Mining Software Engineering Data for Useful Knowledge

Boris Baldassari

École doctorale Sciences Pour l'Ingénieur

T H È S E

présentée en vue d'obtenir le grade de

Docteur, spécialité Informatique

par

Boris Baldassari

Mining Software Engineering Data for Useful Knowledge

preparée dans l'équipe-projet SequeL commune

Soutenue publiquement le 1er Juillet 2014 devant le jury composé de :

Philippe Preux, Professeur des universités	-	Université de Lille 3	-	Directeur
Benoit Baudry, Chargé de recherche INRIA	-	INRIA Rennes	-	Rapporteur
Laurence Duchien, Professeur des universités	-	Université de Lille 1	-	Examinateur
Flavien Huynh, Ingénieur Docteur	-	SQuORING Technologies	-	Examinateur
Pascale Kuntz, Professeur des universités	-	Polytech' Nantes	-	Rapporteur
Martin Monperrus, Maître de conférences	-	Université de Lille 1	-	Examinateur

Preface

Maisqual is a recursive acronym standing for "Maisqual Automagically Improves Software QUALity". It may sound naive or pedantic at first sight, but it clearly stated at one time the expectations of Maisqual. The primary idea for the Maisqual project was to investigate the application of data mining methods to practical concerns of software quality engineering and assessment. This includes the development of innovative techniques for the SQuORE product, and also the verification of some common myths and beliefs about software engineering. It also proposes new tools and methods to extract useful information from software data.

The Maisqual project is deeply rooted in *empirical* software engineering and heavily relies on the experience gathered during years of practice and consulting with the industry. In order to apply the data mining component, we had to widen our knowledge of statistics by reading student manuels and reference books, experimenting with formulae and verifying the results.

As a bridge between two distinct disciplines, we had to pay great attention to the *communication* of ideas and results between practitioners of both fields. Most of Maisqual's users have a software engineering background and one of the requirements of the project was to help them to grasp the benefits of the investigated data mining tools. We intensively used analogies, pictures and common sense in order to explain concepts and to help people map ideas to notions with which they are familiar.

We decided to formulate our different lines of enquiry into projects and established a structure to complete them. Requirements were refined to be more effective, more precise and more practical. From the initial intent we designed a framework and a method to address the various objectives of our search. This enabled us to setup a secure foundation for the establishment of specific tasks, which gave birth to the SQuORE Labs. SQuORE Labs are *clearly-defined projects* addressing major subjects with *pragmatic results and implementations*.

This report retraces the evolution of the Maisqual project from the discovery of new statistical methods to their implementation in practical applications. We tried to make it useful, practical and not too boring.

Acknowledgements

This project has been a huge and long investment, which would never have been possible without many people. First of all, I would like to thank SQuORING Technologies for funding this long-running project, and Philippe Preux from INRIA Lille, who was a faithful director. Philippe kept me up on the rails, orienting research in safe directions and always watching the compass and range of sight.

I would like to thank my dissertation committee members, Benoit Baudry, Laurence Duchien, Flavien Huynh, Pascale Kuntz, and Martin Monperrus, for taking interest in my research and accepting to examine my work. Special thanks go to Pascale Kuntz and Benoit Baudry, for their thorough reviews and insightful comments.

Also, I'll be eternally grateful to all the people that supported me, and took care of me, when I was working on Maisqual and during the many adventures that happened during these three years: family, friends, neighbours, those who left (Tanguy, Mélodie) and those who are still there. Yakacémé, the dutch barge I'm living on and working in, was a safe and dependable home and shelter. Special thanks go to Cassandra Tchen, who was a heartful coach, bringing sound advice and reviews, and helped a lot correcting and enhancing the English prose.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter presents the organisation adopted for running the Maisqual project, introducing the people and organisations involved in the project, and stating the objectives and outputs of the work. It also gives valuable information to better understand the structure and contents of this report, to enable one to capitalise on the knowledge and conclusions delivered in this project.

The first section gives insights in the genesis of Maisqual, how it came to life and who was involved in its setup. The project timeline in section 1.2 depicts the high-level phases the project ran across, and expected outputs and requirements are documented in section 1.3. Typographic and redaction conventions used in the document are explained in section 1.4. A short summary outlines the forthcoming structure of the document in section 1.5.

1.1 Context of the project

1.1.1 Early history of Maisqual

The Maisqual project had its inception back in 2007 when I met Christophe Peron[1] and we started sharing our experiences and beliefs about software engineering. Through various talks, seminars and industrial projects we had in common, the idea of a research work on empirical software engineering steadily evolved.

The project would eventually become a reality in 2011 with the foundation of SQuORING Technologies, a start-up initiated by major actors of the world of software quality and publisher of the SQuORE product. Patrick Artola, co-founder and CEO of SQuORING Technologies, had contacted the INRIA Lille SequeL team for a research project on data mining and automatic learning. The new company needed work power for its beginning, and since the project was considered as being somewhat important and time-consuming, a compromise was found to mix product-oriented development, industrial consultancy and research studies. A deal was signed in April 2011, with 2 days per week for the thesis

[1] Christophe Peron is an experienced software consultant. He was one of the founders of Verilog, has worked for Continuous, Kalimetrix, Telelogic, and IBM, and is now Product Manager for SQuORING Technologies.

work and 3 days per week for SQuORING Technologies. This was accepted by all parts and would run for the next three years.

1.1.2 About INRIA Lille and SequeL

SequeL[2] is an acronym for *Sequential Learning*; it is a joint research project of the LIFL (Laboratoire d'Informatique Fondamentale de Lille, Université de Lille 3), the CNRS (Centre National de Recherche Scientifique) and the INRIA (Institut National de Recherche en Informatique Appliquée) located in the Lille-Nord Europe research center.

The aim of SequeL is to study the resolution of sequential decision problems. For that purpose, they study sequential learning algorithms with focus on reinforcement and bandit learning and put an emphasis on the use of concepts and tools drawn from statistical learning. Work ranges from the theory of learnability, to the design of efficient algorithms, to applications; it is important to note that the team has interests and activities in both fundamental aspects and real applications. SequeL has an important specificity that its members originate from 3 different fields: computer science, applied mathematics, and signal processing. Usually researchers of these fields work independently; in SequeL they work together, and cross-fertilize.

Philippe Preux is the leader of the SequeL team working in the fields of sequential learning and data mining[3]. He teaches at Université de Lille 3 and takes part in various conferences, committees and scientific events.

1.1.3 About SQuORING Technologies

SQuORING Technologies[4] was founded in late 2010 by a group of Software Engineering experts, united by the same vision to provide a *de facto* standard platform for software development projects in the embedded market. The company is located in Toulouse, France, with offices in Paris and Grenoble. As for its international presence, SQuORING Technologies GmbH was founded in 2011 in Bavaria, Germany to target the German market of embedded automotive and space software. Several distributors propose the product on the Asian, European, and US markets.

SQuORING Technologies offers product licensing, consultancy, services and training to help organizations in deploying SQUORE into their operational environment and filling the gap from a hard-to-implement metric-based process to team awareness and successful decision making. Based on the strong operational background and deep knowledge in software-related standards of its staff members, SQuORING Technologies' mission is to improve capability, maturity and performance for its customers in their acquisition, development and maintainance of software products.

[2]For more information see sequel.lille.inria.fr/.

[3]For more information see www.grappa.univ-lille3.fr/~ppreux/rech/publis.php.

[4]More information about SQuORING Technologies can be found on the company's web site: www.squoring.com

Its customers range from medium to large companies in different domains of activities: aeronautics (Airbus, Rockwell-Collins, Thalès), defense (DCNS), healthcare (McKesson), transport and automotive (Valéo, Continental, Delphi, Alstom, IEE, Magneti-Marelli), energy (Schneider Electric), computing and contents (Bull, Sysfera, Technicolor, Infotel).

1.2 Project timeline

The chronicle of Maisqual can be roughly divided into three phases pictured in figure 1.1. They correspond more or less to the three parts listed in this report:

- ⇝ The State of the Art part investigated existing research to establish where we were beginning from.
- ⇝ Building upon this landscape, we decided on a methodological framework on which to lay a secure base for the final objectives of the Maisqual project.
- ⇝ Finally, on top of this foundation the SQuORE Labs projects are the very research topics addressed by Maisqual.

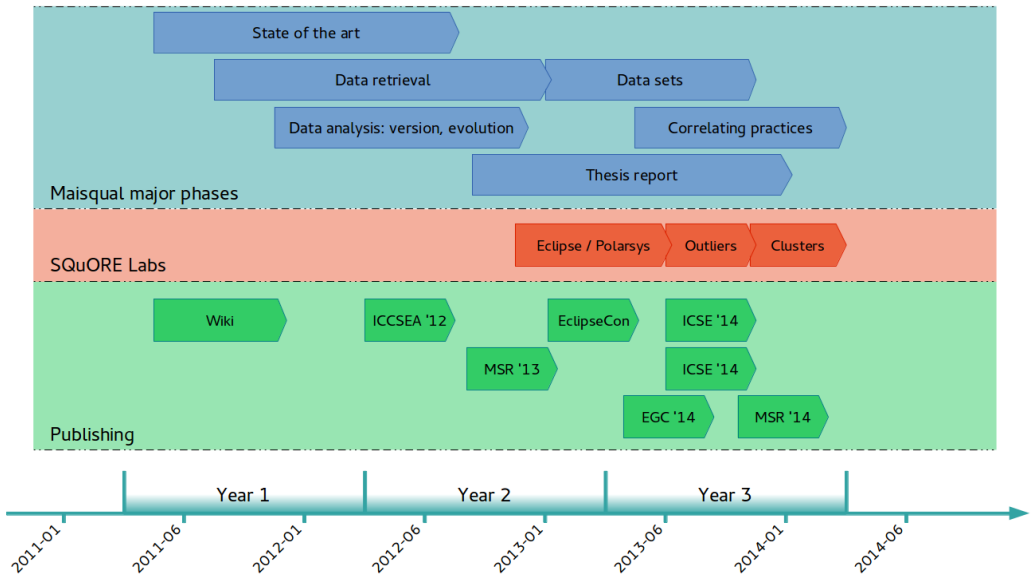


Figure 1.1: Timeline for the Maisqual project.

Each of these phases had a distinct purpose; although they considerably overlap in time, they really have been a successive progression of steps, each stage building upon the results of the preceding one.

1.3 Expected outputs

1.3.1 SQuORE Labs

SQuORE Labs are projects that target practical concerns like new features for the SQuORE product. They have been setup as pragmatic implementations of the primary goals of the project, and secure the transfer of academic and research knowledge into SQuORING Technologies assets. They are time-boxed and have well-identified requirements and deliverables.

SQuORE Labs address a specific software engineering concern and are usually defined as an extension of a promising intuition unveiled by state of the art studies or experts' observations. The following SQuORE Labs have been defined as the thesis' goals:

- ⇝ **The Polarsys quality assessment working group**, initiated by the Eclipse foundation to address maturity concerns for industries of the critical embedded systems market. We participated in the definition and prototyping of the solution with the group to establish process-related measures and define a sound, literature-backed quality model.
- ⇝ **Outliers detection** has been introduced in SQuORE to highlight specific types of artefacts like untestable files or obfuscated code. The advantage of using outliers detection techniques is that thresholds are dynamic and do not depend on fixed values. This makes the highlights work for many different types of software by adapting the acceptable values according to the majority of data instead of relying on generic conventions.
- ⇝ **Clustering** brings automatic scales to SQuORE, by proposing ranges adapted to the characteristics of projects. It enhances the capitalisation base feature of SQuORE and facilitates the manual process of model calibration.
- ⇝ **Regression analysis** of data allows to establish relationships among the metrics and to find correlations between practices, as defined by violations to coding or naming standards, and attributes of software. The intent is to build a body of knowledge to validate or invalidate common beliefs about software engineering.

SQuORE Labs have been put in a separate part of this report to highlight the outputs of the project.

1.3.2 Communication

The Maisqual website, which was started at the beginning of the project, contains a comprehensive glossary of software engineering terms (422 definitions and references), lists standards and quality models (94 entries) related to software along with links and information, references papers and books, and publicises the outputs of the Maisqual project: data sets, publications, analyses. Its home page is depicted on figure 1.2.

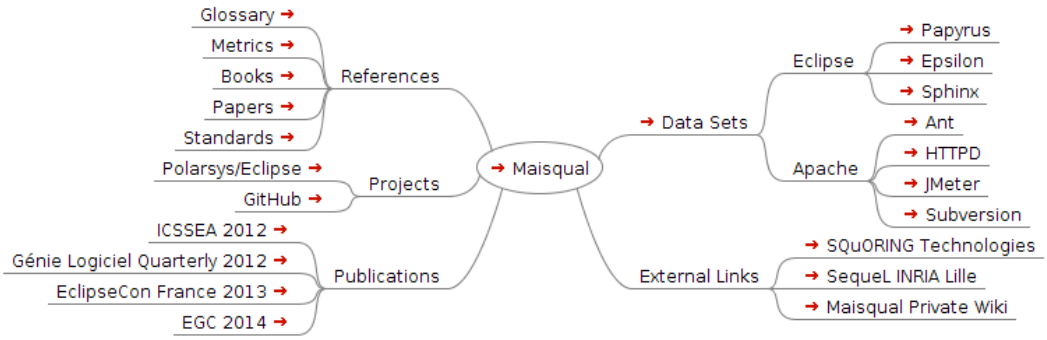


Figure 1.2: Homepage of the Maisqual web site.

Web usage statistics show an increasing traffic on the `maisqual.squoring.com` web site. At the time of writing there is an average of 20 daily visits, as show in figure 1.3, and it is even referenced on other web sites.

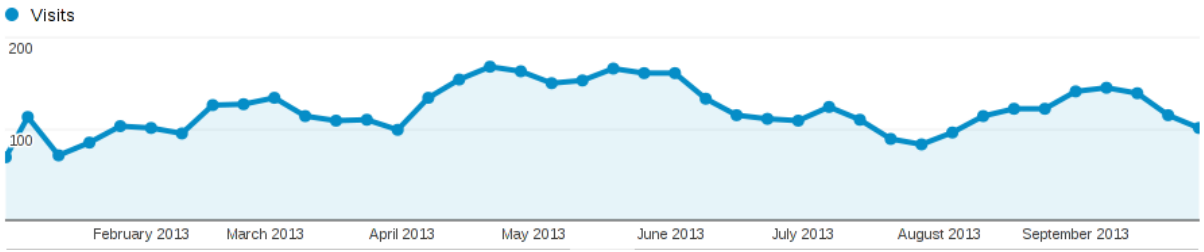


Figure 1.3: Evolution of weekly visits on the Maisqual website in 2013.

The server used for the analysis has also been publicised to show reports of some open-source projects, most notably for Topcased, Eclipse, and GitHub.

1.3.3 Publications

Another requirement of SQuORING Technologies is to have papers published in the company's area of knowledge so as to contribute to the global understanding and handling of quality-related questions.

This has been done through speeches in conferences both in academic and industry worlds, involvement in the Polarsys Eclipse Industry Working Group, and participation in various meetings.

Papers published during the Maisqual project are listed hereafter:

- *SQuORE: a new approach to software project quality measurement* [11] is a presentation of SQuORE principles, and how they can be used to assess software products and projects. It was presented at the 24th International Conference on Software & Systems Engineering and their Applications held in Paris in 2012. See full article in appendix page 199.

⇝ A french version of the above article was published in the Génie Logiciel quarterly: *SQuORE : une nouvelle approche de mesure de la qualité des projets logiciels* [10]. See full article in appendix page 207.

⇝ *Software Quality: the Eclipse Way and Beyond* [12] presents the work accomplished with the Polarsys task force to build a quality program for Eclipse projects. It was presented at the EclipseCon France 2013.

⇝ *De l'ombre à la lumière : plus de visibilité sur l'Eclipse* [13] is another insight into the work conducted with Polarsys, stressing the knowledge extraction process setup. It was presented at the 14th Journées Francophones Extraction et Gestion des Connaissances, held in Renne in January 2014. The complete article submitted is available in appendix page 217. The poster and short article published are respectively reproduced in appendix, pages 233 and 233.

⇝ *Outliers in Software Engineering* [15] describes how we applied outliers detection techniques to software engineering data and has been submitted to the 36th International Conference on Software Engineering, held in Hyderabad, India, in 2014. See full article in appendix, page 235.

⇝ *A Practitioner approach to Software Engineering Data Mining* [14] details the lessons we learned and summarises the experiences acquired when mining software engineering data. It has been submitted to the 36th International Conference on Software Engineering, held in Hyderabad, India, in 2014. See full article in appendix, page 243.

⇝ *Understanding Software Evolution: the Apache Ant data set* [16] describes the data set generated for the Apache Ant open source project, with a thorough description of the retrieval process setup and metrics gathered. It has been submitted to the 2014 Mining Software Repositories data track, held in Hyderabad, India, in 2014. See full article in appendix, page 251.

1.4 About this document

This document has been written using Emacs and the LaTeX document preparation system. All the computations conducted for our research have been executed with R, an open source language and environment for statistical computing and graphics. Similarly, pictures have been generated from R scripts to ensure reproducibility.

About metrics

This document uses a lot of well-known and not-so-well-known software metrics, which may be referenced either by their full name (e.g. *Cyclomatic complexity*) or mnemonic (e.g. (VG). Most common metrics are listed in the index on page 286, and they are extensively described in chapter 6, section 6.1 on page 106.

Visual information & Pictures

Some trade-offs had to be found between the quantity of information on a picture and the size of a page: a scatterplot showing 6 different metrics will be too small and thus unreadable on a screen, but showing only 3 metrics at a time takes two pages of plots. We tried to find optimal compromises in our graphs; most of the pictures presented in this report have been generated with a high resolution and rescaled to fit onto pages. One can have a rough idea of what is displayed, and export the picture to an external file to see it fullscreen in greater detail. These high-resolution pictures can also be zoomed in directly in the document and give very nice graphics when viewed with a 400% scale.

Typographic conventions

Commands, file names or programming language words are typeset with a typewriter font. Metrics are written with small caps. Paths and file names use a typewriter font. The most common typographic conventions are listed thereafter.

Convention	Style	Example
Commands	Typewriter	We used the `hclust` command from the `stats` [147] package.
Code	Typewriter	There shall be no fall through the next `case` in a `switch` statement.
Metrics	Small Caps	The SLOC is a common line-counting metric used to estimate the size and the effort of a software project.
Paths and file names	Typewriter	Results were temporarily stored in `/path/to/file.csv`.

Places where we identified **things to be further investigated**, or room for improvement are typeset like that:

The time series analysis is an emerging technique [91]. The data sets should be used in a time series context to investigate impacts of practices on the project quality.

Citations are outlined this way:

> “ A thinker sees his own actions as experiments and questions–as attempts to find out something. Success and failure are for him answers above all. ”
>
> Friedrich Nietzsche

1.5 Summary

This report intends to restitute the evolution of this project, both in the development in time and in ideas. The state of the art in part I gives a thorough vision of the knowledge we built upon, and provides a semantic context for our research. Part II follows the progression of ideas and steps of actions we took to setup a sound methodological framework and to develop it into a reliable and meaningful process for our purpose. Part III describes the practical application of our research to real-life concerns and needs, most notably for the implementation of the developed techniques into the SQuORE product. Finally, we draw conclusions and propose future directions in part IV.

Part I

State of the art

Learning is the beginning of wealth.
Learning is the beginning of health.
Learning is the beginning of spirituality.
Searching and learning is where the
miracle process all begins.

Jim Rohn

Chapter 2

Software Engineering

Simply put, software engineering can be defined as *the art of building great software*. At its very beginning (60's) software was mere craftsmanship [159]: good and bad practices were dictated by talented gurus, and solutions to problems were *ad hoc*. Later on, practitioners began to record and share their experiences and lessons learned, laying down the first stones of the early industry of software. Knowledge and mechanisms have been slowly institutionalised, with newcomers relying on their elders' experiences and the beginning of an emerging science with its body of knowledge and practices had begun; an example being the arrival of design patterns in late 1980s [22, 70].

With new eyes, new tools, new techniques and methods, software research and industry steadily gained maturity, with established procedures and paradigms. Since then software production has gone mainstream, giving birth to a myriad of philosophies and trends (Lean development, XP, Scrum), and permeating every aspect of daily life. The use of software programs across all disciplines underlines the need to establish the field of software engineering as a mature, professional and scientific engineering discipline.

Software engineering is quite old as a *research topic*: many fundamentals (e.g. McCabe's cyclomatic number [130], Boehm's quality model [27], or Halstead's Software Science [82]) were produced during the seventies. Resting on this sound foundation, standardisation organisms have built glossaries and quality models, and software measurement methods have been designed and improved. This chapter reviews the major software engineering concepts that we used in this project: section 2.1 portrays fundamentals of software development processes and practices; section 2.2 establishes the concerns and rules of measurement as applied to software, section 2.3 gives a few definitions and concepts about software quality, and section 2.4 lists some major quality models that one needs to know when working with software quality.

2.1 The art of building software

2.1.1 Development processes

For small, simple programs, *ad hoc* practices are sufficient. As soon as the software becomes too complex however, with many features and requirements, and hundreds of people working on the same massive piece of code, teams need to formalise what should be done (good practices) or not (bad practices) and generally write down the roadmap of the building process. In this context, the development process is defined as *how* the software product is developed, from its definition to its delivery and maintenance. It usually encompasses a set of methods, tools, and practices, which may either be explicitly formalised (e.g. in the context of industrial projects), or implicitly understood (e.g. as it happens sometimes in community-driven projects) – see "The cathedral and the bazaar" from E. Raymond [148].

Although there are great differences between the different lifecycle models, some steps happen to be present in the definition of all development processes:

- ⇝ **Requirements**: the expectations of the product have to be defined, either completely (waterfall model) or at least partially (iterative models). These can be formalised as use cases or mere lists of capabilities.
- ⇝ **Design**: draw sketches to answer the requirements, define the architecture and plan development.
- ⇝ **Implementation**: write code, tests, documentation and everything needed to generate a working product.
- ⇝ **Verification**: check that the initial requirements are answered with the developed product. This is accomplished through testing *before* delivery, and through customer acceptance *after* delivery.
- ⇝ **Maintenance** is the evolution of the product after delivery, to include bug fixes or new features. It represents the most expensive phase of the product lifecycle since it can last for years.

The first established development process in the 1960s and 1970s was the waterfall model, which has a strong focus on planning and control [20]. Every aspect of the software has to be known up-front, then it is decomposed in smaller parts to tackle complexity (divide-and-conquer approach). The sub-components are developed, tested, and then re-assembled to build the full integrated system. Figure 2.1 depicts a typical waterfall process.

The next-generation development processes use more than one iteration to build a system, delivering at the end of each cycle a partial product with an increasing scope of functionality. The iterative model, introduced by Basili and Turner [19], or the spiral model presented by Boehm [26] are early examples of these. Figure 2.2 shows a typical iterative development process.

Iterative models, which are at the heart of agile-like methods, have now become the mainstream development model [103]. Agile methods define short iterations to deliver

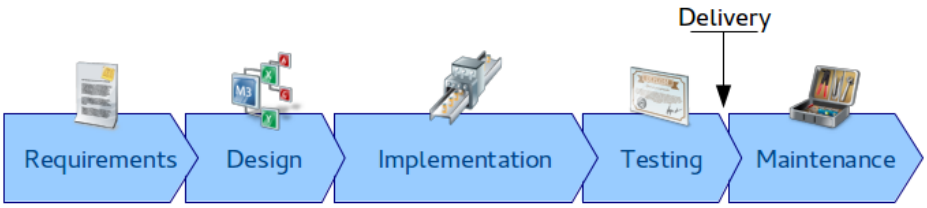


Figure 2.1: A typical waterfall development model.

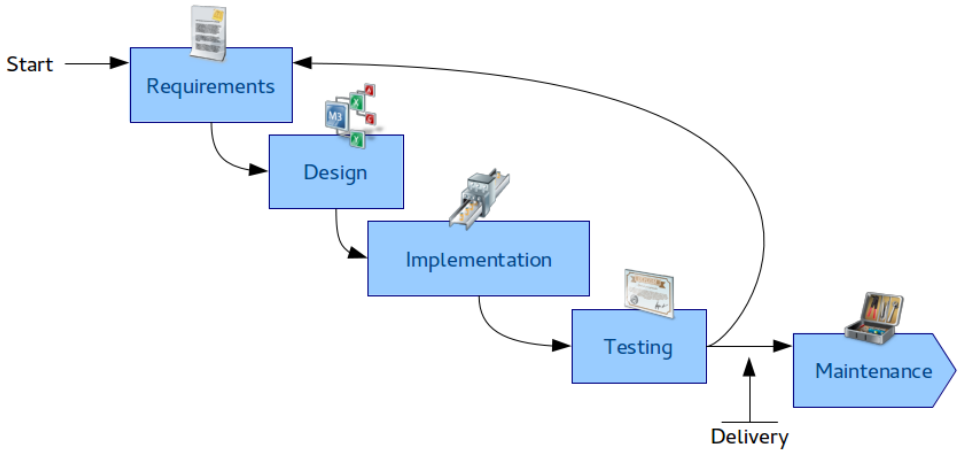


Figure 2.2: A typical iterative development model.

small increments that can be executed and tested, providing early feedback to the team and leveraging the risks. The Agile Manifesto [21], reproduced below, summarises the main principles of the Agile movement. Extreme Programming, Kanban, Scrum, and Lean software development are major representatives of the agile methods.

- ⇝ Individuals and interactions over Processes and tools
- ⇝ Working software over Comprehensive documentation
- ⇝ Customer collaboration over Contract negotiation
- ⇝ Responding to change over Following a plan

2.1.2 Development practices

Practices are methods and techniques used in the elaboration of a product. Experience defines good and bad practices, depending on the impact they have on quality attributes. Together they form the body of knowledge that software engineering builds on its way to maturity.

Product-oriented practices

A good example of practices are design patterns. They are defined by Gamma et al. [70] as "a general reusable solution to a commonly occurring problem within a given context". They provide safe, experience-backed solutions for the design, coding and testing of a system or product that improve areas like flexibility, reusability or understandability [70]. Y.G. Guéhéneuc et al. propose in [75] to use design patterns as the defining bricks of a dedicated quality model, and link patterns to a larger variety of attributes of quality like operability, scalability or robustness.

As for naming conventions, tools like Checkstyle allow to tailor formatting rules to check for the project's local customs. This however implies the use of custom configurations to describe the specific format of the project; only a few projects provide in the configuration management repository a configuration file for the different checks that fit the project traditions.

Some coding convention violations may be detected as well – SQuORE, Checkstyle, PMD, and Findbugs [154, 183, 8, 181] detect a few patterns that are known to be wrong, like missing defaults in switches, racing conditions, or under-optimal constructions. Although programming languages define a wide spectrum of possible constructions, some of them are discouraged by usage recommendations and can be statically verified.

Process-oriented practices

Examples of process-oriented practices are milestones and reviews planning for conducting the project. Most often these practices are informal: people will organise a meeting through mailing lists, doodles, or phone calls, and coding conventions may be implicitly defined or scattered through a wide variety of documents (mailing lists, wiki, code itself). As a consequence, many process-related practices are identified through binary answers like: "Is there a document for coding conventions?" or "Are there code reviews?".

Hence it is not always easy to detect these practices: the variety of development processes and tools, and the human nature of project management, make them difficult to measure consistently. Furthermore when coming to meaningful thresholds it is difficult to find a value that fits all projects: for agile-like methods, milestones and reviews should take place at each sprint (which lasts only a few weeks) while they often happen only later in the process for waterfall-like processes. Projects also often rely on different tools or even different forges: Jenkins uses GitHub for its code base, but has its own continuous integration server, bug tracking (JIRA) and website hosting.

From process practices to product quality

The belief that adherence to a good development process leads to an improved software product was a long-standing open question. Dromey [53] claims this is not enough, and advocates practitioners not to forget to "construct, refine and use adequate product quality models [53]". Kitchenham and Pfleeger reinforce this opinion by stating:

> “ There is little evidence that conformance to process standards guarantees good products. In fact, the critics of this view suggest that process standards guarantee only uniformity of output [...]. ”
>
> B. Kitchenham and S.L. Pfleeger [111]

However, studies conducted at Motorola [56, 49] and Raytheon [80] show that there is indeed a correlation between the maturity level of an organisation as measured by the CMM and the quality of the resulting product. These cases show how a higher maturity level can lead to improved error/defect density, lower error rate, lower cycle time, and better estimation capability.

There is no silver bullet, though, and what works well in some contexts may fail or simply be not applicable in other situations.

2.2 Software measurement

2.2.1 The art of measurement

The art of measurement is quite mature, compared to the field of software engineering. Humans have used measures for milleniums, to count seasons, elks, big white ducks, or land. From this rough counting practice, engineers have built upon and defined clear rules for better measurement. Hence from a rigorous perspective, measurement is *the process by which numbers or symbols are assigned to attributes of entities in the real world in such a way as to describe them according to clearly defined rules* [62, 66].

The measurement theory clearly defines most of the terms used here; however for some more specific aspects of software, we will rather rely on established definitions, either borrowed from standards or recognised authors.

- ⇝ *Measurement* is the process by which numbers or symbols are assigned to attributes of entities in the real world in such a way as to describe them according to clearly defined rules [62].
- ⇝ As a consequence, a *Measure* is the variable (either number or symbol) assigned as the result of a measurement.
- ⇝ A *Metric* is a defined measurement method and the associated measurement scale [94]. An example of metric is the Lines Count, which is the raw number of lines of an artefact (application, file, function) using an ordinal scale (the number of lines).
- ⇝ *Quality Attributes* or *Quality Factors* are attributes of software that contribute to its quality [158]. Examples of quality attributes include readability of code, reliability of product, or popularity of the project.
- ⇝ *Quality Models* organise quality attributes according to a specific hierarchy of characteristics. Examples of quality models include those proposed by McCall [131] and Boehm et al. [28, 27], or standards like ISO 9126 [94] or CMMi [174].

As for the big picture, *quality models* define specific aspects of software (*quality attributes*) like maintainability, usability or reliability, which are decomposed in sub-characteristics: e.g. analysability, changeability, stability, testability for maintainability. These sub-characteristics themselves rely on a set of *metrics*; there is generally a many-to-many relationship between quality attributes and metrics: control-flow complexity has an impact on both analysability and testability, while analysability of a software program is commonly measured as a function of control-flow complexity and respect of coding conventions.

The representation condition of measurement

According to Fenton, another requirement for a valid measurement system is the *representation condition*, which asserts that the correspondence between empirical and numerical relations is two way [62]. As applied to the measurement system of human height it means that: 1. If Jane is taller than Joe, then everyone knows that Jane's measure is greater than Joe's. 2. If Jane's measure is greater than Joe's, then everyone knows that Jane is taller than Joe.

Mathematically put, if we define the measurement representation M and suppose that the binary relation $\prec$ is mapped by M to the numerical relation $<$. Then, formally, we have the following instance:

$$x \prec y \iff M(x) < M(y)$$

This is one of the most failing reasons for software metrics [62]. An example would be to take as a representation the cyclomatic number as a measure of maintainability. Respecting the representation condition would imply that if code A has a higher cyclomatic number than code B, then code A is *always* less maintainable than code B – which is not true, since it also depends on a variety of other parameters like coding conventions and indentation.

The Goal-Question-Metric approach

In [20], Basili et al. review the requirements of a sound measurement system. To be effective, the application of metrics and models in industrial environments must be:

- ⇝ Focused on goals.
- ⇝ Applied to all life-cycle products, processes and resources.
- ⇝ Interpreted based on characterization and understanding of the organisational context, environment and goals.

They propose a three levels model in order to build a measurement system, further detailed in figure 2.3:

- ⇝ At the conceptual level, **goals** are to be defined for objects of measurement (product, process, and resources).

⇝ At the operational level, questions are used to characterise the way the assessment is going to be performed.
⇝ At the quantitative level, a set of metric is associated to every question to answer it in a quantitative way.

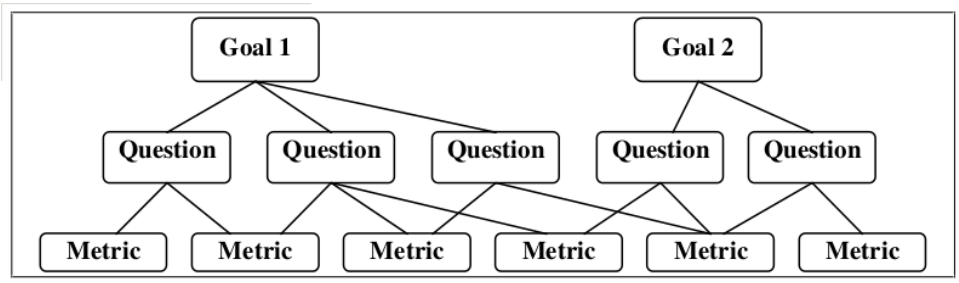


Figure 2.3: The Goal-Question-Metric approach.

This approach has proven to be useful for the consistency and relevance of systems of measurement. It was further commented and expanded by L. Westfall [185], who proposes 12 steps to conduct a successful Goal-Question-Metric process in industrial contexts.

The Hawthorne effect

The idea behind the Hawthorne effect is that participants may, either consciously or unconsciously, alter their behaviour because they know they are being studied, thus undermining the experiment. The term was coined in 1950 [68] and refers to a series of experiments conducted from 1927 to 1933 on factory workers at Western Electric's Hawthorne Plant. It showed that regardless of the changes made in the working conditions (number and duration of breaks, meals during breaks, work time) productivity increased. These changes apparently had nothing to do with the workers' responses, but rather with the fact they saw themselves as special, participants in an experiment, and their inter-relationships improved. Although this study has been deeply debated [141, 99, 124] as for its own conclusions, the influence of a measurement process on people should never be under-estimated.

Ecological inference

Ecological inference is the concept that an empirical finding at an aggregated level (e.g. package or file) can apply at the atomic sub-level (e.g. resp. files or functions). If this inference does not apply, then we have the *ecological fallacy*. The concept of ecological inference is borrowed from geographical and epidemiological research. It was first noticed by Robinson as early as 1950 [152], when he observed that at an aggregated level, immigrant status in U.S. states was positively correlated (+0.526) with educational achievement, but at the individual level it was negatively correlated (-0.118). It was attributed to the

congregational tendency of human nature, but it introduced a confounding phenomenon which jeopardised the internal validity of the study at the aggregated level.

Postnett et al. [146] discuss ecological inference in the software engineering field and highlight the risks of metrics aggregation in software models. There are many cases in software engineering where we want to apply ecological inference. As an example, the cyclomatic complexity as proposed by McCabe [130] is measured at the function level, but practitioners also often consider it at the file or application level.

2.2.2 Technical debt

The technical debt concept is a compromise on one dimension of a project (e.g. maintainability) to meet an urgent demand in some other dimension (e.g. a release deadline). Every work accomplished on the not-quite-right code counts as interest on the debt, until the main debt is repaid through a rewrite. Martin Fowler puts it nicely in a 2009 column [67]:

> “ You have a piece of functionality that you need to add to your system. You see two ways to do it, one is quick to do but is messy - you are sure that it will make further changes harder in the future. The other results in a cleaner design, but will take longer to put in place. ”
>
> Martin Fowler

The concept is not new – it was first coined by Ward Cunningham in 1992 [43] – but is now attracting more and more interest with the increasing concerns about software development productivity. Many software analysis tools and methods propose a component based on the technical debt: e.g. SQUORE, Sqale [122] or SonarQube. Another strength of the technical debt lies in its practical definition: developers and stakeholders easily understand it, both for weighing decisions and from an evaluation perspective. Furthermore, it defines a clear roadmap to better quality.

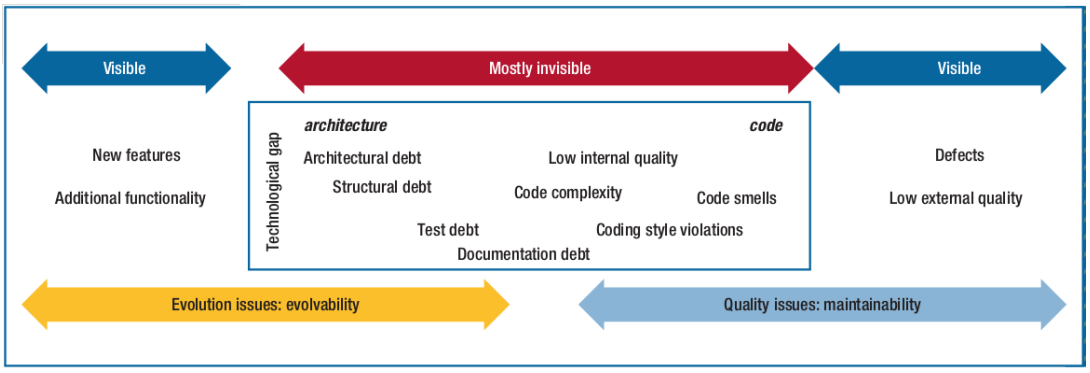


Figure 2.4: Technical debt landscape [117].

Technical debt is not always bad, though: it may be worth doing changes immediately for an upcoming release rather than missing the time-to-market slot. In these cases, practitioners just have to manage the technical debt and keep it under control [31]. Code technical debt (as opposed to architectural or structural, see figure 2.4) is usually based on a few common metrics (e.g. complexity, size) and the number of non-conformities detected in the source [117]. When comparing software products, the technical debt is often expressed as an index by dividing its amount by the number of files or lines of code.

2.3 Quality in software engineering

2.3.1 A few words about quality

Defining quality is not only a matter of knowing who is right or close to the truth: as we will see in this section, the plethora of different definitions show how much this concept can evolve, depending on the domain (e.g. critical embedded systems/desktop quick utility), the development model (e.g. open/closed source), or who is qualifying it (e.g. developer/end-user). Nevertheless, one needs to know some of the canonical definitions to better understand usual expectations and requirements, and to get another view on one's local quality context.

What is called quality is important because it really defines what the different actors of a software project expect from the system or service delivered and what can be improved. The definition of quality directly defines how a quality model is built.

2.3.2 Garvin's perspectives on quality

David Garvin [71] stated five different perspectives of quality that are still relevant to modern views on software quality:

- ⇝ The **transcendental perspective** deals with the metaphysical aspect of quality. In this view of quality, it is "something toward which we strive as an ideal, but may never implement completely" [111]. It can hardly be defined, but is similar to what a federal judge once commented about obscenity: "I know it when I see it" [103].
- ⇝ The **user perspective** is concerned with the appropriateness of the product for a given context of use. Whereas the transcendental view is ethereal, the user view is more concrete, grounded in the product characteristics that meet user's needs [111].
- ⇝ The **manufacturing perspective** represents quality as conformance to requirements. This aspect of quality is stressed by standards such as ISO 9001, which defines quality as "the degree to which a set of inherent characteristics fulfils requirements" (ISO/IEC 1999b). Other models like the Capability Maturity Model state that the quality of a product is directly related to the quality of the engineering process, thus emphasising the need for a manufacturing-like process.
- ⇝ The **product perspective** implies that quality can be appreciated by measuring the inherent characteristics of the product. Such an approach often leads to a

bottom-up approach to software quality: by measuring some attributes of the different components composing a software product, a conclusion can be drawn as to the quality of the end product.

⇝ The **final perspective** of quality is value-based. This perspective recognises that the different perspectives of quality may have different importance, or value, to various stakeholders.

The manufacturing view has been the predominant view in software engineering since the 1960s, when the US department of Defence and IBM gave birth to Software Quality Assurance [182].

2.3.3 Shewhart

Shewhart was a physicist, engineer and statistician working for Bell laboratories in the early 1930's. He was the first to introduce *statistical control* in the manufacturing industry, and is now recognised as the founding father of *process quality control*. His work has been further commented and expanded by Edward Deming. In a book published in 1931, *Economic Control of Quality of Manufactured Product* [160], Shewhart proposes a sound foundation for quality, which is still used and referenced nowadays in the industry:

> “ There are two common aspects of quality: one of them has to do with the consideration of the quality of a thing as an objective reality independent of the existence of man. The other has to do with what we think, feel or sense as a result of the objective reality. In other words, there is a subjective side of quality. ”
>
> Walter A. Shewhart

2.3.4 Crosby

In “Quality is free: The art of making quality certain” [42], Crosby puts the *conformance to requirements* as the main characteristic of quality, mainly because it allows to more easily define, measure and manage the concept. However, he tries to widen the scope of requirements by including an external perspective to the traditional production perspective.

> “ As follows quality must be defined as *conformance to requirements* if we are to manage it. Consequently, the non-conformance detected is the absence of quality, quality problems become non-conformance problems, and quality becomes definable. ”
>
> Philip B. Crosby

In the same essay, Crosby enumerates four key points to be the foundations of a quality system:

⇝ Conformance to requirements, as opposed to goodness or elegance.
⇝ System for causing quality is prevention, not appraisal.
⇝ Performance standard must be Zero Defect.
⇝ Measurement of quality is the price of non-conformance.

2.3.5 Feigenbaum

Feigenbaum is one of the originators of the Total Quality Management movement [61]. He stresses the importance of meeting customer needs, either "stated or unstated, conscious or merely sensed, technically operational or entirely subjective". As such, quality is a dynamic concept in constant change. Product and service quality is a multidimensional concept and must be comprehensively analysed; it can be defined as "the total composite product and service characteristics of marketing, engineering, manufacture and maintenance through which the product and service in use will meet the expectations of the customer".

Feigenbaum and the Total Quality Management system greatly influenced quality management in the industry: it was a sound starting point for Hewlett-Packard's Total Quality Control, Motorola's Six Sigma, or IBM's Market Driven Quality [103].

2.3.6 Deming

Deming was an early supporter of Shewhart's views on quality. In "Out of the crisis: quality, productivity and competitive position" [46], he defines quality in terms of customer satisfaction. Deming's approach goes beyond the product characteristics and encompasses all aspects of software production, stressing the role of management: meeting and exceeding customer expectations should be the ultimate task of everyone within the company or organisation.

Deming also introduces 14 points to help people understand and implement a sound quality control process that spans from organisational advice (e.g. break down barriers between departments, remove numerical targets) to social and philosophical considerations (e.g. drive out fear, institute leadership) . Interestingly, agile practitioners would probably recognise some aspects of this list which are common in modern agile processes: drive out fear, educate people, constantly improve quality and reduce waste...

2.3.7 Juran

In his "Quality Control Handbook" [101], Juran proposes two definitions for quality: 1. quality consists of those product features which meet the needs of customers and thereby provide product satisfaction; and 2. quality consists of freedom from deficiencies. He concludes the paragraph however with a short, standardised definition of quality as "fitness for use":

> “ Of all concepts in the quality function, none is so far-reaching and vital as “fitness for use”. [...] Fitness for use is judged *a*s seen by the user, not by the manufacturer, merchant, or repair shop. ”
>
> J.M. Juran [101]

According to Juran, fitness for use is the result of some identified parameters that go beyond the product itself:

- ⇝ Quality of design: identifying and building the product or service that meets the user's needs. It is composed of:
  1. Identification of what constitutes fitness for use to the user: quality of market research;
  2. Choice of a concept or product or service to be responsive to the identified needs of the user; and
  3. Translation of the chosen product concept into a detailed set of specifications which, if faithfully executed, will then meet the user's needs.

  There are also some “abilities” specific to long-lived products: availability, reliability, and maintainability.
- ⇝ Quality of Conformance. The design must reflect the needs of fitness for use, and the product must also conform to the design. The extent to which the product does conform to the design is called “quality of conformance”.
- ⇝ Field Service: following the sale, the user's ability to secure continuity of service depends largely on some services the organisation should provide: training, clear and unequivocal contracts, repairing, conducting affairs with courtesy and integrity.

2.4 Quality Models in Software Engineering

2.4.1 A few words about quality models

Why do we need quality models?

Quality models help organise the different aspects of quality into an extensive and consistent structure [45]. In some sense, they help to explicitly define implicit needs for the product or service. By linking the quality characteristics to defined measures, one is then able to assess the compliance of a product or project to the expectations [54, 93].

By putting words on concepts, individuals can formalise the fuzzy notion of quality and communicate them to others [93, 45]. Since there is no single definition of quality, it allows people to at least agree on a localised, context-safe compromise on what users (developers, stakeholders, management..) expect from from a *good* product or project. Quality models ultimately provide a structure to collaborate on a common framework for quality assessment – which is the first step to measure, then improve quality.

Common critics

Quality models that propose means to evaluate software quality from measures generally rely on numerical or ratio scales. But in the end, appreciation of quality is human-judged, and humans often use non-numeric scales [62, 105]. There have been various attempt to integrate new areas of knowledge: quality models target different types of quality measurement and factors, from community-related characteristics in FLOSS models to user satisfaction surveys. Khosravi et al. [110] propose to introduce different categories of inputs to catch information present in artefacts that go beyond source code.

A single quality model cannot rule all software development situations. The variety of software development projects poses a big threat on the pertinence of universally defined attributes of quality [110]. Small, geographically localised projects (like a ticketing system for a small shop) have different quality requirements than large, distributed, critical banking systems. Links between quality attributes and metrics are jeopardised by many factors, both dependent on the type of project and on its environment [105]. Some metrics are not available in specific contexts (e.g. because of the tools or the development process used) or have a different meaning. This in turn induces different thresholds values of metrics for what is fair, poor or forbidden.

Standard requirements evolve. Software, for industry in the 80's, had different constraints than we have nowadays: processing time was a lot more expensive, and some conceptions were still associated to hardware constraints. The requirements of quality have thus greatly evolved, and some characteristics of older quality models are no longer relevant. Still, one needs to know the different approaches for both the referencing part (terminology, semantics, structure) and the model completeness (there is no single quality model with all available characteristics).

2.4.2 Product-oriented models

McCall

The McCall model, also known as the General Electrics Model, was published by Jim McCall et al. in 1977 [131]. It can be seen as an attempt to bridge the gap between users and developers by focusing on a number of quality factors that reflect both the user's views and the developer's priorities.

McCall identifies 3 areas of software quality:

- ⇝ *Product Operation* refers to the product's ability to be quickly understood, efficiently operated and capable of providing the results required by the user. It includes correctness, reliability, efficiency, integrity, and usability.
- ⇝ *Product Revision* is the ability to undergo changes, including error correction and system adaptation. It includes maintainability, flexibility, and testability.
- ⇝ *Product Transition* is the adaptability to new environments, e.g. distributed processing or rapidly changing hardware. It includes portability, reusability, and interoperability.

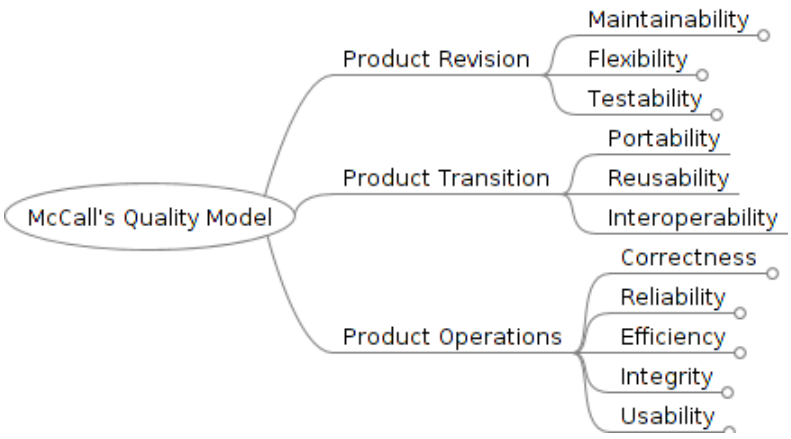


Figure 2.5: McCall's quality model.

Each quality factor is defined as a linear combination of metrics:

$$F_a = c_1.m_1 + c_2.m_2 + \ldots + c_n.m_n$$

McCall defines 21 metrics, from auditability (the ease with which conformance to standards can be checked) to data commonalities (use of standard data structures and types) or instrumentation (the degree to which a program monitors its own operations and identifies errors that do occur). Unfortunately, many of these metrics can only be defined subjectively [111]: an example of a question, corresponding to the Self-documentation metric is: "is all documentation structured and written clearly and simply such that procedures, functions, algorithms, and so forth can easily be understood?".

Boehm

Barry W. Boehm proposed in a 1976 article on the "Quantitative evaluation of software characteristics" [28] a new model targeted at product quality. It discusses the organisation of quality characteristics for different contexts and proposes an iterative approach to link them to metrics. One of the strong points of Boehm is the pragmatic usability of the model and its associated method: "one is generally far more interested in *where* and *how* rather than *how often* the product is deficient."

The model defines 7 quality factors and 15 sub-characteristics that together represent the qualities expected from a software system:

- ⇝ Portability: it can be operated easily and well on computer configurations other than its current one.
- ⇝ Reliability: it can be expected to perform its intended functions satisfactorily.
- ⇝ Efficiency: it fulfils its purpose without waste of resource.
- ⇝ Usability: it is reliable, efficient and human-engineered.
- ⇝ Testability: it facilitates the establishment of verification criteria and supports evaluation of its performance.

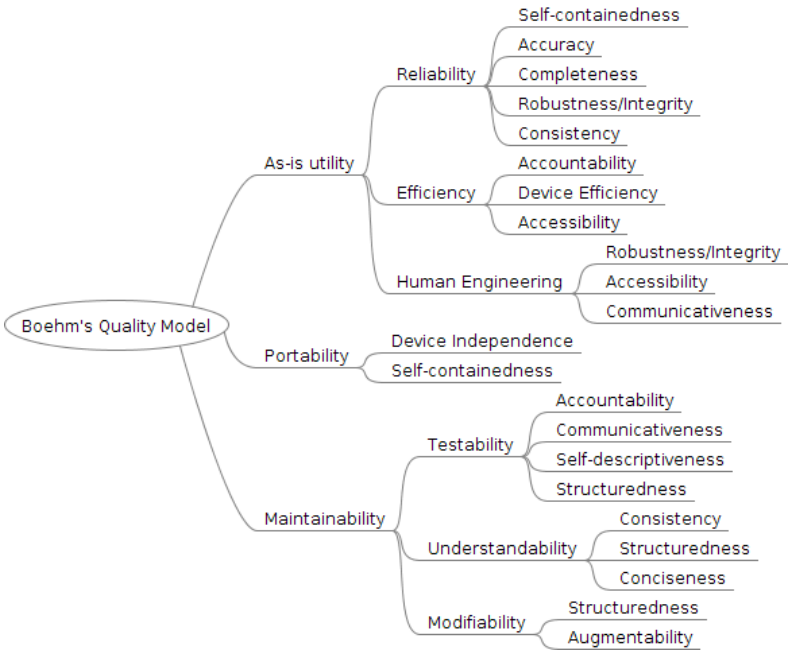


Figure 2.6: Boehm's quality model.

- ⇝ Understandability: its purpose is clear to the inspector.
- ⇝ Modifiability: it facilitates the incorporation of changes, once the nature of the desired change has been determined.

FURPS/FURPS+

The FURPS model was originally presented by Robert Grady and extended by Rational Software into FURPS+3. It is basically structured in the same manner as Boehm's and McCall's models. FURPS is an acronym which stands for:

- ⇝ Functionality – which may include feature sets, capabilities and security.
- ⇝ Usability – which may include human factors, aesthetics, consistency in the user interface, online and context-sensitive help, wizards and agents, user documentation, and training materials.
- ⇝ Reliability – which may include frequency and severity of failure, recoverability, predictability, accuracy, and mean time between failure.
- ⇝ Performance – imposes conditions on functional requirements such as speed, efficiency, availability, accuracy, throughput, response time, recovery time, and resource usage.
- ⇝ Supportability – which may include testability, extensibility, adaptability, maintainability, compatibility, configurability, serviceability, install ability, localisability (internationalisation).

ISO/IEC 9126

The ISO/IEC 9126 standard is probably the most widespread quality framework in use nowadays in the industry. It defines an extensive glossary of terms for software concepts (process, product, quality characteristics, measurement, etc.), a recognised framework for product quality assessment and it even proposes links to some common metrics identified as being related to the quality attributes. According to Deissenboeck [45] topology, the ISO/IEC 9126 standard is a *definition model*. Metrics furnished aren't pragmatic enough however to raise it to an *assessment model*.

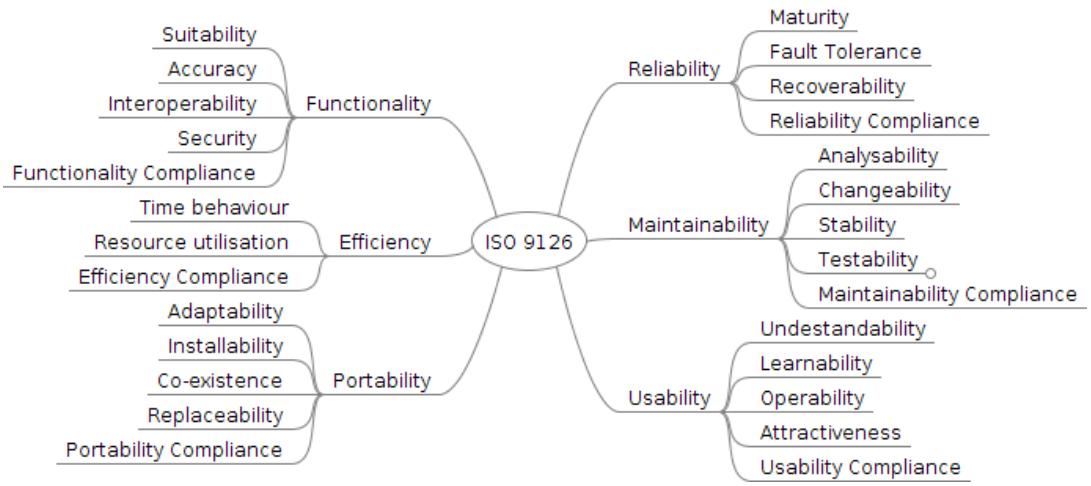


Figure 2.7: ISO/IEC 9126 quality model.

The standard makes an interesting difference between the various factors of quality. *Internal quality* is defined by the quality requirements from an internal view, i.e. independently from a usage scenario, and can be directly measured on the code. *External quality* is measured and evaluated when the software is executed in a given context, i.e. typically during testing. Finally, *quality in use* is the user's perspective on the software product when it is used in a specific environment and a specific context of use. It measures the extent to which users can achieve their goals in a particular environment, rather than measuring the properties of the software itself. *Quality in use* and *external quality* may have some metrics in common.

The first version of the model, ISO/IEC 9126:1991, has been replaced by two related standards: ISO/IEC 9126:2001 (Product quality) and ISO/IEC 14598 (Product evaluation). These in turn have been superseded by the ISO 250xx SQuaRE series of standards.

ISO/IEC 250xx SQuaRE

The SQuaRE series of standards was initiated in 2005 as the intended successor of ISO 9126. Most of its structure is modelled after its ancestor, with some enhancements on the scope (computer systems are now included) and a few changes on the sub-characteristics

(eight product quality characteristics instead of ISO 9126's six).

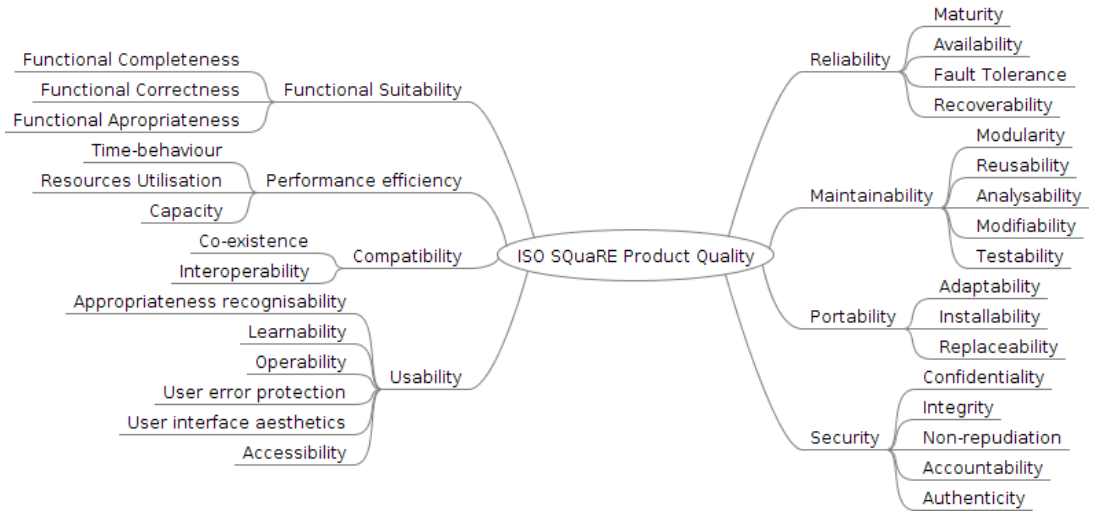


Figure 2.8: ISO/IEC 250xx quality model.

The series spans from ISO/IEC 25000 to 25099 and is decomposed in a four sub-series of standards:

- ⇝ Quality management. ISO/IEC 2500n presents and describes the SQuaRE series of standards, giving advice on how to read and use it. It can be seen as an introduction to the remaining documents.
- ⇝ Quality model. ISO/IEC 25010 defines the *quality in use* and *product quality* models. ISO/IEC 25012 defines a model for *data quality*, complementary to the above models. It provides help to define software and system requirements, the design and testing objectives, and the quality control criteria.
- ⇝ Quality measurement. ISO/IEC 2502n provides a measurement reference model and a guide for measuring the quality characteristics defined in ISO/IEC 25010, and sets requirements for the selection and construction of quality measures.
- ⇝ Quality requirement. ISO/IEC 2503n provides recommendations for quality requirements, and guidance for the processes used to define and analyse quality requirements. It is intended as a method to use the quality model defined in ISO/IEC 25010.
- ⇝ Quality evaluation. ISO/IEC 2504n revises the ISO/IEC 14598-1 standard for the evaluation of software product quality and establishes the relationship of the evaluation reference model to the other SQuaRE documents.

Although some parts are already published, the standard is still an ongoing work: available slots in the 25000-25099 range should be filled with upcoming complements, like 25050 for COTS (Components Off The Shelf) evaluation.

SQALE

SQALE (Software Quality Assessment based on Lifecycle Expectations) is a method to support the evaluation of a software application source code. Its authors intended it as an analysis model compliant with the representation condition of quality measurement [123]. It is a generic method, independent of the language and source code analysis tools, and relies on the ISO 9126 for the quality factors structure. Its main advantage over older quality models is it proposes pragmatic metrics to quantitatively compute quality from lower levels (code) to upper quality factors. It uses some concepts borrowed from the technical debt [122] and is gaining some interest from the industry in recent years.

2.4.3 Process-oriented models

The Quality Management Maturity Grid

The Quality Management Maturity Grid (QMMG) was introduced by P. Crosby in his book "Quality is free" [42] to assess the degree of maturity of organisations and processes. Although it has largely been superseded by other models like CMM (which recognises Crosby's work as a successor), it was an early precursor that laid down the foundation of all next-generation maturity assessment models. The model defines categories of measures (e.g. management understanding and attitude, quality organisation status or problem handling), and five levels of maturity:

- ⇝ *Uncertainty*: We don't know why we have problems with quality.
- ⇝ *Awakening*: Why do we always have problems with quality?
- ⇝ *Enlightenment*: We are identifying and resolving our problems.
- ⇝ *Wisdom*: Defect prevention is a routine part of our operation.
- ⇝ *Certainty*: We know why we do not have problems with quality.

The Capability Maturity Model

In the 1980's, several US military projects involving software subcontractors ran over-time, over-budget or even failed at an unacceptable rate. In an effort to determine why this was occurring, the United States Air Force funded a study at Carnegie Mellon's Software Engineering Institute. This project eventually gave birth to version 1.1 [142] of the Capability Maturity Model in 1993. Critics of the model, most of them revolving around the lack of integration between the different processes in the organisation, led to the development and first publication of the CMMi in 2002 [173]. The latest version of the CMMi is the 1.3 [174], published in 2010.

The CMMi defines two complementary representations for processes maturity evaluation. The *staged* view attaches a set of activities, known as process areas, to each *level of maturity* of the organisation. The *continuous* representation uses *capability levels* to characterise the state of the organisation's processes relative to individual areas. The CMMi is declined in 3 areas of interest: development, services, acquisitions.

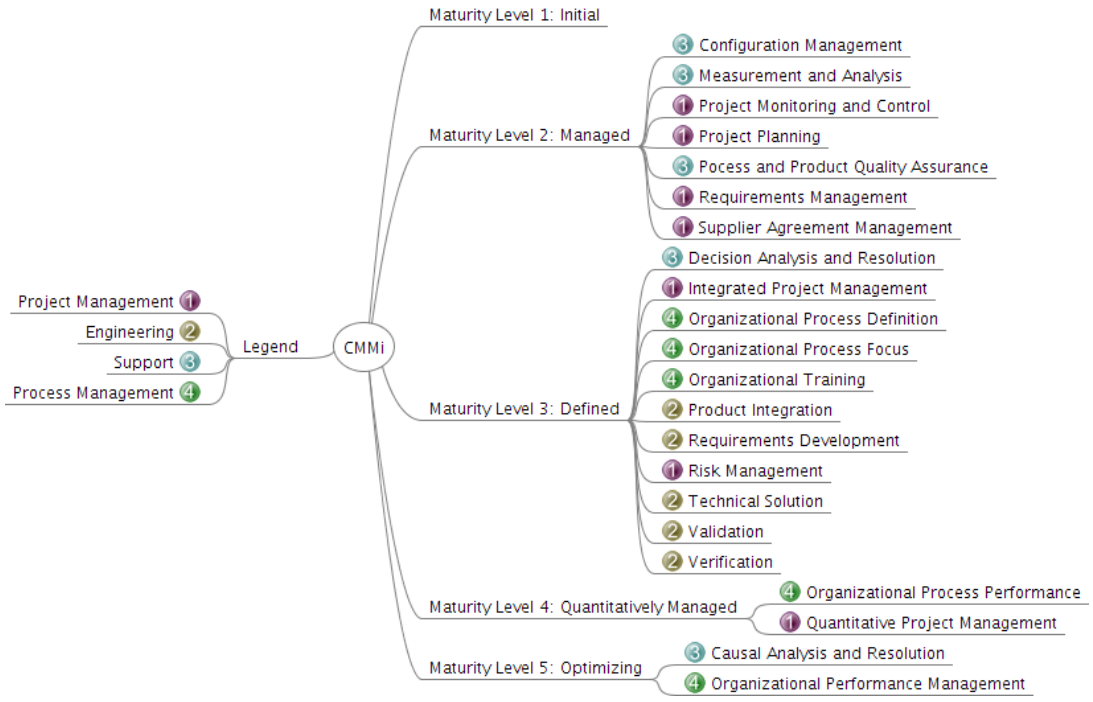


Figure 2.9: The Capability Maturity Model Integration.

CMMi-Development 1.3 defines five maturity levels and twenty-two process areas, illustrated in figure 2.9. Organisation-related maturity levels are initial, managed, defined, quantitatively managed, and optimising. Examples of process areas include causal analysis and resolution, configuration management, requirements management, risk management, or project planning. In the continuous representation, process areas are assessed through the following four capability levels: incomplete, performed, managed, and defined.

The CMMi information centre has recently been moved from the SEI web site to the CMMi Institute[1], which is hosted by Carnegie Innovations, the Carnegie-controlled technology commercialisation enterprise.

ISO 15504

The ISO 15504 standard, also known as SPICE (Software Process Improvement and Capability dEtermination), provides a guide for performing an assessment. ISO/IEC 15504 initially was derived from process lifecycle standard ISO/IEC 12207 and from maturity models like Bootstrap, Trillium and the CMM.

It includes the assessment process, the model for the assessment, and any tools used in the assessment, and is decomposed in two dimensions of software quality:

- ⇝ Process areas, decomposed in six categories: customer/supplier, engineering, supporting, management, organisation.

[1] http://cmmiinstitute.com/

⇝ Capability levels to measure the achievement of the different process attributes: incomplete, performed, managed, established, predictable, optimising. These are similar to CMM's levels of maturity.

2.4.4 FLOSS models

First-generation FLOSS models

Traditional quality models ignore various aspects of software unique to FLOSS (Free/Libre Open Source Software), most notably the importance of the community. Between 2003 and 2005 the first generation of quality assessment models emerged on the FLOSS scene. They were:

⇝ Open Source Maturity Model (OSMM), Capgemini, provided under a non-free license [55].
⇝ Open Source Maturity Model (OSMM), Navica, provided under the Academic Free license [137].
⇝ Qualification and Selection of Open Source Software, QSOS, provided by Atos Origin under the GNU Free Documentation license [140].
⇝ Open Business Readiness Rating, OpenBRR, provided by Carnegie Mellon West Centre for Open Source Investigation, sponsored by O'Reilly CodeZoo, SpikeSource, and Intel, and made available under a Creative Commons Attribution-NonCommercial-ShareAlike 2.5 license [184].

All models are based on a manual work, supported by evaluation forms. The most sophisticated tool support can be found in QSOS, where the evaluation is supported by either a stand-alone program or a Firefox plugin which enables feeding results back to the QSOS website for others to download. Still, the data gathering and evaluation is manual.

The status in 2010 is that none of these FLOSS quality models have been widely adopted, and few or none of them can really be considered a success yet. The OSMM Capgemini model has a weak public presence in the open source community, the web resources for the OSMM Navica are no longer available, and the OpenBRR community consists of an abandoned web site that is frequently not available. The QSOS project shows a slow growth in popularity [186], with half a dozen of papers published and many web sites referencing it. Unlike other quality models it is still under development, with a defined roadmap and active committers.

