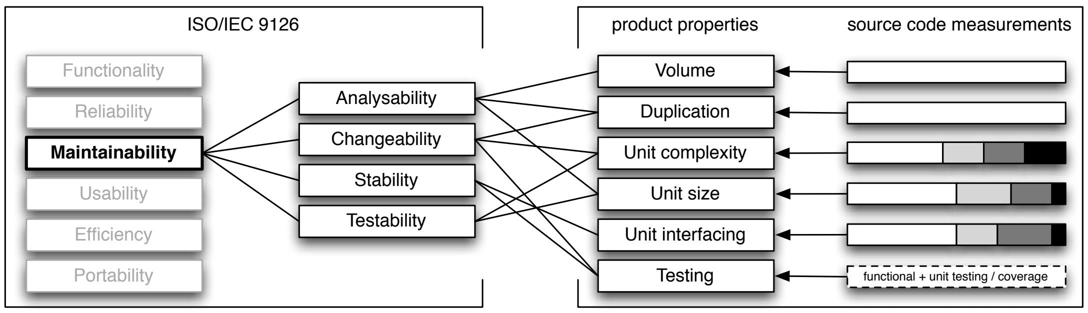


Figure 1.3: Quality model overview. On the left-hand side, the quality characteristics and the maintainability sub-characteristics of the ISO/IEC 9126 standard for software product quality are shown. The right-hand side relates the product properties defined by SIG with the maintainability sub-characteristics. In the source code measurements, the empty rectangle indicates system-level measurements, the four-piece rectangles indicate measurements aggregated using risk profiles, and the dashed-line rectangle indicates the use of criteria. This figure was adapted from Luijten et al. [63].

pend on human observation of software product behavior and its environment instead of its objective measurement.

To support the automatic analysis and evaluation of software products, Heitlager et al. proposed the SIG quality model for maintainability based on the ISO/IEC 9126. An extended version of this model is presented in Figure 1.3.

The SIG quality model for maintainability not only operationalizes ISO/IEC 9126, but also enables fully automatic evaluation of software quality. The automatization is possible due to the use of statically-derived source code metrics. Metrics are extracted using the Software Analysis Toolkit (SAT) also developed by SIG.

The SIG quality model was designed to use metrics that follow three requirements: technology independence, simplicity, and root-cause analysis capabilities. Technology independence is required in order to support multiple programming-languages hence not restricting the model to a single technology but enabling its general application. Simplicity was required both in terms of implementation/computation (so that it can scale to very large systems) and in terms of definition (so that it can be easily explained and used to communicate with non-technical people). Finally, root-cause analysis is required to narrow down the area of a potential problem and provide an explanation for it.

The quality model presented in Figure 1.3 makes use of seven system properties which are defined by the following metrics: *volume*, *duplication*, *unit complexity*, *unit size*, *unit interfacing* and *test quality*. While *volume*, *duplication* and *test quality* are measured at system-level, all the other metrics are measured at *unit*-level. “Unit” is used as generic term for designating the smallest block of code of a programming language. For Java and C# programming languages a unit will correspond to a method, while for C/C++ it will correspond to a function. *Volume* measures the overall system size in man-years via backfiring functions points, which is achieved by counting SLOC per technology and using the Programming Languages Table of Software Productivity Research LLC [59] to arrive at the final value in man-years. *Duplication* measures the percentage of code, in SLOC, that occurs more than once, in identical blocks of code of at least 6 lines. *Unit complexity* or McCabe cyclomatic complexity [67] measures the number of paths or conditions defined in a unit. *Unit size* measures the size in SLOC (blank lines and comments excluded). *Unit interfacing* measures the number of arguments that need be used to call this unit. *Test quality* measures the percentage of the overall system code that is covered by tests.

Each system-property is measured using a 5-point rating scale, obtained by aggregating the raw source code metrics as defined in [41]. System-level metrics are scaled to a rating by comparing the metric value to a rating interval. Unit-level metrics are aggregated in a two-level process using thresholds. Using first *risk thresholds*, a *risk profile* is created representing the percentage of SLOC that falls into *Low*, *Moderate*, *High* and *Very-high* categories. Then, using *rating thresholds*, the risk profile is aggregated to a rating. After computing the rating for all system-properties, the rating for the maintainability sub-characteristics and characteristics is computed by the weighted mean of one or more system properties and one or more sub-characteristics, respectively.

The SIG quality model for maintainability and the SAT that supports it were used as the starting point of this PhD research. These provided initial guidance on quality evaluation since this model was validated by SIG’s consultants. SAT also allowed for rapidly producing metrics for a representative set of industrial and Open-Source Software (OSS) systems which were used throughout this research. Although based on SIG's quality model and SAT, this research is not restricted in any way to this quality model metrics or to the SAT tool. As will become apparent in due course, all the results presented in this dissertation should be equally applicable, with minor effort, to other metrics or tools.

1.4 Benchmarks for software evaluation

One of the key challenges in software empirical studies is to have a large and representative set of software systems to analyze. For this research we used a benchmark of 100 systems constructed in modern Object-Oriented (OO) technologies (Java and C#). The benchmark systems, which come from both SIG customers and OSS projects, were developed by different organizations and cover a broad range of domains. System sizes range from over 3K LOC to near 800K SLOC, with a total of near 12 million SLOC. These numbers correspond to manually-maintained production code only (test, generated and library code are not included), as defined in [3]. Table 1.1 records the number of systems per technology (Java or C#) and license type (proprietary or OSS). Table 1.2 classify such software systems functionality according to the taxonomy defined by International Software Benchmarking Standards Group (ISBSG) in [61].

This benchmark was collected and curated during the SIG consultancy activities and their use is authorized by clients as long as the data remain anonymous. In contrast to other benchmarks, such as the Quality Corpus [85] or that made from SourceForge[1] projects as reported in [37], the SIG's benchmark offers two main advantages. The first advantage is that it contains a large percentage (near 80%) of industrial systems. Although having industrial systems pose some challenges for others to replicate measurements, their inclusion in the benchmark offers a more representative view of software systems than other OSS benchmarks. The second advantage is the access to

[1]http://sourceforge.net/

Table 1.1: Number of systems in the benchmark per technology and license.

Technology	License	# Systems	Overall size (SLOC)
Java	Proprietary	60	8,435K
	OSS	22	2,756K
C#	Proprietary	17	794K
	OSS	1	10K
	Total	100	11,996K

Table 1.2: Number of systems in the benchmark per functionality according to the ISBSG classification.

Functionality type	# Systems
Catalogue or register of things or events	8
Customer billing or relationship management	5
Document management	5
Electronic data interchange	3
Financial transaction processing and accounting	12
Geographic or spatial information systems	2
Graphics and publishing tools or system	2
Embedded software for machine control	3
Job, case, incident or project management	6
Logistic or supply planning and control	8
Management or performance reporting	2
Mathematical modeling (finance or engineering)	1
Online analysis and reporting	6
Operating systems or software utility	14
Software development tool	3
Stock control and order processing	1
Trading	1
Workflow support and management	10
Other	8
Total	100

key personnel with knowledge about the systems in SIG's benchmark. Whenever there is a question about a particular system either this question can be directly answered by a SIG consultant, who has overall knowledge of the system, or the consultant can directly contact the owner company to find the answer. This easy access to knowledge is not so common in OSS projects, even for those projects that are supported by large communities, since it is sometimes difficult to find the right person to answer a question and it always requires a lot of communication effort.

Another key issue of using a benchmark of software systems is the definition of system measurement scopes, i.e., the definition, for each system, of how the different software artifacts are taken into account and used in an analysis. This problem was identified in [85] and further investigated in [3]. An example of scope configuration can contain the distinction between production and test code, manually-maintained and generated code, etc. For instance, as reported in [3], the generated code for some systems can represent as high as 80% of the overall system code (in SLOC). It is clear that the correct differentiation of the system artifacts is of extreme importance since failure to recognize one of these categories can lead to an incorrect interpretation of results.

Unless otherwise stated, this research will use the SIG benchmark introduced in this section.

1.5 Problem statement and research questions

This dissertation deals with two fundamental problems that have been hindering the adoption and use of software product metrics for software analysis and evaluation: *(i)* how to interpret raw measurements? and *(ii)* how to obtain a meaningful overview of a given metric? The aim of this dissertation is to establish a body of knowledge allowing a software engineering to select a set of metrics and use these to infer knowledge about a particular software system.

The stakes are quite high. Metrics have been around since the beginning of the programming languages (e.g. SLOC). An extensive literature about software product metrics has been developed, including definition and formalization of software metrics, measuring theory, case studies and validation. Yet, it is still difficult to use metrics to capture and communicate about the quality of a system. This is mainly due to the lack of methodologies for establishing baseline values for software metrics and for aggregating and combining metrics in ways that make it possible to trace back the original problem. This dissertation aims to solve these problems. More specifically, the research questions that this dissertation aims to answer are:

(i) How to establish thresholds of software product metrics and use them to show the extent of problems in the code?

(ii) How to summarize a software product metric while preserving the capability of root-cause analysis?

(iii) How can quality ratings based on internal metrics be validated against external quality characteristics?

(iv) How to combine different metrics to fully characterize and compare the quality of software systems based on a benchmark?

Research Question 1

How to establish thresholds of software product metrics and use them to show the extent of problems in the code?

The most serious criticism about metrics is that they are simply numbers and hence very little can be said about them. Information about the numerical properties of the metrics can be derived. For instance, for metrics in ordinal scale we can compare two individual measurements and state that one is bigger/smaller than the other. We can additionally quantify the difference between two measurements. However, this does not allow one to infer which action (if any) to undertake driven by a specific measurement.

The real value of using metrics starts when comparing individual measurements with thresholds. This allows the division of the measurement space into regions, e.g. acceptable or non-acceptable, putting measurements into context, adding context to measurements. Based on this information, we can investigate and quantify non-acceptable measurements, taking action on them if necessary. This is the first step to infer knowledge about what is being measured. Then it is of importance to establish meaningful thresholds, i.e., thresholds that are easy to derive, explainable and justifiable, and that can be used in practice.

Several methodologies have been attempted in the past but failed by making unjustifiable assumptions. Thus, the challenge of developing a methodology for deriving metric thresholds is still to be met. In this dissertation, a new methodology is proposed by applying a series of transformations to the data, weighting measurements by relative size and deriving thresholds that are representative of benchmark data.

Research Question 2

> How to summarize a software product metric while preserving the capability of root-cause analysis?

The second serious problem when using metrics is that they provide very little information about the overall system. Metrics such as overall size, duplication and test coverage are defined at system level hence providing information about the overall system. However, the majority of metrics are defined at smaller granularity levels such as method, class or packages. Hence, using these metrics directly does not allow one to infer information about the system as a whole.

To infer information about the system from a given metric it is necessary to synthesize the measurements into a meaningful value. By meaningful it is meant that, this value not only captures the information of all measurements, but also it enables (some) traceability back to the original measurements. This synthesized value can then be used to communicate about the system, perform comparisons among systems or be used to track evolution or to establish targets.

The common techniques to summarize metrics at system level aggregate measurements using arithmetical addition or average functions (e.g. median, weighted/geometric/vanilla mean). However these techniques fail to capture the information of all measurements and the results are not traceable. In this dissertation a new methodology is proposed to synthesize metrics into ratings achieved by calibrating ratings using a large benchmark of software systems.

Research Question 3

> How can quality ratings based on internal metrics be validated against external quality characteristics?

By showing how to evaluate and compare systems according to an internal quality view obtained from source code metrics does not necessarily mean that this quality view can have a relation with external quality as perceived by a user. Although much research have been done in order to prove that in general source code metrics can be used as predictors for external quality characteristics, does this also apply when aggregating metrics using benchmark-derived thresholds?

By deriving thresholds for a new metric introduced in this dissertation, Static Estimation of Test Coverage (SETC), we investigate whether this static estimator can predict real coverage and be validated against bug solving efficiency.

We demonstrate that not only SETC can be used as early indicator for real coverage but also that SETC has a positive correlation with bug solving efficiency. This provides indirect evidence, that using thresholds to aggregate metrics is a powerful and valuable mechanisms for interpretation and evaluation of metrics.

Research Question 4

> How to combine different metrics to fully characterize and compare the quality of software systems based on a benchmark?

When the SIG quality model was introduced it made use of thresholds defined by expert opinion. By using benchmark-based thresholds is it still possible to infer valuable information about a software system?

The use of benchmark-based thresholds allow us to relate the final evaluation of a particular system with the systems of the benchmark. This way, measurements are put into context not only acting as a form of validation, but also adding knowledge to the evaluation results.

This dissertation demonstrates how to use thresholds obtained from a benchmark in order to do meaningful analysis and comparison of software systems. First, thresholds are used to investigate about quality issues. Second, using such thresholds to aggregate measurements into ratings, a quality comparison between the two different systems is done.

1.6 Sources of chapters

Each chapter of this dissertation is based on one peer-reviewed publication presented in an international conference. The first author is the main contributor of all chapters. The publication title, conference and list of co-authors is presented below for each individual chapter.

Chapter 2. Deriving Metric Thresholds from Benchmark Data. The source of this chapter was published in the proceedings of the 26th IEEE International Conference on Software Maintenance (ICSM 2010) as [6]. It is co-authored by Christiaan Ypma and Joost Visser.

Chapter 3. Benchmark-based Aggregation of Metrics to Ratings The source of this chapter was published in the proceedings of the Joint Conference of the 21th International Workshop on Software Measurement and the 6th International Conference on Software Process and Product Measurement (IWSM/MENSURA 2011) as [4]. It is co-authored by José Pedro Correia and Joost Visser.

Chapter 5. Assessment of Product Maintainability for Two Space Domain Simulators The source of this chapter was published in the proceedings of the 26th IEEE International Conference on Software Maintenance (ICSM 2010) as [2].

Chapter 4. Static Estimation of Test Coverage The source of this chapter was published in the proceedings of the 9th IEEE International Working Conference on

Source Code Analysis and Manipulation (SCAM 2009) as [5]. It is co-authored by Joost Visser.

1.7 Other contributions

During the course of this PhD other contributions were made in related areas of research which have indirectly contributed to the author's perspective on software quality. Such side stream contributions are listed below.

Categories of Source Code in Industrial Systems This paper was published in the proceedings of the 5th International Symposium on Empirical Software Engineering and Measurement (ESEM 2011), IEEE Computer Society Press.

Comparative Study of Code Query Technologies This paper was published in the proceedings of the 11th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2011), IEEE Computer Society Press. It is co-authored by Peter Rademaker and Jurriaan Hage.

A Case Study in Grammar Engineering This paper was published in the proceedings of the 1st International Conference on Software Language Engineering (SLE 2008), Springer-Verlag Lecture Notes in Computer Science Series. It is co-authored by Joost Visser.

Type-safe Evolution of Spreadsheets This paper was published in the proceedings of the conference Fundamental Approaches to Software Engineering (FASE 2011), Springer-Verlag Lecture Notes in Computer Science. It is authored by Jácome Cunha and co-authored by Joost Visser and João Saraiva.

Constraint-aware Schema Transformation This paper was accepted and presented at the 9th International Workshop on Rule-Based Programming (RULE 2008). Due to editorial problems, the proceedings of this conference were still not available at the time of writing this dissertation. It is co-authored by Paulo F. Silva and Joost Visser.

Chapter 2

Benchmark-based Derivation of Risk Thresholds

A wide variety of software metrics have been proposed and a broad range of tools is available to measure them [35, 22, 21, 90, 71]. However, the effective use of software metrics is hindered by the lack of meaningful thresholds. Thresholds have been proposed for a few metrics only, mostly based on expert opinion and a small number of observations.

Previously proposed methodologies for *systematically* deriving metric thresholds have made unjustified assumptions about the statistical properties of source code metrics. As a result, the general applicability of the derived thresholds is jeopardized.

The design of a method that determines metric thresholds empirically from measurement data is the main subject of this chapter. The measurement data for different software systems are pooled and aggregated after which thresholds are selected that *(i)* bring out the metric's variability among systems and *(ii)* help focusing on a reasonable percentage of the source code volume. The proposed method respects the distributions and scales of source code metrics, and it is resilient against outliers in metric values or system size.

The method has been tested by applying it to a benchmark of 100 Object-Oriented

(OO) software systems, both proprietary and Open-Source Software (OSS), to derive thresholds for metrics included in the Software Improvement Group (SIG) quality model for maintainability.

2.1 Introduction

Software metrics have been around since the dawn of software engineering. Well-known source code metrics include Source Lines of Code (SLOC), the McCabe metric [67], and the Chidamber-Kemerer suite of OO metrics [22]. Metrics are intended as a control instrument in the software development and maintenance process. For example, metrics have been proposed to identify problematic locations in source code to allow effective allocation of maintenance resources. Tracking metric values over time can be used to assess progress in development or to detect quality erosion during maintenance. Metrics can also be used to compare or rate the quality of software products, and thus form the basis of acceptance criteria or service-level agreements between software producer and client.

In spite of the potential benefits of metrics, their effective use has proven elusive. Although metrics have been used successfully for quantification, their use have generally failed to adequately support subsequent decision-making [34].

To promote the use of metrics from measurement to decision-making, it is essential to define meaningful *threshold values*. These have been defined for some metrics. For example, McCabe proposed a threshold value of 10 for his complexity metrics, beyond which a subroutine was deemed unmaintainable and untestable [67]. This threshold was inspired by experience in a particular context and not intended as universally applicable. For most metrics, thresholds are lacking or do not generalize beyond the context of their inception.

This chapter, presents a method to derive metric threshold values empirically from the measurement data of a benchmark of software systems. The measurement data for different software systems are first pooled and aggregated. Then thresholds are determined that *(i)* bring out the metric's variability among systems and *(ii)* help focusing on a reasonable percentage of the source code volume.

The design of the proposed method takes several requirements into account to avoid the problems of thresholds based on expert opinion and of earlier approaches to systematic derivation of thresholds. The method should:

1. be driven by measurement data from a representative set of systems (data-driven), rather than by expert opinion;

2. respect the statistical properties of the metric, such as metric scale and distribution and should be resilient against outliers in metric values and system size (robust);

3. be repeatable, transparent and straightforward to carry out (pragmatic).

In the explanation of the method given in the sequel the satisfaction of these requirements will be addressed in detail.

This chapter is structured as follows. Section 2.2 demonstrates the use of thresholds derived with the method introduced in this chapter, taking the McCabe metric as example. In fact, this metric is used as a vehicle throughout the chapter for explaining and justifying the method to derive thresholds. Section 2.3 provides an overview of the method itself and Section 2.4 provides a detailed explanation of its key steps. Section 2.5 discusses variants of the method and possible threats. Section 2.6 provides evidence of the wider applicability of the method by generalization to other metrics included in the SIG quality model for maintainability [41]. Section 2.7 presents an overview of earlier attempts to determine thresholds. Finally, Section 2.8 summarizes the contributions of this chapter.

Table 2.1: Risk profiles for the McCabe metric of four P2P systems.

System version	Low risk (%)	Moderate risk (%)	High risk (%)	Very-high risk (%)
JMule 0.4.1	70.52	6.04	11.82	11.62
LimeWire 4.13.1	78.21	6.73	9.98	5.08
FrostWire 4.17.2	75.10	7.30	11.03	6.57
Vuze 4.0.04	51.95	7.41	15.32	25.33

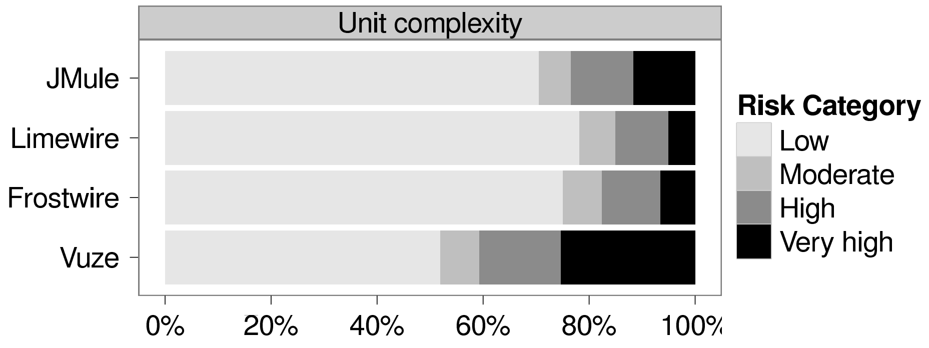


Figure 2.1: Risk profiles for the McCabe metric of four P2P systems.

2.2 Motivating example

Suppose one wants to compare the technical quality of four Peer-to-Peer (P2P) systems: JMule, LimeWire, FrostWire and Vuze[1]. Using the SIG quality model for maintainability [41] we can arrive at a judgement of technical quality of those systems, first by calculating *risk profiles* for each metric and then combining them into a rating. Focusing on the risk profiles, and using as example throughout this chapter the McCabe metric, the importance of thresholds to analyze quality will be demonstrated.

A risk profile defines the percentages of SLOC of all methods that fall in each of the following categories: low risk, moderate risk, high risk and very-high risk. To categorize the methods thresholds are used. For now let us assume thresholds 6, 8 and 15 that represent 70%, 80% and 90% of all code. This assumption will be justified in sequel when explaining the methodology for deriving thresholds from a benchmark of systems. Hence, using these thresholds it is possible to define four

[1]http://jmule.org/, http://www.limewire.com/, http://www.frostwire.com/, http://www.vuze.com/

intervals[2] that divide the McCabe measurements into four categories: $]0, 6]$ for low risk, $]6, 8]$ for moderate risk, $]8, 15]$ for high risk, and $]15, \infty[$ for very-high risk. Using the categorized measurements, the risk profiles are then calculated by summing the SLOC of all methods in each category and dividing by the overall system SLOC.

Figure 2.1 and Table 2.1 show the risk profiles of four P2P systems. Pinpointing potential problems can be done by looking at the methods that fall in the very-high risk category. Looking at the percentages of the risk profiles we can have an overview about overall complexity. For instance, the Vuze system contains 48% of code in moderate or higher risk categories, of which 25% is in the very-high risk category. Finally, quality comparisons can be performed: LimeWire is the least complex of the four systems, with 22% of its code in moderate or higher risk categories, followed by FrostWire (25%), then by JMule (30%) and, finally, Vuze (48%).

2.3 Benchmark-based threshold derivation

The methodology proposed in this section was designed according to the requirements declared earlier: *(i)* it should be based on data analysis from a representative set of systems (benchmark); *(ii)* it should respect the statistical properties of the metric, such as scale and distribution; *(iii)* it should be repeatable, transparent and straightforward to execute.

With these requirements in mind, Figure 2.2 summarizes the six steps of the methodology proposed in this chapter to derive thresholds.

1. Metrics extraction: metrics are extracted from a benchmark of software systems. For each system $System$, and for each entity $Entity$ belonging to $System$ (e.g. method), we record a metric value, $Metric$, and metric weight, $Weight$ for that system's entity. As weight we will consider the SLOC of the entity. As example, the method (entity) called `MyTorrentsView.createTabs()`, from the Vuze system, has a McCabe metric value of 17 and weight value of 119 SLOC.

[2]Intervals are represented using the ISO/IEC 80000–2 notation [48].

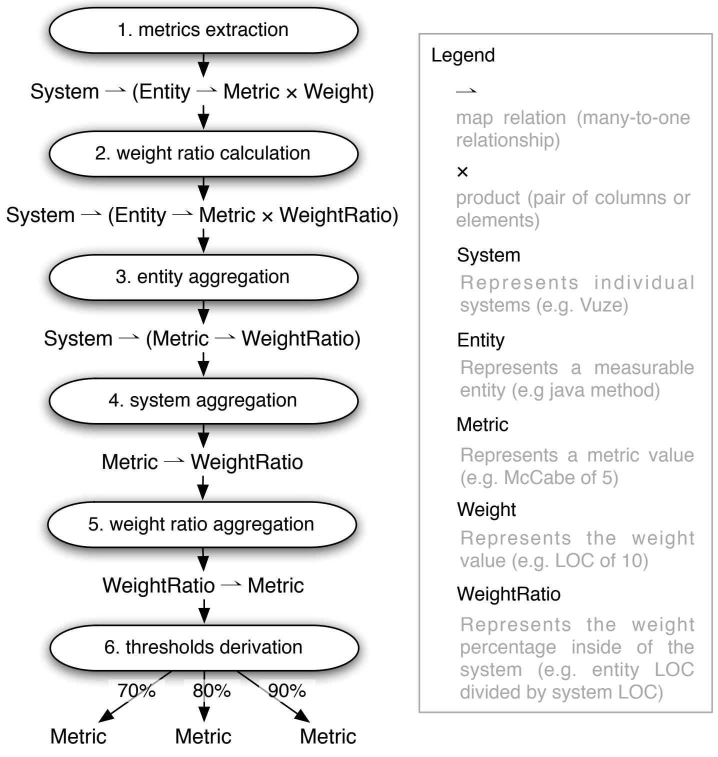

1. metrics extraction
System → (Entity → Metric × Weight)
2. weight ratio calculation
System → (Entity → Metric × WeightRatio)
3. entity aggregation
System → (Metric → WeightRatio)
4. system aggregation
Metric → WeightRatio
5. weight ratio aggregation
WeightRatio → Metric
6. thresholds derivation
70%
80%
90%
Metric
Metric
Metric
Legend
→
map relation (many-to-one relationship)
×
product (pair of columns or elements)
System
Represents individual systems (e.g. Vuze)
Entity
Represents a measurable entity (e.g java method)
Metric
Represents a metric value (e.g. McCabe of 5)
Weight
Represents the weight value (e.g. LOC of 10)
WeightRatio
Represents the weight percentage inside of the system (e.g. entity LOC divided by system LOC)

Figure 2.2: Summary of the methodology steps to derive risk threshold.

2. *Weight ratio calculation*: for each entity, we compute its weight percentage within its system, i.e., we divide the entity weight by the sum of all weights of the same system. For each system, the sum of all entities $WeightRatio$ must be 100%. As example, for the `MyTorrentsView.createTabs()` method entity, we divide 119 by $329,765$ (total SLOC for Vuze) which represents 0.036% of the overall Vuze system.

3. *Entity aggregation*: we aggregate the weights of all entities per metric value, which is equivalent to computing a weighted histogram (the sum of all bins must be 100%). Hence, for each system we have a histogram describing the distribution of weight per metric value. As example, all entities with a McCabe value of 17 represent 1.458% of the overall SLOC of the Vuze system.

4. *System aggregation*: we normalize the weights for the number of systems and then aggregate the weight for all systems. Normalization ensures that the sum of all

bins remains 100%, and then the aggregation is just a sum of the weight ratio per metric value. Hence, we have a histogram describing a weighted metric distribution. As example, a McCabe value of 17 corresponds to 0.658% of all code in the benchmark.

5. Weight ratio aggregation: we order the metric values in ascending way and take the maximal metric value that represents each weight percentile, e.g. 1%, 2%, ..., 100% of the weight. This is equivalent to computing a density function, in which the x-axis represents the weight ratio (0-100%), and the y-axis the metric scale. As example, according to the benchmark used in this dissertation, for 60% of the overall code the maximal McCabe value is 2.

6. Thresholds derivation: thresholds are derived by choosing the percentage of the overall code we want to represent. For instance, to represent 90% of the overall code for the McCabe metric, the derived threshold is 14. This threshold is meaningful, since not only it means that it represents 90% of the code of a benchmark of systems, but it also can be used to identify 10% of the worst code.

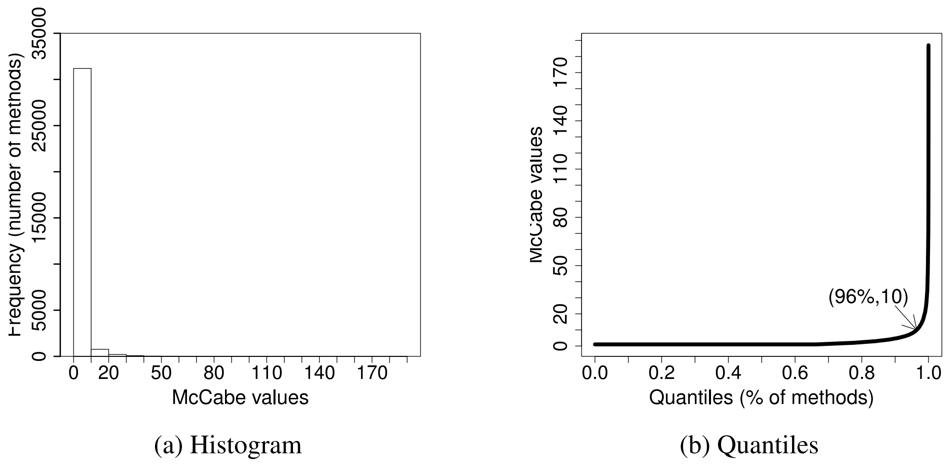
As a final example, SIG uses thresholds derived by choosing 70%, 80% and 90% of the overall code, which yields thresholds 6, 8 and 14, respectively. This allows to identify code to be fixed in long-term, medium-term and short-term, respectively. Furthermore, these percentiles are used in risk profiles to characterize code according to four categories: low risk $[0\%, 70\%]$, moderate risk $]70\%, 80\%]$, high risk $]80\%, 90\%]$ and very-high risk $]90\%, 100\%]$.

An analysis of these steps is presented in Section 2.4.

2.4 Analysis of the methodology steps

The methodology introduced in Section 2.3 makes two major decisions: weighting by size, and using relative size as weight. This section, provides thorough explanations based on data analysis about these decisions that are a fundamental part of the methodology. The representativeness of the derived thresholds is also investigated.

Section 2.4.1 introduces the statistical analysis and plots used throughout the dis-



(a) Histogram (b) Quantiles

Figure 2.3: McCabe distribution for Vuze system depicted with a histogram and a quantile plot.

sertation. Section 2.4.2 provides a detailed explanation about the effect of weighting by size. Section 2.4.3 shows the importance of the use of relative size when aggregating measurements from different systems. Finally, Section 2.4.4 provides evidence of the representativeness of the derived thresholds by applying the thresholds to the benchmark data and checking the results. Variants and threats are discussed later in Section 2.5.

2.4.1 Background

A common technique to visualize a distribution is to plot a histogram. Figure 2.3a depicts the distribution of the McCabe metric for the Vuze system. The x-axis represents the metric values and the y-axis represents the number of methods that have such a metric value (frequency). Figure 2.3a allows us to observe that more than 30.000 methods have a McCabe value ≤ 10 (the frequency of the first bin is 30.000).

Histograms, however, have several shortcomings. The choice of bins affects the shape of the histogram possibly causing misinterpretation of data. Also, it is difficult to compare the distributions of two systems when they have different sizes since the y-axis can have significantly different values. Finally, histograms are not very good to represent the bins with lower frequency.