First-generation FLOSS models

The next generation of FLOSS quality model has learned from both traditional quality models and first-generation FLOSS quality models, notably with a more extensive tool support. They are:

- ⇝ QualOSS [47] is a high-level and semi-automated methodology to benchmark the quality of open-source software. Product, community and process are considered to be of equal importance for the quality of a FLOSS endeavour. It can be applied to both FLOSS products and components.
- ⇝ QualiPSo Open Source Maturity Model [144], a CMM-like model for FLOSS. It focuses on process quality and improvement, and only indirectly on the product quality [79].
- ⇝ Software Quality Observatory for Open Source Software (SQO-OSS) proposes a quality model and a platform with quality assessment plug-ins. It comprises a core tool with software quality assessment plug-ins and an assortment of UIs, including a Web UI and an Eclipse plugin [157]. The SQO-OSS is being maintained, but the quality model itself is not yet mature, and most of the focus is on the development of the infrastructure to enable the easy development of plug-ins.

2.5 Summary

In this chapter we walked through the landscape of software engineering, identifying the building blocks of our research. First, we quickly described major development principles, processes and practices needed to localise where and how we can measure and act on a software project. We defined the basis of measurement as applied to software projects, the pitfalls that one may face when applying it in different contexts, and the experience gathered on software measurement programs in the past decades.

We then presented a few definitions and models of quality. They provide a sound and (experience-grown) safe foundation to build custom, context-aware quality models. This is a mandatory step for establishing a common terminology and comprehension of the concepts used during the project, and helps to identify concerns and issues encountered in many different situations and in large, real-life projects. On the one hand, there are complex but fully-featured quality models, with pragmatic links between quality attributes and metrics and a method for their application. On the other hand, the most widespread quality models are those which are much simpler to understand or apply, even if they show some concerns or discrepancies.

What remains from this fish-eye view is the numerous definitions of quality for research and practitioners: even as of today, and despite of the many references available, people do not easily agree on it. Another difficult point is the link between the quality attributes and the metrics that allow to measure it pragmatically because of the definition, understanding and availability of metrics. As an example there is no common accord on the metrics that definitely measure maintainability and comply with the representational condition of measurement.

Chapter 3

Data mining

Data mining is the art of examining large data sources and generating a new level of knowledge to better understand and predict a system's behaviour. In this chapter we review some statistical and data mining techniques, looking more specifically at how they have been applied to software engineering challenges. As often as possible we give practical examples extracted from the experiments we conducted.

Statistics and data mining methods had evolved on their own and were not introduced into software-related topics until 1999 when Mendonce and Sunderhaft published a serious survey [132] summarising the state of data mining for software engineering for the DACS (Data and Analysis Center for Software). Since then *data mining methods for software engineering* as a specific domain of interest for research have emerged, with works on software quality predictors (Khoshgoftaar et al. [109] in 1999), test automation (M. Last et al. [119] in 2003), software history discovery (Jensens [97] in 2004)), prediction of source code changes (A. Ying et al. [195]), and recommenders to help developers find related changes when modifying a file (Zimmermann [202] in 2005). The specific application of data mining techniques to software engineering concerns is now becoming an established field of knowledge [190, 172].

This chapter is organised as follows. We start with some basic exploratory analysis tools in section 3.1. We then review the main techniques we investigated for our purpose: clustering in section 3.3, outliers detection in section 3.4, regression analysis in section 3.5, time series analysis in section 3.6, and distribution functions in section 3.7. Section 3.8 summarises what we learned.

3.1 Exploratory analysis

3.1.1 Basic statistic tools

The most basic exploration technique is a quick statistical summary of data: minimum, maximum, median, mean, variance of attributes. Table 3.1 shows an example of a summary for a few metrics gathered on Ant 1.8.1. This allows us to get some important information on the general shape of metrics, and to quickly identify and dismiss erroneous metrics

(e.g. if min and max are equal).

Table 3.1: Quick summary of common software metrics for Ant 1.8.1.

Metric	Min	1st Qu.	Median	Mean	3rd Qu.	Max	Var
CFT	0	4	16	37.43	41	669	3960.21
CLAS	1	1	1	1.42	1	14	1.65
COMR	5.14	39.2	50	51.13	64	92.86	334.22
SLOC	2	22	56	107.5	121	1617	25 222.24
VG	0	4	11	22.7	25	379	1344.14
SCM_COMMITS	0	4	12	22.11	30	302	962.92
SCM_COMMITTERS	0	1	2	3.44	5	19	10.97
SCM_FIXES	0	0	1	1.54	2	39	8.03

3.1.2 Scatterplots

Scatterplots are very useful to visually identify relationships in a multivariate dataset. As an example, figure 3.1 shows a scatterplot of some common file metrics for the Ant project: SLOC, VG, and SCM_COMMITS_TOTAL. We can immediately say that the SLOC and VG metrics seem to be somewhat correlated: they follow a rough line with little variance.

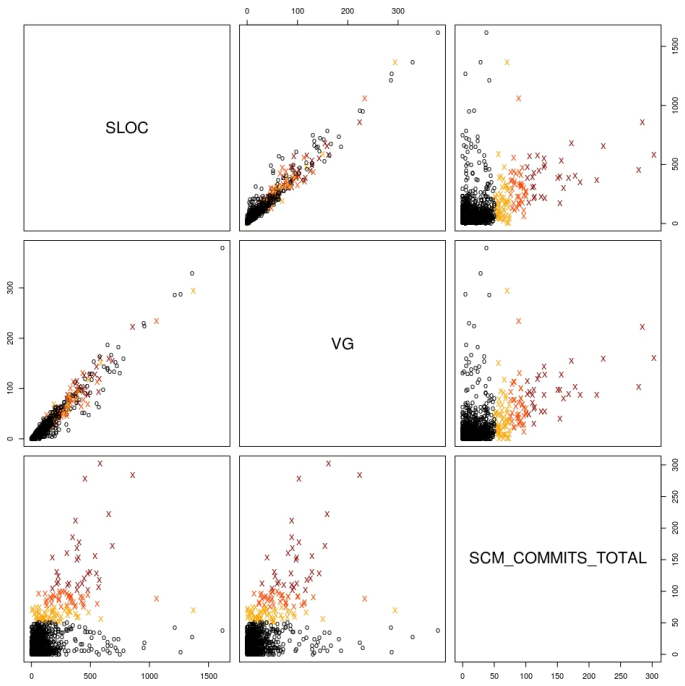


Figure 3.1: Scatterplots of some file metrics for Ant

Cook and Swayne [39] use scatterplots to show specific characteristics of data by

visually differencing some elements. It adds another dimension of information to the plot: in figure 3.1, files that have been heavily modified (more than 50, 75 and 100 commits) are plotted with resp. orange, light red and dark red crosses. The most unstable files happen at 2/3 of the metric's range, and only a few of the bigger (with a high value of SLOC) files are heavily modified.

3.2 Principal Component Analysis

The idea of principal component analysis (PCA) is to find a small number of linear combinations of the variables so as to capture most of the variation in the data as a whole. With a large number of variables it may be easier to consider a small number of combinations of the original data rather than the entire data set.

In other words, principal component analysis finds a set of orthogonal standardised linear combinations which together explain all of the variation in the original data. There are potentially as many principal components as there are variables, but typically it is the first few that explain important amounts of the total variation.

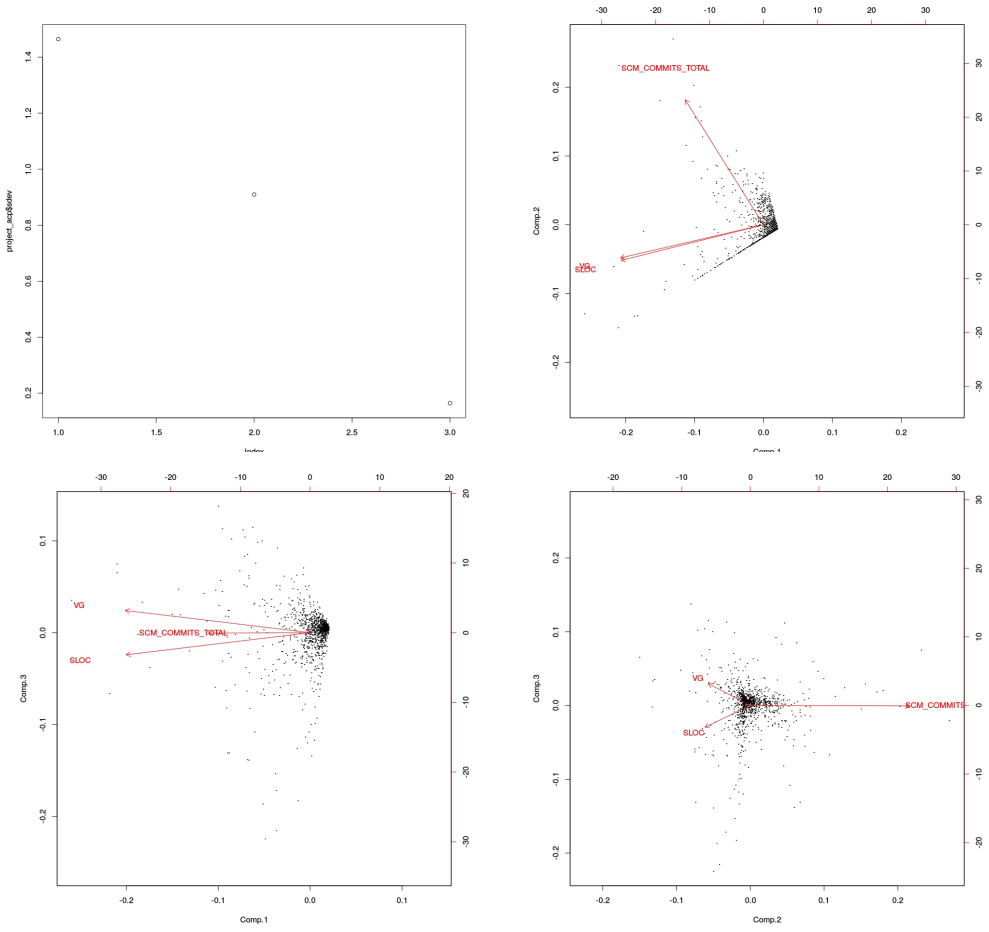


Figure 3.2: Principal Component Analysis for three metrics.

Figure 3.2 illustrates the application of principal component analysis on a set of three variables: SLOC, VG, SCM_COMMITS_TOTAL for the Ant 1.8.0 files. We know that SLOC and VG are correlated, i.e. they carry a similar load of information and tend to vary together. The first picture (upper left) shows the decreasing relative importance of each principal component as given by its variance. In this case the third component is insignificant compared to the variance of the first two, which means that most of the variations in the three variables can be summarised with only two of them. The three next plots show the projection of the metrics on the three components: SLOC and VG overlap on the first plot since they carry the same information, and are almost perpendicular to the SCM_COMMITS_TOTAL. The number of components to be selected varies; in our work the rule we adopted was to select only principal components that represent 20% of the first principal component amplitude.

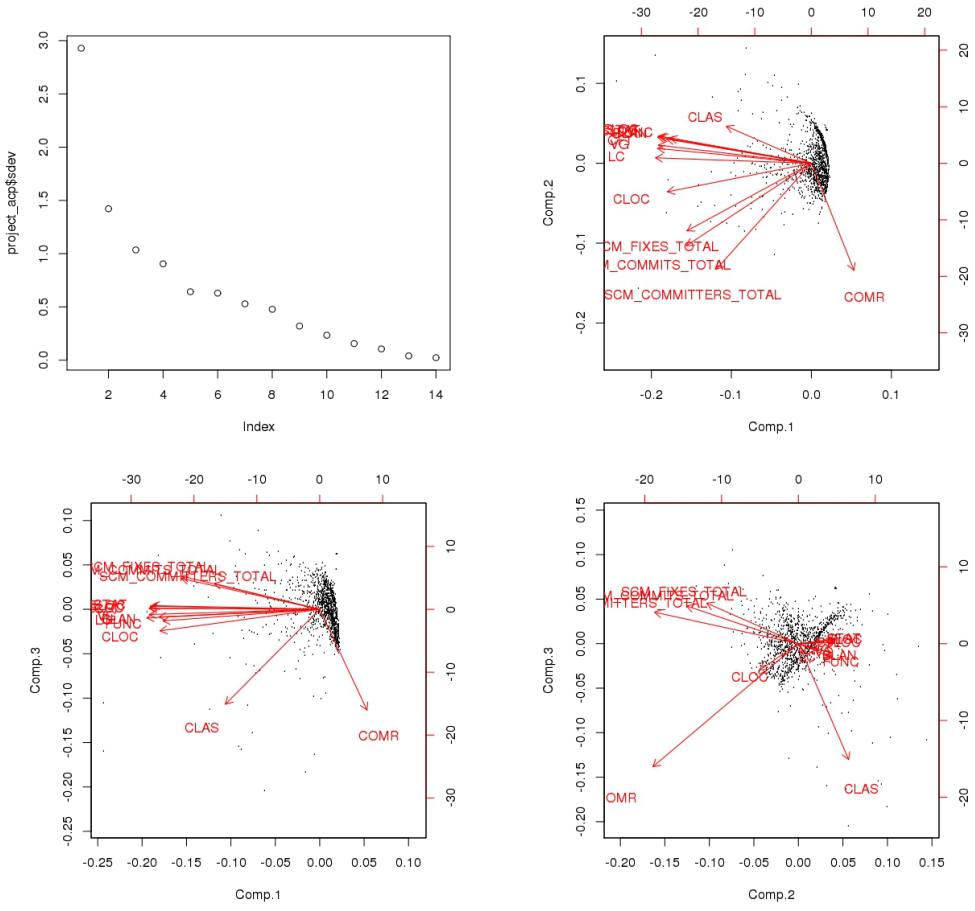


Figure 3.3: Principal Component Analysis for Ant 1.8.1 file metrics.

Figure 3.3 extends the previous example to a larger number of variables corresponding to the whole set of a project's files metrics. There are more components with a relatively high variance, which means that all of the variations cannot be summarised in one or two

axes. In the case depicted here, only the 4 first components are selected by the 20% rule.

It should also be noted that PCA works well on common shapes, i.e. with variables that show a Gaussian-like distribution, but is unable to sort out complex shapes like imbricated circles – most common algorithms will fail on these anyway.

Jung et al. [100] apply principal component analysis to the results of a survey on the comprehension of ISO 9126 quality characteristics by practitioners. They use it to propose a new decomposition of quality attributes that better suit people's perception of software quality. In [179] Turan uses backward feature selection and PCA for dimensionality reduction. On the NASA public data sets, the original set of 42 metrics is reduced to 15 with almost no information loss. Tsai and Chan [178] review various dimensionality reduction techniques and compare PCA with other methods. They find out that PCA works well for identifying coordinates and linear correlations in high dimensions, but is unsuitable for nonlinear relationships among metrics.

Kagdi et al. [102] apply PCA to software measurement data to identify main domains of variations among metrics and to build a prediction model for post-release defects. Nagappan et al. [135] also use PCA to remove correlated metrics before building statistical predictors to estimate post-release failure-proneness of components of the Windows Server 2003 operating system.

3.3 Clustering

Clustering techniques group items on the basis of similarities in their characteristics, with respect to multiple dimensions of measurement. Representing data by fewer clusters necessarily causes the loss of certain fine details but achieves simplification [24]. They are also used to automatically decide on an automatic repartition of elements based on their own characteristics [200].

The usability and efficiency of clustering techniques depend on the nature of data. In peculiar cases (e.g. with concentric circles or worm-like shapes) some algorithms simply will not work. The preparation of data is of primary importance to get the best out of these methods; treating variables through a monotonal transformation, or considering them in a different referential may be needed to get better results.

Clustering algorithms may be combined as well with of other techniques to achieve better results. As an example, Antonellis et al. [7] apply a variant of k-means clustering on top of the results of the AHP (Analytical Hierarchical Processing) of the output of a survey on ISO 9126 quality characterisitcs. This is what we also did for the principal component analysis and correlation matrix results of our product version analysis document – see section 4.2.5. Applying clustering to the correlation results brings more knowledge by identifying groups of inter-dependent artefacts or metrics.

Clustering is also used in outliers detection approaches by selecting elements that either lie out of clusters or are grouped in small clusters [23, 165]. This is described later in section 3.4.

3.3.1 K-means clustering

K-means is the most popular clustering tool used in scientific and industrial applications [24]; it aims at grouping n observations into k clusters in which each observation belongs to the cluster with the nearest *mean*. This results in a partitioning of the data space into Voronoi cells [59]. k-means algorithms are considered to be quite efficient on large data sets and often terminate at a local optimum, but are sensitive to noise [115].

There are alternatives to the K-means algorithm, which use other computation methods for the distance: *k-medoid* uses a median point in clusters instead of a weighted mean, and *k-attractors*, which was introduced by Kanellopoulos et al. to address the specificities of software measurement data [104]. In another study published in 2008 [193] Kanellopoulos uses the k-attractors clustering method to group software artefacts according to the ISO 9126 quality characteristics.

k-means clustering has also been used for evaluation of success of software reuse: Kaur et al. [107] apply k-means clustering to the results of a survey on the factors of success for software reuse, and are able to extract meaningful categories of practices with a good accuracy. Herbold et al. [88] apply k-means to time data (namely SLOC and BUGS) of Eclipse projects to automatically detect milestones. Zhong et al. compare in [200] clusters produced by two unsupervised techniques (k-means and neural-gas) to a set of categories defined by a local expert and discuss the implications on noise reduction. They find that both algorithms generally perform well, with differences on time and resource performance and errors varying according to the analysed projects. In another paper [201] the same authors cluster files in large software projects to extract a few representative samples, which are then submitted to experts to assess software quality of the whole file set. More recently Naib [136] applies k-means and k-medoids clustering to fault data (error criticality and error density) of a real-time C project to identify error-prone modules and assess software quality.

In another area of software, Dickinson et al. examine in [50] data obtained from random execution sampling of instrumented code and focus on comparing procedures for filtering and selecting data, each of which involves a choice of a sampling strategy and a clustering metric. They find that for identifying failures in groups of execution traces, clustering procedures are more effective than simple random sampling. Liu and Han present in [127] a new failure proximity metric for grouping of error types, which compares execution traces and considers them as similar if the fault appears to come from a close location. They use a statistical debugging tool to automatically localise faults and better determine failure proximity.

3.3.2 Hierarchical clustering

The idea behind hierarchical clustering is to build a binary tree of the data that successively merges similar groups of points [134, 115, 24]. The algorithm starts with each individual as a separate entity and ends up with a single aggregation. It relies on the distance matrix to show which artefact is similar to another, and to group these similar artefacts in the

same limb of a tree. Artefacts that heavily differ are placed in another limb.

The algorithm only requires a measure of similarity between groups of data points and a linkage criterion which specifies the dissimilarity of sets as a function of the pairwise distances of observations in the sets. The measure of distance can be e.g. Euclidean distance, squared Euclidean distance, Manhattan distance, or maximum distance. For text or other non-numeric data, metrics such as the Hamming distance or Levenshtein distance are often used. The following table lists the mathematical functions associated to these distances.

Euclidean distance	$\|a-b\|_2 = \sqrt{\sum_i (a_i - b_i)^2}$
Squared Euclidean distance	$\|a-b\|_2^2 = \sum_i (a_i - b_i)^2$
Manhattan distance	$\|a-b\|_1 = \sum_i \|a_i - b_i\|$
Maximum distance	$\|a-b\|_\infty = \max_i \|a_i - b_i\|$

The most common linkage criteria are the complete [44] (similarity of the furthest pair), the single [162] (similarity of the closest pair), and the average [168] (average similarity between groups) methods. They are listed in the next table with their computation formulae.

Complete clustering	$\max \{\, d(a,b) : a \in A,\, b \in B \,\}.$
Single clustering	$\min \{\, d(a,b) : a \in A,\, b \in B \,\}$
Average clustering	$\frac{1}{\|A\|\|B\|} \sum_{a \in A} \sum_{b \in B} d(a,b)$

Single linkage can produce chaining, where a sequence of close observations in different groups cause early merges of those groups. Complete linkage has the opposite problem: it might not merge close groups because of outlier members that are far apart. Group average represents a natural compromise, but depends on the scale of the similarities. Other linkage criteria include the Ward, McQuitty, Median, and Centroid methods, all of which are available in the `stats` R package.

Hierarchical clustering is usually visualised using a dendogram, as shown in figure 3.4. Each iteration of clustering (either joining or splitting, depending on the linkage) corresponds to a limb of the tree. One can get as many clusters as needed by selecting the right height cut on the tree: on the aforementioned figure the blue rectangles split the set in 3 clusters while the green rectangles create 5 clusters.

There are a few issues with hierarchical clustering, however. Firstly, most hierarchical algorithms do not revisit their splitting (or joining) decisions once constructed [24]. Secondly the algorithm imposes a hierarchical structure on the data, even for data for which such structure is not appropriate. Furthermore, finding the right number of natural

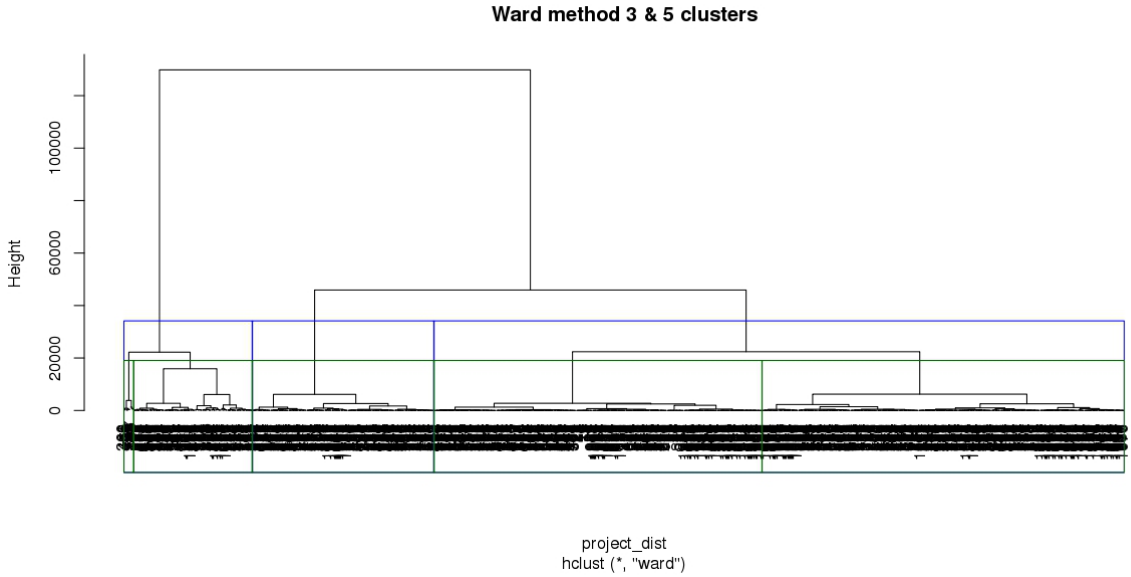


Figure 3.4: Hierarchical Clustering dendogram for Ant 1.7.0 files.

clusters may also be hard to achieve. Almeida and Barbosa propose in [3] an approach to automatically find the natural number of clusters present in the underlying organisation of data. They also remove outliers before applying clustering and re-integrate them in the established clusters afterwards, which significantly improves the algorithm's robustness.

Variants of hierarchical clustering include CURE and CHAMELEON. These are relevant for arbitrary shapes of data, and perform well on large volumes [24]. The BIRCH algorithm, introduced by Zhang [199] is known to be very scalable, both for large volumes and for high dimensional data.

3.3.3 DBSCAN clustering

Density-based clustering methods [115] try to find groups based on density of data points in a region. The key idea of such algorithms is that for each instance of a cluster the neighbourhood of a given radius (*Eps*) has to contain at least a minimum number of instances (*MinPts*).

One of the most well-known density-based algorithm is DBSCAN [59]; it starts with an arbitrary point in the data set and retrieves all instances with respect to Eps and MinPts. The algorithm uses a spatial data structure (SRtree [106]) to locate points within Eps distance from the core points of the clusters. Ester et al. propose an incremental version of the DBSCAN algorithm in [58].

3.4 Outliers detection

3.4.1 What is an outlier?

Basically put, outliers are artefacts that differ heavily from their neighbours. More elaborated definitions abund in the literature; in the following list the first two definitions are generic and the three latter are more targeted to software engineering measurement data.

- ⇝ For Hawkins [85] an outlier is *an observation that deviates so much from other observations as to arouse suspicion that it was generated by a different mechanism.*
- ⇝ For Barnett and Lewis [18], an outlier is *an observation (or subsets of observations) which appears to be inconsistent with the remainder of that set of data.*
- ⇝ Yoon et al. [197] define outliers as *the software data which is inconsistent with the majority data.*
- ⇝ For Wohlin et al. [187] an outlier denotes *a value that is atypical and unexpected in the data set.*
- ⇝ Lincke et al. [126] take a more statistical and practical perspective and define outliers as *artefacts with a metric value that is within the highest/lowest 15% of the value range defined by all classes in the system.*

The plot on the left of figure 3.5 shows 7 outliers highlighted on a set of 3-dimensions points. These differ from their neighbours by standing far from the average points on one or more of their dimensions. The coloured plot on the right presents an interesting case: most of the highlighted outliers are visually differing from the majority of points, excepted for two of them which are hidden in the mass of points. These are different from their neighbours on another dimension but seem to be normal on the two axes selected here.

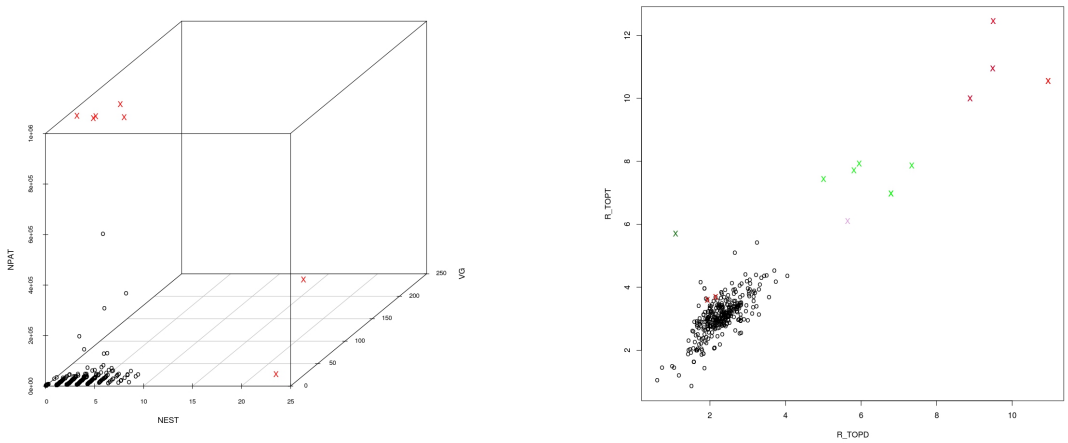


Figure 3.5: Examples of outliers on a few data sets.

Hodge [92], Ben-Gal [23] and Chandola [35] provide extensive reviews of outliers detection techniques. Cateni et al. [34] review their usage in a broad industrial application perspective. Many studies apply outliers detection to security-related data for e.g. insurance, credit card or telecommunication fraud detection [116, 145], insider trading detection [166] and network intrusion detection systems [121] by detecting deviating behaviours and unusual patterns. Outliers detection methods have also been applied to healthcare data [120, 188] to detect anomalous events in patient's health, military surveillance for enemy activities, image processing [166], or industrial damages detection [35]. Mark C. Paulk [143] applies outliers detection and Process Control charts in the context of the Personal Software Process (PSP) to highlight individual performance.

Another similar research area is the detection of outliers in large multi-dimensional data sets. There is a huge variety of metrics that can be gathered from a software project, from code to execution traces or software project's repositories, and important software systems commonly have thousands of files, mails exchanged in mailing lists or commits. This has two consequences: first in high dimensional space the data is sparse and the notion of proximity fails to retain its meaningfulness [1]. Second some algorithms are inefficient or even not applicable in high dimensions [1, 65].

It should also be noted that too many outliers may be a synonym of errors in the dataset [92, 35], e.g. missing values treated as zero's, bugs in the retrieval process, wrong measurement procedure. If they are justified, they can bring very meaningful information by showing values at exceptional ranges. Take bugs as an example: a high number of defects for a file would deserve more investigation into its other characteristics to determine what criteria are correlated to this fault-proneness.

3.4.2 Boxplot

Laurikkala et al. [120] use informal box plots to pinpoint outliers in both univariate and multivariate data sets. Boxplots show a great deal of information: the upper and lower limits of the box are respectively the first and third quartile[1] while the line plotted in the box is the median[2]. Points more than 1.5 times the interquartile range[3] *above the third quartile* and points more than 1.5 times the interquartile range *below the first quartile* are defined as outliers.

Figure 3.6 shows four boxplots for a sample of metrics: CLOC, ELOC, SLOC, LC, plotted with (right) and without (left) outliers. The points above the boxplots are the outliers, and the boxes define the first and last quartiles. As is shown on the figure, in the context of software measures there is often more space *above* the box than *below* the box, and outliers are mostly (if not all, as displayed in the figure) above the upper end of the

[1]The quartiles of a ranked set of data values are the three points that divide the data set into four equal groups, each group comprising a quarter of the data. First quartile is the value of the element standing at the first quarter, third quartile is the value of the element standing at the third quarter.

[2]The median is the numerical value separating the higher half of a data sample from its lower half.

[3]The interquartile range (IQR) is the difference between the third and the first quartiles.

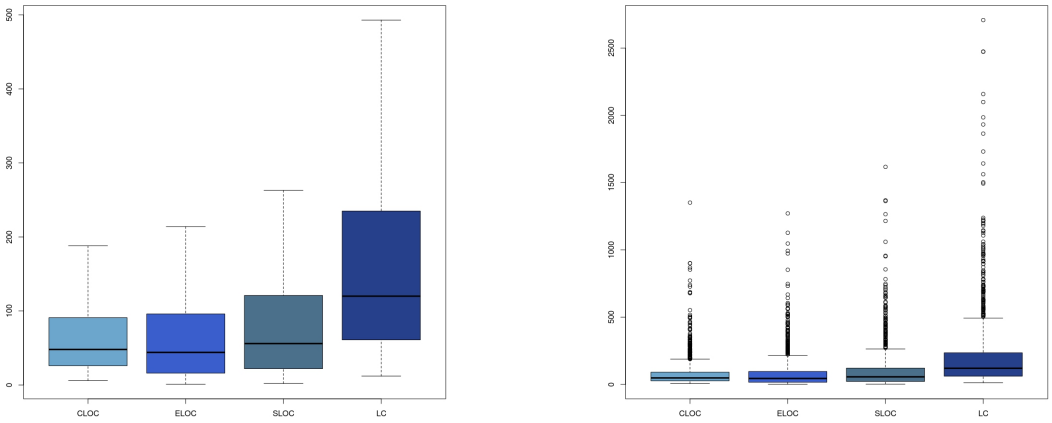


Figure 3.6: Boxplots of some common metrics for Ant 1.8.1, without and with outliers.

whiskers. This is consistent with the long-tailed distribution function observed for such metrics, as presented in sections 3.7 and 4.2.2.

3.4.3 Local Outlier Factor

LOF (Local Outlier Factor) is an algorithm for identifying density-based local outliers [30]. The local density of a point is compared with that of its neighbours. If the former is significantly lower than the latter (with a LOF value greater than 1), the point is in a sparser region than its neighbours, which suggests it is an outlier. A drawback of the local outlier factor method is that it works only for numerical values. Cateni et al. [34] compares the local outlier factor method with fuzzy-logic clustering techniques, showing that the latter outperforms LOF but needs more computational time (roughly by a factor of 10). LOF is also used by Dokas et al. [52] in network intrusion detection to identify attacks in TCP connection data sets.

One may use the `lofactor` function from the `DMwR` package [177] for such a task; the number of neighbours used for the LOF density can optionally be set. LOF outlier detection may be used either on univariate or multivariate data. As for visualisation, it should be noted that when the dissimilarities are computed on all metrics the highlighted outliers may not visually stand out on the two metrics used for the plot.

3.4.4 Clustering-based techniques

Clustering techniques are also often used for outliers detection. The rationale is that data points that either do not fit into any cluster or constitute very small clusters are indeed dissimilar to the remaining of the data [165]. Yoon et al. [197] propose an approach based on k-means clustering to detect outliers in software measurement data, only to remove them as they threaten the conclusions drawn from the measurement program. In the

latter case the detection process involves the intervention of software experts and thus misses full automation. Al-Zoubi [2] uses Partitioning Around Medians (PAM) to first remove small clusters (considered as outliers) for robustness and then re-applies clustering on the remaining data set. Other mining techniques have been applied with some success to software engineering concerns, such as recommendation systems to aid in software maintenance and evolution [33, 151] or testing [119].

3.5 Regression analysis

Regression analysis allows to model, examine, and explore relationships between a dependent (response or outcome) variable and independent (predictor or explanatory) variables. Regression analysis can help understand the dynamics of variable evolution and explain the factors behind observed patterns. *Regression models* define the formula followed by the variables and are used to forecast outcomes.

Regression analysis is used when both the response variable and the explanatory variable are continuous, and relies on the following assumptions [86]: 1) constant variance, 2) zero mean error, and 3) independence of independent input parameters.

The adequacy of the model is assessed using the coefficient of determination R^2 [133]. It measures the fraction of the total variation in y that is explained by variation in x. A value of $R^2 = 1$ means that all of the variations in the response variable are explained by variation in the explanatory variable. One of the drawbacks of R^2 is it artificially increases when new explanatory variables are added to the model. To circumvent this, the *adjusted* R^2 takes into account the number of explanatory terms in a model relative to the number of data points.

Usage of regression analysis

Postnett et al. use logistic regression in [146] to confirm relationships between the metrics-based selection of defective components and the actual bugs found post-release. Sowe et al. [169] use regression analysis to establish correlations between size, bugs and mailing list activity data of a set of large open-source projects. They find out that mailing list activity is significantly correlated ($R^2 = 0.79$) with the number of bugs and size is highly correlated ($R^2 = 0.9$) with bugs. Gyimothy et al. [78] apply machine learning techniques (decision tree and neural network) and regression analysis (logistic and linear) on fault data against object-oriented metrics for several versions of Mozilla. All four methods yield similar results, and classify metrics according to their ability to predict fault-proneness modules.

Regression analysis is widely used to build prediction models for effort estimation, software quality attributes evolution (maintainability, faults, community). In [60], Fedotova et al. draw a comprehensive study of effort estimation models and use multiple linear regression to predict efforts during a CMMi improvement program in a development organisation. Results outperformed expert-based judgement in the testing phase but

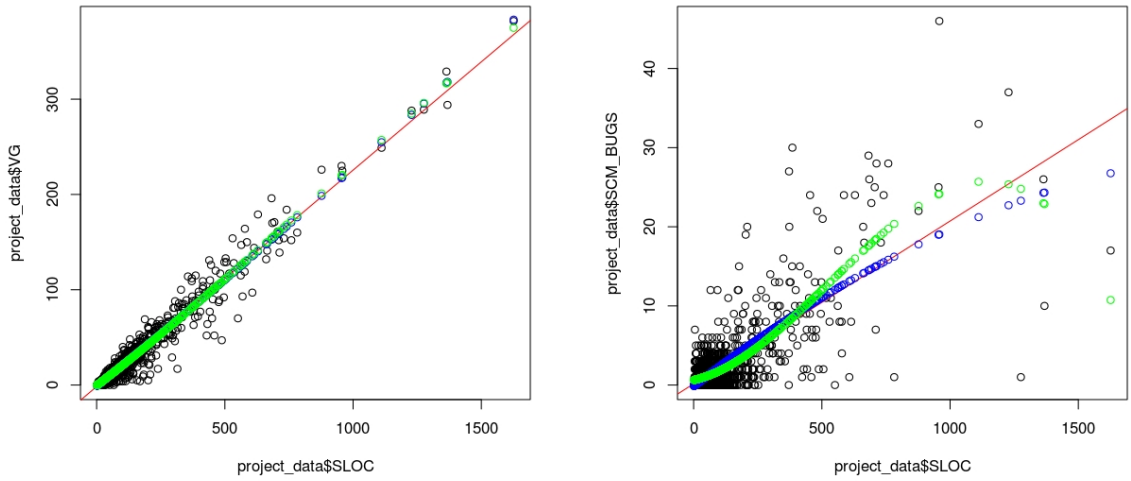
dramatically failed in the development phase. Okanović et al. [139] apply linear regression analysis to software profiling data to predict performance of large distributed systems.

Linear and polynomial regression

Linear regression tries to fit the data to a model of the form given in equation 3.1. Polynomial regression tries to fit data to a second- (equation 3.2), third- (equation 3.3) or higher level order polynomial expression.

$$y = ax + b \tag{3.1}$$
$$y = ax^2 + bx + c \tag{3.2}$$
$$y = ax^3 + bx^2 + cx + d \tag{3.3}$$

A common practice in modern statistics is to use the **maximum likelihood estimates** of the parameters as providing the "best" estimates. That is to say, given the data, and having selected a linear model, we want to find the values for a and b that make the data most likely.



(a) SLOC vs. VG

(b) SLOC vs. SCM_FIXES

Figure 3.7: Linear regression examples

An example of a linear regression analysis is easily visible when the number of source lines of code (SLOC) is plotted against the cyclomatic complexity (VG). The result shows a kind of more or less tight "exploding cloud" with a long shape, as shown in figure 3.7a. The linear regression analysis draws a red line which is closest to the overall set of points. The blue plot is the prediction of a second order regression model and the green plot is the prediction of a third order regression model. If the shape of values is too large however (i.e. variance is high), as shown in figure 3.7b, the prediction becomes inaccurate and is not suitable.

3.6 Time series

Time series are defined as vectors of numbers, typically regularly spaced in time. Yearly counts of animals, monthly means of temperature, daily prices of shares, and minute-by-minute details of blood pressure are examples of time series. Time series analysis accounts for the fact that data points taken over time may have an internal structure (such as e.g. autocorrelation, trend or seasonal variation, or lagged correlation between series) that should be accounted for. Time series analysis brings useful information on the possible cycles, structure and relationships with internal or external factors. They also allow practitioners to build prediction models for forecasting events or measures.

3.6.1 Seasonal-trend decomposition

STL (Seasonal-trend decomposition) performs seasonal decomposition of a time series into seasonal, trend and irregular components using `loess`. The trend component stands for long term trend, the seasonal component is seasonal variation, the cyclical is repeated but non-periodic fluctuations, and the residuals are irregular components, or outliers. Time series can be decomposed in R with the `stl` function from the `stats` package [147], which uses the seasonal-trend decomposition based on loess introduced by Cleveland et al in [37].

3.6.2 Time series modeling

Time series models were first introduced by Box and Jenkins in 1976 [29] and are intensively used to predict software characteristics like reliability [81, 76] and maintainability [150]. As an example, some common flavours of time series models are presented below:

Moving average (MA) models where

$$X_t = \sum_{0}^{q} \beta_j \epsilon_{t-j}$$

Autoregressive (AR) models where

$$X_t = \sum_{1}^{p} \alpha_j X_{t-i} + \epsilon_t$$

Autoregressive moving average (ARMA) models where

$$X_t = \sum_{1}^{p} \alpha_j X_{t-i} + \epsilon_t + \sum_{0}^{q} \beta_j \epsilon_{t-j}$$

A moving average of order q averages the random variation over the last q time periods. An autoregressive model of order p computes X_t as a function of the last p values of X.

So for a second-order process, we would use

$$X_t = \alpha_1 X_{t-1} + \alpha_2 X_{t-2} + \epsilon_t$$

Autocorrelation

The autocorrelation function is a useful tool for revealing the inter-relationships within a time series. It is a collection of correlations, p_k for $k = 1, 2, 3, \ldots$, where p_k is the correlation between all pairs of data points that are exactly k steps apart. The presence of autocorrelations is one indication that an autoregressive integrated moving average (ARIMA) model could fit the time series. Conversely if no autocorrelation is detected in the time series it is useless to try to build a prediction model.

Amasaki et al. [4] propose to use time series to assess the maturity of components in the testing phase as a possible replacement for current reliability growth models. By analysing fault-related data they identify trend patterns in metrics which are demonstrated to be correlated with fault-prone modules. In [89], Herraiz et al. use both linear regression models and ARIMA time-series modeling to predict size evolution of large open-source projects. They find out that the mean squared relative error of the regression models varies from 7% to 17% whereas ARIMA models' R^2 ranges from 1.5% to 4%.

Herraiz et al. [90] use ARIMA models to predict the number of changes in Eclipse components. Another study from the same authors [91] apply ARIMA modeling on three large software projects, using the Box-Jenkins method to establish the model parameters. They provide safe guidance on the application of such methods, listing threats to validity and giving sound advice on the practical approach that they use.

3.6.3 Time series clustering

T. Liao draws a comprehensive survey on time series clustering in [125]. He identifies three major approaches illustrated in figure 3.8: raw-data-based, which applies generic clustering techniques to raw data, features-based, which relies on some specific features extraction, and model-based approaches, which assume there is some kind of model or a mixture of underlying probability distributions in data. Liao also lists the similarity/dissimilarity distance used, the criteria to evaluate the performance of the analysis, and the fields of application for each type of analysis.

3.6.4 Outliers detection in time series

Gupta et al. [76] takes a comprehensive and very interesting tour on outliers detection techniques for time series data. He proposes a taxonomy (depicted in figure 3.9) to classify them, then reviews the different approaches: supervised, unsupervised, parametric, model-based, pattern-based, sliding-window. Stream series is also considered, which is useful for current-trend analysis of real-time data. Limitations, advantages and pragmatic application of methods are also discussed.

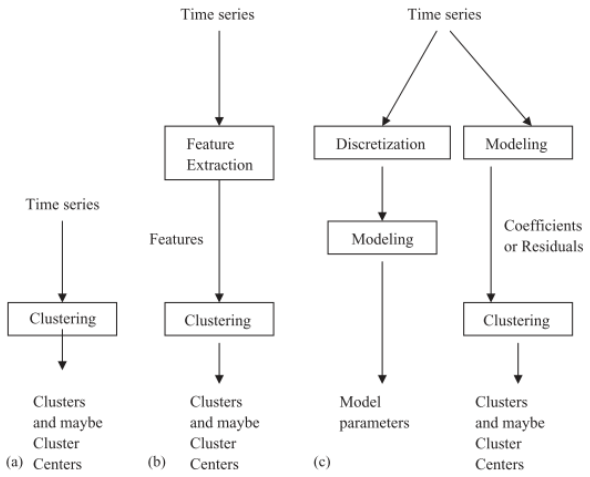


Figure 3.8: Major approaches for time series clustering [125].

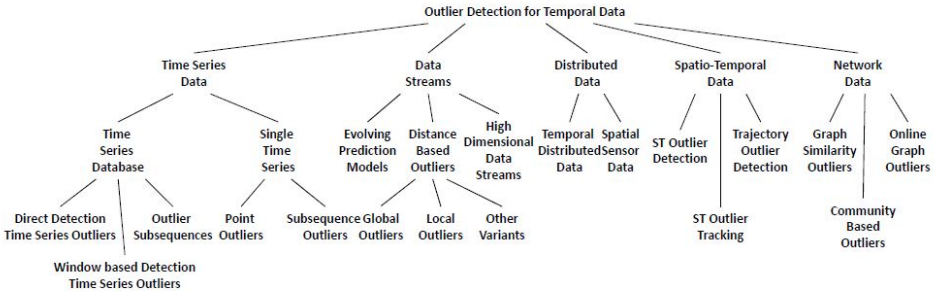


Figure 1: Organization of the Tutorial

Figure 3.9: Major approaches for outliers detection in time series [76].

Many classical outliers detection in time series techniques assume linear correlations in data, and have encountered various limitations in real applications. More robust alternatives have recently arisen through non-linear analysis, as reviewed in [74] by Gounder et al.

3.7 Distribution of measures

A distribution function gives the frequency of values in a set. Distribution functions may be discrete, when only a restricted set of values is allowed, or continuous, e.g. when values are real. Some common distribution functions are the normal and exponential distributions, depicted respectively on the left and right of figure 3.10. Distribution functions are characterised by an equation and a few parameters; as an example the normal distribution is defined by equation 3.7 and parameters mean and standard deviation.

$$P(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp^{-(x-\mu)^2/(2\sigma^2)}$$

Distribution functions essentially show what is the typical repartition of values for an

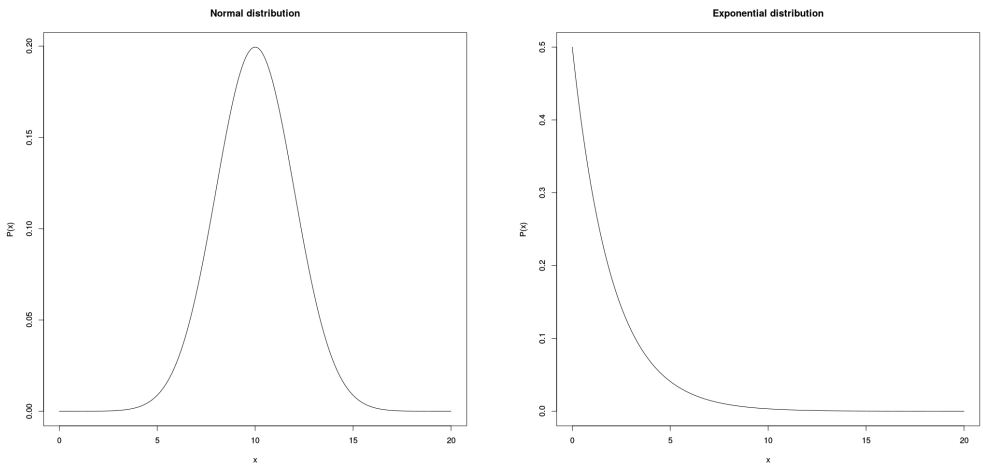


Figure 3.10: Normal and exponential distribution function examples.

attribute of data. They are usually illustrated with a histogram (see figure 4.2 on page 73 for an example), plotting the values against their occurrence frequency. The impact of distribution of software measures is twofold:

- ⇝ It allows practitioners to safely build estimation (guess at a parameter value for a known configuration) and prediction (i.e. guess at some value that is not part of the data set) models. For exploratory analysis it gives typical values for the measures and thus helps to identify outliers and erroneous data (e.g. missing values or retrieval issues).
- ⇝ Many statistical algorithms rely on the fact that the underlying distribution of data is normal. Although this assumption is correct in a vast majority of domains, one may want to know how close (or how far) reality is from this common basic hypothesis, as in the case of software engineering.

It is not easy though to establish the formal distribution of a set of measures. Techniques like qq-plots and Shapiro test allow to check data for a specific distribution, but the variety of distribution shapes available nowadays makes it difficult to try them all. Classic distributions are not always well-fitted for all metrics.

In [198], Zhang fits software faults data to Weibull and Pareto distributions and uses regression analysis to derive the parameters of the different distribution functions. The coefficient of determination is then used to assess the goodness of fit.

3.7.1 The Pareto distribution

Fenton and Ohlsson published in 2000 [64] a study that states that the number of faults in large software systems follows the Pareto principle [101], also known as the "80-20 rule": 20 percent of the code produces 80 percent of the faults. Andersson and Runseon replicated the study on different data sets in 2007 [5], globally confirming this assumption. The Pareto distribution can be defined as [40]:

$$P(x) = 1 - (\frac{\gamma}{x})^{\beta} \qquad \text{with } (\gamma > 0, \beta > 0)$$

In recent studies the Pareto principle has been further re-enforced by Desharnais et al. in [48] for the ISO/IEC 9126 quality attributes repartition, and by Goeminne for open source software activity in [72].

3.7.2 The Weibull distribution

However Zhang [198], further elaborating on Fenton and Ohlsson study on fault distribution, proposed in 2008 a better match with the Weibull distribution as shown in figure 3.11.

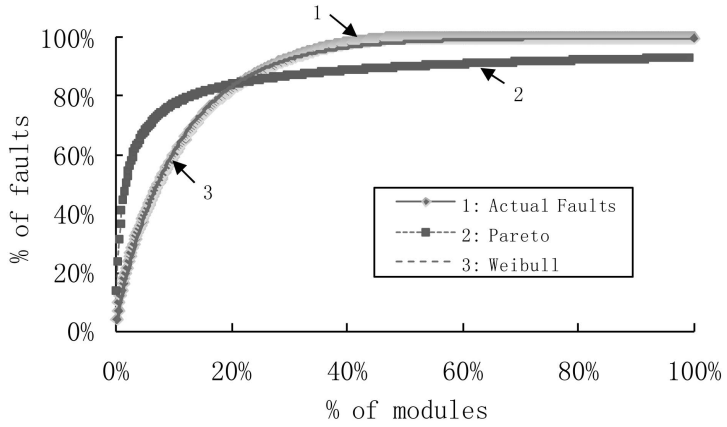


Figure 3.11: Distribution of prerelease faults in Eclipse 2.1 [198].

The Weibull distribution can be formally defined as:

$$P(x) = 1 - \exp(-(\frac{x}{\gamma})^{\beta}) \qquad \text{with } (\gamma > 0, \beta > 0)$$

Simmons also proposed in [163] and [164] the Weibull distribution for defect arrival modeling. This is especially useful in answering long-running questions like *when to stop testing?*.

3.8 Summary

In this chapter we reviewed the main data mining techniques that will be useful for the purpose of our research. We explored principal component analysis for the upcoming dimensionality reduction paper, clustering and outliers detection analysis for the SQuORE Labs, and time series analysis for software evolution. The papers listed herein will be used often throughout the forthcoming work, either as references or for further investigation.

On the one hand, data mining techniques have recently greatly improved, especially regarding their robustness and ability to handle non-normal and non-linear cases. The

domains driving this development are mainly economics, business analysis, and bioinformatics. On the other hand, these techniques have scarcely been applied to software engineering data, because even if statisticians may be occasional developers or computer fanatics, they do not necessarily know much about software engineering. Software engineers do not necessarily have a statistician background either.

Nevertheless in recent years the application of data mining techniques to software engineering data really gained wide interest both in academic research and in industrial applications. The MSR (Mining Software Repositories) conference, started in 2004, fertilises the field every year with many new articles and ideas [86]. Tao Xie has been working on this specific domain for years, building a comprehensive bibliography [189] and publishing numerous important studies.

The trends and advances observed in recent years in this area of knowledge should be carefully watched. The very specificities of software measurement data (e.g. non-normal distributions of metrics, changes in variance on the measurement's range, non-cyclic evolution of variables) need specific treatments: algorithms with an improved robustness, models and methods for non-linear and non-normal data.

Part II

The Maisqual project

> A learning experience is one of those things that says, "You know that thing you just did? Don't do that."
>
> Douglas Adams

Overview

During the second phase of the project, we try out and organise our toolshelf, and prepare the foundations for the upcoming SQuORE Labs.

Literate analysis documents developed during this phase and presented in chapter 4 provide a sandbox for investigations and trials. These are meant to be instanciated and optimised into dedicated documents for research on specific areas and concerns.

We elaborate in chapter 5 a method for data retrieval and analysis to ensure consistency and meaning of results. It is implemented and fully automated to provide a sound and practical basis for further research.

We generate in chapter 6 a set of metrics related to various projects written in C and Java, thoroughly describing the metrics and rules extracted from the project. Time ranges span from a few random extracts to extensive weekly historical records. The measures are retrieved from unusual repositories, allowing us to extract information targeted at new areas of quality like community (e.g. activity, support) and process (e.g. planning, risk, change requests management).

The output of this part of the project consists in the publication of the data sets and the description of the methodological and tooling framework designed for the SQuORE Labs detailed in part III.

Chapter 4

Foundations

In addition to building the state of the art for both data mining and software engineering, we also wished to implement some of the methods we found in the papers we read. Also, from the project organisation perspective, we wanted to run reproducible experiments to better appreciate and understand both the problem and the available ways to solve it. This application of academic and industrial knowledge and experience to real-life data and concerns is the genesis of Maisqual.

The present chapter describes the early steps of Maisqual and the process we followed to run trials and investigate various paths. It is for the most part written in chronological order to reflect the progression of work and development of ideas. The output of this time-consuming phase is a clear roadmap and a well-structured architecture for the next programmed steps.

Section 4.1 describes how we fashioned the data mining process and managed to retrieve the first round of data and run the first task. We quickly used literate data analysis to run semantic-safe investigations. Two main documents were written: one a study of a single version of a project, described in section 4.2; and another a study of a series of consecutive snapshots of a project, described in section 4.3. The experience gathered during these trials provided us with lessons expounded in chapter 4.4 that allowed us to formalise different axes summarised in a roadmap in section 4.5.

4.1 Understanding the problem

4.1.1 Where to begin?

While investigating the state of the art we wanted to apply some of the described techniques on real data to experiment with the methods described. To achieve this we decided at first to gather manually various metrics from different sources (like configuration management tools, bug trackers) for a few open-source projects and started playing with R.

We wrote shell and Perl scripts to consistently retrieve the same set of information and faced the first retrieval problems. Projects use different tools, different workflows, bugs search filters were not consistent with the code architecture, or past mailboxes

were not available. Scripts became more and more complex as local adaptations were added. We applied R scripts on the extracted data sets to print results on output and generate pictures, and integrated them in the toolchain. Since the data retrieval steps were error-prone, we wrote Perl modules for the most common operations (retrieval, data preparation and analysis) and all the configuration information needed to individually treat all projects. R chunks of code were embedded in Perl scripts using the `Statistics::R` CPAN module.

However we quickly ended up with too many scripts, and had trouble understanding afterwards the same outputs. Comments in R scripts are great for maintainability, but fail to provide a sound semantic context to the results. We needed a tool to preserve the complex semantic of the computations and the results. These Perl scripts and modules have really been a workbench for our tests: some of the R code has been moved to a dedicated package (`maisqual`) or to Knitr documents, while other parts have been integrated as Data Providers for SQuORE.

4.1.2 About literate data analysis

At this point, we found the literate analysis principle very convenient to generate reports with contextual information on how graphics are produced, what mining method is being applied with its parameters' values, and to some extent how it behaves through automatic validation of results[1]. *Literate data analysis* has been introduced by Leisch [69] as an application of Donald Knuth's principles on literate programming [112]. The primary intent was to allow reproducible research, but its benefits reach far beyond later verification of results: practitionners get an immediate understanding from the textual and graphical reports, and documents and conclusions are easier to share and to communicate.

Leisch wrote `Sweave` to dynamically insert results of R code chunks into LaTeX documents. These results can either be textual (i.e. outputs a single information, or a full table of values) or graphical through R extended graphical capabilities. Xie further enhanced this approach with Knitr [191], an R package that integrates many Sweave extensions and adds other useful features. Among them, Knitr allows to use caching for chunks of code: when the code has not changed the chunk is not executed and the stored results are retrieved, which gives remarkable optimisations on some heavy computations. Conditional execution of chunks also allows to propose textual advice depending on computational results, like Ohloh's factoids[2]. This is a very important point since it provides readers with contextually sound advice.

[1] As an example, p-values returned by regression analysis algorithms can be interpreted quite easily.

[2] www.ohloh.net lists simple facts from the project's statistics. Examples include "*mostly written in Java*", "*mature, well-established codebase*", "*increasing year-over-year development activity*", or "*extremely well-commented source code*".

4.2 Version analysis

We wrote a first Sweave document, which served as a workshop to explore techniques and as a laboratory for the papers we read. Since our primary background is software engineering, we had to experiment with the data mining algorithms to understand how they work and how they can be applied to software engineering concerns. The analysis was performed both at the file and function levels, with minor changes between them – some metrics like configuration management metadata not being available at the function level. At one point we migrated this document to Knitr [191] because of the useful features it brings (namely caching and conditional execution). We heavily relied on *The R Book* [41] from M. Crawley, and the *R Cookbook* [175] from P. Teetor to efficiently implement the different algorithms.

4.2.1 Simple summary

The simple summary shows a table with some common statistical information on the metrics, including ranges of variables, mean, variance, modes, and number of NAs (Not Available). An example is shown below for JMeter[3].

Metric	Min	Max	Mean	Var	Modes	NAs
CFT	0	282	25.5	1274.3	128	0
COMR	5.6	95.9	47.5	348.2	778	0
NCC	0	532	59.6	4905.8	200	0
SLOC	3	695	89.2	10233.4	253	0
SCM_COMMITS	0	202	13.5	321.8	43	0
SCM_COMMITTERS	0	8	2.3	1.9	9	0

These tables show the usual ranges of values and thus allows to detect faulty measures, and provides quick insights on the dispersion of values (variance). The number of modes and NAs is useful to detect missing values or errors in the retrieval process.

Scatterplots of metrics are also plotted, as shown in figure 4.1 for the JMeter extract of 2007-01-01. One can immediately identify some common shapes (e.g. all plots involving COMR have values spread on the full range with a specific silhouette) and relationships (e.g. for SLOC vs. ELOC or SLOC vs. LC). The document preamble also filters metrics that may harm the applied algorithms: columns with NAs, or metrics with a low number of modes (e.g. binary values) are displayed and discarded.

[3]For a complete description of metrics used in this document, check section 6.1 on page 106.

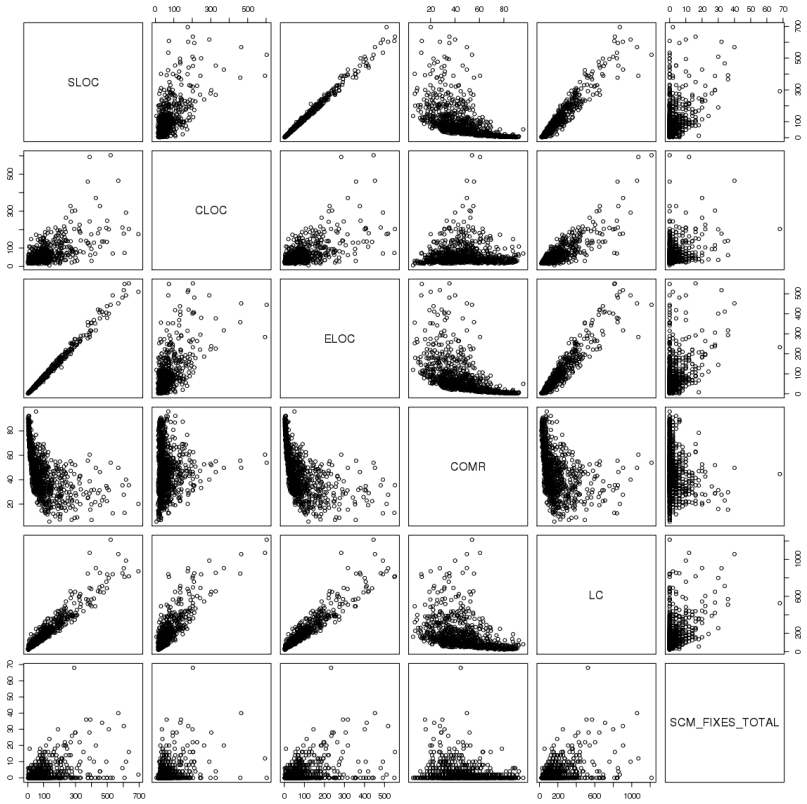


Figure 4.1: Scatterplots of common metrics for JMeter 2007-01-01.

4.2.2 Distribution of variables

Since many algorithms make the assumption of a normal distribution of values, we wanted to check data sets for normality. For this we plotted the histogram of some measures, as pictured in figure 4.2. We realised that for most of the metrics considered, the distribution was actually not Gaussian, with a few metrics as possible exceptions (e.g. COMR). This later on led to questions about the shapes of software[4].

4.2.3 Outliers detection

Outliers detection is also tested through boxplots [120] and Local Outlier Factor [30]. Boxplot is applied first to every metric individually: for each metric, we count the number of outliers and compute percent of whole set, and list the top ten outliers on a few important metrics (e.g. SCM_FIXES or VG). Multivariate outliers detection is computed on a combination of two metrics (SLOC vs. VG and SLOC vs. SCM_FIXES) and on a combination of three metrics (SLOC vs. VG vs. SCM_FIXES). Examples of scatterplots showing the outliers on these metrics are provided in figures 4.2a and 4.2b. These plots illustrate how outliers on one metric are hidden in the mass of points when plotted on

[4]A research article is in progress about shapes of software – see conclusion on page 176 for more information.

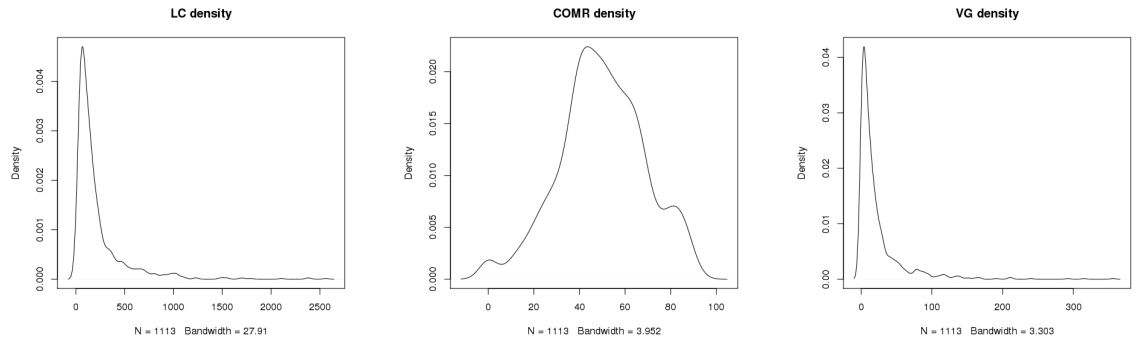
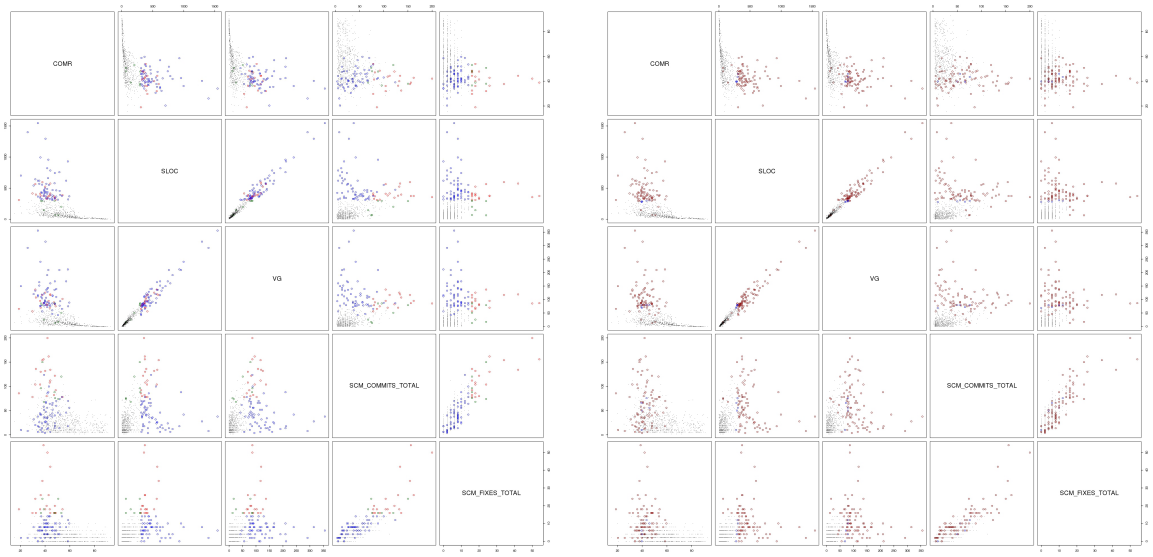


Figure 4.2: Examples of distribution of metrics for Ant 1.7.

another metric. We also tried to plot the overall union and intersection of outliers on more than 3 metrics, but it consistently gave unusable results, with respectively all and no points selected.



(a) SLOC and VG

(b) SLOC, VG and SCM_FIXES

Figure 4.2: Combinations of boxplots outliers for Ant 1.7.

These have been the early steps of the Outliers detection SQuORE Labs – cf. chapter 8 on page 137.

4.2.4 Regression analysis

We applied linear regression analysis and correlation matrices to uncover the linear relationships among measures. We used a generalised approach by applying a pairwise linear regression to all metrics and getting the adjusted R^2 for every linear regression.

To help identify correlations, only cases that have an adjusted R^2 value higher than

	BLAN	CFT	CLAS	CLOC	COMR	ELOC	FUNC	LC	SCM_COMMITS	SCM_COMMITTERS	SCM_FIXES	SLOC	STAT	VG
BLAN	1.00	0.803	0.201	0.706		0.838	0.719	0.876	0.348		0.358	0.843	0.82	0.804
CFT	0.80	**1**	0.248	0.717		**0.959**	0.702	**0.931**	0.476	0.247	0.481	**0.971**	**0.966**	**0.951**
CLAS	0.20	0.248	**1**			0.258	0.291	0.248				0.268	0.229	0.265
CLOC	0.71	0.717		**1**		0.686	0.682	0.886	0.595	0.366	0.555	0.709	0.66	0.762
COMR					**1**									
ELOC	0.84	**0.959**	0.258	0.686		**1**	0.73	**0.935**	0.439	0.211	0.456	**0.997**	**0.978**	**0.919**
FUNC	0.72	0.702	0.291	0.682		0.73	**1**	0.792	0.389		0.398	0.758	0.662	0.799
LC	0.88	**0.931**	0.248	0.886		**0.935**	0.792	**1**	0.537	0.292	0.532	**0.947**	**0.91**	**0.936**
SCM_COMMITS	0.35	0.476		0.595		0.439	0.389	0.537	**1**	0.73	0.852	0.451	0.414	0.493
SCM_COMMITTERS		0.247		0.366		0.211		0.292	0.73	**1**	0.521	0.218	0.207	0.255
SCM_FIXES	0.36	0.481		0.555		0.456	0.398	0.532	0.852	0.521	**1**	0.466	0.43	0.495
SLOC	0.84	**0.971**	0.268	0.709		**0.997**	0.758	**0.947**	0.451	0.218	0.466	**1**	**0.978**	**0.941**
STAT	0.82	**0.966**	0.229	0.66		**0.978**	0.662	**0.91**	0.414	0.207	0.43	**0.978**	**1**	**0.918**
VG	0.80	**0.951**	0.265	0.762		**0.919**	0.799	**0.936**	0.493	0.255	0.495	**0.941**	**0.918**	**1**

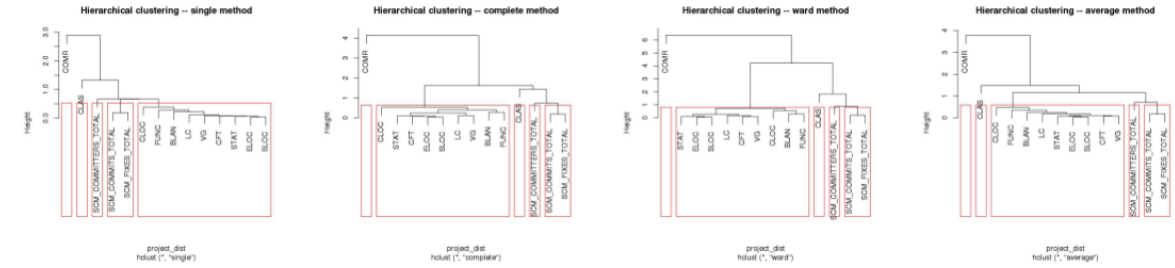


Figure 4.3: Linear regression analysis for Ant 2007-01-01 file metrics.

0.2 are displayed, and higher values (with $R^2 > 0.9$) were typeset with a bold font. An example of result for Ant 2007-01-01 is shown on figure 4.3. In the upper table one can find the afore-mentioned correlations between SLOC and ELOC (0.997) and between SLOC and LC (0.947). The hierarchical clustering dendograms at the bottom of the figure classify similar metrics (e.g. ELOC, LC and SLOC) together on the same limb of the tree.

We also computed hierarchical clusters on results to help identify metrics that have high correlation rates. Several aggregation methods were used and displayed, both for our own comprehension of the aggregation methods and to highlight metrics that are *always* close. The correlation matrix, also computed, gave similar results.

4.2.5 Principal Component Analysis

Principal component analysis is used to find relevant metrics on a data set. Many software measures are highly correlated, as it is the case, for example, for all line-counting metrics. Using a principal component analysis would allow to find the most orthogonal metrics in terms of information. The variance gives the principal components representativity.

For our trials, we computed PCA on metrics using R `princomp` from MASS [180], and kept only principal components with a variance greater than 20% of the maximum variance (i.e. variance of the first component). The restricted set of components was output in a table as shown in the following table, with values lower than 0.2 hidden to help identify components repartition. One can notice that in this example the first component is predominently composed of size-related metrics with similar loadings (around 0.3).

Metric	1st Comp.	2nd Comp.	3rd Comp.	4th Comp.
BLAN	-0.29			
CFT	-0.31			
CLAS		0.21		0.95
CLOC	-0.28		-0.30	
COMR		-0.38	-0.86	
ELOC	-0.31			
FUNC	-0.29			
LC	-0.31			
SCM_COMMITS	-0.24	-0.47		
SCM_COMMITTERS		-0.54	0.27	
SCM_FIXES	-0.24	-0.43		
SLOC	-0.31			
STAT	-0.30			
VG	-0.31			

However this gave inconsistent and unpredictable results when applied over multiple projects or multiple releases: metrics composing the first components are not always the same, and loadings are not representative and constant enough either. We decided to leave this to be continued in a later work on metrics selection[5].

4.2.6 Clustering

Various unsupervised classification methods were applied on data: hierarchical and K-means clustering from the `stats` package, and DBSCAN from the `fpc` package [87].

Hierarchical clustering was applied both on metrics (to identify similar variables) and artefacts (to identify categories of files or functions). Extensive tests were run on this, with various aggregation methods (Ward, Average, Single, Complete, McQuitty, Median, Centroid) and numbers of clusters (3, 5, 7, 10). The repartition into clusters was drawn in colourful plots. In the dendogram depicted in figure 4.4 the blue and green boxes define respectively 3 and 5 clusters into the forest of artefacts. Plots underneath illustrate the repartition of artefacts in 5 coloured clusters.

Two types of tables were output for hierarchical clustering: the number of items in each cluster, as presented in next section for k-means clustering, and the repartition of files or functions in a static number of clusters with the various aggregation methods. The latter produces tables as shown below for five clusters:

[5]A research article is in progress about metrics selection – see conclusion on page 176 for more information.

Method used	Cl1	Cl2	Cl3	Cl4	Cl5	Total
Ward	365	11	202	403	132	1113
Average	1005	7	97	1	3	1113
Single	1109	1	1	1	1	1113
Complete	1029	7	73	1	3	1113
McQuitty	915	7	187	1	3	1113
Median	308	7	794	1	3	1113
Centroid	1000	7	102	1	3	1113

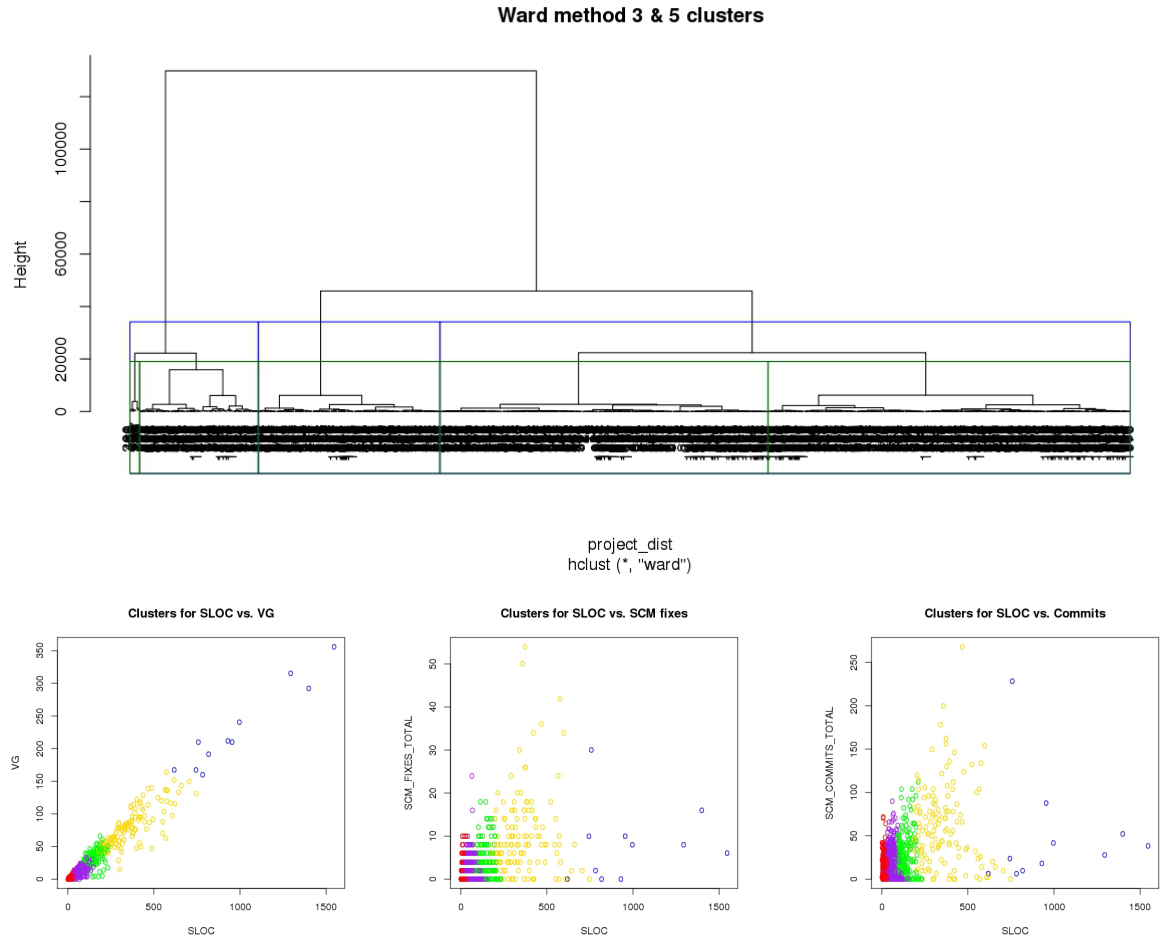


Figure 4.4: Hierarchical clustering of file metrics for Ant 1.7 – 5 clusters.

K-means clustering was applied on the whole set of metrics, with a number of clusters of 3, 5 and 7. The repartition into clusters was drawn in colourful plots as shown in figure 4.5. The number of artefacts in each cluster was output in tables like the following:

Number of clusters	Cl1	Cl2	Cl3	Cl4	Cl5	Cl6	Cl7
3	37	914	162				
5	58	317	114	613	11		
7	58	317	155	319	88	59	11

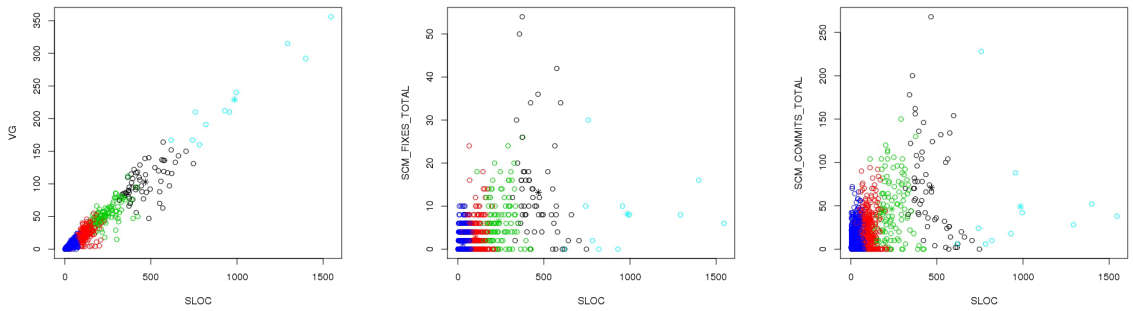


Figure 4.5: K-means clustering of file metrics for Ant 1.7 – 5 clusters.

4.2.7 Survival analysis

Survival analysis is often used to analyse mortality data, e.g. for time or risk of death of a class of individuals. In our context, survival analysis can be applied to the analysis of *failures of components*; it involves the number of failures, the timing of failures, or the risks of failure to which a class of components are exposed.

Typically, each component is followed until it encounters a bug, then the conditions and time of failure are recorded (this will be the response variable). Components that live until the end of the experiment (i.e. that had no registered bug until the time of analysis) are said to be censored. This is especially useful to study the Mean Time To Failure (MTTF) or Mean Time Between Failure (MTBF) of a component.

We tried to define a set of metrics pertinent to the time of failure of a component.

- ⇝ The number of commits before the first bug is detected in commit messages.
- ⇝ The age in days of the file before the first bug is detected in commit messages.
- ⇝ The average number of commits between two bugs-related commit messages.
- ⇝ The average number of days between two bugs-related commit messages.

However these metrics were difficult to compute reliably because of the lack of consistency in naming conventions (i.e. how to detect that a commit is related to a bug?), and we could not get consistent, predictable results.

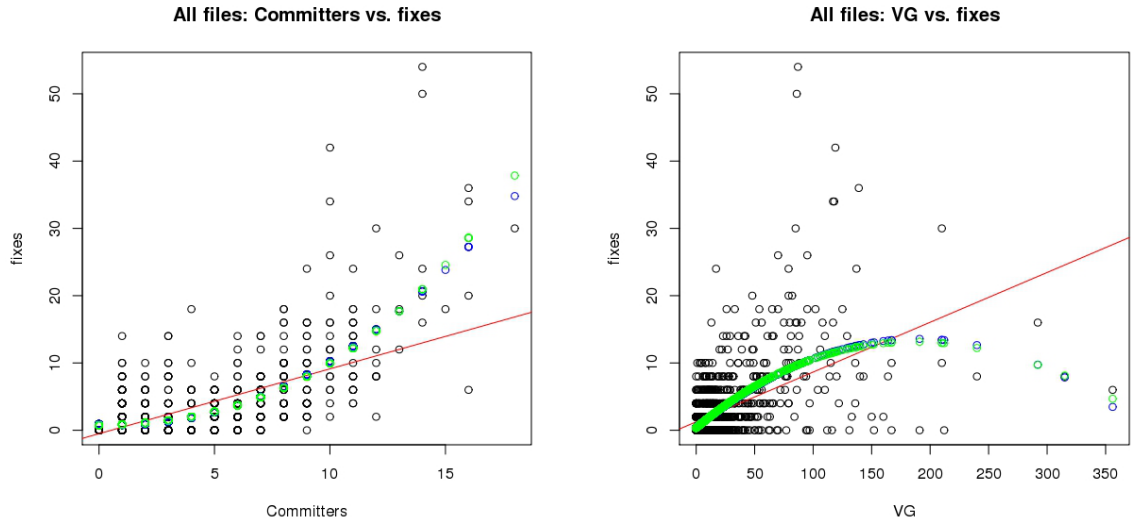
To our knowledge, there is very little research on the application of survival analysis techniques to software engineering data, although there exists some early work for power-law shaped failure-time distributions. Reed proposes in [149] a hazard-rate function adapted to power-law adjusted Weibull, gamma, log-gamma, generalised gamma, lognormal and Pareto distributions.

Survival analysis, as applied to failure-time of software components, provides interesting tools and new insights for software engineering. Although many methods assume a normal distribution, more robust algorithms have lately arisen that could be useful for software data, such as [149]. One may get better results by establishing reliable links between configuration management and bug tracking systems and mapping the circumstances of death (e.g. component impacted, type of bug, time of rework, or severity) to applications and file characteristics.
Examples of practical usage for these methods include: enhancing or replacing current reliability-growth models, which are used to estimate the number of residual bugs after testing, and analysing the impact of practices on the reliability and failure-time of components.

4.2.8 Specific concerns

The intention of this section was to formulate specific questions, but it was quickly put aside since we switched to dedicated documents to address specific concerns (e.g. for the SQuORE Labs). Only one target was actually defined for regression analysis on the SCM_FIXES metric. We applied first, second and third-order polynomial regression to explain this variable with other meaningful metrics such as SLOC, VG, SCM_COMMITS or SCM_COMMITTERS. This was intended as a verification of some urban legends that one may often hear. Figure 4.5 plots the number of fixes identified on files (SCM_FIXES) against their cyclomatic number (VG) and the number of different committers that worked on the file (SCM_COMMITTERS). The red line is a linear regression ($y = ax + b$), the blue plot is the prediction of a second order regression model ($y = ax^2 + bx + c$), and the green plot is the prediction of a third order regression model ($y = ax^3 + bx^2 + cx + d$). In the left plot curves suggest an order 2 polynomial correlation in this case (see chapter 9 for further investigations).

Some significant numbers were also computed and displayed in full sentences: the average number of fixes per 1000 Source Lines Of Code, and the average number of fixes per cyclomatic complexity unit. However, we have not been very confident in these measures for the same reasons that Kaner exposed for the MTTF in [105].



(a) SCM_FIXES vs. SCM_COMMITTERS (b) SCM_FIXES vs. VG

Figure 4.5: First, second and third order regression analysis for Ant 1.7.

4.3 Evolution analysis

The second Sweave document we wrote targeted the evolution of metrics across the successive versions of a software project. Very useful information can be extracted from temporal data, regarding the trend and evolution of project. For business intelligence or project management the latter is of primary importance.

The first version of the document included all application-, file- and function- levels analyses. This proved to be a bad idea for a number of reasons:

- ⇝ It was very long to run and error-prone. A single run lasted for 20 to 30 minutes, depending on the number of versions analysed. Furthermore, it was difficult to have all algorithms run correctly on such a huge amount of data: when an error pops up on a chunk of code, e.g. on a function-level analyse, then the whole document has to be re-executed. When it runs fine (after many trials) on a project, then chances are good that it will fail again on another set of data.
- ⇝ Considering all files and functions across successive versions introduces a bias which may considerably threaten the results. If a file is present in more than one version, then we are analysing twice or more the same artefact – even if it has differing measures. For algorithms like outliers detection or clustering, this is a blocker: if a file lies outside of typical values on a single version, it has all its siblings from other versions next to it when considering multiple versions, and is not an outlier anymore. We introduced some formulae to compute application-level summaries of each version, but it took far too much time to be usable.

So we quickly split this huge document into three smaller articles for the different levels of analysis (application, files, functions), and soon after only the application-level

document was updated and used. Many of the techniques applied here were borrowed from the version analysis Sweave document. Besides these, the following new algorithms were defined in the application-level evolution analysis documents.

4.3.1 Evolution of metrics

One of the first steps to get to know a variable is to plot its evolution on the full time range. Data is converted to time series with the `xts` R package [155], and then plotted and treated with the package's facilities. Examples of metrics evolution for the Ant data set[6] are displayed in figure 4.6.

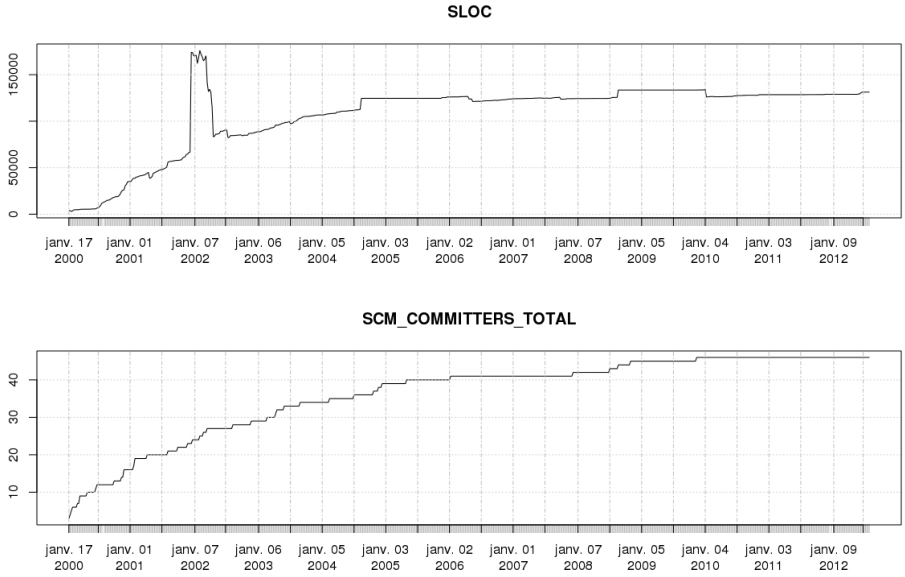


Figure 4.6: Metrics evolution for Ant: SLOC and SCM_COMMITTERS.

4.3.2 Autocorrelation

Autocorrelation in time series analysis describes how this week's values (since we are working on weekly extracts) are correlated to last week's values – this is the autocorrelation at lag 1. Partial autocorrelation describes the relationship between this week's values and the values at lag t once we have controlled for the correlations between all of the successive weeks between this week and week t. Autocorrelation can be applied on a single variable, as shown in figure 4.7 for the VG, SLOC and SCM_FIXES for Ant 1.7. The trends show that values at time t highly depend on values at time $t-1$, midly depends on values at time $t-2$, etc. In other words, the impact of past values on recent values decreases with time.

[6] See section 6.3.1 for a description of this data set.

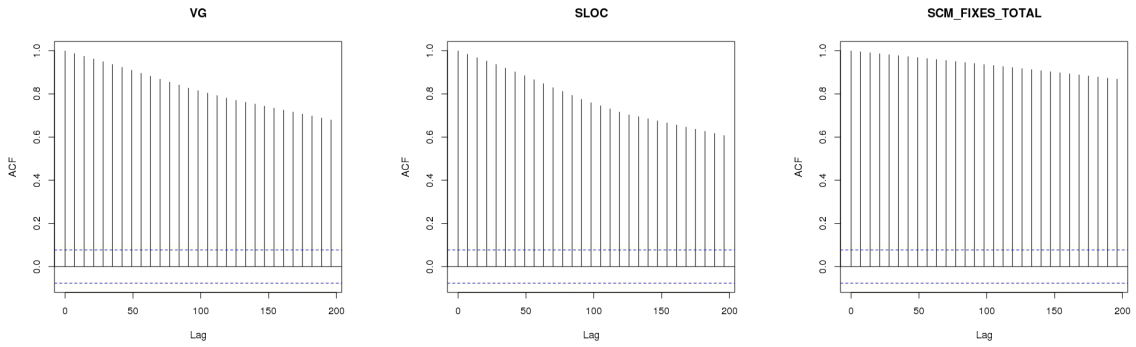


Figure 4.7: Univariate time-series autocorrelation on VG, SLOC and SCM_FIXES for Ant.

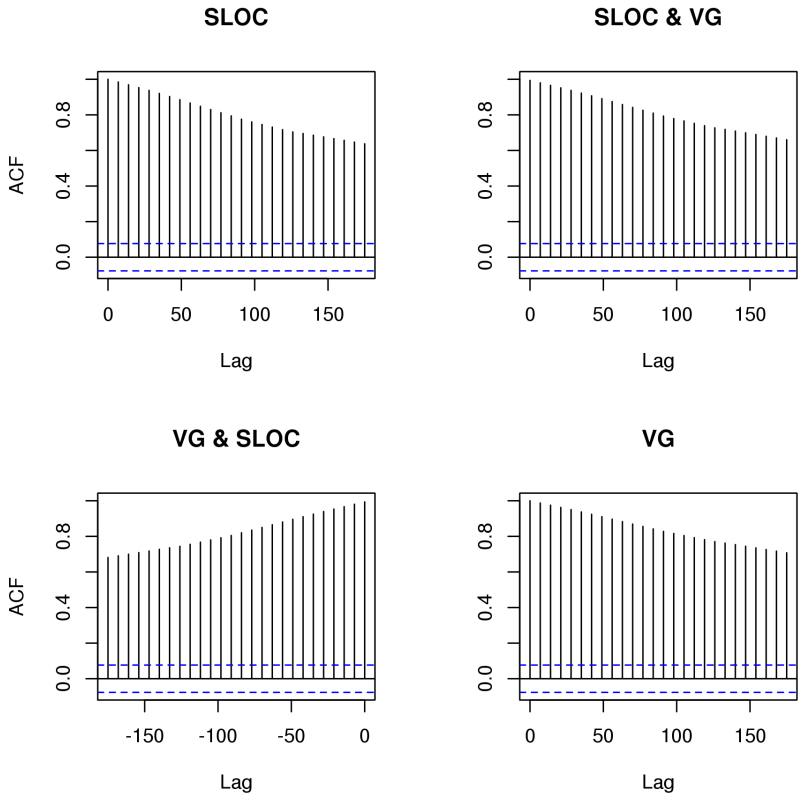


Figure 4.8: Multivariate time-series autocorrelation on VG, SLOC for Ant.

The autocorrelation can be computed on different metrics, to check if the evolution of some metrics depend on the values of others. The scatterplot displayed on figure 4.8 shows how the evolution of the SLOC and VG metrics are inter-dependent.

4.3.3 Moving average & loess

A simple way of seeing patterns in time series data is to plot the moving average. In our study we use the three-points and five-points moving average:

$$y'_i = \frac{y_{i-1} + y_i + y_{i+1}}{3}$$

$$y''_i = \frac{y_{i-2} + y_{i-1} + y_i + y_{i+1} + y_{i+2}}{5}$$

It should be noted that a moving average can never capture the maxima or minima of a series. Furthermore, it is highly dependent on the variable variations and is usually really close to the actual data.

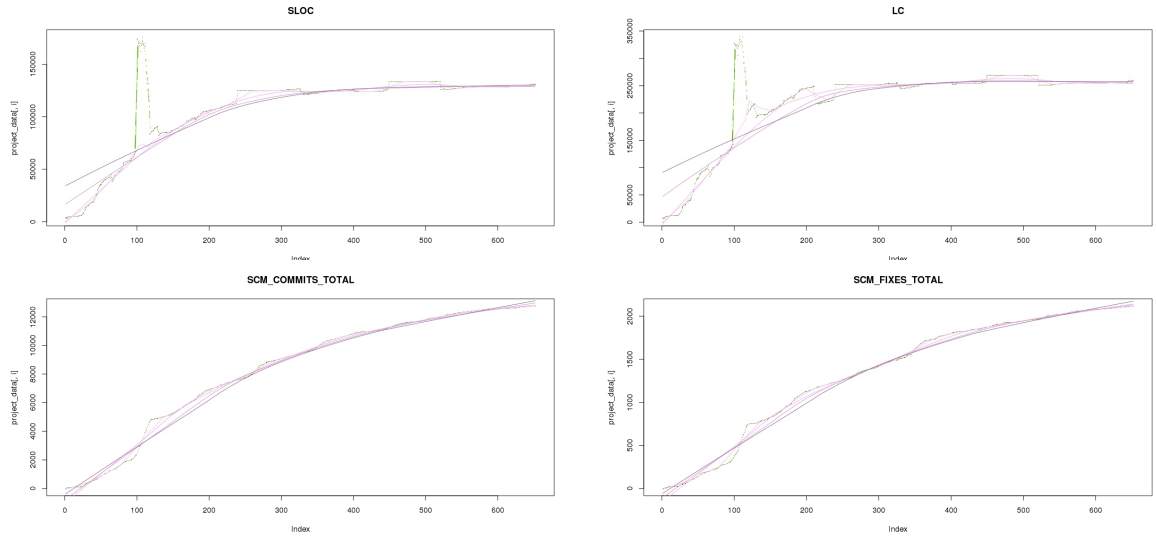


Figure 4.9: Moving average and loess on SLOC, LC, SCM_COMMITTERS and SCM_FIXES for Ant 1.7.

`loess` is an iterative smoothing algorithm, taking as parameter the proportion of points in the plot which influence the smooth. It is defined by a complex algorithm (Ratfor by W. S. Cleveland); normally a local linear polynomial fit is used, but under some circumstances a local constant fit can be used. In figure 4.9 the green plots represent the 3-points moving average (light green, chartreuse3) and 5-points moving average (dark green, chartreuse4). Loess curve is plotted in purple, with from light to dark purple smoother spans of 0.1, 0.3, 0.5 and 2/3 (the default setting).

4.3.4 Time series decomposition

Time series were first decomposed with robust regression using the R function `stl` from the `xts` package [155]. STL (Seasonal-trend decomposition) performs seasonal decomposition of a time series into seasonal, trend and irregular components using `loess`. The trend component stands for long term trend, the seasonal component is seasonal variation, the cyclical is repeated but non-periodic fluctuations, and the residuals are irregular component, or outliers.

We quickly abandonned STL decomposition, for the data we had at that time was not periodic and the algorithm constantly failed to find seasonal and cyclical terms. For projects that use an iterative development process however, one should be able to decompose many measures at the iteration cycles.

One of the issues we had with time series was that analysed projects did not show cycles that would allow usage of STL decomposition and thus predict future behaviour and propose typcial templates for metrics evolution. To further investigate this, we need to apply our Knitr documents to agile projects (as can be found e.g. on github) and study the evolution of metrics across multiple iterations.

4.3.5 Time series forecasting

This section applied regression models on the temporal evolution of metrics. We give an example of a run on the Ant evolution data set hereafter, with some explanations on the optimisation methods we use.

Table 4.1: ARMA components optimisation for Ant evolution.

Order	AR MA=0	AIC	MA AR=0	AIC	MA AR+	AIC	Diff	AIC
1	1.00	168.61	1.00	4062.91	1.00	170.73	1.00	89.42
2	2.00	160.54	2.00	3283.74	2.00	173.91	2.00	71.91
3	3.00	158.16	3.00	2722.46	3.00	188.52	3.00	126.06
4	4.00	157.53	4.00	2225.62	4.00	160.53		
5	5.00	158.21	5.00	1917.32	5.00	158.97		
6	6.00	159.90	6.00	1639.14	6.00	130.65		

Three time series models were experimented on data: MA (moving average), AR (autoregressive), and ARMA (autoregressive moving average), borrowed from [29]. We try to find the best parameters for the ARMA model: the order of the autoregressive operators, the number of differences, and the order of moving average operators. We look

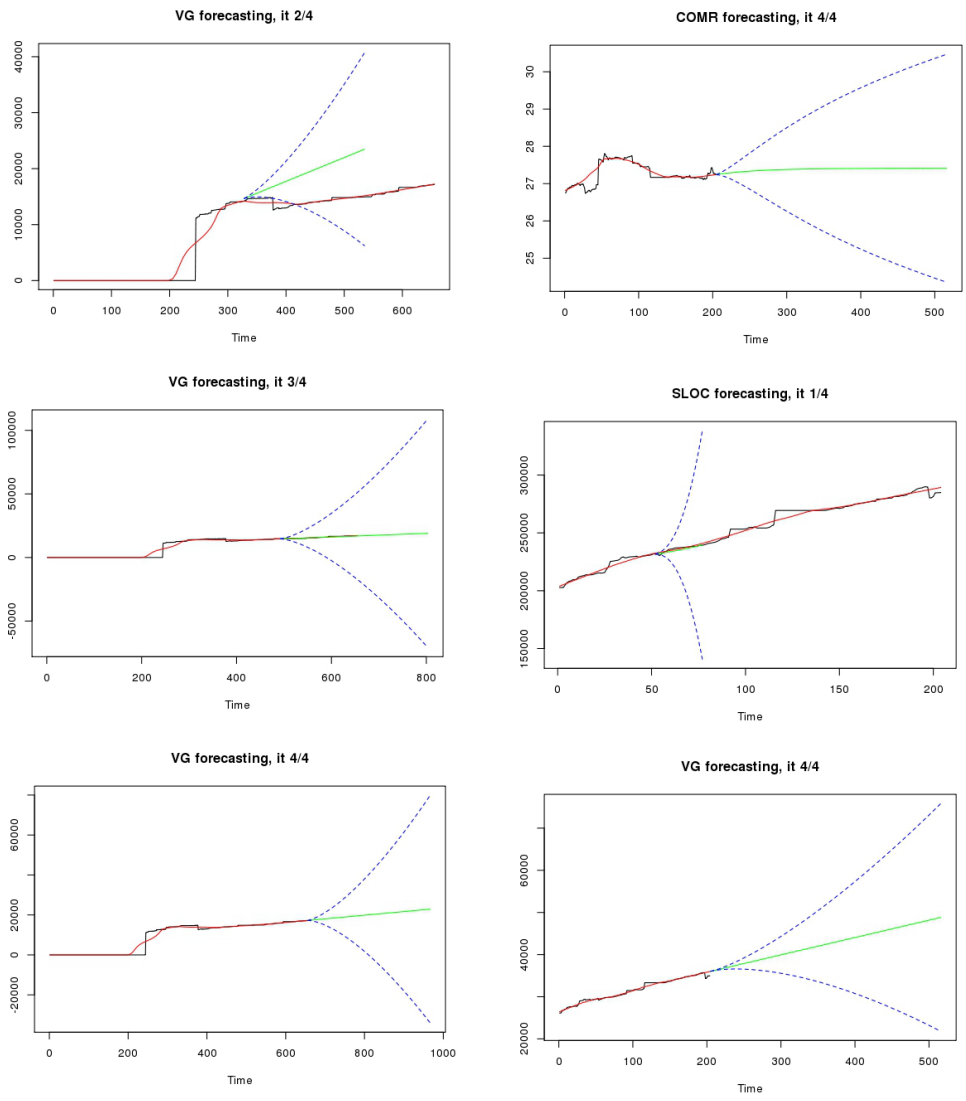


Figure 4.10: Time series forecasting for Subversion and JMeter.

at the autoregressive operator first: the `arima` function from the `stats` [147] package is invoked with 6 different orders, while MA and differences are null, and we get the minimum AIC. The same technique is applied to the moving average operators, zero-ing AR and differences. For differences, the optimised values for the AR and MA components are used and the same iterative process is executed to detect the lowest AIC. Results of ARMA parameters optimisation for Ant files are displayed in table 4.1.

In our example, the best order for the autoregressive process (AR lag) (without moving average component) is 4, the best order for moving average (without autoregressive component) is 6 and the optimal number of differences is 2. These settings are used to forecast metrics at given times of the project history: first, second and third quarters, and at the last time in history (which is 2012-07-30 in our case).

In practice this never worked well on the data we were working on: confidence intervals

were far too wide to be usable, as shown in figure 4.10.

4.3.6 Conditional execution & factoids

As presented earlier in this chapter, Knitr offers an interesting feature in the conditional execution of chunks of code. It allows to disable fragile algorithms which would fail in some cases (e.g. when a distance matrix cannot be computed), and thus considerably enhances the reliability of documents. We used it frequently in our documents to restrict the analysis of class related information to object-oriented code.

LaTeX code also defines `if` statements to conditionally display text. Under some assumptions, one can retrieve the value of variables from the R chunks into the LaTeX formatting structure and use it to print custom sentences with dynamically generated numbers.

We used it to mimic Ohloh factoids as a proof of concept, but this could be extended to provide advice based on the execution of complex algorithms – a few examples being [194, 195, 88, 107]. The following two examples in R and LaTeX show how the project size Ohloh factoid was implemented in the evolution analysis document.

```
if (project_length < 1) {
    # From 0 to 1 year => New
    MFACTTIMEA <- project_length
    MFACTTIMEB <- FALSE
    MFACTTIMEC <- FALSE
    MFACTTIMED <- FALSE
}
```

```
\ifMFACTTIMEA
\item The project is all new. It has been created less than one year ago.
\fi
\ifMFACTTIMEB
\item The project is young: it has between one and three years of development.
\fi
```

Factoids should be further investigated because they offer significant opportunities for custom advice in reports based on high-complexity analysis. They are similar to the *Action Items* provided by SQuORE (see chapter 8 for more information) but greatly extend the scope of the methods available to detect them. Recommender systems [83, 83] may also be included for better advice based on the experience gathered.

4.4 Primary lessons

4.4.1 Data quality and mining algorithms

Since we had very little knowledge of the mining characteristics of software engineering data at that time, we had to try several methods before finding one that really did what was intended. Because of the distribution function of many software metrics and the very specificities of software measurement data (e.g. correlations amongst metrics) some algorithms give bad, unusable results or simply do not work.

The first example is about missing values. In many cases a single NA in the whole matrix causes algorithms to fail. They have to be identified first and then either replaced with some default values (e.g. zeros, depending on the metric) or have the whole column removed. Some algorithms, like matrix distances or clustering, need a minimum variance on data to be safely applied – and metrics that only have a few different values make them constantly fail. Data has to be checked, cleaned, and repaired or discarded if needed.

Another example of a difficult method to apply was the time series decomposition of evolution metrics. We tried to find metrics that would show cycles or repeated patterns (possibly with lag), to normalise or more generally prepare data, or try different algorithms. We also had a hard time using ARIMA models for prediction: they often failed (i.e. stopped the R script execution) when the data was too erratic to conclude on parameters. Even when it worked (i.e. passed on to the next command) results were not convincing, with very low p-values and bad curves on prediction graphs.

We encountered some issues with configuration management metrics. Firstly, Configuration management purpose is to remember *everything*, including human or technical errors. One example is when someone accidentally copies a whole hierarchy and does not notice it before making the commit. Even if this is corrected shortly after, say in next commit, it will still appear as a huge peak in statistics. Secondly SCM operations, like merges, can introduce big bang changes in the repository contents, potentially impacting almost every other metric. Sources of such actions may be error (typos, thoughtlessness...), poor knowledge or bad practices, or may even be intended and correct.

One answer to this problem is to always check for consistency and relevance of results, with confidence intervals and statistical tests. From there on, we used p-values for our regressions and prediction models.

4.4.2 Volume of data

We worked with a huge amount of data, and this had a deep impact on the way we conducted our analyses. As an example the GCC project is one of the biggest projects that we analysed. It spans over several years, and gathering weekly versions of its sources took a huge amount of time. A single version of its sources (Subversion extract at 2007-08-27) is 473MB, and contains roughly 59 000 files. The code analysis of this version takes nearly one and a half hours to complete, data providers (Checkstyle, PMD) take 40 minutes,

and the Sweave analysis takes another 30 minutes. Running the analysis on the full GCC weekly data set takes fifteen days.

Table 4.2 lists some of the projects gathered for our analyses. The full hierarchy of source files takes approximately 250GB of disk space.

Table 4.2: Number of files and size on disk for some projects.

Project	Files	Size on disk
Ant releases	37 336	400MB
Ant weekly	1 421 063	14.8GB
Papyrus weekly	2 896 579	44GB
JMeter releases	44 096	600MB
JMeter weekly	930 706	14.1GB
GCC releases	1 939 018	18GB
GCC weekly	27 363 196	88GB
Subversion releases	140 554	3.2GB
Subversion weekly	407 167	8.8GB

The computing power and resources used conducted how we worked. For long-running tasks spanning over days or weeks (e.g. weekly extracts and analyses), if a single error occurs during the process (e.g. bad XML entities in PMD or Checkstyle files) then the full extract may be corrupted, and thrown away. Projects may also put requests or bandwidth limits to preserve their resources, especially for huge lists of bugs or data-intensive needs; scripts sometimes have to introduce random delays to not meet timeouts.

One has to setup checkpoints that allow to restart the process (either for extraction or analysis) where it failed in order to not have to start it all over again. Another good practice is to consistently check the data from the beginning to the end of the process, as subsequently explained.

4.4.3 About scientific software

Scientific software, as every software product, is subject to bugs, faults and evolutions. This has been pointed out by Les Hatton [84] and D. Kelly and R. Sanders [108], who argue that most scientific results relying on computational methods have some bias. These may either be introduced by non-conformance to explicit requirements (i.e. bugs in code) or evolving implicit requirements.

The Maisqual case is no exception: the products we rely on (SQuORE, R, Checkstyle or PMD) have their own identified bugs – or even worse, non-identified faults. On a time range of three years this may have a significant impact on results. We list below a few examples on the different products we use:

- ⇝ SQuORE had a few changes in its requirements: one in 2012 relating to Java inner types detection, and another in 2013 regarding files and functions headers. Should

anonymous classes be considered as self-standing classes? Although they cannot compare to typical classes as for the number of public attributes or complexity, they still are class structures as defined by the Java language.

⇝ Checkstyle has about 60 bugs registered on the 5.0 version[7] and PMD has 14 registered issues on the 5.0.5 version[8]. Issues span from false positives to violent crashes. The versions for R (3.0.1) and its packages also have a number of issues: 111 registered issues as of 2013 on R-core[9], 9 registered bugs on Knitr 1.6[10].

Another source of bugs comes from the integration between tools: if the output format of the tool evolves across versions, some constructions may not be seen by the next tool in the process. We also have to do our *mea culpa*: the scripts we wrote have been found guilty as well and corrected during time.

As an answer to these concerns, we decided to value the consistency of results over exact conformance to a subjective truth: if the SLOC as computed by SQuORE differs from some essays in literature we simply stick to our definition, providing all the information needed for others to reproduce results (through clear requirements and examples) and trying to be consistent in our own process. When we encountered a bug that changed output, we had to re-run the process to ensure consistency and correctness of all data sets.

4.4.4 Check data

Retrieved data should be checked at every step of the mining process for consistency and completeness. Every project has its own local traditions, customs and tools, and this variety makes automation mandatory but very fragile. Sanity checks must be done from time to time at important steps of the retrieval and analysis processes – there is far too much information to be entirely checked at once. Examples of problems that may arise, and those we met during our experiences, include:

⇝ When the project is not very active (e.g. typically at the beginning of the project), one may ask for a day which has no revision attached to it. In this case, Subversion simply failed to retrieve the content of the repository for that day, and we had to manually extract code at the previous revision.

⇝ When a data provider tool fails (e.g. an exception occurs because of some strange character or an unusal code construction) then all findings delivered by this tool are unavailable for the specific version. This may be a *silent* error, and the only way to identify it is to check for semantic consistency of data (say, 0 instead of 8 on some columns).

Some errors can be detected via automatic treatments (e.g. sum up all metrics delivered by a tool and check it is not null) while others *must* be manually checked to be uncovered.

[7] http://sourceforge.net/p/checkstyle/bugs/milestone/release_5.0/.

[8] http://sourceforge.net/p/pmd/bugs/milestone/PMD-5.0.5/.

[9] https://bugs.r-project.org/bugzilla3/buglist.cgi?version=R%203.0.1.

[10] https://github.com/yihui/knitr/issues.

It is not always easy to check for the validity of measures, however: double-checking the total number of possible execution paths in a huge file may be impossible to reproduce exactly. Unable to formally verify the information, one should still validate that the data at least *looks* ok.

Publishing the data sets is an opportunity to get feedback from other people: some errors may be easily caught (e.g. a failing data provider results in a null column in the data set) while other errors may look fine until someone with a specific knowledge or need can highlight them. In other words, many eyes will see more potential problems, as spotted by E. Raymond in [148] for open-source projects.

> “ Given a large enough beta-tester and co-developer base, almost every problem will be characterized quickly and the fix will be obvious to someone. ”
>
> Erick S. Raymond

4.5 Summary & Roadmap

This preliminary step was useful in laying down solid foundations for the upcoming steps of the Maisqual project. It allowed us to better know the data we were going to analyse, what algorithms could be applied to it, and moreover how to treat this information. From there we designed the roadmap depicted in figure 4.11, and the following clear objectives were decided:

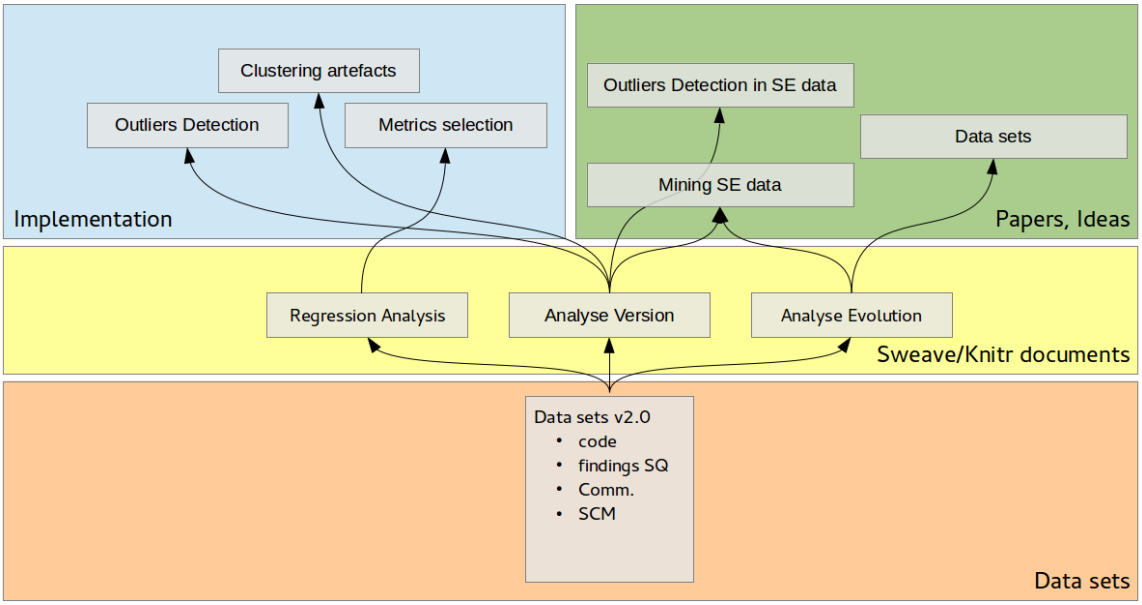


Figure 4.11: Synopsis of the Maisqual roadmap.

Produce data sets to allow reproducible research and maintain a secure consistency between versions of analysis document. Readily available data sets also provide en easy means to quickly test new techniques without having to re-check data for consistency or completeness. The introduction of new data types (e.g. from configuration management tools) permits us to easily investigate new areas for the quality models provided in the SQuORE configuration.

Write rough Knitr documents to select and test the statistical methods that can be used on software engineering data and generic concerns, and more generally try a lot of ideas without much difficulty. This was achieved through the project analysis version and evolution working documents, which provided useful insights on the applicability of techniques and served as a basis for future work.

Write optimised Knitr documents targeted at specific techniques and concerns, like outliers detection or automatic scales for SQuORE. These usually start as simple extracts of the generic document, which are then enhanced. They run faster (since all unnecessary elements have been removed from them), are more reliable (issues not related to the very concerns of the document are wiped out) and are more focused and user-friendly, with a helpful and textual description of the context and goals of the document.

Implement the solution to make it available to users: a genius idea is worthless if no one uses it. It should either be integrated into SQuORE or made available in order to benefit from it.

Chapter 5

First stones: building the project

After learning the basic concepts and issues of software engineering data mining, we drew a plan to fulfill our goals. Clear requirements were established in a new roadmap shown in section 4.5. This chapter describes how the Maisqual project was modified after these experiences and how the precepts uncovered in the previous chapter were implemented in Maisqual.

Here we describe how we investigated the nature of software engineering data and established a typical topology of artefacts in section 5.1. The methodological approach we decided to apply is explained in section 5.2. Section 5.3 describes how the process was designed, implemented and automated on a dedicated server and executed to generate the data sets and analyses.

5.1 Topology of a software project

5.1.1 The big picture

Repositories hold all of the assets and information available for a software project, from code to review reports and discussions. In the mining process it is necessary to find a common basis of mandatory repositories and measures for all projects to be analysed: e.g. configuration management, change management (bugs and change requests), communication (e.g. forums or mailing lists) and publication (website, wiki). Figure 5.1 shows repositories that can be typically defined for a software project and examples of metrics that can be retrieved from them.

It is useful to retrieve measures at different time frames (e.g. yesterday, last week, last month, last three months) to better grasp the dynamics of the information. Evolution is an important aspect of the analysis since it allows users to understand the link between measures, practices and quality attributes. It also makes them realise how much better or worse they do with time.

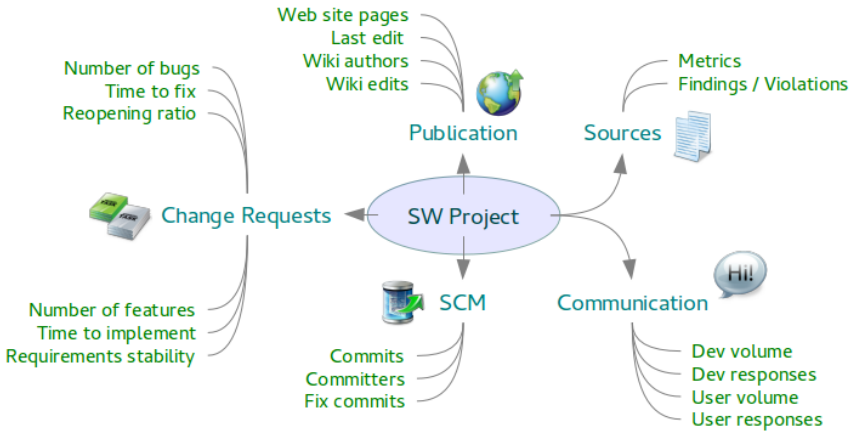


Figure 5.1: Software repositories and examples of metrics extracted.

5.1.2 Artefact Types

Artefacts are all items than can be retrieved from a software repository or source of data and has some information associated with it. In the context of software development a variety of different artefacts may be defined depending on the purpose of the search activity. Most evident examples of types are related to the product (source code, documentation), communication, or process.

At the product level, data attributes are usually software metrics and findings, attached to artefact types of the following:

Application An application is the set of files considered as a release. It is useful in a portfolio perspective to compare projects, and to track the evolution of some characteristics of the project over time. It can be built up from pure application-level metrics (e.g. the number of commits) or from the aggregation of lower-level measures[1].

Folder A folder is associated to a directory on the file system. For the Java language, it is also equivalent to a package. Folders are useful for mapping and highlighting *parts* of the software design that have common or remarkable characteristics: e.g. components or plugins.

File A file belongs to the repository source hierarchy and may contain declarations, classes, functions, instructions, or even comments only. Most analysers only consider files with an extension matching the language under analysis.

Class A class can either be the main class from the file, or an anonymous type – see section 4.4.3 for the latter. In object-oriented architectures code reusability is mostly

[1]When building up an attribute from lower-level measures, one has to pay attention to ecological inference, see section 2.2.1.

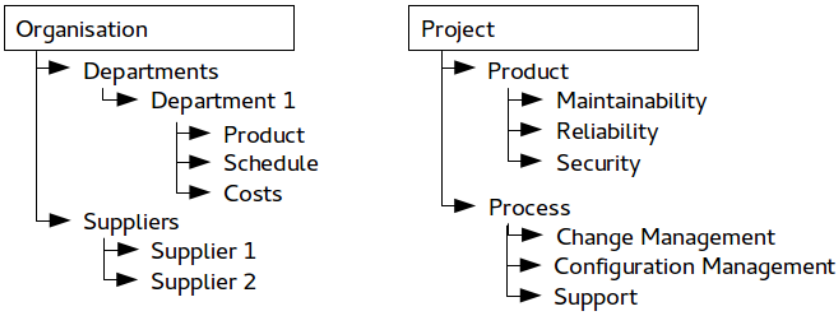


Figure 5.2: Different artefacts models for different uses.

achieved at the class level, so quality characteristics that involve reusability will often use a class-level index.

Function – or methods in the case of a class – are most useful to developpers to identify where findings are located (most violation detection tools provide the line as well).

In the data sets we generated the artefact types defined are application, file and function. A collection of information is attached to each artefact, corresponding to the file columns and extracted from different sources (static code analysis, tools metadata, mailing lists).

But as it is in the case for a measurement program, goals are the very drivers for the selection of an artefact. If one considers the activity of a repository, then considering commits as artefacts is probably the best guess. The architecture used to model the system under measurement has a great impact on the artefacts selected as well: the decomposition will differ depending on the quality characteristics one wants to measure and manage, as figure 5.2 illustrates it.

At the process level, many other types of artefacts can be defined. Most of them are attached to an application, since people and practices are usually considered at the organisation or project level. We provide below a few examples of possible artefact types.

Bugs contain a wealth of diverse information, including insights on product reliability and project activity, among others. Tools can often be queried through a documented API. Some examples of attributes that can be retrieved from there are the total volume of issues, ratio of issues rejected, the time for a bug to be taken in consideration or to be resolved.

Requirements are sometimes managed as features within the change request tracker. Agile projects often use it to define the contents of the sprint (iteration). Examples of information that can be identified from there are: the scope of release, progress of work for the next release, or requirements stability.

Web site The project's official web site is the main communication means towards the public: it may propose a description of the project, its goals and features, or means to join the project. These can be retrieved as boolean measures for the assessment

of the project's communication management. It also holds information about the project's organisation and roadmap: plannified release dates[2], frequency of news postings or edits.

Wiki A wiki is a user-edited place to share information about the project and tells a lot about community activity. Examples of attributes that can be retrieved from it are the number of different editors, daily edits, or stability (ratio of number of edited pages by total number of pages).

People can be considered as artefacts in the case of social science investigations. But this may be considered bad practice if the measure is liable to be used as a performance indicator. One has to pay attention to the Hawthorn effect (see section 2.2.1) as well when involving people in the measurement process.

5.1.3 Source code

Source code is generally extracted from the configuration management repository at a specific date and for a specific version (or branch) of the product. Source releases as published by open source projects can be used as well but may heavily differ from direct extracts, because they represent a subset of the full project repository and may have been modified during the release process.

Sources files usually includes source files, and tests. Some practitionners advocate that tests should not be included in analysis since they are not a part of the delivered product. We argue this is often due to management and productivity considerations to artificially reduce technical debt or maintenance costs; our belief is that test code has to be maintained as well and therefore needs quality assessment.

Source code analysis [128] tools usually provide *metrics* and *findings*. *Metrics* target intrinsic characteristics of the code, like its size or the number of nested loops. *Findings* are occurrences of non-conformities to some standard rules or good practices.

Static analysis tools like SQuORE [11], Checkstyle [32] or PMD [51, 183] check for anti-patterns or violations of conventions, and thus provide valuable information on acquired development practices [183]. Dynamic analysis tools like FindBugs [8, 183, 181] or Polyspace give more information on the product performance and behaviour [57, 170], but need compiled, executable products – which is difficult to achieve automatically on a large scale. Dynamic and static analysis are complementary techniques on completeness, scope, and precision [17].

5.1.4 Configuration management

A configuration management system allows to record and restitute all changes on versioned artefacts, with some useful information about who did it, when, and (to some extent) why. It brings useful information on artefacts' successive modifications and on the activity and diversity of actors in the project.

[2]When it comes to time-related information, *delays* are easier to use as measures than dates.

The primary source of data is the verbose log of the repository branch or trunk, as provided in various formats by all configuration management tools. The interpretation of metrics depends considerably on the configuration management tool in use and its associated workflow: as an example, commits on a centralised subversion repository do not have the same meaning as on a Git distributed repository because of the branching system and philosophy. A Subversion repository with hundreds of branches is probably the sign of a bad usage of the branching system, while it can be considered normal with Git. In such a context it may be useful to setup some scales to adapt ranges and compare tools together. The positioning of the analysis in the configuration management branches also influences its meaning: working on maintenance-only branches (i.e. with many bug fixes and few new features) or on the trunk (next release, with mainly new features and potentially large refactoring or big-bang changes) does not yield the same results.

Examples of metrics that can be gathered from there are the number of commits, committers or fix-related commits on a given period of time.

5.1.5 Change management

A tracker, or bug tracking system, allows to record any type of items with a defined set of associated attributes. They are typically used to track bugs and enhancements requests, but may as well be used for requirements or support requests. The comments posted on issues offer a plethora of useful information regarding the bug itself and people's behaviour; some studies even treat them as a communication channel.

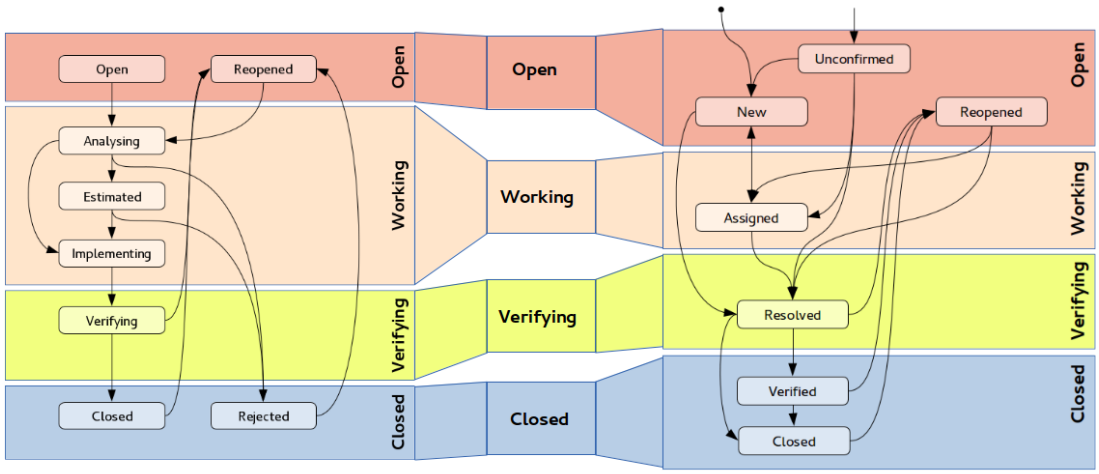


Figure 5.3: Mapping of bug tracking processes.

A bias may be introduced by different tools and workflows, or even different interpretations of the same status. Hemmati et al. [86] warns that due to the variety of different processes, only closed and fixed issues may provide safe measures. A common work-around

is to map actual states to a minimalist set of useful canonical states: e.g. open, working, verifying, closed. Almost all life cycles can be mapped to these simple steps without twisting them too much. Figure 5.3 shows a few examples of mapping to real-world bug tracking workflows: the one used at SQuORING Technologies and the default lifecycle proposed by the Eclipse foundation for the projects.

Examples of metrics that can be retrieved from change management systems include the time to closure, number of comments, or votes, depending on the system's features.

5.1.6 Communication

Actors of a software project need to communicate to coordinate efforts, ensure some consistency in coding practices, or simply get help [148]. This communication may flow through different channels: mailing lists, forums, or news servers. Every project is supposed to have at least two communication media, one for contributors to exchange on the product development (the developers mailing list) and another one for the product usage (the users mailing list).

Local customs may impact the analysis of communication channels. In some cases low-activity projects send commits or bugs notification to the developer mailing list as a convenience to follow repository activity, or contributors may use different email addresses or logins when posting – which makes it difficult, if not impossible, to link their communications to activity in other repositories. The different communication media can all be parsed to retrieve a common set of metrics like the number of threads, mails, authors or response ratio, and time data like the median time to first response.

Another way to get historical information about project communications lies in mailing lists archives, which are public web sites that archive and index for full-search all messages for mailing lists of open-source projects. We list hereafter a few of them:

- ⇝ Gmane: `http://gmane.org`
- ⇝ Nabble: `http://nabble.org`
- ⇝ The Mail Archive: `http://www.mail-archive.com`
- ⇝ MarkMail: `http://markmail.org`

5.1.7 Publication

A project has to make the final product available to its users along with a number of artefacts such as documentation, FAQ or How-to's, or project-related information like team members, history of the project or advancement of the next release. A user-edited wiki is a more sophisticated communication channel often used by open source projects. The analysis of these repositories may be difficult because of the variety of publishing engines available: some of them display the time of last modification and who did it, while other projects use static pages with almost no associated meta data. As an example user-edited wikis are quite easy to parse because they display a whole bouquet of information

about their history, but may be very time-consuming to parse because of their intrinsic changeability.

Metrics that can be gathered from these repositories are commonly linked to the product documentation and community wealth characteristics. Examples include the number of pages, recent modifications or entries in the FAQ, the number of different authors and readers, or the age of entries (which may denote obsolete pages).

5.2 An approach for data mining

As highlighted in our state of the art, one has to pay great attention to the approach used in the measurement or mining process to ensure consistency and meaning of the measures. Besides relying on the experience of software measurement programs and meaning-safe methods, we propose a few guidelines to be followed when building up a data mining process. They are summarised in figure 5.4.

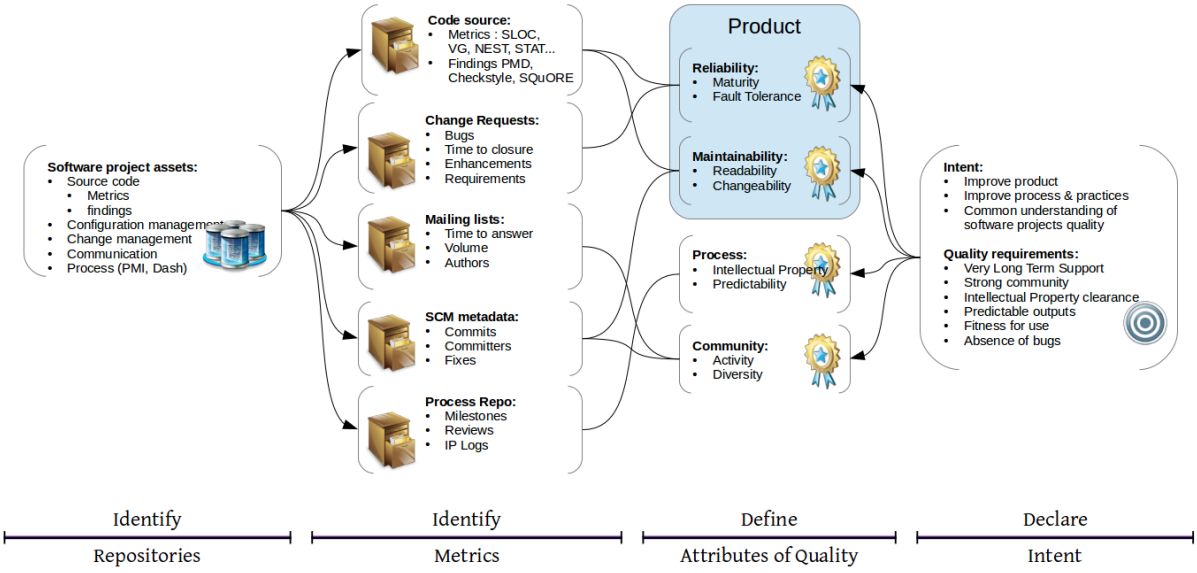


Figure 5.4: From quality definition to repository metrics.

We applied them on the few experiences we had during the Maisqual project and found them practical and useful. These proved to be successful in ensuring the integrity and usability of information and keeping all the actors synchronised on the same concerns and solutions.

5.2.1 Declare the intent

The whole mining process is driven by its stated goals. The quality model and attributes, means to measure it, and presentation of the results will differ if the program is designed

as an audit-like, acceptance test for projects, or as an incremental quality improvement program to ease evolution and maintenance of projects. The users of the mining program, who may be developers, managers, buyers, or end-users of the product, have to map its objectives to concepts and needs they are familiar with. Including users in the definition of the intent of the program also helps to prevent counter-productive use of the metrics and quality models.

The intent must be simple, clearly expressed in a few sentences, and published for all considered users of the program.

5.2.2 Identify quality attributes

The concerns identified in the intent are then decomposed into quality attributes. Firstly this gives a structure to the quality model, and secondly allows to rely on well-defined characteristics – which greatly simplifies the communication and exchange of views. Recognised standards and established practices provide useful frameworks and definitions for this step. One should strive for simplicity when elaborating quality attributes and concepts. Common sense is a good argument, and even actors that have no knowledge of the field associated to the quality attributes should be able to understand them. Obscurity is a source of fear and distrust and must be avoided.

The output of this step is a fully-featured quality model that reflects all of the expected needs and views of the mining program. The produced quality model is also a point of convergence for all actors: requirements of different origin and nature are bound together and form a unified, consistent view.

5.2.3 Identify available metrics

Once we precisely known what quality characteristics we are looking for, we have to identify measures that reflect this information need. Data retrieval is a fragile step of the mining process. Depending on the information we are looking for, various artefact types and measures may be available: one has to select them carefully according to their intended purpose. The different repositories available for the projects being analysed should be listed, with the measures that may be retrieved from them. Selected metrics have to be stable and reliable (i.e. their meaning to users must remain constant over time and usage), and their retrieval automated (i.e. no human intervention is required).

This step also defines how the metrics are aggregated up to the top quality characteristics. Since there is no universally recognised agreement on these relationships one has to rely on local understanding and conventions. All actors, or at least a vast majority of them, should agree on the meaning of the selected metrics and the links to the quality attributes.

5.2.4 Implementation

The mining process must be fully automated, from data retrieval to results presentation, and transparently published. Automation enables to reliably and regularly collect the information, even when people are not available[3] or not willing to do it[4]. When digging back into historical records, missing data poses a serious threat to the consistency of results and to the algorithms used to analyse them – information is often easier to get at the moment than later on. The publishing of the entire process also helps people understand what quality is in this context and how it is measured (i.e. no magic), making them more confident in the process.

5.2.5 Presentation of results

Visualisation of results is of primary importance for the efficiency of the information we want to transmit. In some cases (e.g. for incremental improvement of quality) a short list of action items will be enough because the human mind feels more comfortable correcting a few warnings than hundreds of them. But if our goal is to audit the technical debt of the product, it would be better to list them all to get a good idea of the actual status of the product's maintainability. Pictures and plots also show to be very useful to illustrate ideas and are sometimes worth a thousand words. If we want to highlight unstable files, a plot showing the number of recent modifications would immediately spot the most unstable of them and show how much volatile they are compared to the average.

5.3 Implementation

5.3.1 Selected tools

Besides the tools used by the software project themselves (e.g. Subversion, Git, MHonArc..), we had to select tools to setup and automate the retrieval and analysis process. We naturally relied on SQuORE for the static analysis part, and on a set of open-source tools commonly used by free/libre projects. This has two advantages: firstly they can be downloaded and used for free, and secondly they provide well-known features, checks, and results. As an example on static analysis tools, since we restricted our study to Java projects selected tools for bad practices and violations detections have been Checkstyle and PMD. Practitioners using the generated data sets will be able to investigate e.g. how the checks are implemented and map them to documented practices they know or have seen in projects.

[3] Nobody has time for data retrieval before a release or during holidays.

[4] Kaner cites in [105] the interesting case of testers waiting for the release before submitting new bugs, pinning them on the wall for the time being.

Checkstyle

Checkstyle is an open-source static code analyser that checks for violations of style and naming rules in Java files. It was originally developed by Oliver Burn back in 2001, mainly to check code layout issues, but since the architecture refactoring that occured in version 3 many new types of checks have been added. As of today, Checkstyle provides a few checks to find class design problems, duplicate code, or bug patterns like double checked locking.