To overcome these problems, an alternative way to examine a distribution of values is to plot its Cumulative Density Function (CDF) or the CDF inverse, the Quantile function. Figure 2.3b depicts the distribution of the McCabe values for the Vuze system using a Quantile plot. The x-axis represents the percentage of observations (percentage of methods) and the y-axis represents the McCabe metric values. The use of the quantile function is justifiable, because we want to determine thresholds (the dependent variable, in this case the McCabe values) as a function of the percentage of observations (independent variable). Also, by using the percentage of observations instead of the frequency, the scale becomes independent of the size of the system making it possible to compare different distributions. In Figure 2.3b we can observe that 96% of methods have a McCabe value ≤ 10.

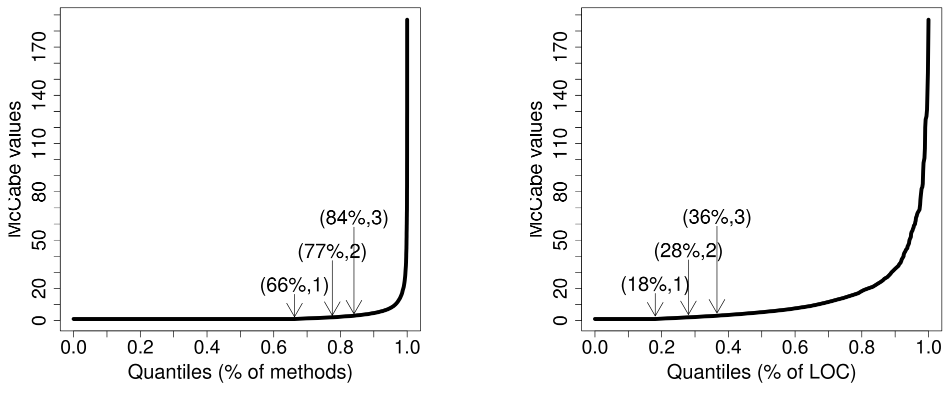
Despite that histograms and quantile plots represent the same information, the latter allows for better visualization of the full metric distribution. Therefore, in this dissertation all distributions will be depicted with quantile plots.

All the statistical analysis and charts were done with the R tool [84].

2.4.2 Weighting by size

Figure 2.4a depicts the McCabe metric distribution for the Vuze system already presented in Figure 2.3b in which we annotated the quantiles for the first three changes of the metric value. We can observe that up to the 66% quantile the McCabe value is 1, i.e., 66% of all methods have a metric value of 1. Up to the 77% quantile, the McCabe values are smaller than or equal to 2 ($77 - 66 = 11\%$ of the methods have a metric value of 2), and up to the 84% quantile have a metric value smaller than or equal to 3. Only 16% of methods have a McCabe value higher than 3. Hence, Figure 2.4a shows that the metric variation is concentrated in just a small percentage of the overall methods.

Instead of considering every method equally (every method has a weight of 1),



(a) Non-weighted (b) Weighted

Figure 2.4: McCabe distribution for the Vuze system (non-weighted and weighted by SLOC) annotated with the x and y values for the first three changes of the metric.

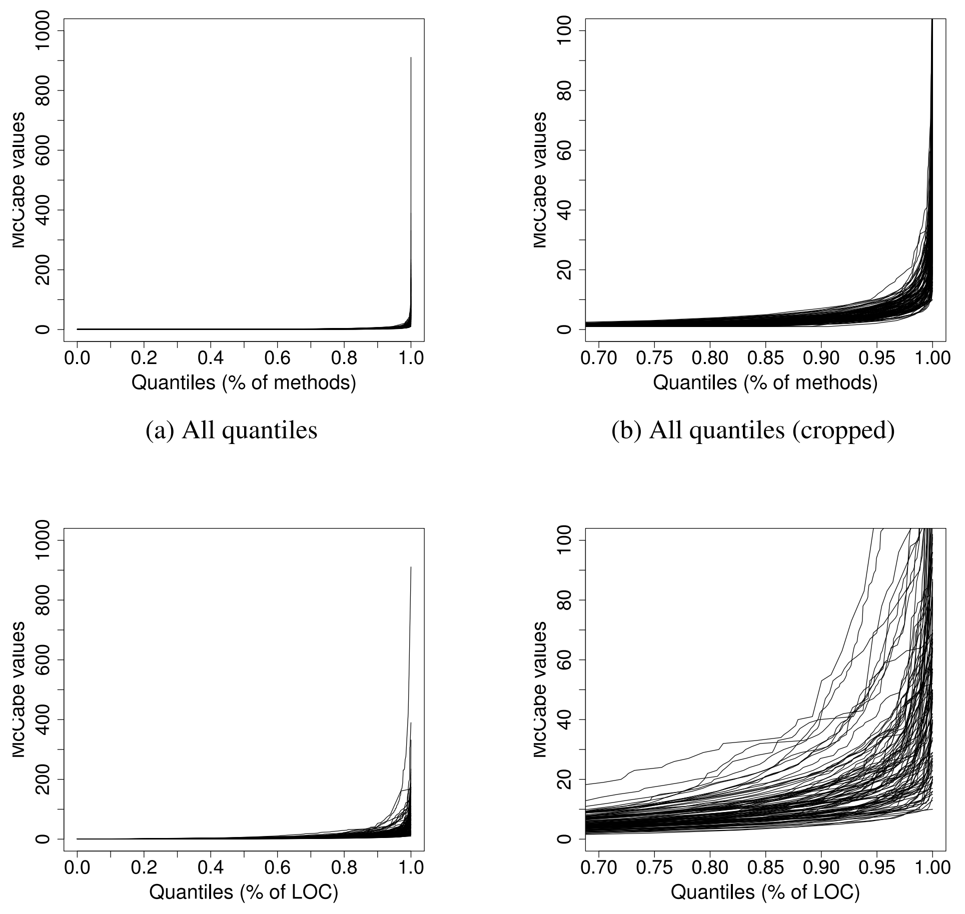
SLOC will be used as its weight. Figure 2.4b depicts the weighted distribution of the McCabe values for the Vuze system. Hence, the x-axis instead of representing the percentage of methods will now represent the percentage of SLOC.

Comparing Figure 2.4a to Figure 2.4b, we can observe that in the weighted distribution the variation of the McCabe values starts much earlier. The first three changes for the McCabe values are at 18%, 28% and 36% quantiles.

In sum, for the Vuze system, both weighted and non-weighted plots show that large McCabe values are concentrated in just a small percentage of code. However, while in the non-weighted distribution the variation of McCabe values happen in the tail of the distribution (66% quantile), for the weighted distribution the variation starts much earlier, at the 18% quantile.

Figure 2.5 depicts the non-weighted and weighted distributions of the McCabe metric for 100 projects. Each line represents an individual system. Figures 2.5a and 2.5c depict the full McCabe distribution and Figures 2.5b and 2.5d depict a cropped version of the previous, restricted to quantiles at least 70% and to a maximal McCabe value of 100.

When comparing Figure 2.4 to Figure 2.5 we observe that, as seen for the Vuze



(a) All quantiles

(b) All quantiles (cropped)

(c) All weighted quantiles

(d) All weighted quantiles (cropped)

Figure 2.5: Non-weighted and weighted McCabe distributions for 100 projects of the benchmark.

system, weighting by SLOC emphasizes the metric variability.

Hence, weighting by SLOC not only emphasizes the difference among methods in a single system, but also make the differences among systems more evident. A discussion about the correlation of the SLOC metric with other metrics and its impact on the methodology presented in Section 2.5.1.

2.4.3 Using relative size

To derive thresholds it is necessary to summarize the metric, i.e., to aggregate the measurements from all systems.

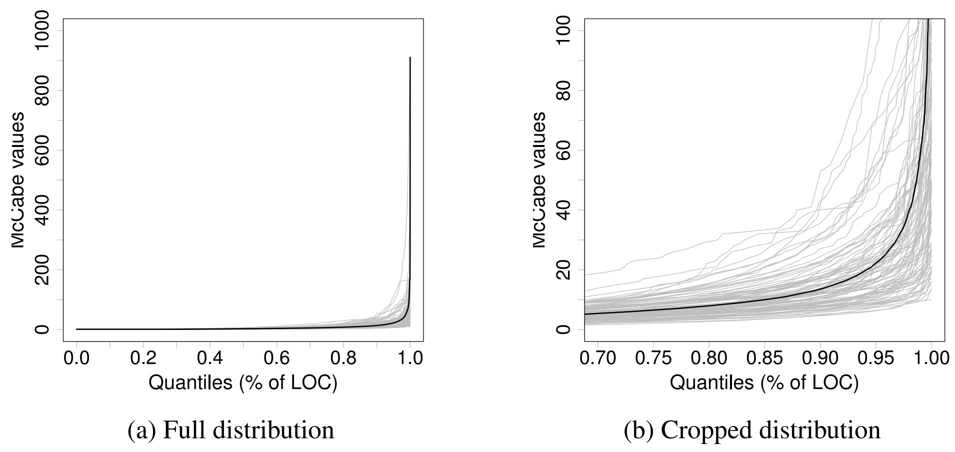


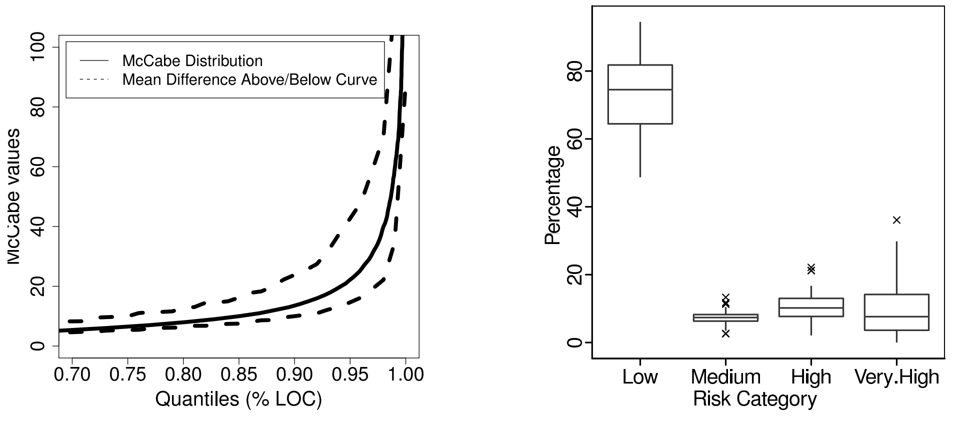
Figure 2.6: Summarized McCabe distribution. The line in black represents the summarized McCabe distribution. Each gray line depicts the McCabe distribution of a single system.

To summarize the metric, first a weight normalization step is performed. For each method, the percentage of SLOC that it represents in the system is computed, i.e., the method's SLOC is divided by the total SLOC of the system it belongs to. This corresponds to Step 2 in Figure 2.2 where upon all measurements can be used together.

Conceptually, to summarize the McCabe metric, one takes all density function curves for all systems and combine them into a single curve. Performing weight normalization ensures that every system is represented equally in the benchmark, limiting the influence of bigger systems over small systems in the overall result.

Figure 2.6 depicts the density functions of the summarized McCabe metric (plotted in black) and the McCabe metric for all individual systems (plotted in gray), also shown in Figure 2.5d. As expected, the summarized density function respects the shape of individual system's density function.

A discussion of alternatives to summarize metric distributions will be presented in Section 2.5.2.



(a) Distribution mean differences (b) Risk profiles variability

Figure 2.7: McCabe distribution variability among benchmark systems.

2.4.4 Choosing percentile thresholds

We have observed in Figures 2.5 and 2.6 that systems differentiate the most in the last quantiles. This section provides evidence that it is justifiable to choose thresholds in the tail of the distribution and that the derived thresholds are meaningful.

Figure 2.7a quantifies the variability of the McCabe distribution among systems. The full line depicts the McCabe distribution (also shown in Figure 2.6) and the dashed lines depict the median absolute deviation (MAD) above the distribution and below the distribution. The MAD is a measure of variability defined as the mean of the absolute differences between each value and a central point. Figure 2.7a shows that both the MAD above and below the curve increase rapidly towards the last quantiles (it has a similar shape as the metric distribution itself). In summary, from Figure 2.7a, we can observe that the variability among systems is concentrated in the tail of the distribution. It is important to take this variability into account when choosing a quantile for deriving a threshold. Choosing a quantile for which there is very low variability (e.g. 20%) will result in a threshold unable to distinguish quality among systems. Choosing a quantile for which there is too much variability (e.g. 99%) might fail to identify code in many systems. Hence, to derive thresholds it is justifiable to choose quantiles from the tail of the distribution.

As part of our methodology it was proposed the use of the 70%, 80% and 90% quantiles to derive thresholds. For the McCabe metric, using the benchmark, these quantiles yield to thresholds 6, 8 and 14, respectively.

Now we are interested to investigate if the thresholds are indeed representative of such percentages of code. For this, risk profiles for each system in the benchmark were computed. For low risk, it was consider the SLOC for methods with McCabe between 1–6, for moderate risk 7–8, for high risk 9–14, and for very-high risk > 14. This means that for low risk we expect to identify around 70% of the code, and for each of the other risk categories 10% more. Figure 2.7b depicts a box plot for all systems per risk category. The x-axis represents the four risk categories, and the y-axis represents the percentage of volume (SLOC) of each system per risk category. The size of the box is the interquartile range (IQR) and is a measure of variability. The vertical lines indicate the lowest/highest value within 1.5 IQR. The crosses in the charts represent systems whose risk category is higher than 1.5 IQR. In the low risk category, we observe large variability which is explained because it is considering a large percentage of code. For the other categories, from moderate to very-high risk, variability increases. This increase of variability is to be expected, since the variability of the metric is higher for the last quantiles of the metric. Only a few crosses per risk category exist, which indicates that most of the systems are represented by the box plot. Finally, for all risk categories, look at to the line in the middle of the box, the median of all observations, to observe that indeed this meets our expectations. For low risk category, the median is near 70%, while for other categories the median is near 10% which indicates that the derived thresholds are representative of the chosen percentiles.

Summing up, the box plot shows that the derived thresholds allow for observing differences among systems in all risk categories. As expected, around 70% of the code is identified in the low risk category and around 10% is identified for the moderate, high and very-high risk categories since the boxes are centered around the expected percentages for each category.

2.5 Variants and threats

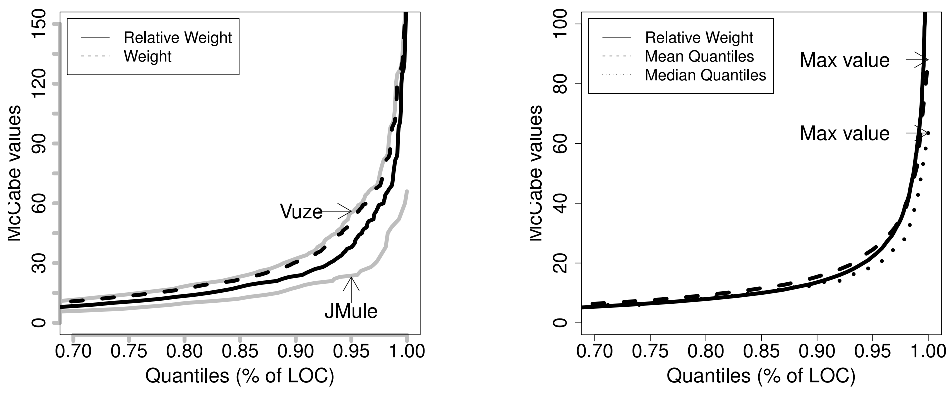
This section presents a more elaborated discussion and alternatives taken into account regarding two decisions in the methodology: weighting with size, and using relative size. Issues such as regarding removal of outliers and other issues affecting metric computation are also discussed.

Section 2.5.1 addresses the rationale behind using SLOC to weight the metric and possible risks due to metric correlation. Section 2.5.2 discusses the need for aggregating measurements using relative weight, and discusses possible alternatives to achieve similar results. Section 2.5.3 explains how to identify outliers and to find criteria to remove them. Finally, Section 2.5.4 explains the impact of using different tools or configurations when deriving metrics.

2.5.1 Weight by size

A fundamental part of the methodology is the combination of two metrics. More precisely, a metric for which thresholds are going to be derived with a size metric such as e.g. SLOC. In some contexts, particular attention should be paid when combining two metrics. For instance, when designing a software quality model it is desirable that each metric measures a unique attribute of the software. When two metrics are correlated it is often the case that they are measuring the same attribute. In this case only one should be used. We acknowledge such correlation between McCabe and SLOC. The Spearman correlation value between McCabe and SLOC for our data set (100 systems) is 0.779 with very-high confidence (p-value < 0.01).

The combination of metrics has a different purpose in the methodology. SLOC is regarded as a measure of size and used to improve the representation of the part of the system we are characterizing. Instead of assuming every unit (e.g. method) of



(a) Effect of large systems in aggregation

(b) Mean, Median as alternatives to Relative Weight

Figure 2.8: Effect of using relative weight in the presence of large systems and comparison with alternatives.

the same size, we take its size in the system measured in SLOC. Earlier on, it was emphasized that the variation of the metric allows for a more clear distinction among software systems. Hence, the correlation among SLOC and other metrics poses no problem.

The SLOC metric was measured considering physical lines of code, however logical lines of code could also been used. Similar results are to be expected in any case, since these metrics are highly correlated. Although other SLOC metrics could be considered, the study of such alternatives being deferred to future work.

2.5.2 Relative weight

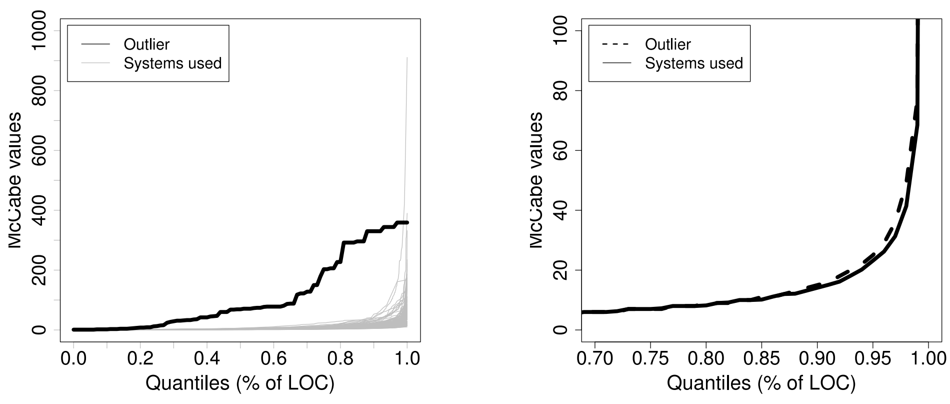
Section 2.4.3 advocates the use of relative weight in aggregating measurements from all systems. The rationale is that, since all the systems have similar distributions, the overall result should equally represent all systems. If we consider all measurements together without applying any aggregation technique the larger systems (systems with a bigger SLOC) would take over and influence the overall result.

Figure 2.8 compares the influence of size between simply aggregating all measurements together (black dashed line) and using relative weight (black full line) for the Vuze and JMule McCabe distributions (depicted in gray). Vuze has about 330 thousand SLOC, while JMule has about 40 thousand SLOC. In Figure 2.8a, were all measurements were simply added together, we can observe that the dashed line is very close to the Vuze system. This means that the Vuze system has a strong influence in the overall result. In comparison, Figure 2.8b shows that using the relative weight results in a distribution in the middle of the Vuze and JMule distributions as depicted in the full line. Hence, the use of relative weight is justifiable since it ensures size independence and takes into account all measurements in equal proportion.

As alternative to the use of relative weight, the mean/median quantile for all systems could be considered. With these techniques, the aggregated distribution would be computed by taking the mean/median of all distributions for each quantile. Figure 2.8b compares relative weight to mean quantile and median quantile. As can be observed, all distribution shapes are similar, thought the thresholds for the 70%, 80% and 90% quantiles would be different. However, there are reasons that point that use of the mean/median quantiles is not indicated. Mean is a good measure of central tendency if the underneath distribution is normal. Shapiro-Wilk test [65] for normality was applied for all quantiles and verified that the distribution is not normal. Additionally, the mean is sensitive to extreme values, and would favor higher values when aggregating measurements which is a non-desirable property. Finally, by using the mean/median the metric maximal value will not correspond to the maximal observable value, hiding information about the metric distribution. For the benchmark, the maximal McCabe value is 911. However, from Figure 2.8b, for the mean and median, the metric values for the 100% quantile (maximal value of the metric) are much lower.

2.5.3 Outliers

In statistics, it is common practice to check for the existence of outliers. An outlier is an observation whose value is distant relative to a set of observations. According



(a) McCabe distribution with an outlier

(b) McCabe characterization with and without an outlier

Figure 2.9: Example of outliers and outlier effect on the McCabe characterization.

to Mason et al. [65], outliers are relevant because they can obfuscate the phenomena being studied or may contain interesting information that is not contained in other observations. There are several strategies to deal with outliers: observations removal, or use of outlier-resistant techniques.

When preparing the SIG benchmark, the metric distribution among all systems were compared. Figure 2.9a depicts the distribution of the McCabe metric for our data set of 100 systems (in gray) plus one outlier system (in black) that was not included in the benchmark. Clearly, the outlier system has a metric distribution radically different from the other systems.

Figure 2.9b depicts the impact of the outlier on summarizing the McCabe metric. The full line represents the curve that summarizes the McCabe distribution for 100 systems, previously shown in Figure 2.6, and the dashed line represents the result of the 100 systems plus the outlier. Figure 2.9b, shows that the presence of the outlier has limited influence in the overall result, meaning that the methodology has resilience against outliers.

2.5.4 Impact of the tools/scoping

The computation of metric values and metric thresholds can be affected by the measurement tool and by scoping.

Different tools implement different variations of the same metrics. Taking as example the McCabe metric, some tools implement the Extended McCabe metric, while others might implement the Strict McCabe metric. As the values from these metrics can be different, the computed thresholds can also be different. To overcome this problem, the same tool should be used both to derive thresholds and to analyze systems using the derived thresholds.

The configuration of the tool with respect to which files to include or exclude in the analysis (scoping) also influences the computed thresholds. For instance, the existence of unit test code, which contains very little complexity, will result in lower threshold values. On the other hand, the existence of generated code, which is normally of very high complexity, will result in higher threshold values. Hence, it is extremely of crucial importance to know which data is used for calibration. As previously stated, for deriving thresholds we removed both generated code and test code from our analysis.

2.6 Thresholds for SIG's quality model metrics

Thus far, the McCabe metric was used as case study. To further investigate the applicability of the method to other metrics, its analysis was repeated for the SIG quality model metrics. The conclusion was that the method can be successfully applied to derive thresholds for all these metrics.

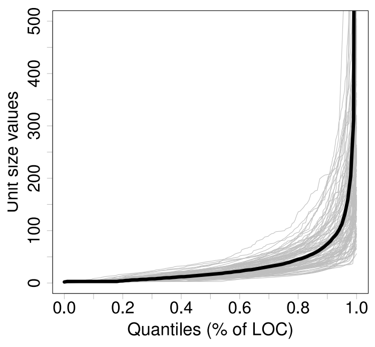
Table 2.2 summarizes the quantiles adopted and the derived thresholds for all the metrics from the SIG quality model.

Figures 2.10, 2.11, 2.12, and 2.13 depict the distribution and the box plot per risk category for unit size (method size in SLOC), unit interfacing (number of parameters per method), module inward coupling (file fan-in), and module interface size (number

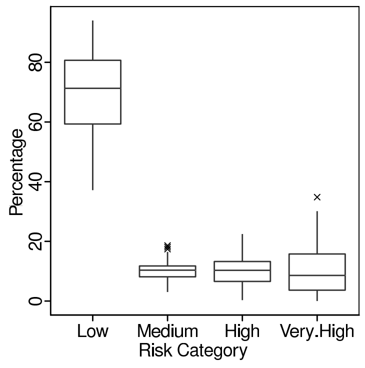
Table 2.2: Metric thresholds and adopted quantiles for the SIG quality model metrics.

Metric / Quantiles	70%	80%	90%
Unit complexity	6	8	14
Unit size	30	44	74
Module inward coupling	10	22	56
Module interface size	29	42	73

Metric / Quantiles	80%	90%	95%
Unit interfacing	2	3	4

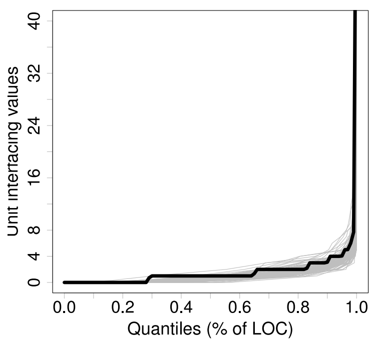


(a) Metric distribution

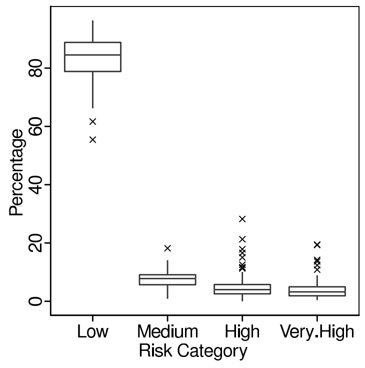


(b) Box plot per risk category

Figure 2.10: Unit size (method size in SLOC).



(a) Metric distribution

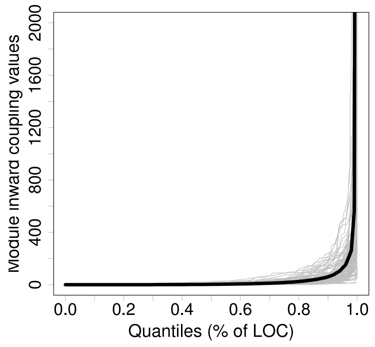


(b) Box plot per risk category

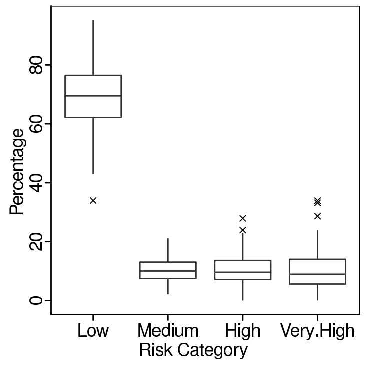
Figure 2.11: Unit interface (number of parameters).

of methods per file), respectively.

The distribution plots shows that, as for McCabe, for all metrics both the highest values and the variability among systems is concentrated in the last quantiles.

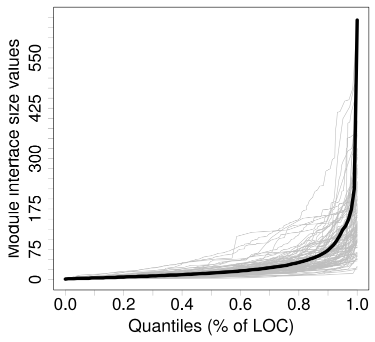


(a) Metric distribution

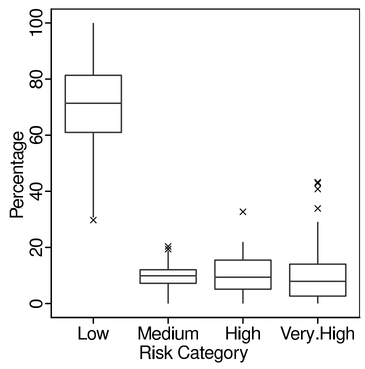


(b) Box plot per risk category

Figure 2.12: Module Inward Coupling (file fan-in).



(a) Metric distribution



(b) Box plot per risk category

Figure 2.13: Module Interface Size (number of methods per file).

As for the McCabe metric, for all metrics it was verified if the thresholds were representative of the chosen quantiles. The results are again similar. For all metrics except unit interfacing metric, the low risk category is centered around 70% of the code and all other categories around 10%. For the unit interfacing metric, since the variability is relative small until the 80% quantile, 80%, 90% and 95% quantiles were used instead to derive thresholds. For this metric, the low risk category is around 80%, the moderate risk is nearly 10% and the other two around 5%. Hence, from the box plots we can observe that the thresholds are indeed recognizing code around the defined quantiles.

2.7 Related Work

This section reviews previous attempts to define metric thresholds and compares it to the method proposed in this chapter. Works where thresholds are defined by experience are first discussed. Then, a comparison with other methods that derive thresholds based on data analysis is presented. An overview about approaches to derive thresholds based on error information and from cluster analysis is made. Finally, techniques to analyze and summarize metric distributions are discussed.

2.7.1 Thresholds derived from experience

Many authors have defined metric thresholds according to their experience. For example, for the McCabe metric 10 was defined as the threshold [67], and for the NPATH metric 200 was defined as the threshold [68]. Above these values, methods should be refactored. For the Maintainability Index metric, 65 and 85 are defined as thresholds [23]. Methods whose metric values are higher than 85 are highly-maintainable, between 85 and 65 are moderately maintainable and, smaller than 65 are difficult to maintain.

Since these values rely on experience, they are difficult to reproduce or generalize. Also, lack of scientific support leads to disputes about the values. For instance, someone dealing with small and low complexity systems would suggest smaller thresholds for complexity than someone dealing with very large and complex systems.

In contrast, having a methodology to define metric thresholds enables reproducibility and validation of results.

2.7.2 Thresholds from metric analysis

Erni et al. [30] propose the use of mean (μ) and standard deviation (σ) to derive a threshold T from project data. A threshold T is calculated as $T = \mu + \sigma$ or $T = \mu - \sigma$ when *high* or *low* values of a metric indicate potential problems, respectively. This methodology is a common statistical technique which, when data are normally distributed, identifies 16% of the observations. However, Erni et al. do not analyze the underlying distribution, and only apply it to one system, albeit using three releases.

The problem with the use of this method is that metrics are assumed to be normally distributed without justification, thus compromising its validity in general. Consequently, there is no guarantee that 16% of observations will be identified as problematic code. For metrics with high values and high variability, this methodology will identify less than 16% of code, while for metrics with low values or low variability, it will identify more than 16% of code.

In contrast, the method proposed in this chapter does not assume data normality. Moreover, it has been applied to 100 projects, both proprietary and OSS.

French [36] also proposes a formula based on the mean (μ) and standard deviation (σ) but using additionally the Chebyshev's inequality theorem.

A metric threshold T, whose validity is not restricted to normal distributions, is calculated as $T = \mu + k \times \sigma$, where k is the number of standard deviations. According to Chebyshev's theorem, for any distribution $1/k^2$ is the maximal portion of observations outside k standard deviations. As example, to identify a 10% maximum of code, we determine the value of k by resolving $0.1 = 1/k^2$.

However, French's method divides the Chebyshev's formula by two, which is only valid for two-tailed symmetric distributions. The assumption of two-tailed symmetric distributions is not justified. For one tailed distributions, the Cantelli's formula, $1/(1 + k^2)$, should have been used instead.

Additionally, this formula is sensitive to large numbers or outliers. For metrics with high range or high variation, this technique will identify a smaller percentage of observations than its theoretical maximum.