The open-source community has widely adopted Checkstyle: many projects propose a Checkstyle configuration file to ensure consistency of the practices in use. Checkstyle checks can be seamlessly integrated in the development process of projects through several integrations with common tools used in Java like Ant, Maven, Jenkins, Eclipse, Emacs or Vim.

It offers a comprehensive set of built-in rules, and allows to easily define new custom checks. The types of checks available out-of-the-box include *coding style* checks, that target code layout and structure conformance to established standards, or *naming conventions* checks for generic or custom naming of artefacts.

When we started to work on Maisqual, we used Checkstyle 5.5, which was the latest version at that time and the one supported by the SQuORE engine. The 5.6 version has since been released, introducing new checks and fixing various bugs, and SQuORE adopted it in the 2013-B release. Version 3.0 of the data sets has been generated with Checkstyle 5.6 and some of the rules checked are described in section 6.2.3.

PMD

PMD is another static code analyzer commonly used in open-source projects to detect bad programming practices. It finds common Java programming flaws like unused variables, empty catch blocks, unnecessary object creation, among others. Its main target is Java, although some rules target other languages like JavaScript, XML, and XSL. It also integrates with various common Java tools, including Ant, Maven, Jenkins, Eclipse, and editors like Emacs JDE and Vim.

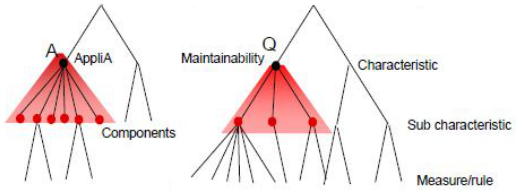
Compared to Checkstyle's, PMD checks are rather oriented towards design issues and bad coding practices. Examples include dead code, exception handling, or direct implementations instead of using interfaces.

The version of PMD included in SQuORE at the beginning of the project was PMD 4.3. As for Checkstyle, we switched to PMD 5.0.5 for the generation of version 3.0 of the data sets. Some of the rules checked are described in section 6.2.4.

SQuORE

To ensure consistency between all artefact measures retrieved from different sources, we relied on SQuORE, a professional tool for software project quality evaluation and business intelligence [11]. It features a *parser*, which computes code metrics and builds a tree of

artefacts for the various data providers. The *engine* then associates measures to each node, aggregates data to upper levels, and stores them in a Postgres database.



SQuORE has two features that proved to be useful for our purpose. Firstly, the input to the engine can be any type of information or format; as an example we could write specific data providers to analyse configuration management metadata and to retrieve communication channels information through various means (Forums, NNTP, MBoxes). Secondly the aggregation of data as well as the measures and scales used can be entirely configured in custom quality models. A quality model defines a hierarchy of quality attributes and links them to metrics. Findings from rule checking tools are integrated through derived metrics like the *number of violations of rule X on this artefact*.

We used the SQuORE dashboard to quickly verify measures as they were retrieved, with temporal plots and tables as picture in figure 5.5.

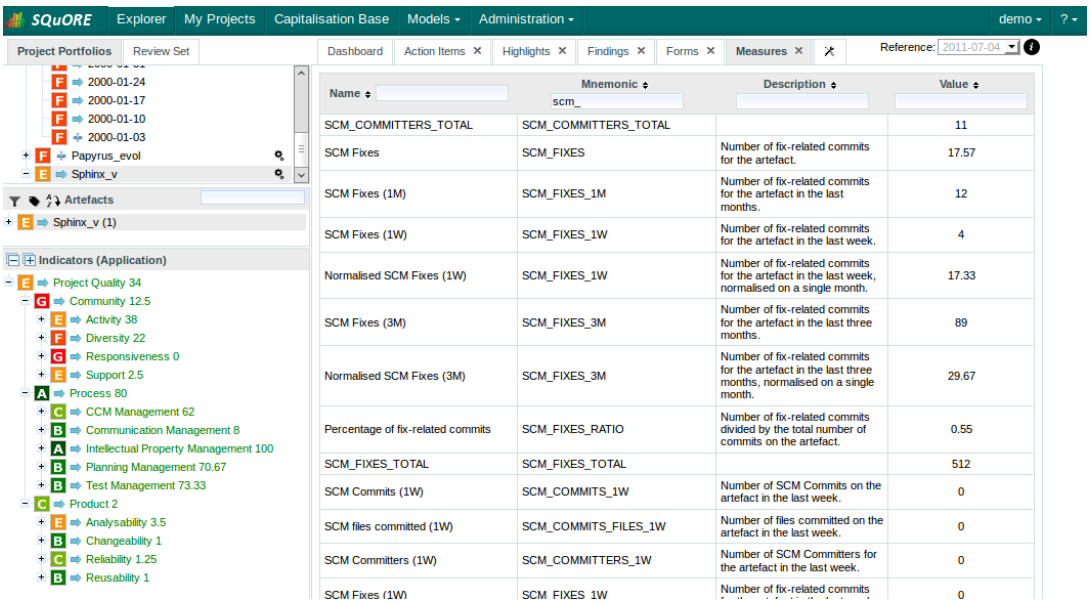


Figure 5.5: Validation of data in the SQuORE dashboard.

5.3.2 Data retrieval

Figure 5.6 depicts the whole process: metrics are retrieved from source code with the SQuORE parser and external tools (Checkstyle, PMD) while configuration management metadata and mailing lists are dug with custom data providers written in Perl. Once SQuORE has finished aggregating the different sources of information, we retrieve them from the database to generate the CSV files. Data can be checked visually in the SQuORE dashboard and is validated through R/Knitr scripts.

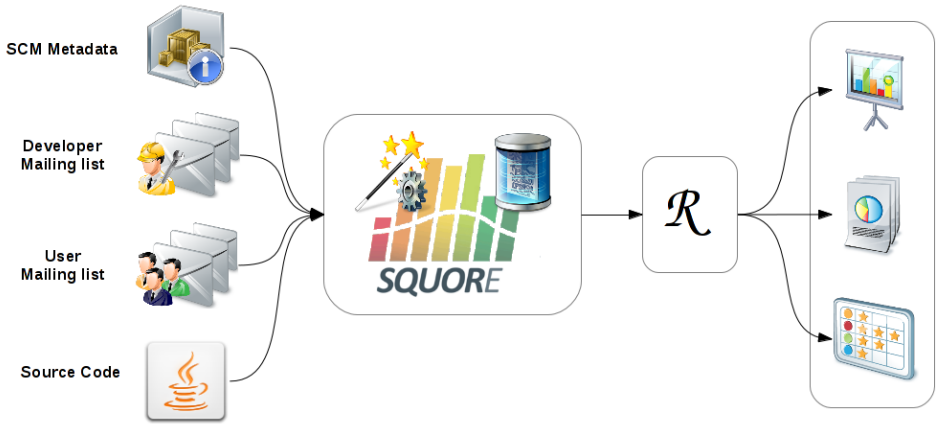


Figure 5.6: Data retrieval process.

Source code

Source code is extracted from cvs, subversion or git repositories at specific dates. We had to check exported snapshots manually and to correct them when needed: some dates were missing (e.g. when there is no commit at the specified date), or incomplete (e.g. when the remote server cuts bandwidth to preserve resources).

Using Subversion one can use the `export` command to directly fetch the contents of the repository at a specific date:

```
1 svn export http://svn.apache.org/repos/asf/ant/core/trunk -r {2000-02-21}
```

Using Git one has first to clone the repository, then reset it to the commit corresponding to the desired date. The Git-specific directories (namely `.git` and `.gitignore`) can then safely be deleted to preserve disk space:

```
1 git clone https://git.eclipse.org/r/p/mylyn/org.eclipse.mylyn.git
2 git reset --hard $(git rev-list -1 $(git rev-parse --until=${i}) master)
3 rm -rf .git .gitignore
```

Releases were manually downloaded from the project web site, and extracted to the sources directory[5]. It should be noted that releases versioning may be discontinuous (e.g. jumping from 1.2.1 to 1.2.3); this might happen when a critical bug is found after the release process has been started (i.e. sources have been tagged) but before the publishing has occured. In these cases the next version usually gets out within a few days.

Configuration management metadata

Basically all configuration management systems implement the same core functionalities – e.g. checkout, commits or revisions, history or log of artefacts. Although some tools provide supplemental kinds of information (e.g. a reliable link to the change requests associated with the commit) there is a common basis of meta data that is available from

[5]The full set of sources takes up to 300GB of disk space.

all decent modern tools. The basic information we want to retrieve is the author, date and reason (message) of each commit that occured at the file and application levels. We retrieve this information with the following command:

Using Subversion:

```
svn log -v --xml http://svn.apache.org/repos/asf/ant/core/ > ant_log.xml
```

Using Git:

```
git log --stat=4096 -- . > git_log.txt
```

We defined in our context a fix-related commit as having one of the terms *fix*, *bug*, *issue* or *error* in the message associated with the revision. Depending on the project tooling and conventions, more sophisticated methods may be set up to establish the link between bugs and commits [9].

We could not observe consistent naming practices for commit messages across the different projects and forges that we mined. Some projects published guidelines or explicitely relied on more generic conventions like the official Java Coding Convention while some others did not even mention it. None of them enforced these practices however, e.g. by setting up pre-commit hooks on the repository.

5.3.3 Data analysis

The main place for trying things out was the **version analysis** Knitr document. It steadily evolved to include new techniques and graphics and is now 85 pages. Sections that were too time-consuming or not robust enough were shut down when needed. Specific Knitr documents were written to address

5.3.4 Automation

We used Jenkins as a continuous integration server to automate the different processes. Basically put, it allows to define sequential steps and schedule long runs in an automated, repeatable manner. It integrates well with a lot of useful third-party components: build tools, configuration management and bug tracking systems, and can be finely tuned to fit the most extravagant processes.

One useful feature of Jenkins is it enables one to configure template projects of which the steps can be reused by other projects as illustrated in figure 5.7. We defined a few common jobs for the analysis of an arbitrary single version, a series of successive extracts, and the current state of a project. Project-specific jobs were then setup that would run the steps defined in the templates with customised values.

The Jenkins process hosts all processes executed during the analysis of a project, from the Checkstyle, PMD and SQuORE parser to the R analyses, and consumes a lot of resources. The generated builds take 45 GB of disk space while the Jenkins workspace

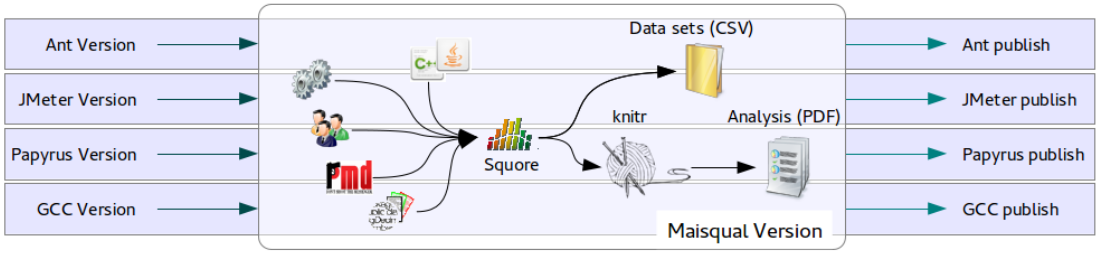


Figure 5.7: Implementation of the retrieval process in Jenkins.

takes up to 145 GB. It runs on a fairly good server[6] which has eight processors and 24 GB of RAM.

5.4 Summary

In this chapter we defined the building blocks of Maisqual. We ascertained the structure of available information sources and delineated a nominal topology for the projects we intended to analyse. From a methodological perspective, we instituted requirements to ensure the relevance and semantic consistancy of our results and eventually developed the requirements on a dedicated server. This provided us with a strong framework upon which to build reproducible and theoretically sound research.

The work accomplished during this preparation step has also benefited SQUORING Technologies in the following areas:

- ⇝ The dedicated data providers developed for the acquisition of unusual data have uncovered new areas for the SQUORE product. The configuration management metadata extraction of application- and file-level measures is already integrated in the SQUORE repository. The preparation work for other areas (change requests, communication means, and other process-related sources) provides a sound premise for upcoming data providers.
- ⇝ The application of state of the art research, the pitfalls and common issues identified during the retrieval and analysis process, and the techniques used to detect and defeat them, also contributed to the SQUORING Technologies knowledge and experience. This proved to be useful for investigating paths for the project quality assessment initiative of the SQUORE product, and to widen the scientific level of knowledge of the teams.

[6]The server is hosted by ovh at http://ns228394.ovh.net. The Jenkins instance can be accessed at http://ns228394.ovh.net:8080 and the SQUORE instance can be accessed at http://ns228394.ovh.bet:8180

Chapter 6

Generating the data sets

This chapter introduces the data sets that were generated for the SQUORE Labs using the framework setup in chapter 5. The data sets produced during this phase have different structures and characteristics. Hence three different types of sets are defined:

- ⇝ *Evolution data sets* show a weekly snapshot of the project repository over a long period of time – up to 12 years, in the case of Ant. The time interval between extracts is constant: the source has been extracted every Monday over the defined period of time. Characteristics include common and differential code metrics, communication measures, and configuration management measures.
- ⇝ *Release data sets* show analyses of the source releases of the software product, as published by the project. They differ from the evolution data sets in their nature (the source release does not necessarily show the same structure and contents as the project repository) and in time structure (releases show no time structure: a new release may happen only a few day after the previous one – e.g. in the case of blocking issues, or months after).
- ⇝ *Version data sets* present information on a single release or extract of a product. They include only static information: all time-dependent attributes are discarded, but they still provide valuable insights into the intrinsic measures and detected practices.

We first detail in section 6.1 the metrics selected for extraction. We give a short description for each metric and provide pragmatic examples to serve as requirements for reproducible research. The rules and practices checked on source code are then presented in section 6.2. Some of the major projects that we analysed are quickly described in section 6.3 to provide insights on their context and history. Finally section 6.4 summarises the different types of data sets that were generated.

6.1 Defining metrics

6.1.1 About metrics

The definition of metrics is an important step for the comprehension and the consistency of the measurement program, as pointed out by Fenton [62] and Kaner [105]. Their arguments can be summarised in as follows:

- ⇝ The exact definition of the measurement process and how to retrieve the information reliably. A common example proposed by Kaner in [105] is the Mean Time Between Failures (MTBF) metric. This point is maintained both at the retrieval stage (is the actual measure really the information we intended to get?) and at the analysis phase (how do people understand and use the definition of the measure?).
- ⇝ The meaning and interpretation of measures, which vary greatly according to the development process, local practices and the program objectives. Compared to the previous case, even if a measure signifies exactly what was intended, its implications will not be the same according to the external and internal *context* of the project. As an example, developers working in a collaborative open-source project will not have the same productivity and hierarchical constraints as a private company with financial and time constraints [148].

To circumvent these pitfalls, we paid great attention to:

- ⇝ Clearly define the metrics to be comprehended by everyone.
- ⇝ Provide examples of values for a given chunk of code.
- ⇝ Give context and some indications and common beliefs about the metric's usage.

Giving code samples with values of metrics serves both for understandability and as a reference. On the one hand, one is able to get an approximate idea for the values of metrics on a sample of code structures and try her understanding on real code. On the other hand, these samples can be used as requirements for a custom development if needed for reasons of reproducibility.

Some metrics are only available for specific contexts, like the number of classes for object-oriented programming. Metrics available for each type of data set are described in table 6.1. *OO* metrics are CLAS (number of classes defined in the artefact) and DITM (Depth of Inheritance Tree). *Diff* metrics are LADD, LMOD, LREM (Number of lines added, modified and removed since the last analysis). *Time* metrics for SCM are SCM_*_1W, SCM_*_1M, and SCM_*_3M. *Total* metrics for SCM are SCM_COMMITS_TOTAL, SCM_COMMITTERS_TOTAL, SCM_COMMITS_FILES_TOTAL and SCM_FIXES_TOTAL. *Time* metrics for Communication are COM_*_1W, COM_*_1M, and COM_*_3M.

Metrics defined on a lower level (e.g. function) can be aggregated to upper level in a smart manner: as an example, the cyclomatic number at the file level is the overall sum of its function's cyclomatic numbers. The meaning of the upper-level metric shall be interpreted with this fact in mind, since it may introduce a bias (also known as the

Table 6.1: Metrics artefact types.

	Code			SCM		COM
	Common	OO	Diff.	Total	Time	Time
Java Evolution	X	X	X	X	X	X
Java Releases	X	X	X	X		
Java Versions	X	X		X		
C Evolution	X		X	X	X	X
C Releases	X		X	X		
C Versions	X			X		

Ecological fallacy, cf. section 2.2.1). When needed, the *smart manner* used to aggregate information at upper levels is described hereafter.

All data sets are structured in three files, corresponding to the different artefact types that were investigated: application, file and function. The artefact levels available for each metric are detailed in their own sections: tables 6.2 and 6.3 for code metrics, table 6.4 for configuration management metrics, and table 6.5 for communication metrics.

6.1.2 Code metrics

We present in table 6.2 the code metrics that are available in all data sets, with the artefact levels they are available at.

Artefact counting metrics

Artefact counting metrics are FILE, FUNC and CLAS (which is defined in the section dedicated to object-oriented measures). The **number of files** (FILE) counts the number of source files in the project, i.e. which have an extension corresponding to the defined language (`.java` for Java or `.c` and `.h` files for C). The **number of functions** (FUNC) sums up the number of methods or functions recursively defined in the artefact.

Line counting metrics

Line counting metrics propose a variety of different means to grasp the size of code from different perspectives. It includes STAT, SLOC, ELOC, CLOC, MLOC, and BRAC.

The **number of statements** (STAT) counts the total number of instructions. Examples of instructions include control-flow tokens, plus `else`, `cases`, and assignments.

Source lines of code (SLOC) is the number of non-blank and non-comment lines in code. **Effective lines of code** (ELOC) also removes the number of lines that contain only braces.

Comment lines of code (CLOC) counts the number of lines that include a comment in the artefact. If a line includes both code and comment, it will be counted in SLOC, CLOC and MLOC metrics.

Table 6.2: Artefact types for common code metrics.

Artefact counting metrics	Mnemo	App	File	Func.
Number of files	FILE	X		
Number of functions	FUNC	X	X	
Line counting metrics	**Mnemo**	**App**	**File**	**Func.**
Lines of braces	BRAC	X	X	X
Blank lines	BLAN	X	X	X
Effective lines of code	ELOC	X	X	X
Source lines of code	SLOC	X	X	X
Line count	LC	X	X	X
Mixed lines of code	MLOC	X	X	X
Comment lines of code	CLOC	X	X	X
Miscellaneous	**Mnemo**	**App**	**File**	**Func.**
Non conformities count	NCC	X	X	X
Comment rate	COMR	X	X	X
Number of statements	STAT	X	X	X
Control flow complexity metrics	**Mnemo**	**App**	**File**	**Func.**
Maximum nesting level	NEST			X
Number of Paths	NPAT			X
Cyclomatic number	VG	X	X	X
Control flow tokens	CFT	X	X	X
Halstead metrics	**Mnemo**	**App**	**File**	**Func.**
Total number of operators	TOPT			X
Number of distinct operators	DOPT			X
Total number of operand	TOPD			X
Number of distinct operands	DOPD			X

Mixed lines of code (MLOC) counts the number of lines that have both code and comments, and **braces** (BRAC) counts the number of lines that contain only braces. In this context, we have the following relationships:

$$\text{SLOC} = \text{ELOC} + \text{BRAC}$$

$$\text{LC} = \text{SLOC} + \text{BLAN} + \text{CLOC} - \text{MLOC}$$

$$\text{LC} = (\text{ELOC} + \text{BRAC}) + \text{BLAN} + \text{CLOC} - \text{MLOCCOMR} \quad = ((\text{CLOC} + \text{MLOC}) \times 100)/(\text{ELOC} + \text{CLOC}))$$

The following examples of line counting metrics are provided for reference. The `forceLoadClass` function has 10 lines (LC), 8 source lines of code (SLOC), 6 effective lines of code (ELOC), 2 blank lines (BLAN), and 0 comment lines of code (CLOC).

```
public Class forceLoadClass(String classname) throws ClassNotFoundException {
  log("force␣loading␣" + classname, Project.MSG_DEBUG);

  Class theClass = findLoadedClass(classname);

  if (theClass == null) {
    theClass = findClass(classname);
  }
  return theClass;
}
```

The `findClassInComponents` function has 30 lines (LC), 27 source lines of code (SLOC), 21 effective lines of code (ELOC), 0 blank line (BLAN), 3 comment lines of code (CLOC), 0 mixed lines of code (MLOC), and 5 lines of braces (BRAC).

```
private Class findClassInComponents(String name)
  throws ClassNotFoundException {
  // we need to search the components of the path to see if
  // we can find the class we want.
  InputStream stream = null;
  String classFilename = getClassFilename(name);
  try {
    Enumeration e = pathComponents.elements();
    while (e.hasMoreElements()) {
      File pathComponent = (File) e.nextElement();
      try {
        stream = getResourceStream(pathComponent, classFilename);
        if (stream != null) {
          log("Loaded␣from␣" + pathComponent + "␣"
            + classFilename, Project.MSG_DEBUG);
          return getClassFromStream(stream, name, pathComponent);
        }
      } catch (SecurityException se) {
        throw se;
      } catch (IOException ioe) {
        // ioe.printStackTrace();
        log("Exception␣reading␣component␣" + pathComponent + "␣(reason:␣"
            + ioe.getMessage() + ")", Project.MSG_VERBOSE);
      }
    }
    throw new ClassNotFoundException(name);
  } finally {
    FileUtils.close(stream);
```

```
29   }
30 }
```

Control flow complexity metrics

The **maximum nesting level** (NEST) counts the highest number of imbricated code (including conditions and loops) in a function. Deeper nesting threatens understandability of code and induces more test cases to run the different branches. Practitioners usually consider that a function with three or more nested levels becomes significantly more difficult for the human mind to apprehend how it works.

The **number of execution paths** (NPAT) is an estimate of the possible execution paths in a function. Higher values induce more test cases to test all possible ways the function can execute depending on its parameters. An infinite number of execution paths typically indicates that some combination of parameters may cause an infinite loop during execution.

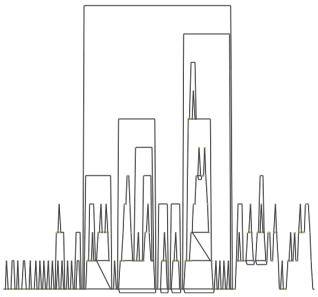
The **cyclomatic number** (VG), a measure borrowed from graph theory and introduced by McCabe in [130] is the number of linearly independent paths that comprise the program. To have good testability and maintainability, McCabe recommends that no program modules (or functions as for Java) should exceed a cyclomatic number of 10. It is primarily defined at the function level and is summed up for higher levels of artefacts.

The **number of control-flow tokens** (CFT) counts the number of control-flow oriented operators (e.g. `if`, `while`, `for`, `switch`, `throw`, `return`, ternary operators, blocks of execution). `else` and `case` are typically considered a part of respectively `if` and `switch` and are not counted.

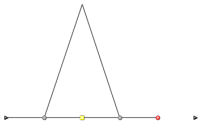
The control flow graph of a function visually plots all paths available when executing it. Examples of control flow are provided in figure 6.0; figures 6.0a and 6.0b shows two Java examples on Ant (the code of these functions is reproduced in appendix D.1 page 279 for reference) and figure 6.0c shows a C example extracted from GCC. But control flows can be a lot more complex, as exemplified in figure 6.0d for an industrial application.

In these examples, the functions have the following characteristics.

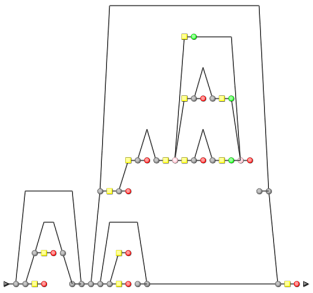
Metric	(a)	(b)	(c)
Language	Java	Java	C
Line Count	407	6	99
Source Lines of Code	337	6	95
Executable Statements	271	4	62
Cyclomatic Complexity	78	2	18
Maximum Nested Structures	7	1	5
Non-Cyclic Paths	inf	2	68
Comment rate	9 %	0 %	4 %



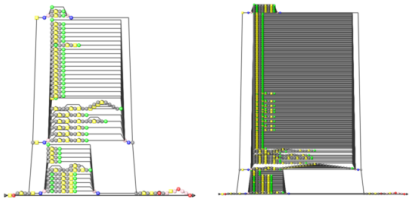
(a) Ant `Javadoc.java/execute()`



(b) Ant `Javadoc.java/createSourcepath()`



(c) GCC `GC_make_array_descriptor()`



(d) Control flow examples

Figure 6.0: Control flow examples

Halstead metrics

Halstead proposed in his *Elements of Software Science* [82] a set of metrics to estimate some important characteristics of a software. He starts by defining 4 base measures: the **number of distinct operands** (DOPD, or n_2), the **number of distinct operators** (DOPT, or n_1), and the **total number of operands** (TOPD, or N_2) and the **total number of operators** (TOPT, or N_1). Together they constitute the following higher-level derived metrics:

- program vocabulary: $n = n_1 + n_2$,
- program length: $N = N_1 + N_2$,
- program difficulty: $D = \frac{n_1}{2} \times \frac{N_2}{n_2}$,
- program volume: $V = N \log_2 n$,
- estimated effort needed for program comprehension: $E = D \times V$,
- estimated number of delivered bugs: $B = \frac{E^{2/3}}{3000}$.

In the data sets, only the four base measures are retrieved: DOPD, DOPT, TOPD, and TOPT. Derived measures are not provided in the data sets since they can all be computed from the provided base measures.

Rules-oriented measures

NCC is the **number of non-conformities** detected on an artefact. From the practices perspective, it sums the number of times any rule has been transgressed on the artefact (application, file or function). The **rate of acquired practices** (ROKR) is the number of respected rules (i.e. with no violation detected on the artefact) divided by the total number of rules defined for the run. It shows the number of acquired practices with regards to the full rule set.

Specific code metrics

Table 6.3: Artefact types for specific code metrics.

Differential metrics	Mnemo	App	File	Func.
Lines added	LADD	X	X	X
Lines modified	LMOD	X	X	X
Lines removed	LREM	X	X	X
Object-Oriented metrics	**Mnemo**	**App**	**File**	**Func.**
Maximum Depth of Inheritance Tree	DITM	X		
Number of classes	CLAS	X	X	X
Rate of acquired rules	ROKR	X	X	X

Differential measures

Differential measures are only available for evolution and release data sets. They quantify the **number of lines added** (LADD), **modified** (LMOD) or **removed** (LREM) since the last analysis, be it a week (for evolution data sets) or a random delay between two releases (which varies from one week to one year). They give an idea about the volume of changes (either bug fixes or new features) that occured between two releases. In the case of large refactoring, or between major releases, there may be a massive number of lines modified, whereas only small increments may be displayed for more agile-like, often-released projects (e.g. Jenkins). As shown in table 6.3 differential measures are available at all artefact levels: application, file, function.

Object-oriented measures

Three measures are only available for object-oriented code. They are the number of classes (CLAS), the maximum depth of inheritance tree (DITM), and the above-mentionned rate of acquired rules (ROKR). Table 6.2 lists the artefact levels available for these metrics.

The **number of classes** sums up the number of classes defined in the artefact and its sub-defined artefacts. One file may include several classes, and in Java anonymous classes may be included in functions. The **maximum depth of inheritance tree** of a class

within the inheritance hierarchy is defined as the maximum length from the considered class to the root of the class hierarchy tree and is measured by the number of ancestor classes. In cases involving multiple inheritance, the DITM is the maximum length from the node to the root of the tree [161]. It is available solely at the application level.

A deep inheritance tree makes the understanding of the object-oriented architecture difficult. Well structured OO systems have a forest of classes rather than one large inheritance lattice. The deeper the class is within the hierarchy, the greater the number of methods it is likely to inherit, making it more complex to predict its behavior and, therefore, more fault-prone [77]. However, the deeper a particular tree is in a class, the greater potential reuse of inherited methods [161].

ROKR is another measure specific to object-oriented code, since it is computed relatively to the full number of rules, including Java-related checks. In the case of C projects only the SQuORE rules are checked, so it loses its meaning and is not generated.

6.1.3 Software Configuration Management metrics

Configuration management systems hold a number of meta-information about the modifications committed to the project repository. The following metrics are defined:

The **number of commits** (SCM_COMMITS) counts the commits registered by the software configuration management tool for the artefact on the repository (either the trunk or a branch). At the application level commits can concern any type of artefacts (e.g. code, documentation, or web site). Commits can be executed for many different purposes: e.g. add feature, fix bug, add comments, refactor, or even simply re-indent code.

The **number of fixes** (SCM_FIXES) counts the number of fix-related commits, i.e. commits that include either the *fix*, *issue*, *problem* or *error* keywords in their message. At the application level, all commits with these keywords in message are considered until the date of analysis. At the file level, it represents the number of fix-related revisions associated to the file. If a file is created while fixing code (i.e. its first version is associated to a fix commit) the fix isn't counted since the file cannot be considered responsible for a problem that has been detected when it wasn't there.

The **number of distinct committers** (SCM_COMMITTERS) is the total number of different committers registered by the software configuration management tool on the artefact. On the one hand, having less committers enforces cohesion, makes keeping coding and naming conventions respected easier, and allows easy communication and quick connection between developers. On the other hand, having a large number of committers means the project is active; it attracts more talented developers and more eyes to look at problems. The project has also better chances to be maintained over the years.

It should be noted that some practices may threaten the validity of this metric. As an example occasional contributors may send their patches to official maintainers who review it before integrating it in the repository. In such cases, the commit is executed by the official committer, although the code has been originally modified by an anonymous (at least for us) developer. Some core maintainers use a convention stating the name

or identifier of the contributor, but there is no established or enforced usage of such conventions. Another point is that multiple online personas can cause individuals to be represented as multiple people [86].

The **number of files committed** (SCM_COMMIT_FILES) is the number of files associated to commits registered by the software configuration management tool. This measure allows to identify big commits, which usually imply big changes in the code.

Table 6.4: Artefact types for configuration management.

Configuration management	Mnemo: SCM_	App.	File	Func.
SCM Fixes	FIXES	X	X	
SCM Commits	COMMITS	X	X	
SCM Committers	COMMITTERS	X	X	
SCM Committed files	COMMITTED_FILES	X		

To reflect recent activity on the repository, we retrieved measures both on a limited time basis and on a global basis: in the last week (e.g. SCM_COMMITS_1W), in the last month (e.g. SCM_COMMITS_1M), and in the last three months (e.g. SCM_COMMITS_3M), and in total (e.g. SCM_COMMITS_TOTAL). Metrics available at the different levels of artefacts are presented in table 6.4.

6.1.4 Communication metrics

Communication metrics show an unusual part of the project: people's activity and interactions during the elaboration of the product. Most software projects have two communication media: one targeted at the internal development of the product, for developers who actively contribute to the project by committing in the source repository, testing the product, or finding bugs (a.k.a. developers mailing list); and one targeted at end-users for general help and good use of the product (a.k.a. user mailing list).

The type of media varies across the different forges or projects: most of the time mailing lists are used, with a web interface like MHonArc or mod_mbox. In some cases, projects may use as well forums (especially for user-oriented communication) or NNTP news servers, as for the Eclipse foundation projects. The variety of media and tools makes it difficult to be extensive; however data providers can be written to map these to the common mbox format. We wrote connectors for mboxes, MHonArc, GMane and FUDForum (used by Eclipse). The following metrics are defined:

The **number of posts** (COM_DEV_VOL, COM_USR_VOL) is the total number of mails posted on the mailing list during the considered period of time. All posts are counted, regardless of their depth (i.e. new posts or answers).

The **number of distinct authors** (COM_DEV_AUTH, COM_USR_AUTH) is the number of people having posted at least once on the mailing list during the considered period of time. Authors are counted once even if they posted multiple times, based on their email address.

The **number of threads** (COM_DEV_SUBJ, COM_USR_SUBJ) is the number of diffent subjects (i.e. a question and its responses) that have been posted on the mailing list during the considered period of time. Subjects that are replies to other subjects are not counted, even if the subject text is different.

The **number of answers** (COM_DEV_RESP_VOL, COM_USR_RESP_VOL) is the total number of replies to requests on the user mailing list during the considered period of time. A message is considered as an answer if it is using the *Reply-to* header field. The number of answers is often associated to the number of threads to compute the useful *response ratio* metric.

The **median time to first reply** (COM_DEV_RESP_TIME_MED, COM_USR_RESP_TIME_MED) is the number of seconds between a question (first post of a thread) and the first response (second post of a thread) on the mailing list during the considered period of time.

Table 6.5: Artefact types for communication channels.

Communication metrics	Mnemo: COM_	App	File	Func.
Dev Volume	DEV_VOL	X		
Dev Subjects	DEV_SUBJ	X		
Dev Authors	DEV_AUTH	X		
Dev Median response time	DEV_RESP_TIME_MED	X		
User Volume	USR_VOL	X		
User Subjects	USR_SUBJ	X		
User Authors	USR_AUTH	X		
User Median response time	USR_RESP_TIME_MED	X		

As for configuration management metrics, we worked on temporal measures to produce measures for the last week, last month, and last three months. Communication metrics are only available at the application level, as shown in table 6.5.

6.2 Defining rules and practices

6.2.1 About rules

Rules are associated to practices that have an impact on some characteristic of quality. We list hereafter the rule checks that we decided to include in our data sets, with information about their associated practices and further references. The categories defined in the rule sets are mapped to the ISO 9126 decomposition of quality (analysability, changeability, stability, testability, etc.) and to development concerns (programming techniques, architecture, etc.).

Many of these rules can be linked to coding conventions published by standardisation

organisms like MISRA[1] (the Motor Industry Software Reliability Association, which C rule set is well spread) or CERT[2] (Carnegie Mellon's secure coding instance). They usually give a ranking on the remediation cost and the severity of the rule. There are also language-specific coding conventions, as is the case with Sun's coding conventions for the Java programming language [171].

There are 39 rules from Checkstyle, 58 rules from PMD, and 21 rules from SQuORE. We define hereafter only a subset of them, corresponding to the most common and prejudicial errors. In the data sets, conformity to rules is displayed as a *number of violations* of the rule for the given artefact.

6.2.2 SQuORE

SQuORE rules apply to C, C++ and Java code. The following families of rules are defined: fault tolerance (2 rules), analysability (7 rules), maturity (1 rule), stability (10 rules), changeability (12 rules) and testability (13 rules). The most common checks are:

- **No fall through** (R_NOFALLTHROUGH). There shall be no fall through the next `case` in a `switch` statement. It threatens analysability of code (one may not understand where execution goes through) and stability (one needs to modify existing code to add a feature). It is related to rules Cert MSC17-C, Cert MSC18-CPP, Cert MSC20-C and MISRA-C (2004) 15-2.
- **Compound if** (R_COMPOUNDIFELSE). An `if (expression)` construct shall be followed by a compound statement. The `else` keyword shall be followed by either a compound statement, or another if statement. It is related to MISRA-C (2004) rule 14.9.
- **Final else** (R_ELSEFINAL). All `if ... else if` constructs shall be terminated with an `else` clause. This impacts changeability, because developers can identify quickly what is the default treatment, and fault tolerance because unseen cases are caught. It is related to MISRA-C (2004) rule 14.10.
- **No multiple breaks** (R_SGLBRK). For any iteration statement there shall be at most one `break` statement used for loop termination. It is related to MISRA-C (2004) rule 14.6 and impacts the analysability (developers will have trouble understanding the control flow) and testability (more paths to test) of code.
- **No goto** (R_NOGOTO). `Gotos` are considered bad practice (MISRA-C (2004) rule 14.4) and may be hazardous (see CERT MSC35-CPP): they threaten the analysability of code, because one needs to scroll through the file instead of following a clear sequence of steps, and makes the test cases harder to write. Similarly, the **no backward goto** (R_BWGOTO) rule searches for `goto` operators that link to code that lies *before* the `goto`. Backward gotos shall not be used. They shall be rewritten using a loop instead.

[1] See the official MISRA web site at http://www.misra.org.uk/.
[2] See Carnegie Mellon's web site: https://www.securecoding.cert.org.

- ⇝ **No assignment in Boolean** (R_NOASGINBOOL). Assignment operators shall not be used in expressions that yield a boolean value. It is related to Cert EXP45-C, Cert EXP19-CPP and MISRA-C (2004) rule 13.1.
- ⇝ **No assignment in expressions without comparison** (R_NOASGCOND). Assignment operators shall not be used in expressions that do not contain comparison operators.
- ⇝ **Case in switch** (R_ONECASE). Every `switch` statement shall have at least one `case` clause. It is releated to MISRA-C (2004) rule 15.5.
- ⇝ **Label out of a switch** (R_NOLABEL). A `switch` label shall only be used when the most closely-enclosing compound statement is the body of a `switch` statement. It is related to MISRA-C (2004) rule 15.1.
- ⇝ **Missing Default** (R_DEFAULT). The final clause of a `switch` statement shall be the `default` clause. It is related to Cert MSC01-C, Cert MSC01-CPP and MISRA-C (2004) rule 15.3.
- ⇝ **Code before first case** (R_NOCODEBEFORECASE). There shall be no code before the first case of a `switch` statement. It is related to Cert DCL41-C.

6.2.3 Checkstyle

We identified 39 elements from the Checkstyle 5.6 rule set, corresponding to useful practices generally well adopted by the community. The quality attributes impacted by these rules are: analysability (23 rules), reusability (11 rules), reliability (5 rules), efficiency (5 rules), testability (3 rules), robustness (2 rules) and portability (1 rule). All rules are described on the project page at `http://checkstyle.sourceforge.net/config.html`.

- ⇝ **Javadoc checks** (JAVADOCMETHOD, JAVADOCPACKAGE, JAVADOCTYPE, JAVADOCVARIABLE) ensure that Javadoc comments are present at the different levels of information as it is recommended by Sun [171].
- ⇝ **Import checks**. UNUSEDIMPORTS looks for declared imports that are not used in the file. They clutter space and are misleading for readers. AVOIDSTARIMPORT checks that no generic import is used; specific import statements shall be used to help the reader grasp what is inherited in the file. REDUNDANTIMPORT looks for duplicates in imports, which uselessly takes up visual space.
- ⇝ **Equals Hash Code** (EQUALSHASHCODE). A class that overrides `equals()` shall also override `hashCode()`. A caller may use both methods without knowing that one of them has not been modified to fit the behaviour intended when modifying the other one (consistency in behaviour). It impacts reusability, reliability and fault tolerance.
- ⇝ **No Hard Coded Constant** (MAGICNUMBER). Hard coded constant or magic numbers shall not be used. Magic numbers are actual numbers like 27 that appear in the code that require the reader to figure out what 27 is being used for. One should consider using named constants for any number other than 0 and 1. Using meaningful names for constants instead of using magic numbers in the code makes

the code self-documenting, reducing the need for trailing comments. This rule is related to programming technics and changeability.

- ⇝ **Illegal Throws** (ILLEGALTHROWS) lists exceptions that are illegal or too generic; throwing java.lang.Error or java.lang.RuntimeException is considered to be almost never acceptable. In these cases a new exception type shall be defined to reflect the distinctive features of the throw.
- ⇝ **New line at end of file** (NEWLINEATENDOFFILE). Checks that there is a trailing newline at the end of each file. This is an ages-old convention, but many tools still complain when they find no trailing newline. Examples include `diff` or `cat` commands, and some SCM systems like CVS will print a warning when they encounter a file that does not end with a newline.
- ⇝ **Anonymous Inner Length** (ANONINNERLENGTH). Checks for long anonymous inner classes. For analysability reasons these should be defined as self-standing classes if they embed too much logic.
- ⇝ **Multiple String Literals** (MULTIPLESTRINGLITERALS) checks for multiple occurrences of the same string literal within a single file. It should be defined as a constant, both for reusability and changeability, so people can change the string at first shot without forgetting occurrences.

6.2.4 PMD

We selected 58 rules from the PMD 5.0.5 rule set. These are related to the following quality attributes: analysability (26 rules), maturity (31 rules), testability (13 rules), changeability (5 rules), and efficiency (5 rules). The full rule set is documented on the PMD web site: http://pmd.sourceforge.net/pmd-5.0.5/rules/.

- ⇝ **Jumbled incrementer** (JUMBLEDINCREMENTER) detects when a variable used in a structure is modified in a nested structure. One shall avoid jumbled loop incrementers – it is usually a mistake, and even when it is intended it is confusing for the reader.
- ⇝ **Return from finally block** (RETURNFROMFINALLYBLOCK). One shall avoid returning from a finally block since this can discard exceptions. This rule has an effect on analysability (developers will have trouble understanding where the exception comes from) and fault tolerance (the return method in the finally block may be stopping the exception that happened in the try block from propagating up even though it is not caught). It is related to the Cert ERR04-J rule.
- ⇝ **Unconditional if statements** (UNCONDITIONALIFSTATEMENT), **empty if statements** (EMPTYIFSTMT), **empty switch statements** (EMPTYSWITCHSTATEMENTS), **empty synchronized block** (EMPTYSYNCHRONIZEDBLOCK), and **empty while statements** (EMPTYWHILESTMT) are useless and clutter code. They impact analysability – developers will spend more time trying to understand what they are for, and they may have undesirable side effects. As for the empty while statements, if it is a timing loop then `Thread.sleep()` is better suited; if it does a

lot in the exit expression then it should be rewritten to make it clearer. All these are related to Cert MSC12-C.

- ⇝ **Empty catch blocks** (EMPTYCATCHBLOCK) are instances where an exception is caught, but nothing is done. In most circumstances an exception should either be acted on or reported, as examplified in the Cert ERR00-J rule. **Empty try blocks** (EMPTYTRYBLOCK) and **empty finally blocks** (EMPTYFINALLYBLOCK) serve no purpose and should be remove because they clutter the file's analysability.
- ⇝ The **God Class** (GODCLASS) rule detects the God Class design flaw using metrics. God classes do too many things, are very big and overly complex. They should be split apart to be more object-oriented. The rule uses the detection strategy described in *Object-Oriented Metrics in Practice* [118].
- ⇝ **Avoid Catching Throwable** (AVOIDCATCHINGTHROWABLE). Catching `Throwable` errors is not recommended since its scope is very broad. It includes runtime issues such as `OutOfMemoryError` that should be exposed and managed separately. It is related to the ERR07-J rule.
- ⇝ **Avoid Catching NPE** (AVOIDCATCHINGNPE) Code should never throw `NullPointerExceptions` under normal circumstances. A `catch` block for such an exception may hide the original error, causing other, more subtle problems later on. It is related to the Cert ERR08-J rule.
- ⇝ **Non Thread Safe Singleton** (NONTHREADSAFESINGLETON) Non-thread safe singletons can result in bad, unpredictable state changes. Static singletons are usually not needed as only a single instance exists anyway: they can be eliminated by instanciating the object directly. Other possible fixes are to synchronize the entire method or to use an initialize-on-demand holder class. See *Effective Java* [25], item 48 and Cert MSC07-J.

6.3 Projects

This section lists the projects selected for the data sets. They are open-source projects with a set of repositories that we are able to request through the data providers developed for that purpose (e.g. Subversion or Git for configuration management).

When the early versions of the data sets were generated, no language had been selected so projects both written in C/C++ and Java were retrieved and analysed. Later on, when we focused on the Java language, all the sources and information for the previous C projects were already there, so we managed to generate new versions of these data sets at the same time. However, Checkstyle and PMD analyses are not available for C/C++ projects, which considerably diminishes the volume of rules and practices verified on code – only rules from SQuORE remain.

Since many of the examples provided in this document rely on a subset of projects, we describe them more precisely here to provide insights into their history and *modus operandi*.

6.3.1 Apache Ant

The early history of Ant begins in the late nineties with the donation of the Tomcat software from Sun to Apache. From a specific build tool, it evolved steadily through Tomcat contributions to be more generic and usable. James Duncan Davidson announced the creation of the Ant project on the 13 January 2000, with its own mailing lists, source repository and issue tracking. There have been many versions since then: 8 major releases and 15 updates (minor releases). The data set ends in July 2012, and the last version officially released at that time is 1.8.4. Table 6.6 lists major releases of Ant with some characteristics of official builds as published. It should be noted that these characteristics may show inconsistencies with the data set, since the build process extracts and transforms a subset of the actual repository content.

Table 6.6: Major releases of Ant.

Date	Version	SLOC	Files	Functions
2000-07-18	1.1	9671	87	876
2000-10-24	1.2	18864	171	1809
2001-03-02	1.3	33347	385	3332
2001-09-03	1.4	43599	425	4277
2002-07-15	1.5	72315	716	6782
2003-12-18	1.6	97925	906	9453
2006-12-19	1.7	115973	1113	12036
2010-02-08	1.8	126230	1173	12964

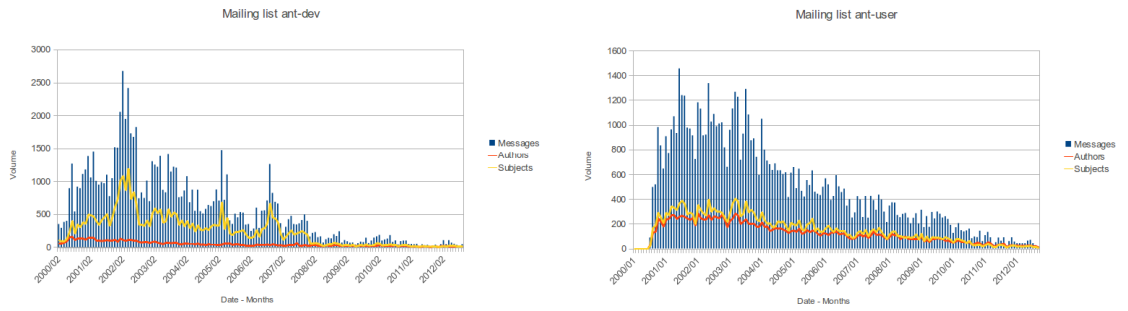


Figure 6.1: Ant mailing list activity.

Ant is arguably one of the most relevant examples of a successful open source project: from 2000 to 2003, the project attracted more than 30 developers whose efforts contributed to nominations for awards and to its recognition as a reliable, extendable and well-supported build standard for both the industry and the open source community. Figure 6.1 shows the mailing lists' activity on the data set time range. An interesting aspect of the Ant

project is the amount of information available on the lifespan of a project: from its early beginnings in 2000, activity had its climax around 2002-2003 and then decreased steadily. Although the project is actively maintained and still brings regular releases the list of new features is decreasing with the years. It is still hosted by the Apache Foundation, which is known to have a high interest in software product and process quality.

The releases considered for the data sets are reproduced in appendix B.1 page 255 with their dates of publishing.

6.3.2 Apache Httpd

Apache httpd is an open-source, fully-featured web server written in C. It is the most widely used server on the Web according to netcraft surveys[3] and has a strong reputation for robustness, reliability and scalability.

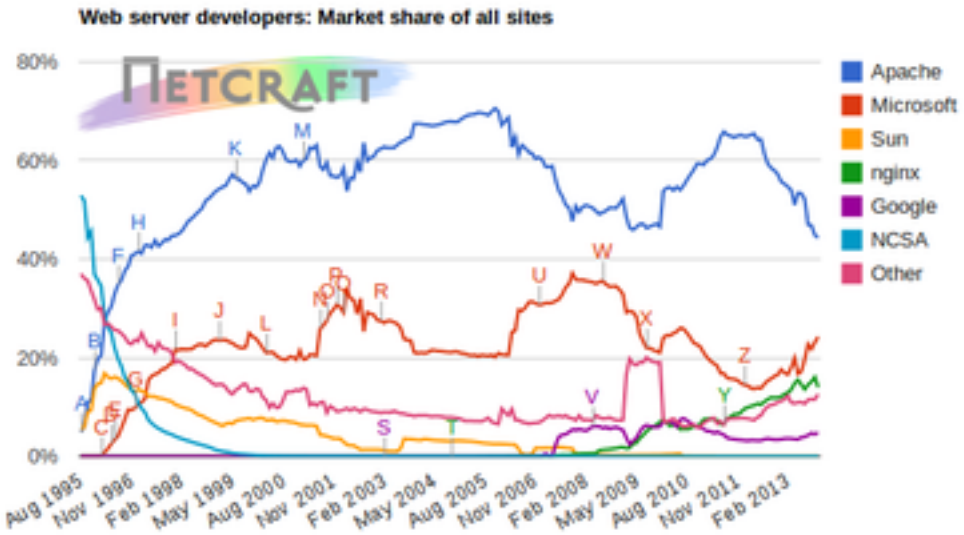


Figure 6.2: History of Apache httpd usage on the Web.

The Apache httpd project started in 1995 when Rob McCool left the NCSA (National Center for Supercomputing Applications), stalling the development of the NCSA web server – which was the pre-eminent server at that time. From the NCSA 1.3 code base, they collected bug fixes and worthwhile enhancements from webmasters around the world, and released the first official public release (0.6.2) in April 1995. It was quite a big hit, the community around the project grew steadily and development continued with new features (0.7.x) and a new architecture (0.8.8). After extensive testing, ports to a variety of obscure platforms, and a new set of documentation, Apache 1.0 went out on December, 1995.

[3]Netcraft maintains a landscape of the web, and most notably the servers used. See http://www.netcraft.com/survey/ for more information.

The Apache Software Foundation was founded in 1999 to provide organizational, legal, and financial support for the Apache httpd Server. Since then the foundation has greatly expanded the number of open-source projects falling under its umbrella, and is largely recognised for its quality concerns, both for the products it provides and the development process in use.

There have been three major branches of the Apache httpd server: Apache 1.3 was published on June 6, 1998 and reached its end of life with 1.3.42, on Feb. 2, 2010. The 2.0 series was first released in 2004 and ended with the 2.0.65 release July 2013. The full list of releases for the 2.0.x and 2.2.x series, along with their date of publication considered for the data set, is reproduced in appendix B.2 page 255.

6.3.3 Apache JMeter

The Apache JMeter desktop application is an open source, 100% pure Java application designed to load test functional behavior and measure performance of Java client/server applications. It is widely used both in the industry and in open-source projects and is an essential part of the classical tool suite for Java development [167].

Development of JMeter was initially started by Stefano Mazzocchi of the Apache Software Foundation, primarily to test the performance of Apache JServ (a project that has since been replaced by the Apache Tomcat project). JMeter steadily grew to enhance the GUI and to add functional-testing capabilities, and became a subproject of Apache Jakarta in 2001, with dedicated mailing lists and development resources. JMeter became eventually a top level Apache project in November 2011, with its own Project Management Commitee and a dedicated website.

The evolution data set starts on the 2004-09-13 and ends on the 2012-07-30, which constitutes 411 versions across 9 years. The release data set spans from 1.8.1, the first version published on the Apache web site on the 2003-02-03, to 2.10, the last version at the time of writing published on the 2013-10-21. This represents 22 versions spanning across more than 10 years.

6.3.4 Apache Subversion

Subversion is a fully-featured version control system written in C. In a period of 6 years (from 2004 to 2010), it has become a *de facto* standard for configuration management needs both in open-source projects and in the industry.

CollabNet founded the Subversion project in 2000 as an effort to write an open-source version-control system which operated much like CVS but which fixed the bugs and supplied some features missing in CVS. By 2001, Subversion had advanced

sufficiently to host its own source code[4]. In November 2009, Subversion was accepted into Apache Incubator: this marked the beginning of the process to become a standard top-level Apache project. It became a top-level Apache project on February 17, 2010.

The releases data set considers all published releases from 1.0.0 (the first release available in archives, published on the 2004-02-23) to 1.8.5 (the last release at the time of this writing, which was published on the 2013-11-25), which represents 71 versions across almost 10 years. The full list of releases with their date of publication considered for the data set is reproduced in appendix B.4 page 256.

6.4 Summary

In this chapter we presented how we managed to extract a full set of consistent metrics from various open-source software projects, carefully defining the metrics and violations retrieved and providing useful hints to understand the history and evolution of projects. Table 6.7 summarises all data sets that have been published at the time of writing, with the metrics extracted for them and the number of versions available.

Table 6.7: Summary of data sets.

			Code			SCM		Com.	Rules		
Project	Lang.	Extracts	Common	OO	Diff	Time	Total	Time	SQuORE	PMD	Checkstyle
Ant	Java	654	X	X	X	X	X	X	X	X	X
JMeter	Java	411	X	X	X	X	X	X	X	X	X
httpd 2.4	C	584	X		X	X	X	X	X		
Ant	Java	22	X	X	X		X		X	X	X
JMeter	Java	22	X	X	X		X		X	X	X
httpd 2.0	C	24	X		X		X		X		
httpd 2.2	C	20	X		X		X		X		
Subversion	C	71	X		X		X		X		
Epsilon	Java	2	X	X			X		X	X	X
Subversion	C	2	X				X		X		
Topcased Gendoc	Java	4	X	X			X		X	X	X
Topcased MM	Java	4	X	X			X		X	X	X
Topcased gPM	Java	2	X	X			X		X	X	X

Evolution data sets are weekly data sets describing the evolution of the project over a long range of time. They include temporal metrics (LADD, LMOD and LREM) and measures extracted from SCM and Communication repositories. They can be used

[4]See the full history of Subversion at http://svnbook.red-bean.com/en/1.7/svn.intro.whatis.html.

to analyse the evolution of metrics and practices during a long period. Since they are extracted from the project repository they really correspond to what developers work with, and give good insights on the practices really in use for day-to-day development. Since they provide equally-spaced, regular information on data, they provide a comfortable basis for time series analysis. It should be noted that when activity is low (e.g. at the beginning of the project), the repository may not evolve for more than a week. In such cases the retrieval of source code is identical for both dates.

Release data sets show characteristics of the product sources when delivered to the public, i.e. when they are considered good enough to be published. They provide useful information about what the *users* know about the product; some practices may be tolerated during development but tackled before release, so these may be hidden in the published source release. Furthermore, the source code tarball as published by the project may be different from the configuration management repository files: some of them may be generated, moved, or filtered during the build and release process. Temporal metrics are not included in releases, because their dates are not evenly distributed in time and a measure with a static number of weeks or months would be misleading: the last release may have happened weeks or months ago.

Version data sets only provide a few extracts or releases of the project. They may be used to investigate static software characteristics and practices, or relationships between the different levels of artefacts. Metrics linked to a previous state of the project, like the number of lines changed since the previous version or the number of commits during the last week, lack semantic context and become meaningless. These metrics have thus been discarded in the version data sets.

More projects are to be analysed now that the analysis process is clearly defined and fully automated. The architecture we setup allows us to quickly add new samples. A full weekly analysis of some Eclipse projects (including Papyrus, Mylyn, Sphinx, and Epsilon) is about to be published, and single snapshots are added regularly.

Part III

SQuORE Labs

> In theory, theory and practice are the same. In practice, they are not.
>
> Albert Einstein

Chapter 7

Working with the Eclipse foundation

The first SQuORE Lab was initiated in November 2012 as a collaboration with Airbus on the Topcased project, an open-source initiative targeted at embedded systems development and led by major actors of the aerospace industry. SQuORING Technologies had been in touch with Pierre Gaufillet[1] and Gaël Blondelle[2] to work on a quality model and a process for the various components of the project and we implemented it on the Maisqual server. In 2013 Topcased was included in the new Polarsys Industry Working Group, operating under the Eclipse foundation umbrella, and we were invited to participate in the Maturity Assessment task force.

For six months we participated in conference calls and meetings, really pushing the project forward, discussing issues with major actors from the industry (Airbus, Ericsson, Mercedes, BMW, among others) and the Eclipse Foundation, and we finally came up with a fully-blended quality model for Polarsys. We also worked with Wayne Beaton[3] during this time.

This chapter describes how we conducted this project, the steps we followed to propose a tailored quality model and its implementation. Section 7.1 gives a quick description of the context of the mining program. The data mining program was shaped according to the approach defined in Foundations: sections 7.2, 7.3 and 7.4 respectively declare the intent of the program, define quality requirements and describe the metrics to be used. Section 7.6 shows some results and section 7.7 summarises what we learned from this project and our contribution.

[1] Pierre Gaufillet is a software engineer at Airbus in Toulouse. Advocating open source strategy and model driven engineering, he is also involved in Topcased and OPEES initiatives.

[2] Gaël Blondelle is the french representative and community manager for Eclipse. He co-founded the Toulouse JUG in 2003, and is now heavily involved in the Polarsys IWG.

[3] Wayne Beaton is responsible for the Eclipse Project Management Infrastructure (PMI) and Common Build Infrastructure (CBI) projects.

7.1 The Eclipse Foundation

In the late 90's IBM started the development of a Java IDE, with a strong emphasis on modularity. Initial partners were reluctant to invest in a new, unproven technology, so in November 2001 IBM decided to adopt the open source licensing and operating model for this technology to increase exposure and accelerate adoption. The Eclipse consortium, composed of IBM and eight other organisations, was established to develop and promote the new platform.

The initial assumption was that the community would own the code and the commercial consortium would drive marketing and commercial relations. The community started to grow, but the project was still considered by many as an IBM-controlled effort. To circumvent this perception the Eclipse Foundation was created in 2003 as a non-profit organization with its own independent, paid professional staff, financed by member companies. This move has been a success. From its roots, and thanks to its highly modular architecture, hundreds of projects have flourished, bringing more tools, programming languages and features. Now Eclipse is the *de-facto* standard both for the open source community and industry. It is one of the largest open source software foundations, and a wonderful example of open source success.

An important characteristic of the Eclipse Foundation is its strong link to industry. Many companies evolve around the Eclipse ecosystem, shipping products based on the platform, offering consulting services, or simply using it on a large scale. This collaboration takes place at first in Industry Working Groups (IWG), which is a type of task force working on a specific subject, with clear requirements, milestones, and well-defined outputs.

The Polarsys Industry Working Group was started in 2012[4] to address concerns specific to the development of critical embedded systems. Namely, its objectives are to:

- ⇝ Provide Very Long Term Support – up to 10 and 75 years.
- ⇝ Provide certification to ease the tools qualification in complex certification processes.
- ⇝ Develop the ecosystem of Eclipse tools for Critical Embedded Systems.

7.2 Declaration of intent

The objectives of the conducted program have been identified as follows:

- ⇝ *Assess project maturity* as defined by the Polarsys working group stated goals. Is the project ready for large-scale deployments in high-constraint software organisations? Does it conform to the Eclipse foundation recommendations?
- ⇝ *Help teams develop better software* regarding the quality requirements defined. Propose guidance to improve the management, process and product of projects.

[4]See www.polarsys.org.

⇝ *Establish the foundation for a global agreement on quality* conforming to the Eclipse way. A framework to collaborate on the semantics of quality is the first step to a better understanding and awareness of these concerns in the Eclipse community.

7.3 Quality requirements

The first thing we did was to establish the quality requirements of the Eclipse foundation, and more specifically in the context of embedded systems for Polarsys. We gathered a set of concerns from the foundation's stated guidelines, recommendations and rules, and discussed with the Polarsys team the establishment of criteria for *good-citizen* projects.

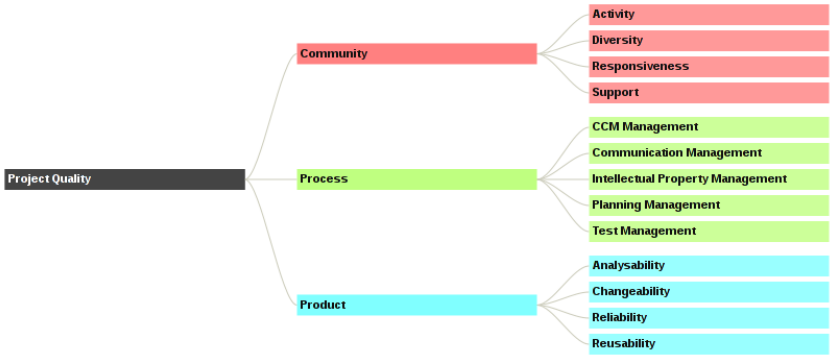


Figure 7.1: Proposed Eclipse quality model.

Because of their open-source nature, Eclipse components have strong maintainability concerns, which are actually re-enforced for the Polarsys long-term support requirements. In the spirit of the ISO/IEC 9126 standard[94] these concerns are decomposed in **analysability**, **changeability** and **reusability**. Another quality requirement is **reliability**: Eclipse components are meant to be used in bundles or stacks of software, and the failure of a single component may threaten the whole application. For an industrial deployment over thousands of people in worldwide locations this may have serious repercussions.

The Eclipse foundation has a strong interest in IP management and predictability of outputs. Projects are classified into phases, from proposal (defining the project) to incubating (growing the project) and mature (ensuring project maintenance and vitality). Different constraints are imposed on each phase: e.g. setting up reviews and milestones, publishing a roadmap, or documenting APIs backward compatibility. In the first iteration of the quality model only **IP management** and **planning management** are addressed.

Communities really lie at the heart of the Eclipse way. The Eclipse manifesto defines three communities: developers, adopters, and users. Projects must constitute, then grow and nurture their communities regarding their **activity**, **diversity**, **responsiveness** and **support** capability. In the context of our program we will mainly focus on developer and user communities.

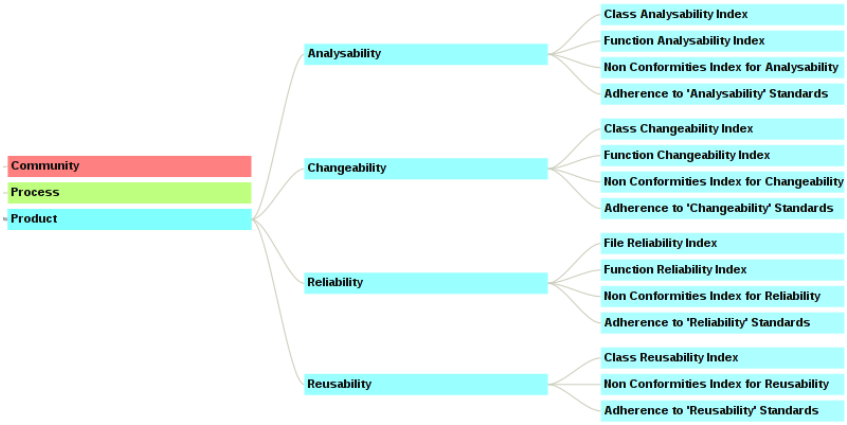


Figure 7.2: Proposed Eclipse quality model: Product.

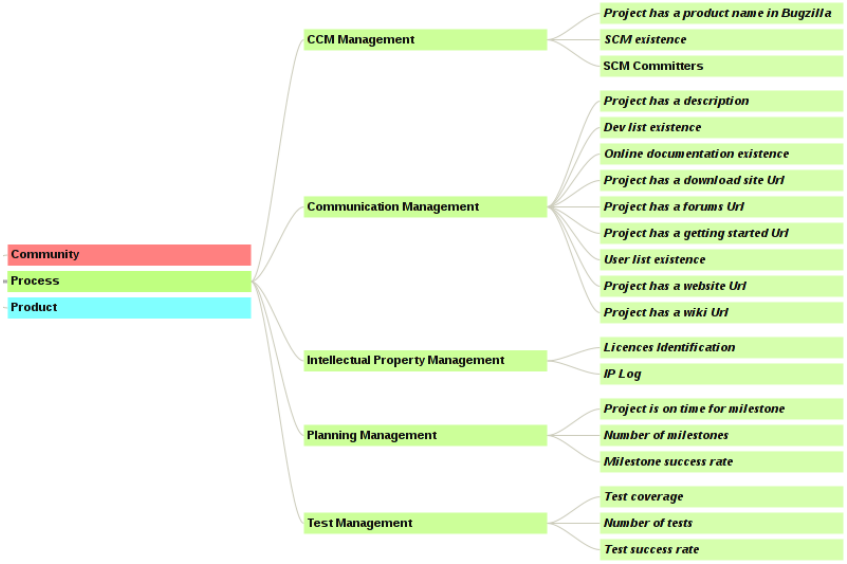


Figure 7.3: Proposed Eclipse quality model: Process.

7.4 Metrics identification

Source code Source code is extracted from Subversion or Git repositories and analysed with SQuORE [11]. Static analysis was preferred because automatically compiling the different projects was unsafe and unreliable. Further analysis may be integrated with continuous build servers to run dynamic analysis as well. *Metrics* include common established measures like line-counting (comment lines, source lines, effective lines, statements), control-flow complexity (cyclomatic complexity, maximum level of nesting, number of paths, number of control-flow tokens) or Halstead's software science metrics (number of distinct and total operators and operands). *Findings* include violations from SQuORE, Checkstyle and PMD. All of these have established lists of rules that are mapped to practices and classified according to their impact (e.g. reliability, readability).

Figure 7.4: Proposed Eclipse quality model: Community.

SCM metadata A data provider was developed to retrieve software configuration metadata from the two software configuration management systems used inside Eclipse, Subversion and Git. The metrics identified are the following: number of commits, committers, committed files, and fix-related commits. All measures are computed for the last week, last month and last three months as well as in total. This enables us to identify recent evolution of the project by giving more weight to the most recent modifications.

Communication Communication channels used at Eclipse are NNTP news, mailing lists and web forums. All of these can be mapped and converted to the mbox format, which we then parse to extract the metrics we need. Metrics retrieved are the number of threads, distinct authors, the response ratio and median time to first answer. Metrics are gathered for both user and developer communities, on different time frames to better grasp the dynamics of the project evolution (one week, one month, three months).

Process repository The Eclipse foundation has recently started an initiative to centralise and publish real-time process-related information. This information is available through a public API returning JSON and XML data about every Eclipse component. At that time only a few fields were actually fed by projects or robots, and although we worked with Wayne Beaton on the accessibility of metrics it was decided to postpone their utilisation to the next iteration. Here are some examples of the metrics defined in the JSON file: description of the basic characteristics of the project, the number and status of milestones and reviews, their scope in terms of Bugzilla change requests. The XML API provides valuable information about intellectual property logs coverage.

7.5 From metrics to quality attributes

The next step in the mining process is to define the relationships between attributes of quality and the metrics defined. As pointed out earlier (cf. section 2.3), there is no single definitive truth here and the people involved in the process have to be in agreement and

the work published. As for the product quality part (cf. figure 7.2), all sub-characteristics have been constructed with the same structure:

1. One or more technical debt indexes computed at different levels (e.g. File, Class, Function *Characteristic* Index). As an example, reusability is often considered at the class level, hence a *Class Reusability Index*.
2. A number of violations to rules (a.k.a. Non conformities Index for *Characteristic*) that are linked to the quality characteristic (e.g. fault tolerance, analysability).
3. A ratio of acquired practices (a.k.a. Adherence to *Characteristic* Standards), which is the number of rules that are never violated divided by the total number of rules for the quality characteristic.

The process part is loosely mapped to process-related standards such as the CMMi. Sub-characteristics are linked to metrics from the process repository (e.g. boolean values such as *Is there a wiki?* or *Is there a download site?*, milestones or reviews), IP logs, test results, and communication metrics. The exact decomposition is shown on figure 7.3.

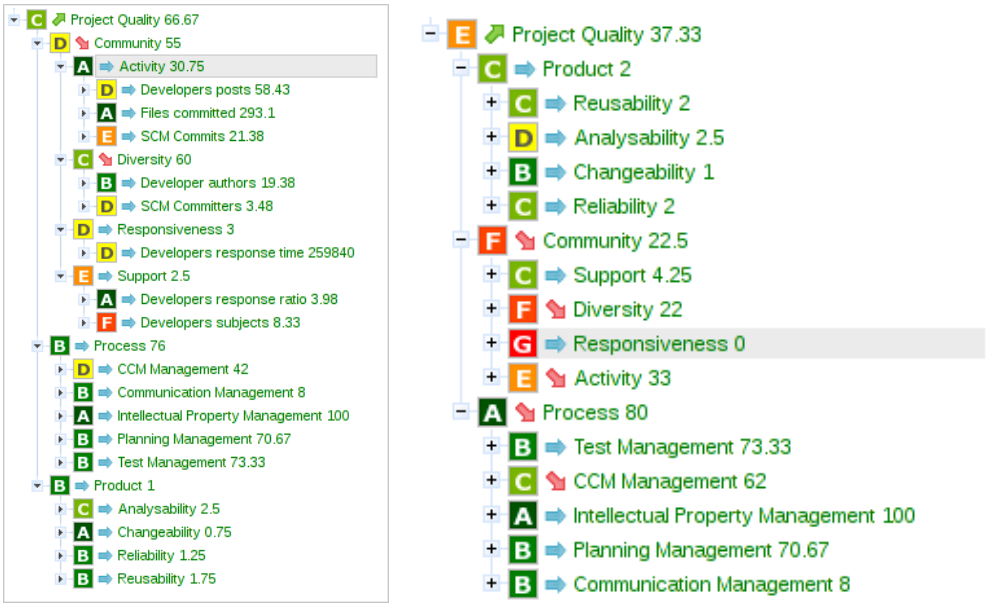
The community quality part is tied to SCM metrics (e.g. number of commits, or committers) and communication metrics (e.g. number of distinct authors, ratio of responses, or median time to first answer). The exact decomposition is shown on figure 7.4.

7.6 Results

The first output of this work is the **Eclipse quality model** depicted in figure 7.1 on page 129. We made a proposition to the working group and from this proposition, the members of the group discussed and improved the quality attributes and model. They were also able to better visualise the decomposition of quality and relationships between quality attributes and metrics. This led to new ideas, constructive feedback and fruitful discussion, and the group could eventually agree on a common understanding of quality and build upon it.

The prototype for this first iteration of the quality program has been implemented with SQUORE[11], which provides several graphical and textual means to deliver the information:

- ⇝ A *quality tree* that lists the quality attributes and shows the non-conformance to the identified requirements (cf figure 7.5a). The evolution or trend since the last analysis is also depicted for every quality attribute, which enables one to immediately identify the areas that have been improved or shattered.
- ⇝ *Action lists*, classified according to the different concerns identified: overly complex or untestable files, naming conventions violations, or a refactoring wish list. An example is shown on figure 7.6b.
- ⇝ *Clear coloured graphics* that immediately illustrate specific concepts as shown on figure 7.6a. This proves to be especially useful when the characteristics of an incriminated file make it significantly different than the average so it can be identified at first sight, for example for highly complex or unstable files.



(a) Epsilon quality tree.

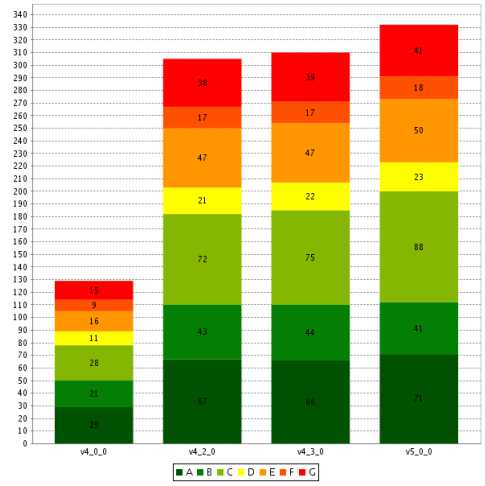
(b) Sphinx quality tree.

Figure 7.5: Examples of results for Polarsys.

From the **maturity assessment** perspective, the quality model tree as shown in figure 7.5 allows identifying quickly the requirements that are not fulfilled by the project. A good example is the Sphinx project, which has a fairly good product and process quality, but receives low marks on the community aspects. When investigating, we found out there is a single company behind all commits and they don't respond to the mailing list, which is a serious concern for the project's maintainability and overall durability. This is something hard to identify without the indicators that have been setup.

The other objective of the project was to propose **pragmatic advice on quality improvement**, both at the product and process levels. This is achieved through several means:

- ⇝ Lists of violations on the defined practices (figure 7.6b), with their description, precise location (line number in file), impact, and possible solutions. The delta with the last analysis is also shown to highlight what has been recently introduced.
- ⇝ Action items list dedicated to some important violations or specific areas of improvement. The model typically proposes to act first on the practices that are almost acquired, e.g. with a low number of violations, before acting on practices that demand a huge effort to comply with. The intent is both to secure the acquired practices and to not discourage developers with an extensive (and useless) list of thousands of warnings.
- ⇝ The decomposition in artefacts (application > folders > files > classes > functions) allows to focus on the parts of the product that have a lower quality, both to warn developers on some modifications and for refactoring.



(a) Evolution of Gendoc files notations.

Method Returns Internal Array	3	+3	PMD
getArguments()	1	+1	
Line: 100			
getDependentObjects()	1	+1	
getQualifiers()	1	+1	
Missing Break In Switch	1	0	PMD
parseValue(CharacterIterator)	1	0	
Missing Default	2	+1	SQuORE
parseValue(CharacterIterator)	1	0	
serviceChanged(ServiceEvent)	1	+1	

(b) Eclipse E4 findings.

Figure 7.6: Examples of results for Polarsys.

⇝ Graphics that allow to grasp some aspects of the software. Examples are the repartition of file evaluations across different versions to show their trend (cf. figure 7.6a) or plots showing the instability of files (number of commits and fixes).

7.7 Summary

This was a very instructional project, which benefited all actors involved in the process. We achieved the following objectives during this prototype:

⇝ Lay down the foundations of a constructive discussion on software quality and sow the seeds of a common agreement on its meaning and ways to improve it. A first, fully-featured version of the quality model gathered enough agreement to be recognised by all parts. The quality model would then be enhanced during the next iterations, with new quality attributes and new metrics.

⇝ Define a wide spectrum of metrics to address the different quality attributes, from code to process and community. Practices and metrics have been publicly defined, their retrieval process has been automated and a common agreement has been established as for their links to the various quality attributes.

⇝ Build a functional proof of concept: the prototype we installed on the Maisqual server demonstrated the feasibility and usability of such a program.

A licensing problem arose in July, 2013 when Polarsys decided that all tools involved in the working group had to be open source. Unfortunately, the Eclipse foundation and SQuORING Technologies could not come to an accord and the involvement in Polarsys stopped in August, 2013. Before leaving we described in the Polarsys wiki the method, metrics and quality model for the community to reuse and enhance. Gaël Blondelle and

Victoria Torres (from the Universitat Politècnica de València, Spain) are now working on the next iteration of the prototype, continuing the early steps we began.

From the SQuORING Technologies perspective, this work brought two substantial benefits: the marketing service could use it as a communication campaign, and the experience gathered on the process- and community- aspects has been integrated in the SQuORE product models. This eventually gave birth to another process-oriented assessment project on GitHub.

From the Maisqual perspective, this was a successful implementation of the approach we defined earlier, in an industrial context (see section 5.2). People were happy to build something new, conscious of the different constraints and pitfalls, and our collaboration was very profitable. We could produce both a quality model and a prototype that were considered satisfying by stakeholders and users.

Finally the goals, definitions, metrics and resulting quality model were presented at the EclipseCon France conference held in May, 2013 in Toulouse and received good and constructive feedback[5].

> “ Squoring proposal for the Eclipse Quality Model at #eclipsecon sounds great! Hope it will be adopted by the whole eclipse community. ”
>
> Fabien Toral, Twitted during EclipseCon France.

[5]Slides for the EclipseCon France 2013 session are still available on the EclipseCon web site: www.eclipsecon.org/france2013/sessions/software-quality-eclipse-way-and-beyond.

Chapter 8

Outliers detection

One of the key features of SQuORE resides in *Action Items*, which are lists of artefacts with specific characteristics: e.g. very complex functions that may be untestable, or files with too many rule violations. They are very flexible in their definitions and can be customised for the specific needs of customers, providing powerful clues for maintenance and decision making. As of today they rely on static thresholds, which have two drawbacks: static values do not fit all situations (critical embedded systems have different constraints than desktop applications) and they frequently output a huge number of artefacts (too many for incremental quality improvement).

The intent of this work on outliers is to replace the statically triggered action items with dynamic techniques that auto-adapt the detection thresholds to the project characteristics. The first section defines the requirements of this project. Section 8.2 details the statistical techniques we use for the detection of outliers. The actual implementation of the process and the precise description of the outliers we are looking for is shown in section 8.3. Results are discussed in section 8.4 and conclusions expanded in section 8.5.

8.1 Requirements: what are we looking for?

Outliers have a specific meaning in the semantic context of software engineering. Before trying to find *something special*, we first need to define what kind of information and specificity we are looking for (e.g. files that have a control flow complexity that is *significantly higher* than the average files in the project). We also need to identify the characteristics and the detection technique to trigger them automatically with a good level of confidence.

We intend to detect in the first round of trials the following three types of outliers:

- *Unreadable code*: files or functions that may be difficult to read, understand and modify by other developers – obfuscated code is an extreme case of an unreadable file. We define two levels of unreadability: medium (code is difficult to read) and high (obfuscated code). For this we rely on a measure of density of operands and

operators, as output by Halstead metrics. SQuORE does not propose any mechanism to detect them as of today.

- ⇝ *Untestable functions* that have a huge control-flow complexity. This induces a large number of test cases to obtain a fair coverage and increases the overall complexity of testing.
- ⇝ *Cloned code inside a function* is detected through frequency of vocabulary and control-flow complexity. Cloning is bad for maintainability: any modification (bug correction or new feature) in one part should be implemented in the other cloned parts as well.

A strong requirement is the number of artefacts output by the method; since most of SQuORE models deal with incremental quality improvement, we are looking for a rather small number of artefacts to focus on. Doing this prevents developers from being overwhelmed by too many issues: they can work on small increments and steadily increase their practices and the code quality.

The validation of results is achieved via two means. Firstly, the SQuORE product comes out-of-the-box with a pre-defined set of action items, which we will use to validate our own findings when available. Since static thresholds are used however, we don't expect to get the same number of artefacts, but rather an intersecting set with a low number of items. Secondly, when no action item exists for a given purpose or if we get a different set of artefacts, we manually check them to ensure they suit the search.

8.2 Statistical methods

We used and combined three outliers detection techniques: boxplots (as described in Laurikkala et al. [120] and an adjusted version for skewed distributions from Roosseeuw [153]), clustering-based and tail-cutting, a selection method that we setup for our needs.

8.2.1 Simple tail-cutting

As presented in section 4.2.2 many metric distributions have long tails. If we use the definition for outliers of Lincke et al. [126] and select artefacts with metrics that have values greater than 85% of their maximum, we usually get a fairly small number of artefacts with especially high values. The figure on the right illustrates the selection method on one of the distribution function we met.

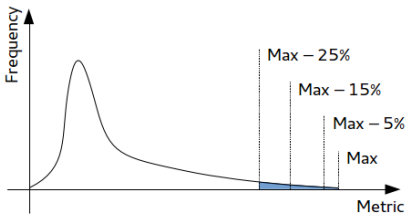


Table 8.1 shows the maximum value of a sample of metrics and their 95% (Max - 5%), 85% (Max - 15%), and 75% (Max - 25%) thresholds with the number of artefacts that fit in the range. Projects analysed are Ant 1.7 (1113

files), and various extracts from SVN: JMeter (768 files), Epsilon (777 files) and Sphinx (467 files).

Table 8.1: Values of different thresholds (75%, 85%, 95% and maximum value for metrics) for tail-cutting, with the number of outliers selected in bold.

Project	Metrics	75%	85%	95%	Max value (100%)
Ant	SLOC	1160 (**3**)	1315 (**2**)	1469 (**1**)	1546
	NCC	507 (**3**)	575 (**2**)	643 (**2**)	676
	VG	267 (**3**)	303 (**2**)	339 (**1**)	356
JMeter	SLOC	729 (**3**)	827 (**1**)	924 (**1**)	972
	NCC	502 (**5**)	569 (**3**)	636 (**1**)	669
	VG	129 (**4**)	147 (**2**)	164 (**1**)	172
Epsilon	SLOC	3812 (**6**)	4320 (**6**)	4828 (**6**)	5082
	NCC	7396 (**6**)	8382 (**6**)	9368 (**2**)	9861
	VG	813 (**6**)	922 (**6**)	1030 (**6**)	1084
Sphinx	SLOC	897 (**5**)	1016 (**4**)	1136 (**1**)	1195
	NCC	801 (**2**)	907 (**2**)	1014 (**1**)	1067
	VG	233 (**2**)	264 (**2**)	295 (**2**)	310

The main advantage of this method is it permits us to find points without knowing the threshold value that makes them peculiar: e.g. for the cyclomatic complexity, the recommended and argued value of 10 is replaced by a value that makes those artefacts *significantly* more complex than the vast majority of artefacts. This technique is a lot more selective than boxplots and thus produces less artefacts with higher values.

8.2.2 Boxplots

One of the simplest statistical outlier detection methods was introduced by Laurikkala et al. [120] and uses informal box plots to pinpoint outliers on individual variables.

We applied the idea of series union and intersection[1] from Cook et al. [39] to outliers detection, a data point being a multi-dimensional outlier if many of its variables are themselves outliers. The rationale is that a file with unusually high complexity, size, number of commits, and a very bad comment ratio *is* an outlier, because of its maintainability issues and development history. We first tried to apply this method to the full set of available metrics, by sorting the components according to the number of variables that are detected as univariate outliers. The drawback of this accumulative technique is its dependence on the selected set of variables: highly correlated metrics like line-counting measures will quickly trigger big components even if all of their other variables show

[1] In [39] authors use univariate series unions and intersections to better visualise and understand data repartition.

standard values. Having a limited set of orthogonal metrics with low correlation is needed to balance the available information.

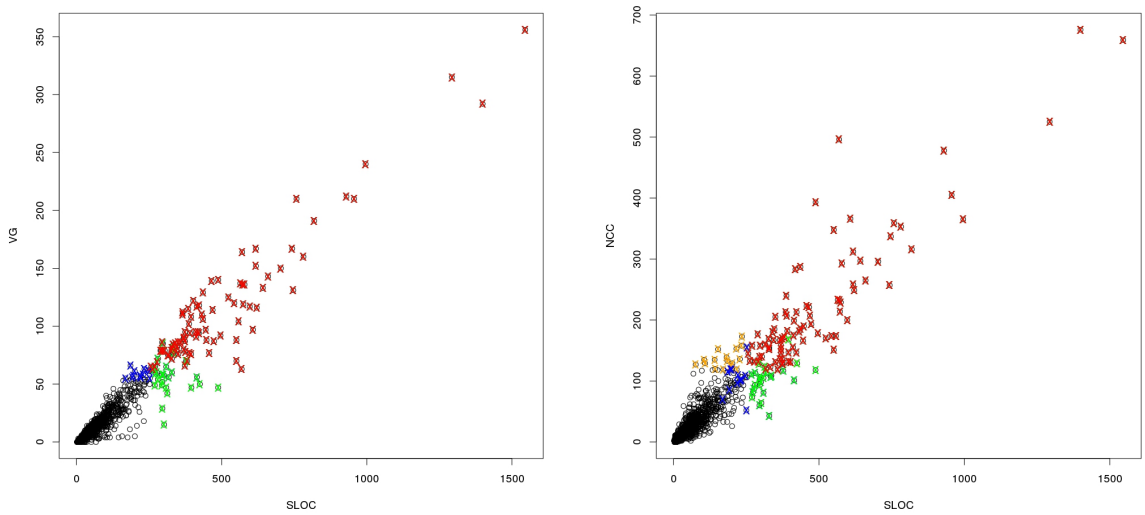


Figure 8.1: Combined univariate boxplot outliers on metrics for Ant 1.7: SLOC, VG (left) and SLOC, VG, NCC (right).

Figure 8.1 (left) shows the SLOC and VG metrics on files for the Ant 1.7 release, with outliers highlighted in blue for VG (111 items), green for SLOC (110 items), and red for the intersection of both (99 items). The same technique was applied with three metrics (SLOC, VG, NCC) and their intersection in the figure on the right. There are 108 NCC outliers (plotted in orange) and the intersecting set (plotted in red) has 85 outliers. Intersecting artefacts accumulate outstanding values on the three selected metrics. Because of the underlying distribution of measures however the classical boxplot shows too many outliers – or at least too many for practical improvement. We used a more robust boxplots algorithm: adjBoxStats [153]. This algorithm is targeted at skewed distributions[2] and gave better results with fewer, more atypical artefacts.

8.2.3 Clustering

Clustering techniques allow us to find categories of similar data in a set. If a data point cannot fit into a cluster, or if it is in a small cluster (i.e. there are very few items that have these similar characteristics), then it can be considered an outlier [23, 165]. In this situation, we want to use clustering algorithms that produce unbalanced trees rather than evenly distributed sets. Typical clustering algorithms used for outliers detection are k-means [98, 197] and hierarchical [73, 3]. We used the latter because of the very small

[2]Extremes of the upper and whiskers of the adjusted boxplots are computed using the medcouple, a robust measure of skewness.

Table 8.2: Cardinality of clusters for hierarchical clustering for Ant 1.7 files (7 clusters).

Euclidean distance							
Method used	Cl1	Cl2	Cl3	Cl4	Cl5	Cl6	Cl7
Ward	226	334	31	97	232	145	48
Average	1006	6	19	76	3	1	2
Single	1105	3	1	1	1	1	1
Complete	998	17	67	3	21	3	4
McQuitty	1034	6	57	10	3	1	2
Median	1034	6	66	3	1	2	1
Centroid	940	24	142	3	1	2	1
Manhattan distance							
Method used	Cl1	Cl2	Cl3	Cl4	Cl5	Cl6	Cl7
Ward	404	22	87	276	62	196	66
Average	943	21	139	4	3	1	2
Single	1105	3	1	1	1	1	1
Complete	987	17	52	3	47	3	4
McQuitty	1031	12	60	3	1	4	2
Median	984	6	116	3	1	2	1
Centroid	942	24	140	3	1	2	1

clusters it produces and its fast implementation in R [147]. Clustering techniques have two advantages: first they don't need to be supervised, and second they can be used in an incremental mode: after learning the clusters, new points can be inserted into the set and tested for outliers [95].

We applied hierarchical clustering to file measures with different distances and agglomeration methods. On the Apache Ant 1.7 source release (1113 files) we got the repartition of artefacts shown in table 8.2 with Euclidean and Manhattan distances. Linkage methods investigated are Ward, Average, Single, Complete, McQuitty, Median and Centroid. The Manhattan distance produces more clusters with fewer elements than the Euclidean distance. The aggregation method used also has a great impact: the ward method draws more evenly distributed clusters while the single method consistently gives many clusters with only a few individuals.

8.2.4 A note on metrics selection

Metrics selection greatly influences the results: they have to be carefully selected according to the target of the analysis. The metrics used in traditional action items are selected by software engineering experts and usually show a great relevance. We still have to check them for redundancy and pertinence, sometimes removing some measures that do not bring any added value. Using too many metrics usually gives poor or unusable results

because of their relationships, the diversity of information they deliver and our ability to capture this information and its ramifications.

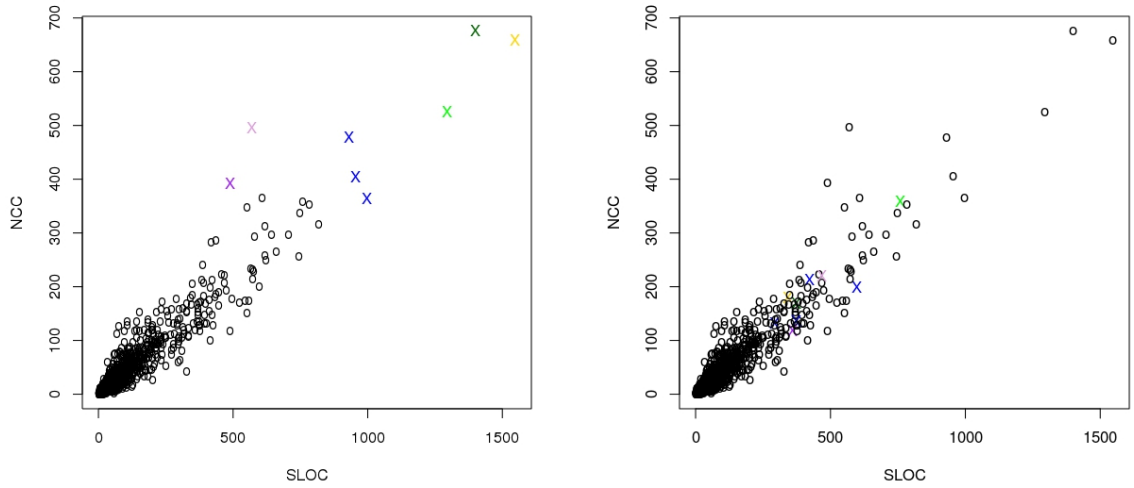


Figure 8.2: Outliers in clusters: different sets of metrics select different outliers.

As an example, the same detection technique (i.e. same distance and linkage method) was applied to different sets of metrics in figure 8.2: outliers are completely different depending on the metric sets used, and are even difficult to visually identify. We got the best results by first selecting the metrics identified by our experience with customers, and then challenging them by removing some (e.g. correlated) measures, or adding new dimensions that may have had an impact on the practice we wanted to target. Results were manually checked: plots allow to quickly dismiss bad results and lists of items allow fine-grained verification.

8.3 Implementation

8.3.1 Knitr documents

The first trials had been run through a dedicated Knitr document. It featured different outliers detection techniques, combined and applied on different sets of metrics and different projects. The different techniques implemented were:

Boxplots outliers detection, both univariate and multivariate, on common metrics (e.g. VG, CFT or NCC). Graphs are plotted to highlight the outliers visually (see appendix C.1). A second part lists the top 10 outliers on each metric.

Univariate boxplots sorted: files with a higher number of outliering metrics are highlighted and displayed as a list, with the metrics that are considered as outliers for them. An example is provided in appendix C.1.

Hierarchical clustering, based on Euclidean and Manhattan distances. Various agregation methods and number of clusters are tried and output as tables and graphs. An example of a run on the Papyrus project is shown in appendices C.1 at page 266 and C.1 at page 267.

Local Outlier Factor [30] implemented in `DMwR::lofactor` [177] is applied both on a set of single metrics and on a multivariate set.

PCOut [65], which is a fast algorithm for identifying multivariate outliers in high-dimensional and/or large datasets. Based on the robustly sphered data, semi-robust principal components are computed which are needed for determining distances for each observation. Separate weights for location and scatter outliers are computed based on these distances. The combined weights are then used for outlier identification. Without optimisation it produces very high rates of outliers: 95 out of 784 files (12%) for JMeter (2005-03-07 extract), and 1285 out of 5012 files (25%) for Papyrus (2010-01-18 extract).

covMCD [153] from V. Todorov [176] is another outliers detection algorithm developed for large data sets. It computes a robust multivariate location and scale estimate using the fast MCD (Minimum Covariance Determinant) estimator. It is well fitted for normal data, but unfortunately in the samples we ran the number of outliers is far too high to be usable in our context: 225 out of 784 files (28%) for JMeter, 1873 out of 5012 files (37%) for Papyrus.

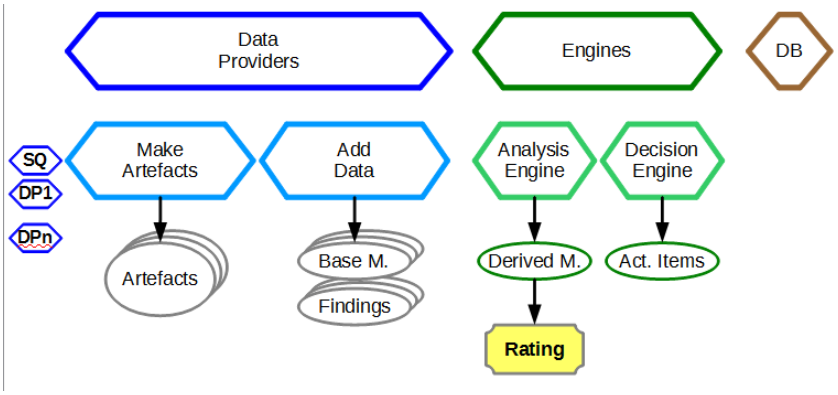
For each technique, results are both displayed through graphics (to grasp the essence of the specificities of selected artefacts) and lists of outliers. The document is long (112 pages as an average, although it depends on the number of outliers listed) and could not be included in its entirety in the appendix. However a few pages have been extracted from the Papyrus analysis to demonstrate specific concerns in appendix C.1:

- ⇝ The list of figures and tables – lists of outliers are output with a different environment and do not appear there.
- ⇝ The basic summary of metrics showing minimum, maximum, mean, variance and number of modes for all measures.
- ⇝ An example of scatterplot for 3-metrics combination of boxplots outliers.
- ⇝ The table of metrics with the count of outliers and percent of data sets, for the univariate sorting method.
- ⇝ An example of plots for the hierarchical clustering, with a dendogram of artefacts for the Euclidean distance and Ward agregation method, and a table summarising the number of elements in each cluster for the different agregation methods.
- ⇝ Another example of hierarchical clustering with the Manhattan distance and the single agregation method. Compared to the previous sample, the ward method separates the artefacts in several equilibrated clusters while the single method designs a huge cluster with many adjacent, small clusters.

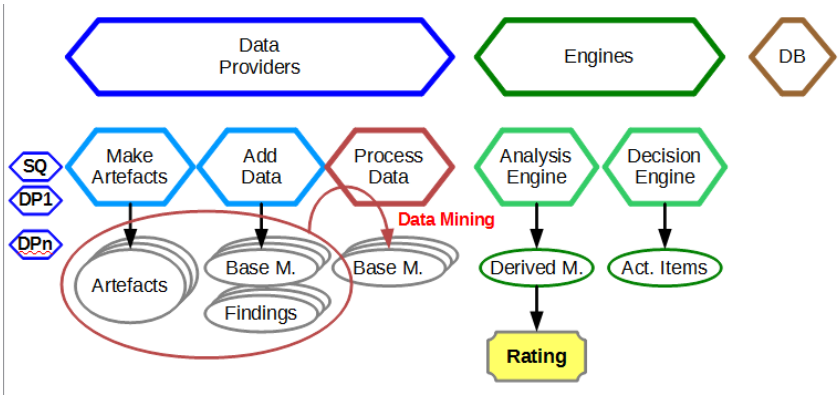
The prototype was considered performant enough to allow for implementation in the SQuORE product.

8.3.2 Integration in SQuORE

The integration in SQuORE implied a few changes in the engine itself, since some inputs needed for the outliers detection mechanism were available only in a later stage in the process. A SQuORE analysis first executes the SQuORE analyser, then runs the data providers to attach information to the artefacts, runs the computations for the derived data and then computes action items. We had to slightly modify the execution flow to implement back loops to use the findings at the `AddData` stage and provide outliers detection information to the engine, as shown in figure 8.2.



(a) SQuORE architecture before OD.



(b) SQuORE architecture with OD.

Figure 8.2: Modifications in the architecture of SQuORE for outliers detection.

We implemented a modular architecture to allow easy integration of new outliers types. Outliers are detected using R algorithms provided as external scripts which are automatically executed by the engine when they are put in the data provider directory. As for the data provider itself, the sequence of actions is as follows:

1. Metrics for files and functions output by the SQuORE analyser are read. Some basic sanitary checks are performed as for the availability of metrics and consistency of input files.
2. The `od_init.R` file, located in the data provider directory, is executed as a preamble to the outliers detection. It computes some derived measures which we do not have at that time in SQuORE such as VG, VOCF, or R_DOPD (DOPD/SLOC).
3. Each type of outlier has a specific file in the same directory. These are automatically read and executed in a R session when the data provider is run. Artefacts are tagged with a measure of their outlierness and results can be used afterwards in graphs and tables in the SQuORE interface.

8.3.3 R modular scripts

A naming convention was defined for the name and type of artefacts of the detection technique used. The files used to configure detection have the following format:

`od_<type>_<name>.conf`

Where <type> is one of c("file", "function") and <name> is the descriptor of the outliers found. In the quality model used afterwards the outliers have a measure IS_OUTLIER_<name> set to 1. Hence it is very easy to add a new type of outlier and its associated detection technique. The goal is to allow new types and implementations of outliers in specific contexts, using specific metrics, quickly and easily – one does not even need to restart SQuORE to take modifications into account.

The structure of these files is twofold. We first define the metrics we want to use in our detection mechanic, which will be dynamically generated and made available to the R script through command-line parameters upon execution. The second part gives the R code that actually computes the outliers and returns them stored in an array. The following example shows a typical sample that simply returns the first 10 lines of the data set as outliers (purposely named `outs`), and needs three metrics: VG, SLOC, and NCC.

```
= START METRICS =
VG
SLOC
NCC
= END METRICS =

= START R =
outs <- project[1:10,1]
= END R =
```

8.4 Use cases

Once a comfortable setup had been designed and implemented, we could experiment with our data easily (see next paragraph for the data sets considered). We defined three outliers detection techniques for the use cases defined in Section 8.1: hard to read files and functions, untestables functions, and code cloning inside a function.

We conducted the tests and the validation phase on a set of 8 open-source projects of various sizes. 4 are written in C and 4 are written in Java. Results are reproduced in the tables below, and elements of comparison are given when available – i.e. when existing action items provided static threshold in the SQuORE base product.

8.4.1 Hard to read files and functions

Inputs

This type of outlier has no equivalence in the SQuORE product. We discovered that code with a high density of operands and operators – as shown by Halstead's DOPD, DOPT, TOPD, TOPT measures – is more difficult to read and understand, at least when it is compressed on a few lines. For this purpose four derived density measures are computed in the initialisation file (`od_init.R`) and passed on to the script:

R_DOPD is the number of distinct operands divided by the number of source lines of code.

R_DOPT is the number of distinct operators divided by the number of source lines of code.

R_TOPD is the total number of operands divided by the number of source lines of code.

R_TOPT is the total number of operators divided by the number of source lines of code.

The SLOC metric is also used as an input to the script in order to remove files that are too small. The rationale is that people reading this code need to track more items (i.e. high values for TOPD and TOPT), in a more complex control flow (i.e. high values for DOPD and DOPT) to grasp the algorithm. An example of an obfuscated file that can be easily identified with such high ratios is provided below:

```
#  include<stdio.h>//  .IOCCC                                          Fluid-  #
#  include <unistd.h>  //2012                                          _Sim!_  #
#  include<complex.h>  //||||                    ,____.               IOCCC-  #
#  define              h for(                    x=011;               2012/*  #
#  */-1>x              ++;)b[                    x]//-'               winner  #
#  define              f(p,e)                                         for(/*  #
#  */p=a;              e,p<r;                                        p+=5)//  #
#  define              z(e,i)                                        f(p,p/*  #
## */[i]=e)f(q,w=cabs  (d=*p-  *q)/2-     1)if(0  <(x=1-      w))p[i]+=w*/// ##
   double complex a [  97687]  ,*p,*q     ,*r=a,  w=0,d;    int x,y;char b/* ##
## */[6856]="\x1b[2J"  "\x1b"  "[1;1H     ", *o=  b, *t;   int main   (){/** ##
## */for(              ;0<(x=  getc (     stdin)  );)w=x  >10?32<     x?4[/* ##
## */*r++              =w,r]=  w+1,*r     =r[5]=  x==35,  r+=9:0      ,w-I/* ##
## */:(x=              w+2);;  for(;;     puts(o  ),o=b+  4){z(p      [1]*/* ##
## */9,2)              w;z(G,  3)(d*(     3-p[2]  -q[2])  *P+p[4      ]*V-/* ##
## */q[4]              *V)/p[  2];h=0     ;f(p,(  t=b+10  +(x=*p      *I)+/* ##
## */80*(              y=*p/2  ),*p+=p    [4]+=p  [3]/10  *!p[1])     )x=0/* ##
## */ <=x              &&x<79   &&0<=y&&y<23?1[1  [*t|=8   ,t]|=4,t+=80]=1/* ##
## */, *t              |=2:0;    h=" ’‘-.|//,\\"  "|\\_"    "\\/\x23\n"[x/** ##
## */%80-              9?x[b]      :16];;usleep(  12321)      ;}return 0;}/* ##
####                                                                    ####
################################################################################
**********************************************************************************/
```

Algorithm

We applied two detection techniques to identify hard to read files: tail-cutting and cluster-based. Artefacts found by both methods were considered as heavy, high-impact unreadable files, whereas artefacts that were exclusively identified by only one of the methods were considered as medium-impact unreadable. We also had to add a criterion on the size of files, since the enumerated ratios can be set to artificially high values because of a very small SLOC without having a large DOPD/DOPT/etc., as it is the case for Java interfaces. The size threshold has been set to SLOC > 10. The R code serving this purpose for files is quite long and is shown in appendix D.3 on page 285.

At the function level, best results were achieved with cluster selection only. In the following code, only clusters which represent less than 10% of the total population are retained and their included items selected:

```
= START R =
clusts <- 7
project_d <- dist(project_data, method="manhattan")
hc <- hclust(d=project_d, method="single")
groups_single <- cutree(hc, k=clusts)
mylen <- ceiling( nrow(project_data)*0.1)
outs <- project[0,1:2]
names(outs) <- c("Name", "Cluster")
for (i in 1:clusts) {
  if ( length( groups_single[groups_single == i] ) < mylen ) {
    d <- project[groups_single == i, 1:2];
    names(d) <- c("Name", "Cluster");
    d[,2] <- i; outs <- rbind(outs, d)
  }
}
outs_all <- outs
outs <- outs_all[,1]
= END R =
```

Results

These methods gave good results. Figure 8.3 shows the number of hard to read files and functions for the various projects we analysed. We always find a fair number of them, defining a small set of items to watch or refactor for developers.

For the C projects we retrieved some outliers from the International Obfuscated C Contest web site (see IOCCC [138]) and scattered them in the test projects. We intended to identify all of them with our method, plus a few files in the project that definitively had analysability concerns. It worked quite well since the algorithm found the obfuscated files in all cases, either as high-impact or medium-impact. The high-impact method unveiled obfuscated files only, and one obfuscated file (the most readable one among the obfuscated files) was caught by the medium-impact rule. When more files were found by the Medium-impact rule, they were found to be hard to read as well. The following tables

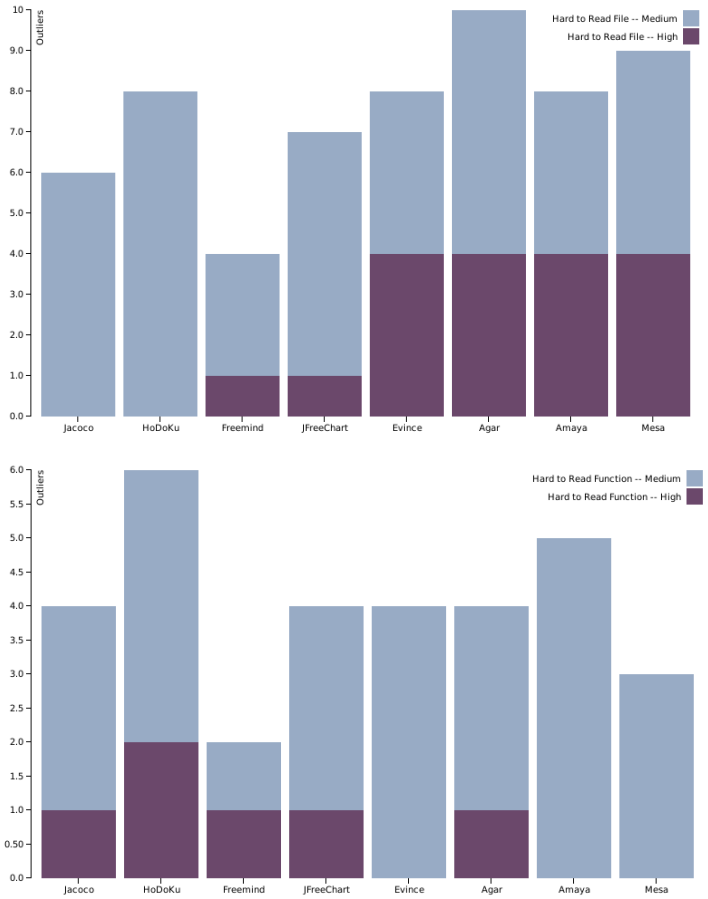


Figure 8.3: Hard To Read files and functions with our outliers detection method.

show numeric results on the Java and C projects: we get only a few Hard To Read files and they all show readability concerns.

Project	Jacoco	HoDoKu	Freemind	JFreeChart
File HTR High	0	0	1	1
File HTR Medium	6	8	3	6
Func HTR High	1	2	1	1
Func HTR Medium	3	4	1	3

Project	Evince	Agar	Amaya	Mesa
File HTR High (injected/natural)	4/0	4/0	4/0	4/0
File HTR Medium (injected/natural)	1/3	1/5	1/3	1/4
Func HTR High	0	1	0	0
Func HTR Medium	4	3	5	3

Figure 8.4 highlights the number of hard to read files detected with our method: all obfuscated files are identified (purple and blue) and a few other natural files (orange) are proposed, which all have analysability concerns.

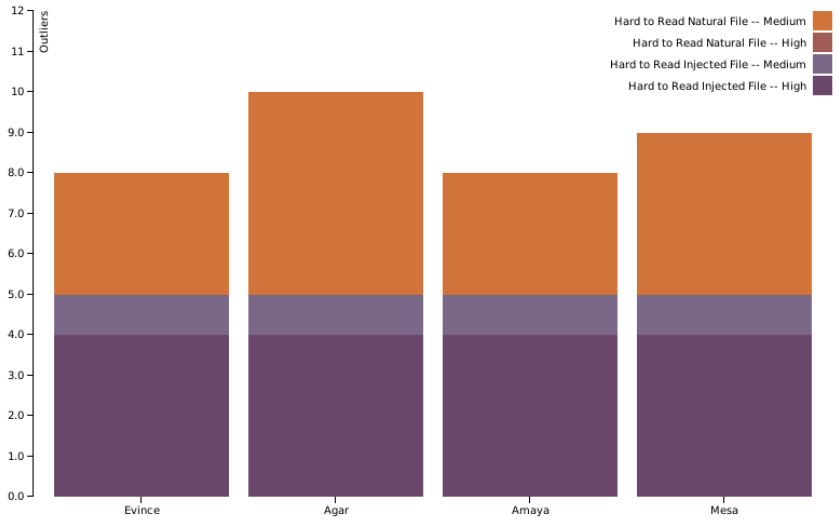


Figure 8.4: Hard To Read files with injected outliers.

Obfuscated files do not define functions that would trigger the Hard-To-Read flag. As a consequence, the functions highlighted by the outliers detection mechanism are only real-life functions found in the project itself and are hugely complex. An example of such a function is shown in appendix D.2.

8.4.2 Untestable functions

Inputs

Untestable functions are detected according to three measures that reflect the complexity of the code control flow. All of them have to be triggered to make the action item pop up:

VG is the cyclomatic complexity, a graph-complexity measure derived from the graph theory by McCabe [130]. Its forbidden range is defined as $[30; \infty[$.

NEST is the level of control-flow structures nesting. Its forbidden range is defined as $[4; \infty[$.

NPAT is the total number of execution paths. Its forbidden range is defined as $[800; \infty[$.

For domains with high constraints on testing and verification (like space or aeronautics, for which all (100% of) execution paths of the program must be covered by tests) this may be very expensive and may even block certification.

Algorithm

We want to highlight functions that cumulate high values on many of their attributes. To do this, we rely on the recipe experimented in the Version Analysis Knitr document (see section 4.2). Outliers are identified with adjusted boxplots on each metric, and artefacts

which have the higher number of outliering variables are selected. The R algorithm used is the following:

```
= START R =
require('robustbase')
dim_cols <- ncol(project)
dim_rows <- nrow(project)
outs_all <- data.frame(matrix(FALSE,
                              nrow = dim_rows,
                              ncol = dim_cols)
                       )
colnames(outs_all) <- names(project)
outs_all$Name <- project$Name
for (i in 2:ncol(outs_all)) {
  myouts <- which(project[,i] %in% adjboxStats(project[,i])$out);
  outs_all[myouts, i] <- TRUE
}
for (i in 1:nrow(outs_all)) {
  nb <- 0;for (j in 1:dim_cols) {
    nb <- nb+(if(outs_all[i,j]==TRUE) 1 else 0)
  };
  outs_all[i,1+dim_cols] <- nb
}
outs_all <- outs_all[order(outs_all[,1+dim_cols], decreasing=T),]
outs <- subset(
  outs_all,
  outs_all[,1+dim_cols] == max(outs_all[,1+dim_cols])
)
outs <- project_function[row.names(outs[,0]),"Name"]
= END R =
```

Results

In the tables shown below, OD lines show the number of artefacts identified by the outliers detection method while AI lines give the number of artefacts found by traditional, statically-defined action items. OD & AI lines give the number of artefacts that are found by both techniques. The *coverage* line indicates how many of the outliers are effectively detected by the action items while the *matching* line shows how many of the action items are found by the outliers. A 100% coverage means that all outliers are found by action items, and a 100% match means that all action items are covered by outliers. From our results, we took the best out of both measures because in the first case the outliers of the system are only a part of the action items (because they have really higher values than the average), and in the second case there are more outliers because the threshold has been lowered compared to the average files of the project.

Project (Java)	Jacoco	HoDoKu	Freemind	JFreeChart
Number of functions	1620	2231	4003	7851
OD Untest.	13	6	30	32
AI Untest.	5	23	4	10
OD & AI Untest.	5	6	4	10
Coverage	38%	100%	13%	31%
Matching	100%	26%	100%	100%
Overall intersection	100%	100%	100%	100%

Results are very good: in almost all cases we find an optimum intersection between action items and outliers, with a fair (i.e. low) number of artefacts. The Mesa project is the only exception, with some distinct outliers identified by both methods (AI & OD) – which is a very interesting feature, as we will see later on.

Project (C)	Evince	Agar	Amaya	Mesa
Number of functions	2818	4277	7206	11176
OD Untest.	3	5	1	24
AI Untest.	13	17	448	76
OD & AI Untest.	3	5	1	16
Coverage	100%	100%	100%	67%
Matching	23%	29%	0%	21%
Overall intersection	100%	100%	100%	67%

Figure 8.5 illustrates the repartition of detected files: in most cases one of the methods finds more outliers than the other, except for the Mesa project which has specific outliers detected by each method.

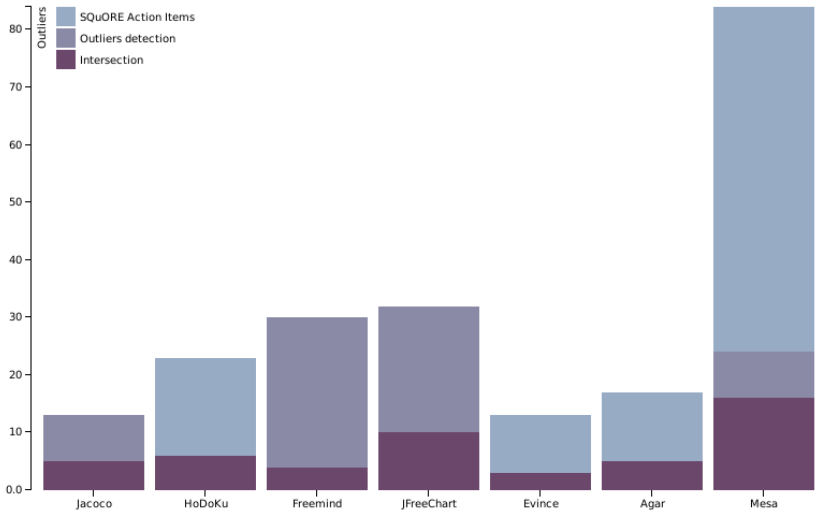
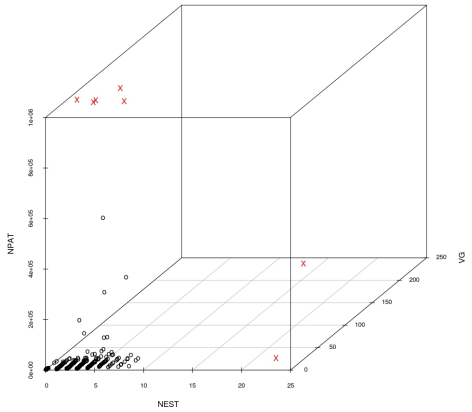


Figure 8.5: Untestable functions with SQuORE's action items and with our outliers detection method.

The Mesa project shows an interesting feature: some functions are identified by the outliers but not raised by action items. These are functions with a really high value on one or two of their metrics, which are not caught by the action items since their other metrics have standard values. They are however very difficult to read and test and deserve to be highlighted. The three-dimensional plot on the right clearly highlights these artefacts by showing their difference as compared to the average of files.



8.4.3 Code cloning in functions

Inputs

Code cloning refers to the duplication of code *inside* a function. It is one of the favourite SQuORE action items since it was introduced by our software engineering experts from their experience with customers and it works quite well. It relies on the following two metrics to identify potential duplicated code:

VOCF is the vocabulary frequency. VOCF > 10 reveals high reuse of the same vocabulary, i.e. operands and operators.

VG is the cyclomatic number of the function. VG $>= 30$ reveals a complex control flow.

Algorithm

We use the `adjboxStats` R function for skewed distributions to get extreme values on each metric, then apply a median-cut filter to remove lower values – we are not interested in low-value outliers. Only artefacts that have both metrics in outliering values are selected.

```
= START R =
require('robustbase')
dim_cols <- ncol(project)
dim_rows <- nrow(project)
outs_all <- data.frame(matrix(FALSE, nrow = dim_rows, ncol = dim_cols))
colnames(outs_all) <- names(project)
outs_all$Name <- project$Name
for (i in 2:ncol(outs_all)) {
  myouts <- which(project[,i] %in% adjboxStats(project[,i])$out);
  outs_all[myouts, i] <- TRUE
}
for (i in 1:nrow(outs_all)) {
  nb <- 0;
  for (j in 1:dim_cols) {
    nb<-nb+(if(outs_all[i,j]==TRUE) 1 else 0)
  };
  outs_all[i,1+dim_cols] <- nb
```

```
}
outs_all <- outs_all[order(outs_all[,1+dim_cols], decreasing=T),]
outs <- subset(
  outs_all,
  outs_all[,1+dim_cols] == max(outs_all[,1+dim_cols]))
outs <- project_function[row.names(outs[,0]),"Name"]
= END R =
```

Results

In the tables shown below, OD lines show the number of artefacts identified by the outliers detection method while AI lines give the number of artefacts found by traditional, statically-defined action items. OD & AI lines give the number of artefacts that are found by both techniques.

Project (Java)	Jacoco	HoDoKu	Freemind	JFreeChart
Number of functions	2818	4277	7206	11176
OD Clone.	3	8	10	13
AI Clone.	1	8	0	8
OD & AI Clone.	1	6	0	7
Coverage	33%	75%	0%	54%
Matching	100%	75%	100%	88%
Overall intersection	100%	75%	100%	88%

Project (C)	Evince	Agar	Amaya	Mesa
Number of functions	2818	4277	7206	11176
OD Clone.	2	6	10	35
AI Clone.	2	6	154	80
OD & AI Clone.	2	6	10	33
Coverage	100%	100%	100%	94%
Matching	100%	100%	6%	41%
Overall intersection	100%	100%	100%	94%

The method works quite well (although less than the above-mentionned untestable functions) on the set of C and Java projects used for validation: the intersection between action items and outliers varies from 75% to 100%. Null values could be marked as 100% since they simply show that the action items did not find any artefacts with the defined static thresholds.

The number of outliers output is always relatively small, even when action items return too many artefacts (as it is the case for Amaya). Figure 8.6 depicts the number of functions with code cloning for each project; our algorithm finds a relatively small number of functions, even when SQuORE action items find none (e.g. for Freemind) or too many of them (e.g. for Amaya). These results have been cross-validated with our staff of experts:

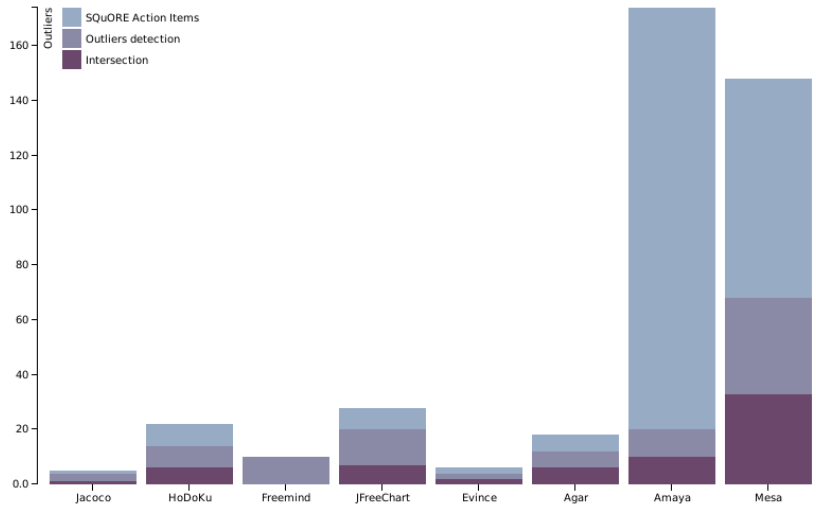


Figure 8.6: Code cloning in functions with SQuORE's action items and with our outliers detection method.

the cases where outliers detection didn't find the same artefacts than the classical action items (as for Mesa, HoDoKu or JFreeChart) still have potential for refactoring because the latter cannot catch smaller pieces of duplicated code nor duplicates that have been intensively edited (but are still subject to refactoring).

8.5 Summary

The technique presented here, based on automatic outliers detection, is more specific than statically computed triggers and highlights types of files that are not discovered by classic techniques. The overall method is more robust: it will highlight extreme values regardless of the shape of the software. This robustness is demonstrated by two examples: we could detect specific untestable files within Mesa, and some code cloning functions within the Freemind project, where SQuORE could not identify them. Another great advantage of our method is the fair number of files or functions selected: we usually get less than 20-30 artefacts, even on bigger projects.

This SQuORE Lab was the first one to implement the Maisqual roadmap as defined in section 4.5, and the first pragmatic implementation of data mining techniques into the SQuORE product. We first relied on Knitr documents as working drafts to run test drives and improve the methods, so we were confident enough in our results before implementing a clean integration in SQuORE. The Knitr documents were applied on the Maisqual data sets, and a different set of projects was used for the validation – since the validation of the full chain (from project parsing to results visualisation in SQuORE interface) had to start from project sources, not flat csv files. The final integration in SQuORE is the ultimate achievement of this work.

The integration is now fully functional and provides, out-of-the-box, the three detection

techniques described here. This first round of outliers should be expanded and improved with the specific knowledge of people involved with customers' needs. The next step is to involve consultants in the definition of new rules and let them write their own detection techniques, thanks to the modular and easy-to-use architecture. New rules should be added to the standard configuration of the SQuORE product as people become more confident with it.

The SQuORE Labs outliers project took 28 man-days to complete, from requirements definition to delivery. Flavien Huynh worked on the integration into SQuORE and I worked on the modular architecture for configuration scripts. Accuracy of results has been considered satisfactory enough by the product owner to be integrated in the product trunk and should be made available in the upcoming 2014 major release of SQuORE.

Chapter 9

Clustering

Clustering, or unsupervised classification, is a very useful tool to automatically define categories of data according to their intrinsic characteristics. It has been intensively applied in areas like medecine, document classification or finance but until now has not given practical and applicable results to software.

However there is a need for unsupervised classification methods in software engineering. They offer two advantages: firstly, they automatically adapt values and thresholds to the context of the analysis, which answers some of the issues of the huge diversity of software projects. Secondly they allow for autonomous rating of artefacts without the need of supervisers. This second point is mandatory for the pragmatic application of such metrics, since most projects will not take the time to train a model before using it.

In this chapter we investigate the use of unsupervised classification algorithms to define optimised groups of artefacts for quality assessment and incremental improvement. This is an on-going work: we still need to explore new metrics, test new clustering algorithms, find better optimisations for the different input parameters, and interact with experts to fine-tune our methods.

Section 9.1 quickly reviews some existing work on software measurement data clustering. In sections 9.2 and 9.3 we describe the problems we want to address in the context of SQuORE: automatic scaling of measures and multi-dimensional automatic quality assessment. We illustrate the paths we followed and the early results for both axes. Finally section 9.4 lists some of the pitfalls and semantic issues that arose during this work and proposes directions for future research. Appendix C.2 shows a few extracts of the Knitr documents we used.

9.1 Overview of existing techniques

Clustering methods

This chapter investigates two clustering methods: k-means and hierarchical clustering. K-means determines group membership by calculating the centroid (or mean for univariate data) for each group. The user asks for a number of clusters, such that the within-cluster

sum of squares from these centres is minimised, based on a distance measure like Euclidean or Manhattan. Other techniques may use different ways to compute the center of clusters, like selecting the median instead of the mean for k-medoids [115, 24].

K-means cluster visualisation is usually achieved by plotting coloured data points; centers of clusters can also be displayed to show the gravity centers of clusters. An example of k-means clustering on file SLOC for Ant 1.8.1 is shown in the left plots of figure 9.1 page 161.

The R implementation of **hierarchical clustering** [134, 115, 24] we are using is `hclust` from the `stat` [147] package. It implements two distances, Euclidean and Manhattan, and several aggregation methods: Ward, Complete, Single, Average, Median, McQuitty, Centroid. Hierarchical clustering results are usually displayed through a dendogram as shown for Ant 1.7.0 files in figure 3.4 on page 52.

We are more specifically looking for evenly distributed clusters: having all items rated at the same level is not very useful. Furthermore, with evenly distributed clusters, one has enough examples to practically compare files with different ratings and identify good and bad practices.

The objective of hierarchical clustering is to find underlying structures in the data's characteristics. Depending on the type of data considered, the number of natural clusters may differ and establishing a fixed number of groups may be inaccurate. Almeida et al. propose in [3] an improved method for hierarchical clustering that automatically find the natural number of clusters, if any. Outliers are filtered before the analysis and re-attached to identified clusters afterwards, which considerably increase the robustness of the algorithm.

Usage of clustering

K-means clustering was applied by Herbol [88] on time series to detect feature freeze before milestones in Eclipse projects. Zhong et al. [200] apply two different clustering algorithms (namely k-means and neural-gas) to software measurement data sets to ease the inspection process by experts: instead of evaluating thousands of software components, experts only had to look at 20 to 30 groups at most. They also apply various classifiers to fault-related metrics and compare the results with the expert-based classification, showing large false positive rates in some cases (between 20 and 40 %). Naib [136] also uses k-means to classify fault-related data on large C projects to estimate fault-proneness of software components.

In [6] Antonellis et al. use the analytical hierarchy process[1] to weigh expert-based

[1] AHP is a decision making technique that allows consideration of both qualitative and quantitative aspects of decisions [156]. It reduces complex decisions to a series of one-to-one comparisons and then synthesizes the results.

evaluations of the maintainability of the project's files. *k-attractors* [104] clustering is then applied to the resulting maintainability measure and conclusions are manually drawn from the different groups identified and their correlations with individual metrics.

9.2 Automatic classification of artefacts

9.2.1 SQuORE indicators

In SQuORE, *measures* (i.e. raw numbers like 1393 SLOC) are matched against *scales* to build *indicators*. These are qualitative levels allowing practitioners to instantly know if the measured value can be considered as good, fair, or poor in their context. Most scales have 7 levels, because it is a number that is easily captured by the mind. SQuORE uses designations from level A to level G in a way that mimics the well-known Energy Efficiency Rating logo. In our tests, we will use 7 clusters since we want to check our results against ratings proposed by SQuORE.

Scales are defined as static intervals: triggering values are defined during the configuration phase of the measurement program and should not be modified after going live, because doing so would impact the rating of artefacts and threaten the temporal consistency of the analysis.

9.2.2 Process description

From the consulting perspective, the typical execution of a SQuORE assessment process is as follows:

- ⇝ The customer has first to define her requirements for the quality assessment program.
- ⇝ The consultant helps the customer to select a model that best fits her needs and identified requirements – SQuORE has a predefined set of models for many different types of industry.
- ⇝ When the quality model is defined the consultant has to *calibrate* it: meaningful values have to be chosen for the rating of artefacts. As an example, complexity thresholds are lowered for critical embedded systems with strict certification constraints.

Calibration is achieved by running the quality model on real data sources and projects and make it match the customer's feeling of what is good, fair or bad. The intent is to enforce the representation condition of the measurement program.

9.2.3 Application: the auto-calibration wizard

We wanted a non-intrusive tool to help in the calibration phase of a SQuORE deployment without interfering with consultant's usual methods. For this we wrote a Knitr document taking as input the results of a project analysis run, and proposing scales tailored to

the project values. The document can be applied on a large data set containing several components to compute optimised thresholds for each metric from the full range of values in the projects. Since a quality model is supposed to be used by many projects of various sizes and with different types of constraints such proposals are useful to setup balanced ranges for the very context of the customer.

Univariate k-means clustering is applied to each metric individually and various information is displayed to help the consultant calibrate the model. The limits or triggering values that define the clusters are inferred from the computed repartition, and the document proposes an adapted range with the associated repartition of artefacts. It draws colourful plots of data to visually grasp the repartition of artefacts according to their rating. Table 9.1 shows the table output by the document for the VG and SLOC metrics on the Ant 1.8.1 release files. For each metric, we give the number of items in the cluster, and the minimum and maximum values delimiting the cluster's boundaries.

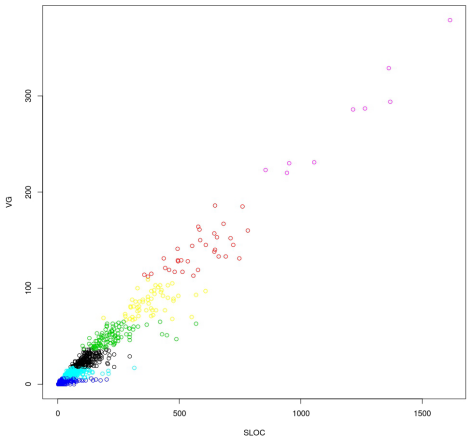
Table 9.1: Auto-calibration ranges for Ant 1.8.1 file metrics.

Metric	Levels	A	B	C	D	E	F	G
VG	Number of items in cluster	707	45	27	6	111	275	5
VG	Min range	0	15	38	72	117	183	286
VG	Max range	14	37	71	115	167	234	379
SLOC	Number of items in cluster	167	27	531	8	85	53	305
SLOC	Min range	2	48	107	192	328	529	952
SLOC	Max range	47	106	188	320	513	855	1617

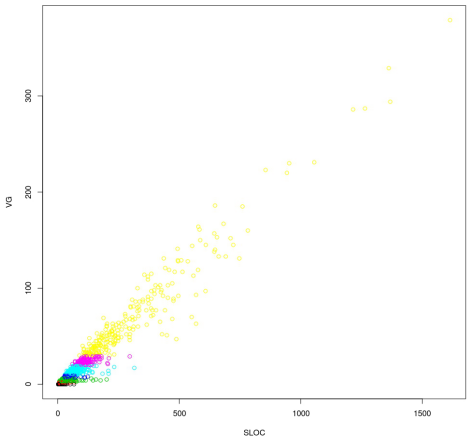
The first (upper) row in figure 9.1 shows the repartition of file artefacts for Ant 1.8.0 on the VG metric, with the k-means clustering algorithm and with SQuORE static values. The next pictures show the same output for the SLOC metric. These figures clearly show that the repartition of artefacts is much more equilibrated in the case of automatic clustering than with static threholds. This is especially true for higher values, which is considered to be good for highlighting artefacts that have outstanding characteristics.

One can notice that the range proposed by the algorithm is discontinuous. We modify it to make it a continuous scale and include outer limits so the full range spans from 0 to infinite. In the VG example, the following scale is proposed from the above ranges. It is formatted according to the structure of SQuORE XML configuration files, so practitioners just have to copy and paste the desired scale:

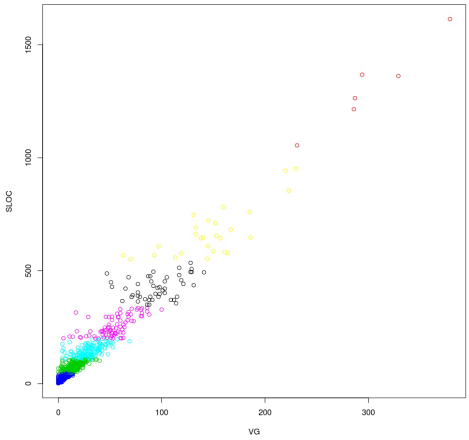
```
<Scale scaleId="SCALE_NAME">
    <ScaleLevel levelId="LEVELA" bounds="[0;14]" rank="0" />
    <ScaleLevel levelId="LEVELB" bounds="]14;37]" rank="1" />
    <ScaleLevel levelId="LEVELC" bounds="]37;71]" rank="2" />
    <ScaleLevel levelId="LEVELD" bounds="]71;115]" rank="4" />
    <ScaleLevel levelId="LEVELE" bounds="]115;167]" rank="8" />
    <ScaleLevel levelId="LEVELF" bounds="]167;234]" rank="16" />
    <ScaleLevel levelId="LEVELG" bounds="]234;[" rank="32" />
```



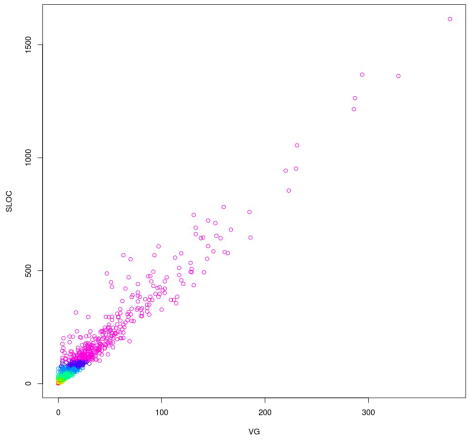
(a) VG with k-means.



(b) VG with SQUORE.



(c) SLOC with k-means.



(d) SLOC with SQUORE.

Figure 9.1: Examples of univariate classification of files with k-means and SQUORE.

```
</Scale>
```

9.3 Multi-dimensional quality assessment

The idea of multi-dimensional quality assessment is that a few metrics can summarise most of the variations on a given quality factor across artefacts. We would like to use it to autonomously identify artefacts that have good, fair or bad quality characteristics.

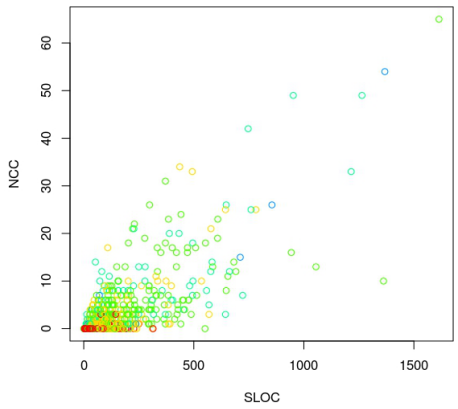
For that purpose we applied a multivariate clustering algorithm to a carefully selected subset of metrics that are known to contribute to some quality characteristic. We relied on the experience and knowledge of our software engineering experts to select metrics. Two examples of empirical findings are provided: first we investigated *maturity* (as formulated by embedded engineers), which is often sought to be roughly dependent on the size (measured as SLOC), complexity (measured as VG) and number of non-conformities (NCC) of an artefact. We then investigated *testability*, which is estimated using control-flow complexity (VG), the maximum level of nesting in a control structure (NEST), and the number of potential execution paths (NPAT).