In contrast, the proposed method derives thresholds from benchmark data and is resilient to high variation of data outliers. Also, while French applies his technique to eight Ada95 and C++ systems, the proposed methods uses 100 Java and C# systems.

2.7.3 Thresholds using error models

Shatnawi et al. [79] investigate the use of the Receiver-Operating Characteristic (ROC) method to identify thresholds for predicting the existence of bugs in different error categories. They perform an experiment using the Chidamber and Kemerer (CDK) metrics [22] and apply the technique to three releases of Eclipse.

Although Shatnawi et al. were able to derive thresholds to predict errors, there are two drawbacks in their results. First, the methodology does not succeed in deriving monotonic thresholds, i.e., lower thresholds were derived for higher error categories than for lower error categories. Second, for different releases of Eclipse, different thresholds were derived.

In comparison, the proposed methodology is based only in metric distribution analysis, it guarantees monotonic thresholds and the addition of more systems causes only negligible deviations.

Benlarbi et al. [14] investigate the relation of metric thresholds and software failures for a subset of the CDK metrics using linear regression. Two error probability models are compared, one with threshold and another without. For the model with threshold, zero probability of error exists for metric values below the threshold. The authors conclude that there is no empirical evidence supporting the model with threshold as there is no significant difference among the models.

El Eman et al. [29] argue that there is no optimal class size based on a study comparing class size and faults. The existence of an optimal size is based on the Goldilocks conjecture which states that the error probability of a class increases for a metric values higher or lower a specific threshold (resembling a U-shape).

The studies of Benlarbi et al. [14] and El Eman et al. [29] show that there is no empirical evidence for the threshold model used to predict faults. However, these results are only valid for the specific error prediction model and for the metrics the authors took into account. Other models can, potentially, give different results.

In contrast to using errors to derive thresholds, the proposed method derives meaningful thresholds which represent overall volume of code from a benchmark of systems.

2.7.4 Thresholds using cluster techniques

Yoon et al. [92] investigate the use of the K-means Cluster algorithm to identify outliers in the data measurements.

Outliers can be identified by observations that appear either in isolated clusters (external outliers), or by observations that appear far away from other observations within the same cluster (internal outliers). However, this algorithm suffers from several shortcomings: it requires an input parameter that affects both the performance and the accuracy of the results; the process of identifying the outliers is manual; after identifying outliers the algorithm should be executed again; if new systems are added to the sample the thresholds might change significantly.

In contrast, the accuracy of the proposed method is not influenced by input parameters, it is automatic and it is stable (the addition of more systems results only in small variation).

2.7.5 Methodologies for characterizing metric distribution

Chidamber and Kemerer [22] use histograms to characterize and analyze data. For each of their 6 metrics, they plot histograms per programming language to discuss metric distribution and spot outliers in two C++ projects and one Smaltalk project.

Spinellis [82] compares metrics of four operating system kernels: Windows, Linux, FreeBSD and OpenSolaris. For each metric, box plots of the four kernels are put side-by-side showing the smallest observation, lower quantile, median, mean, higher quantile, highest observation and identify outliers. The box-plots are then analyzed by the author in a way which associates ranks, $+$ or $-$, to each kernel. However, as the author states, ranks are given subjectively.

Vasa et al. [87] propose the use of Gini coefficients to summarize a metric distribution across a system. Their analysis of the Gini coefficient for 10 class-level metrics using 50 Java and C# systems reveals that most of the systems have common values. Moreover, higher Gini values indicate problems and, when analyzing subsequent releases of source code, a difference higher than 0.04 indicates significant changes in the code.

Finally, several studies show that different software metrics follow power law distributions [24, 62, 89]. Concast et al. [24] show that for a large Smalltalk system most Chidamber and Kemerer metrics [22] follow power laws. Louridas et al. [62] show that the dependencies of different software artifacts also follow power laws. Wheeldon et al. [89] show that different class relationships follow power laws distributions.

All such data analyses studies clearly demonstrate that metrics do not follow normal distributions, invalidating the use of any statistical technique assuming a normal distribution. However the same studies fall short in concluding how to use these distributions, and the coefficients of the distributions, to establish baseline values to judge systems. Moreover, should such baseline values be established it would not be possible to identify the code responsible for deviations (there is no traceability of results).

In contrast, the method proposed in this chapter is focused on defining thresholds with direct application to differentiate software systems, judge quality and pinpoint problems.

2.8 Summary

Contributions A novel method for deriving software metric thresholds was proposed. The used strategy improves over others by fulfilling three fundamental requirements: *(i)* it is based on data analysis from a representative set of systems (benchmark); *(ii)* it respects the statistical properties of the metric, such as metric scale and distribution; *(iii)* it is repeatable, transparent and straightforward to carry out. These requirements were achieved by aggregating measurements from different systems using relative size weighting. The proposed method was applied to a large set of systems and thresholds were derived by choosing specific percentages of overall code of the benchmark.

Discussion A new method for deriving thresholds was explained in detail using as example the McCabe metric and a benchmark of 100 OO systems (C# and Java), both proprietary and OSS. it was shown that the distribution of the metric is preserved and that the method is resilient to the influence of large systems or outliers. Thresholds were derived using 70%, 80% and 90% quantiles and checked against the benchmark to show that thresholds indeed represent these quantiles. The analysis of these results was replicated with success using four other metrics from the SIG quality model. Variants in the method were analyzed as well as threats to the overall methodology.

The method has proven effective in deriving thresholds for all the metrics of the SIG quality model. For unit interfacing the 80%, 90% and 95% triple was considered since the metric variability only increases much later than for other metrics whose thresholds are 70%, 80% and 90%. For all metrics, the method shows that the derived thresholds are representative of the chosen quantiles.

Industrial applications Thresholds derived with the method introduced in this chapter have been successfully put into practice by SIG for software analysis [41], benchmarking [26] and certification [27]. Thresholds that were initially defined based on expert opinion have been replaced by the derived thresholds and have been used with success.

This method has also been applied to other metrics. Luijten et al. [63] found empirical evidence that systems with higher technical quality have higher issue solving efficiency. The thresholds used for classifying issue efficiency were derived using the methodology described in this chapter.

Chapter 3

Benchmark-based Aggregation of Metrics to Ratings

Software metrics have been proposed as instruments, not only to guide individual developers in their coding tasks, but also to obtain high-level quality indicators for entire software systems. Such system-level indicators are intended to enable meaningful comparisons among systems or to serve as triggers for a deeper analysis.

Common methods for aggregation range from simple mathematical operations (e.g. addition [58, 22] and central tendency [82, 23]) to more complex techniques such as distribution fitting [24, 89, 62], wealth inequality metrics (e.g. Gini coefficient [87] and Theil Index [78]) and custom formulae [50]. However, these methodologies provide little guidance for interpreting the aggregated results and tracing back to the individual measurements. To resolve such limitations, Heitlager et al. [41] proposed a two-stage rating approach where (i) measurement values are compared to thresholds and summarized into risk profiles, and (ii) risk profiles are mapped to ratings.

This chapter extends the technique for deriving risk thresholds from benchmark data, presented in Chapter 2, into a methodology for benchmark-based calibration of two-stage aggregation of metrics into ratings. The core algorithm behind this process will be explained, together with a demonstration of its application to various metrics of the Software Improvement Group (SIG) quality model, using a benchmark of 100 software systems. The sensitivity of the algorithm to the underlying data will also be addressed.

3.1 Introduction

Software metrics have been proposed to analyze and evaluate software by quantitatively capturing a specific characteristic or view of a software system. Despite much research, the practical application of software metrics remains challenging.

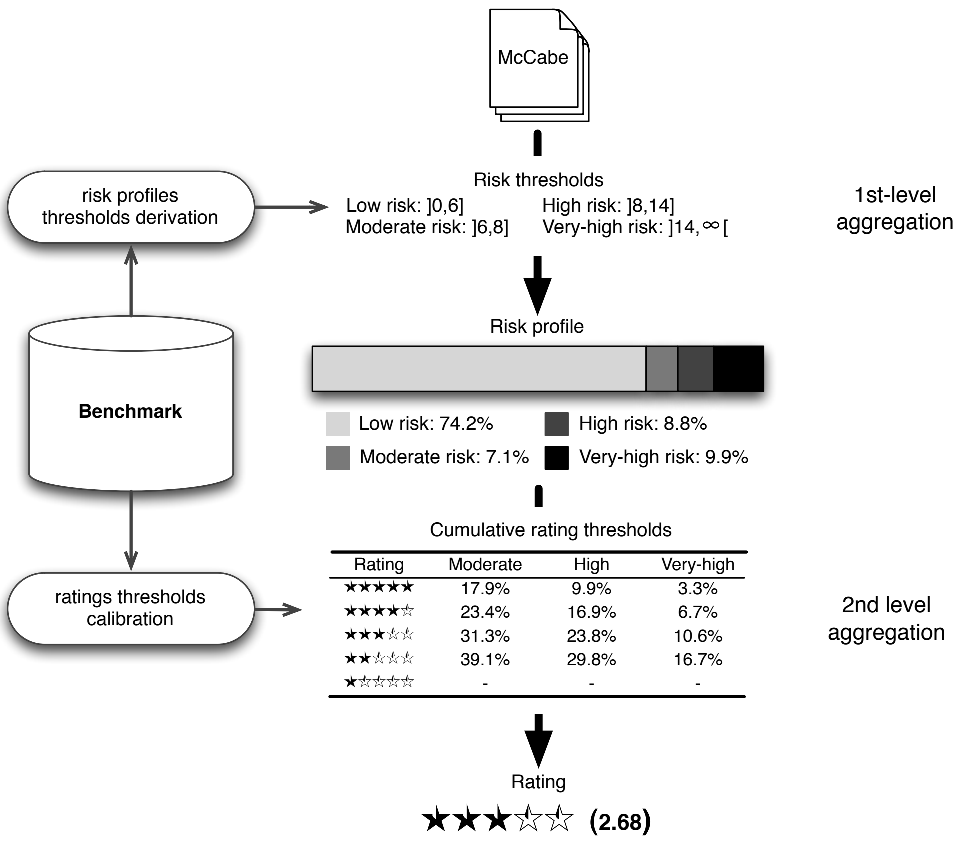
One of the main problems with software metrics is how to aggregate individual measurements into a single value capturing information of the overall system. This is a general problem, as noted by Concas et al. [24], since most metrics do not have a definition at system-level. For instance, the McCabe metric [67] has been proposed to measure complexity at unit level (e.g. method or function). The use of this metric, however, can easily generate several thousands of measurements which will be difficult to analyze in order to arrive to a judgement about how complex the overall system is.

Several approaches have been proposed for measurement aggregation but they suffer from several drawbacks. For instance, mathematical addition is meaningless for metrics such as e.g. Coupling between Object Classes [22]; central tendency measures often hide underlying distributions; distribution fitting and wealth inequality measures (e.g Gini factor [87] and Theil Index [78]) are hard to interpret and their result difficult to trace back to the metric measurements; custom formulae are hard to validate. In sum, these approaches lack the ability of aggregating measurements into a meaningful result that: *i)* is easy to explain and interpret; *ii)* is representative of real systems allowing comparison and ranking, and *iii)* captures enough information to enable traceability to individual measurements allowing to pinpoint problems.

An alternative approach to aggregate metrics is to use thresholds to map measurements to a particular scale. This was first introduced by Heitlager et al. [41] and later demonstrated by Correia and Visser [27] to certify software systems. Measurements are aggregated to a star-rating in a two-step process. First, thresholds on metrics are used to aggregate individual measurements into risk profiles. Second, rating thresholds are used to map risk profiles into a 5-point rating scale. When this method of aggregation was proposed in [41] and [27] both 1st and 2nd-level thresholds were based on experience. Later, a technique for deriving metric thresholds (1st-level) from benchmark data was proposed by the author and others in [6] and explained in Chapter 2. In the present chapter, a methodology is proposed to calibrate rating thresholds (2nd-level).

This approach for aggregating individual measurements into an N-point rating scale based on thresholds relies on a novel algorithm that calibrates a set of thresholds per rating based on benchmark data, chained with another algorithm which calculates ratings based on those thresholds. The algorithm is applied to all metrics of the SIG quality model using an industry-based benchmark of 100 systems and an analysis of the sensitivity of these thresholds to the underlying data is performed. Justification for this algorithm is provided and various choices made in its design are discussed. The ratings are easy to explain and interpret, representative of an industry benchmark of software systems and enable traceability back to individual measurements.

This chapter is structured as follows. Section 3.2 provides a high-level overview of the process, explaining how measurements are aggregated to a rating, how the rating can be interpreted and its meaning traced back to individual measurements. Section 3.3 defines both the algorithm to calibrate rating thresholds and the algorithm that uses the thresholds to calculate ratings. Section 3.4 provides further explanation of the calibration algorithm and investigates possible alternatives. Section 3.5 demonstrates the applicability of rating calibration to the metrics of the SIG quality model using the benchmark already introduced in Section 1.4. Section 3.6 provides an analysis of algorithm stability with respect to the used data. Section 3.7 discusses related methodologies to aggregate measurements. Finally, Section 3.8 presents a summary of contributions.



Rating	Moderate	High	Very-high
★★★★★	17.9%	9.9%	3.3%
★★★★☆	23.4%	16.9%	6.7%
★★★☆☆	31.3%	23.8%	10.6%
★★☆☆☆	39.1%	29.8%	16.7%
★☆☆☆☆	-	-	-

Figure 3.1: Process overview of the aggregation of code-level measurements to system-level ratings, using as example the McCabe complexity metric for ArgoUML 0.29.4. In the 1st-level aggregation, thresholds are used to define four ranges and classify measurements into four risk categories: Low, Moderate, High and Very-high. The categorized measurements will then be used to create a risk profile, which represents the percentage of volume of each risk category. In the 2nd-level aggregation, risk profiles are aggregated into a star-rating using 2nd-level thresholds (depicted with a table). The ArgoUML rating is 3 out of 5 stars (or 2.68 stars in a continuous scale). 1st-level thresholds are derived from a benchmark using the technique explained in Chapter 2. The calibration of 2nd-level thresholds from a benchmark is explained in this chapter.

3.2 Approach

Figure 3.1 presents an overview of the approach to aggregate measurements to ratings using benchmark-based thresholds. This section explains how to aggregate measurements to ratings using thresholds and trace back individual measurements from ratings. The McCabe metric was chosen as example since it is a very well known source code metric. ArgoUML was chosen since it is, probably, one of the most studied projects in software engineering research.

3.2.1 Aggregation of measurements to ratings

The aggregation of individual measurements to ratings is a two-level process based on two types of thresholds, as illustrated in Figure 3.1.

First, individual measurements are aggregated to risk profiles using metric thresholds [41, 6]. A risk profile represents the percentage of overall code that falls into each of the four risk categories: Low, Moderate, High and Very-high. Throughout this work, we will refer to the aggregation of measurements to risk profiles as *1st-level aggregation*, and to the thresholds used in this process as *1st-level thresholds*. 1st-level thresholds can be derived from a benchmark by a methodology previously presented in Chapter 2.

Second, risk profiles are aggregated to a 5-point star scale using rating thresholds. Each rating is calibrated to represent a specific percentage of systems in the benchmark. Throughout this work, we will refer to the aggregation of risk profiles to a star rating as *2nd-level aggregation*, and to the thresholds used in this process as *2nd-level thresholds*. The calibration of 2nd-level thresholds from a benchmark will be introduced in Section 3.3.

1st-level aggregation

The aggregation of individual measurements to risk profiles using 1st-level thresholds is done by computing the relative size of the system that falls into each risk category. Size is measured using the Source Lines of Code (SLOC) metric.

Since a risk profile is composed of four categories, we need four intervals to classify all measurements. The intervals defining these categories for the McCabe metrics are shown in Figure 3.1 and are represented using the ISO/IEC 80000–2 notation [48]. These intervals were defined with three thresholds, 6, 8 and 14, representing the upper bound of the Low, Moderate and High risk categories, respectively. The thresholds were derived with the methodology presented in Chapter 2 from a benchmark of 100 systems, representing 70%, 80% and 90% of the benchmark code, respectively.

To compute a risk profile for the ArgoUML system, we use the intervals shown in Figure 3.1 to categorize all the methods into four risk categories. Then, for each category we sum the size of all those methods and then divide them by the overall size of the system, resulting in the relative size (or percentage) of the system that falls into each risk category. For instance, the Low risk category of ArgoUML is computed by considering all methods that have a McCabe value that fall in the $]0, 6]$ interval, i.e., all methods that have a McCabe value up to 6. Then, we sum the SLOC of all those methods ($95, 262$) and divide by the overall size[1] of ArgoUML ($128, 316$), resulting in a total of 74.2%. The risk profile for ArgoUML is depicted in Figure 3.1 containing 74.2% of code in the Low risk, 7.1% for Moderate risk, 8.8% for High risk, and 9.9% for Very-high risk.

2**nd-level aggregation**

The aggregation of risk profiles into a rating is done by determining the minimum rating for which the cumulative relative size of all risk profile categories does not exceed a set of 2nd-level thresholds.

Since we are using a 5-point star rating scale, a minimum of 4 sets of thresholds defining the upper-bound are necessary to cover all possible risk profile values[2]. Each set of thresholds defines the cumulative upper-boundaries for the Moderate, High and Very-high risk categories. The cumulative upper-boundary for a category takes into account the volume of code for that category plus all higher categories (e.g. the cumulative upper-boundary for Moderate risk takes into account the percentage of volume of the Moderate, High and Very-high categories of a risk profile). Note that, since the cumulative Low risk category will always be 100% there is no need to specify thresholds for it. Figure 3.1 shows a table containing the 2nd-level thresholds for the McCabe metric, calibrated with the algorithm introduced in this chapter. These ratings

[1]For the analysis of ArgoUML only production code was considered - test code was not included in these numbers since it hinders the overall complexity of the system.

[2]For an N-point scale a minimum of $N - 1$ sets of thresholds are needed.

were calibrated for a 5-point scale with a distribution 20–20–20–20–20, meaning that each star represents equally 20% of the systems in the benchmark. The distribution of the 2nd-level thresholds is an input of the calibration algorithm and will be detailed later, in Section 3.3.

To determine the rating for ArgoUML, we first calculate the cumulative risk profile, i.e., the cumulative relative size for the Moderate, High and Very-high risk categories. This is done by considering the relative size of the each risk category plus all higher categories, resulting in 25.8% for Moderate risk, 18.7% for High risk, and 9.9% for the Very-high risk. These values are then compared to the McCabe rating thresholds, shown in Figure 3.1, and a rating of 3 stars is obtained. Using an interpolated function results in a rating value of 2.68. The rating for ArgoUML is depicted in Figure 3.1: the stars in black depict the rating, and the stars in white represents the scale. Since the rating thresholds were calibrated with a 20–20–20–20–20 distribution, a rating of 3 stars indicates that ArgoUML has average quality, meaning that there are 40% of systems that are better and 40% of systems that are worse. The functions to calculate ratings are defined in Section 3.3.2.

3.2.2 Ratings to measurements traceability

The traceability of the 3 star rating back to individual measurements is achieved by using again the 2nd and 1st-level thresholds. This traceability is important not only to be able to explain the rating, but also to gain information about potential problems which can then be used to support decisions.

Let us then try to understand why ArgoUML rated 3 stars. This can be done by comparing the values of the risk profile to the table defining the 2nd-level thresholds presented in Figure 3.1. By focusing on the Very-high risk category, for instance, we can see that ArgoUML has a total of 9.9% of code which is limiting the rating to 3 stars. By looking at the intervals defining the 1st-level thresholds we can see that methods with a McCabe value higher than 14 are considered Very-high risk. The use of 1st and 2nd-level thresholds allow us to identify the ArgoUML methods responsible for limiting the ArgoUML rating to 3 stars, calling for further investigation to determine if these methods are indeed problematic.

This traceability approach can also be used to support decision making. Let us consider a scenario where we want to improve ArgoUML from a rating of 3 to 4 stars. In order for ArgoUML to rate 4 stars, according to 2nd-level thresholds shown in Figure 3.1, it should have a maximum of 6.7% of the code in the Very-high risk category, 16.9% in High and Very-high risk categories, and 23.4% of code in Moderate, High and Very-high risk categories. Focusing in the Very-high risk category, the ArgoUML rating can improve by reducing the percentage of code in that category from 9.9% (current value) to 6.7% (maximum allowed value). Hence, improving the ArgoUML rating from 3 to 4 stars amounts to fixing 3.2% of the overall code. This can be achieved by refactoring methods with a McCabe higher than 14 to a maximum McCabe value of 6. Of course, it might not be feasible to refactor such methods for having a maximum McCabe value of 6. In this case, the rating will remain unchanged requiring extra refactoring effort to reduce the code in the higher risk categories. For instance, if we were only able to refactor 3.2% of the code classified as Very-high risk to High risk (with a maximum McCabe value of 14), this would change the ArgoUML risk profile in the following way: the Very-high risk category would decrease from 9.9% to 6.7%, and the High risk category would increase from 8.8% to 12%, while all other categories remain unchanged. Although the percentage of code in the High risk category increases, the cumulative value in the High risk is still the same[3] as the code just moves from Very-High to High risk, accounting for 18.7%. Since the cumulative value in the High risk category is higher than the threshold for 4 stars (16.9%) the rating will remain unchanged. The use of cumulative thresholds will be further justified in Section 3.4.4. A higher rating can still be achieved, albeit with extra effort, by refactoring code that is considered High risk into code that is considered Moderate risk.

[3]Not only the cumulative value of the High risk is the same, but also the cumulative value of all lower risk categories will remain unchanged.

Algorithm 1 Ratings calibration algorithm for a given N-point partition of systems.

```
Require: $riskprofiles : (Moderate \times High \times VeryHigh)^*, partition^{N-1}$
 1: $thresholds \leftarrow []$
 2: $ordered[Moderate] \leftarrow sort(riskprofiles.Moderate)$
 3: $ordered[High] \leftarrow sort(riskprofiles.High)$
 4: $ordered[VeryHigh] \leftarrow sort(riskprofiles.VeryHigh)$
 5: for $rating = 1$ to $(N-1)$ do
 6:     $i \leftarrow 0$
 7:     repeat
 8:         $i \leftarrow i + 1$
 9:         $thresholds[rating][Moderate] \leftarrow ordered[Moderate][i]$
10:         $thresholds[rating][High] \leftarrow ordered[High][i]$
11:         $thresholds[rating][VeryHigh] \leftarrow ordered[VeryHigh][i]$
12:     until $distribution(riskprofiles, thresholds[rating]) \geq partition[rating]$ or $i = length(riskprofiles)$
13:     $index \leftarrow i$
14:     for all $risk$ in $(Moderate, High, VeryHigh)$ do
15:         $i \leftarrow index$
16:         $done \leftarrow False$
17:         while $i > 0$ and not $done$ do
18:             $thresholds.old \leftarrow thresholds$
19:             $i \leftarrow i - 1$
20:             $thresholds[rating][risk] \leftarrow ordered[risk][i]$
21:             if $distribution(riskprofiles, thresholds[rating]) < partition[rating]$ then
22:                 $thresholds \leftarrow thresholds.old$
23:                 $done \leftarrow True$
24:             end if
25:         end while
26:     end for
27: end for
28: return $thresholds$
```

3.3 Rating Calibration and Calculation Algorithms

In short, what the calibration algorithm does is to take the risk profiles for all systems in a benchmark, and search for the minimum thresholds that can divide those systems according to a given distribution (or system partition). This section provides the definition and explanation of the algorithm which calibrates thresholds for an N-point scale rating, and the algorithm to use such thresholds for ratings calculation.

3.3.1 Ratings calibration algorithm

The algorithm to calibrate N-point ratings is presented in Algorithm 1. It takes two arguments as input: cumulative risk profiles for all the systems in the benchmark, and a partition defining the desired distribution of the systems per rating. The cumulative risk profiles are computed using the 1st-level thresholds, as specified before, for each individual system of the benchmark. The partition, of size $N - 1$, defines the number of systems for each rating (from the highest to the lowest rating). As example, for our benchmark of 100 systems and a 5-point rating with uniform distribution, each rating represents equally 20% of the systems and thus the partition is 20–20–20–20–20.

The algorithm starts, in line 1, by initializing the variable $thresholds$ which will hold the result of the calibration algorithm (the rating thresholds). Then, in lines 2–4, each risk category of the risk profiles is ordered and saved as a matrix in the $ordered$ variable. The columns of the matrix will be the three risk categories and the lines will represent the values for the risk categories of the benchmark. This matrix plays an important role, since each position will be iterated in order to find thresholds for each rating.

The main calibration algorithm, which executes for each rating, is defined in lines 5–27. The algorithm has two main parts: finding an initial set of thresholds that fulfills the desired number of systems for that rating (lines 7–12), and an optimization part, which is responsible for finding the smallest possible thresholds for the three risk categories (lines 13–26).

Finding an initial set of thresholds is done in lines 7–12. The index is incremented by one for iterating through the ordered risk profiles (line 8). Then, thresholds are set from the values of the ordered risk profiles for that index (lines 9–11). Two conditions are verified (line 12): first if the current thresholds can identify at least as many systems as specified for that specific rating; second, if the counter i is not out of bounds, i.e., if counter has not exceeded the total number of systems. What the first condition does is to check if the combination of three thresholds allows to identify the specified number of systems. However this condition is not sufficient to guarantee that all three thresholds are as strict as possible.

To guarantee that all three thresholds are as strict as possible, the optimization part (lines 13–26) is executed. In general, the optimization tries, for each risk category, to use smaller thresholds while preserving the same distribution. The optimization starts by saving the counter i containing the position of the three thresholds previously found (line 13) which will be the starting point to optimize the thresholds of each risk category. Then, for each risk category, the optimization algorithm is executed (lines 14–26). The counter is initiated to the position of the three thresholds previously found (line 15). The flag $done$, used to stop the search loop, is set to $False$ (line 16). Then, while the index i is greater than zero (the index does not reach the beginning of the ordered list) and the flag is not set to true, it performs a search for smaller thresholds (lines 17–25). This search first saves the previously computed thresholds in the $thresholds.old$ variable (line 18). Then, it decreases the counter i by one (line 19) and sets the threshold for the risk category currently under optimization (line 20). If the intended distribution is not preserved (line 21), then it means that the algorithm went one step to far. The current thresholds are replaced with the previously saved thresholds $thresholds.old$ (line 22) and the flag $done$ is set to $True$ to finalize the search (line 23). If the intended distribution is still preserved, then the search continues.

The algorithm finishes (line 28) by returning, for each risk profile category, $N - 1$ thresholds. These thresholds define the rating maximum values for each risk category. The lowest rating is attributed if risk profiles exceed the thresholds calibrated by the algorithm.

3.3.2 Ratings calculation algorithm

Ratings can be represented in both discrete and continuous scales. A discrete scale is achieved by comparing the values of a risk profile to thresholds. A continuous scale is achieved by using an interpolation function among the values of the risk profiles and the lower and upper thresholds. Below, we will provide the algorithm and explanation how to compute ratings for both scales.

Discrete scale

The calculation of a discrete rating, for an N-point scale, is done by finding the set of minimum thresholds such that these thresholds are higher or equal to the values of risk profiles, and then from the order of such thresholds deriving the rating. This calculation is formally described as follows.

$$RP_{M \times H \times VH} \times \begin{vmatrix} T_{M_1} & T_{H_1} & T_{VH_1} \\ T_{M_2} & T_{H_2} & T_{VH_2} \\ \ldots & \ldots & \ldots \\ T_{M_{N-1}} & T_{H_{N-1}} & T_{VH_{N-1}} \end{vmatrix} \rightharpoonup R$$

Meaning that a rating $R \in \{1, N\}$, is computed from a risk profile RP and a set of $N - 1$ thresholds, such that:

$$R = N - min(I_M, I_H, I_{VH}) + 1$$

The rating R is determined by finding the minimum index I of each risk profile category, and then adjusting that value for the correct order. Since the thresholds are placed in ascending order (from low values to higher values) representing ratings in descending order (from higher rating to lower rating) we need to adjust the value of the index to a rating. For instance, if the minimum index is 1 the rating should be N. The index for each risk category is determined as follows:

$$
\begin{aligned}
I_M &= min_I(RP_M \leq T_{M_I}) \\
I_H &= min_I(RP_H \leq T_{H_I}) \\
I_{VH} &= min_I(RP_{VH} \leq T_{VH_I})
\end{aligned}
$$

The index for each risk category is determined by finding the position of the lowest thresholds such that the value of the risk category is lower than or equal to the threshold. For the case that all $N - 1$ thresholds are lower than the value in the risk category, then the index will be equal to N.

Continuous scale

A continuous scale can be obtained using the linear interpolation function of Equation 3.1, which is parametric on the discrete rating and the lower and upper thresholds for the risk profile:

$$
s(v) = s_0 + 0.5 - (v - t_0)\frac{1}{t_1 - t_0} \tag{3.1}
$$

where

$s(v)$ Final continuous rating.

v Percentage of volume in the risk profile.

s_0 Initial discrete rating.

t_0 Lower threshold for the risk profile.

t_1 Upper threshold for the risk profile.

The final interpolated rating $R \in \{0.5, N + 0.5\}$, for N-point scale is then obtained by taking the minimum rating of all risk categories, defined as follows:

$$
R = min\,(s(RP_M), s(RP_H), s(RP_{VH}))
$$

The choice of range from 0.5 to $N + 0.5$ is so that the number of stars can be calculated by standard, round half up, arithmetic rounding. Note that it is possible, in an extreme situation, to achieve a maximum continuous rating of $N + 0.5$. This implies that, for instance in a 5-point scale, a continuous rating value of 5.5 is likely. Hence, when converting a continuous to a discrete rating, this situation should be handled by truncating the value instead of rounding it.

3.4 Considerations

Previous sections deferred the discussion of details of the algorithm and implicit decisions. In this section we provide further explanation about rating scales, distributions, use of cumulative risk profiles and data transformations.

3.4.1 Rating scale

The rating scale defines the values to which the measurements will be mapped. Section 3.2 proposes the use of a 5-point scale represented using stars, 1 star representing the lowest value and 5 stars the highest value.

The use of a 5-point scale can be found in many other fields. An example from social sciences is the Likert scale [60] used for questionnaires.

The calibration algorithm presented in Section 3.3 calibrates ratings for an N-point scale. Hence, 3-point or 10-point scales could be used as well. However, a scale with a small number of points might not discriminate enough (e.g. 1 or 2 points), and a scale with many points (e.g. 50) might be too hard to explain and use. Also, for an N-point scale a minimum of N systems in the benchmark is necessary. Nevertheless, in order to ensure that the thresholds are representative, the larger the number of systems in the benchmark the better.

3.4.2 Distribution/Partition

A distribution defines the percentage of systems of a benchmark that will be mapped to each rating value. A partition is similar, but it is instantiated for a given benchmark, defining the number of systems per rating.