Figure 9.2: Computation of testability in SQUORE's ISO9126 OO quality model.

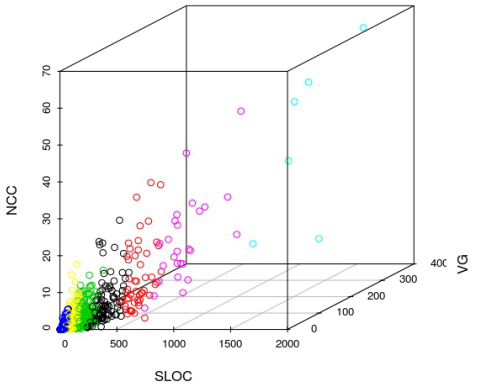
In order to validate our findings, we wanted to compare the clusters produced when the algorithm is applied on these three metrics with a measure of testability for the artefacts. SQUORE produces such a measure, based on the above-mentioned metrics plus a number of others like the number of non-conformities to testability rules, data and control flow complexity, and the number of classes and functions. The static aggregation method used in SQUORE for the ISO 9126 OO quality model is depicted in figure 9.2.

Figure 9.3 shows the results of multi-dimensional clustering and compares them with the ratings output by SQUORE. Plots in the first row deal with maturity; the clustered repartition is obviously driven by the SLOC metric while SQUORE draws a fuzzy repartition in the perspective of the two plotted metrics. Results are similar for the testability comparison pictured on the second raw.

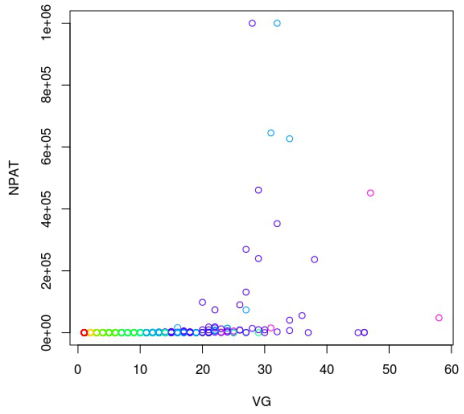
The shape of clusters is clearly not similar with both methods (SQUORE and k-means). This can be partially explained however, and improved: firstly SQUORE relies on a



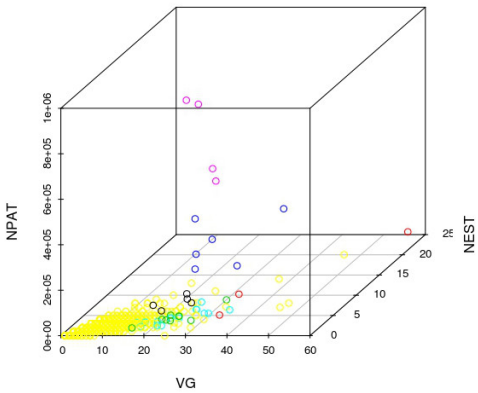
(a) SQUORE results for maturity.



(b) Clustering for SLOC, VG and NCC.



(c) SQUORE results for testability.



(d) Clustering for VG, NEST and NPAT.

Figure 9.3: Examples of multivariate classification of files for Ant 1.8.0.

number of extra parameters, which grasp different aspects of testability and introduce new dimensions to the knowledge space. The impact of these measures on results should be investigated, and the set of selected metrics modified or expanded. Secondly another bias lies in the visualisation of these repartitions: SQUORE output should be visualised on different axes to identify the driving metrics. Doing so we should be able to find a better perspective to visually identify ranges of values and make them match the k-means-clustered repartition of artefacts.

We extracted some arbitrary files and manually checked them for their maintainability and testability with our team of experts. Results were good, i.e. files that were considered by the clustering algorithm to be hard to test *really* were, which confirms that the metrics selected still show interesting properties for the considered quality characteristic. We consider however that more work is needed to build a more consistent and more verifiable

automatic assessment method.

9.4 Summary & future work

Univariate clustering shows interesting insights on an optimised repartition of artefacts to establish a scale on individual indicators. We wrote Knitr documents to demonstrate the application of such methods on real projects and proposed means to apply them seamlessly in the course of a typical SQuORE analysis. This research looks promising because the clusters defined on the artefacts are much more in equilibrium than with the SQuORE product and correspond to a natural repartition of values. Doing so brings useful insights on the categories to be watched and helps in the understanding and improvement of software quality of artefacts (files, classes, functions) in a product.

We also investigated *multi-dimensional analysis* to look for groups of artefacts with similar quality characteristics. By using a few carefully selected metrics, we clustered files from a projects and compared the results with categories of artefacts selected by SQuORE and our team of experts, according to some high-level quality attribute (testability, analysability, maturity). Results are incomplete on that part, however, and need to be further refined because of the semantic and practical applicability issues described hereafter.

- ⇝ If scales are automatically computed for each project, then files cannot be compared across projects: if a project is very well written and another shows poor maintainability, G-rated files from the former may be equivalent in terms of quality to the A-rated files from the latter. This implies a redefinition of some of the SQuORE assumptions and mechanisms to ensure consistency of results.
- ⇝ One great advantage of SQuORE is it really describes the computation mechanism used to aggregate measures to upper quality characteristics, allowing users to discover *why* they get a bad (or good!) rating. The drawback of automatic clustering methods is that they do not always clearly show which metric or set of metrics is responsible for rating an artefact in an unexpected category. We need to develop techniques to explain the classification of artefacts, by identifying which metrics or combination of metrics have unusal values.

There a few solutions that should be investigated, though. Other clustering algorithms should be evaluated for this task, with better robustness and pertinence for software measurement data. Methods based on the Mahalanobis distance look promising [24, 95], and new papers and ideas specifically targeted at software measurement data are being published regularly [6, 107, 3, 136] and should be checked in this context.

Once again, one of the very advantages of clustering methods is they do not need to be supervised; we believe this is a determinant factor of success and practical utilisation in real-world software development projects since users or managers have no time (nor possibly will or knowledge) to do much configuration. Practitioners also often consider

that an automous algorithm is more objective than a manually-trained one. This SQuORE Lab is currently being investigated means to improve this aspect and integrate them smoothly in the process.

The clustering methods we implemented unveiled interesting paths to be investigated, but are not mature enough as for now to be used in a production environment. The calibration document has been presented to developers and consultants, and we are in the process of setting up a protocol to get feedback from the field, and make it a part of the typical SQuORE calibration process. However the culmination of this SQuORE Lab is to be integrated in the SQuORE product as an extension of the existing facilities in the capitalisation base.

Chapter 10

Correlating practices and attributes of software

Regression analysis allows one to find an underlying structure or relationships in a set of measures (see section 3.5 in data mining chapter). One of the intents of the Maisqual project was to investigate the impact and consequences of development practices on the software metrics we defined. Examples of correlations we are looking for include e.g. SCM_COMMITS and SCM_FIXES to be correlated with the cyclomatic complexity (i.e. more modifications are needed when the file is complex), or the respect of coding conventions to be correlated to the number of distinct committers.

This chapter dives into the links that exist between the behaviour and characteristics of a software project, including its code, the people working on it, and its local culture. As defined earlier we wrote a Knitr document to uncover the basics of the problem, then refined it to address more specific patterns or issues.

Section 10.1 presents the approach used during this work, and section 10.2 describes the documents developed to achieve stated goals. Preliminary results are discussed in section 10.3. Section 10.4 states some early conclusions and guidelines for future work on this subject.

10.1 Nature of data

The early stages of this work took place when we wanted to extract practices-related information from software repositories. Most rule-checking tools typically indicate transgressions as *findings* attached to lines of code. Two transformations are usually applied to these findings in order to extract numerical information: either counting the number of violations on the artefact (may it be an application, folder, file, class or function), or counting the number of *observed rules* on the artefact (however often transgressed rules are violated).

In this case, *Development practices* are primarily identified by the number of rule violations on artefacts. Some metrics can be considered as practices as well, since they

may be representative of a development policy or custom. Examples of such metrics include the number of lines with only braces (BRAC), which denotes a coding convention style, and the comment rate (COMR) as a measure of code documentation. Of course, rule-related metrics like ROKR and NCC are considered for the practice analysis. The *attributes of software* are the characteristics of the project or product we identified when building the data sets – see section 6.1 for a comprehensive list of metrics gathered.

One of the assumptions of regression analysis is the independence of the base measures. This is not true for all of our measures, through; the following list gives some of the most obvious combinations observed in the metrics we gathered.

$$\begin{aligned} \text{SLOC} &= \text{ELOC} + \text{BRAC} \\ \text{LC} &= \text{SLOC} + \text{BLAN} + \text{CLOC} - \text{MLOC} \\ \text{LC} &= (\text{ELOC} + \text{BRAC}) + \text{BLAN} + \text{CLOC} - \text{MLOC} \\ \text{COMR} &= ((\text{CLOC} + \text{MLOC}) \times 100)/(\text{ELOC} + \text{CLOC})) \end{aligned}$$

This is even less true if we consider the high correlation among line counting metrics or between size and complexity metrics [63, 64]. For these reasons we need to investigate and take into account the independence of variables during the analysis and interpretation of results.

Non-robust regression algorithms are by definition sensible to noise. Rule-checking tools are known to have false positives [183]. We tried to avoid rules that engendered to much debate as for their relevance or reliability regarding the number of false-positives. Nevertheless, since we measure the number of warnings of tools (and not real issues in the code), noise associated to false positives is assumed to be constant over the data set.

10.2 Knitr investigations

We wrote a first Knitr document to automatically apply various regression mechanisms to the file-level data sets we generated. It featured a quick statistics summary of data, a study on the relationships between the metrics themselves, and the analysis of rules. Three categories of rules are defined, depending on the tool used: SQUORE, PMD and Checkstyle. Since they target different types of checks (see section 6.2), results vary greatly from one set to another.

We computed pairwise regression using different algorithms and techniques; this gave us full control over the regression commands and parameters, and provided us with access points for plotting specific values. The methods investigated were:

- ⇝ The standard `lm` command from the `stats` package can be used to apply many different regression models. We first tried to apply linear correlation, using the following formula:

  $$y = ax + b$$

  The returned `lm` object supplies the adjusted R^2 to assess the goodness of fit.

⇝ We also applied quadratic regression using `lm` and the following formula:

$$y = ax^2 + bx + c$$

We relied on the adjusted R^2 of the fit to assess its goodness.

⇝ The `MASS` package introduces the `rlm` command for robust regression using an M estimator [196]. In `rlm` fitting is done by iterated re-weighted least squares. Initial estimate is obtained from a least-squares fit using weights.

⇝ The `robustbase` package provides another command for robust regression: `lmrob`. It computes an MM-type regression estimator as described by Yohai [196] and Koller and Stahel [114]. We configure the call to use a 4-steps sequence: the initial estimator uses a S-estimator which is computed using the Fast-S algorithm []. Nonsingular subsampling [113] is used for improved robustness. Next steps use a M-regression estimate, then a Design Adaptive Scale [114] estimate and again a M-estimator.

We applied these regressions on two different sets: the first to uncover the correlations among metrics (i.e. `metricA` ∼ `metricX`), as explained in the previous section, and the second to explain metrics with rules (i.e. `metricA` ∼ `ruleX`). Results are delivered in tables giving for each metric the adjusted R^2 for each metrics and rules they are regressed with.

Extracts of the Knitr document are provided in appendix C.3 page 272, showing some correlation tables on metrics and rules at the file level for all releases of JMeter. For the sake of conciseness and visualisation comfort, we highlighted specific values in the generated tables. For pure metrics correlation tables, values lower than 0.01 are not displayed and values higher than 0.9 are typeset in bold. For metrics-rules correlation tables, values higher than 0.6 are typeset in bold.

10.3 Results

At the file level, applying correlation on metrics with rules as explanatory variables gives very poor results. For linear and polynomial regression, adjusted R^2 varies between 0 and 0.4 with most values being under 0.2. Robust algorithms simply do not converge,

Unsurprisingly, sets of metrics that have been identified to be correlated in the pure-metrics correlation matrix (as stated in previous section) change in similar proportions. As an example, the table generated for the JMeter releases set of files, provided in appendix C.3 on page 278, shows that the LINELENGTH check is slightly[1] correlated with the number of statements (0.619), and also with SLOC (0.568), NCC (0.731), LC (0.55) and ELOC (0.617). If we take the table of pure-metrics correlations shown in appendix C.3 on page 274 we observe these metrics are indeed highly correlated (Adjusted R^2 being between 0.864 and 0.984).

[1] At least compared to the remaining of the table. Statisticians consider there is a correlation for values higher than 0.9.

The repartition of rule violations is peculiar: on the rule set we selected many rules are rarely violated, while other rules target common harmless warnings and hence show a large number of violations. This is highlighted on figure 10.1: most rules have their third quartile set to zero (i.e. the box is not visible), and values strictly positive (i.e. at least one violation) are most of the time considered as outliers. The redness of outliers shows that the density of violations in files rapidly decreases. Most contravened rules are immediately identified with their box popping over the x axis; they are R_RETURN (SQUORE), INTERFACEISTYPE (PMD), and JAVADOCMETHOD (PMD).

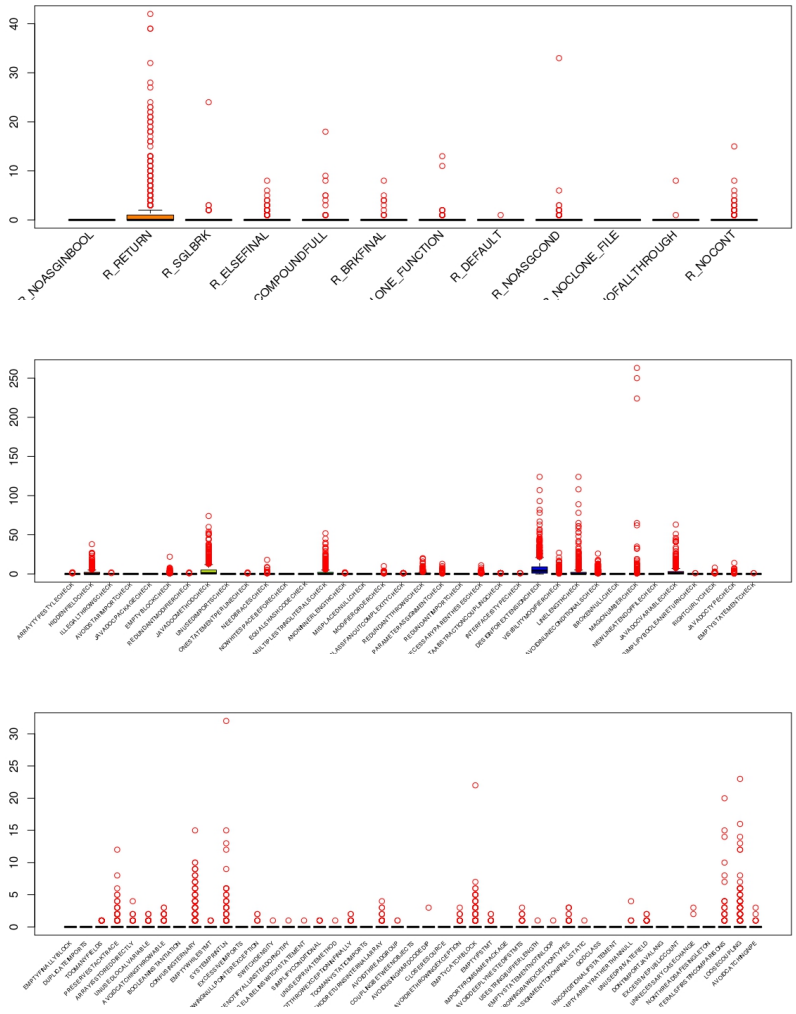


Figure 10.1: Repartition of violations by rule for Ant 1.8.0 files.

This creates a very sparse matrix, which causes standard algorithms to fail. The Knitr document produced many warnings stating that the `rlm` could not converge because the matrix is singular. `lmrob` consistently yields it could not converge in the allowed number of iterations. We raised the number of iterations from 20 (the default) to 40 and 60, and tried different methods (e.g. different estimators and parameters, and alternative defaults

for `lmrob` as proposed in [114]) without success.

10.4 Summary & future work

In this chapter, we described the experiments performed to establish relationships between practices, measured as violations to known coding rules, and metrics. We applied various regression methods and eventually could not corroborate verifiable relationships using these methods. Although much work has to be done with other, different techniques before giving definitive judgement, the following intermediate conclusions can already been drawn:

- ⇝ This is not as trivial as it seems. We identified two bottlenecks: firstly measurement of daily development practices is subject to many biases and can be measured from different perspectives. Secondly the nature of data is unusual and gets us out of the scope of comfortable, more classical shapes and treatments.
- ⇝ Using metrics as output and rules as explanatory variables, linear and polynomial regression methods give very bad results with adjusted R squared values in the range $[0 : 0.33]$. We could not establish any definitive relationship on the data sets analysed with these algorithms.
- ⇝ Rules-related data, when extracted from rule checking tools, are generally sparse and have an overall unusual shape, which causes many algorithms to fail. Classical robust regression (`rlm` or `lmrob` algorithms) is not able to converge on such data.

Results are incomplete as for now and need further work to show reliable and meaningful conclusions. From what we have learned, a few paths may be worth exploring:

- ⇝ Further reflecting on **how practices can be discovered** in software metrics and rule findings may bring novel insights to the initial question. Different means to measure similar characteristics may open new perspectives by introducing new data, with more usable distributions, or may enable the use of different algorithms (such as those used for process discovery, see [96, 97, 192, 86]).
- ⇝ We applied pairwise only correlations, but the underlying structure of the measured system may involve an arbitrary combination of variables. While this would be a time-consuming task, investigating **multiple regression** would bring useful insights into the complex structure of influences among metrics.
- ⇝ There are many different types of **robust algorithms** for regression analysis. Logistic, more elaborated non-parametric, or sparse-data targeted regression techniques may be more successful. Also **transforming data** (e.g. normalisation, principal components analysis) may leverage its accessibility for some algorithms.

Part IV

Conclusion

Initial objectives

During these three years we learned to apply methods from the statistical and data mining field onto domain-specific concerns of software engineering. The roadmap we roughly followed is depicted in figure 10.2: we elaborated a semantic, methodological and technical framework to generate safe data sets; these were used in specific workshops. Conclusions and consequences were either developed in SQuORE Labs or expressed in papers and ideas.

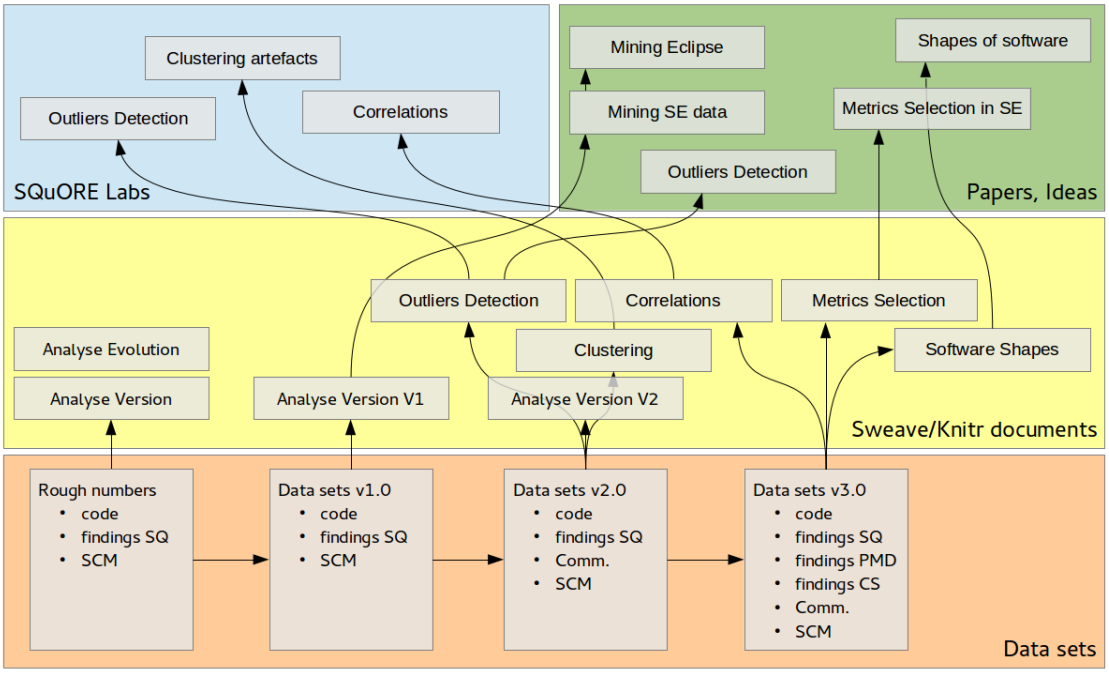


Figure 10.2: Synopsis of Maisqual.

What has been done

The Maisqual project allowed us to uncover how data mining methods can help in software engineering practices and quality assessment. During these three years, we were able to:

- ⇝ Study the most prominent and advanced techniques in the intersecting fields of software engineering and data mining to exchange ideas with people and expand the knowledge of the SQuORING Technologies team and practitioners we met.
- ⇝ Setup a reliable framework, both on the methodological and pragmatic implementation aspects. This permitted us to communicate our preliminary findings, and to submit articles to conferences and journals.
- ⇝ Run a restricted set of projects to address specific software engineering concerns, which led to direct implementation of new features into the SQuORE product and pragmatic application in projects.

This last part, composed of the SQuORE Labs, has been the very projects that enabled us to draw practical conclusions from Maisqual. The Outliers detection and Clustering SQuORE Labs contributed practical developments to SQuORING Technologies and enabled us to provide useful tools and methodological help for the consultants. The Correlation SQuORE Lab, albeit not yet complete, has already provided some good insights pertaining to the nature of data and the tooling needed for further analysis. Besides the implementation of novel ideas into pragmatic applications, it was an opportunity for SQuORING Technologies and our collaborators to build a bridge with the academic field of data mining and its thrilling contributions.

After taking a step back, doing an extensive research project was difficult to achieve in the designated time of two days a week. We had to put aside many interesting ideas and constantly check the scope of the project to make it fit into the restricted time frame and still deliver valuable and practical results. Here are a few of the interesting paths that we did not follow up on but are undoubtedly very promising.

What lies ahead?

A few papers are on the way, in collaboration with researchers from INRIA, and should be submitted during the course of 2014 to software engineering conferences.

Following the observations we made on the distribution of software metrics in section 4.2.2, we are in the process of writing an article on **the shapes of software**. The idea is to investigate power laws [36] in metric distributions, and a first Knitr has already been written that implements common checks on distribution functions. Some studies targeting power laws in software have already been published recently for graph-oriented measures of software, e.g. for software dependency graphs [129] or for various software characteristics as in Concas et al. [38], but to our knowledge there is no comprehensive study on the various software measures we are dealing with. We are working with Martin Monperrus and Vincenzo Musco from the INRIA Lille team.

The **selection of metrics** using techniques including principal component analysis, is another area of great interest, especially for SQuORE: the capacity to know what metrics should be selected for analysis would economise resources and time for the engine. There are a few studies that investigate dimensionality reduction for software metrics for specific concerns like cost estimation [179], however we believe that a more generic method can be used to uncover the main drivers of software analysis.

There is still room for improvement in the SQuORE Labs that were run with SQuORING Technologies. The **outliers detection** project should be enriched with new outliers detection techniques like LOF, k-means or Mahalanobis distance, and new types of outliers should be investigated. We want to assist SQuORING Technologies consultants to apply their acquired knowledge and experience to the writing of algorithms for specific customer needs. The **clustering** SQuORE Lab is not finished and will be continued, either to setup a configuring wizard for the team and customers or to be directly integrated into the SQuORE product. Last but not least, the SQuORE Lab working on **correlations**

between metrics and practices needs to be completed and its conclusions publicised and integrated into the body of knowledge.

Throughout the course of this study some interesting ideas had emerged. They could not be investigated in depth: they may not have been in the scope of this thesis, and the allocated time for research was filled to capacity so we were forced to carefully select our goals. Nevertheless, these ideas have been put on a back burner for later study because they could quite possibly provide meaningful conclusions or important features for the SQuORE product.

We finally need to better **communicate about our work and results** to foster our collaboration with practitioners and to investigate new uses and applications for our research. First, the data sets should be publicised, with a thorough description of the metrics extracted (definition, samples as requirements for reproduction, and links between metrics), a clear description of the retrieval process, a quick analysis showing important events on the project, and examples of usage for the data set with working R code samples. New projects will be added with time thanks to the automated architecture we setup. Second, the Maisqual wiki will be further elaborated upon and updated with new information and insights resulting from our work.

Bibliography

[1] C. C. Aggarwal and P. S. Yu. Outlier detection for high dimensional data. *ACM SIGMOD Record*, 30(2):37–46, June 2001.

[2] M. Al-Zoubi. An effective clustering-based approach for outlier detection. *European Journal of Scientific Research*, 2009.

[3] J. Almeida and L. Barbosa. Improving hierarchical cluster analysis: A new method with outlier detection and automatic clustering. *Chenometrics and Intelligent Laboratory Systems*, 87(2):208–217, 2007.

[4] S. Amasaki, T. Yoshitomi, and O. Mizuno. A new challenge for applying time series metrics data to software quality estimation. *Software Quality Journal*, 13:177–193, 2005.

[5] C. Andersson and P. Runeson. A replicated quantitative analysis of fault distributions in complex software systems. *IEEE Transactions on Software Engineering*, 33(5):273–286, 2007.

[6] P. Antonellis, D. Antoniou, Y. Kanellopoulos, C. Makris, E. Theodoridis, C. Tjortjis, and Nikos Tsirakis. Employing Clustering for Assisting Source Code Maintainability Evaluation according to ISO/IEC-9126. In *Intelligence Techniques in Software Engineering Workshop*, 2008.

[7] N. Antonellis, P. and Antoniou, D. and Kanellopoulos, Y. and Makris, C. and Theodoridis, E. and Tjortjis, C. and Tsirakis. A data mining methodology for evaluating maintainability according to ISO/IEC-9126 software engineering–product quality standard. *Special Session on System Quality and Maintainability-SQM2007*, 2007.

[8] N. Ayewah, W. Pugh, D. Hovemeyer, J. D. Morgenthaler, and J. Penix. Using static analysis to find bugs. *IEEE Software*, 25(5):22–29, 2008.

[9] A. Bachmann, C. Bird, F. Rahman, P. Devanbu, and A. Bernstein. The Missing Links: Bugs and Bug-fix Commits. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 97–106, 2010.

[10] B. Baldassari. SQuORE : une nouvelle approche de mesure de la qualité des projets logiciels. *Génie Logiciel*, 103:9–14, 2012.

[11] B. Baldassari. SQuORE: A new approach to software project quality measurement. In *International Conference on Software & Systems Engineering and their Applications*, 2012.

[12] B. Baldassari and F. Huynh. Software Quality: the Eclipse Way and Beyond. In *EclipseCon France*, 2013.

[13] B. Baldassari, F. Huynh, and P. Preux. De l'ombre à la lumière : plus de visibilité sur l'Eclipse. In *Extraction et Gestion des Connaissances*, 2014.

[14] B. Baldassari and P. Preux. A practitioner approach to software engineering data mining. In *36th International Conference on Software Engineering*, 2014.

[15] B. Baldassari and P. Preux. Outliers Detection in Software Engineering data. In *36th International Conference on Software Engineering*, 2014.

[16] B. Baldassari and P. Preux. Understanding software evolution: The Maisqual Ant data set. In *Mining Software Repositories*, number July, 2014.

[17] T. Ball. The Concept of Dynamic Analysis. In *Software Engineering—ESEC/FSE'99*, pages 216–234, 1999.

[18] V. Barnett and T. Lewis. *Outliers in Statistical Data (Wiley Series in Probability & Statistics)*. Wiley, 1994.

[19] V. R. Basil and A. J. Turner. Iterative enhancement: A practical technique for software development. *IEEE Transactions on Software Engineering*, 4:390–396, 1975.

[20] V. Basili and J. Musa. The future generation of software: a management perspective. *Computer*, 24(9):90–96, 1991.

[21] K. Beck, A. Cockburn, W. Cunningham, M. Fowler, M. Beedle, A. van Bennekum, J. Grenning, J. Highsmith, A. Hunt, and R. Jeffries. The Agile Manifesto. Technical Report 1, 2001.

[22] K. Beck and W. Cunningham. Using pattern Languages for Object-Oriented Programs. In *Object-Oriented Programming, Systems, Languages & Applications*, 1987.

[23] I. Ben-Gal, O. Maimon, and L. Rokach. Outlier detection. In *Data Mining and Knowledge Discovery Handbook*, pages 131–147. Kluwer Academic Publishers, 2005.

[24] P. Berkhin. A survey of clustering data mining techniques. In *Grouping multidimensional data*, pages 25–71. 2006.

[25] J. Bloch. *Effective Java (2nd Edition)*. Addison-Wesley, 2008.

[26] B. W. Boehm. A spiral model of software development and enhancement. *Computer*, 21(5):61–72, 1988.

[27] B. W. Boehm, J. R. Brown, H. Kaspar, M. Lipow, G. J. MacLeod, and M. J. Merrit. *Characteristics of Software Quality*, volume 1. Elsevier Science Ltd, 1978.

[28] B. W. Boehm, J. R. Brown, and M. Lipow. Quantitative evaluation of software quality. In *Proceedings of the 2nd international conference on Software engineering*, pages 592–605, San Francisco, California, United States, 1976. IEEE Computer Society Press.

[29] G. Box, G. Jenkins, and G. Reinsel. *Time series analysis: forecasting and control*. Wiley, 2013.

[30] M. M. Breunig, H.-P. Kriegel, R. T. Ng, and J. Sander. Local Outlier Factor: Identifying Density-Based Local Outliers. *ACM SIGMOD Record*, 29(2):93–104, June 2000.

[31] N. Brown, Y. Cai, Y. Guo, R. Kazman, M. Kim, P. Kruchten, E. Lim, A. MacCormack, R. Nord, and I. Ozkaya. Managing technical debt in software-reliant systems. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, pages 47–52, 2010.

[32] O. Burn. Checkstyle, 2001.

[33] G. Canfora and L. Cerulo. Impact analysis by mining software and change request repositories. In *11th IEEE International Symposium on Software Metrics*, pages 1–9, 2005.

[34] S. Cateni, V. Colla, and M. Vannucci. Outlier Detection Methods for Industrial Applications. *Advances in robotics, automation and control*, pages 265–282, 2008.

[35] V. Chandola, A. Banerjee, and V. Kumar. Outlier Detection : A Survey. *ACM Computing Surveys*, 2007.

[36] A. Clauset, C. R. Shalizi, and M. E. Newman. Power-law distributions in empirical data. *SIAM review*, 51(4):661–703, 2009.

[37] W. Cleveland. *Visualizing data*. Hobart Press, 1993.

[38] G. Concas, M. Marchesi, S. Pinna, and N. Serra. Power-Laws in a Large Object-Oriented Software System. *IEEE Transactions on Software Engineering*, 33(10):687–708, 2007.

[39] D. Cook and D. Swayne. *Interactive and Dynamic Graphics for Data Analysis: With R and GGobi*. 2007.

[40] R. Cooper and T. Weekes. *Data, models, and statistical analysis*. 1983.

[41] M. J. Crawley. *The R Book*. Wiley, 2007.

[42] P. B. Crosby. *Quality is free: The art of making quality certain*, volume 94. McGraw-Hill New York, 1979.

[43] W. Cunningham. The WyCash portfolio management system. *ACM SIGPLAN OOPS Messenger*, 4(2):29–30, 1992.

[44] D. Defays. An efficient algorithm for a complete link method. *The Computer Journal*, 20(4):364–366, Apr. 1977.

[45] F. Deissenboeck, E. Juergens, K. Lochmann, and S. Wagner. Software quality models: Purposes, usage scenarios and requirements. In *ICSE 2009 Workshop on Software Quality*, pages 9–14, 2009.

[46] W. Deming. *Out of the crisis: quality, productivity and competitive position*. Cambridge University Press, 1988.

[47] J. Deprez, K. Haaland, F. Kamseu, and U. de la Paix. QualOSS Methodology and QUALOSS assessment methods. *QualOSS project Deliverable*, 2008.

[48] J. Desharnais, A. Abran, and W. Suryn. Attributes Within ISO 9126: A Pareto Analysis. *Quality Software Management*, 2009.

[49] M. Diaz and J. Sligo. How software process improvement helped Motorola. *IEEE Software*, 14(5):75–81, 1997.

[50] W. Dickinson, D. Leon, and A. Podgurski. Finding failures by cluster analysis of execution profiles. In *Proceedings of the 23rd international conference on Software engineering*, pages 339–348, 2001.

[51] D. Dixon-Peugh. PMD, 2003.

[52] P. Dokas, L. Ertoz, V. Kumar, A. Lazarevic, J. Srivastava, and P.-N. Tan. Data mining for network intrusion detection. In *NSF Workshop on Next Generation Data Mining*, pages 21–30, 2002.

[53] R. Dromey. Cornering the Chimera. *IEEE Software*, 13(1):33–43, 1996.

[54] R. G. Dromey. A Model for Software Product Quality. *IEEE Transactions on Software Engineering*, 21(2):146–162, 1995.

[55] F.-W. Duijnhouwer and C. Widdows. Capgemini Expert Letter Open Source Maturity Model. Technical report, 2003.

[56] N. Eickelman. An Insider's view of CMM Level 5. *IEEE Software*, 20(4):79–81, 2003.

[57] T. Eisenbarth. Aiding program comprehension by static and dynamic feature analysis. In *International Conference on Software Maintenance*, 2001.

[58] M. Ester, H. Kriegel, J. Sander, M. Wimmer, and X. Xu. Incremental clustering for mining in a data warehousing environment. In *Proceedings of the 24th Very Large Data Bases International Conference*, pages 323–333, 1998.

[59] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. *Knowledge Discovery in Database*, 96:226–231, 1996.

[60] O. Fedotova, L. Teixeira, and H. Alvelos. Software Effort Estimation with Multiple Linear Regression: review and practical application. *Journal of Information Science and Engineering*, 2011.

[61] A. V. Feigenbaum. *Total Quality Control 4th edition.* McGraw-Hill Professional, 2006.

[62] N. Fenton. Software Measurement: a Necessary Scientific Basis. *IEEE Transactions on Software Engineering*, 20(3):199–206, Mar. 1994.

[63] N. Fenton and M. Neil. Software metrics: roadmap. In *Proceedings of the Conference on the Future of Software Engineering*, pages 357–370, 2000.

[64] N. Fenton and N. Ohlsson. Quantitative analysis of faults and failures in a complex software system. *IEEE Transactions on Software Engineering*, 26(8):797–814, 2000.

[65] P. Filzmoser, R. Maronna, and M. Werner. Outlier identification in high dimensions. *Computational Statistics & Data Analysis*, 52(3):1694–1711, 2008.

[66] L. Finkelstein and M. Leaning. A review of the fundamental concepts of measurement. *Measurement*, 2(1):25–34, Jan. 1984.

[67] M. Fowler. Martin Fowler on Technical Debt, 2009.

[68] J. French. Field experiments: Changing group productivity. *Experiments in social process: A symposium on social psychology*, 81:96, 1950.

[69] Friedrich Leisch. Sweave. Dynamic generation of statistical reports using literate data analysis. Technical Report 69, SFB Adaptive Information Systems and Modelling in Economics and Management Science, WU Vienna University of Economics and Business, Vienna, 2002.

[70] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[71] D. A. Garvin. *Managing Quality - the strategic and competitive edge.* Free Press [u.a.], New York, NY, 1988.

[72] M. Goeminne and T. Mens. Evidence for the pareto principle in open source software activity. In *Joint Proceedings of the 1st International workshop on Model Driven Software Maintenance and 5th International Workshop on Software Quality and Maintainability*, pages 74–82, 2011.

[73] A. Gordon. *Classification, 2nd Edition*. Chapman and Hall/CRC, 1999.

[74] M. K. Gounder, M. Shitan, and R. Imon. Detection of outliers in non-linear time series: A review. *Festschrift in honor of distinguished professor Mir Masoom Ali on the occasion of his retirement*, pages 18—-19, 2007.

[75] H. Guéhéneuc, Y.G. and Guyomarc'h, J.Y. and Khosravi, K. and Sahraoui. Design patterns as laws of quality. *Object-oriented Design Knowledge: Principles, Heuristics, and Best Practices*, pages 105–142, 2006.

[76] A. Gupta, B. R. Mohan, S. Sharma, R. Agarwal, and K. Kavya. Prediction of Software Anomalies using Time Series Analysis – A recent study. *International Journal on Advanced Computer Theory and Engineering (IJACTE)*, 2(3):101–108, 2013.

[77] D. A. Gustafson and B. Prasad. Properties of software measures. In *Formal Aspects of Measurement, Proceedings of the BCS-FACS Workshop on Formal Aspects of Measurement, South Bank University, London, 5 May 1991*, pages 179–193. Springer, 1991.

[78] T. Gyimothy, R. Ferenc, and I. Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software Engineering*, 31(10):897–910, 2005.

[79] K. Haaland, A.-k. Groven, N. Regnesentral, R. Glott, and A. Tannenberg. Free/Libre Open Source Quality Models - a comparison between two approaches. In *4th FLOS International Workshop on Free/Libre/Open Source Software*, volume 0, pages 1–2, 2010.

[80] T. Haley. Software process improvement at Raytheon. *IEEE Software*, 13(6):33–41, 1996.

[81] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell. A Systematic Literature Review on Fault Prediction Performance in Software Engineering. pages 1–31, 2010.

[82] M. H. Halstead. *Elements of Software Science*. Elsevier Science Inc., 1977.

[83] H.-J. Happel and W. Maalej. Potentials and challenges of recommendation systems for software development. In *Proceedings of the 2008 international workshop on Recommendation systems for software engineering*, pages 11–15. ACM, 2008.

[84] L. Hatton. The chimera of software quality. *Computer*, 40(8):103–104, 2007.

[85] D. Hawkins. *Identification of outliers*, volume 11. Chapman and Hall London, 1980.

[86] H. Hemmati, S. Nadi, O. Baysal, O. Kononenko, W. Wang, R. Holmes, and M. W. Godfrey. The MSR Cookbook. In *10th International Workshop on Mining Software Repositories*, pages 343–352, 2013.

[87] C. Hennig, T. Harabasz, and M. Hennig. Package 'fpc': Flexible procedures for clustering, 2012.

[88] S. Herbold, J. Grabowski, H. Neukirchen, and S. Waack. Retrospective Analysis of Software Projects using k-Means Clustering. In *Proceedings of the 2nd Design for Future 2010 Workshop (DFF 2010), May 3rd 2010, Bad Honnef, Germany*, 2010.

[89] I. Herraiz. A Statistical Examination of the Evolution and Properties of Libre Software. In *Proceedings of the 25th IEEE International Conference on Software Maintenance (ICSM)*, pages 439–442. IEEE Computer Society, 2009.

[90] I. Herraiz, J. M. Gonzalez-Barahona, and G. Robles. Forecasting the Number of Changes in Eclipse Using Time Series Analysis. In *Fourth International Workshop on Mining Software Repositories (MSR'07:ICSE Workshops 2007)*, pages 32–32. IEEE, May 2007.

[91] I. Herraiz, J. M. Gonzalez-Barahona, G. Robles, and D. M. German. On the prediction of the evolution of libre software projects. In *IEEE International Conference on Software Maintenance*, pages 405–414, 2007.

[92] V. Hodge and J. Austin. A survey of outlier detection methodologies. *Artificial Intelligence Review*, 22(2):85–126, 2004.

[93] M. Ing and E. Georgiadou. Software Quality Model Requirements for Software Quality Engineering. In *14th International Conference on the Software Quality Requirement*, 2003.

[94] ISO. ISO/IEC 9126 Software Engineering - Product Quality - Parts 1-4. Technical report, ISO/IEC, 2005.

[95] G. S. D. S. Jayakumar and B. J. Thomas. A New Procedure of Clustering Based on Multivariate Outlier Detection. *Journal of Data Science*, 11:69–84, 2013.

[96] C. Jensen and W. Scacchi. Applying a Reference Framework to Open Source Software Process Discovery. In *Proceedings of the 1st Workshop on Open Source in an Industrial Context*, 2003.

[97] C. Jensen and W. Scacchi. Data mining for software process discovery in open source software development communities. *Proc. Workshop on Mining Software Repositories*, pages 96–100, 2004.

[98] M. Jiang, S. Tseng, and C. Su. Two-phase clustering process for outliers detection. *Pattern recognition letters*, 22(6):691–700, 2001.

[99] S. R. G. Jones. Was There a Hawthorne Effect? *American Journal of Sociology*, 98(3):451, Nov. 1992.

[100] H.-W. Jung, K. Seung-Gweon, and C.-S. Chung. Measuring Software Product Quality: A Survey of ISO/IEC 9126. *IEEE Software*, 21(05):88–92, Sept. 2004.

[101] J. M. Juran, F. M. Gryna, and R. Bingham. *Quality control handbook*. McGraw-Hill, third edition, 1974.

[102] H. Kagdi, M. Collard, and J. Maletic. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *Journal of Software Maintenance and Evolution: Research and Practice*, 19(2):77–131, 2007.

[103] S. H. Kan. *Metrics and Models in Software Quality Engineering*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002.

[104] Y. Kanellopoulos. k-Attractors: A clustering algorithm for software measurement data analysis. In *19th IEEE International Conference on Tools with Artificial Intelligence*, pages 358–365, 2007.

[105] C. Kaner and W. P. Bond. Software engineering metrics: What do they measure and how do we know? *Methodology*, 8(6):1–12, 2004.

[106] N. Katayama and S. Satoh. The SR-tree: An index structure for high-dimensional nearest neighbor queries. *ACM SIGMOD Record*, 26(2):369–380, 1997.

[107] J. Kaur, S. Gupta, and S. Kundra. A k-means Clustering Based Approach for Evaluation of Success of Software Reuse. In *Proceedings of International Conference on Intelligent Computational Systems (ICICS'2011)*, 2011.

[108] D. Kelly and R. Sanders. Assessing the Quality of Scientific Software. In *First International workshop on Software Engineering for Computational Science and Engineering*, number May, Leipzig, Germany, 2008.

[109] T. M. Khoshgoftaar, E. B. Allen, W. D. Jones, and J. P. Hudepohl. Data mining for predictors of software quality. *International Journal of Software Engineering and Knowledge Engineering*, 9(5):547–563, 1999.

[110] K. Khosravi and Y. Guéhéneuc. On issues with software quality models. In *The Proceedings of the 11th Working Conference on Reverse Engineering*, pages 172–181, 2004.

[111] B. Kitchenham and S. Pfleeger. Software quality: the elusive target. *IEEE Software*, 13(1):12–21, 1996.

[112] D. E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, 1984.

[113] M. Koller. Nonsingular subsampling for S-estimators with categorical predictors. Technical report, 2012.

[114] M. Koller and W. Stahel. Sharpening wald-type inference in robust regression for small samples. *Computational Statistics & Data Analysis*, 55(8):2504–2515, 2011.

[115] S. Kotsiantis and P. Pintelas. Recent advances in clustering: A brief survey. *WSEAS Transactions on Information Science and Applications*, 1(1):73–81, 2004.

[116] Y. Kou, C.-T. Lu, S. Sirwongwattana, and Y.-P. Huang. Survey of fraud detection techniques. In *2004 IEEE International Conference on Networking, Sensing and Control*, pages 749–754, 2004.

[117] P. Kruchten, R. Nord, and I. Ozkaya. Technical debt: from metaphor to theory and practice. *IEEE Software*, 29(6):18–21, 2012.

[118] M. Lanza and R. Marinescu. *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems*. 2006.

[119] M. Last, M. Friedman, and A. Kandel. The data mining approach to automated software testing. In *Proceedings of the 9th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 388–396, 2003.

[120] J. Laurikkala, M. Juhola, and E. Kentala. Informal identification of outliers in medical data. In *Proceedings of the 5th International Workshop on Intelligent Data Analysis in Medicine and Pharmacology*, pages 20–24, 2000.

[121] A. Lazarevic, L. Ertoz, and V. Kumar. A comparative study of anomaly detection schemes in network intrusion detection. *Proc. SIAM*, 2003.

[122] J.-L. Letouzey. The SQALE method for evaluating Technical Debt. In *Third International Workshop on Managing Technical Debt (MTD)*, pages 31–36. IEEE, 2012.

[123] J.-L. Letouzey and T. Coq. The sqale analysis model: An analysis model compliant with the representation condition for assessing the quality of software source code. In *Second International Conference on Advances in System Testing and Validation Lifecycle*, pages 43–48, 2010.

[124] S. D. Levitt and J. A. List. Was There Really a Hawthorne Effect at the Hawthorne Plant? An Analysis of the Original Illumination Experiments. *American Economic Journal: Applied Economics*, 3(1):224–238, Jan. 2011.

[125] T. W. Liao. Clustering of time series data — a survey. *Pattern Recognition*, 38(1):1857–1874, 2005.

[126] R. Lincke, J. Lundberg, and W. Löwe. Comparing software metrics tools. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, pages 131–142, New York, New York, USA, 2008. ACM Press.

[127] X. Liu, G. Kane, and M. Bambroo. An intelligent early warning system for software quality improvement and project management. *Journal of Systems and Software*, 2006.

[128] P. Louridas. Static Code Analysis. *IEEE Software*, 23(4):58–61, 2006.

[129] P. Louridas, D. Spinellis, and V. Vlachos. Power laws in software. *ACM Transactions on Software Engineering and Methodology*, 18(1):1–26, Sept. 2008.

[130] T. McCabe. A complexity measure. *Software Engineering, IEEE Transactions on*, (4):308–320, 1976.

[131] J. McCall. *Factors in Software Quality: Preliminary Handbook on Software Quality for an Acquisiton Manager*. Information Systems Programs, General Electric Company, 1977.

[132] M. Mendonca and N. Sunderhaft. Mining software engineering data: A survey. Technical report, 1999.

[133] D. C. Montgomery, E. A. Peck, and G. G. Vining. *Introduction to Linear Regression Analysis, 4th edition Student Solutions Manual (Wiley Series in Probability and Statistics)*. Wiley-Interscience.

[134] F. Murtagh. A survey of recent advances in hierarchical clustering algorithms. *The Computer Journal*, 26(4):354–359, 1983.

[135] N. Nagappan, T. Ball, and B. Murphy. Using historical in-process and product metrics for early estimation of software failures. In *Proceedings of the 17th International Symposium on Software Reliability Engineering*, pages 62–74, 2006.

[136] B. B. Naib. An Improved Clustering Approach for Software Quality Analysis. *International Journal of Engineering, Applied and Management Sciences Pradigms*, 05(01):96–100, 2013.

[137] Navica Software. The Open Source Maturity Model is a vital tool for planning open source succes. Technical report, 2009.

[138] L. C. Noll, S. Cooper, P. Seebach, and A. B. Leonid. The International Obfuscated C Code Contest.

[139] D. Okanović and M. Vidaković. Software Performance Prediction Using Linear Regression. In *Proceedings of the 2nd International Conference on Information Society Technology and Management*, pages 60–64, 2012.

[140] A. Origin. Method for Qualification and Selection of Open Source software (QSOS), version 1.6. Technical report, 2006.

[141] H. M. Parsons. What Happened at Hawthorne?: New evidence suggests the Hawthorne effect resulted from operant reinforcement contingencies. *Science (New York, N.Y.)*, 183(4128):922–32, Mar. 1974.

[142] M. Paulk, B. Curtis, M. Chrissis, and C. Weber. Capability Maturity Model, version 1.1. Technical report, 1993.

[143] M. C. Paulk, K. L. Needy, and J. Rajgopal. Identify outliers, understand the Process. *ASQ Software Quality Professional*, 11(2):28–37, 2009.

[144] E. Petrinja, R. Nambakam, and A. Sillitti. Introducing the OpenSource Maturity Model. In *2009 ICSE Workshop on Emerging Trends in Free/Libre/Open Source Software Research and Development*, pages 37–41. IEEE, May 2009.

[145] C. Phua, V. Lee, K. Smith, and R. Gayler. A Comprehensive Survey of Data Mining-based Fraud Detection Research. *arXiv preprint arXiv:1009.6119*, page 14, Sept. 2010.

[146] D. Posnett, V. Filkov, and P. Devanbu. Ecological inference in empirical software engineering. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 362–371. IEEE Computer Society, Nov. 2011.

[147] R Core Team. R: A Language and Environment for Statistical Computing, 2013.

[148] E. Raymond. The cathedral and the bazaar. *Knowledge, Technology & Policy*, 1999.

[149] W. J. Reed. Power-Law Adjusted Survival Models. *Communications in Statistics-Theory and Methods*, 41(20):3692–3703, 2012.

[150] M. Riaz, E. Mendes, and E. Tempero. A Systematic Review of Software Maintainability Prediction and Metrics. In *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, pages 367–377, 2009.

[151] M. Robillard and R. Walker. Recommendation systems for software engineering. *IEEE Software*, 27(4):80–86, 2010.

[152] W. Robinson. Ecological correlations and the behavior of individuals. *International journal of epidemiology*, 15(3):351–357, 2009.

[153] P. Rousseeuw, C. Croux, V. Todorov, A. Ruckstuhl, M. Salibian-Barrera, T. Verbeke, M. Koller, and M. Maechler. robustbase: Basic Robust Statistics. R package version 0.9-10., 2013.

[154] N. Rutar, C. B. Almazan, and J. S. Foster. A comparison of bug finding tools for Java. In *15th International Symposium on Software Reliability Engineering*, pages 245–256, 2004.

[155] J. A. Ryan and J. M. Ulrich. xts: eXtensible Time Series, 2013.

[156] T. L. Saaty. *Fundamentals of decision making and priority theory: with the analytic hierarchy process*. Rws Publications, 1994.

[157] I. Samoladas, G. Gousios, D. Spinellis, and I. Stamelos. The SQO-OSS quality model: measurement-based open source software evaluation. In *Open source development, communities and quality*, pages 237–248. Springer, 2008.

[158] N. Schneidewind. Methodology for validating software metrics. *IEEE Transactions on Software Engineering*, 18(5):410–422, 1992.

[159] M. Shaw. Prospects for an Engineering Discipline of Software. *IEEE Software*, (November):15–24, 1990.

[160] W. Shewhart. *Economic control of quality of manufactured product*. Van Nostrand, 1931.

[161] Shyam R. Chidamber and Chris F. Kemerer. A Metrics Suite for Object Oriented Design. 1993.

[162] R. Sibson. SLINK: An optimally efficient algorithm for the single-link cluster method. *The Computer Journal*, 16(1):30–34, Jan. 1973.

[163] E. Simmons. When Will We be Done Testing? Software Defect Arrival Modeling Using the Weibull Distribution. In *Northwest Software Quality Conference, Portland, OR*, 2000.

[164] E. Simmons. Defect Arrival Modeling Using the Weibull Distribution. *International Software Quality Week, San Francisco, CA*, 2002.

[165] G. Singh and V. Kumar. An Efficient Clustering and Distance Based Approach for Outlier Detection. *International Journal of Computer Trends and Technology*, 4(7):2067–2072, 2013.

[166] K. Singh and S. Upadhyaya. Outlier Detection : Applications And Techniques. In *International Journal of Computer Science*, volume 9, pages 307–323, 2012.

[167] J. F. Smart. *Java Power Tools*. O'Reilly Media, 2008.

[168] R. Sokal and C. Michener. A statistical method for evaluating systematic relationships. *University of Kansas Scientific Bulletin*, 28:1409–1438, 1958.

[169] S. Sowe, R. Ghosh, and K. Haaland. A Multi-Repository Approach to Study the Topology of Open Source Bugs Communities: Implications for Software and Code Quality. In *3rd IEEE International Conference on Information Management and Engineering*, 2011.

[170] M.-A. Storey. Theories, methods and tools in program comprehension: Past, present and future. In *13th International Workshop on Program Comprehension (IWPC 2005)*. IEEE Computer Society, 2005.

[171] Sun. Code Conventions for the Java Programming Language. Technical report, 1999.

[172] Q. Taylor and C. Giraud-Carrier. Applications of data mining in software engineering. *International Journal of Data Analysis Techniques and Strategies*, 2(3):243–257, July 2010.

[173] C. P. Team. CMMi® for Development, Version 1.1. Technical report, 2002.

[174] C. P. Team. CMMI® for Development, Version 1.3. Technical report, Carnegie Mellon University, 2010.

[175] P. Teetor. *R Cookbook*. O'Reilly Publishing, 2011.

[176] V. Todorov and P. Filzmoser. An Object-Oriented Framework for Robust Multivariate Analysis. *Journal of Statistical Software*, 32(3):1–47, 2009.

[177] L. Torgo. *Data Mining with R, learning with case studies*. Chapman and Hall/CRC, 2010.

[178] F. Tsai and K. Chan. Dimensionality reduction techniques for data exploration. In *6th International Conference on Information, Communications & Signal Processing*, pages 1–5. IEEE, 2007.

[179] M. Turan and Z. \c{C}ataltepe. Clustering and dimensionality reduction to determine important software quality metrics. In *22nd International Symposium on Computer and Information Sciences*, pages 1–6, Nov. 2007.

[180] W. N. Venables and B. D. Ripley. *Modern Applied Statistics with S*. Springer, 2002.

[181] A. Vetro', M. Torchiano, and M. Morisio. Assessing the precision of FindBugs by mining Java projects developed at a university. In *7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, pages 110–113. Ieee, May 2010.

[182] J. Voas. Assuring software quality assurance. *IEEE Software*, 20(3):48–49, 2003.

[183] S. Wagner, F. Deissenboeck, M. Aichner, J. Wimmer, and M. Schwalb. An Evaluation of Two Bug Pattern Tools for Java. In *2008 International Conference on Software Testing, Verification, and Validation*, pages 248–257. Ieee, Apr. 2008.

[184] A. I. Wasserman, M. Pal, and C. Chan. Business Readiness Rating Project. Technical report, 2005.

[185] L. Westfall and C. Road. 12 Steps to Useful Software Metrics. *Proceedings of the 17th Annual Pacific Northwest Software Quality Conference*, 57 Suppl 1(May 2006):S40–3, 2005.

[186] J. A. J. Wilson. Open Source Maturity Model. Technical report, 2006.

[187] C. Wohlin, M. Höst, and K. Henningsson. Empirical research methods in software engineering. *Empirical Methods and Studies in Software Engineering*, pages 7–23, 2003.

[188] W. Wong, A. Moore, G. Cooper, and M. Wagner. Bayesian network anomaly pattern detection for disease outbreaks. In *ICML*, pages 808–815, 2003.

[189] T. Xie. Bibliography on mining software engineering data, 2010.

[190] T. Xie, S. Thummalapenta, D. Lo, and C. Liu. Data mining for software engineering. *IEEE Computer*, 42(8):35–42, 2009.

[191] Y. Xie. Knitr: A general-purpose package for dynamic report generation in R. Technical report, 2013.

[192] Q. Yang and X. Wu. 10 challenging problems in data mining research. *International Journal of Information Technology & Decision Making*, 5(4):597–604, 2006.

[193] C. T. Yiannis Kanellopoulos. Interpretation of Source Code Clusters in Terms of the ISO/IEC-9126 Maintainability Characteristics. In *12th European Conference on Software Maintenance and Reengineering*, pages 63–72, 2008.

[194] A. Ying. *Predicting source code changes by mining revision history.* PhD thesis, 2003.

[195] A. Ying and G. Murphy. Predicting source code changes by mining change history. *IEEE Transactions on Software Engineering*, 30(9):574–586, 2004.

[196] V. Yohai. High breakdown-point and high efficiency robust estimates for regression. *The Annals of Statistics*, 15:642–65, 1987.

[197] K.-A. Yoon, O.-S. Kwon, and D.-H. Bae. An Approach to Outlier Detection of Software Measurement Data using the K-means Clustering Method. In *Empirical Software Engineering and Measurement*, pages 443–445, 2007.

[198] H. Zhang. On the distribution of software faults. *Software Engineering, IEEE Transactions on*, 2008.

[199] T. Zhang, R. Ramakrishnan, and M. Livny. BIRCH: A new data clustering algorithm and its applications. *Data Mining and Knowledge Discovery*, 1(2):141–182, 1997.

[200] S. Zhong, T. Khoshgoftaar, and N. Seliya. Analyzing software measurement data with clustering techniques. *IEEE Intelligent Systems*, 19(2):20–27, Mar. 2004.

[201] S. Zhong, T. Khoshgoftaar, and N. Seliya. Unsupervised learning for expert-based software quality estimation. In *Proceeding of the Eighth IEEE International Symposium on High Assurance Systems Engineering*, pages 149–155, 2004.

[202] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, 31(6):429–445, 2005.

Appendices

Appendix A

Papers and articles published

SQuORE: a new approach to software project assessment.

Boris Baldassari
SQuORING Technologies
76, Alles Jean Jaurs,
31000 Toulouse - France
www.squoring.com
boris.baldassari@squoring.com

July 10, 2013

Abstract

Quality has a price. But non-quality is even more expensive. Knowing the cost and consequences of software assets, being able to understand and control the development process of a service, or quickly evaluating the quality of external developments are of primary importance for every company relying on software. Standards and tools have tried with varying degrees of success to address these concerns, but there are many difficulties to be overcome: the diversity of software projects, the measurement process – from goals and metrics selection to data presentation, or the user's understanding of the reports. These are situations where the SQuORE business intelligence tool introduces a novel decision-based approach to software projects quality assessment by providing a more reliable, more intuitive, and more context-aware view on quality. This in turn allows all actors of the project to share a common vision of the project progress and performance, which then allows efficient enhancing of the product and process. This position paper presents how SQuORE solves the quality dilemma, and showcases two real-life examples of industrial projects: a unit testing improvement program, and a fully-featured software project management model.

Key words: software quality, key performance indicators, trend analysis, measurement, quality models, process evaluation, business intelligence, project management.

1 Introduction

Despite an increasing interest in software quality, many still think about quality achievement as an expensive and unproductive process. On the other hand, as Shaw [23] pointed out a few years earlier, the software engineering discipline is currently in the process of maturing: in the past decade, new methods and new processes have grown, standards have been published, and the many years of experience gathered in the field brought much feedback and a new maturity to the discipline. The time has come for a new era in business intelligence [5] for software projects.

In the second section of this paper, we lay down the ground foundations of our approach to software measurement and present the state of practice, along with some terms and concepts about software development quality and management. In section three, we discuss the SQuORE approach to software projects assessment, with its features and benefits. Section four

further expands the scope of quality assessment by showing two real-life implementations that demonstrate the use of SQuORE, with unit testing prioritisation and project- and schedule dashboards and models.

2 State of practice

2.1 The cost of failures

There are many examples of software project failures, and their associated costs – either in human or financial losses. All of them originate from non-quality or lack of control on development. Some well-known example bugs in the past decades include:

- Therac-25: six deaths before beeing fixed, took two years to diagnose and fix [15].
- Mars Climate Orbiter: race conditions on bus priority, system rebooted continuously and robot eventually crashed[19].
- Patriot missile target shifted 6 meters every hour due to float precision bug. 28 soldiers killed in Dhahran [25].
- Ariane 5 infamous buffer overrun crash due to abusive reuse of Ariane 4 software[16]: 500 millions $ pure loss.
- AT & T failure of 1990: software upgrade of switch network led to a 9 hours crash, traced back to a missing break[20]. 60 Million $ lost revenue.

According to a study conducted by the U.S. Department of Commerce's National Institute of Standards and Technology (NIST), the bugs and glitches cost the U.S. economy about 59.5 billion dollars a year[21]. The Standish Group CHAOS 2004 [24] report shows failure rates of almost 70%.

2.2 Standards for software products and processes

Many quality-oriented standards and norms have been published in the last decades, some focusing on product quality (from Boehm [1] and McCall [18], further simplified and enhanced by ISO 9126 [9]), while other rather consider the process quality (e.g. ISO/IEC 15504 [6] and CMMi [2]). More recently, two quality assessment standards have been developed: SQALE [14, 13], a generic method independent of the language and source code analysis tools, mainly relying on technical and design debts, and ISO SQuARE [7], the successor of ISO 9126, which is still being developed. Furthermore, some domains have their own de-facto standards: HIS and ISO 26262 [8] for the automotive industry, or DO-178 [22] for the aeronautics and critical embedded systems.

But some objections have been opposed to established standards, because:

- they may be seen as mere gut-feeling and opinions from experts, as pointed out by Jung et al [10] for the ISO 9126, and
- they dont fit well every situation and view on quality, as they are rather scope-fixed [11].

Another point, also related to the complexity and diversity of software projects, is that published standards dont provide any pragmatic measures or tool references, which leads to misunderstanding and misconceptions of what is measured as an example, consider the thousands of different ideas and concepts behind the Mean Time To Failure metric [12].

2.3 Metrics for software quality measurement

There is a huge amount of software-oriented metrics available in the literature. Examples of weidly-used metrics include McCabe's cyclomatic complexity for control flow[17], Halstead's complexity for data flow[4], size or coupling measures. Each measure is supposed to characterise some attributes of software quality, and they have to be put together to give the complete picture.

Another mean to assess software quality is the number of *non-conformities* to a given reference. As an example, if naming or coding conventions or programming patterns have been decided, then any violations of these rules is supposed to decrease the

quality, because it threatens some of the characteristics of quality, like analysability (for conventions), or reliability (for coding patterns).

The concept of *technical debt*, coined by Ward Cunningham in 1992 [3] and gaining more and more interest nowadays, can be considered as the distance to the desired state of quality. In that sense, it is largely driven by the number and importance of non-conformities.

The trend of software measurement globally tends to multi-dimensional analysis [12]: quality or progress of a software project or product is a composite of many different measures, reflecting its different characteristics or attributes. The next step is the way information can be aggregated and consolidated.

3 Principles of SQuORE

The purpose of SQuORE is to retrieve information from several sources, compute a consolidation of the data, and show an optimised report on software or project state. The rating of the application is displayed on a common, eye-catching 7-steps scale which allows immediate understanding of the results, as shown in Figure 1.

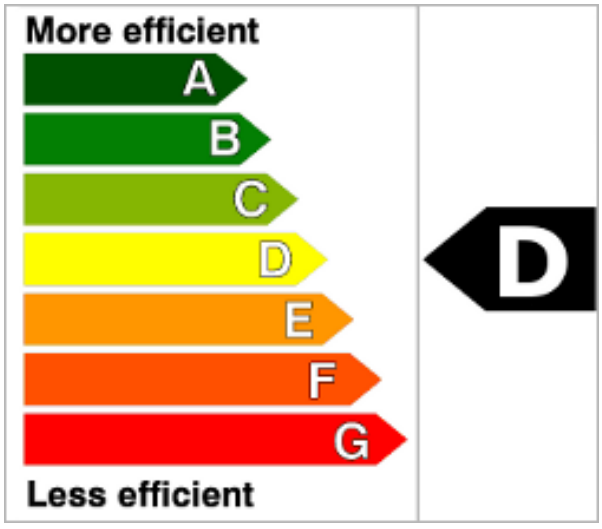


Figure 1: The 7-levels SQuORE rating

3.1 Architecture

SQuORE analysis process can be broken down in three separate steps: *data providers* that take care of gathering inputs from different sources, the *engine*, which computes the consolidation and rating from the base measures gathered by data providers, and the *dashboard* to present information in a smart and efficient way.

3.2 Data Providers

As stated in our introduction, there are nowadays many tools available, each one having a specific domain of expertise and an interesting, but partial, view on quality. SQuORE brings in the glue and consistency between them all, by importing this information any type of input is accepted, from xml or csv to Microsoft binary files and processing it globally.

The SQuORE analyser runs first. It is fast, does not need third-party dependencies, and constitutes a tree of artefacts corresponding to the items measured: source code, tests, schedule, hardware components, or more generally speaking any kind of item able to represent a node of the project hierarchy. In addition, the SQuORE engine adds the findings and information from external tools, attaching them to the right artefacts with their values and meaning.

3.3 Data Consolidation

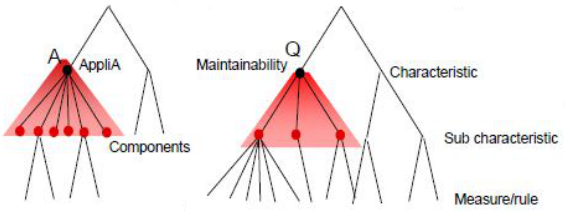


Figure 2: Artefacts and Quality Trees

Once the base measures have been collected, SQuORE computes derived measures as defined in the quality model and builds the quality hierarchy for every node of the artefact tree, as shown in Figure 2.

3.3.1 Quality Models

Quality models define how data are aggregated and summarised from the leaves up to the root artefact (usually the application or project): base measures collected by data providers are transformed into derived measures and associated to the different attributes of quality.

As stated before, there is no silver bullet for quality models[11]: one has to tailor the assessment method, considering the specific needs and goals of the development. In many cases existing models[1] constitute a good start, and they should simply be fine-tuned to fit most common needs. But for specific or uncommon situations, SQuORE proposes everything one would need in such a task, from basic operations on measures to complex, cross-tree computations.

3.3.2 Metris, Scales, Indicators

Raw measures give the status of a characteristic, without qualitative judgement. *Indicators* give this information, by comparing the measure to a scale. *Scales* define specific levels for the measure and their associated rating, which allow fine-tuning the model with specific thresholds and weights. As an example, the well-known cyclomatic complexity metric [17] for functions could be compared to a four levels scale, such that:

- from 0 to 7 rating is A (very good) and weight for technical debt is 0,
- from 7 to 15 rating is B (ok) and weight is 2,
- from 15 to 25 rating is C (bad) and weight is 4,
- above 25 rating is D (very bad) and weight is 16 because you really should refactor it.

Considering this, the cyclomatic complexity indicator gives at first sight the status of the intended meaning of the metric.

[1] The default SQuORE setup proposes several models and standards: SQALE, ISO 9126 Maintainability and Automotive HIS are available right out-of-the-box.

3.3.3 Action Items

Quite often, the dynamics of development depend on many different factors: as an example, if a function is quite long, has an important control complexity and many non-conformities, and a poor comment rate, then it should be really looked at although none of these individual indicators, taken separately, would be worse rising it. Action items serve this goal: the quality officer can define triggers, which can be any combination of indicators on the artefact tree, and SQuORE will create action items if one or all criteria are met. A helpful description about the problem and its resolution is displayed as well. Because they can be applied on any artefact type, the possibilities of action items are almost limitless. They are often the best way to implement experience-based heuristics, and automate long, tedious, and error-prone checking processes.

3.4 From Quality Assessment to Project Monitoring

Quality assessment, as a static figure, is the first step to project control: if you don't know where you are, a map won't help. The next step is to monitor the dynamics of the project, by following the evolution of this static status across iterations – this can be thought of as search-based decision making, as described by Hassan et al. in [5]. SQuORE proposes for this several mechanisms:

- Trends show at first sight how an artefact or attribute of quality did evolve.
- Drill-downs, sorts and filters help identify quickly *what artefacts* actually went wrong.
- The quality view helps understand *why* the rating went down, and what should be done to get it back to a good state.
- Action items help identifying complex evolution schemas, by specifying multiple criteria based on artefacts, measures and trends.

4 Use Cases

4.1 General Feedback

From our experience, there are some common reactions to a SQuORE evaluation:

- People are able to quickly identify issues and are not overwhelmed by the amount of information, which allows finding in a few minutes serious issues like missing breaks. Specialised tools are indeed able to uncover such issues, but due to the sheer volume of results they generate, it is not uncommon for end users to miss important results.

- People are glad to see that their general feeling about some applications is verified by pragmatic evidence. This re-enforces the representativeness of measurement and puts facts on words[2].

- Developers are concerned by their rating: the simple, eye-catching mark is immediately recognised as a rating standard. Further investigations help them understand why it is so, and how they could improve it[3].

4.2 Unit Test Monitoring

One of our customers needed to know what parts of software had to be tested first for maximum efficiency. Until now, the process was human-driven: files to be tested were selected depending on their history and recent changes, complexity of their functions, and their number of non-conformities. The team had developed home-grown heuristics gathered from years of experience, with defined attributes and thresholds.

We built a model with inputs from two external tools, QAC and Rational Test Real Time, and the SQuORE analyser. From these, we defined four main measures: non-conformities from QAC, RTRT, and SQuORE, plus cyclomatic complexity. Weights and computations were then selected in such a way that the files would get exponential-like ratings: as an example, if a file had only one of these main measures marked as bad, it would get a weight of 2. For two, three or four bad measures, it would get resp. a weight of 8, 16 or 32. This allowed quickly identifying the worst files in the artefact hierarchy.

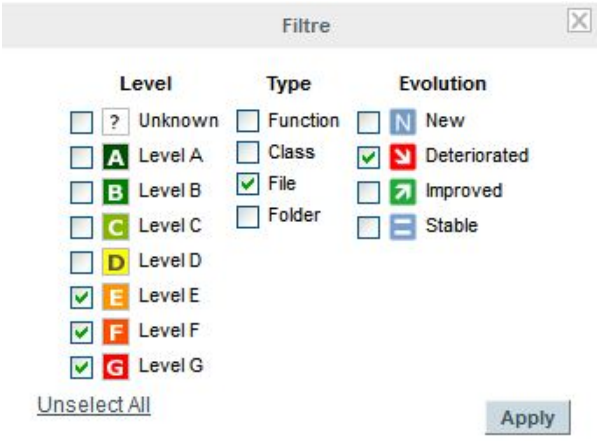


Figure 3: Artefact filters

Folder ratings were computed according to the number of bad files they had under their hierarchy and their relative badness, which allowed focusing on worst components easily by sorting the artefact tree by folder ratings.

Action items were setup for the files that really needed to be checked, because they had either really bad ratings, or cyclomatic complexities or number of non-conformities that exceeded by far the defined criteria. Such action items allowed identifying problematic files hidden in hierarchies of good or not-so-bad files.

We were able to achieve the following goals:

- Define a standardised, widely-accepted mean of estimating testing efforts.

- Reproduce some gut-feeling mechanisms that had been thoroughly experienced and fine-tuned along the years by human minds, without having been explicitly formalised until now.

[2] In other words: *I told you this code was ugly!*
[3] In other words: *I suspected this part needed refactoring.*

Dashboard graphs were setup to get immediate visual information on the rate of bad files in components. The evolution of component ratings also helped to identify parts of software that tended to entropy or bad testability, and take appropriate actions with development teams.

4.3 Project Monitoring

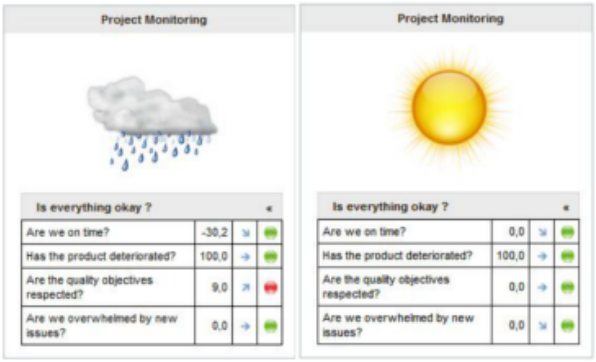


Figure 4: Example scorecards

Another experience encompassed a full software project monitoring solution: the customer wanted to have a factual and unified vision on the overall progress of his developments.

SQuORE analyser was used for the source code quality assessment. Additional data providers were defined to retrieve data from change management, tests, and scheduling (as the number of open/closed tasks) tools.

Considering these entries, we defined the following axes in the quality model, as shown in Figure 4:

- *Are the quality objectives respected?* – based on the ISO 9126 maintainability assessment of code.
- *Are we on-time?* – an agile-like reporting on tasks completion.
- *Are we overwhelmed by new issues?* – the amount of defect reports waiting for treatments.
- *Was the product deteriorated?* – which reflects the state of test coverage and success: regression, unit, and system tests.

These axes are summarised in a global indicator (Figure 5), showing the composite progress of the project. In this case, we chose to take the worst sub-characteristic as the global rating to directly identify problems on progress.

Action items were developed to identify:

- Parts of code that had a bad maintainability rating, were not enough covered by tests, and got many defect change requests.
- Schedule shifts, when the number of opened tasks was too high for the remaining time and many change requests were incoming (opened).

Figure 5: Project Quality model

The Jenkins continuous integration server was used to execute SQuORE automatically on a daily basis. Automatic retrieval of data and analysis was needed to ensure reliability and consistency of data – the constant availability being one of the keys to meaningful data.

Dashboards were defined for the main concerns of the project: evolution of the maintainability rating on files, evolution of change requests treatment, and failing tests. The scheduling information was represented using burn-down and burn-up charts, and dot graphs with trends and upper- and lower- limits.

We were able to:

- Provide real-time information about current state and progress of project. Web links to the Mantis change management system even allowed knowing exactly what actions are on-going.
- Get back confidence and control on project to team leaders and developers by enabling a crystal-clear, shared and consistent vision.

References

[1] B. W. Boehm, J. R. Brown, and M. Lipow. Quantitative evaluation of software quality. In *Proceedings of the 2nd international conference on Software engineering*, pages 592–605, San Francisco, California, United States, 1976. IEEE Computer Society Press.

[2] CMMI Product Team. CMMI for Development, Version 1.3. Technical report, Carnegie Mellon University, 2010.

[3] Martin Fowler. Martin Fowler on Technical Debt, 2004.

[4] Maurice H. Halstead. *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc., New York, NY, USA, 1977.

[5] Ahmed E Hassan and Tao Xie. Software Intelligence: The Future of Mining Software Engineering Data. In *Proc. FSE/SDP Workshop on the Future of Software Engineering Research (FoSER 2010)*, pages 161–166. ACM, 2010.

[6] ISO IEC. Iso/iec 15504-1 – information technology – process assessment. *Software Process: Improvement and Practice*, 2(1):35–50, 2004.

[7] ISO IEC. Iso/iec 25000 – software engineering – software product quality requirements and evaluation (square) – guide to square. *Systems Engineering*, page 41, 2005.

[8] ISO. ISO/DIS 26262-1 - Road vehicles Functional safety Part 1 Glossary. Technical report, International Organization for Standardization / Technical Committee 22 (ISO/TC 22), 2009.

[9] ISO/IEC. *ISO/IEC 9126 – Software engineering – Product quality.* 2001.

[10] Ho-Won Jung, Seung-Gweon Kim, and Chang-Shin Chung. Measuring software product quality: A survey of iso/iec 9126. *IEEE Software*, 21:88–92, 2004.

[11] Stephen H. Kan. *Metrics and Models in Software Quality Engineering.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002.

[12] C Kaner and Walter P. Bond. Software engineering metrics: What do they measure and how do we know? In *10Th International Software Metrics Symposium, METRICS 2004*, pages 1–12, 2004.

[13] Jean-louis Letouzey. The SQALE method for evaluating Technical Debt. In *2012 Third International Workshop on Managing Technical Debt*, pages 31–36, 2012.

[14] Jean-louis Letouzey and Thierry Coq. The SQALE Analysis Model An analysis model compliant with the representation condition for assessing the Quality of Software Source Code. In *2010 Second International Conference on Advances in System Testing and Validation Lifecycle (VALID)*, pages 43–48, 2010.

[15] Nancy Leveson and Clark S. Turner. An Investigation of the Therac-25 Accidents. *IEEE Computer*, 26(7):18–41, 1993.

[16] J.L. Lions. ARIANE 5 Flight 501 failure. Technical report, 1996.

[17] TJ McCabe. A complexity measure. *Software Engineering, IEEE Transactions on*, (4):308–320, 1976.

[18] J.A. McCall. *Factors in Software Quality: Preliminary Handbook on Software Quality for an Acquisiton Manager*. Information Systems Programs, General Electric Company, 1977.

[19] National Aeronautics and Space Administration. Mars Climate Orbiter Mishap Investigation Report. Technical report, National Aeronautics and Space Administration, Washington, DC, 2000.

[20] Peter G. Neumann. Cause of AT&T network failure. *The Risks Digest*, 9(62), 1990.

[21] National Institute of Standards and Department of Commerce. Technology (NIST). Software errors cost u.s. economy $59.5 billion annually, 2002.

[22] RTCA. DO-178B: Software Considerations in Airborne Systems and Equipment Certification. Technical report, Radio Technical Commission for Aeronautics (RTCA), 1982.

[23] Mary Shaw. Prospects for an engineering discipline of software. *IEEE Software*, (November):15–24, 1990.

[24] The Standish Group International Inc. The Standish Group International Inc. Chaos Technical report. Technical report, 2004.

[25] United States General Accounting Office. Patriot Missile Defense: Software Problem Led to System Failure at Dhahran , Saudi Arabia. Technical report, United States General Accounting Office, 1992.

SQuORE : une nouvelle approche de mesure de la qualité des projets logiciels

Boris Baldassari,

SQuORING Technologies, 76 Allées Jean Jaurès, 31000 Toulouse
Tel: +33 581 346 397, E-mail: boris.baldassari@squoring.com

Abstract : La qualité a un prix. Mais la non-qualité est encore plus chère. Connaître les coûts et la valeur du patrimoine logiciel, mieux comprendre les équipes de développement, ou évaluer rapidement la qualité d'un développement externalisé sont des activités centrales pour toute société dont le fonctionnement s'appuie sur l'informatique. Des standards et outils ont été mis au point pour adresser ces éléments, avec plus ou moins de succès, mais les difficultés sont nombreuses, de par la diversité des projets logiciels, la mise en place effective d'un processus de mesure – de la sélection des objectifs et métriques de mesure à l'affichage de l'information, ou la compréhension par les acteurs du processus qualité. C'est pour ces situations que l'outil SQuORE introduit une approche novatrice de la qualité logicielle, plus fiable, plus intuitive et plus contextuelle, appliquant les techniques de la Business Intelligence à l'analyse des projets logiciels. SQuORE offre ainsi une vision et une compréhension unifiées de l'état et des avancées de projets, permettant une amélioration pragmatique du processus et des développements. Cet article présente comment la solution SQuORE résout le dilemme de la qualité, et décrit deux cas issus de projets industriels : un programme de priorisation des tests unitaires, et un modèle de processus plus complet, s'appuyant sur de nombreux éléments hors code source.

1 Introduction

Malgré un intérêt croissant pour les notions de qualité logicielle, beaucoup la considèrent encore comme une activité coûteuse et non productive. Mary Shaw[23] a étudié en 1990 l'évolution de la discipline du génie logiciel, en la comparant avec d'autres sciences plus anciennes et plus matures (telles que l'architecture) ; le génie logiciel n'en est qu'à ses premiers pas en tant que science, et la discipline doit encore évoluer et gagner en maturité pour parfaire son développement. Mais de

nombreux progrès ont été faits ces dernières années, l'expérience s'accumule et l'heure est venue d'une nouvelle ère pour le développement logiciel, avec l'analyse multidimensionnelle [12] et la prise de décision par fouille de données [5].

Dans la seconde section de cet article, nous rappelons quelques termes et définitions, et récapitulons l'état de l'art et des pratiques actuelles. La section 3 décrit l'approche choisie pour SQuORE, ses fonctionnalités et avantages. La section 4 étend davantage le périmètre de l'analyse logicielle en montrant deux exemples industriels d'implémentation de modèles de qualité, pour les tests et pour la gestion de projet.

2 Etat de l'art

1 Coût de la non-qualité

Les exemples d'erreurs logicielles et leurs coûts associés ne manquent pas, que les pertes soient humaines ou financières. Toutes trouvent leur origine dans un manque de qualité et de contrôle sur le processus de développement. Quelques exemples connus sont reproduits ci-après :

- Therac-25 : surdosage de radiations sur un appareil médical : 6 morts et deux ans avant correction [15].
- Marc Climate Observer : durée des tests insuffisante, le robot s'est écrasé suite au remplissage de son espace mémoire [19].
- Missile Patriot : dérive de 6 mètres par heure due à un bug de conversion. 28 soldats tués à Dhahran [25].
- Ariane 5 : réutilisation abusive de routines écrites pour Ariane 4, 500 M$ de pure perte [16].
- Crash du réseau AT&T (90's) : une mise à jour des routeurs rend le réseau inutilisable pendant 9 heures, 60M$ de perte de revenus, cause du bug : un break manquant [20].

Le National Institute for Standards and Technology (NIST) a commandé en 2002 un rapport sur le coût estimé des problèmes logiciels pour l'industrie américaine : 60 Milliard de dollars par an. Le rapport technique CHAOS, édité par le Standish Group, faisait état en 2004 de taux d'échec des projets logiciels de 70%. Les chiffres sont un peu meilleurs depuis, mais le constat reste accablant.

2 Standards pour produits et processus logiciels

De nombreux standards et normes dédiés à la qualité ont été produits par les organismes de standardisation ces dernières décennies, certains adressant plutôt la qualité des produits (de Boehm [1] à McCall [18], simplifié et amélioré par l'ISO 9126 [9]), tandis que d'autres adressent plutôt la qualité du processus (ainsi ISO 15504 [6] et le CMMi [2]). Plus récemment, deux nouveaux modèles ont vu le jour : SQALE [13, 14], une méthode indépendante du langage et des outils d'analyse utilisés, qui s'appuie principalement sur les notions de dette technique et de design, et l'ISO SQuaRE [7], successeur de l'ISO 9126 toujours en cours de développement. De plus, certains domaines ont leur propre standards de-facto : HIS et ISO 26268 [8] pour l'industrie automobile ou

la DO-178 [22] pour l'aéronautique et les systèmes embarqués critiques.

Mais certaines objections à ces modèles ont vu le jour dans les dernières années :

- Les standards, et certains modèles en particulier, sont parfois considérés comme le jugement subjectif et l'opinion d'experts, ainsi que relevé par Jung et al. [10] pour l'ISO 9126.
- Ils ne sont pas adaptés à tous les contextes et tous les besoins de qualité [11].

Un autre argument employé, toujours dans le cadre de la complexité et de la diversité des projets logiciels, est qu'ils ne proposent aucune mesure, ni aucun outil pratique pour mettre en œuvre l'analyse de qualité, ce qui provoque l'incompréhension de certaines mesures – à titre d'exemple, considérons la métrique MTBF, ou Mean Time Between Failure et la myriade de définitions qui lui sont associées [12].

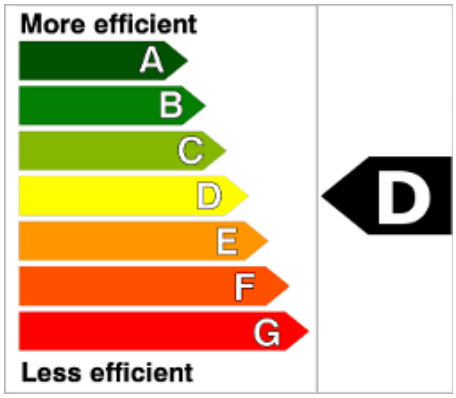
3 Métriques pour la mesure de la qualité logicielle

La littérature propose un grand nombre de métriques orientées logiciel, telles que par exemple la complexité cyclomatique, introduite par McCabe [17] pour mesurer la complexité du flot de contrôle, ou les mesures d'Halstead [4], qui permettent de qualifier le flot de données.

Un autre moyen d'évaluer la qualité d'un code est le nombre de non-conformités à une certaine référence. Par exemple, si des conventions de nommage et de codage ont été décidées, toute violation à ces règles est supposée diminuer la qualité, ou certaines de ses sous-caractéristiques, telles que l'analysabilité (pour les conventions de nommage) ou la fiabilité (pour les conventions de codage).

La notion de dette technique, édictée la première fois par Ward Cunningham en 1992 [3], et suscitant de plus en plus d'intérêt de nos jours, peut être vu comme la distance entre l'état actuel du logiciel et l'état de qualité souhaité. En ce sens, la dette technique est largement liée au nombre de non-conformités présentes dans l'application.

La tendance actuelle de la qualimétrie va vers l'analyse multidimensionnelle [12] : la qualité d'un produit ou d'un processus logiciel est la composée d'un ensemble de caractéristiques ou attributs, et donc de leurs mesures associées. Reste à savoir comment agréger et consolider ces mesures composites.



3 Les principes de SQuORE

En quelques mots, SQuORE récupère des informations issues de sources variées, agrège *intelligemment* les données et présente un rapport optimisé de l'état du logiciel ou du projet. La note principale de l'application est affichée sur une échelle de 7 niveaux, reconnaissable par tous, et donnant une indication visuelle immédiate sur les résultats de l'analyse.

Figure 1 : La notation SQuORE

4 Architecture

Le processus d'analyse SQuORE se décompose en trois étapes distinctes : la récupération des données par les data providers, le calcul des données composées par le moteur (engine), et la présentation des résultats, courbes et tableaux : le dashboard.

5 Data Providers

Ainsi que dit précédemment, il existe aujourd'hui de nombreux outils d'analyse de la qualité, chacun avec son propre domaine d'expertise et apportant une vue intéressante, mais partielle, sur la qualité. SQuORE rassemble ces informations et apporte la cohérence nécessaire à une utilisation intelligente de ces données. Tous les types de données sont acceptés : XML, CSV, appels API, jusqu'aux formats binaires, tels les anciens documents Microsoft Word.

L'analyseur SQuORE est exécuté en premier ; il est rapide, ne demande aucune compilation ou dépendance, et construit l'arbre des artefacts (quel que soit leur type : code source, tests, documentation, composants hardware, etc.) sur lesquels viendront se greffer les mesures issues d'outils tiers.

6 Consolidation des données

Une fois les mesures de base collectées, SQuORE agrège l'ensemble et calcule les mesures dérivées définies dans le modèle qualité pour chaque artefact, ainsi que montré en Figure 2.

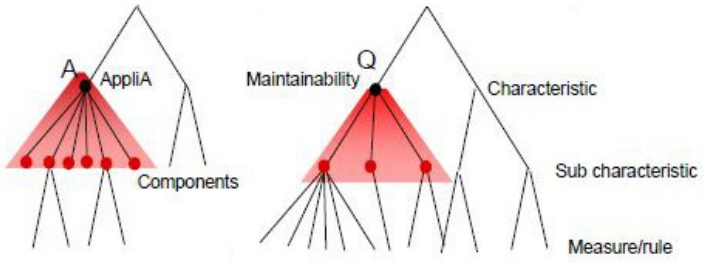


Figure 2 : Arbres des artefacts et de qualité

Modèles de qualité

Les modèles de qualité définissent la manière dont sont agrégées les métriques, des feuilles (fonctions, tests, etc.) jusqu'à la racine du projet : les mesures composites sont associées à un attribut de la qualité, lui-même affecté à chaque artefact de l'arbre.

Il n'existe pas de modèle qualité parfait et universel : chacun doit adapter la méthode d'analyse, en prenant en considération les besoins spécifiques et les objectifs de développement. Dans la plupart des cas l'élaboration du modèle de qualité se fait à partir de l'un des modèles existants, qui est adapté, pondéré différemment en fonction du contexte. Pour des situations plus complexes SQuORE propose tous les moyens nécessaires à la mise en œuvre d'un nouveau modèle, des calculs les plus simples aux requêtes les plus alambiquées.

Métriques, échelles, indicateurs

Les mesures en elles-mêmes donnent la valeur d'une caractéristique, sans jugement qualitatif. Les indicateurs fournissent cette information en comparant la valeur à une échelle de niveaux adaptée, ce qui permet de configurer les seuils et poids pour chaque niveau de jugement. Par exemple, le célèbre nombre cyclomatique [17] au niveau des fonctions pourrait être réparti sur une échelle de 4 niveaux :

- De 1 à 7, la fonction est notée A (très bon) et son poids de dette technique est de 0.
- De 8 à 15, la fonction est notée B (ok), et son poids est de 2.
- De 16 à 25, la fonction est notée C (mauvais) et son poids est de 4,
- Au-delà de 25, la fonction est notée D (très mauvais) et son poids est de 16 – car un refactoring est nécessaire.

Appliqué à cette échelle, l'indicateur du nombre cyclomatique donne immédiatement l'état de complexité de la fonction, quels que soient les triggers utilisés pour le contexte local.

Action Items

Il arrive souvent qu'une caractéristique que nous souhaitons évaluer dépende de plusieurs mesures de base. Par exemple, si une fonction est plutôt longue, a une complexité au-dessus de la moyenne, de nombreuses violations et un faible taux de commentaire, il est certainement intéressant de s'y pencher, alors même que chacun de ces éléments, pris individuellement, ne sont pas décisifs. Les action items permettent de mettre en place des triggers qui se déclenchent sur une variété de critères entièrement paramétrables. Une description du problème et des pistes pour sa résolution sont affichées, avec l'emplacement précis des différents critères qui ont déclenché le trigger. Les triggers peuvent être positionnés sur tout type d'artefact, ce qui fait d'eux un moteur puissant, par exemple pour formaliser et automatiser des heuristiques issues de l'expérience et du savoir-faire des équipes.

7 De la qualité produit à la gestion de projets

L'évaluation de la qualité d'un produit ou d'un processus est la première étape pour mieux contrôler ses développements (si l'on ne sait pas où l'on est, une carte est de peu d'utilité). L'étape suivante est de surveiller l'évolution du projet au fil du temps et des versions, et d'en extraire les éléments nécessaires à la prise de décision – processus décrit par Hassan et al. [5]. SQuORE propose de nombreux mécanismes d'exploration pour aider à cette démarche :

- Les graphiques d'évolution montrent l'historique des mesures, et attributs de qualité au fil des versions, et la tendance.
- Les nombreuses options de filtre et de tri permettent de localiser rapidement certains éléments (e.g. éléments dégradés, nouvelles fonctions, fichiers les plus mal notés).
- L'arbre de qualité montre quelles caractéristiques de la qualité se sont dégradées : maintenabilité du code, dérive du planning, tests échoués ?
- Les actions items permettent d'identifier des schémas d'évolution complexes, basés sur des caractéristiques actuelles ou sur certaines tendances (par exemple, dégradation de la testabilité, échecs nombreux sur les tests existants, diminution du taux de couverture de test).

4 Cas d'utilisation

8 Retours immédiats des équipes

Lors de nos présentations de l'outil SQuORE, certaines situations et réactions reviennent régulièrement:

- Les personnes sont capables d'identifier rapidement certains problèmes sérieux, sans être submergées par le flot d'informations. Souvent, ces problèmes avaient déjà été détectés par d'autres outils, mais l'information n'avait pas été identifiée ou suffisamment mise en valeur.
- Le sentiment général à propos de certaines applications connues des équipes est confirmé par des preuves factuelles et démontrables. Cela démontre la représentativité de la mesure et rend visibles et argumentables les pratiques de développement.
- Les développeurs sont concernés par la note de leur code et sa représentation simple et concise. L'utilisation de l'échelle d'énergie, reconnue de tous, renforce le caractère de standard (local) du modèle de qualité.

9 Modèle de test unitaire

Les délais de développement étant toujours trop courts pour le marché, il arrive parfois de devoir prioriser l'effort de test pour maximiser la rentabilité et le nombre de problèmes trouvés avant livraison, sachant que l'exécution complète des tests est de toute façon impossible dans les temps impartis.

L'un de nos clients avait seulement quelques jours de test pour qualifier une version logicielle. Jusque-là, le processus était essentiellement humain : les fichiers à tester étaient sélectionnés par rapport à leur historique de test, leurs récents changements, la complexité des fonctions et le nombre de violations détectées. A force d'expérience, certaines heuristiques avaient été développées pour aider à l'identification des fichiers prioritaires, avec attributs et seuils de tolérance.

Nous avons donc construit un modèle de qualité dédié, prenant en entrées les sorties de deux outils utilisés par l'équipe : QAC et Rational Test Real Time, plus l'analyseur SQuORE. Quatre mesures ont été définies : nombre de non-conformités QAC, RTRT, SQuORE, et le nombre cyclomatique de chaque fonction. La notation du modèle est construite sur une base exponentielle. Par exemple, si un fichier a une seule de ses mesures en défaut, sa note de dette technique sera de 2. Si deux, trois, ou quatre de ces mesures sont en défaut, le fichier recevra une note qui sera respectivement de 8, 16, 32. Cette structure permet d'identifier rapidement les pires fichiers.

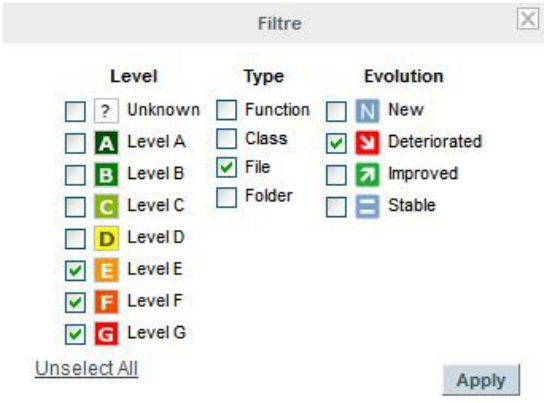


Figure 3 : Filtres d'artefacts

La notation des répertoires est basée sur le ratio de « mauvais » fichiers récursivement trouvés sous ledit répertoire, afin de mettre en relief les pires composants du logiciel simplement en triant l'arbre des artefacts par rapport à la notation de chaque répertoire.

Des action items spécifiques ont été mis en place pour les fichiers présentant un risque réel pour la validation, parce qu'ils ont une complexité cyclomatique réellement importante, ont un nombre de violations inquiétant ou une très mauvaise notation. Ces triggers permettent notamment de trouver de mauvais éléments cachés dans des hiérarchies de « bons » fichiers ou composants.

Nous avons ainsi pu :

- Mettre en place un modèle de prédiction d'efforts de test formalisé, et reconnu par les acteurs comme étant efficace. Le temps gagné à ce moment du processus est capital pour la mise sur le marché, et permet d'exécuter davantage de tests, puisque la détection est plus rapide.
- Reproduire et automatiser une heuristique « au ressenti » qui était auparavant affaire d'expérience et de savoir-faire. En le formalisant et en l'automatisant, les équipes ont pu davantage se concentrer sur l'effort de test.

10 Suivi de projet

Une autre expérience industrielle concernait la mise en place d'un suivi de projet plus complet, avec progression du backlog, avancées des tests et surveillance de la qualité logicielle.

L'analyseur SQuORE a été utilisé pour la qualité produit (code Java), et des data providers ont été mis en place pour récupérer les informations des outils de gestion des changements, test, et planning.

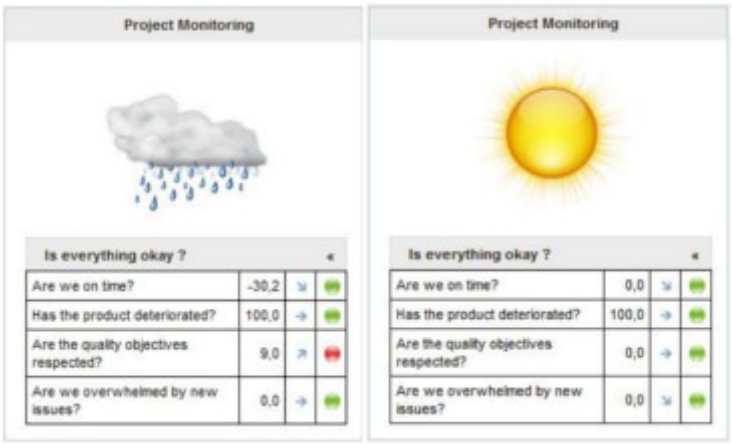


Figure 4: Scorecards de suivi de projet

4 axes de qualité ont été définis par rapport à ces entrées, résumés par les questions suivantes (cf. l'approche de V. Basili : Goal-Question-Metric [26]).

- « Est-ce que les objectifs de qualité sont respectés ? », basé sur la norme ISO9126.
- « Sommes-nous dans les temps? », utilisant les notions agiles de tâches.
- « Sommes-nous surchargés par les nouvelles requêtes ? », avec le nombre de bugs reportés en attente de traitement.
- « Le produit s'est-il détérioré ? », qui reprend les résultats de test (couverture et exécution).

Ces axes sont ramenés sur un unique indicateur, montrant l'avancement global du projet. Dans ce cas, la pire note trouvée parmi les sous-caractéristiques est utilisée en note globale, car il est considéré comme nominal que chacun de ces indicateurs soit positif.

Nous avons développé des action items afin d'identifier :

- Les parties du logiciel (composants ou fichiers) qui ont une mauvaise note de

maintenabilité, sont peu couverts par les tests, et sont concernées par beaucoup de reports de bugs.

- Les dérives de planning, lorsque le nombre de tâches ouvertes est trop important et de nombreux bugs sont ouverts.

L'automatisation des opérations de récupération des données, puis d'exécution de l'analyse et de publication des résultats est un élément clef de la réussite d'un projet de qualimétrie [27] ; Jenkins a donc été mis en place comme serveur d'intégration continue pour exécuter SQuORE régulièrement (exécutions quotidienne et hebdomadaire).

Des dashboards spécifiques ont été mis en place pour suivre aisément les critères importants de la gestion de projet : maintenabilité, requêtes ouvertes, tâches en attente, tests exécutés avec succès... Les informations de planning ont été représentées par les graphiques agiles de burn-up et burn-down.

Figure 5 : Arbre de qualité suivi de projet

Nous avons ainsi pu :

- Mettre en place un suivi quotidien et consolidé de l'état d'avancement des développements, avec surveillance des régressions et de la qualité.
- Reprendre le contrôle du projet et redonner confiance aux acteurs, managers et développeurs, en proposant une vision claire et concise, partagée et cohérente des développements.

5 References

[1] B. W. Boehm, J. R. Brown, and M. Lipow. Quantitative evaluation of software quality. In *Proceedings of the 2nd international conference on Software engineering* , pages 592-605, San Francisco, California, United States, 1976. IEEE Computer Society Press.

[2] CMMI Product Team. CMMI for Development, Version 1.3. Technical report, Carnegie Mellon University, 2010.

[3] Martin Fowler. Martin Fowler on Technical Debt, 2004.

[4] Maurice H. Halstead. *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc., New York, NY, USA, 1977.

[5] Ahmed E Hassan and Tao Xie. Software Intelligence: The Future of Mining Software Engineering Data. In *Proc. FSE/SDP Workshop on the Future of Software Engineering Research (FoSER 2010)*, pages 161-166. ACM, 2010.

[6] ISO IEC. ISO/IEC 15504-1 – Information Technology – Process assessment. *Software Process: Improvement and Practice*, 2(1):35{50, 2004.

[7] ISO IEC. ISO/IEC 25000 – Software Engineering – software product quality requirements and evaluation (SQuaRE) – guide to SQuaRE. *Systems Engineering*, page 41, 2005.

[8] ISO. ISO/DIS 26262-1 - Road vehicles Functional safety Part 1 Glossary. Technical report, International Organization for Standardization / Technical Committee 22 (ISO/TC 22), 2009.

[9] ISO/IEC. ISO/IEC 9126 – *Software Engineering – Product quality*. 2001.

[10] Ho-Won Jung, Seung-Gweon Kim, and Chang-Shin Chung. Measuring software product quality: A survey of ISO/IEC 9126. *IEEE Software*, 21:88-92, 2004.

[11] Stephen H. Kan. *Metrics and Models in Software Quality Engineering*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002.

[12] C Kaner andWalter P. Bond. Software engineering metrics: What do they measure and how do we know? *In 10Th International Software Metrics Symposium, METRICS 2004*, pages 1-12, 2004.

[13] Jean-louis Letouzey. The SQALE method for evaluating Technical Debt. In 2012 Third International Workshop on Managing Technical Debt, pages 31-36, 2012.

[14] Jean-louis Letouzey and Thierry Coq. The SQALE Analysis Model An analysis model compliant with the representation condition for assessing the Quality of Software Source Code. In *2010 Second International Conference on Advances in System Testing and Validation Lifecycle (VALID)*, pages 43-48, 2010.

[15] Nancy Leveson and Clark S. Turner. An Investigation of the Therac-25 Accidents. *IEEE Computer*, 26(7):18{41, 1993.

[16] J.L. Lions. ARIANE 5 Flight 501 failure. Technical report, 1996.

[17] TJ McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, (4):308-320, 1976.

[18] J.A. McCall. *Factors in Software Quality: Preliminary Handbook on Software Quality for an Acquisition Manager*. *Information Systems Programs, General Electric Company*, 1977.

[19] National Aeronautics and Space Administration. Mars Climate Orbiter Mishap Investigation Report. Technical report, National Aeronautics and Space Administration, Washington, DC, 2000.

[20] Peter G. Neumann. Cause of AT&T network failure. *The Risks Digest*, 9(62), 1990.

[21] National Institute of Standards and Technology (NIST), Department of Commerce. Software errors cost U.S. economy $59.5 billion annually, 2002.

[22] RTCA. DO-178B: Software Considerations in Airborne Systems and Equipment Certification. Technical report, Radio Technical Commission for Aeronautics (RTCA), 1982.

[23] Mary Shaw. Prospects for an engineering discipline of software. *IEEE Software*, (November):15-24, 1990.

[24] The Standish Group International Inc. The Standish Group International Inc. Chaos Technical report. Technical report, 2004.

[25] United States General Accounting Office. Patriot Missile Defense: Software Problem Led to System Failure at Dhahran, Saudi Arabia. Technical report, United States General Accounting Office, 1992.

[26] Basili, V. R., Caldiera, G., & Rombach, H. D. (1994). The goal question metric approach. Encyclopedia of Software Engineering. Wiley.

[27] Basili, V., & Weiss, D. (1984). A methodology for collecting valid software engineering data. IEEE Trans. Software Eng., 10(6), 728-738.

De l'ombre à la lumière : plus de visibilité sur l'Eclipse

Résumé. L'extraction de connaissances à partir de données issues du génie logiciel est un domaine qui s'est beaucoup développé ces dix dernières années, avec notamment la fouille de référentiels logiciels (Mining Software Repositories) et l'application de méthodes statistiques (partitionnement, détection d'outliers) à des thématiques du processus de développement logiciel.

La fondation Eclipse, l'une des plus importantes forges du monde libre, a mis en place récemment une infrastructure de centralisation de ces données et a initié le groupe de travail Polarsys pour le suivi de qualité de projets impliqués dans le développement de systèmes embarqués critiques.

Cet article présente la démarche de fouille de données mise en œuvre dans le cadre de Polarsys, de la définition des exigences à la proposition d'un modèle de qualité dédié et à son implémentation sur un prototype. Les principaux problèmes rencontrés et les leçons tirées sont également passés en revue.

1 Introduction

Les méthodes statistiques ont été introduites il y a plusieurs décennies déjà dans le domaine du suivi et du contrôle de processus de production, avec par exemple les méthodes SPC (Statistical Process Control) ou Six Sigma. Bien que l'industrie logicielle en ait considérablement bénéficié depuis, ce n'est que récemment que les problématiques particulières du génie logiciel, c'est-à-dire l'art de construire de bons logiciels, ont été spécifiquement adressées. L'arrivée de nouvelles méthodes issues de la fouille de données (e.g. partitionnement, détection d'outliers) ouvre cependant de nouveaux horizons pour la compréhension et l'amélioration du logiciel et des pratiques de développement.

De nouveaux types de données sont également utilisés : métadonnées des outils du processus (gestion de configuration, gestion des changements), analyse des moyens de communication, statistiques de publications, etc. Il est nécessaire de définir et d'étudier ces nouveaux types d'artéfacts pour pouvoir les collecter de manière fiable, puis les exploiter et leur définir

un lien pertinent avec les attributs de qualité. Les données issues du processus de développement logiciel ont par ailleurs des propriétés particulières : leur distribution n'est pas normale, il y a de fortes corrélations entre certaines d'entre elles, et leur sémantique doit être pleinement comprise pour les rendre pleinement utilisables. En ce sens de nombreuses questions sur le processus de mesure de données logicielles, notamment relevées par Basili et Weiss (1984), Fenton (1994) et Kaner et Bond (2004), restent d'actualité pour ce domaine émergeant. Les approches proposées dans ces dix dernières années (Zhong et al., 2004; Herbold et al., 2010; Yoon et al., 2007) n'ont pas su tenir compte de ces caractéristiques et manquent encore d'applications concrètes – ceci étant une caractéristique du manque de maturité de la discipline, ainsi que le soulignait Shaw (1990).

Nous proposons dans cet article quelques pistes pour répondre à ces différentes problématiques, et démontrons son application dans le cadre de la fondation Eclipse. La section 2 pose les bases de notre réflexion et présente une *topologie nominale* pour les projets de développement logiciel. La section 3 propose une approche adaptée à l'application de méthodes de fouilles de données au processus de développement logiciel, et la section 4 décrit sa mise en pratique dans le cadre du groupe de travail Eclipse Polarsys. Quelques exemples de résultats sont montrés en section 5. Finalement, nous présentons en section 6 les prochaines évolutions de la mesure de qualité pour Polarsys et récapitulons le travail et ses apports dans la section 7.

2 Prolégomènes : comprendre le problème

2.1 Nature des données

Lorsque l'on parle de fouille de données logicielles, la source la plus intuitive est sans doute le code source, puisque c'est l'objectif principal de tout développement et une ressource commune à tout projet logiciel. Cependant, en élargissant notre champ de vision il existe de nombreuses autres mesures qui apportent des informations importantes pour le projet, notamment relatives au processus de développement lui-même.

Chaque référentiel d'information utilisé par le projet possède des informations exploitables. Il importe de lister les référentiels disponibles, puis d'identifier pour chacun d'eux les artéfacts et mesures disponibles, et de leur donner un sens. Certains types d'artéfacts sont plus pertinents que d'autres pour certains usage ; par exemple l'activité d'un projet est visible sur le nombre de révisions et/ou la diversité des développeurs, alors que la réactivité du support est davantage visible sur le système de gestion des tickets.

Notamment, le croisement des données entre artéfacts s'avère très utiles mais peut être difficile à établir – par exemple les liens entre modifications de code et bugs (Bachmann et al., 2010). Les conventions de nommage sont un moyen courant de tracer les informations entre les référentiels, mais elles sont locales à chaque projet et ne sont pas nécessairement respectées ou vérifiées.

2.2 Les sources de données logicielles

Bien que l'environnement utilisé varie en fonction de la maturité et du contexte du projet, on peut définir pour base commune un ensemble d'outils et de référentiels communs à tout projet de développement logiciel.

Le code source est le moyen le plus utilisé pour l'analyse de projets logiciels. Du point de vue de l'analyse statique (Louridas, 2006; Spinellis, 2006), les informations que l'on peut récupérer d'un code source sont :

- des métriques, correspondant à la mesure de caractéristiques définies du logiciel : e.g. sa taille, la complexité de son flot de contrôle, ou le nombre maximal d'imbrications.
- des violations, correspondant au non-respect de conventions de codage ou de nommage. Par exemple, ne pas dépasser une valeur de métrique, l'interdiction des goto inverses ou l'obligation de clauses default dans un switch. Ces informations sont fournies par des analyseurs tels que Checkstyle [1], PMD [2] ou SQuORE [3].

La gestion de configuration contient l'ensemble des modifications faites sur l'arborescence du projet, associées à un ensemble de méta-informations relatives à la date, l'auteur et la version (branche) du produit visée par la révision. Le positionnement dans l'arbre des versions est capital, car les résultats ne seront pas les mêmes pour une version en cours de développement (active avec beaucoup de nouvelles fonctionnalités, typiquement développée sur le tronc) que pour une version en maintenance (moins active, avec une majorité de corrections de bugs, typiquement développée sur une branche). Dans nos analyses, c'est le tronc que nous référençons afin d'analyser et améliorer la *prochaine* version du produit.

La gestion des tickets recense l'ensemble des demandes de changement faites sur le projet. Ce peuvent être des problèmes (bugs), de nouvelles fonctionnalités ou de simples questions. Certains projets utilisent également les demandes de changement pour tracer les exigences liées à un produit.

Les listes de diffusion sont les principaux moyens de communcation utilisés au sein de projets logiciels. Il existe en général au moins deux listes, une dédiée au développement et l'autre aux questions utilisateur. Leurs archives sont en général disponibles, que ce soit au format mbox ou par une interface web (e.g. mhonarc, mod_mbox, gmane ou markmail).

2.3 La mesure de caractéristiques logicielles

L'art de la mesure est bien plus mature que celui du développement logiciel. Par exemple la notion que l'on a de la taille d'une personne est commune à tout le monde, et tout un chacun sait ce que veut dire que d'être plus grand. Mais beaucoup d'évidences s'estompent cependant lorsque l'on parle de logiciel, ainsi que l'ont fait remarquer Fenton (1994) et Kaner et Bond (2004). A titre d'exemple, à fonctionnalité équivalente la taille d'un programme en Java n'est pas comparable à celle d'un code C ou en Perl, à cause du niveau d'abstraction du langage, des conventions de nommage et des structures de données disponibles.

Cela est d'autant plus vrai dans le cas de caractéristiques abstraites telles que les notions de qualité, de fiabilité ou d'utilisabilité. De la conformité aux exigences à la satisfaction utilisateurs beaucoup de définitions, puis de modèles de qualité ont été proposés. Mais chacune de ces notions est différente, en fonction de l'expérience et de l'expertise de chacun, du domaine d'application et des contraintes de développement. Pour cette raison il importe en préambule à toute activité de fouille de données logicielles de définir les attentes des utilisateurs et de fournir à tous un socle commun pour la compréhension et l'interprétation des objectifs du pro-

1. http ://checkstyle.sourceforge.net
2. http ://pmd.sourceforge.net
3. http ://www.squoring.com

gramme de fouille – dans notre cas ce sont par exemple les développeurs et utilisateurs finaux du produit logiciel.

Il est préférable pour la définition de la notion de qualité de s'appuyer sur des modèles ou standards éprouvés et (re)connus, car les concepts et définitions ont déjà fait l'objet de débats et d'un consensus commun sur lesquels s'appuyer. En outre, la littérature et les retours d'expérience accessibles facilitent la communication sur le programme et donnent de solides fondations à l'approche choisie – par exemple pour étayer la sélection des métriques en regard des attributs de qualité. Du point de vue de **la qualité produit**, la référence *de facto* semble être l'ISO 9126 (ISO, 2005). La relève, en cours d'élaboration par l'organisme de standardisation ISO, est la série 250xx SQuaRE (Organization, 2005). La **maturité du processus de développement** est adressée par des initiatives largement reconnues telle que le CMMi ou l'ISO 15504. Enfin, des modèles de qualité dédiés ont été proposés pour certains domaines tels que ECSS pour l'espace, HIS pour l'automobile, ou SQO-OSS pour l'évaluation de logiciels libres.

L'intégrité des métriques souffre de cette diversité et de ce manque de compréhension. Afin de clarifier la démarche de mesure le suivi de programmes établis tels que l'approche Goal-Question-Metric proposée par Basili et al. (1994) et reprise par Westfall et Road (2005) permet une approche plus rigoureuse, qui préserve l'efficacité de l'analyse et le sens de ses résultats.

3 Présentation de la méthode d'extraction

La fouille de données logicielles est similaire en plusieurs points à la conduite de programmes de mesures. L'expérience glanée au fil des années sur ce dernier domaine (voir notamment Iversen et Mathiassen (2000), Gopal et al. (2002) et Menzies et al. (2011)) a permis l'établissement de bonnes pratiques qui aident à la définition, à l'implémentation et à l'utilisation des résultats d'analyse.

En nous inspirant de cette expérience, nous avons suivi le déroulement décrit ci-après pour nous assurer de l'utilisabilité et de l'intégrité sémantique des résultats.

Déclaration d'intention La déclaration d'intention du travail de fouille et du modèle de qualité est primordiale car elle donne un cadre méthodologique et sémantique pour la compréhension des utilisateurs et l'interprétation des résultats. Par exemple, les mesures accessibles et les résultats attendus ne seront pas les mêmes pour un audit et pour un programme d'amélioration continue de la qualité. L'intention doit être simple et tenir en quelques phrases.

Décomposition des attributs de qualité L'intention du programme de fouille est ensuite formalisée et décomposée en attributs et sous-attributs de qualité. Cette étape forme le lien entre une déclaration informelle d'intention et les mesures concrètes qui vont permettre de la mesurer et de l'améliorer ; elle doit faire l'objet d'un consensus général et être validée par les acteurs du programme. Le livrable attendu à l'issue de cette phase est un modèle de qualité qui décrit et décompose les besoins identifiés, tels que montrés en exemple en figure 1.

Définition des métriques accessibles L'identification et la collecte des données de mesure est une étape fragile du processus, et une pierre d'achoppement classique (Menzies et al., 2011). Nous devons d'abord établir la liste des référentiels exploitables (voir section 2.2),

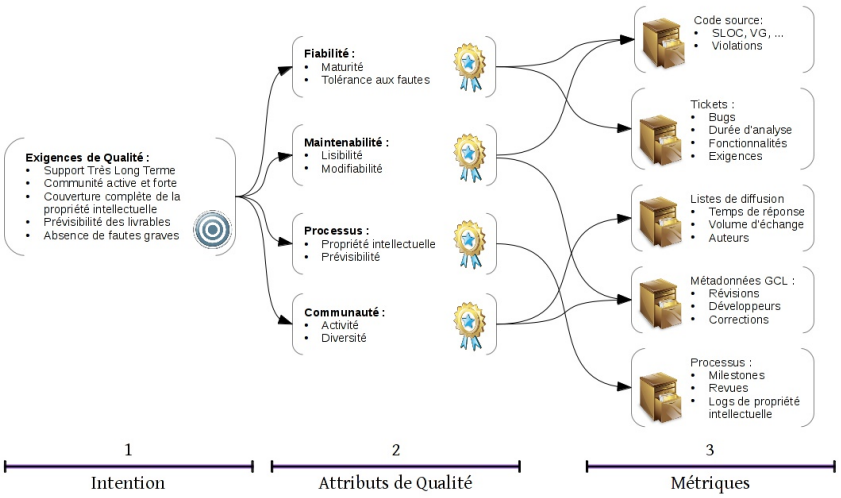


FIG. 1 – *Processus de définition du modèle de qualité et du choix des métriques.*

puis identifier les mesures accessibles pour chacun d'entre eux, et les relier aux caractéristiques de qualité. Il importe que la collecte soit **fiable** (i.e. l'information recherchée est systématiquement présente et valide), **automatisable** afin de réduire le facteur d'erreur humaine, et **compréhensible** pour garder la confiance des acteurs dans le processus.

Implémentation
Le processus de collecte et d'analyse doit être intégralement **automatisé**, de la collecte des données à la publication des résultats, et **exécuté de manière régulière** – par exemple au moyen d'un serveur d'intégration continue tel que Jenkins (Smart, 2011). Son implémentation doit être **transparente** pour que les utilisateurs puissent se l'approprier et éventuellement soumettre des problèmes ou améliorations [4].

Présentation
La manière dont l'information est présentée est capitale et peut mettre en péril l'ensemble de la démarche. Dans certains cas une liste concise d'artéfacts est suffisante, alors que dans d'autres cas un graphique bien choisi sera plus adapté et délivrera en quelques secondes l'essentiel du message. Pour chacun des intérêts et objectifs identifiés il importe de choisir un mode de visualisation qui permette sa compréhension et son exploitation aisément. Par exemple si l'on veut présenter une liste d'artéfacts à retravailler, celle-ci doit être suffisamment courte pour ne pas décourager les développeurs et sa signification suffisamment claire pour savoir immédiatement ce qui doit être amélioré.

Reproductibilité de l'analyse Les données manipulées sont nombreuses (les métriques pouvant être récupérées sur l'ensemble du cycle de vie d'un projet logiciel se comptent par

4. Non sans une certaine touche d'ironie, les logiciels d'analyse de la qualité ou de détection de fautes ne sont eux-mêmes pas exempts de problèmes. Lire à ce propos Hatton (2007).

dizaines une fois filtrées) et complexes (de par leur sémantique et leurs interactions). Afin de garantir la cohérence des informations extraites il importe de préserver la reproductibilité des analyses exécutées. L'utilisation de techniques **d'analyse littéraire**, introduites par D. Knuth pour le développement logiciel et reprises par F. Leisch et Xie (2013) [5] pour l'analyse de données, aide de beaucoup à la compréhension des résultats d'analyse. L'information peut être fournie sous forme de tableaux ou de graphes générés dynamiquement, mais également sous forme de texte explicatifs. L'analyse de données littéraire garantit ainsi que l'ensemble des projets utilise le même processus d'analyse, ce qui les rend comparables, et donne un contexte sémantique aux résultats.

4 Mise en pratique avec Eclipse

4.1 Présentation du contexte

La fondation Eclipse est l'une des forges majeures du monde libre et un exemple remarqué de succès du modèle open source. Elle abrite aujourd'hui plusieurs centaines de projets, leur fournissant une forge et un éventail de services liés au développement : gestion de la propriété intellectuelle, conseil pour la gestion des projets, et développement de l'écosystème.

L'une des caractéristiques fortes de la fondation Eclipse est son accointance avec le monde industriel. Des groupes de travail (IWG, Industry Working Group) sont constitués sur des sujets précis, avec un calendrier et des livrables définis, pour proposer des solutions à des problématiques concrètes. Ainsi le groupe de travail Polarsys [6] a pour tâche de promouvoir l'utilisation d'Eclipse dans le monde des systèmes embarqués avec un environnement de développement dédié et un support à très long terme (jusqu'à 75 ans) sur les outils livrés.

Par ailleurs la fondation a lancé récemment plusieurs initiatives pour fédérer les outils du processus de développement et rendre leurs données publiquement accessibles, notamment le projet *PMI* [7] (Project Management Infrastructure) qui recense les informations liées au processus de développement, et *Dash* [8] propose des outils d'analyse des informations.

4.2 La qualité au sein d'Eclipse

4.2.1 Déclaration d'intention

Les objectifs de ce programme de fouille ont été identifiés comme suit :

- **Evaluer la maturité du projet.** Peut-on raisonnablement s'appuyer dessus lorsque l'on construit une application ? Le projet suit-il les recommandations de la fondation ?
- **Aider les équipes à construire un meilleur logiciel**, en leur proposant une vision claire sur leurs développements et des actions simples pour améliorer le produit ou le processus de développement.
- **Fonder une compréhension commune de la qualité** au sein de la fondation, et poser les bases d'une discussion saine sur l'évaluation et l'amélioration des projets Eclipse.

5. Nous avons utilisé Knitr pour nos analyses : http ://yihui.name/knitr/.
6. https ://polarsys.org/wiki//index.php/Main_Page.
7. http ://wiki.eclipse.org/Project_Management_Infrastructure.
8. http ://wiki.eclipse.org/Dash_Project.

4.2.2 Exigences de qualité pour Eclipse

Il n'existe pas de contrainte forte de qualité produit pour les projets Eclipse. Il est recommandé aux projets de mettre en place des conventions de développement propres et de travailler sur la qualité du projet, mais cela n'est ni obligatoire ni vérifié. Néanmoins en navigant dans les ressources en ligne de la fondation plusieurs mots-clefs reviennent régulièrement, notamment en ce qui concerne la **lisibilité** et la **modifiabilité** du code [9]. Le processus de développement Eclipse prévoit plusieurs phases pour un projet : proposition, incubation, mature, archivé. Chaque phase est associée à un certain nombre de contraintes : mise en place de **revues et de milestones**, vérification des informations de **propriété intellectuelle**, etc. La plupart de ces règles sont obligatoires. Enfin, Eclipse définit trois communautés fondamentales pour un projet, que celui-ci doit développer et maintenir : développeurs, utilisateurs, et adopteurs. Dans la perspective de notre évaluation, nous n'en retriendrons que deux : développeurs et utilisateurs. Ces communautés doivent être **réactives** (le projet ne peut être inactif très longtemps) et **diversifiées** (le projet ne peut dépendre que d'un individu ou d'une société).

4.2.3 Exigences de qualité pour Polarsys

Polarsys délivre des assemblages constitués de composants Eclipse pour l'industrie de l'embarqué. Cela implique des contraintes de qualité particulières, particulièrement en ce qui concerne la capacité d'industrialisation de la solution et sa pérennité.

- **Fiabilité** : une faute dans un seul des composants peut mettre en péril l'intégralité de la pile logicielle. Plus précisément, la notion de fiabilité exprimée par le groupe de travail est à la croisée de la maturité et de la tolérance au faute selon le modèle ISO 9126.
- **Maintenabilité** : le support doit être assuré jusqu'à 75 ans pour certain systèmes (e.g. espace et aéronautique). Outre la maintenabilité du produit lui-même, cela implique que les communautés entourant le projet soient actives et diversifiées.
- **Prédictibilité** : la solution sera déployée sur des milliers de postes dans le monde et aura des conséquences importantes sur la productivité des équipes. Il est vital de pouvoir compter sur la date et la qualité des livraisons.

Globalement, les contraintes de qualité de Polarsys sont similaires à celles de la fondation Eclipse, mais sont renforcées pour le contexte industriel spécifique de l'assemblage. La participation financière des industriels impliqués dans le groupe permet d'assurer une force de travail conséquente pour assumer ces exigences et d'imposer certaines contraintes aux projets sélectionnés.

4.2.4 Proposition d'un modèle de qualité

A partir des exigences identifiées lors de l'analyse préalable, trois axes de qualité ont été identifiés : la qualité du produit, du processus et de la communauté. Ceux-ci sont décomposés en sous-caractéristiques selon la hiérarchie montrée en figure 2

Ces sous-caractéristiques, qui sont la granularité la plus fine d'attributs de qualité, sont ensuite liés aux métriques selon une méthode décidée en concertation avec les acteurs, et publiée pour les utilisateurs (développeurs et leaders).

9. Les problématiques de maintenabilité sont évidemment accentuées pour tous les projets libres, destinés à être maintenus par tout un chacun.

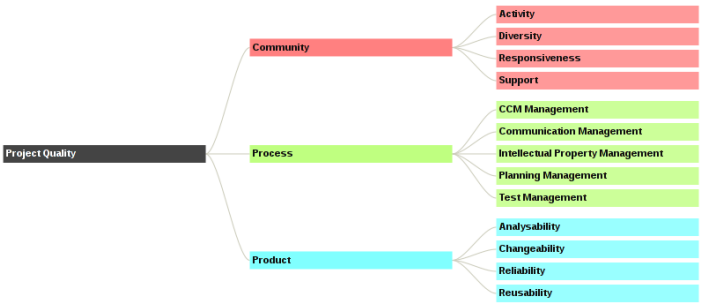


FIG. 2 – *Modèle de qualité proposé pour Polarsys.*

4.3 Sélection des métriques

Nous avons dressé la liste des référentiels disponibles et des informations qu'ils renfermaient. Le principe de la forge [10] aide beaucoup en limitant le nombre de connecteurs différents, cependant chaque projet garde un certain nombre de *coutumes locales*. Pour chaque référentiel nous avons ensuite défini des indicateurs pertinents pour les attributes de qualité précédemment identifiés.

Les métadonnées de gestion de configuration sont extraites à partir des logs du serveur. Deux connecteurs ont été développés, l'un pour Subversion et l'autre pour Git. Les métriques définies aux niveaux application et fichier sont : le nombre de révisions, de révisions de correction et de développeurs. Au niveau application le nombre de fichiers associés aux révisions est également utilisé. Toutes ces métriques sont récupérées sur quatre périodes de temps pour mieux appréhender l'évolution récente des métriques : depuis l'origine, depuis une semaine, un mois et trois mois.

Les moyens de communication fournis par Eclipse sont les forums en ligne (souvent utilisés pour les listes utilisateur), les listes de diffusion et un serveur NNTP (souvent utilisés pour les listes de développement). Plusieurs scripts ont été écrits pour transformer ces différents accès en un format unique, le format mbox. A partir de ce socle commun les métriques suivantes sont extraites au niveau application : le nombre de mails échangés, de fils de discussion, d'auteurs distincts, de réponses et le temps médian de première réponse. Les données sont récupérées sur des laps de temps de une semaine, un mois et trois mois.

D'autres référentiels ont été investigués, mais seront inclus dans une prochaine itération du modèle de qualité.

Les informations de processus sont issues de l'initiative Eclipse PMI, qui est en cours d'implémentation. La liste des informations [11] référencées est donc mouvante, mais aujourd'hui les informations récupérées sont : le nombre de milestones et de revues définies pour le projet, le nombre de fonctionnalités de haut niveau décidées pour le périmètre de la release, et le nombre d'exigences bas-niveau associées.

10. Une forge offre un environnement complet et intégré de travail pour le développement d'un projet, et limite donc le nombre d'outils disponibles.
11. Voir la liste des méta-données : http ://wiki.eclipse.org/Project_Management_Infrastructure/Project_Metadata.

Les logs de propriété intellectuelle sont enregistrés dans un référentiel dédié, et rendus exploitable au travers d'une API publique. Les logs sont utilisés pour contrôler la couverture des participations afin qu'aucune ne soit laissée sans déclaration de propriété intellectuelle.

4.4 Liens entre attributs de qualité et métriques

Les métriques qui permettent la mesure des attributs de qualité sont agréées par le groupe de travail. Une proposition a été faite, ses principes exposés et un débat constructif a permis de faire évoluer, puis de valider le modèle. Les sous-attributs de la qualité produit sont basés sur un schéma identique : le nombre de non-conformités aux règles de codage impactant la caractéristique, l'adhérence aux conventions (pratiques non acquises, quelque soit le nombre de leurs violations), et un ou plusieurs index issus de bonnes pratiques reconnues – ce dernier critère requérant particulièrement l'assentiment du groupe pour être valide [12]. La composante communautaire du projet est analysée selon les axes suivants : **activité** du projet (le nombre de contributions récentes), sa **diversité** (le nombre d'acteurs distincts), la **réactivité** (le temps de réponse aux requêtes) et le **support** (la capacité de réponse aux requêtes).

5 Résultats

Le prototype implémenté dans le cadre du groupe de travail Eclipse s'est appuyé sur l'outil SQuORE Baldassari (2012) et a été appliqué à un sous-ensemble de projets pilotes. Les résultats des analyses hebdomadaires ont été rendus disponibles en même temps que la publication de rapports Knitr générés automatiquement. Les résultats de cette première itération du programme sont probants.

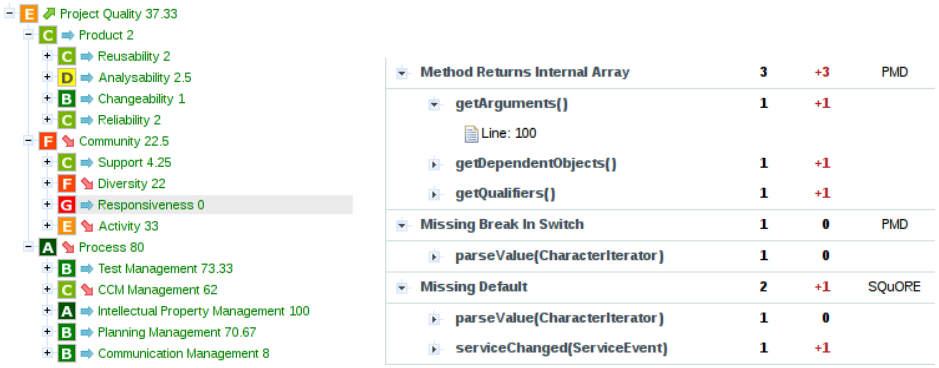


FIG. 3 – *Exemples de résultats : arbre de qualité et violations.*

Du point de vue de **l'aide à la prise de décision** premièrement, les axes de qualité choisis permettent d'évaluer la maturité du projet d'un coup d'œil, selon les critères propres à l'organisation. Par exemple le projet analysé en figure 3 montre une bonne qualité produit, mais

12. D'après Fenton (1994) la théorie de la mesure considère comme valide une mesure qui obtient l'agrément minimum de la majorité des experts dans la communauté.

une faible activité de communication et un petit nombre de participants, car l'essentiel des contributions est fait par une unique société. Cette caractéristique peu visible *a priori* en fait un risque pour la pérennité du projet, rapidement identifiable sur l'arbre de qualité.

Du côté des équipes de développement, **l'aide à l'amélioration de la qualité** se fait au moyen de listes de violations et de points d'action, qui permettent d'identifier et d'améliorer les pratiques non acquises et fournissent des recommandations pragmatiques pour l'amélioration du code ou du processus. En outre, les non-conformités aux exigences édictées par la fondation Eclipse ou par les projets eux-même sont rendues clairement visibles par des points d'action listant les sujets considérés comme faibles (e.g. activité des listes de diffusion, retard sur les milestones, etc.). La capture d'écran située à droite de la figure 3 liste ainsi certaines violations de code détectées sur un projet, avec les fonctions incriminées et le delta par rapport à la dernière analyse – il est toujours plus facile de reprendre un code qui vient d'être modifié. Enfin, certaines préoccupations font l'objet d'un traitement spécifique, comme les fichiers très complexes (flots de contrôle et de données, taille) ou instables (ayant eu un grand nombre de révisions). Leur identification visuelle donne une autre vision de la structure du projet et permet d'améliorer directement la qualité du produit.

6 Directions futures

La seconde itération du modèle de qualité Eclipse s'appuiera sur les travaux réalisés pour raffiner les exigences de qualité, améliorer les liens avec les métriques et calibrer le modèle pour définir les seuils acceptables de validité des projets. De nouvelles sources de données seront intégrées au modèle de qualité : ainsi les tests (e.g. : nombre, ratio, couverture), les informations d'intégration continue (nombre et ratio de builds échoués, temps moyen de compilation et de test), et les statistiques issues des moyens de communication (wiki, site web, downloads). De nouveaux types d'artéfacts et mesures sont également prévus à partir des référentiels existants : exigences tirées de la gestion des changements, nouvelles informations tirées du référentiel de processus.

Ce travail a été présenté à la communauté Eclipse lors de la conférence EclipseCon France 2013 sise à Toulouse en Juin 2013, et son industrialisation est en cours pour une publication prévue en 2014. Il est également envisagé d'étendre la démarche de qualité initiée pour Polarsys aux projets Eclipse pour compléter les initiatives actuelles PMI et Dash. La démarche sera rejouée pour définir un nouveau modèle de qualité adapté, avec des exigences de qualité différentes et des contraintes plus légères.

7 Conclusion

Le prototype développé dans le cadre du groupe de travail Polarsys a permis de :

– **Définir un socle méthodologique commun** pour discuter de la notion de qualité. Le modèle de qualité évoluera pour inclure de nouvelles fonctionnalités ou contraintes de qualité, mais sa structure a fait l'objet d'un assentiment général dans le groupe de travail et la communauté attachée.