Section 3.2 proposes the use of a uniform distribution, meaning that each rating represents an equal number of systems. Using a 5-point scale, the distribution will be 20–20–20–20–20, indicating that each star represents 20% of the systems in the benchmark. A uniform distribution was chosen for the sake of simplicity, as the calibration algorithm works for any given partition. For instance, it is possible to calibrate ratings for a 5–30–30–30–5 distribution, as proposed by Baggen et al. [11], or for a normal-like distribution (e.g. 5–25–40–25–5), resulting in a different set of rating thresholds. However changing neither the aggregation or traceability methodology, the choice of partition might influence the results when using ratings for empirical validation.

3.4.3 Using other 1st-level thresholds

An essential part of the aggregation of individual metrics to ratings is to compute risk profiles from 1st-level thresholds. A method for deriving 1st-level thresholds from benchmark data has been proposed previously in Chapter 2 and both Sections 3.2 and 3.5 use the thresholds presented in that chapter. However, the calibration algorithm presented in Section 3.3 does not depend on those specific thresholds.

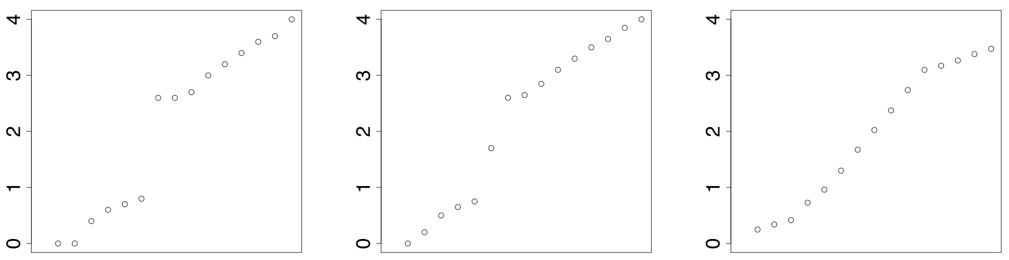
The requirement for the 1st-level thresholds is that they should be valid for the benchmark data that is used. By valid it is meant that the existing systems should have measurements both higher and lower than those thresholds. Calibration of rating thresholds for the McCabe metric with 1st-level thresholds 10, 20 and 50 was attempted with success. However, if the chosen 1st-level thresholds are too high, the calibration algorithm will not be able to guarantee that the desired distribution will be met. Furthermore, the calibration of rating thresholds is independent of the metric distribution. This chapter only uses metrics with an exponential distribution and for which high values indicate higher risk. However, the calibration of ratings for metrics with different distributions was achieved with success. Chapter 4 will show the calibration of test coverage, which has a normal-like distribution and for which low values indicate higher risk.

3.4.4 Cumulative rating thresholds

Cumulative rating thresholds were introduced in Sections 3.2 and 3.3 but a more detailed explanation of their use was deferred. Cumulative thresholds are necessary to avoid problems arising from the values of the risk profile being very close to the thresholds.

As an example, we will use the rating thresholds for the McCabe metric (presented in Section 3.2) in two scenarios: cumulative and non-cumulative. In the first scenario, let us assume that a given system has a cumulative risk profile of 16.7% in the High risk category and 6.7% in the Very-high risk category. In this boundary situation, according to the McCabe rating thresholds (16.9% and 6.7%, for High and Very-high risk, respectively), the system will rate 4 stars. Now let us assume that, by refactoring, we move 1% of code from the Very-high risk to High risk category. In the risk profile, the Very-high risk category will decrease from 6.7% to 5.7% and the High risk category will remain the same (since it is cumulative, the decrease of the Very-high risk category is cancelled by the increase in the High risk category). After the refactoring, although the complexity decreases, the rating remains unchanged as expected.

In the second scenario, let us assume the use of non-cumulative rating thresholds. The non-cumulative risk profile, for the two highest categories, for the same system is then 10% for the High risk and 6.7% for the Very-high risk. The rating thresholds will be non-cumulative as well, being 10.2% for the High risk and 6.7% for the Very-high risk categories. Performing the same refactoring, where 1% of the code is moved from the Very-high risk to the High risk category, will decrease the Very-high risk code from 6.7% to 5.7% but increase the High-risk code from 10% to 11%. With the



(a) None (b) Interpolation (c) Moving mean

Figure 3.2: Example effect of data transformations. The y-axis represents the percentage of code in the very-high risk category and the x-axis represents 15 systems of the benchmark.

non-cumulative thresholds, since the High risk code now exceeds the thresholds the system rating will decrease from 4 to 3 stars, contradicting the expectations. Having observed this in practice, it was decided to introduce cumulative thresholds to prevent the decrease of rating in cases where there is an improvement in quality.

3.4.5 Data transformations

One potential threat to validity in this approach is the risk of over-fitting a particular set of systems. Namely, the specific thresholds obtained can be conditioned by particular discontinuities in the data. In order to reduce this effect, smoothing transformations can be applied to each risk category where thresholds are picked out from.

Two transformations were experimented: (i) interpolation, in which each data point is replaced by the interpolation of its value and the consecutive one, and (ii) moving mean, in which each data point is replaced by the mean of a window of points around it (this example used a 2-point window).

The effect of each transformation is shown in Figure 3.2, using a part of the Very-high risk category for the Module Inward Coupling metric where a discontinuity can be observed in the raw data. Other transformations could be used and easily added to the algorithm. In practice, due to the large number of data points used, no significant differences in the calibrated thresholds were observed. Also, no other issues were observed with the use of thresholds from transformed data.

When using a small number of data points (benchmark with few systems) there might be many discontinuities and data transformations might be relevant to smooth the data. In theory, by smoothing the data we are compensating for the lack of more systems, hence reducing the risk of over-fitting a particular set of systems. Smoothing the data requires only minimal changes to the calibration algorithm, the ratings calculation algorithm and the traceability capabilities being the same. However, other implications of this approach are still open for research.

3.4.6 Failing to achieve the expected distribution

When calibrating rating thresholds for some metrics, it was observed in practice cases where the final distribution differs from the expected distribution.

Small differences, where the partition of a rating contains one or two more systems than expected, might be due to the existence of ties. A tie exists when two or more systems have the same values for the Moderate, High and Very-high risk categories and hence it is not possible to distinguish them. If the set of chosen thresholds matches systems with ties, it is likely that the final distribution will differ the expected one. For small differences, this situation it is not problematic and the calibration thresholds can be used anyway. However, the presence of ties should be investigated because it might indicate the presence of the same system repeated or different releases of the same system which are very similar.

Big differences, where there is an unbalanced distribution with too many systems calibrated for one rating and too few for other ratings, might be due to the use of wrong 1st-level thresholds. An arbitrary choice of 1st-level thresholds might not allow to differentiate systems (e.g. choosing thresholds higher than those of the benchmark) and hence the final calibration will differ from the expected. This situation can be solved by choosing a different set of thresholds or using the method presented in Chapter 2 to derive thresholds.

Big differences between the final and expected distributions might also be due to the metric properties themselves causing a large number of ties. This situation might be solved by choosing a rating scale with a smaller number of points such as e.g. 3-point rating instead of a 5-point rating. In an extreme case, where there are still big differences for a rating scale with a small number of points, this might indicate that the metric does not capture enough information to differentiate software systems.

3.5 Application to the SIG quality model metrics

Using the benchmark described in Section 1.4, the methodology to calibrate rating thresholds was successfully applied to all metrics of the SIG quality model [41]: unit complexity (McCabe at method level), unit size (SLOC at method level), unit interfacing (number of parameters at method level) and module inward coupling (Fan-in [42] at file level). This section discusses the calibrated thresholds.

Table 3.1 presents both the 1st-level thresholds derived as in Chapter 2, and the 2nd-level thresholds calibrated with the algorithm presented in Section 3.3. The metric thresholds for the unit complexity were previously presented in Section 3.2.

As we can observe, all the thresholds are monotonic between risk categories and ratings, i.e., thresholds become more lenient from high to lower risk categories and from higher to lower ratings. Moreover, there are no repeated thresholds for all metrics. This indicates that the calibration algorithm was successful in differentiating the benchmark systems and consequently deriving good rating thresholds.

For all metrics, all benchmark systems were rated from the calibrated rating thresholds. The idea was to verify if the expected distribution, 20–20–20-20–20, was in fact met by the calibration algorithm. For all metrics we verified that the expected distribution was achieved with no deviations.

Table 3.1: Risk and rating thresholds for the SIG quality model metrics. Risk thresholds are defined in the headers, and rating thresholds are defined in the table body.

(a) Unit Size metric (SLOC at method level).

Star rating	Low risk $]0, 30]$	Moderate risk $]30, 44]$	High risk $]44, 74]$	Very-high risk $]74, \infty[$
★★★★★	-	19.5	10.9	3.9
★★★★☆	-	26.0	15.5	6.5
★★★☆☆	-	34.1	22.2	11.0
★★☆☆☆	-	45.9	31.4	18.1

(b) Unit Complexity metric (McCabe at method level).

Star rating	Low risk $]0, 6]$	Moderate risk $]6, 8]$	High risk $]8, 14]$	Very-high risk $]14, \infty[$
★★★★★	-	17.9	9.9	3.3
★★★★☆	-	23.4	16.9	6.7
★★★☆☆	-	31.3	23.8	10.6
★★☆☆☆	-	39.1	29.8	16.7

(c) Unit Interfacing metric (Number of parameters at method level).

Star rating	Low risk $[0, 2]$	Moderate risk $[2, 3[$	High risk $[3, 4[$	Very-high risk $[4, \infty[$
★★★★★	-	12.1	5.4	2.2
★★★★☆	-	14.9	7.2	3.1
★★★☆☆	-	17.7	10.2	4.8
★★☆☆☆	-	25.2	15.3	7.1

(d) Module Inward Coupling metric (Fan-in at file level).

Star rating	Low risk $[0, 10]$	Moderate risk $[10, 22[$	High risk $[22, 56[$	Very-high risk $[56, \infty[$
★★★★★	-	23.9	12.8	6.4
★★★★☆	-	31.2	20.3	9.3
★★★☆☆	-	34.5	22.5	11.9
★★☆☆☆	-	41.8	30.6	19.6

3.6 Stability analysis

In order to assess the reliability of the obtained thresholds, a *stability analysis* was performed. The general approach is to run the calibration algorithm n times, each with a randomly sampled subset of the systems present in the original set. The result is n threshold tables per metric. Two ways of assessing these numbers make sense: (i)

Table 3.2: Variability of the rating thresholds for 100 runs, randomly sampling 90% of the systems in the benchmark.

(a) Unit Size metric.

Star rating	Moderate risk	High risk	Very-high risk
★★★★★	18.5 - 20.6	8.9 - 11.1	3.7 - 3.9
★★★★☆	24.6 - 28.2	14.4 - 18.0	5.8 - 7.8
★★★☆☆	33.5 - 35.9	21.1 - 26.0	10.0 - 12.7
★★☆☆☆	43.2 - 46.4	30.0 - 33.3	17.3 - 19.5

(b) Unit Complexity metric.

Star rating	Moderate risk	High risk	Very-high risk
★★★★★	17.3 - 20.0	9.8 - 12.3	3.2 - 4.2
★★★★☆	23.5 - 25.5	16.1 - 18.9	6.2 - 8.5
★★★☆☆	29.5 - 32.9	20.8 - 24.8	9.7 - 12.6
★★☆☆☆	35.9 - 40.9	28.0 - 30.8	14.5 - 17.1

(c) Unit Interfacing metric.

Star rating	Moderate risk	High risk	Very-high risk
★★★★★	11.1 - 13.0	4.7 - 5.7	2.0 - 2.3
★★★★☆	14.8 - 15.7	6.9 - 7.6	2.8 - 3.5
★★★☆☆	17.2 - 21.2	8.3 - 10.2	4.5 - 5.0
★★☆☆☆	25.2 - 27.6	12.8 - 18.0	6.1 - 7.1

(d) Module Inward Coupling metric.

Star rating	Moderate risk	High risk	Very-high risk
★★★★★	23.0 - 26.0	12.8 - 15.1	5.7 - 7.4
★★★★☆	29.3 - 31.6	18.6 - 20.7	8.4 - 10.0
★★★☆☆	34.5 - 36.9	21.9 - 23.7	10.1 - 13.3
★★☆☆☆	41.5 - 48.7	28.9 - 36.4	15.1 - 20.7

inspect the variability in the threshold values; (ii) apply the thresholds to the original set of systems and inspect the differences in ratings.

Using, again, the benchmark presented in Section 1.4, 100 runs ($n = 100$) were performed, each of these with 90% of the systems (randomly sampled).

To assess the stability of the rating thresholds, it was calculated for each threshold the absolute relative differences from its median throughout all the runs[4]. This amounts

[4]Thus, for a threshold t_i (calculated in run number $i \in [1, n[$) one has $\delta t_i = \frac{|t_i - median(t_n)|}{median(t_n)}$.

Table 3.3: Summary statistics on the stability of rating thresholds.

	Q_1	Median	Q_3	95%	Max	μ
Unit size	0.0	0.4	3.2	10.7	23.8	2.6
Unit complexity	0.0	0.3	4.4	11.7	27.3	3.2
Unit interfacing	0.0	0.0	5.5	14.1	25.9	3.3
Module coupling	0.0	0.0	5.3	18.0	23.4	3.3

Table 3.4: Summary statistics on stability of the computed ratings.

	Q_1	Median	Q_3	95%	Max	μ
Unit size	0.0	0.3	1.0	3.5	7.9	0.8
Unit complexity	0.0	0.3	1.5	4.3	9.7	1.0
Unit interfacing	0.0	0.4	1.5	5.1	12.1	1.1
Module coupling	0.0	0.4	1.4	5.9	14.3	1.2

to 12 threshold values per run, thus $100 \times 12 = 1200$ data points. Table 3.2 presents the obtained threshold ranges for the individual metrics and Table 3.3 presents summary statistics as percentages. Looking at Table 3.3 we observe that all properties exhibit a very stable behavior, with 75% of the data points (Q_3) deviating less than 6% from the medians. There are some extreme values, the largest one being a 27.3% deviation in Unit complexity. Nevertheless, even taking into account the full range, thresholds were observed to never overlap and maintain their strictly monotonous behavior, increasing confidence in both the method and the calibration set.

In order to assess the stability of the threshold values in terms of the computed ratings, for each system it was calculated the absolute differences from its median rating throughout all the runs. This amounts to $100 \times 100 = 10000$ data points. Table 3.4 presents summary statistics on the results, made relative to the possible range of $5.5 - 0.5 = 5$ (shown as percentages). Again, all properties exhibit a very stable behavior, with 75% of the data points (Q_3) deviating less than 1.6% from the medians.

We observe that the model is more stable at the ratings level than at the threshold level. This is to be expected since differences in thresholds of different risk categories for the same rating cancel each other.

As conclusion, the impact of including or excluding specific systems or small groups of systems is limited and small. This indicates good stability of the results obtained using this benchmark.

3.7 Related work

Various alternatives for aggregating measurements have been proposed: addition, central tendency measures, distribution parameter fitting, wealth inequality measures or custom formulae. This section, discusses these alternatives and compares them to the approach introduced in this chapter.

3.7.1 Addition

A most basic way of aggregating measurements is addition. Individual measurements are all added together and the total is reported at system level. Lanza and Marinescu [58] use addition to aggregate the NOM (Number of Operations) and CYCLO (McCabe cyclomatic number) metrics at system-level. Chidamber and Kemerer [22] use addition to aggregate the individual complexity numbers of methods into the WMC (Weighted Methods per Class) metric at class level.

However, addition does not make sense for all metrics (e.g. the Fan-in metric which is not defined at system-level). Also, when adding measurements together we lose information about how these measurements are distributed in the code, thus precluding pinpointing potential problems. For example, the WMC metric does not distinguish between a class with many methods of moderate size and complexity and a class with a single huge and highly complex method. In the introduced approach, the risk profiles and the mappings to ratings ensure that such differences in distribution are reflected in the system-level ratings.

3.7.2 Central tendency

Central tendency functions such as mean (simple, weighted or geometric) or the median have also been used to aggregate metrics by many authors. For example, Spinellis [82] aggregates metrics at system level using mean and median, and uses these values to perform comparisons among four different operating-system kernels. Coleman et al. [23], in the Maintainability Index model, aggregate measurements at system-level using mean.

The simple mean directly inherits the drawbacks of aggregation by addition, since the mean is calculated as the sum of measurements divided by their number.

In general, central tendency functions fail to do justice to the skewed nature of most software-related metrics. Many authors have shown that software metrics are heavily skewed [22, 87, 24, 89, 62] providing evidence that metrics should be aggregated using other techniques. Spinellis [82] provided additional evidence that central tendency measures should not be used to aggregate measurements, concluding that they are unable to clearly differentiate the studied systems.

The skewed nature of source code metrics is accommodated by the use of risk profiles in the introduced methodology. The derivation of thresholds from benchmark data ensure that differences in skewness among systems are captured well at the first level [6]. In the calibration of the mappings to ratings at the second level, the ranking of the systems is also based on their performance against thresholds, rather than on a central tendency measure, which ensures that the ranking adequately takes the differences among systems in the tails of the distributions into account.

3.7.3 Distribution fitting

A common statistical method to describe data is to fit it against a particular distribution. This involves the estimation of distribution parameters and quantification of goodness of fit. These parameters can then be used to characterize the distribution data. Hence, fitting a distribution parameter can be seen as a method to aggregate measurements at system-level. For instance, Concas et al. [24], Wheeldon and Counsell [89] and Louridas et al. [62] have shown that several metrics follow power-law or log-normal distributions, computing the distribution parameters for a set of systems as case study.

This method has several drawbacks. First, the assumption that for all systems a metric follows the same distribution (albeit with different parameters) may be wrong. In fact, when a group of developers starts to act on the measurements for the code they are developing, the distribution of that metric may rapidly change into a different shape. As a result, the assumed statistical model no longer holds and the distribution parameters stop to be meaningful. Second, to understand the characterization of a system by its distribution parameters requires software engineering practitioners to understand the assumed statistical model, which undermines the understandability of this aggregation method. Third, root-cause analysis, i.e., tracing distribution parameters back to problems at particular source code locations, is not straightforward.

The introduced methodology does not assume a particular statistical model, and could therefore be described as *non-parametric* in this respect. This makes it robust against lack of conformance to such a model by particular systems, for instance due to quality feedback mechanisms in development environments and processes.

3.7.4 Wealth inequality

Aggregation of software metrics has also been proposed using the Gini coefficient by Vasa et al. [87] and the Theil index by Serebrenik and Van den Brand [78]. The Gini coefficient and the Theil index are used in economics to quantify the inequality of wealth. An inequality value of 0 means that measurements follow a constant distribution, i.e., all have the same value. A high value of inequality, on the other hand, indicates a skewed distribution, where some measurements are much higher than the others.

Both Gini and Theil adequately deal with the skewness of source code metrics without making assumptions about an underlying distribution. Both have shown good results when applied to software evolution analysis and to detect automatically generated code. Still they suffer of major shortcomings when used to aggregate source code measurement data.

Both Gini and Theil provide indications of the *differences* in quality of source code elements within a system, not of the degree of quality itself. This can easily be seen from an example. Let us assume that for a class-level metric M, higher values indicate poorer quality. Let be a system A with three equally-sized classes with metric values 1, 1 and 100 (Gini=0.65 and Theil=0.99), and B be another system also with three equally-sized classes with values 1, 100 and 100 (Gini=0.33 and Theil=0.38). Clearly, system A has *higher* quality, since one third rather than two thirds of its code suffers from poor quality. However, both Gini and Theil indicate that A has greater inequality making it score lower than B. Thus, even though inequality in quality may often indicate low quality, they are conceptually different.

Both Gini and Theil do not allow root-cause analysis, i.e., they do not provide means to directly identify the underlying measurements which explain the computed inequality. The Theil index, which improves over the Gini factor, provides means to explain the inequality according to a specific partition by reporting a value for how much a specific partition of measurements accounts for the overall inequality. However, Theil is limited to provide insight at the partition level not providing information at lower levels.

Similarly to the Gini and Theil, the ratings based on risk-profile proposed in this chapter capture differences among systems that occur when quality values are distributed unevenly over source code elements. But, by contrast, the proposed approach is also based on the magnitude of the metric values rather than exclusively on the inequality among them.

3.7.5 Custom formula

Jansen [50] proposed the *confidence factor* as a metric to aggregate violations reported by static code checkers. The *confidence factor* is derived by a formula taking into account the total number of rules, the number of violations, the severity of violations and the overall size of the system, and the percentage of files that are successfully checked, reporting a value between 0 and 100. The higher the value the higher the *confidence* on the system. A value of 80 is normally set as minimum threshold.

Although Jansen reports the usefulness of the metric, he states that the formula definition is based on heuristics and requires formal foundation and validation. Also, root-cause analysis can only be achieved by investigating the extremal values of underlying measurements. In contrast the proposed approach to derive thresholds from benchmark data can be used both to aggregate measurements into ratings and to trace back the ratings back to measurements.

3.8 Summary

This chapter was devoted to the calibration of mappings from code-level measurements to system-level ratings. Calibration is done against a benchmark of software systems and their associated code measurements. The presented methodology adds to earlier work, described in Chapter 2, on deriving thresholds for source code metrics from such benchmark data [6].

The core of our approach is an iterative algorithm that (i) ranks systems by their performance against pre-determined thresholds and (ii) based on the obtained ranking determines how performance against the thresholds translates into ratings on a unit-less scale. The contributions of this chapter are:

- An algorithm to perform calibration of thresholds for risk profiles;
- Formalization of the calculation of ratings from risk profiles;

- Discussion of the caveats and options to consider when using the approach;
- An application of the method to determine thresholds for 4 source code metrics;
- A procedure to assess the stability of the thresholds obtained with a particular data set.

The combination of methods to derive thresholds and calibrate ratings enable a generic blueprint for building metrics-based quality models. As such, both methods combined can be applied to any situation where one would like to perform automatic qualitative assessments based on multi-dimensional quantitative data. The presence of a large, well-curated repository of benchmark data is, nevertheless, a prerequisite for successful application of our methodology. Other than that, one can imagine applying it to assess software product quality but using a different set of metrics, assessing software process or software development community quality, or even applying it to areas outside software development.

The methodology is applied by SIG to annually re-calibrate the SIG quality model [41, 27], which forms the basis of the evaluation and certification of software maintainability conducted by SIG and TÜViT [10].

Chapter 4

Static Evaluation of Test Quality

Test coverage is an important indicator for unit test quality. Tools such as Clover[1] compute coverage by first instrumenting the code with logging functionality, and then logging which parts are executed during unit test runs.

Since computation of test coverage is a dynamic analysis, it assumes a working installation of the software. In the context of software quality assessment by an independent third party, a working installation is often not available. The evaluator may not have access to the required libraries or hardware platform. The installation procedure may not be automated or documented.

This chapter proposes a technique for estimating test coverage at method level through static analysis only. The technique uses slicing of static call graphs to estimate the dynamic test coverage. Both the technique and its implementation are explained. The metric calculated with this technique will be called Static Estimation of Test Coverage (SETC) to differentiate it from dynamic coverage metrics. The results of the SETC metric are validated by statistical comparison to values obtained through dynamic analysis using Clover. A positive correlation with high significance will be found at system, package and class levels.

To evaluate test quality using the SETC metric, risk thresholds are derived for

[1] http://www.atlassian.com/software/clover/

class-level coverage and rating thresholds calibrated to compute a system-level coverage rating. In validating coverage rating against system-level coverage a significant and high correlation between the two metrics is found. Further validation of coverage ratings against indicators for performance of issue resolution is done indicating that systems with higher coverage rating have higher productivity in resolving issues reported in an Issue Tracking System (ITS).

4.1 Introduction

In the Object-Oriented (OO) community, *unit testing* is a white-box testing method for developers to validate the correct functioning of the smallest testable parts of source code [13]. OO unit testing has received broad attention and enjoys increasing popularity, also in industry [40].

A range of frameworks has become available to support unit testing, including SUnit, JUnit, and NUnit[2]. These frameworks allow developers to specify unit tests in source code and run suites of tests during the development cycle.

A commonly used indicator to monitor the quality of unit tests is *code coverage*. This notion refers to the portion of a software application that is actually executed during a particular execution run. The coverage obtained when running a particular suite of tests can be used as an indicator of the quality of the test suite and, by extension, of the quality of the software if the test suite is passed successfully.

Tools are available to compute code coverage during test runs [91] which work by instrumenting the code with logging functionality before execution. The logging information collected during execution is then aggregated and reported. For example, Clover[3] instruments Java source code and reports statement coverage and branch coverage at the level of methods, classes, packages and the overall system. Emma[4] instruments Java bytecode, and reports statement coverage and method coverage at the

[2]http://sunit.sourceforge.net, http://www.junit.org, http://www.nunit.org
[3]http://www.atlassian.com/software/clover/
[4]http://emma.sourceforge.net/

same levels. The detailed reports of such tools provide valuable input to increase or maintain the quality of test code.

Computing code coverage involves running the application code and hence requires a working installation of the software. In the context of software development, satisfaction of this requirement does not pose any new challenge.

However, in other contexts this requirement can be highly impractical or impossible to satisfy. For example, when an independent party evaluates the quality and inherent risks of a software system [86, 56], there are several compelling reasons that put availability of a working installation out of reach. The software may require hardware not available to the assessor. The build and deployment process may not be reproducible due to a lack of automation or documentation. The software may require proprietary libraries under a non-transferrable license. In embedded software, for instance, instrumented applications by coverage tools may not run or may display altered behavior due to space or performance changes. Finally, for a very large system it might be too expensive to frequently execute the complete test suite, and subsequently, compute coverage reports.

These limitations derive from industrial practice in analyzing software that may be incomplete and may not be possible to execute. To overcome these limitations a lightweight technique to estimate test coverage prior to running the test cases is necessary.

The question that naturally arises is: could code coverage by tests possibly be determined without actually running the tests? And which trade-off may be made between sophistication of such a static analysis and its accuracy?

Further investigation into the possibility of using static coverage as indicator for test quality was done. Aiming for 100% coverage is most of the time unpractical (if not at all impossible). Thus, the question arises: can we define an objective baseline, that is representative of a benchmark of software systems that allow us to reach to an evaluation about test quality?

This chapter is structured as follows. A static analysis technique for estimating code coverage, SETC, based on slicing of call graphs, is proposed in Section 4.2. A discussion of the sources of imprecision inherent in this analysis as well as the impact of imprecision on the results is the subject of Section 4.3. Section 4.4 provides experimental assessment of the quality of the static estimates compared to the dynamically determined code coverage results for a range of proprietary and Open-Source Software (OSS) software systems. Derivation of risk and rating thresholds for static coverage and experimental analysis of their relation with external quality using metrics for defect resolution performance is given in Section 4.5. Related work is reviewed in Section 4.6 which is followed by a summary of contributions in Section 4.7.

4.2 Approach

The approach for estimating code coverage involves reachability analysis on a graph structure (also known as graph slicing [57]). This graph is obtained from source code via static analysis. The granularity of the graph is at the method level, and control or data flow information is not assumed. Test coverage is estimated (SETC) by calculating the ratio between the number of production code methods reached from tests and the overall number of production code methods.

An overview of the various steps of the static estimation of test coverage is given in Figure 4.1. We briefly enumerate the steps before explaining them in detail in the upcoming sections:

1. From all source files F, including both production and test code, a graph G is extracted via static analysis which records both structural information and call information. Also, the test classes are collected in a set T.

2. From the set of test classes T, test methods are determined and used as slicing criteria. From test method nodes, the graph G is sliced, primarily along call edges, to collect all methods that are reached in a set M.

3. For each production class in the graph, the number of methods defined in that class is counted. Also the set of covered methods is used to arrive at a count

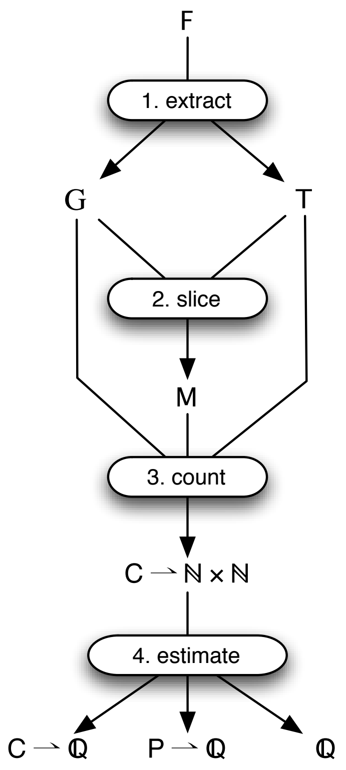


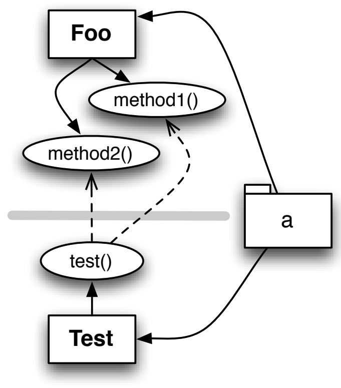
Figure 4.1: Overview of the approach. The input is a set of files (F). From these files, a call graph is constructed (G) and the test classes of the system are identified (T). Slicing is performed on the graph with the identified test classes as entry points, and the production code methods (M) in the resulting slice are collected. The methods thus covered allow to count for each class (C) in the graph (i) how many methods it defines (ii) how many of these are covered. Finally, coverage ratio estimation is then computed on the class, package and system levels. The harpoon arrow denotes a finite map.

of covered methods in that class. This is depicted in Figure 4.1 as a map from classes to a pair of numbers.

4. The final estimates at class, package, and system levels are obtained as ratios from the counts per class.

Note that the main difference between the steps of this approach and dynamic analysis tools, like Clover, can be found in step 2. Instead of using precise information recorded by logging the methods that are executed, we use an estimation of the methods that are called, determined via static analysis. Moreover, while some dynamic analysis tools take test results into account, this approach does not.

This proposed approach is designed with a number of desirable characteristics in



```
package a;
class Foo {
   void method1() { }
   void method2() { }
}
__________
package a;
import junit.framework.TestCase;
class Test extends TestCase {
   void test() {
      Foo f = new Foo();
      f.method1();
      f.method2();
}  }
```

Figure 4.2: Source code fragment and the corresponding graph structure, showing different types of nodes (package, class and method) and edges (class and method definition and method calls).

mind: only static analysis is used; the graph contains call information extracted from the source code; it is scalable to large systems; granularity is limited to the method level to keep whole-system analysis tractable; it is robust against partial availability of source code; finally, missing information is not blocking, though it may lead to less accurate estimates. The extent to which these properties are realized will become clear in Section 4.4. First, the various steps of our approach will be explained in more detail.

Graph construction

Using static analysis, a graph is derived representing packages, classes, interfaces, and methods, as well as various relations among them. An example is provided in Figure 4.2. Below, the node and edge types that are present in such graphs will be explained. Derivation of the graph relies on Java source code extraction provided by the SemmleCode tool [28]. This section provides a discussion of the required functionality, independent of that implementation.

A directed graph can be represented by a pair $G = (V, E)$, where V is the set of vertices (nodes) and E is the set of edges between these vertices. four types of vertices are distinguished, corresponding to packages (P), classes (C), interfaces (I), and methods (M). Thus, the set V of vertices can be partitioned into four disjoint subsets which can be written as $N_n \in V$ where node type $n \in \{P, C, I, M\}$. In the various figures in this chapter, packages will be represented as folder icons, classes and interfaces as rectangles, and methods as ellipses. The set of nodes that represent classes, interfaces, and methods is also partitioned to differentiate between production (PC) and test code (TC). N_n^c is written where code type $c \in \{PC, TC\}$. The various figures show production code above a gray separation line and test code below. The edges in the extracted graph structure represent both structural and call information. For structural information two types of edges are used: The *defines type* edges (DT) express that a package contains a class or an interface; the *defines method* edges (DM) express that a class or interface defines a method. For call information, two types of edges are used: *direct call* and *virtual call*. A *direct call* edge (DC) represents a method invocation. The origin of the call is typically a method but can also be a class or interface in case of method invocation in initializers. The target of the call edge is the method definition to which the method invocation can be *statically* resolved. A *virtual call* edge (VC) is constructed between a caller and any implementation of the called method that might be resolved to during runtime, due to dynamic dispatch. An example will be shown in Section 4.3.1. The set of edges is actually a relation between vertices such that $E \subseteq \{(u, v)_e \,|\, u, v \in V\}$, where $e \in \{DT, DM, DC, VC\}$. E_e is written for the four partitions of E according to the various edges types. In the figures, solid arrows depict *defines* edges and dashed arrows depict *calls*. Further explanation of the two types of call edges are provided in Section 4.3.1.

Identifying test classes

Several techniques can be used to identify test code, namely, recognizing the use of test libraries or naming conventions. The possibility to statically determine test code by recognizing the use of a known test library, such as JUnit, was investigated. A class is considered as a test class if it uses the testing library. Although this technique is completely automatic, it fails to recognize test helper classes, i.e., classes with the single purpose of easing the process of testing, which do not need to have any reference

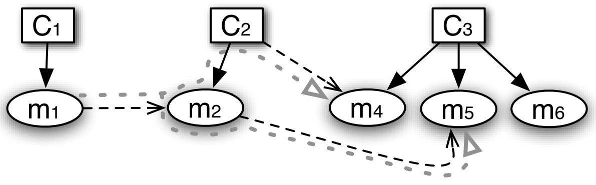


Figure 4.3: Modified graph slicing algorithm in which calls are taken into account originating from both methods and object initializers. Black arrows represent edges determined via static analysis, and grey arrows depict the slicing traversal. Full lines are used for method definitions and dashed lines are used for method calls.

to a test framework. Alternatively, naming conventions are used to determine test code. For the majority of proprietary and OSS systems production and test code are stored in different file system paths. The only drawback of this technique is that, for each system, this path must be manually determined since each project uses its own naming convention.

Slicing to collect covered methods

In the second step, graph slicing [57] is applied to collect all methods covered by tests. The identified set of test classes and their methods are used as slicing criteria (starting points). The various kinds of *call* edges are then followed in forward direction to reach all covered methods. In addition, the slicing algorithm is refined to take into account *call* edges originating from the object initializers. The modification consists in following *define method* edges backward from covered methods to their defining classes, which then triggers subsequent traversal to the methods invoked by the initializers of those classes. The modified slicing algorithm is depicted in Figure 4.3. The edge from C_2 to m_4 illustrates an initializer call.

The modified slicing algorithm can be defined as follows. We write $n \xrightarrow{call} m$ for an edge in the graph that represents a node n calling a method m, where the call type can be vanilla or virtual. Notation $m \xleftarrow{def} c$ means the inverse of a *define method* edge, i.e., meaning a function that returns the class c in which a method m is defined. We write $n \xrightarrow{init} m$ for $n \xrightarrow{call} m_i \xleftarrow{def} c \xrightarrow{call} m$, i.e., to denote that a method m is reached from a node n via a class initialization triggered by a call to method m_i (e.g., $m_1 \xrightarrow{init} m_4$, in which $m_1 \xrightarrow{call} m_2 \xleftarrow{def} C_2 \xrightarrow{call} m_4$). Finally, $n \xrightarrow{invoke} m$ is written for $n \xrightarrow{call} m$ or $n \xrightarrow{init} m$. Now, let n be a graph node corresponding to a class, interface or a method (package nodes are not considered). Then, a method m is said to be reachable from a node n if $n \xrightarrow{invoke}^{+} m$ where R^+ denotes the transitive closure of the relation R.

These declarative definitions can be encoded in a graph traversal algorithm in a straightforward way. The implementation, however, was carried out in the relational query language .QL [28], in which these definitions are expressed almost directly.

Count methods per class

The third step computes the two core metrics for the static test coverage estimation:

- **Number of defined methods per class (DM)**, defined as $DM : n_C \rightharpoonup \mathbb{N}$. This metric is calculated by counting the number of outgoing *define method* edges per class.
- **Number of covered methods per class (CM)**, defined as $CM : n_C \rightharpoonup \mathbb{N}$. This metric is calculated by counting the number of outgoing *define method* edges where the target is a method contained in the set of covered methods.

These statically computed metrics are stored in a finite map $n_C \rightharpoonup \mathbb{N} \times \mathbb{N}$. This map will be used to compute coverage at class, package and system levels as shown below.

Estimate static test coverage

After computing the two basic metrics we can obtain derived metrics: coverage per class, packages, and system.

- **Class coverage** Method coverage at the class level is the ratio between covered and defined methods per class:

$$CC(c) = \frac{CM(c)}{DM(c)} \times 100\%$$

- **Package coverage** Method coverage at the package level is the ratio between the total number of covered methods and the total number of defined methods per package,

$$PC(p) = \frac{\sum\limits_{c \in p} CM(c)}{\sum\limits_{c \in p} DM(c)} \times 100\%$$

where $c \in p$ iff $c \in V^P \wedge c \overset{def}{\leftarrow} p$, i.e., c is a production class in package p.

- **System coverage** Method coverage at system level is the ratio between the total number of covered methods and the total number of defined methods in the overall system,

$$SC = \frac{\sum\limits_{c \in G} \mathit{CM(c)}}{\sum\limits_{c \in G} \mathit{DM(c)}} \times 100\%$$

where $c \in G$ iff $c \in V^P$, i.e., c is a production class.

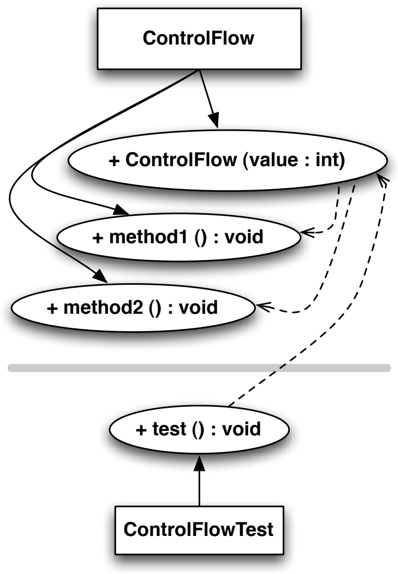
4.3 Imprecision

The static coverage analysis is based on a statically derived graph, in which the structural information is exact and the method call information is an estimation of the dynamic execution. The precision of the estimation is a function of the precision of the call graph extraction. As mentioned before, this relies on SemmleCode.

This section discusses various sources of imprecision independent of tool of choice: control flow, dynamic dispatch, framework/library calls, identification of production and test code, and failing tests. How to deal with imprecision is also discussed here.

4.3.1 Sources of imprecision

Control flow Figure 4.4 presents an example where the graph contains imprecision due to control flow, i.e., due to the occurrence of method calls under conditional statements. In this example, if `value` is greater than zero `method1` is called, otherwise



```
class ControlFlow {
  ControlFlow(int value) {
     if (value > 0)
        method1();
     else
        method2();
  }
  void method1() { }
  void method2() { }
}
```

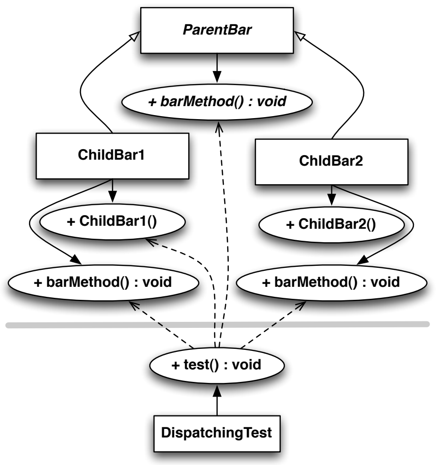
```
import junit.framework.*;

class ControlFlowTest
      extends TestCase {
   void test() {
      ControlFlow cf =
         new ControlFlow(3);
}  }
```

Figure 4.4: Imprecision related to control flow.

`method2` is called. In the test, the value 3 is passed as argument and `method1` is called. However, without data flow analysis or partial evaluation, it is not possible to statically determine which branch is taken, and which methods are called. For now we will consider an optimistic estimation, considering both `method1` and `method2` calls. Further explanations about how to deal with imprecision are given in Section 4.3.2.

Other types of control-flow statements will likewise lead to imprecision in call graphs, namely `switch` statements, looping statements (`for`, `while`), and branching statements (`break`, `continue`, `return`).

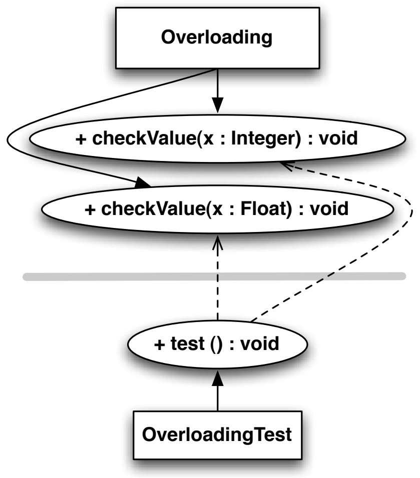
Dynamic dispatch Figure 4.5 presents an example of imprecision due to dynamic dispatch. A parent class `ParentBar` defines `barMethod`, which is redefined by two subclasses (`ChildBar1` and `ChildBar2`). In the test, a `ChildBar1` object is assigned to a variable of the `ParentBar` type, and the `barMethod` is invoked. During test execution, the `barMethod` of `ChildBar1` is called. Static analysis, however, identifies all three implementations of `barMethod` as potential call targets, represented in the graph as three edges: one *direct call* edge to the `ParentBar` and two



```
class ParentBar {
   void barMethod() { };
}
class ChildBar1
      extends ParentBar {
   ChildBar1() { }

   void barMethod() { }
}
class ChildBar2
      extends ParentBar {
   ChildBar2() { }

   void barMethod() { }
}
```

```
import junit.framework.*;

class DynamicDispatchTest
      extends TestCase {
   void test() {
      ParentBar p =
         new ChildBar1();
      p.barMethod();
}  }
```

Figure 4.5: Imprecision: dynamic dispatch.


Overloading
+ checkValue(x : Integer) : void
+ checkValue(x : Float) : void
+ test () : void
OverloadingTest

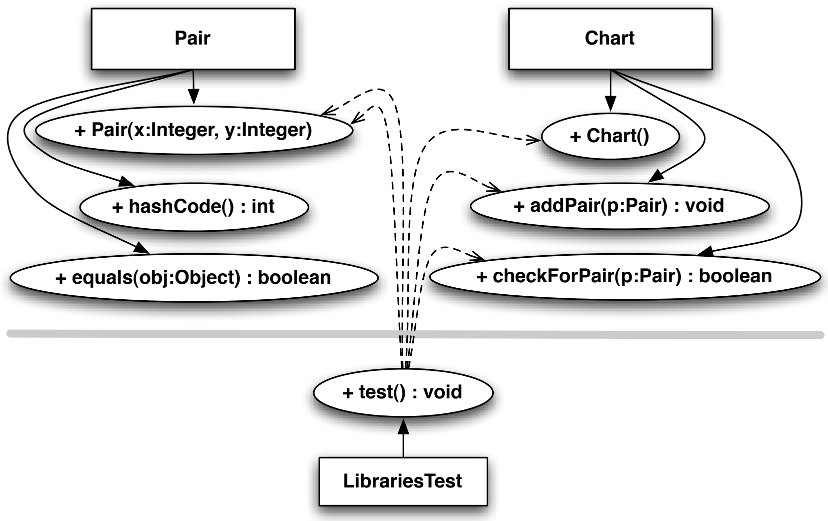
```
class Overloading {
  void checkValue(Integer x) {}
  void checkValue(Float x) {}
}
```

```
import junit.framework.*;

class ControlFlowTest
      extends TestCase {
   void test() {
      Overloading o =
         new Overloading();
      o.checkValue(3);
}  }
```

Figure 4.6: Imprecision: method overloading.

virtual call edges to the `ParentChild1` and `ParentChild2` implementations.

Overloading Figure 4.6 presents an example where the graph is imprecise due to overloading. The class `Overloading` contains two methods with the same name but with different argument types: Integer and Float. The test calls the `checkValue` method with the constant value 3 and the method with Integer argument is called.


Pair
Chart
+ Pair(x:Integer, y:Integer)
+ Chart()
+ hashCode() : int
+ addPair(p:Pair) : void
+ equals(obj:Object) : boolean
+ checkForPair(p:Pair) : boolean
+ test() : void
LibrariesTest

```
class Pair {
   Integer x; Integer y;

   Pair(Integer x, Integer y) {
     ...
   }
   int hashCode() { ... }
   boolean equals(Object obj) {
      ...
}  }
class Chart {
   Set pairs;

   Chart() { pairs = new HashSet(); }

   void addPair(Pair p) {
      pairs.add(p);
   }
   void checkForPair(Pair p) {
      return pairs.contains(p);
}  }
```

```
import junit.framework.*;

class LibrariesTest
      extends TestCase {

   void test() {
      Chart c = new Chart();

      Pair p1 = new Pair(3,5);
      c.addPair(p1);

      Pair p2 = new Pair(3,5);
      c.checkForPair(p2);
}  }
```

Figure 4.7: Imprecision: library calls.

However, the call graph is constructed without dynamic type analysis and will include calls to both methods.

Frameworks / Libraries Figure 4.7 presents yet another example of imprecision caused by frameworks/library calls of which no code is available for analysis. The class `Pair` represents a two-dimensional coordinate, and class `Chart` contains all the points of a chart. `Pair` defines a constructor and redefines the `equals` and `hashCode` methods to enable the comparison of two objects of the same type. In the test, a `Chart` object is created and coordinate $(3, 5)$ is added to the chart. Then another object with the same coordinates is created and checked to exist in the chart.

When a `Pair` object is added to the set, and when checking if an object exists in a set, the methods `hashCode` and `equals` are called. These calls are not present in the call graph.

Identification of production and test code Failing to distinguish production from test code has a direct impact on test coverage estimation. Recognizing tests as production code increases the size of the overall production code and hides calls from tests to production code, possibly causing a decrease of coverage (underestimation). Recognizing production code as tests has the opposite effect, decreasing the size of overall production code and increasing the number of calls resulting in either a higher (overestimation) or lower (underestimation) coverage.

As previously stated, the distinction between production and test code is done using file system path information. A class is considered test code if it is inside a test folder, and considered production code if it is inside a non-test folder. Since most projects and tools (e.g. Clover) respect and use this convention this can be regarded as a safe approach.

Failing tests Unit testing requires assertion of the state and/or results of a unit of code to detect faults. If the test succeeds the unit under test is considered as test covered. However, if the test does not succeed two alternatives are possible. First, the unit test is regarded as not covered, but the unit under test is considered as test covered. Second, both the unit test and the unit under test are considered as not test covered. Emma is an example of the first case, while Clover is an example of the second.

Failing tests can cause imprecision since Clover will consider the functionality under test as not covered while the static approach, which is not sensitive to test results, will consider the same functionality as test covered. However, failing tests are not common in released software.

4.3.2 Dealing with imprecision

Among all sources of imprecision, we shall be concerned with control flow, dynamic dispatch and frameworks/libraries only. Less common imprecision caused by method overloading, test code identification and failing tests will not be considered.

Two approaches are possible in dealing with imprecision without resorting to detailed control and data flow analyses. In the pessimistic approach, only call edges that are guaranteed to be exercised during execution are followed. This will result in a systematic underestimation of test coverage. In an optimistic approach, all potential call edges are followed, even if they are not necessarily exercised during execution. Under this approach, test coverage will be overestimated.

In the particular context of quality and risk assessment, only the optimistic approach is suitable. A pessimistic approach would lead to a large number of false negatives (methods that are erroneously reported to be uncovered). In the optimistic approach, methods reported to be uncovered are with high certainty indeed not covered. Since the purpose is to detect the lack of coverage, only the optimistic approach makes sense. Hence, only values for the optimistic approach we will reported. However, the optimistic approach is not consistently optimistic for all the uncertainties previously mentioned: imprecision due to library/framework will always cause underestimation. This underestimation can influence coverage estimation to values lower than Clover coverage. Nevertheless, if a particular functionality is only reached via frameworks/libraries, i.e., it can not be statically reached from a unit test, it is fair to assume that this functionality is not unit test covered, albeit considered covered by a test.

In the sequel, the consequences of these choices for the accuracy of the analysis will be experimentally investigated.

Table 4.1: Description of the systems used in the experiment

System	Version	Author / Owner	Description
JPacMan	3.04	Arie van Deursen	Game used for OOP education
Certification	20080731	SIG	Tool for software quality rating
G System	20080214	C Company	Database synchronization tool
Dom4j	1.6.1	MetaStuff	Library for XML processing
Utils	1.61	SIG	Toolkit for static code analysis
JGAP	3.3.3	Klaus Meffert	Library of Java genetic algorithms
Collections	3.2.1	Apache	Library of data structures
PMD	5.0b6340	Xavier Le Vourch	Java static code analyzer
R System	20080214	C Company	System for contracts management
JFreeChart	1.0.10	JFree	Java chart library
DocGen	r40981	SIG	Cobol documentation generator
Analyses	1.39	SIG	Tools for static code analysis

4.4 Comparison of static and dynamic coverage

A comparison of the results of static estimation of coverage against dynamically computed coverage for several software systems was experimented. An additional comparison for several revisions of the same software system was also done in order to investigate if our static estimation technique is sensitive to coverage fluctuations.

Coverage will be reported at system, package and class levels to gain insight about the precision of the static coverage when compared to dynamic coverage.

4.4.1 Experimental design

Systems analyzed Twelve Java systems were analyzed, ranging from 2.5k to 268k Source Lines of Code (SLOC), with a total of 840k SLOC (production and test code). The description of the systems is listed in Table 4.1 and metric information about those systems is listed in Table 4.2. The systems are diverse both in terms of size and scope. JPacMan is a tiny system developed for education. Dom4j, JGAP, Collections and JFreeChart are Open-Source Software (OSS) libraries and PMD is an OSS tool[5]. G System and R System are anonymized proprietary systems. Certification, Analyses and DocGen are proprietary tools and Utils is a proprietary library.

[5]http://www.dom4j.org, http://jgap.sourceforge.net, http://commons.apache.org/collections, http://www.jfree.org/jfreechart, http://pmd.sourceforge.net

Table 4.2: Characterization of the systems used in the experiment.

System	SLOC		# Packages		# Classes		# Methods	
	Prod.	Test	Prod.	Test	Prod.	Test	Prod.	Test
JPacMan	1,539	960	2	3	29	17	223	112
Certification	2,220	1,563	14	9	71	28	256	157
G System	3,459	2,910	15	16	56	70	504	285
Dom4j	18,305	5,996	14	11	166	105	2921	685
Utils	20,851	16,887	37	32	323	183	3243	1290
JGAP	23,579	19,340	25	20	267	184	2990	2005
Collections	26,323	29,075	12	11	422	292	4098	2876
PMD	51,427	11,546	66	44	688	206	5508	1348
R System	48,256	34,079	62	55	623	353	8433	2662
JFreeChart	83,038	44,634	36	24	476	399	7660	3020
DocGen	73,152	54,576	111	85	1359	427	11442	3467
Analyses	131,476	136,066	278	234	1897	1302	13886	8429

Additionally, 52 releases of the Utils project were analyzed with a total of over 1.2M LOC, with sizes ranging from 4.5k LOC, for version 1.0, to 37.7k LOC, for version 1.61, spanning several years of development.

For all the analyzed systems, production and test code could be distinguished by file path and no failing tests existed.

Measurement For the dynamic coverage measurement, XML reports were produced by the Clover tool. While for some projects the Clover report was readily available, others had to be computed by modifying the build scripts, finding all necessary library and running the tests. XSLT transformations were used to extract the required information: names of packages and classes; numbers of total and covered methods.

For the static estimation of coverage, the extract, slice, and count steps of the approach were implemented in relational .QL queries in the SemmleCode tool [28].

Statistical analysis For statistical analysis the R tool was used [84]. Histograms were created to inspect the distribution of the estimate (static) coverage and the true (Clover) coverage. To visually compare these distributions, scatter plots of one against the other, and histograms of their differences were created. To inspect central tendency and dispersion of the true and estimated coverage as well as of their differences, we used descriptive statistics, such as median and interquartile range. To investigate correlation of true and estimated coverage a non-parametric method (Spearman's rank correlation coefficient [81]) and a parametric method (Pearson product-moment correlation coefficient [70]) were used. Spearman is used when no assumptions about data distributions can be made. Pearson, on the other hand, is more precise than Spearman, but can only be used if data are assumed to be normal. For testing data normality the Anderson-Darling test [7] for a 5% significance level was used The null hypothesis of the test states that the data can be assumed to follow the normal distribution, while the alternative hypothesis states that the data cannot be assumed to follow a normal distribution. The null hypothesis is rejected for a computed p-value smaller or equal than 0.05.

4.4.2 Experiment results

The results of the experiment are discussed, first at the level of complete systems and then by analyzing several releases of the same project. Further analysis is done by looking at at class and package level results and, finally, by looking at one system in more detail.

System coverage results Table 4.3 and the scatter plot in Figure 4.8 show the estimated (static) and the true (Clover) system-level coverage for all systems. Each dot in Figure 4.8 represents a system. Table 4.3 shows that the differences range from -16.5 to 19.5 percent points. In percent points, the average of absolute differences is 9.35 and the average difference is 3.57. Figure 4.8 shows that static coverage values are close to the diagonal, which depicts the true coverage. For one third of the systems coverage was underestimated while two thirds was overestimated.

Assuming no distribution about data normality, Spearman correlation can be used. Spearman correlation reports 0.769 with high significance (p-value < 0.01). Using Anderson-Darling test for data normality, the p-values are 0.920 and 0.522 for static

Table 4.3: Static and dynamic (Clover) coverage, and coverage differences at system level.

System	Static (%)	Clover (%)	Differences (%)
JPacMan	88.06	93.53	−5.47
Certification	92.82	90.09	2.73
G System	89.61	94.81	−5.19
Dom4j	57.40	39.37	18.03
Utils	74.95	70.47	4.48
JGAP	70.51	50.99	19.52
Collections	82.62	78.39	4.23
PMD	80.10	70.76	9.34
R System	65.10	72.65	−7.55
JFreeChart	69.88	61.55	8.33
DocGen	79.92	69.08	10.84
Analyses	71.74	88.23	−16.49

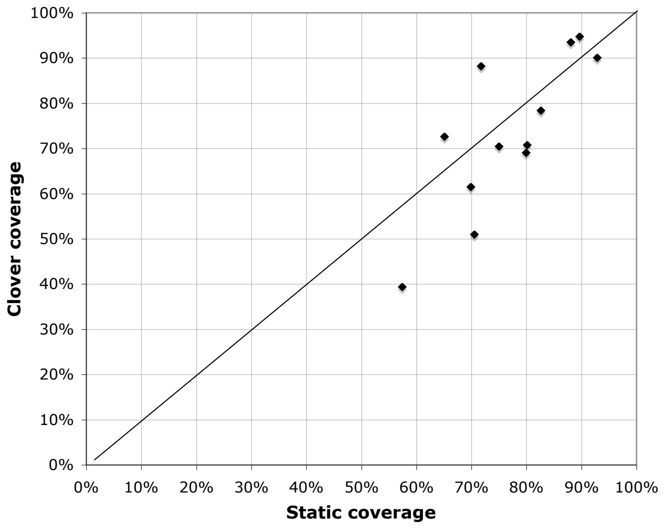


Figure 4.8: Scatter plot comparing static and dynamic (Clover) coverage for each system.

coverage and clover coverage, respectively. Since, the null hypothesis cannot be reject, i.e., it is not possible to reject that the data does not belong to a normal distribution, a more accurate method for correlation can be used, Pearson correlation, which reports 0.802 and p-value < 0.01. Hence, static and clover coverage are highly correlated with high significance.

Coverage comparison for the Utils project releases Figure 4.9 plots a comparison between static and dynamic coverage for 52 releases of Utils, from releases 1.0 to 1.61 (some releases were skipped due to compilation problems).

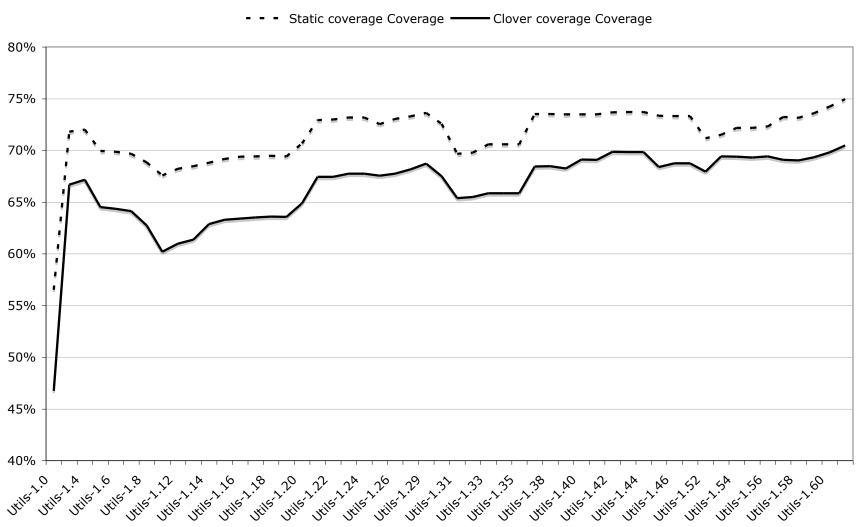


Figure 4.9: Plot comparing static and dynamic coverage for 52 releases of Utils.

Static coverage is consistently higher than Clover coverage. Despite this over-estimation, static coverage follows the same variations as reported by Clover which indicates that static coverage is able to detect coverage fluctuations.

The application of Anderson-Darling test rejects the null hypothesis for 5% of significance. Hence, correlation can only be computed with the non-parametric Spearman test. Spearman correlation reports a value of 0.888 with high significance (p-value $<$ 0.01), reinforcing that estimated and true system-level coverage are highly correlated with high significance.

From Figure 4.9 we can additionally observe that although static coverage is consistently higher than Clover coverage, this difference decreases along releases. This can be due to the increasing size of the system or simply due to the increase of coverage.

Using Spearman correlation to test between system size, measured in SLOC, and coverage difference resulted in a value of -0.851, meaning high negative correlation with high significance (p-value < 0.01). This means that the bigger the system the lower the coverage difference.

Spearman correlation between real (Clover) coverage and coverage differences re-

Table 4.4: Statistical analysis reporting correlation between static and dynamic (Clover) coverage, and median and interquartile ranges (IQR) for coverage differences at class and package levels. Stars are used to depict correlation significance, no star for not significant, one star for significant and two stars for highly significant.

System	Spearman		Median		IQR	
	Class	Package	Class	Package	Class	Package
JPacMan	0.467^{*}	1.000	0	-0.130	0.037	-
Certification	0.368^{**}	0.520	0	0.000	0.000	0.015
G System	0.774^{**}	0.694^{**}	0	0.000	0.000	0.045
Dom4j	0.584^{**}	0.620^{*}	0.167	0.118	0.333	0.220
Utils	0.825^{**}	0.778^{**}	0	0.014	0.000	0.100
JGAP	0.733^{**}	0.786^{**}	0	0.000	0.433	0.125
Collections	0.549^{**}	0.776^{**}	0	0.049	0.027	0.062
PMD	0.638^{**}	0.655^{**}	0	0.058	0.097	0.166
R System	0.727^{**}	0.723^{**}	0	-0.079	0.043	0.162
JFreeChart	0.632^{**}	0.694^{**}	0	0.048	0.175	0.172
DocGen	0.397^{**}	0.459^{**}	0	0.100	0.400	0.386
Analyses	0.391^{**}	0.486^{**}	0	-0.016	0.333	0.316

ports -0.848, high negative correlation, with high significance (p-value < 0.01). This means that the higher the coverage the lower the coverage difference.

However, measuring correlation between system size and real coverage reports 0.851, high correlation, with high significance (p-value < 0.01). This means that as code is increasing there was also an effort to simultaneously improve coverage. Hence, from these data, it is not conclusive whether code size has an effect on coverage difference or not.

Package and Class coverage results Despite the encouraging system-level results, it is also important, and interesting, to analyze the results at class and package levels.

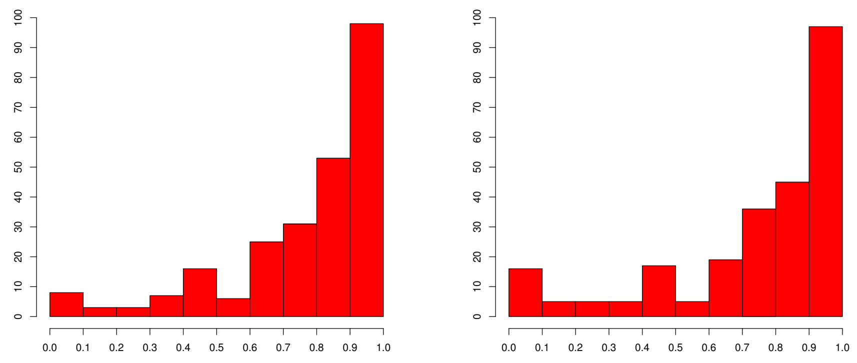
Table 4.4 reports for each system the Spearman correlation and significance, and the median (central tendency) and interquartile range (IQR) of the differences between estimated and true values. Correlation significance is depicted without a star for p-value ≥ 0.05, meaning not significant, with a single star for p-value < 0.05, meaning significant, and two stars for p-value < 0.01, meaning highly significant. Since Anderson-Darling test rejected that the data set belongs to a normal distribution, Spearman correlation was used.

At class level, except for JPacMan (due to its small size), all systems report high significance. The correlation, however, varies from 0.368 (low correlation) for Certification, to 0.825 (high correlation) for Utils. Spearman correlation, for all systems, reports a moderate correlation value of 0.503 with high significance (p-value < 0.01). With respect to the median, all systems are centered in zero, except for Dom4j in which there is a slight overestimation. The IQR is not uniform among systems varying from extremely low values (Collections) to relatively high values (JGAP and DocGen), meaning that the dispersion is not uniform among systems. For the systems with a high IQR, Spearman shows lower correlation values, as to be expected.

At package level, except for JPacMan, Certification and Dom4j, all systems report high significance. Dom4j correlation is significant at 0.05 level while for JPacMan and Certification it is not significant. Such low significance levels are due to the small number of packages. Correlation, again, varies from 0.459 (moderate correlation) for DocGen, to 0.786 (high correlation) for JGAP. The correlation value for JPacMan is not taken into account since it is not significant. Spearman correlation, for all systems, reports a moderate correlation value of 0.536 with high significance (p-value < 0.01). Regarding the median, and in contrast to what was observed for class level, only three systems reported a value of 0. However, except for JPacMan, Dom4j and DocGen all other systems report values very close to zero. IQR shows more homogeneous values than at class level, with values below 0.17 except for JPacMan (which does not have IQR due to its sample size of 3) and Dom4j, DocGen and Analyses whose values are higher than 0.20.

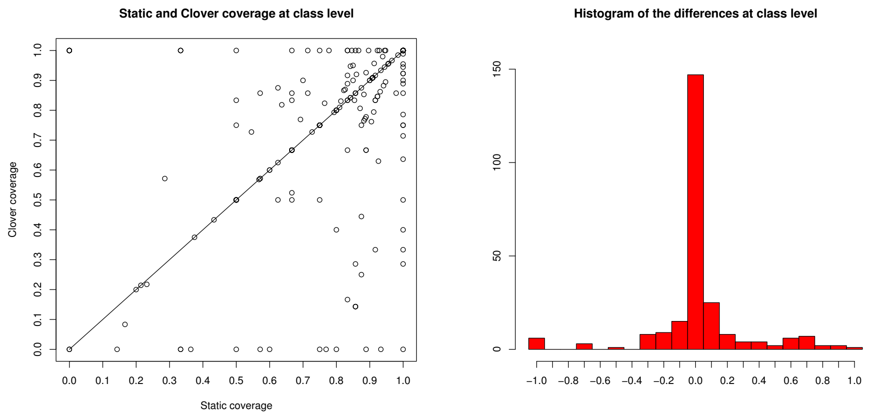
The results for the Collections project are scrutinized below. Since similar results were observed for other systems they will not be shown.

Collections system analysis Figure 4.10 shows two histograms for the distributions of estimated (static) and true (Clover) class-level coverage for the Collections library. The figure reveals that static coverage was accurate to estimate true coverage in all ranges, with minor oscillations in the 70–80% and in the 80–90% ranges, where a



(a) Static coverage (b) Dynamic (Clover) coverage

Figure 4.10: Histograms of static and dynamic (Clover) class coverage for Collections

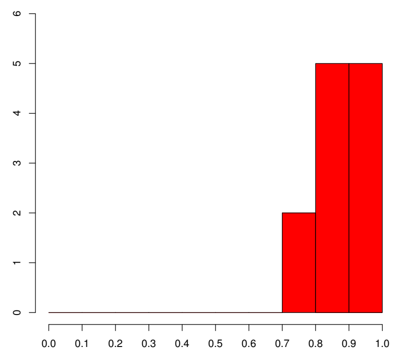


(a) Static vs. dynamic coverage (b) Histogram of differences

Figure 4.11: Scatter of static and Clover coverage and histogram of the coverage differences for Collections at class level.

lower and higher number of classes was recognized, respectively.

Figure 4.11 depicts a scatter plot for estimate and true value (with a diagonal line where the estimate is correct), and a histogram of the differences between estimated and true values. The scatter plot shows that several points are on the diagonal, and an similar number of points lies above the line (underestimation) and below the line (overestimation). The histogram shows that for a big number of classes static coverage matches Clover coverage (difference between -0.5% and 0.5% coverage). This can


(a) Static coverage

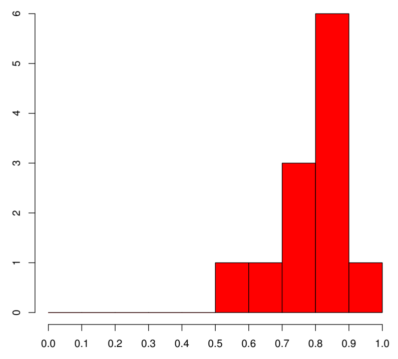

(b) Dynamic (Clover) coverage

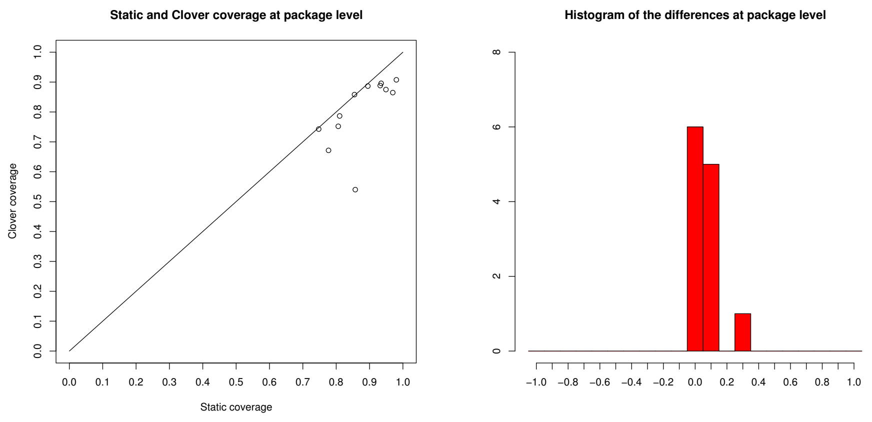
Figure 4.12: Histograms of package-level static and Clover coverage for Collections.

be observed by a high bar represented above 0. On both sides of this bar, differences decrease resembling a normal distribution. For a large number of classes estimated coverage matched true coverage. This can be observed in the scatter plot, in which several points are on the diagonal, and in the histogram, by the high bar above 0. The histogram additionally shows that, on both sides of this bar, differences decrease indicating a small estimation error.

Recalling Spearmans's correlation value, 0.549, we understand that the correlation is not higher due to the considerable number of classes for which static coverage overestimates or underestimates results without a clear trend.

Package-level results are shown in Figure 4.12. In contrast to class-level results, the histograms at package level do not look so similar. In the 50–70% range static estimate fails to recognize coverage. Comparing to Clover, in the 70–80% and 90–100% ranges static estimate reports lower coverage and in the 80–90% range reports higher coverage.

Figure 4.13 shows the scatter plot and the histogram of differences at the package level. In the scatter plot we can observe that for a significant number of packages, static coverage was overestimated. However, in the histogram, we see that for 6 packages estimates are correct, while for the remaining 5 packages, estimates are slightly



(a) Static vs. dynamic (b) Histogram of difference

Figure 4.13: Scatter of static and Clover coverage and histogram of the coverage differences for Collections at package level.

overestimated. This is in line with the correlation value of 0.776, of Table 4.4.

Thus, for the Collections project, the estimated coverage at class and package level can be considered good.

4.4.3 Evaluation

Static estimation of coverage is highly correlated with true coverage at all levels for a large number of projects. The results at system level allow us to positively answer the question: can test coverage be determined without running tests? The tradeoff, as we have shown, is some precision loss with a mean of the absolute differences around 9%. The analysis on 52 releases of the Utils project provides additional confidence in the results for system coverage. As observed, static coverage not only can be used as predictor for real coverage, but it also detects coverage fluctuations.

According to the expectations, static coverage at package level reports better correlation than at class level. However, the correlation for all systems at package level is just slightly higher than for class level. Grouping classes was expected to possibly cancel more imprecision and hence provide better results. However, this is not always the case. For a small number of packages, static coverage produces large overestimations or underestimations, causing outliers, having a negative impact in both correlation and dispersion.

At class level, the correlation values are quite high, but the dispersion of differences is still high, meaning that precision at class level could be further improved.

As can be observed, control flow and dynamic dispatch cause an overestimation, while frameworks/libraries cause underestimation of coverage which, in some cases, is responsible for coverage values lower than true coverage.

4.5 Static coverage as indicator for solving defects

Risk and rating thresholds are derived from the benchmark described in Chapter 1, using the techniques introduced in Chapter 2 and 3, respectively. Static coverage rating was used as internal quality metric and an experiment was set up to validate it against external quality captured by different defect resolution efficiency metrics extracted from an ITS. The correlation analysis between internal and external metrics is reported and analyzed. The correlation between static coverage rating and different metrics for defect resolution efficiency from an ITS was analyzed for several releases of software systems.

4.5.1 Risk and rating thresholds

Figure 4.14 shows a cumulative quantile plot depicting the distribution of the SETC metric. Each gray line represents one out of 78 systems of the benchmark described in Section 1.4. The black line characterizes the SETC metric for those 78 systems, by applying the relative size aggregation technique introduced in Section 2.4.3. The y-axis represents the SETC metric values, and the x-axis represents the percentage of volume of classes (measured in SLOC).

Although the benchmark defined in Section 1.4 consists of 100 systems written in both Java and C#, for the SETC metric only 78 systems are taken into account.

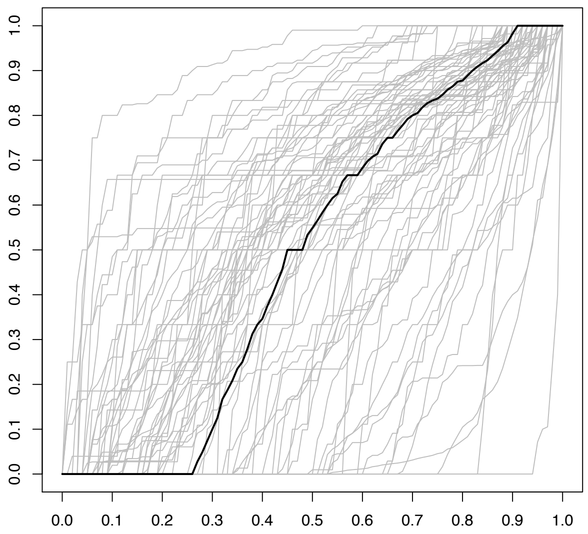


Figure 4.14: Distribution of the SETC metric for 78 Java systems of the benchmark. Each line depicted in gray represents an individual system. The black line characterizes the distribution of all benchmark systems. The y-axis represents coverage values per class and the x-axis represents the percentage of size (SLOC) of all classes.

This is due to two reasons. First, the SETC analysis is only available for Java thus far. Second, only systems that have more than 5% of test coverage were considered to avoid the analysis of systems with just a couple of tests.

Focusing on the black line of Figure 4.14, we observe that for up to 25–26% of all classes (x-axis) the SETC is 0% (y-axis). This means that for a software system, typically around 25% of its volume is not test covered. Then, from around 25% to 90% of the volume, we observe a linear growth in terms of coverage indicating that there is a linear growth of coverage for the classes. Finally, from around 90% upwards, in volume, classes have 100% of coverage, indicating that only 10% of a system classes are fully test covered (100% of test coverage).

When comparing the SETC metric distribution to the distributions of the Software Improvement Group (SIG) quality model metrics, presented in Section 2.6, we can observe two main differences. The first difference is the distribution shape: the SETC metric does not have an exponential distribution as the metrics of the SIG qual-

Table 4.5: Risk and rating thresholds for the static coverage metric. The risk thresholds are defined in the headers, and the rating thresholds are defined in the table body.

Star rating	Low risk $]83.74, 100]$	Moderate risk $]54.88, 83.74]$	High risk $]0, 54.88]$	Very-high risk 0
★★★★★	-	72.04	40.09	9.82
★★★★☆	-	78.71	48.77	19.84
★★★☆☆	-	84.26	58.22	34.28
★★☆☆☆	-	93.07	77.44	53.05

ity model. Instead, the SETC metric follows a normal-like distribution, which can be confirmed by the fact that Figure 4.14 resembles the cumulative distributions of a normal probability function. The second difference is that while for the SIG quality model metrics, large values are associated with higher risks, for the SETC metric this is the opposite. Small values of the SETC indicate higher risk, as it indicates lack of coverage, and higher values of the SETC indicate lower risk, as it indicates full coverage.

Risk thresholds for the SETC metric are derived using the methodology introduced in Section 2.3. As input for the methodology 25%, 50% and 75% quantiles were chosen since this allows to create equal risk categories, i.e., each risk category represents equally 25% of the code. The choice of these quantiles is justified due to the shape of the distribution, i.e., there is a clear distinction among metric values. The outcome of the methodology to derive thresholds, using the above defined quantiles, was 0, 54.88 and 83.74. Table 4.5 shows the risk intervals defined by these thresholds. We consider as Low risk code all classes that have a coverage in the $]83.74, 100]$ interval, Moderate risk in the $]54.88, 83.74]$ interval, High risk in the $]0, 54.88]$ interval, and Very-high risk all classes that are not covered (0% of coverage).

Identical to what was done in Chapter 3, ratings for the SETC metric were calibrated using a 5-point scale and a distribution of 20–20–20–20–20. This means that the SETC metric will have a rating from 1 to 5 stars, each star representing equally 20% of the systems of the benchmark. However, it is not possible to equally distribute

78 systems by 5 ratings (as this would give a total of 15.6 systems per rating). Hence, a partition containing 15–16–16–16–15 systems is defined, representing a distribution of 19.23–20.51–20.51–20.51–19.23. Table 4.5 shows the calibrated rating thresholds.

Using the SETC risk and rating thresholds the ratings for all the 78 systems of the benchmark are calculated. This was done to verify if the expected distribution was in fact met, i.e., if each star rating represents around 20% of the systems. It was found, that the ratings do not follow exactly the partition defined for 78 systems - some ratings have 1 system more and other ratings have 1 system less, which was due to ties (two systems having identical risk profiles). Since the deviations were small, by only 1 system, we considered them as insignificant.

Is the SETC rating correlated with SETC at system-level? The rational behind this is that if ratings are meant to be a representative means to aggregate metrics at high-level then there should be a significant and positive correlation between SETC ratings and SETC values at system-level. In order to investigate this we started by analyzing if the data from both both metrics are normally distributed using Anderson-Darling test [7]. For both SETC rating and SETC at system-level it was not possible to reject normality (the p-value for both tests was greater than 0.05) and, hence, one can use Pearson correlation test. Computing the Pearson correlation between SETC rating and SETC at system-level, resulted in a value of 0.809 with a p-value of 0.001. This indicates high correlation with high significance, meaning that ratings are representative of the metric at system-level.

4.5.2 Analysis of defect resolution metrics

To validate the SETC rating against an external quality metric a benchmark of ITS metrics is used. Table 4.6 characterizes the benchmark showing the systems[6], the number of releases analyzed, the size for the latest release (in SLOC), number of total

[6]http://ant.apache.org/, http://argouml.tigris.org/, http://checkstyle.sourceforge.net/, http://www.hibernate.org/, http://hsqldb.org/, http://ibatis.apache.org/, http://jabref.sourceforge.net/, http://jmol.sourceforge.net/, http://logging.apache.org/log4j/, http://lucene.apache.org/, http://www.omegat.org/, http://www.springsource.org/, http://www.stripesframework.org/, http://subclipse.tigris.org/

Table 4.6: Characterization of the benchmark used for external quality validation. ITS metrics were derived for each system.

System	# versions	SLOC (latest)	Total issues	Total defects
Ant	7	100,340	25,608	17,733
ArgoUML	9	162,579	11,065	8,568
Checkstyle	7	47,313	5,154	2,696
Hibernate-core	3	105,460	10,560	6,859
HyperSQL	6	68,674	6,198	4,890
iBATIS	4	30,179	2,496	1,453
JabRef	5	82,679	4,616	3,245
Jmol	4	91,631	2,090	1,672
log4j	5	23,549	4,065	3,357
Lucene	4	81,975	37,036	32,586
OmegaT	5	111,960	3,965	1,857
Spring Framework	4	144,998	23,453	11,339
Stripes Framework	4	17,351	2,732	1,359
SubEclipse	4	92,877	3,469	2,585
$N = 14$	71	1,161,565	142,507	100,199

issues (open and resolved) and number of total defects (open and resolved). This benchmark is being collected by SIG as an effort to validate their quality model for maintainability. Further descriptions about this benchmark and results of the validation of the SIG quality model can be found in Bijlsma [16], Bijlsma et. al [17], Luijten and Visser [63] and Athanasiou [9].

To validate the SETC metric against external quality, two ITS metrics initially[7] defined by Bijlsma [16] are adopted: Throughput and Productivity. These two ITS metrics are defined as indicators for issue handling efficiency, i.e., as quality indicators for how developers solve issues reported about a software system.

Throughput is defined as follows:

$$throughput = \frac{\#\ resolved\ issues\ per\ month}{KLOC}$$

Throughput measures the overall efficiency of the team based on resolved issues. This

[7]The metrics defined by Bijlsma [16] are called project and developer productivity. However, in later works they were redefined and renamed as throughput and productivity, respectively. The later designation is also used by Athanasiou [9].

metric is normalized per month, to minimize fluctuations caused by specific events (e.g. vacations) and by size, measured in SLOC, to allow comparison among projects of different sizes.

Productivity is defined as follows:

$$productivity = \frac{\#\ resolved\ issues\ per\ month}{\#\ developers}$$

Productivity measures the efficiency of the developers based on resolved issues. This metric is again normalized per month, and per number of developers.

We want to validate SETC rating against these metrics. The rationale is that systems with a higher coverage allow developers to do changes faster, not only because it gives developers more confidence in changing the code with less impact, but also because tests act as a form of documentation helping developers to understand the behavior of code. Hence two hypotheses to validate are defined:

Hypothesis 1 SETC rating is positively correlated with Throughput.

Hypothesis 2 SETC rating is positively correlated with Productivity.

To validate the SETC rating with both metrics, we first analyze if the benchmark data follows a normal distribution using Anderson-Darling test [7] in order to chose between Pearson and Spearman correlations tests. Using Anderson-Darling test to check if the SETC rating for the ITS benchmark results in a p-value smaller than 0.05, which means that it is not possible to assume normally distributed data and hence Spearman correlation test will have to be used, choosing the standard 95% of significance.

Hypothesis 1 Using the Spearman correlation test between SETC ratings and Throughput results in a correlation value of 0.242 with a p-value of 0.047. The p-value is smaller that 0.05, indicating a significant correlation. The correlation value is positive but low. This confirms that there is a significant positive correlation between the two metrics. However, since the correlation is low the SETC rating cannot be assumed as a good predictor for Throughput.

Hypothesis 2 Using Spearman correlation test between SETC rating and Productivity results in a correlation value of 0.439 with a p-value of 0.001. The p-value is smaller than 0.01, indicating a highly significant correlation. The correlation value is positive and moderate. This moderate correlation and highly significance level indicates that SETC rating predicts Productivity. This result indicates that for systems with a higher SETC rating the developers have higher productivity in resolving issues.

4.5.3 Evaluation

This chapter started by introducing a methodology to estimate test coverage based on static analysis, SETC. The SETC metric proves to not only correlate with real coverage but also SETC ratings correlate with the efficiency of developers resolving issues. These findings show promising results for both the SETC metric and for the use of risk and rating thresholds approach introduced in Chapters 2 and 3.

The empirical validation used in this section was inspired on previous work by Athanasiou [9] and Bijlsma [16]. Hence, the threats of validity (construct, internal, external and conclusion) are similar and for this reason deferred to those works.

In spite of good results about the correlations found it would be interesting to further extend the benchmark of systems used for validation. This would allow stronger claim about how coverage rating affects developer productivity in resolving issues.

4.6 Related Work

No attempt to compute test coverage using static analysis was found in the literature. However, there is a long record of work sharing similar underlying techniques.

Koster et al. [54] introduce a new test adequacy criterion, *state coverage*. A program is state-covered if all statements that contribute to its output or side effects are covered by a test. The granularity of state coverage is at the statement level, while our technique is at the method level. State coverage limits coverage to system only, while we report system, package and class coverage. State coverage also uses static analysis and slicing. However, while a data flow graph from bytecode is used, SETC uses a call graph extracted from source code. Koster et al. do not identify sources of imprecision, and use as case study a small OSS project. In contrast, this chapter describes sources of imprecision and presents a comparison using several projects ranging from proprietary and OSS and from small to large sizes.

Ren et al. [72] propose a tool, Chianti, for change impact analysis. The tool analyses two versions of a program, original and modified. A first algorithm identifies tests potentially affected by changes, and a second algorithm detects a subset of the code potentially changing the behavior of tests. Both algorithms use slicing on a call graph annotated with change information. Ren et al. use dynamic analysis for deriving the graph. However, in a previous publication of Chianti [75], Ryder et al. used static analysis. Our technique also makes use of graph slicing at method level granularity with the purpose of making the analysis more amenable. Chianti performs slicing twice. First, from production code to test code and second from tests to production code. By contrast, SETC requires slicing to be performed only once, from tests to production code, to identify production code reached by tests. Finally, despite using a similar technique to Chianti the purpose of SETC is to estimate test coverage.

Binkley [18] proposes a regression test selection (RTS) technique to reduce both the program to be tested and the tests to be executed. This technique is based on two algorithms. The first algorithm extracts a smaller program, *differences*, from the semantics differences between a previously tested program, *certified*, and the program to be tested, *modified*. The second algorithm identifies and discards the tests for which *certified* and *differences* produce the same result, avoiding the execution of unnecessary tests. Both algorithms make use of static slicing (backward and forward) over a system dependence graph, containing statement-level and control-flow information.

By contrast, SETC uses only forward slicing over a call graph, which contains less information and requires a simpler program analysis to construct.

Rothermel and Harrold [73], present comprehensive surveys in which they analyze and compare thirteen techniques for RTS. As previously stated, RTS techniques attempt to reduce the number of tests to execute, by selecting only those who cover the components affected in the evolution process. RTS techniques share two important ingredients with our technique: static analysis and slicing. However, while most RTS techniques use graphs with detailed information, e.g., system dependence graphs, program dependence graphs, data flow graphs, SETC uses less detailed information. Moreover, SETC also shares with these techniques the basic principles. RTS techniques analyze code that is covered by tests in order to select which tests to run. SETC, on the other hand, analyzes code under test in order to estimate coverage.

Harrold [39] and Bertolino [15] present a survey about software testing and research challenges to be met. Testing is a challenging and expensive activity and there are ample opportunities for improvement in, for instance, test adequacy, regression test selection and prioritization. This chapter shows that SETC can be used to assess test adequacy. Rothermel et al. [74] surveys nine test prioritization techniques from which four are based on test coverage. These four techniques assume the existence of coverage information produced by prior execution of test cases. Our technique could be used as input replacing the execution of tests. Finally, Lyu [64] has surveyed the state of the art of software reliability engineering. Lyu describes and compares eight reports of the relation between static coverage and reliability. In the presence of a very large test suite, SETC could be used to substitute the coverage value by an approximation and used as input of a reliability model.

4.7 Summary

An approach for estimating code coverage through static analysis was described. This approach does not require detailed control or data flow analysis in order to scale to very large systems and it can be applied to incomplete source code.

The sources of imprecision of the analysis are discussed leading to experimental investigation of their accuracy. The experiments comparing static and dynamic analysis show a strong correlation at system, package and class level. The correlation at system-level is higher than at package and class levels, indicating opportunities for further improvement.

The use of risk and rating thresholds was investigated as an approach to evaluate test quality on a 1 to 5 star rating basis. Experimental investigation of the correlation between this rating and indicators for performance of issue resolution and revealed that higher static coverage rating indicates better developer performance resolving issues.

Chapter 5

Assessment of Product Maintainability for Two Space Domain Simulators

The software life-cycle of applications supporting space missions follows a rigorous process in order to ensure the application complies with all the specified requirements. Ensuring the correct behavior of the application is critical since an error can lead, ultimately, to the loss of a complete space mission. However, it is not only important to ensure the correct behavior of the application but also to achieve good product quality since applications need to be maintained for several years. Then, the question arises: *is a rigorous process enough to guarantee good product maintainability?*

In this chapter the software product maintainability of two simulators used to support space missions is assessed. The assessment is carried out using both a standardized analysis, using the Software Improvement Group (SIG) quality model for maintainability, and a customized copyright license analysis. The assessment results reveal several quality problems leading to three lessons. First, rigorous process requirements by themselves do not ensure product quality. Second, quality models can be used not only to pinpoint code problems but also to reveal team issues. Finally, tailored analyses, complementing quality models, are necessary for in-depth quality investigation.

5.1 Introduction

A space mission running a satellite is a long-term project that can take a decade to prepare and that may run for several decades. Simulators play an important role in the overall mission. Before the spacecraft launch, simulators are used to design, develop and validate many spacecraft components; validate communications and control infrastructure; train operations procedures. After the launch, they are used to diagnose problems or validate new conditions (e.g. hardware failure, changes in communication systems).

During such a long period of time, inevitably, glitches in both hardware and software will appear. To minimize the impact of these problems it has become clear that standards are necessary to achieve very high quality [51]. The first standards were defined in 1977 [51] being currently under administration of the European Cooperation for Space Standardization (ECSS). The ECSS[1] is represented by the European Space Agency (ESA), the Eurospace[2] organization representing European space industry and several European national space agencies.

ECSS standards are enforced by ESA and applicable to all projects developed for the space domain, covering project management, product assurance and space engineering. Two standards are specific for software. The space engineering software standard [31] defines the requirements for the software life-cycle process. The software product assurance standard [32] defines quality requirements for software development and maintenance activities.

In the ECSS space engineering standard [31] a rigorous software process is defined. This includes clear project phases (e.g. requirements), activities which can determine if the project is continued (e.g. reviews), and deliverables to be produced (e.g. documents). For example, in the requirements phase the Requirement Baseline document must be produced. Only after the review and approval of this document to be done

[1]http://www.ecss.nl/
[2]http://www.eurospace.org/

by all the parties involved in the System Requirements Review phase, the project is allowed to continue.

In the ECSS software product assurance standard [32] the use of a quality model is considered. However, the standard does not define or provide recommendations for any specific quality model. This omission in the standard explains why, no quality model being enforced, software suppliers are given the freedom to choose or propose a model. As consequence, the product quality of space software, in practice, relies only on the strict process standards.

The question arises: *is the current rigorous process enough to guarantee good product quality?* To answer this question, it was analzed the software product maintainability of two simulators used in the space domain, developed with similar standards: EuroSim and SimSat. The EuroSim is a commercially available simulator system developed by a consortium of companies. The SimSat is a simulator owned by ESA and developed by external companies selected via a bidding process.

Both EuroSim and SimSat were analyzed using the SIG quality model for maintainability [41], based on the ISO/IEC 9126 standard for software product quality [44]. Additionally, for EuroSim a custom analysis of the copyright licenses was performed to check for possible software distribution restrictions.

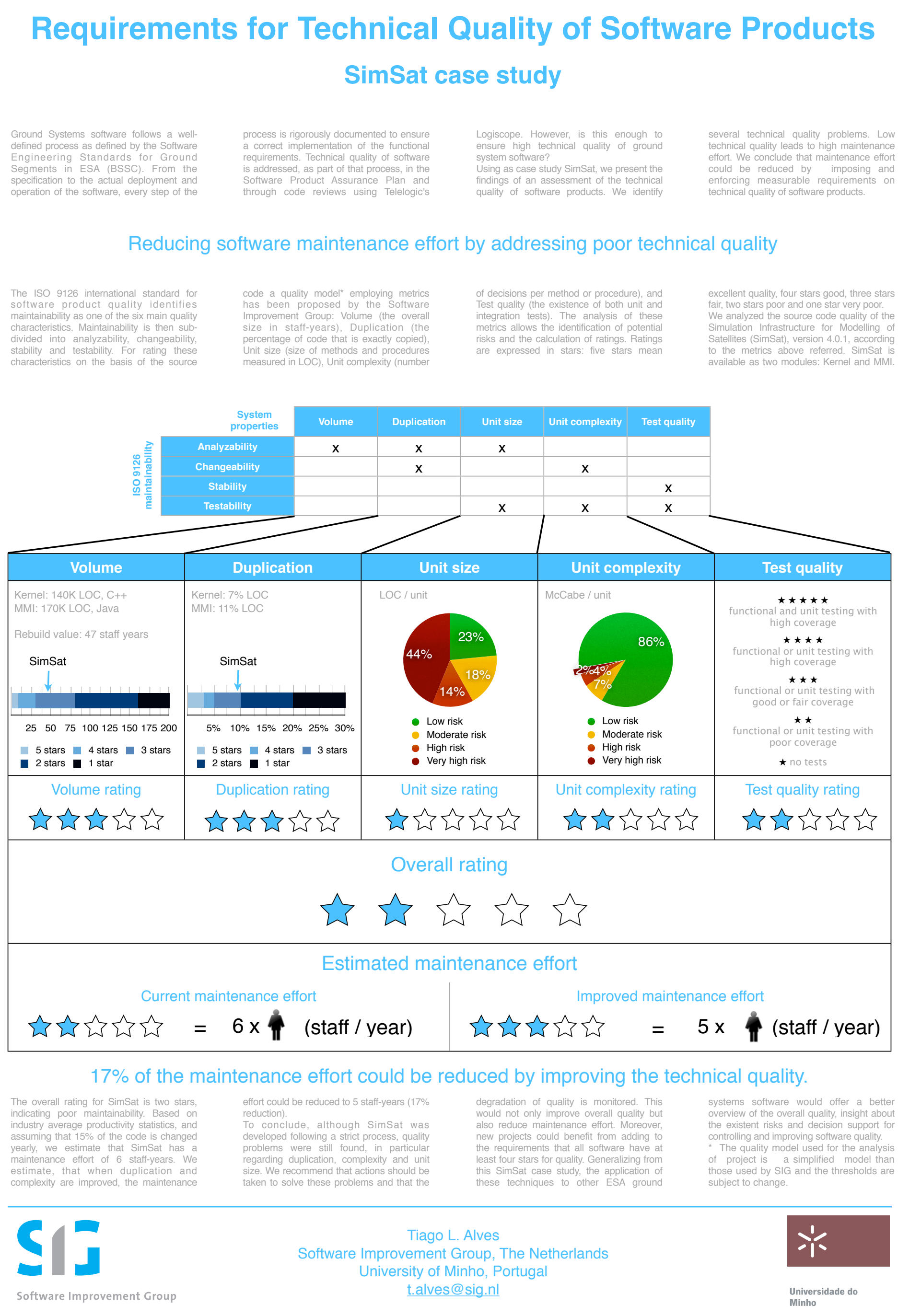
From the results of the analysis three lessons were learned.

i) Rigorous process requirements do not assure good product maintainability, supported by the fact that both the EuroSim and SimSat ranked less than four stars in the SIG quality model.

ii) Quality models can reveal team problems, supported by the discovery that some of the code issues pinpointed by the quality model could be attributed to specific teams involved in the project.

iii) Tailored analyses are necessary for further investigation of product quality, supported by the discovery of code structure problems using copyright license analysis.

We conclude that having a quality model is a fundamental element to achieve good quality allowing to pinpoint potential problems and monitor quality degradation. Also,



Requirements for Technical Quality of Software Products

SimSat case study

Ground Systems software follows a well-defined process as defined by the Software Engineering Standards for Ground Segments in ESA (BSSC). From the specification to the actual deployment and operation of the software, every step of the process is rigorously documented to ensure a correct implementation of the functional requirements. Technical quality of software is addressed, as part of that process, in the Software Product Assurance Plan and through code reviews using Telelogic's Logiscope. However, is this enough to ensure high technical quality of ground system software?
Using as case study SimSat, we present the findings of an assessment of the technical quality of software products. We identify several technical quality problems. Low technical quality leads to high maintenance effort. We conclude that maintenance effort could be reduced by imposing and enforcing measurable requirements on technical quality of software products.

Reducing software maintenance effort by addressing poor technical quality

The ISO 9126 international standard for software product quality identifies maintainability as one of the six main quality characteristics. Maintainability is then sub-divided into analyzability, changeability, stability and testability. For rating these characteristics on the basis of the source code a quality model* employing metrics has been proposed by the Software Improvement Group: Volume (the overall size in staff-years), Duplication (the percentage of code that is exactly copied), Unit size (size of methods and procedures measured in LOC), Unit complexity (number of decisions per method or procedure), and Test quality (the existence of both unit and integration tests). The analysis of these metrics allows the identification of potential risks and the calculation of ratings. Ratings are expressed in stars: five stars mean excellent quality, four stars good, three stars fair, two stars poor and one star very poor.
We analyzed the source code quality of the Simulation Infrastructure for Modelling of Satellites (SimSat), version 4.0.1, according to the metrics above referred. SimSat is available as two modules: Kernel and MMI.

ISO 9126 maintainability / System properties	Volume	Duplication	Unit size	Unit complexity	Test quality
Analyzability	x	x	x		
Changeability		x		x	
Stability					x
Testability			x	x	x

17% of the maintenance effort could be reduced by improving the technical quality.

The overall rating for SimSat is two stars, indicating poor maintainability. Based on industry average productivity statistics, and assuming that 15% of the code is changed yearly, we estimate that SimSat has a maintenance effort of 6 staff-years. We estimate, that when duplication and complexity are improved, the maintenance effort could be reduced to 5 staff-years (17% reduction).
To conclude, although SimSat was developed following a strict process, quality problems were still found, in particular regarding duplication, complexity and unit size. We recommend that actions should be taken to solve these problems and that the degradation of quality is monitored. This would not only improve overall quality but also reduce maintenance effort. Moreover, new projects could benefit from adding to the requirements that all software have at least four stars for quality. Generalizing from this SimSat case study, the application of these techniques to other ESA ground systems software would offer a better overview of the overall quality, insight about the existent risks and decision support for controlling and improving software quality.
* The quality model used for the analysis of project is a simplified model than those used by SIG and the thresholds are subject to change.

Software Improvement Group

Tiago L. Alves
Software Improvement Group, The Netherlands
University of Minho, Portugal
t.alves@sig.nl

Universidade do Minho

Figure 5.1: Poster presented to European Ground System Architecture Workshop (ESAW'09) in the early phase of the work which lead to Chapter 5 of this dissertation.

quality models should be complemented with tailored analysis in order to check for further potential problems.

This chapter is structured as follows. Section 5.2 provides a description of the two analyzed simulators. Section 5.3 introduces the quality model and the copyright license analysis used to evaluate product quality. Section 5.4 presents the results of the quality assessment and describes the quality issues found and how they could be prevented. Summary of the lessons learned is presented in Section 5.6 and of the contributions in Section 5.7.

5.2 Systems under analysis

This section provides a brief overview of the two simulators analyzed: EuroSim (European Real Time Operations Simulator) and SimSat (Simulation Infrastructure for the Modeling of Satellites).

5.2.1 EuroSim

EuroSim is a commercial simulator[3] developed and owned by a consortium of companies including Dutch Space, NLR and TASK24[4].

The development of EuroSim started in 1997. It is mainly developed in C/C++, supporting interfaces for several programming languages (e.g. Ada, Fortran, Java and MATLAB). EuroSim supports hard real-time simulation with the possibility of hardware-in-the-loop and/or man-in-the-loop additions.

EuroSim is used to support the design, development and verification of critical systems. These include, for instance, the verification of spacecraft on-board software, communications systems and other on-board instruments. Additionally, outside the space domain, EuroSim has been used for combat aircraft training purposes and to simulate autonomous underwater vehicles.

[3]http://www.eurosim.nl/

[4]http://www.dutchspace.nl/ http://www.nlr.nl/ http://www.task24.nl

For the analysis EuroSim mk4.1.5 was used.

5.2.2 SimSat

SimSat is a simulator owned by ESA[5], developed and maintained by different external companies chosen via a bidding process. In contrast to EuroSim, SimSat is not a commercial tool and is freely available to any member of the European Community.

The development of SimSat started in 1998 but its code has been rewritten several times. The analyzed version is based on the codebase developed in 2003. SimSat consists of two main modules: the simulation kernel, developed in C/C++; and the Graphical User Interface (GUI), developed in Java using Eclipse RCP [66]. Only soft real-time simulation is supported.

SimSat is used for spacecraft operation simulations. This involves the simulation of the spacecraft state and control communication (housekeeping telemetry and control). The on-board instruments (payload) are not simulated. The simulator is used to train the spacecraft operator team and validate operational software, such as the systems to control satellites, ground station antennas and diverse network equipment.

For the analysis SimSat 4.0.1 issue 2 was used.

EuroSim and SimSat have three commonalities. First, they are used to support the validation of space sub-systems. Second, according to companies involved in the development of EuroSim [33] and SimSat [80], they are both used for the simulation of (different) components of the European Global Navigation System (Galileo). Third, both EuroSim and SimSat were developed using strict equivalent software process standards, compatible with the ECSS standards.

[5]http://www.egos.esa.int/portal/egos-web/products/Simulators/SIMSAT/

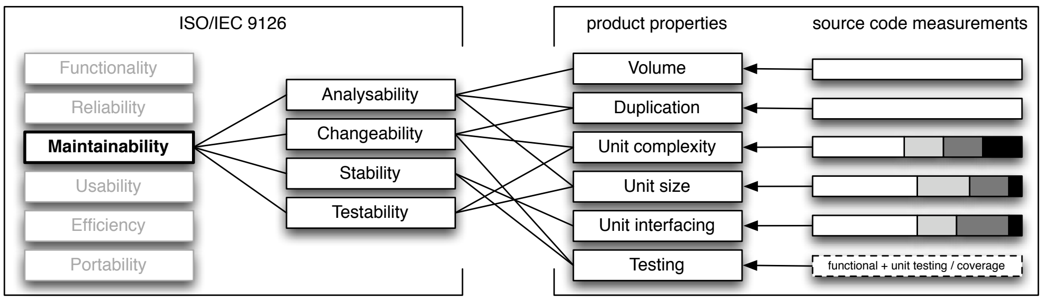


Figure 5.2: Quality model overview. On the left-hand side, the quality characteristics and the maintainability sub-characteristics of the ISO/IEC 9126 standard for software product quality are shown. On the right-hand side, the product properties defined by SIG and its relation with the maintainability sub-characteristics are shown. In the source code measurements, the empty rectangle indicates the use system-level measurements, the four-piece rectangles indicate measurements aggregated using risk profiles, and the dashed-line rectangle indicates the use of criteria. This figure is adapted from Luijten et al. [63].

5.3 Software analyses

Two types of analyses were done. One standardized analysis, applied to both EuroSim and SimSat, using the SIG quality model for maintainability. One custom analysis, applied only to EuroSim, for copyright license detection.

5.3.1 SIG quality model for maintainability

The ISO/IEC 9126 standard for software product quality [44] defines a model to characterize software product quality according to 6 main characteristics: functionality, reliability, maintainability, usability, efficiency and portability. To assess maintainability, SIG developed a layered model using statically derived source code metrics [41]. An overview of the SIG quality model and its relation to the ISO/IEC 9126 standard is shown in Figure 5.2.

The SIG quality model has been used for software analysis [41], benchmarking [26] and certification [27] and is a core instrument in the SIG consultancy services. Also, Bijlsma [16], Bijlsma et. al [17], Luijten and Visser [63] found empirical evidence that systems with higher technical quality have higher issue solving efficiency.

The model is layered, allowing to drill down from the maintainability level, to sub-characteristic level (as defined in the ISO/IEC 9126: analyzability, changeability, stability and testability), to individual product properties. Quality is assessed using a five star ranking: five stars is used for very-good quality, four stars for good, three stars for moderate, two stars for low and one star for very-low. The star ranking is derived from source code measurements using thresholds calibrated using a large benchmark of software systems [27].

To assess maintainability, a simplified version of the quality model was used using the product properties described below. A short description of each product property is provided.

***Volume*:** measures overall system size in staff-months (estimated using the Programming Languages Table of Software Productivity Research LLC [59]). The smaller the system volume, the smaller the maintenance team required avoiding communication overhead of big teams.

***Duplication*:** measures the relative amount of code that has an exact copy (clone) somewhere else in the system. The smaller the duplication the easier to do bug fixes and testing, since functionality is specified in a single location.

***Unit size*:** measures the size of units (methods or functions) in Source Lines of Code (SLOC). The smaller the units the lower the complexity and the easier it is to understand and reuse.

***Unit complexity*:** measures the McCabe cyclomatic complexity [67] of units (methods or functions). The lower the complexity the easier to understand, test and modify.

***Unit interfacing*:** measures the number of arguments of units (methods or functions). The smaller the unit interface the better encapsulation and therefore the smaller the impact of changes.

***Testing*:** provides an indication of how testing is done taking into account the presence of unit and integration testing, usage of a test framework and the amount of test cases. The better the test quality the better the quality of the code.

As can be observed, the ratings for product properties are derived in different ways.

Volume and *Duplication* are calculated at system level.

Unit size, *Unit complexity* and *Unit interfacing* are metrics calculated at method or function level, and aggregated to system level using risk profiles. A risk profile characterizes a metric through the percentages of overall lines of code that fall into four categories: low risk, moderate risk, high risk and very-high risk. The methods are categorized in these categories using metric thresholds. The ratings are calculated using (a different set of) thresholds to ensure that five stars represent 5% of the (best) systems, four, three and two stars represent each 30% of the systems, and that one star represents the 5% of the (worse) systems.

Finally, the rating for *Test quality* is derived using the following criteria: five stars for unit and functional testing with high coverage; four stars for unit or function testing with high coverage; three stars for unit or functional testing with good or fair coverage; two stars for function or unit testing with poor coverage; one star for no tests. This method was taken from [27]. Alternatively, a method for estimating test quality as the one introduced in Chapter 4 could be used. However, the use of this method would require a C/C++ implementation which, at the time of this work was carried out, was not available.

For a more detailed explanation of the quality model, are referred to references [41, 25] and [11].

5.3.2 Copyright License Detection

A customized analysis was developed to find and extract copyright licenses used in EuroSim. This analysis is identical to the one the author used in [3] to identify library code. The analysis of copyright licenses was done to investigate if any of the used licenses poses legal restrictions in the distribution of EuroSim.

Table 5.1: Criteria to determine the implications of the use of licensed source code.

	Open-source license		Other license	
	copyleft	copyright	consortium	external
Mandate distribution of library changes?	yes	yes	no	investigate
Mandate all software distribution under same license?	yes	no	no	investigate

The analysis of copyright license was executed in two steps. First, by implementing and running an automatic script to detect copyright and license statements. Second, by manually checking each license type using Table 5.1.

The developed script was implemented by defining regular expressions in *grep* [12], to match keywords such as *license*, *copyright*, and *author*. Although the approach is simple, it enables a powerful and generic way of detecting copyright statements. This is necessary since there is no standardized way to specify copyright information (this is mostly available as free form text in code comments).

Such a copyright statement list was then manually processed to detect false positives (keywords recognized that do not refer to actual licenses or authorship information). After validation, false positives were removed from the list.

Table 5.1 was then used to help checking if a license found poses any risk to the project. For instance, finding an Open-Source Software (OSS) copyleft license, such as the GNU GPL license, would not only mandate to distribute library changes (if any) but also mandate EuroSim to be available under the GNU GPL license. As consequence, this would legally allow the free or commercial distribution of EuroSim by third-parties. The use of OSS copyright licenses, or licenses from consortium companies does not pose any restriction in software distribution. However, should external licenses be found then, this condition should be investigated.

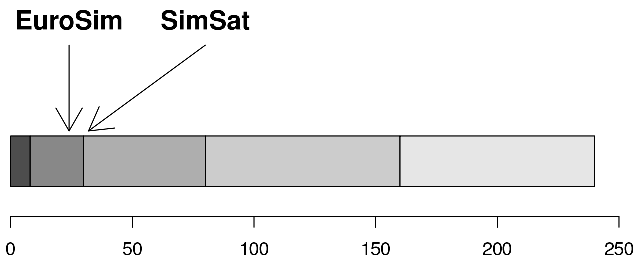


Figure 5.3: Volume comparison for EuroSim and SimSat (scale in staff–year).

5.4 Analysis results

This section describes the results of the application of the SIG quality model for maintainability to both EuroSim and SimSat, and those concerning the copyright license detection analysis done for EuroSim.

5.4.1 Software Product Maintainability

Using the SIG quality model, described in Section 5.3.1, EuroSim ranks three stars while SimSat ranks two stars. The following sections provide a more detailed analysis of the 6 product properties measured by the quality model: Volume, Duplication, Unit complexity, Unit size, Unit interfacing and Testing. The first two metrics are measured at system-level and are presented in a scale, from five stars to one star, read from left to right. All the other metrics, except testing, are measured at unit level and are presented in a pie-chart, in which very-high risk is depicted in black, high risk in gray, moderate risk in light gray, and low risk in white.

Volume

Figure 5.3 compares the volume of EuroSim and SimSat. EuroSim contains 275K SLOC of C/C++ and 4K SLOC of Java, representing an estimated rebuild value of 24 staff–year, ranking four stars. SimSat contains 122K SLOC of C/C++ and 189K SLOC of Java, representing an estimated rebuild value of 32 staff–year, ranking three stars.

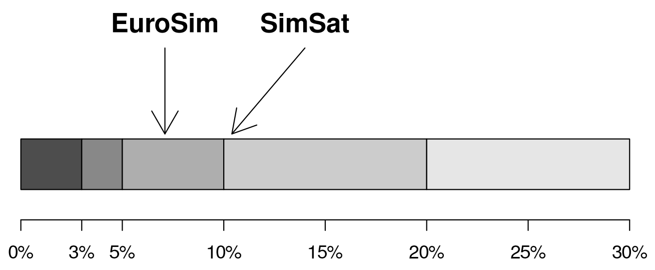


Figure 5.4: Duplication comparison for EuroSim and SimSat.

For EuroSim, the Java part is responsible only for exposing API access to Java programs. Both GUI and core of the application are developed in C/C++.

For SimSat, the Java part is responsible for the GUI, while the C/C++ is responsible for the core of the application. It is interesting to observe that the SimSat GUI is larger than the core of the application.

Since both simulators rank three stars or higher for volume, this indicates that the maintenance effort is possible with a small team composed by a couple of elements, maintenance effort being smaller for EuroSim than for SimSat.

Duplication

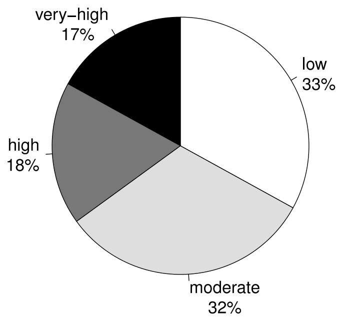
Figure 5.4 compares the duplication of EuroSim and SimSat. EuroSim contains 7.1% of duplication, ranking three stars, while SimSat contains 10.4% of duplication, ranking two stars.

In both EuroSim and SimSat, duplication problems were found in several modules, showing that this is not a localized problem. Also, for both systems, duplicated files were found.

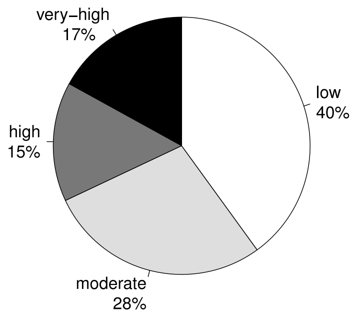
For EuroSim, we uncovered several clones in the library code supporting three different operating systems. This fact surprised the EuroSim maintainers as they expected the code to be completely independent.

For SimSat, surprisingly, we found a large number of duplicates for the Java part which, in newly-developed code, indicates lack of reuse and abstraction.

As final observations, although EuroSim is much older than SimSat, and hence



(a) EuroSim



(b) SimSat

Figure 5.5: Unit size comparison for EuroSim and SimSat.

more exposed to code erosion, the overall duplication in EuroSim is smaller than that in SimSat. Also, for both systems, several clones found were due to the (different) implementations of the ESA Simulation Model Portability library (SMP).

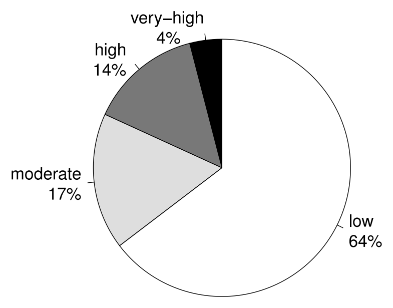
Unit size

Figure 5.5 compares the unit size of EuroSim and SimSat using risk profiles. Both EuroSim and SimSat contain a large percentage of code in the high-risk category, 17% of the overall code, leading to a ranking of two stars.

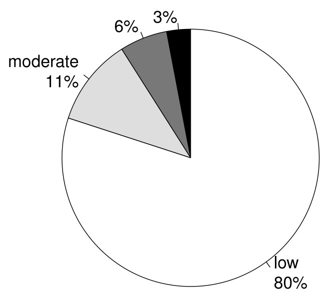
Looking at the distribution of the risk categories for both EuroSim and SimSat, we can see that they have similar amounts of code in (very) high risk categories, indicating the presence of very-large (over 100 SLOC) methods and functions.

In EuroSim, it was surprising to find a method with over 600 SLOC, and a few over 300 SLOC, most of them regarding the implementation of device drivers. At the result validation phase, this was explained to be due to manual optimization of code.

In SimSat, the largest C/C++ function contains over 6000 SLOC. However, inspection of the function revealed that it is an initialization function, having only a single argument and a single logic condition. Several methods over 300 SLOC were also found, most of them are in the Java part.



(a) EuroSim



(b) SimSat

Figure 5.6: Unit complexity comparison for EuroSim and SimSat.

Unit complexity

Figure 5.6 compares the unit complexity of EuroSim and SimSat using risk profiles. Both EuroSim and SimSat contain a similar (large) percentage of code in the high-risk category, 4% and 3%, respectively, leading to a ranking of two stars.

Considering the percentage of code of the three highest risk categories (very-high, high and moderate risk), EuroSim contains 30% of unit complexity while SimSat contains 20%. For both systems, the highest McCabe value found is around 170 decisions in a single C/C++ function (for EuroSim) and Java method (for SimSat).

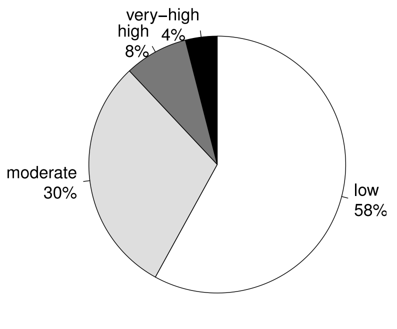
In EuroSim, methods with very-high complexity are spread in the system. Interestingly, faced with module-level measurements, the EuroSim maintainers observed that particular consortium members were responsible for modules with the worse complexity.

In SimSat, it is worth noting that the majority of the methods with very-high complexity were localized in just a dozen of files.

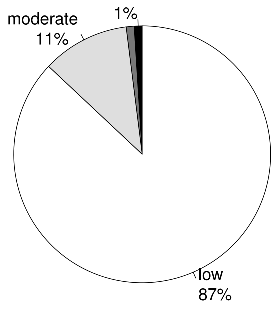
Unit interfacing

Figure 5.7 compares the Unit interfacing of EuroSim and SimSat using risk profiles. EuroSim ranks two stars while SimSat ranks three stars.

While EuroSim contains 42% of the overall code in the moderate and (very) high



(a) EuroSim



(b) SimSat

Figure 5.7: Unit interfacing comparison for EuroSim and SimSat.

risk categories, SimSat contains 13%. For both systems, the highest number of parameters is around 15. Also, for both systems, methods considered as very-high risk are found spread over the system.

In SimSat, surprisingly, no very-high risk methods were found in the Java code, only in the C/C++ code. This was surprising since for all other product properties we observed an abnormal quantity of problems in the Java code.

Testing

Both EuroSim and SimSat have similar test practices, both ranking two stars for testing due to the existence of only functional tests. Most of the testing is done manually by testing teams who follow scenarios to check if the functionality is correctly implemented.

Automatic functional/component tests are available for both systems. However, none of the systems revealed unit tests, i.e. tests to check the behavior of specific functions or methods.

Test frameworks were only found for the SimSat system, but restricted to the Java code only, and without test coverage measuring. For the C/C++ code, for both systems, no test framework has been used in the development of the tests.

For both systems test coverage information was not available. However, comparing the ratio between test and production code, for both systems, showed that there is roughly 1 line of test code per 10 lines of production code, which typically indicates low test coverage.

Regarding to EuroSim, it was observed that for different parts of the system, slightly different naming conventions were used, indicating that tests were developed in a non-structured way.

In summary, the testing practices could be improved for both systems.

5.4.2 Copyright License Detection

The copyright license analysis in EuroSim identified a total of 25 different licenses: 2 LGPL licenses, 7 licenses from consortium members, 11 licenses from hardware vendors and libraries, and 5 licenses copyrighting software to individuals.

None of the copyright licenses found poses restrictions on software distribution. However, copyright license analysis revealed OSS library code mixed with code developed by the EuroSim consortium. While for some external libraries specific folders were used, this practice was not consistent, indicating code structure problems. This issue is particularly important when updating libraries, requiring to keep track of the locations of these libraries (or to manually determine them), thus calling for extra maintenance effort.

Additionally, for software developed by a consortium of companies, the code would be expected to sit under a unique license defined by the consortium. Instead, it was discovered that each consortium company used its own license. While this is not a problem, extra effort is required in case any of the licenses needs to be updated.

Finally, it was surprising to find copyright belonging to individual developers in some files. Since this happens only for device driver extension code this also poses no serious problem.

The analysis, despite using a very simple technique, provided valuable information to the EuroSim team. Not only the team was not aware of the number and diversity of licenses in the source code, but also they were surprised to discover copyright statements to individuals.

In summary, although this analysis did not reveal copyright license issues, it helped to uncover code structure issues.

5.5 Given recommendations

Based on the analysis presented in the previous section the following recommendations were given to EuroSim and SimSat owners.

For both systems, if significant maintenance effort is planned, it was advised to monitor the code quality in order to prevent further code degradation and to improve automatic testings. Furthermore, a clear separation between production and test code should be made and testing frameworks with coverage support should be adopted.

Code monitoring should focus on Duplication (specially for SimSat, since part of it was recently built). It is also important to monitor Unit size and Unit complexity to check if the large and/or complex methods are subject of frequent maintenance. If they are, they should be refactored as this can reduce the maintenance effort and can make the testing process easier. In case of SimSat, the large methods in the recently built Java part indicate lack of encapsulation and reuse, hence requiring more attention.

With respect to overall volume, in SimSat, we recommended to completely divide the system into two parts if continuous growth is observed in order to reduced maintenance effort.

Finally, for EuroSim, it was recommended that code from external libraries should be clearly separated from production code and that the use of a unique license for all the consortium should be considered.

5.6 Lessons learned

Strict process requirements do not assure product quality

Both EuroSim and SimSat were developed using similar process requirements. Although the development is done following a rigorous process, the product quality analyses revealed moderate maintainability for EuroSim (three stars) and low maintainability for SimSat (two stars). For both systems, in-depth analysis revealed duplication, size and complexity problems causing several product properties to rank down to two stars.

To assure good software product quality it is necessary to define clear criteria to assess quality, establish quality targets and provide means to check if the targets are being met (preferably using tool-based analyses). This can be achieved using a tool-supported quality model.

The quality model defines criteria, i.e. defines the metrics/analyses that are going to be applied to the software and that check if the results are within boundaries. When using the SIG quality model, an example of a quality target is that the overall maintainability should rank a minimum of four stars. Finally, to validate that the quality target is being met it is necessary to continuously apply the quality model to monitor the evolution of the system quality.

The continuous application of a quality model during the software life-cycle offers many advantages: it allows, at any moment, to check if the quality targets are being met; it can provide early-warning when a potential quality problem is introduced - pinpointing the problem; and it allows for continuous monitoring of code quality degradation.

Quality models can reveal team problems

When assessing software product maintainability using a quality model, potential team problems can be revealed by the observation of rapid software degradation or by the unusual lack of quality in newly-built software. Team problems were revealed for both EuroSim and SimSat.

EuroSim was developed by several teams (from the consortium of companies) which, through time, contributed with different parts of the system. When analyzing software quality we observed quality differences that could be attributed to specific teams, i.e., some teams delivered code with worse quality than others. Homogeneous code quality among the consortium members can be achieved by using a shared quality model and establishing common quality targets.

SimSat was developed and maintained by external companies selected via a bidding process. In the last iteration of the product, a new technology, Eclipse RCP [66], was used to develop the GUI. When analyzing software quality for the GUI it was observed unusual duplication, size and complexity for the newly developed part, responsible for the system low maintainability rating. These quality problems suggested low expertise of the contracted company with this new technology, which was confirmed by ESA. A shared quality model between ESA and the external company responsible for software development would allow to reveal the before mentioned quality issues during development. Also, the inclusion in the outsourcing contract of the quality model to be used and the quality targets to be met, would enforce legal obligations for the external company to deliver high quality software.

In summary, the adoption of a quality model and establishing common quality targets allows to create independence between code quality and the development team. This is particular important in environments where team composition changes over time.

Tailored analyses are necessary for further investigation of product quality

Quality models evaluate a fixed set of the software characteristics hence only revealing a limited set of potential problems. Under suspicion of a particular problem, quality models should be complemented with tailored analyses to check for the existence of such problem. Additionally, even if evidence of such particular problem is not found, the analyses can reveal other problems.

For EuroSim, the customized analysis to check for copyright licenses revealed the existence of OSS code mixed with production code without clear separation, providing evidence of problems in code structure. These problems were already suspected since during test quality analysis it was observed inconsistent naming of test folders.

Code structure inconsistencies are an example of a problem that can not be detected using a quality model. Quality models are important, because they can automatically produce an overview of the system quality and provide a basis for quality comparison among systems. However, as we learned from this example, it is important to complement the quality assessment of a quality model with tailored analysis and expert opinion.

5.7 Summary

This chapter presented a quality assessment of two simulators used in the space domain: EuroSim and SimSat. To analyze both systems the SIG quality model for maintainability based on ISO/IEC 9126 standard for software product quality was used. Although both systems followed similar strict standards, quality problems were found in both systems. Not only do both systems rank less than four stars, also problems in duplication, size and complexity were found.

From the analyses of both the EuroSim and SimSat three lessons were learned.

i) Rigorous process requirements do not assure product quality:

ii) Quality models can reveal team problems:

iii) Tailored analyses are necessary for further investigation of quality:

The analyses reported in this chapter provide evidence that, although a quality model does not reveal all problems, it can be an important instrument to manage software quality and to steer the software development and maintenance activities.

Chapter 6

Conclusions

This dissertation is devoted to the evaluation of software product quality using benchmarked quality indicators. This chapter summarizes the main contributions, revisits the starting research questions, and presents avenues for future work.

6.1 Summary of Contributions

The overall contribution of this dissertation is a well-founded and pragmatic approach to use source code metrics to gain knowledge about a software system. Knowledge is obtained at both measurement (micro) and overall system (macro) levels, by turning quantitative values into a qualitative evaluation.

At micro level, the approach divides the measurement space into several risk categories, separated by specific thresholds. By associating a risk category to a measurement, this becomes more than just a number (or a quantitative value) leading to a qualitative evaluation of that measurement. For example, by identifying a particular measurement as very-high risk, we are assigning information to this measurement indicating that this value might be worth investigating.

At macro level, the approach allows one to aggregate all individual measurements into a single meaningful qualitative evaluation. This aggregation is achieved using both risk and rating thresholds. The conversion of all measurements into an N-point star rating allows the possibility of having manageable facts about a system that can be used to communicate and discuss with others. Risk and rating thresholds additionally allow traceability from the rating to individual measurements (drill down), enabling root-cause analysis: using rating thresholds allows one to decompose the rating into different risk areas, and then, subsequently using risk thresholds allows us identify the individual measurements. For example, if the overall result of a metric is 1 star, representing the worst quality evaluation, one has a clear indication that there is something worth investigating about the system. One can also decompose the rating into the measurements that justify that rating by using rating and risk thresholds.

The novelty of the approach comes from the use of a benchmark of software systems.

At micro level, when categorizing a specific measurement as very-high risk, one can relate that evaluation to all the measurements of the benchmark. The benchmark is used to characterize the distribution of a metric which is thereupon used to derive thresholds for each risk category. As example for the McCabe metric, if a measurement is higher than 14 it can be considered as very-high risk because it is among the benchmark 10% highest measurements. This approach, based on benchmark-based thresholds, adds knowledge to individual measurements allowing for better micro-level interpretation of software systems.

At macro level, stating that a system rating is 1 star relates such a rating to all systems (and their measurements) of the benchmark. The benchmark contains a representative sample of software systems from which thresholds are calibrated such that they can be ranked from 1 to N stars ranging from the worst to the best systems, respectively. When aggregating the measurements of a system, using benchmark-based thresholds, the system is rated as 1 star, for a 5 star scale and assuming a uniform distribution in which each star represents equally 20% of the systems. What is asserted is that this system is among the 20% worst systems of the benchmark. This approach, again using benchmark-based thresholds, adds qualitative information to the aggregation of all measurements.

This approach relying on source code metrics to gain knowledge about software systems is demonstrated through two case studies. In the first case study, this methodology is applied to qualitative assessment of test quality. A new metric to estimate test coverage, Static Estimation of Test Coverage (SETC), is proposed with accompanied risk and rating thresholds. When validating this metric against bug resolution efficiency a positive correlation is found. This correlation not only indicates that systems having higher static test coverage allow bugs to be solved more efficiently, but also indirectly validates the methodology. The second case study is more industrial and applies this methodology to rank and evaluate two space-domain systems. The results were validated with the systems owners which confirmed the qualitative evaluation and acknowledged the problems pinpointed.

6.2 Research Questions Revisited

This section revisits the research questions put forward in the beginning of the dissertation, providing a detailed summary on how, and in which chapters, such research question are addressed and answered.

Research Question 1

> How to establish thresholds of software product metrics and use them to show the extent of problems in the code?

Chapter 2 introduces a methodology to derive risk thresholds from a benchmark for a given metric. Based on data transformation, the methodology starts by characterizing a metric distribution that represents all systems in the benchmark. To characterize a metric, the use of the measurement weight is introduced, using Source Lines of Code (SLOC) for each system distribution, and an aggregation technique to obtain a single metric distribution representative of all benchmark systems. After characterizing a metric, thresholds are selected by choosing the percentage of code they will represent. Moreover, Chapter 2 analyzes metrics whose distribution resemble an exponential distribution and higher values of the metric indicate worse quality. Thresholds for these metrics are proposed from the tail of the distribution choosing to represent, for the majority of the metrics, 70%, 80% and 90%, defining the minimum boundaries for moderate, high and very-high risk categories, respectively. Finally, Chapter 4 analyzes a metric whose distribution resembles a normal distribution and higher values of the metric indicate good quality. For this metric, the distribution was divided into equal parts, choosing to represent 25%, 50% and 75%, to define minimum boundaries for the very-high, high and moderate risk categories, respectively.

Research Question 2

> How to summarize a software product metric while preserving the capability of root-cause analysis?

Chapter 3 demonstrates how to aggregate metrics into an N-point star rating. This is achieved by a two-step process, first by using risk thresholds proposed earlier in Chapter 2 and then by using rating thresholds. These rating thresholds are calibrated with an algorithm proposed in Chapter 3 providing the capability of summarizing a set of measurements into a qualitative evaluation (a rating). The calibration process ensures that, similar to risk thresholds, this rating is representative of systems from a benchmark. The use of thresholds allows one to keep essential information while to decomposing the rating into their measurements for root-cause analysis. Root-cause analysis capability is made possible by decomposing the rating into different risk areas, using the rating thresholds, and then by identifying the measurements that fall into each risk category, using the risk thresholds. Chapters 3 and 4 show how to calibrate a 5-point star rating such that each rating represents equally 20% of the benchmark systems. Moreover, Chapter 3 provides an example of the usage of this methodology to compare software systems, and Chapter 4 demonstrates that ratings can be correlated with external quality. Finally, Chapter 5 presents a case study where the quality of two space-domain simulators is compared and analyzed.

Research Question 3

> How can quality ratings based on internal metrics be validated against external quality characteristics?

A new metric called SETC is introduced in Chapter 4 to estimate test coverage, at method level, from static source code measurements. This metric is validated against real coverage (coverage measured with dynamic analysis tools) and a positive correlation is found. After establishing this correlation between static and dynamic coverage, the approach introduced in Chapters 2 and 3 is applied to identify thresholds and define a 5-point star rating based on benchmark data. Using a static coverage rating for several systems an experiment is conducted to check if positive correlation could be found with external quality measurements. It is found that systems with higher SETC rating have a higher developer performance when resolving issues.

Chapter 5 analyzes the ratings of various metrics of two space-domain simulators as example systems. This includes the Software Improvement Group (SIG) quality model for maintainability, which makes uses of risk and rating thresholds that can be derived from approaches presented in Chapters 2 and 3. When analyzing the ratings by breaking them down into risk areas, using rating thresholds, and then by investigation measurements, identified with risk thresholds one is able to identify source code problems (high duplication and complexity). When validating these source code problems with the systems owners, they confirmed that they were due to team problems.

Finding correlation between source code ratings and developer performance when resolving issues, and being able to identify team problems by the analysis of source code ratings are two examples that give evidence of the possibility of identifying external quality characteristics using quality ratings. Furthermore, these two examples indirectly demonstrate that risk and rating thresholds are able to capture meaningful information about software systems.

Research Question 4

> How to combine different metrics to fully characterize and compare the quality of software systems based on a benchmark?

Using the SIG quality model for maintainability, Chapter 5 shows how to fully characterize and compare the quality of various systems, resorting to two space-domain simulators as examples. Key to this quality model is the use of both risk and rating thresholds, as presented in Chapters 2 and 3, respectively. Chapter 5 shows that the blending of different metrics to fully characterize a system quality can be done by combining each metric rating into an overall rating. This overall rating is applied to the two simulators to characterize and compare their system quality. Not only is the overall rating confirmed by the system owners, but one can identify source code risks by decomposing this rating back to individual measurements, which again are validated by the system owners and can be translated to external quality issues.

6.3 Avenues for Future Work

Further improvements on the methodology to derive risk thresholds can be considered. The current methodology requires to choose specific risk percentages to derive thresholds. For most of the SIG quality model metrics, 70%, 80% and 90% have been used as heuristic. However, although the unit interfacing metric followed the same distribution as the other metrics of the SIG quality model, a different set of percentages (80%, 90% and 95%) had to be used. For the SETC metric, on the other hand, the distribution resembled a normal distribution and for this reason the percentages 25%, 50% and 75% were used. This choice of percentages to characterize a metric distribution is guided by visual inspection. However, this could also be done by curve approximation techniques. As example, in theory a fitting technique such as [77] could be used to fit 5 knots into the curve and the 3 inner knots be used as input of the algorithm. It would also be worth to investigate results of using curve fitting techniques in other sciences.

Further improvements on the methodology to calibrate risk thresholds can also be considered. These risk thresholds are fundamental as they allow to define risk profiles that are used as input to the calibration algorithm. The choice of risk thresholds affects the rating calibration results (both the rating thresholds and the distribution of systems per rating obtained with such thresholds). Further research should focus on the impact of risk thresholds on rating thresholds. Moreover, it might be possible to have a unique algorithm to derive both risk and rating thresholds, providing stronger guarantees about the expected distribution. A possibility that was not explored was to start calibrating ratings and then proceed to derive risk thresholds, instead of first deriving risk thresholds and later calibrating ratings. Rating thresholds were calibrated for a 5-star rating scale for a uniform distribution. However, the implications of using different scales and distributions should be further investigated.

The methodologies for deriving risk thresholds and calibrating rating thresholds have been applied to the metrics of the SIG quality model and to the SETC metric. Within SIG, others have used these methodologies for different metrics (Bijlsma [16], Bijlsma et al. [17] and Athanasiou [9]). The applicability to other metrics, e.g. CDK [22] or Halstead [38], or to metrics for other types of software artifacts, e.g., databases or XML schemas, is also worth investigating. Ratings from different metrics could also be used for validation purposes, establishing correlations among them and possibly enabling to identify similar and unique metrics. The methodologies for deriving risk thresholds and calibrating rating thresholds, enabling aggregation of metrics into ratings, were applied in the software domain for source-code and Issue Tracking System (ITS) derived metrics. However, these methodologies are not restricted in any way to the software domain. Since the methodologies are based on data analysis using large benchmarks, it would be worth to investigate the applicability to other domains, e.g. manufacturing.

The comparison of the SETC metric with real coverage was discussed in Chapter 4,

Analysis of this metric showed that it could be used to predict real coverage despite small errors. Evidence is given in Chapter 4 where a positive correlation with high significance was found between SETC and real coverage at system, package and class levels. However, the accuracy of results could be improved by refining the static derivation of call graphs: commonly used frameworks can be factored into the analysis, bytecode analysis can be used to recognize library/framework calls, and methods called by reflection can be estimated. Estimation of statement coverage rather than method coverage could be attempted. To not rely on detailed statement-level information, a simple SLOC count for each method could be used to estimate statement coverage. It would be also worth of investigation the use of a more detailed dependency analysis. The use of a system dependence graph and intra-procedural analysis could, as in Binkley [18], reduce the imprecision caused by dynamic dispatch, improving the estimation. Intra-procedural analysis could also allow for more fine-grained coverage analysis enabling the estimation of statement or branch coverage. Additionally, it would be interesting to evaluate the benefit of techniques to predict the execution of particular code blocks, such as estimation of the likelihood of execution by Boogerd et al. [19]. Although sophistication in the static analysis may improve accuracy, the penalties (e.g. scalability loss) should be carefully considered. Finally, the SETC could be combined with other works, namely by Kanstrén [53]. Kanstrén defines a test adequacy criterion in which code is considered as test covered if it is tested at both the unit level and the integration level. These levels are measured using the distance from the test to the method via the derived call derived dynamically when executing tests. The dynamic analysis could be replaced by a static analysis similar to the one used for the SETC metric. Risk and ratings thresholds could be derived for the levels of testing and a similar empirical validation study against ITS metric could be done.

The use of risk and rating thresholds allows metrics to be aggregated and used to evaluate a software system. This aggregation is used in the SIG quality model, prior to combining all metrics. These methodologies put forward in this dissertation can be used as a step to construct any other quality model. Further investigation could address how to combine ratings from different metrics, using (weighted) averages or more elaborated techniques. Another key issue still not addressed in quality models is that individual calibration of metrics for a particular distribution does not grant this distribution holds for all the systems when combining different metrics. This lack of transitiveness should be investigated.

In this research, although metrics that can be applied to any programming language were used, measurements were only derived from modern Object-Oriented (OO) programming languages, Java and C#. Nevertheless, the majority of systems is built using many different technologies: different programming languages, schemas, grammars, databases, configurations, etc. To have a fully comprehensive qualitative view of a software system it is therefore necessary to take all technologies into account. This research, in particular the work about thresholds, does not make any assumptions about underlying technologies. However, it is worth investigating how these measurements can be used together to reach to a single qualitative evaluation of the full software system. A possible alternative could derive ratings for each technology separately, and then compute weighted averages so that the contribution of each technology's rating is directly linked to the overall weight it represents in the system.

Once the challenge of fully characterize the quality of software system, considering all its technologies, is overcome, emphasis should be put in how to characterize the quality of a set of related systems. This question is of relevance for any organization responsible for a portfolio of systems. For instance, software house organizations responsible for developing and/or maintaining different software systems are interested in having a high-level view of all the software systems they are working on and being able to identify potential problems. Other organizations owning and using a portfolio of systems are also interested in having concrete facts about the quality of their software asset. Hence, the challenge is to find a way to create a meaningful overview of a software portfolio. A possible solution could be the use of system-level ratings that are aggregated to portfolio-level taking into account the importance and weight of each system within the organization.

The benchmark of software systems adopted used in this work and reported in this dissertation is proprietary of the SIG company. The choice to use this benchmark was due to the fact that all research was done in an industrial environment at the SIG. However, since this benchmark is not public it is more difficult to reproduce some research results, in particular the derived risk thresholds and the calibrated rating thresholds. At the time of writing, SIG welcomes researchers to stay a period of time at their offices and use their benchmark for research. Naturally, this situation might not be practical for the majority of researchers. The need of reproducing resutls creates demand for publicly available and curated benchmarks holding non-Open-Source Software (OSS) systems. A possibility is to explore whether the Qualitas Corpus [85] could be such a replacement. It is nevertheless worth exploring the methodologies introduced in this dissertation to derive thresholds using the Qualitas Corpus.

The SIG benchmark was put together by using client software systems that have been analyzed by SIG consultants. Before a system is considered to be part of the benchmark it must undergo a quality checklist to ensure that all technologies are added, that there are no wrong measurements and that the system is not an outlier. Only one version of each system is used in the benchmark and all domains are considered. In Chapter 3 an analysis of the sensitivity of the rating thresholds to the benchmark was presented. This analysis could be further elaborated into a structure procedure for deciding the adequacy of a benchmark repository. There are many interesting research questions that can be additional tackled. For instance, can one define a rigorous, widely accepted criteria for what an outlier system is? Should a benchmark be made of systems of the same domain or spreading multi-domains? Or are there significant differences among systems of different domains meaning that benchmarks should be single-domain only? Should different releases of the same software system be considered or just the latest release? How frequently should systems in a benchmark be updated? When should a system be removed from the benchmark? What is the impact of the evolution of a benchmark? All these questions have started to arise due to the use of benchmarks and, at the time of writing, have not been answered.

The importance of benchmarks was shown not only in deriving knowledge from the metrics themselves but also in validating these metrics against external characteristics. Chapter 4 demonstrated that SETC is correlated with developers performance when resolving issues. This may be regarded as a first step for the validation of metrics. Further empirical studies are still needed to find further evidence that this correlation is also an indicator for causality. Establishing correlation and causality between source code metrics and external quality characteristics are two most import steps in metric validation. Hence, research should focus more and more on these two steps since they can lead to the use and wide acceptance of metrics in industry.

Finally, it is important that industry gradually adopts these methodologies of automatic code evaluation as part of their software product quality assurance process. This adoption would help industry in producing better quality software with less effort. Equally important is the fact that industry can support research in the field by merely providing more data about their software and their processes. Sharing this data could fuel more research in the field which is important to have a better understanding of software quality.

Bibliography

[1] Hiyam Al-Kilidar, Karl Cox, and Barbara Kitchenham. The use and usefulness of the ISO/IEC 9126 quality standard. *International Symposium on Empirical Software Engineering*, 0:7 pp., 2005.

[2] Tiago L. Alves. Assessment of product maintainability for two space domain simulators. In *Proceedings of the 26th IEEE International Conference on Software Maintenance*, pages 1–7. IEEE Computer Society, 2010.

[3] Tiago L. Alves. Categories of source code in industrial systems. In *Proceedings of the 5th International Symposium on Empirical Software Engineering and Measurement*, ESEM'11, 2011.

[4] Tiago L. Alves, José Pedro Correio, and Joost Visser. Benchmark-based aggregation of metrics to ratings. In *Proceedings of the Joint Conference of the 21th International Workshop on Software Measurement and the 6th International Conference on Software Process and Product Measurement*, IWSM/MENSURA'11, pages 20–29, 2011.

[5] Tiago L. Alves and Joost Visser. Static estimation of test coverage. In *Proceedings of the 9th IEEE International Workshop on Source Code Analysis and Manipulation*, SCAM'09, pages 55–64. IEEE Computer Society, 2009.

[6] Tiago L. Alves, Christiaan Ypma, and Joost Visser. Deriving metric thresholds from benchmark data. In *Proceedings of the 26th IEEE International Conference on Software Maintenance*, ICSM'10, pages 1–10. IEEE Computer Society, 2010.

[7] Theodore Wilbur Anderson and Donald A. Darling. Asymptotic theory of certain "goodness of fit" criteria based on stochastic processes. *The Annals of Mathematical Statistics*, 23(2):193–212, June 1952.

[8] Jorge Aranda. How do practitioners perceive software engineering research?, May 2011. http://catenary.wordpress.com/2011/05/19/how-do-practitioners-perceive-software-engineering-research/.

[9] Dimitrios Athanasiou. Constructing a test code quality model and empirically assessing its relation to issue handling performance. Master's thesis, University of Delft, The Netherlands, 2011.

[10] Robert Baggen, José Pedro Correia, Katrin Schill, and Joost Visser. Standardized code quality benchmarking for improving software maintainability. *Software Quality Journal*, pages 1–21, 2011.

[11] Robert Baggen, Katrin Schill, and Joost Visser. Standardized code quality benchmarking for improving software maintainability. In *Proceedings of the 4th International Workshop on Software Quality and Maintainability*, SQM'10, 2010.

[12] John Bambenek and Angieszka Klus. *grep - Pocket Reference: the Basics for an Essential Unix Content-Location Utility*. O'Reilly, 2009.

[13] Kent Beck. Simple smalltalk testing: with patterns. *Smalltalk Report*, 4(2):16–18, 1994.

[14] Saida Benlarbi, Khaled El Emam, Nishith Goel, and Shesh Rai. Thresholds for object-oriented measures. In *Proceedings of the 11th International Symposium on Software Reliability Engineering*, ISSRE'00, pages 24–38. IEEE, 2000.

[15] Antonia Bertolino. Software testing research: Achievements, challenges, dreams. In *Proceedings of the Workshop on The Future of Software Engineering*, FOSE'07, pages 85–103. IEEE Computer Society, 2007.

[16] Dennis Bijlsma. Indicators of issue handling efficiency and their relation to software maintainability. Master's thesis, University of Amsterdam, The Netherlands, 2010.

[17] Dennis Bijlsma, Miguel Ferreira, Bart Luijten, and Joost Visser. Faster issue resolution with higher technical quality of software. *Software Quality Journal*, pages 1–21, 2011.

[18] David Binkley. Semantics guided regression test cost reduction. *IEEE Transactions on Software Engineering*, 23(8):498–516, 1997.

[19] Cathal Boogerd and Leon Moonen. Prioritizing software inspection results using static profiling. In *Proceedings of the 6th IEEE International Workshop on Source Code Analysis and Manipulation*, SCAM'06, pages 149–160. IEEE Computer Society, 2006.

[20] P. Botella, X. Burgués, J. P. Carvallo, X. Franch, G. Grau, J. Marco, and C. Quer. ISO/IEC 9126 in practice: what do we need to know? In *Proceedings of the 1st Software Measurement European Forum*, SMEF'04, January 2004.

[21] Jitender Kumar Chhabra and Varun Gupta. A survey of dynamic software metrics. *Journal of Computer Science and Technology*, 25:1016–1029, September 2010.

[22] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[23] Don Coleman, Bruce Lowther, and Paul Oman. The application of software maintainability models in industrial software systems. *Journal of Systems and Software*, 29(1):3–16, 1995.

[24] Giulio Concas, Michele Marchesi, Sandro Pinna, and Nicola Serra. Power-laws in a large object-oriented software system. *IEEE Transactions on Software Engineering*, 33(10):687–708, 2007.

[25] José Pedro Correia, Yiannis Kanellopoulos, and Joost Visser. A survey-based study of the mapping of system properties to ISO/IEC 9126 maintainability characteristics. In *Proceedings of the 25th IEEE International Conference on Software Maintenance*, ICSM'09, pages 61–70, 2009.

[26] José Pedro Correia and Joost Visser. Benchmarking technical quality of software products. In *Proceedings of the 15th Working Conference on Reverse Engineering*, WCRE'08, pages 297–300. IEEE Computer Society, 2008.

[27] José Pedro Correia and Joost Visser. Certification of technical quality of software products. In *Proceedings of the 2nd International Workshop on Foundations and Techniques for Open Source Software Certification*, OpenCert'08, pages 35–51, 2008.

[28] Oege de Moor, Damien Sereni, Mathieu Verbaere, Elnar Hajiyev, Pavel Avgustinov, Torbjörn Ekman, Neil Ongkingco, and Julian Tibble. .QL: Object-oriented queries made easy. In Ralf Lämmel, Joost Visser, and Jo ao Saraiva, editors, *Generative and Transformational Techniques in Software Engineering II*, pages 78–133. Springer-Verlag, 2008.

[29] Khaled El Emam, Saïda Benlarbi, Nishith Goel, Walcelio Melo, Hakim Lounis, and Shesh N. Rai. The optimal class size for object-oriented software. *IEEE Transactions on Software Engineering*, 28(5):494–509, 2002.

[30] Karin Erni and Claus Lewerentz. Applying design-metrics to object-oriented frameworks. In *Proceedings of the 3rd International Software Metrics Symposium*, METRICS'96, pages 64–74. IEEE Computer Society, 1996.

[31] European Cooperation for Space Standardization (ECSS), Requirements & Standards Divison, Noordwijk, The Netherlands. *Space engineering: software*, March 2009. ECSS-Q-ST-40C.

[32] European Cooperation for Space Standardization (ECSS), Requirements & Standards Divison, Noordwijk, The Netherlands. *Space product assurance: software product assurance*, March 2009. ECSS-Q-ST-80C.

[33] Eurosim applications: Galileo monitoring and uplink control facility AIVP. http://www.eurosim.nl/applications/mucf.shtml. Version of October 2011.

[34] Norman E. Fenton and Martin Neil. Software metrics: roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE'00, pages 357–370, 2000.

[35] Norman E. Fenton and Shari Lawrence Pfleeger. *Software metrics: a rigorous and practical approach.* PWS Publishing Co., 1997. 2nd edition, revised printing.

[36] Vern A. French. Establishing software metric thresholds. In *International Workshop on Software Measurement (IWSM'99)*, IWSM'99, 1999.

[37] Mark Grechanik, Collin McMillan, Luca DeFerrari, Marco Comi, Stefano Crespi, Denys Poshyvanyk, Chen Fu, Qing Xie, and Carlo Ghezzi. An empirical investigation into a large-scale Java open source code repository. In *Proceedings of the 4th International Symposium on Empirical Software Engineering and Measurement*, ESEM'10, pages 11:1–11:10. ACM, 2010.

[38] Maurice H. Halstead. *Elements of Software Science*, volume 7 of *Operating and Programming Systems Series*. Elsevier, 1977.

[39] Mary Jean Harrold. Testing: a roadmap. In *Proceedings of the Workshop on The Future of Software Engineering*, ICSE'00, pages 61–72. ACM, 2000.

[40] Ilja Heitlager, Tobias Kuipers, and Joost Visser. Observing unit test maturity in the wild. http://www3.di.uminho.pt/~joost/publications/UnitTestingInTheWild.pdf. 13th Dutch Testing Day 2007. Version of October 2011.

[41] Ilja Heitlager, Tobias Kuipers, and Joost Visser. A practical model for measuring maintainability. In *Proceedings of the 6th International Conference on the Quality of Information and Communications Technology*, QUATIC'07, pages 30–39, 2007.

[42] S. Henry and D. Kafura. Software structure metrics based on information flow. *IEEE Transactions on Software Engineering*, 7:510–518, September 1981.

[43] Project Management Institute, editor. *A Guide To The Project Management Body Of Knowledge (PMBOK Guides).* Project Management Institute, 2008.

[44] International Standards Organisation (ISO). International standard ISO/IEC 9126. Software engineering – Product quality – Part 1: Quality model, 2001.

[45] International Standards Organisation (ISO). International standard ISO/IEC TR 9126-2. Software engineering – Product quality – Part 2: External metrics, 2003.

[46] International Standards Organisation (ISO). International standard ISO/IEC 15505-1. Information technology process assessment Part 1: Concepts and vocabulary, 2008.

[47] International Standards Organisation (ISO). International standard ISO/IEC 9001. Quality management systems – Requirements, 2008.

[48] International Standards Organisation (ISO). International Standard 80000–2:2009: Quantities and units – Part 2: Mathematical signs and symbols to be used in the natural sciences and technology, December 2009.

[49] International Standards Organisation (ISO). International standard ISO/IEC 25010. Systems and software engineering: Systems and software quality requirements and evaluation (SQuaRE) – System and software quality models, 2011.

[50] Paul Jansen. Turning static code violations into management data. In *Working Session on Industrial Realities of Program Comprehension*, ICPC'08, 2008. http://www.sig.nl/irpc2008/ Version of June 2008.

[51] M. Jones, U.K. Mortensen, and J. Fairclough. The ESA software engineering standards: Past, present and future. *International Symposium on Software Engineering Standards*, pages 119–126, 1997.

[52] Yiannis Kanellopoulos, Panagiotis Antonellis, Dimitris Antoniou, Christos Makris, Evangelos Theodoridis, Christos Tjortjis, and Nikos Tsirakis. Code quality evaluation methodology using the ISO/IEC 9126 standard. *International Journal of Software Engineering & Applications*, 1.3:17–36, 2010.

[53] Teemu Kanstrén. Towards a deeper understanding of test coverage. *Journal of Software Maintenance and Evolution: Research and Practice*, 20(1):59–76, 2008.

[54] Ken Koster and David Kao. State coverage: a structural test adequacy criterion for behavior checking. In *Proceedings of the 6th Joint Meeting on European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering: companion papers*, ESEC-FSE companion'07, pages 541–544. ACM, 2007.

[55] Philippe Kruchten. *The Rational Unified Process: An Introduction.* Addison-Wesley Longman Publishing Co., Inc., 3 edition, 2003.

[56] Tobias Kuipers, Joost Visser, and Gerjon De Vries. Monitoring the quality of outsourced software. In Jos van Hillegersberg et al., editors, *Proceedings of the*

1st International Workshop on Tools for Managing Globally Distributed Software Development, TOMAG'09. CTIT, The Netherlands, 2007.

[57] Arun Lakhotia. Graph theoretic foundations of program slicing and integration. Technical Report CACS TR-91-5-5, University of Southwestern Louisiana, 1991.

[58] Michele Lanza and Radu Marinescu. *Object-Oriented Metrics in Practice. Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-oriented Systems.* Springer Verlag, 2010.

[59] Software Productivity Research LCC. Programming languages table, Februrary 2006. version 2006b.

[60] Rensis Likert. A technique for the measurement of attitudes. *Archives of Psychology*, 22(140):1–55, 1932.

[61] Chris Lokan. The Benchmark Release 10 - Project Planning edition. Technical report, International Software Benchmarking Standards Group Ltd., February 2008.

[62] Panagiotis Louridas, Diomidis Spinellis, and Vasileios Vlachos. Power laws in software. *ACM Transactions on Software Engineering and Methodology (TOSEM'08)*, 18(1):2:1–2:26, September 2008.

[63] Bart Luijten and Joost Visser. Faster defect resolution with higher technical quality of software. In *Proceedings of the 4th International Workshop on Software Quality and Maintainability*, SQM'10, 2010.

[64] Michael R. Lyu. Software reliability engineering: A roadmap. In *Proceedings of the Workshop on The Future of Software Engineering*, ICSE'07, pages 153–170. IEEE Computer Society, 2007.

[65] Robert L. Mason, Richard F. Gunst, and James L. Hess. *Statistical Design and Analysis of Experiments, with Applications to Engineering and Science*. Wiley-Interscience, 2nd edition, 2003.

[66] Jeff McAffer and Jean-Michel Lemieux. *Eclipse Rich Client Platform: Designing, Coding, and Packaging Java(TM) Applications*. Addison-Wesley Professional, 2005.

[67] Thomas J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, December 1976.

[68] Brian A. Nejmeh. NPATH: a measure of execution path complexity and its applications. *Communications of the ACM*, 31(2):188–200, 1988.

[69] Offices of Government Commerce. *Managing Successful Projects with PRINCE2*. Stationery Office Books, 2009.

[70] Karl Pearson. Notes on the history of correlation. *Biometrika*, 13(1):25–45, October 1920.

[71] S. R. Ragab and H. H. Ammar. Object oriented design metrics and tools a survey. In *Proceedings of the 7th International Conference on Informatics and Systems*, INFOS'10, pages 1–7, March 2010.

[72] Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara G. Ryder, and Ophelia Chesley. Chianti: a tool for change impact analysis of Java programs. In *Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA'04, pages 432–448. ACM, 2004.

[73] Gregg Rothermel and Mary Jean Harrold. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering*, 22(8):529–551, August 1996.

[74] Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary Jean Harrold. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, 27(10):929–948, October 2001.

[75] Barbara G. Ryder and Frank Tip. Change impact analysis for object-oriented programs. In *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE'01, pages 46–53. ACM, 2001.

[76] Ken Schwaber. *Agile Project Management With Scrum.* Microsoft Press, Redmond, WA, USA, 2004.

[77] Hubert Schwetlick and Torsten Schütze. Least squares approximation by splines with free knots. *BIT*, 35:361–384, 1995.

[78] Alexander Serebrenik and Mark van den Brand. Theil index for aggregation of software metrics values. In *Proceedings of the 26th IEEE International Conference on Software Maintenance*, ICSM'10, pages 1–9. IEEE Computer Society, 2010.

[79] Raed Shatnawi, Wei Li, James Swain, and Tim Newman. Finding software metrics threshold values using ROC curves. *Journal of Software Maintenance and Evolution: Research and Practice*, 22:1–16, January 2009.

[80] Galileo assembly, integration and verification platform, AIVP. http://www.vegaspace.com/about_us/case_studies/galileo_aivp.aspx. Version of October 2011.

[81] Charles Spearman. The proof and measurement of association between two things. *American Journal of Psychology*, 100(3/4):441–471, 1987.

[82] Diomidis Spinellis. A tale of four kernels. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE'08, pages 381–390. ACM, 2008.

[83] CMMI Product Team. CMMI for development, version 1.3. Technical Report CMU/SEI-2010-TR-033, Software Engineering Institute and Carnegie Mellon, November 2010.

[84] R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2009. ISBN 3-900051-07-0.

[85] Ewan Tempero, Craig Anslow, Jens Dietrich, Ted Han, Jing Li, Markus Lumpe, Hayden Melton, and James Noble. The Qualitas Corpus: A curated collection of Java code for empirical studies. In *Proceedings of the 17th Asia-Pacific Software Engineering Conference*, APSEC'10, pages 336–345. IEEE Computer Society, 2010.

[86] Arie van Deursen and Tobias Kuipers. Source-based software risk assessment. In *Proceedings of the 21th International Conference on Software Maintenance*, ICSM'03, pages 385–388. IEEE Computer Society, 2003.

[87] Rajesh Vasa, Markus Lumpe, Philip Branch, and Oscar Nierstrasz. Comparative analysis of evolving software systems using the Gini coefficient. In *Proceedings of the 25th IEEE International Conference on Software Maintenance*, ICSM'09, pages 179–188. IEEE Computer Society, 2009.

[88] Joost Visser. Software quality and risk analysis. Invited lecture Master Software Technology / Software Engineering, University of Utrecht, Netherlands, October 2007. http://www.cs.uu.nl/docs/vakken/swe/11-swe-softwareimprovementsig.pdf Version of October 2011.

[89] Richard Wheeldon and Steve Counsell. Power law distributions in class relationships. In *Proceedings of the 3rd IEEE International Workshop on Source Code Analysis and Manipulation*, SCAM'03, page 45. IEEE Computer Society, 2003.

[90] M. Xenos, D. Stavrinoudis, K. Zikouli, and D. Christodoulakis. Object-oriented metrics – a survey. In *Proceedings of the European Software Measurement Conference*, FESMA'00, pages 1–10, 2000.

[91] Qian Yang, J. Jenny Li, and David Weiss. A survey of coverage based testing tools. In *Proceedings of the 1st International Workshop on Automation of Software Test*, AST'06, pages 99–103. ACM, 2006.

[92] Kyung-A Yoon, Oh-Sung Kwon, and Doo-Hwan Bae. An approach to outlier detection of software measurement data using the K-means clustering method. In *Proceedings of the 1st International Symposium on Empirical Software Engineering and Measurement*, ESEM'07, pages 443–445. IEEE Computer Society, 2007.