– **Définir un ensemble de métriques** nouvelles ou peu courantes pour évaluer la notion de qualité. Les informations issues des référentiels de développement apportent notamment une vision plus complète du projet.
– **Montrer la faisabilité d'une telle solution**. Le prototype a apporté la preuve que les informations pouvaient être utilisées de manière pragmatique et efficace pour l'amélioration des projets logiciels, que ce soit au niveau du code ou des pratiques de développement.

Mais les leçons tirées de cette expérience de fouille de données dans un contexte industriel et communautaire ne sont pas limitées à l'écosystème Eclipse et peuvent être appliquées à d'autres projets et d'autres forges, tant pour le monde libre que pour les projets de développement internes. L'utilisation de forges complètes et intégrées est d'une grande aide pour la collecte des informations sur le cycle de vie des projets et un facteur important de succès pour la démarche d'évaluation et d'amélioration de la qualité. En ce sens des projets d'interopérabilité tels que OSLC devraient permettre à l'avenir de faciliter cette intégration.

Les concepts clefs évoqués forment un cadre méthodologique qui peut être adapté aisément à d'autres environnements. Ainsi la **participation des acteurs** est fondamentale pour obtenir une synergie constructive et une reconnaissance globale de la démarche. **La déclaration d'intention du programme de mesure** et sa décomposition en un modèle de qualité permettent d'éviter les mauvaises utilisations et dérives et assurent la qualité et l'utilisabilité des informations retirées. **La méthode de calcul des mesures** doit être clairement expliquée afin de permettre aux utilisateurs finaux d'en retirer le plus grand bénéfice et de participer à l'évolution du programme. Enfin, **la présentation des données** permet de transformer l'information en connaissance utile pour l'amélioration du produit ou du processus de développement.

Références

Bachmann, A., C. Bird, F. Rahman, P. Devanbu, et A. Bernstein (2010). The Missing Links : Bugs and Bug-fix Commits.

Baldassari, B. (2012). SQuORE : A new approach to software project quality measurement. In *International Conference on Software & Systems Engineering and their Applications*.

Basili, V. et D. Weiss (1984). A methodology for collecting valid software engineering data. *IEEE Trans. Software Eng. 10*(6), 728–738.

Basili, V. R., G. Caldiera, et H. D. Rombach (1994). The Goal Question Metric approach.

Fenton, N. (1994). Software Measurement : a Necessary Scientific Basis. *IEEE Transactions on Software Engineering 20*(3), 199–206.

Gopal, A., M. Krishnan, T. Mukhopadhyay, et D. Goldenson (2002). Measurement programs in software development : determinants of success. *IEEE Transactions on Software Engineering 28*(9), 863–875.

Hatton, L. (2007). The chimera of software quality. *Computer 40*(8), 103—-104.

Herbold, S., J. Grabowski, H. Neukirchen, et S. Waack (2010). Retrospective Analysis of Software Projects using k-Means Clustering. In *Proceedings of the 2nd Design for Future 2010 Workshop (DFF 2010), May 3rd 2010, Bad Honnef, Germany*.

ISO (2005). ISO/IEC 9126 Software Engineering - Product Quality - Parts 1-4. Technical report, ISO/IEC.

Iversen, J. et L. Mathiassen (2000). Lessons from implementing a software metrics program. In *Proceedings of the 33rd Annual Hawaii International Conference on System Sciences*, pp. 1—-11. IEEE.

Kaner, C. et W. P. Bond (2004). Software engineering metrics : What do they measure and how do we know ? In *10th International Software Metrics Symposium, METRICS 2004*, pp. 1–12.

Louridas, P. (2006). Static Code Analysis. *IEEE Software 23*(4), 58—-61.

Menzies, T., C. Bird, et T. Zimmermann (2011). The inductive software engineering manifesto : Principles for industrial data mining. In *Proceedings of the International Workshop on Machine Learning Technologies in Software Engineering*, Lawrence, Kansas, USA.

Organization, I. S. (2005). ISO/IEC 25000 :2005 Software engineering - Software product Quality Requirements and Evaluation (SQuaRE) - Guide to SQuaRE. Technical report, ISO/IEC, Geneva.

Shaw, M. (1990). Prospects for an Engineering Discipline of Software. *IEEE Software* (November), 15–24.

Smart, J. (2011). *Jenkins : The Definitive Guide*. O'Reilly Media, Inc.

Spinellis, D. (2006). Bug Busters. *IEEE Software 23*(2), 92—-93.

Westfall, L. et C. Road (2005). 12 Steps to Useful Software Metrics. *Proceedings of the Seventeenth Annual Pacific Northwest Software Quality Conference 57 Suppl 1*(May 2006), S40–3.

Xie, Y. (2013). Knitr : A general-purpose package for dynamic report generation in R. Technical report.

Yoon, K.-A., O.-S. Kwon, et D.-H. Bae (2007). An Approach to Outlier Detection of Software Measurement Data using the K-means Clustering Method. In *Empirical Software Engineering and Measurement*, pp. 443–445.
English

Zhong, S., T. Khoshgoftaar, et N. Seliya (2004). Analyzing software measurement data with clustering techniques. *IEEE Intelligent Systems 19*(2), 20–27.

Summary

Knowledge discovery and extraction from software-related repositories had great advances in recent years with the introduction of statistical methods to software engineering concerns and the development of software repositories data mining. In this article, we present a data mining program initiated within the Eclipse foundation to assess and improve software project maturity. We thoroughly describe the approach and the key concepts retained, list the encountered issues and propose solutions for them. Results are discussed and future directions proposed to foster further work in the domain.

De l'ombre à la lumière : plus de visibilité sur l'Eclipse

Boris Baldassari*, Flavien Huynh*, Philippe Preux**

* 76 Allées Jean Jaurès, Toulouse, France.
boris.baldassari@squoring.com – http://www.squoring.com
** Université de Lille 3, LIFL (UMR CNRS) & INRIA, Villeneuve d'Ascq, France
philippe.preux@inria.fr – https://sequel.inria.fr

Résumé. L'extraction de connaissances à partir de données issues du génie logiciel est un domaine qui s'est beaucoup développé ces dix dernières années, avec notamment la fouille de référentiels logiciels (Mining Software Repositories) et l'application de méthodes statistiques (partitionnement, détection d'outliers) à des thématiques du processus de développement logiciel. Cet article présente la démarche de fouille de données mise en œuvre dans le cadre de Polarsys, un groupe de travail de la fondation Eclipse, de la définition des exigences à la proposition d'un modèle de qualité dédié et à son implémentation sur un prototype. Les principaux concepts adoptés et les leçons tirées sont également passés en revue.

1 Introduction

L'évolution et le croisement des disciplines de la fouille de données et du génie logiciel ouvrent de nouveaux horizons pour la compréhension et l'amélioration du logiciel et des pratiques de développement. A l'aube de ce domaine émergent de nombreuses questions restent cependant d'actualité du point de vue de la conduite de programmes de mesure logicielle, notamment relevées par Fenton (1994) et Kaner et Bond (2004). Nous proposons dans cet article quelques pistes pour répondre à ces problématiques et mettre en place un processus de mesure fiable et efficace.

Il est utile pour la définition de la notion de qualité de s'appuyer sur des modèles ou standards reconnus : du point de vue de *la qualité produit*, la référence *de facto* semble être l'ISO 9126 et son futur successeur, la série 250xx SQuaRE. La *maturité du processus de développement* est adressée par des initiatives largement reconnues telle que le CMMi ou l'ISO 15504. Afin de clarifier la démarche de mesure, l'approche *Goal-Question-Metric* proposée par Basili et al. et reprise par Westfal Westfall et Road (2005) permet une approche plus rigoureuse, qui préserve l'efficacité de l'analyse et le sens de ses résultats.

2 Topologie d'un projet de développement logiciel

Chaque référentiel d'information utilisé par le projet possède des informations exploitables. Il importe de lister les référentiels disponibles, puis d'identifier pour chacun d'eux les artéfacts et mesures disponibles, et de leur donner un contexte sémantique.

Le code source est le type d'artéfact le plus utilisé pour l'analyse de projets logiciels. Du point de vue de l'analyse statique (Louridas, 2006), les informations que l'on peut récupérer d'un code source sont les *métriques*, correspondant à la mesure de caractéristiques définies du logiciel (e.g. sa taille ou la complexité de son flot de contrôle), et les *violations*, correspondant au non-respect de bonnes pratiques ou de conventions de codage ou de nommage (e.g. l'obligation de clause default dans un switch). Ces informations sont fournies par des analyseurs tels que Checkstyle, PMD ou SQuORE (Baldassari, 2012).

La gestion de configuration contient l'ensemble des modifications faites sur l'arborescence du projet, avec des méta-informations sur *l'auteur*, la *date* ou *l'intention* des changements. Le positionnement dans l'arbre des versions est important, car les résultats ne seront pas les mêmes pour une version en cours de développement (développée sur le tronc) et pour une version en maintenance (développée sur une branche).

La gestion des tickets recense l'ensemble des demandes de changement faites sur le projet. Ce peuvent être des problèmes (bugs), de nouvelles fonctionnalités, ou de simples questions.

Les listes de diffusion sont les principaux moyens de communication utilisés au sein de projets logiciels. Il existe en général au moins deux listes, une dédiée au développement et l'autre aux questions utilisateur.

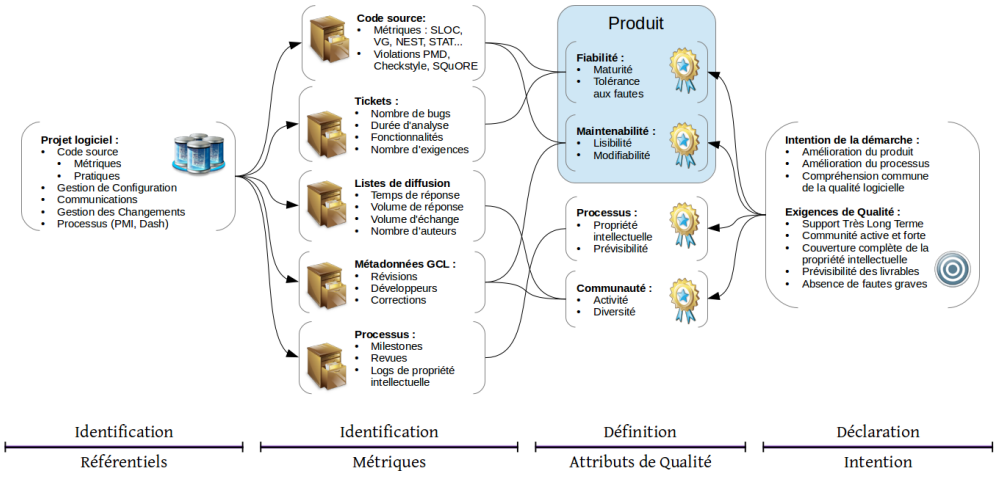


FIG. 1 – *Processus de définition du modèle de qualité et du choix des métriques.*

3 Présentation de la méthode d'extraction

La fouille de données logicielles est similaire en plusieurs points à la conduite de programmes de mesures. L'expérience glanée au fil des années sur ce dernier domaine (voir notamment Gopal et al. (2002) et Menzies et al. (2011)) a permis d'établir de bonnes pratiques qui aident à la définition, à l'implémentation et à l'utilisation des résultats d'analyse.

La déclaration d'intention donne un cadre méthodologique et sémantique pour la compréhension et l'interprétation des résultats. Par exemple, les mesures accessibles et les résultats attendus ne seront pas les mêmes pour un audit et pour un programme d'amélioration continue de la qualité. L'intention doit être *simple* et *tenir en quelques phrases*.

La décomposition des attributs de qualité décompose les objectifs de la fouille et forme le lien entre une déclaration informelle d'intention et les mesures concrètes qui vont permettre de mesurer et d'améliorer la qualité ainsi définie ; elle doit faire l'objet d'un consensus général et être validée par les acteurs du programme.

La définition des métriques accessibles à partir des référentiels identifiés sur le projet doit être *fiable* – i.e. l'information recherchée est systématiquement présente et valide – et *compréhensible* pour garder la confiance des acteurs dans le processus.

L'implémentation du processus de collecte et d'analyse doit être *transparente* pour que les acteurs puissent se l'approprier, intégralement *automatisée*, et *exécutée de manière régulière*.

La présentation des informations est capitale. Dans certains cas une liste concise d'artéfacts est suffisante, alors que dans d'autres cas un graphique bien choisi sera plus adapté et délivrera en quelques secondes l'essentiel du message.

4 Mise en pratique avec Eclipse

Cette approche a été mise en œuvre dans le cadre de Polarsys, un groupe de travail de la fondation Eclipse qui a pour but, entre autres, de proposer un cadre d'évaluation de la qualité des projets de la fondation. L'arbre de qualité montré en figure 2 montre les exigences identifiées pour Eclipse et Polarsys, et leur organisation en attributs de qualité.

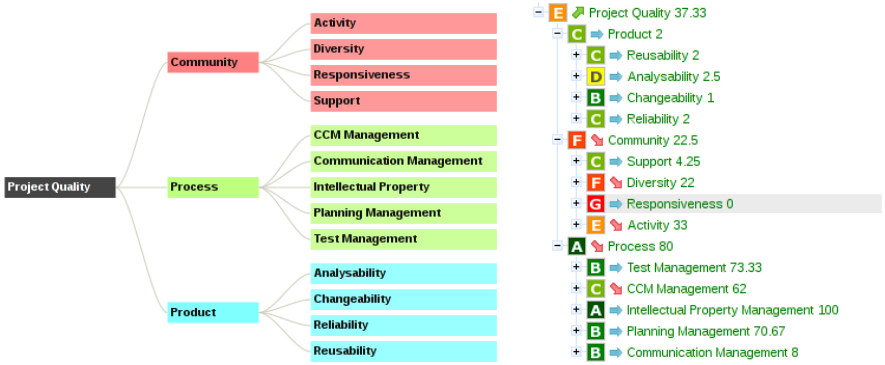


FIG. 2 – *Modèle de qualité proposé pour Polarsys.*

Du point de vue de **l'aide à la prise de décision**, les axes de qualité choisis permettent d'évaluer la maturité du projet immédiatement, selon les critères propres à l'organisation. Par exemple le projet analysé sur la droite de la figure 2 montre une bonne qualité produit, mais une faible activité de communication et un petit nombre de participants, car l'essentiel des contributions est fait par une unique société. Cette caractéristique peu visible *a priori* en fait un risque pour la pérennité du projet, rapidement identifiable sur l'arbre de qualité. Du côté

des équipes de développement, **l'aide à l'amélioration de la qualité** se fait au moyen de listes de violations et de points d'action, qui permettent d'identifier et d'améliorer les pratiques non acquises et fournissent des recommandations pragmatiques pour l'amélioration du code et du processus.

5 Conclusion

Le prototype développé dans le cadre du groupe de travail Polarsys a permis de **définir un socle méthodologique commun** et **un ensemble de métriques** issues de sources nouvelles pour travailler ensemble sur la notion de qualité, et a plus généralement permis de **démontrer la faisabilité d'une telle solution**. Le prototype a apporté la preuve que des connaissances pratiques pouvaient être extraites du projet pour l'évaluation et l'amélioration de la maturité. Ce travail a été présenté à la communauté Eclipse lors de la conférence EclipseCon France 2013 sise à Toulouse en Juin 2013, et son industrialisation est en cours pour une publication prévue en 2014. Il est également envisagé d'étendre la démarche de qualité initiée pour Polarsys aux projets Eclipse pour compléter les initiatives actuelles PMI et Dash.

Références

Baldassari, B. (2012). SQuORE : A new approach to software project quality measurement. In *International Conference on Software & Systems Engineering and their Applications*.

Fenton, N. (1994). Software Measurement : a Necessary Scientific Basis. *IEEE Transactions on Software Engineering 20*(3), 199–206.

Gopal, A., M. Krishnan, T. Mukhopadhyay, et D. Goldenson (2002). Measurement programs in software development : determinants of success. *IEEE Transactions on Software Engineering 28*(9), 863–875.

Kaner, C. et W. P. Bond (2004). Software engineering metrics : What do they measure and how do we know ? In *10th Internationalwestfal Software Metrics Symposium*, pp. 1–12.

Louridas, P. (2006). Static Code Analysis. *IEEE Software 23*(4), 58–61.

Menzies, T., C. Bird, et T. Zimmermann (2011). The inductive software engineering manifesto : Principles for industrial data mining. In *Proceedings of the International Workshop on Machine Learning Technologies in Software Engineering*, Lawrence, Kansas, USA.

Westfall, L. et C. Road (2005). 12 Steps to Useful Software Metrics. *Proceedings of the 17th Annual Pacific Northwest Software Quality Conference 57 Suppl 1*(May 2006), S40–3.

Summary

Knowledge discovery and extraction from software-related repositories had great advances in recent years. In this article, we present a data mining program initiated within the Eclipse foundation to assess and improve software project maturity. We thoroughly describe the approach and the key concepts retained. Results are discussed and future directions proposed to foster further work in the domain.

De l'ombre à la lumière : plus de visibilité sur Eclipse

Boris Baldassari, SQuORING Technologies – **Flavien Huynh**, SQuORING Technologies – **Philippe Preux**, INRIA Lille

Connaissance

Obscurité

Technique

Fonctionnel

Développeurs

- Pratiques acquises
- Points d'action
- Aide à la conduite de projet

Estimation objective de la maturité

Industriels

Utilisateurs

The inductive software engineering manifesto: Principles for industrial data mining.
Menzies, T., C. Bird, et T. Zimmermann (2011).

Projet logiciel :
- Code source
  - Métriques
  - Pratiques
- Gestion de Configuration
- Communications
- Gestion des Changements
- Processus (PMI, Dash)

Code source:
- Métriques : SLOC, VG, NEST, STAT...
- Violations PMD, Checkstyle, SQuORE

Tickets :
- Nombre de bugs
- Durée d'analyse
- Fonctionnalités
- Nombre d'exigences

Listes de diffusion
- Temps de réponse
- Volume de réponse
- Volume d'échange
- Nombre d'auteurs

Métadonnées GCL :
- Révisions
- Développeurs
- Corrections

Processus :
- Milestones
- Revues
- Logs de propriété intellectuelle

Produit

Fiabilité :
- Maturité
- Tolérance aux fautes

Maintenabilité :
- Lisibilité
- Modifiabilité

Processus :
- Propriété intellectuelle
- Prévisibilité

Communauté :
- Activité
- Diversité

Intention de la démarche :
- Amélioration du produit
- Amélioration du processus
- Compréhension commune de la qualité logicielle

Exigences de Qualité :
- Support Très Long Terme
- Communité active et forte
- Couverture complète de la propriété intellectuelle
- Prévisibilité des livrables
- Absence de fautes graves

Software quality models: Purposes, usage scenarios and requirements
F. Deissenboeck, E. Juergens, K. Lochmann, and S. Wagner (2009).

Implémentation

Publication

Sémantique, Méthode de calcul

Métriques

Accessibilité & Fiabilité de l'information

Référentiels

Identification — Référentiels

Identification — Métriques

Définition — Attributs de Qualité

Déclaration — Intention

Prévisible

Maintenable

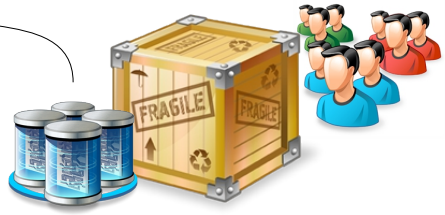
Pérenne

Fiable

ISO

12 Steps to Useful Software Metrics.
Westfall, L. et C. Road (2006).

The Goal Question Metric approach.
Basili, V. R., G. Caldiera, et H. D. Rombach (1994).



PolarSys
Open Source Tools for Embedded Systems

eclipse

Je veux un bon logiciel.

Je veux un logiciel.

Evaluating the quality of open source software
D. Spinellis and G. Gousios (2009).

Software quality: the elusive target
B. Kitchenham and S. Pfleeger (1996).

Outliers Detection in Software Engineering data

Boris Baldassari
SQuORING Technologies
74 Allées Jean Jaurès
31000 Toulosue, France
boris.baldassari@squoring.com

Philippe Preux
SequeL INRIA Lille
Campus
59000 Lille, France
philippe.preux@inria.fr

ABSTRACT

Outliers detection has been intensively studied and applied in recent years, both from the theoretical point of view and its applications in various areas. These advances have brought many useful insights on domain-specific issues giving birth to visible improvements in industry systems.

However this is not yet the case for software engineering: the few studies that target the domain are not well known and lack pragmatic, usable implementation. We believe this is due to two main errors. Firstly the semantics of the outliers search has not been clearly defined, thus threatening the understandability, usability and overall relevance of results. Outliers must be linked to software practitioners' needs to be useful and to software engineering concepts and practices to be understood. Secondly the nature of software engineering data implies specific considerations when working with it.

This article will focus on these two issues to improve and foster the application of efficient and usable outliers detection techniques in software engineering. To achieve this we propose some guidelines to assist in the definition and implementation of software outliers detection programs, and describe the application of three outliers detection methods to real-world software projects, drawing usable conclusions for their utilisation in industrial contexts.

Keywords

Outliers Detection, Data Mining, Software Metrics

1. INTRODUCTION

Generally speaking outliers in a data set are often detected as noise in the data cleaning and preprocessing phase of a mining process [18]. But outliers also have a wealth of useful information to tell and in some cases they are the primary goal of an analysis [9]: their application in specific, domain-aware contexts has led to important advances and numerous reliable integration in industrial products [25, 40].

International Conference on Software Engineering 2014, Hyderabad, India

As observed by Shaw [44] some years ago, the discipline of software engineering is not yet mature – one of the reasons being that mathematical and scientific concepts are still difficult to apply to the domain, and many concepts are not generally agreed upon by specialists. In this context the application of data mining techniques should bring many new insights into the field. Among them, the meaning and usefulness of outliers has not been extensively studied, althought there exists a considerable need from actors of the development process.

Considering the huge variety in nature of data, there is no single generic approach to outliers detection [31]. One has to carefully select and tailor the different techniques at hand to ensure the relevance of the results. Furthermore software measurement data has specific semantics and characteristics that make some algorithms inefficient or useless, and the results often lack practical considerations – i.e. *what we are really looking for* when mining for outliers. We believe this has a strong impact on the afterwards practical usability of these techniques.

In this paper we intend to define a lightweight semantic framework for them, establish use cases for their utilisation and draw practical requirements for their exploitation. We then demonstrate this approach by applying outliers detection to two common software engineering concerns: obfuscated files and untestable functions. Results are compared to experience-based detection techniques and submitted to a team of software engineering experts for cross-validation. It appears that provided a semantic context software developers and practitionners agree on the resulting set and find it both useful and usable.

This paper is organised as follows: in section 2 we give the usual definitions and usages of outliers in statistics and in the specific context of software engineering, and review recent related work. Section 3 defines a semantic framework for our research, stating what we are looking for and what it really means for us. In section 4 we describe some outliers detection techniques we intend to use: tail-cutting, boxplots and clustering-based, and we apply them to real-world software development concerns in section 5 with comments from field experts. We finally draw our conclusions and propose directions for future work in section 6.

2. STATE OF THE ART

2.1 What is an outlier?

One of the oldest definition of outliers still used today comes from Hawkins [9]: an outlier is *an observation that*

deviates so much from other observations as to arouse suspicion that it was generated by a different mechanism. Another often encountered definition for outliers in scientific literature comes from Barnett and Lewis [9], who define an outlier as *an observation (or subsets of observations) which appears to be inconsistent with the remainder of that set of data.* Both are useful to get the idea of outliers, the former addressing rather the causes and the later the consequences.

However in domain-specific contexts (e.g. healthcare, fraud detection, or software engineering) there is a lot more to say about what outliers are and what they mean. As applied to software engineering, Yoon et al. [51] define outliers as *the software data which is inconsistent with the majority data.* For Wohlin et al. [49] an outlier *denotes a value that is atypical and unexpected in the data set.* Both are close to Barnett's definition. Lincke et al. [30] take a more statistical perspective and define outliers as *artefacts with a metric value that is within the highest/lowest 15% of the value range defined by all classes in the system.*

2.2 Usages of outliers detection

From the data mining perspective, one of the main uses of outliers detection techniques is to clean data before processing [18, 33]: simple statistical estimates like variance or mean, correlation coefficients in regression models or prediction models may be biased by individual outliers [27].

Outliers may arise because of human error, instrument error, natural deviations, fraudulent behaviour, changes in behaviour, or faults in the system. In some cases they should be identified to be discarded, in other cases they really represent a specific and valuable information that will help improve the process, the product, or people's behaviour [27].

Many studies apply outliers detection to security-related data for e.g. insurance, credit card or telecommunication fraud detection [25, 40], insider trading detection [46] and network intrusion detection systems [29] by detecting deviating behaviours and unusual patterns. Outliers detection methods have also been applied to healthcare data [28, 50] to detect anomalous events in patient's health, military surveillance for enemy activities, image processing [46], or industrial damages detection [11].

Studies that specifically target the application of mining techniques to the software engineering domain are rare. Mark C. Paulk [39] applies outliers detection and Process Control charts in the context of the Personal Software Process (PSP) to highlight individuals performance. Yoon et al. [51] propose an approach based on k-means clustering to detect outliers in software measurement data, only to remove them as they threaten the conclusions drawn from the measurement program. In the later case the detection process involves the intervention of software experts and thus misses full automation. Other mining techniques have been applied with some success to software engineering concerns, such as recommendation systems to aid in software's maintenance and evolution [8, 42] or testing [26].

2.3 Outliers detection methods

The most trivial approaches to detect outliers are manual inspection of scatterplots or boxplots [28]. However when analysing high-volume or high-dimensional data automatic methods are more effective. Most common methods are listed below; they can be classified according to different criteria: univariate or multivariate, parametric or non-parametric [18, 11].

Distribution-based approaches are parametric in nature: they assume the existence of an underlying distribution of values and flag all points that deviate from the model [9]. They are not recommended in software engineering, mainly because of the distribution assumption [23, 37, 17] and their complexity in high dimensions [9].

Distance-based approaches [24, 45] generalise many concepts from distribution-based approaches and rely on a distance between points to identify outliers: points that have few neighbours in a given distance are considered as outliers [9]. They suffer exponential computational growth as they are founded on the calculation of the distances between all records [18].

Density-based methods [7, 34] assign an "outlierness" coefficient to each object, which depends on the region density around the object. They can be applied on univariate or multivariate series, and are generally more capable of handling large data sets [11].

Clustering-based approaches [51, 2, 45] apply unsupervised classification to the data set and identify points that either are not included in any cluster or constitute very small clusters [45]. They can be used on univariate or multivariate series and scale well to high dimensions, but may be computationally expensive depending on the clustering approach selected [11].

Recent studies also propose outliers detection methods borrowed from artificial intelligence, like fuzzy-based detection [9], neural networks [46] or support vector machines [9].

Another close research area is the detection of outliers in large multi-dimensional data sets. There is a huge variety of metrics that can be gathered from a software project, from code to execution traces or software project's repositories, and important software systems commonly have thousands of files, mails exchanged in mailing lists or commits. This has two consequences: first in high dimensional space the data is sparse and the notion of proximity fails to retain its meaningfulness [1]. Second some algorithms are unefficient or even not applicable in high dimensions [1, 15].

3. OUTLIERS IN SOFTWARE ENGINEERING

3.1 About software engineering data

The nature of software measurement data has a great impact on the methods that can be used on it and on the semantic of results. Firstly every method making an assumption about the underlying repartition of values is biased, as we will see in the next section. Secondly the definition of measures is still a subject of controversy and may vary widely across programming languages, tools, or authors, as identified by Kaner and Bond [20]. Since there is no established standard, one has to clearly state the definition she will be using and seriously consider the consistency of the measures: once a measurement system is decided, all other measurements should happen in the exact same way – reproducible research tools like literate data analysis are precious helpers for this purpose.

Another point is that many software metrics have high correlation rates between them. As an example line-counting metrics (e.g. overall/source/effective lines of code, number of statements) are heavily correlated together – and they

often represent the majority of available metrics. Complexity metrics (cyclomatic number, number control flow tokens, number of paths) are highly correlated as well. When using multivariate methods these relationships must be considered since they may put a heavy bias on results.

From the history mining perspective, practitionners will encounter missing data, migrations between tools and evolving local customs. As an example while most projects send commit or bug notifications to a dedicated mailing list or RSS feed, projects with a low activity may simply send them to the developer mailing list. In this case one needs to filter the messages to get the actual and accurate information. In the generic case every step of the mining analysis must be checked against the specific local requirements of projects.

3.2 Impact of distribution

No assumption can be made on the distribution of software-related observations. Although some models have been proposed in specific areas like the Pareto [14] and Weibull [52] distributions for faults repartition in modules, or power laws in dependency graphs [32], to our knowledge there is very little scientific material about the general shape of common software metrics. And at first sight empirical software data does not go normal or Gaussian, as shown in figure 1 for some metrics of Ant 1.7 source files: source lines of code (SLOC), cyclomatic complexity (VG), comment rate (COMR) and number of commits (SCM_COMMITS).

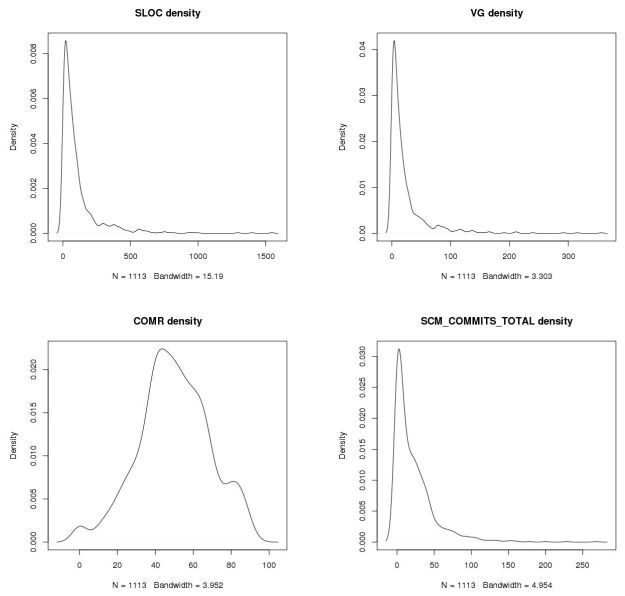


Figure 1: Distribution of some common file metrics for Ant 1.7.

From an univariate perspective the number of extreme outliers is proportional to the surface of the tail of the density function. In the case of heavy tailed distributions more outliers will be found than for lightly tailed distributions. In the case of long tails as seen in many metrics distribution functions, the threshold of 15% of the maximum value defined by Lincke et al. in [30] gives a small number of artefacts considered as outliers.

3.3 Defining outliers for software

One of the early steps in a data mining process is to define exactly the objectives of the search. We believe this is even more true in the context of industry developments to make it both usable and used. End-users and practitionners must understand what outliers are to make good use of them.

Software engineers and developers have an intuitive idea of some outlier types. When looking at a piece of code experienced developers will instinctively tag it and identify special characteristics. As an example they can identify "classes" of files that they may consider as difficult to understand or modify, or "strange patterns" like obfuscated code or declaration-only files. We need to map the outliers we are looking for to those *specific classes of files* known to users.

The volume of outliers output by the method is an important parameter. We may want a fairly small amount of them for incremental improvement of practices and quality, but in some cases (e.g. an audit) an extended list is preferable to help estimate the amount of technical debt of the software.

To achieve this some mandatory information should be stated and published to ensure it will be understood and safely used according to its original aim. We propose the following template to be answered everytime outliers detection (or more generally data mining) technique is applied.

Define what outliers are for the user. They usually correspond to a specific need, like unstable files in the context of a release or overly complex functions for testing. Maybe developers, managers or end-users already have a word coined for this concern.

How is this information used? Metrics programs may have counter-productive effects if they are not well understood [20][1]. To avoid this the selected method and intent should be explicitely linked to requirements and concepts common to the end users so they can attach their own semantic to the results and make better use of them.

How should results be presented? Visualisation is the key to understanding, and a good picture is often worth a thousand words. Select a presentation that shows why the artefact is considered as an outlier. Sometimes a list of artefacts may be enough, or a plot clearly showing the extreme value of the outlier compared to other artefacts may be more eye- and mind- catching.

In the context of software quality improvement, having too many items to watch or identified as outliers would be useless for the end user; selecting a lower value gives more practical and understandable results. In that case clusters that have less than 15% of the maximum value as proposed by Lincke et al [30], outputs a fair and usable number of artefacts due to the light tail of the frequency distribution – one that can be considered by the human mind to be easily improved. In other cases (e.g. audits) an extended list of artefacts may be preferred.

3.4 Examples of outliers types

More specifically, outliers in software measurement data can be used to identify:

Noise.

Artefacts that should be removed from the working set

[1]As an example productivity metrics solely based on lines of code tend to produce very long files without any productivity increase.

because they threaten the analysis and the decisions taken from there. As an example, if some files are automatically generated then including them in the analysis is pointless, because they are not to be maintained and it would be wiser to analyse the model or generation process.

Unstable files.
Files that have been heavily modified, either for bug fix or enhancements.

Complexity watch list.
Artefacts that need to be watched, because they have complexity and maintainability issues. When modifying such artefacts developers should show special care because they are more likely to introduce new bugs [6]. This is rather linked to extreme values of artefacts.

Anomalous.
Artefacts that show a deviance from canonical behaviour or characteristics. A file may have a high but not unusual complexity, along with a very low SLOC – as it is the case for obfuscated code. This is often linked to dissimilarities in the metrics matrix.

Obfuscated code.
Artefacts that contain code that is obfuscated pose a real threat to understandability and maintainability. They should be identified and rewritten.

4. OUTLIERS DETECTION METHODS

4.1 Simple tail-cutting

As presented in section 3.2 many metric distributions have long tails. If we use the definition for outliers of Lincke et al. [30] and select artefacts with metrics that have values greater than 85% of their maximum, we usually get a fairly small number of artefacts with especially high values.

Table 1 shows the maximum value of a few metrics and their 5, 15, and 25% thresholds with the number of artefacts that fit in the range. Projects analysed are Ant 1.7 (1113 files), and various extracts from SVN: JMeter (768 files), Epsilon (777 files) and Sphinx (467 files).

The main advantage of this method is it allows to find points without knowing the threshold value that makes them peculiar: e.g. for the cyclomatic complexity, the recommended and argued value of 10 is replaced by a value that makes those artefacts *significantly* more complex than the vast majority of artefacts. This technique is a lot more selective than boxplots and thus produces less artefacts with higher values.

4.2 Boxplots

One of the simplest statistical outlier detection methods was introduced by Laurikkala et al. [28] and uses informal box plots to pinpoint outliers on individual variables. Simply put, boxplots draw first and third quartile

We applied the idea of series union and intersection[2] from Cook et al. [13] to outliers detection, a data point being a multi-dimensional outlier if many of its variables are themselves outliers. The rationale is that a file with unusually high complexity, size, number of commits, and say a very bad comment ratio *is* an outlier, because of its maintainability issues and development history.

[2]In [13] authors use univariate series unions and intersections to better visualise and understand data repartition.

Table 1: Number of outliers for percents of maximum metric value for some projects.

Ant	Max	25%	15%	5%
SLOC	1546	1160 (3)	1315 (2)	1469 (1)
NCC	676	507 (3)	575 (2)	643 (2)
VG	356	267 (3)	303 (2)	339 (1)
JMeter	Max	25%	15%	5%
SLOC	972	729 (3)	827 (1)	924 (1)
NCC	669	502 (5)	569 (3)	636 (1)
VG	172	129 (4)	147 (2)	164 (1)
Epsilon	Max	25%	15%	5%
SLOC	3812	3812 (6)	4320 (6)	4828 (6)
NCC	9861	7396 (6)	8382 (6)	9368 (2)
VG	1084	813 (6)	922 (6)	1030 (6)
Sphinx	Max	25%	15%	5%
SLOC	1195	897 (5)	1016 (4)	1136 (1)
NCC	1067	801 (2)	907 (2)	1014 (1)
VG	310	233 (2)	264 (2)	295 (2)

We first tried to apply this method to the full set of available metrics, by sorting the components according to the number of variables that are detected as univariate outliers. The drawback of this accumulative technique is its dependence on the selected set of variables: highly correlated metrics like line-counting measures will quickly trigger big components even if all of their other variables show standard values. Having a limited set of orthogonal metrics with low correlation is needed to balance the available information.

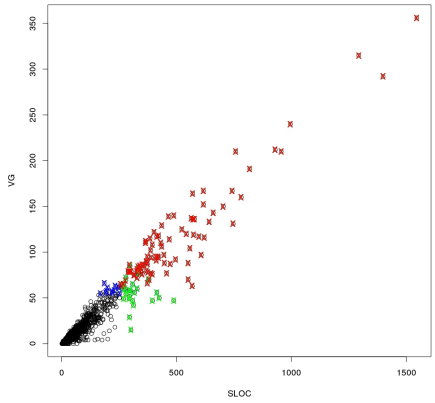


Figure 2: Univariate boxplot outliers on SLOC and VG for Ant 1.7.

Figure 2 shows the SLOC and VG metrics on files for the Ant 1.7 release, with outliers highlighted in blue for VG (111 items), green for SLOC (110 items), and red for the intersection of both (99 items). The same technique was applied with three metrics (SLOC, VG, NCC) and their intersection in figure 3. There are 108 NCC outliers (plotted in orange) and the intersecting set (plotted in red) has 85

outliers. Intersecting artefacts cumulate outstanding values on the three selected metrics.

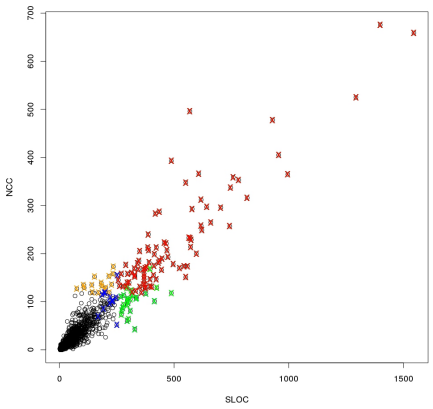


Figure 3: Univariate boxplot outliers on SLOC, VG and NCC for Ant 1.7.

Because of the underlying distribution of measures however the classical boxplot shows too many outliers – or at least too many for practical improvement. We used a more robust boxplots algorithm [43] targeted at skewed distributions[3] and had better results with fewer, more atypical artefacts.

4.3 Clustering

Clustering techniques allow us to find categories of similar data in a set. If a data point cannot fit into a cluster, or if it is in a small cluster (i.e. there are very few items that have these similar characteristics), then it can be considered an outlier [33, 45]. In this situation, we want to use clustering algorithms that produce unbalanced trees rather than evenly distributed sets. Typical clustering algorithms used for outliers detection are k-means [19, 51] and hierarchical [16, 3]. We used the latter because of the very small clusters it produces and its fast implementation in R [41].

We applied hierarchical clustering to file measures with different distances and agglomeration methods. On the Apache Ant 1.7 source release (1113 files) we get the repartition of artefacts shown in table 2 with Euclidean and Manhattan distances. Linkage methods investigated are Ward, Average, Single, Complete, McQuitty, Median and Centroid.

The Manhattan distance gives more small clusters than the Euclidean distance. The aggregation method used also has a great impact: the ward method draws more evenly distributed clusters while the single method consistently gives many clusters with only a few individuals.

Metrics selection has a great impact on results. They have to be carefully selected accordingly to the aim of the analysis. As an example the same detection technique (i.e. same distance and linkage method) was applied to different sets of metrics in figure 4: outliers are completely different, and are even difficult to see[4]. Using too many metrics usually gives bad or unusable results: because of their relationships, the diversity of information they deliver and our ability to capture this information and its consequences.

[3]Extremes of the upper and whiskers of the adjusted boxplots are computed using the medcouple, a robust measure of skewness.

[4]Remember section 3.3 about visualisation.

Table 2: File clusters.

Euclidean distance							
Method used	Cl1	Cl2	Cl3	Cl4	Cl5	Cl6	Cl7
Ward	226	334	31	97	232	145	48
Average	1006	6	19	76	3	1	2
Single	1105	3	1	1	1	1	1
Complete	998	17	67	3	21	3	4
McQuitty	1034	6	57	10	3	1	2
Median	1034	6	66	3	1	2	1
Centroid	940	24	142	3	1	2	1
Manhattan distance							
Method used	Cl1	Cl2	Cl3	Cl4	Cl5	Cl6	Cl7
Ward	404	22	87	276	62	196	66
Average	943	21	139	4	3	1	2
Single	1105	3	1	1	1	1	1
Complete	987	17	52	3	47	3	4
McQuitty	1031	12	60	3	1	4	2
Median	984	6	116	3	1	2	1
Centroid	942	24	140	3	1	2	1

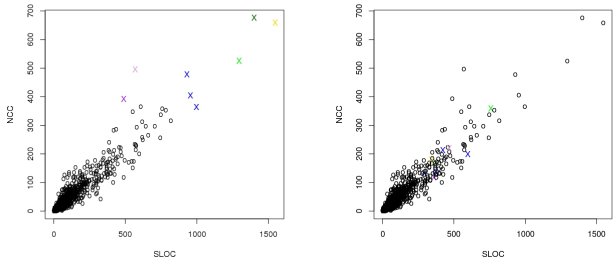


Figure 4: Outliers in clusters: different sets of metrics.

5. IMPLEMENTING OUTLIERS DETECTION

5.1 Overview

These techniques have been prototyped in SQuORE, a commercial software project quality assessment tool [4]. SQuORE has a modular, plugin-based architecture that makes it easy to integrate with various data providers, and provides a fully-featured dashboard for the presentation of results.

For our purpose, we decided to identify two different types of outliers: functions that are considered as untestable (high complexity, deep nesting, huge number of execution paths) and files with obfuscated code. For each of these we follow the directives proposed in section 3.3 and give their definition, usage, and implementation. The validation steps we used are also described.

Detected outliers are displayed in dedicated action lists, with the reasons of their presence in the list and corrective actions that can be undertaken to improve them. Coloured plots allow to highlight artefacts on various metrics, showing graphical evidence of their outlierness.

5.2 Untestable functions

5.2.1 Definition

Complex files are difficult to test and maintain because a large number of tests should be written before reaching a fair test coverage ratio, and developers who need to understand and modify the code will most probably have trouble understanding it, thus increasing the possibility of introducing new bugs [21, 22]. Furthermore complex files induce higher maintenance costs [5]. As a consequence, they shoud be identified so developers know they should be careful when modifying them. They also can be used as a short-list of action items when refactoring [36, 47].

Complexity relies on a number of parameters [48]. Traditional methods used by practitionners to spot untestable files usually rely on a set of control-flow related metrics with static thresholds: cyclomatic complexity (>10) and levels of nesting in control loops (>3) are common examples.

5.2.2 Implementation

We selected with our team of experts a set of three metrics that directly impact the complexity of code, thus threatening its testability, understandability and changeability: level of nesting in control loops (NEST), cyclomatic complexity (VG), and number of execution paths (NPAT). In this specific case we are solely interested in the highest values of variables. We also know that an extreme value on a single metric is enough to threaten the testability of the artefact. Hence we applied the simple tail-cutting algorithm to univariate series, and listed as outliers all artefacts that had at least one extreme metric value.

5.2.3 Validation

This technique was applied to a set of open-source C and Java projects[5] and the resulting set of functions was compared to those identified with traditional thresholds. While most untestable functions were present in both lists, we also identified artefacts that were exclusively selected by one or the other method. The traditional method needs all selected triggers on metrics to be met, hence an artefact with an extremely high nesting but with standard cyclomatic complexity would not be selected. Figure 5 shows such examples of functions with a low number of execution paths but a really deep nesting. On the other hand static triggers may give in some situations a huge number of artefacts, which is inefficient for our purpose.

We cross-validated these results by asking a group of software engineering experts to review the list of untestable functions that were found exclusively by our method. Their analysis confirmed that the resulting set was both complete and more practical: in the cases where our method outputs more untestable artefacts, they were consistently recognised as being actually difficult to test, read and change.

5.3 Obfuscated code

5.3.1 Definition

Code obfuscation is a protection mechanism used to limit the possibility of reverse engineering or attack activities on a software system [10]. They are also sometimes intended

[5]Projects analysed are: Evince, Agar, Amaya, Mesa, Ant, JMeter, Sphinx, Papyrus, Jacoco, HoDoKu, Freemind and JFreeChart.

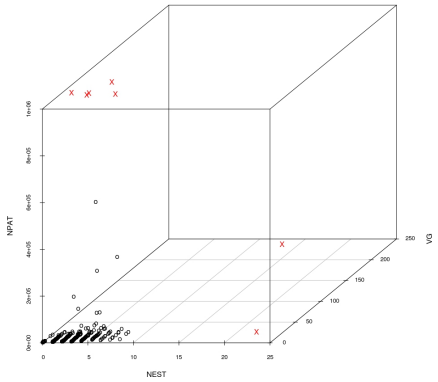


Figure 5: Untestable functions for Evince (2818 functions).

as challenging games by developers [38], but they generally threaten the understandability and maintainability of software. Bugs are also more difficult to detect and fix. Obfuscated code is usually identified by humans, although some automatic methods have been proposed for security-related concerns [12].

Obfuscated files should be identified, analysed and eventually rewritten. Clear warnings should be displayed when modifying them. They are rather rare events in the normal development of a product, but their impact is such that identifying them is of primary importance for the understandability and overall maintainability of a software product.

5.3.2 Implementation

One of the remarkable characteristics of obfuscated code is its unusual amount and diversity of operands and operators compared to the number of lines of code. We thus selected for our search Halstead's base metrics (n_1, n_2, N_1 and N_2) along with the size of artefact, measured as lines of code. Simple tail-cutting and clustering detection techniques were applied at both the function and file levels, and artefacts that were found by both methods tagged as obfuscated.

Very small files however tend to have an artificially high density because of their low number of lines rather than their number of operands or operators. For our purpose we decided to dismiss obfuscated code that is shorter than a few lines in the file-level analysis.

5.3.3 Validation

Obfuscated files retrieved from the International Obfuscated C Code Contest [38] were hidden in the source code of four open-source C projects (Evince, Agar, Amaya and Mesa). Analysis extracted them all with 100% accuracy. If we use the union of tail-cutting and clustering outliers instead of the intersection, we also get regular files from the projects with strong obfuscation characteristics. Figure 6 shows clusters (left) and tail-cutting (right) outliers on Halstead's n_1 and n_2 metrics. It is interesting to note that the coloured artefacts are not necessarily outliers on these two metrics since we are in a multi-dimensional analysis.

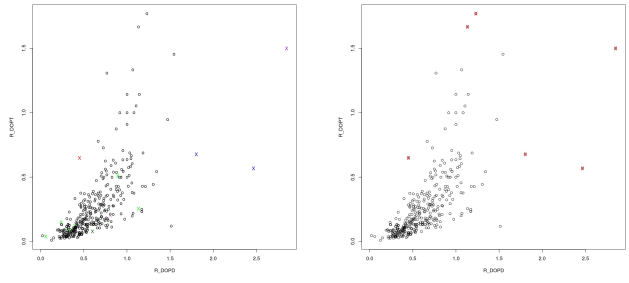


Figure 6: Obfuscated files from clusters and tail-cutting.

6. CONCLUSION

Outliers detection may bring very useful insights for software development, maintenance and comprehension if they are carefully specified. For this we need to define exactly what we are looking for and what it means to the user: specific outliers detection techniques should serve a clear cut purpose and fill specific requirements and needs for the user.

One of the main benefits of outliers detection techniques in the context of software engineering is they propose **dynamic triggering values** for artefacts selection. Dynamic tresholds have two advantages over static limits: firstly they usually find a fair (low) number of outliers where static thresholds may bring hundreds of them on complex systems, and secondly their ability to adapt themselves to a context make them applicable and more robust in a wider range of types of systems. As an example, McCabe recommands to not exceed a threshold of 10 for cyclomatic complexity in functions [35]. This has been widely debated since then, and no definitive value came out that could fit every domain. By using dynamic thresholds we are able to highlight very complex artefacts in the specific context of the project, and even to set a recommended value for the organisation's coding conventions.

The usage of outliers detection techniques shall be driven by the goals and semantic context of the mining task. As an example, it may be wise to detect obfuscated code and remove them *before* the analysis since they don't fit the usual quality requirements and may confuse the presentation and understanding of results. On the other hand maintainability outliers must be included in the analysis and results since they really bring useful information for software maintenance and evolution.

Different outliers detection techniques can be used and combined to get the most of their individual benefits. The three outliers detection techniques we studied have distinct characteristics: tail-cutting and boxplots look for *extreme values* at different output rate, while clusters rather find *similarities* in artefacts. By combining these techniques we were able to adapt the behaviour of the analysis and specifically identify given types of artefacts with great accuracy.

Metrics selection is of primary importance when applying mining techniques to data sets, because of the tight relationships between metrics and the semantic users can attach to them. A small set of carefully selected metrics is easier to understand and gives more predictable results.

7. REFERENCES

[1] C. C. Aggarwal and P. S. Yu. Outlier detection for high dimensional data. *ACM SIGMOD Record*, 30(2):37–46, June 2001.

[2] M. Al-Zoubi. An effective clustering-based approach for outlier detection. *European Journal of Scientific Research*, 2009.

[3] J. Almeida and L. Barbosa. Improving hierarchical cluster analysis: A new method with outlier detection and automatic clustering. *Chenometrics and Intelligent Laboratory Systems*, 87(2):208–217, 2007.

[4] B. Baldassari. SQuORE: A new approach to software project quality measurement. In *International Conference on Software & Systems Engineering and their Applications*, 2012.

[5] R. D. Banker, S. M. Datar, C. F. Kemerer, and D. Zweig. Software complexity and maintenance costs. *Communications of the ACM*, 36(11):81–94, Nov. 1993.

[6] B. Boehm and V. Basili. Software Defect Reduction Top 10 List. *Computer*, pages 123–137, 2001.

[7] M. M. Breunig, H.-P. Kriegel, R. T. Ng, and J. Sander. Local Outlier Factor: Identifying Density-Based Local Outliers. *ACM SIGMOD Record*, 29(2):93–104, June 2000.

[8] G. Canfora and L. Cerulo. Impact analysis by mining software and change request repositories. In *11th IEEE International Symposium on Software Metrics*, pages 1–9, 2005.

[9] S. Cateni, V. Colla, and M. Vannucci. Outlier Detection Methods for Industrial Applications. *Advances in robotics, automation and control*, pages 265–282, 2008.

[10] M. Ceccato, M. Di Penta, J. Nagra, P. Falcarin, F. Ricca, M. Torchiano, and P. Tonella. The effectiveness of source code obfuscation: An experimental assessment. In *2009 IEEE 17th International Conference on Program Comprehension*, pages 178–187. IEEE, May 2009.

[11] V. Chandola, A. Banerjee, and V. Kumar. Outlier Detection : A Survey. *ACM Computing Surveys*, 2007.

[12] M. Christodorescu and S. Jha. Static Analysis of Executables to Detect Malicious Patterns. Technical report, DTIC Document, 2006.

[13] D. Cook and D. Swayne. *Interactive and Dynamic Graphics for Data Analysis: With R and GGobi*. 2007.

[14] N. Fenton and N. Ohlsson. Quantitative analysis of faults and failures in a complex software system. *IEEE Transactions on Software Engineering*, 26(8):797–814, 2000.

[15] P. Filzmoser, R. Maronna, and M. Werner. Outlier identification in high dimensions. *Computational Statistics & Data Analysis*, 52(3):1694–1711, 2008.

[16] A. Gordon. *Classification, 2nd Edition*. Chapman and Hall/CRC, 1999.

[17] H. Hemmati, S. Nadi, O. Baysal, O. Kononenko, W. Wang, R. Holmes, and M. W. Godfrey. The MSR Cookbook. In *10th International Workshop on Mining Software Repositories*, pages 343–352, 2013.

[18] V. Hodge and J. Austin. A survey of outlier detection methodologies. *Artificial Intelligence Review*, 22(2):85–126, 2004.

[19] M. Jiang, S. Tseng, and C. Su. Two-phase clustering process for outliers detection. *Pattern recognition letters*, 22(6):691–700, 2001.

[20] C. Kaner and W. P. Bond. Software engineering metrics: What do they measure and how do we know? In *10th International Software Metrics Symposium, METRICS 2004*, pages 1–12, 2004.

[21] C. F. Kemerer. Software complexity and software maintenance: A survey of empirical research. *Annals of Software Engineering*, 1(1):1–22, Dec. 1995.

[22] T. Khoshgoftaar and J. Munson. Predicting software development errors using software complexity metrics. *IEEE Journal on Selected Areas in Communications*, 8(2):253–261, 1990.

[23] B. Kitchenham, S. Pfleeger, L. M. Pickard, P. W. Jones, D. C. Hoaglin, K. E. Emam, and J. Rosenberg. Preliminary guidelines for empirical research in software engineering. *IEEE Transactions on Software Engineering*, 28(8):721–734, 2002.

[24] E. Knorr and R. Ng. Algorithms for mining distance-based outliers in large datasets. In *Proceedings of the International Conference on Very Large Data Bases*, volume 8.3, 1998.

[25] Y. Kou, C.-T. Lu, S. Sirwongwattana, and Y.-P. Huang. Survey of fraud detection techniques. In *2004 IEEE International Conference on Networking, Sensing and Control*, pages 749–754, 2004.

[26] M. Last, M. Friedman, and A. Kandel. The data mining approach to automated software testing. In *Proceedings of the 9th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 388–396, 2003.

[27] M. Last and A. Kandel. Automated Detection of Outliers in Real-World Data. In *Proceedings of the second international conference on intelligent technologies*, pages 292–301, 2001.

[28] J. Laurikkala, M. Juhola, and E. Kentala. Informal identification of outliers in medical data. In *Proceedings of the 5th International Workshop on Intelligent Data Analysis in Medicine and Pharmacology*, pages 20–24, 2000.

[29] A. Lazarevic, L. Ertoz, and V. Kumar. A comparative study of anomaly detection schemes in network intrusion detection. *Proc. SIAM*, 2003.

[30] R. Lincke, J. Lundberg, and W. Löwe. Comparing software metrics tools. *Proceedings of the 2008 international symposium on Software testing and analysis - ISSTA '08*, page 131, 2008.

[31] A. Loureiro, L. Torgo, and C. Soares. Outlier detection using clustering methods: a data cleaning application. In *roceedings of KDNet Symposium on Knowledge-based systems for the Public Sector*, 2004.

[32] P. Louridas, D. Spinellis, and V. Vlachos. Power laws in software. *ACM Transactions on Software Engineering and Methodology*, 18(1):1–26, Sept. 2008.

[33] O. Maimon and L. Rokach. *Data Mining and Knowledge Discovery Handbook*. Kluwer Academic Publishers, 2005.

[34] M. Mansur, M. Sap, and M. Noor. Outlier Detection Technique in Data Mining: A Research Perspective. In *Proceedings of the Postgraduate Annual Research Seminar*, pages 23–31, 2005.

[35] T. McCabe. A complexity measure. *Software Engineering, IEEE Transactions on*, (4):308–320, 1976.

[36] T. Mens and T. Tourwé. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, 2004.

[37] T. Menzies, C. Bird, and T. Zimmermann. The inductive software engineering manifesto: Principles for industrial data mining. In *Proceedings of the International Workshop on Machine Learning Technologies in Software Engineering*, Lawrence, Kansas, USA, 2011.

[38] L. C. Noll, S. Cooper, P. Seebach, and A. B. Leonid. The International Obfuscated C Code Contest.

[39] M. C. Paulk, K. L. Needy, and J. Rajgopal. Identify outliers, understand the Process. *ASQ Software Quality Professional*, 11(2):28–37, 2009.

[40] C. Phua, V. Lee, K. Smith, and R. Gayler. A comprehensive survey of data mining-based fraud detection research. *arXiv preprint arXiv:1009.6119*, 2010.

[41] R Core Team. R: A Language and Environment for Statistical Computing, 2013.

[42] M. Robillard and R. Walker. Recommendation systems for software engineering. *Software, IEEE*, pages 80–86, 2010.

[43] P. Rousseeuw, C. Croux, V. Todorov, A. Ruckstuhl, M. Salibian-Barrera, T. Verbeke, M. Koller, and M. Maechler. {robustbase}: Basic Robust Statistics. R package version 0.9-10., 2013.

[44] M. Shaw. Prospects for an Engineering Discipline of Software. *IEEE Software*, (November):15–24, 1990.

[45] G. Singh and V. Kumar. An Efficient Clustering and Distance Based Approach for Outlier Detection. *International Journal of Computer Trends and Technology*, 4(7):2067–2072, 2013.

[46] K. Singh and S. Upadhyaya. Outlier Detection : Applications And Techniques. In *International Journal of Computer Science*, volume 9, pages 307–323, 2012.

[47] K. Stroggylos and D. Spinellis. Refactoring – Does It Improve Software Quality? In *Fifth International Workshop on Software Quality, ICSE Workshops 2007*, pages 10–16. IEEE, 2007.

[48] E. Weyuker. Evaluating software complexity measures. *IEEE Transactions on Software Engineering*, 14(9):1357–1365, 1988.

[49] C. Wohlin, M. Höst, and K. Henningsson. Empirical research methods in software engineering. *Empirical Methods and Studies in Software Engineering*, pages 7–23, 2003.

[50] W. Wong, A. Moore, G. Cooper, and M. Wagner. Bayesian network anomaly pattern detection for disease outbreaks. In *ICML*, pages 808–815, 2003.

[51] K.-A. Yoon, O.-S. Kwon, and D.-H. Bae. An Approach to Outlier Detection of Software Measurement Data using the K-means Clustering Method. In *Empirical Software Engineering and Measurement*, pages 443–445, 2007.

[52] H. Zhang. On the distribution of software faults. *Software Engineering, IEEE Transactions on*, 2008.

A practitioner approach to software engineering data mining

Boris Baldassari
SQuORING Technologies
76, Avenue Jean Jaurès
Toulouse, France
boris.baldassari@squoring.com

Philippe Preux
INRIA Lille
165, Avenue de Bretagne
Lille, France
philippe.preux@lille.inria.fr

ABSTRACT

As an emerging discipline, software engineering data mining has made important advances in recent years. These in turn have produced new tools and methods for the analysis and improvement of software products and practices.

Building upon the experience gathered on software measurement programs, data mining takes a wider and deeper view on software projects by introducing unusual kinds of information and addressing new software engineering concerns. On the one hand software repository mining brings new types of process- and product-oriented metrics along with new methods to use them. On the other hand the data mining field introduces more robust algorithms and practical tools, thus allowing to face new challenges and requirements for mining programs.

However some of the concerns risen years ago about software measurement programs and the validity of their results are still relevant to this novel approach. From data retrieval to organisation of quality attributes and results presentation, one has to constantly check the validity of the analysis process and the semantics of the results.

This paper focuses on practical software engineering data mining. We first draw a quick landscape of software engineering data, how they can be retrieved, used and organised. Then we propose some guidelines to conduct a successful and efficient software data mining program and describe how we applied it to the case of Polarsys, an Eclipse working group targeted at project quality assessment and improvement.

Keywords

Data Mining, Software Development, Software Metrics, Mining Repositories

1. INTRODUCTION

Software engineering data mining is at the crossroads of two developing fields that tremendously advanced in the recent years. From the software engineering perspective, lessons learned from years of failing and successful software projects have been cataloged and now provide a comprehensive set of practices and empirical knowledge – although some concepts still miss a common, established consensus among actors of the field. Software measurement programs have been widely studied and debated and now offer a comprehensive set of recommendations and known issues for practitioners. On the data mining side research areas like software repository mining [16], categorisation (clustering) of artefacts, program comprehension [11, 32], or developer assistance [36, 15, 25, 7] have brought new insights and new perspectives with regards to data available for analyses and the guidance that can be drawn from them.

This has led to a new era of knowledge for software measurement programs, which expands their scope beyond simple assessment concerns. Data mining programs have enough information at hand to leverage quality assessment to pragmatic recommendations and improvement of the process, practices and overall quality of software projects.

In this paper we review practices established for software measurement programs and consider them from the perspective of practical quality improvement, from data retrieval in software repositories to definition and organisation of quality attributes and to the publication of results. The experience gathered in the context of a high-constraints industrial software measurement program is reported to illustrate the proposed approach.

This paper is organised as follows: section 2 takes an inventory of the experience gained on software measurement programs and proposes in section 3 a series of steps to be followed to preserve the consistency and validity of data mining programs. Section 4 draws a landscape of the information that can be gathered on software projects, pointing out the difficulties and pitfalls of data retrieval. The approach is then illustrated through the Eclipse use case in section 5. Finally, section 6 summarises the main ideas of the paper and proposes future directions for upcoming work.

2. FROM MEASUREMENT TO DATA MINING

2.1 Software measurement programs

The art of measurement is very old and mature compared to software engineering. Some of the concepts that seem natural for usual measures (such as the height of a person) become difficult to apply when it comes to software engineering [12, 22]. As an example, one of the issues encountered with software measurement programs is the gap between quality models and metrics. Although many pa-

International Conference on Software Engineering 2014 Hyderabad, India

rameters are known to influence each quality factor (e.g. the maintainability), there is no established, comprehensive and recognised mapping between metrics and attributes of quality.

Furthermore people have different ideas and experiences of software characteristics, which makes it difficult to work on the same footing[1]. If the analysis process is too complex or not transparent enough, users will have trouble to retain the meaningfulness of data and results. Practictioners setting up a mining program should be familiar with the experience and conclusions of software measurement programs [19, 14, 17, 26] to avoid most common pitfalls.

To circumvent these problems, Victor Basili proposed in [5] the Goal-Question-Metric approach to better understand and conduct a software measurement program. Linda Westfall et al. [34] further enhanced this approach with 12 steps to be thoroughly followed for better chances of success. Many aspects of the approach we propose rely on the principles they have codified.

2.2 Notions of quality

If one intends to *improve software quality* then one will first need to define it in one's own specific context. Dozens of definitions have been proposed and widely debated for software quality, from *conformance to requirements* [9] to *fitness for use* [21], but none of them gained a definitive acceptance among the community or the industry – because the notion of quality may vary.

Standards like ISO 9126 [18] or the 250XX series propose interesting references for product quality and have been quite well adopted by the software industry. The ISO 15504 and CMMi standards provide useful guidelines for process maturity assessment and improvement at the organisation level. Kitchenham and Pfleeger [23], further citing Garvin's teachings on product quality, conclude that "quality is a complex and multifaceted concept that can be described from five different perspectives":

- The *transcendental view* sees quality as something that can be recognised but hardly defined.
- The *user view* sees quality as fitness for purpose.
- The *manufacturing* view sees quality as conformance to specification.
- The *product view* attaches quality to inherent characteristics of the product.
- The *value-based* view sees quality as the interest or money users will put on it.

Even in the restricted situation of a specific software project most people will implicitly have different meanings for quality. For that reason the requirements and notion of quality need to be publicly discussed and explicitly stated. Relying on well-known definitions and concepts issued from both standards and experts in the domain greatly helps actors to reach an agreement.

[1]As an example the idea of high complexity heavily differs when considered in critical embedded systems or in desktop products

2.3 Specific benefits of data mining

Data mining offers a new perspective on the information available from software projects, thus unleashing new, powerful tools and methods for our activity. While software measurement programs allow to assess aspects of the product or project quality, software data mining programs offer enough information to build upon the assessment phase and deliver practical recommendations for the improvement of the desired attributes of quality. In the context of software development this can be achieved through e.g. action lists for refactoring, maintainability warnings for overly complex files, or notice when too many support questions lie unanswered.

Data mining methods offer a plethora of useful tools for this purpose. Outliers detection [1, 28, 31] gives precious information for artefacts or project characteristics that show a deviation from so-called normal behaviour. Clustering [20, 27] allows to classify artefacts according to multi-dimensional criteria[2]. Recommender systems [15, 7, 30] can propose well-known patterns, code snippets or detect bug patterns.

3. DEFINING THE MINING PROCESS

Considering the above-mentioned perils, we propose a few guidelines to be followed when setting up a data mining process. These allow to ensure the integrity and usability of information and keep all actors synchronised on the same concerns and solution.

3.1 Declare the intent

The whole mining process is driven by its stated goals. The quality model and attributes, means to measure it, and presentation of the results will differ if the program is designed as an audit-like, acceptance test for projects, or as an incremental quality improvement program to ease evolution and maintenance of projects. The users of the mining program, who may be developers, managers, buyers, or end-users of the product, have to map its objectives to concepts and needs they are familiar with. Including users in the definition of the intent of the program also helps preventing counter-productive use of the metrics and quality models.

The intent must be simple, clearly expressed in a few sentences, and published for all considered users of the program.

3.2 Identify quality attributes

The concerns identified in the intent are then decomposed into quality attributes. This firstly gives a structure to the quality model, and secondly allows to rely on well-defined characteristics – which greatly simplifies the communication and exchange of views. Recognised standards and established practices provide useful frameworks and definitions for this step. One should strive for simplicity when elaborating quality attributes and concepts. Common sense is a good argument, and even actors that have no knowledge of the field associated to the quality attributes should be able to understand them. Obscurity is a source of fear and distrust and must be avoided.

The output of this step is a fully-featured quality model that reflects all of the expected needs and views of the mining program. The produced quality model is also a point of convergence for all actors: requirements of different origin

[2]As an example, maintainability is often empirically estimated using control and data flow complexity.

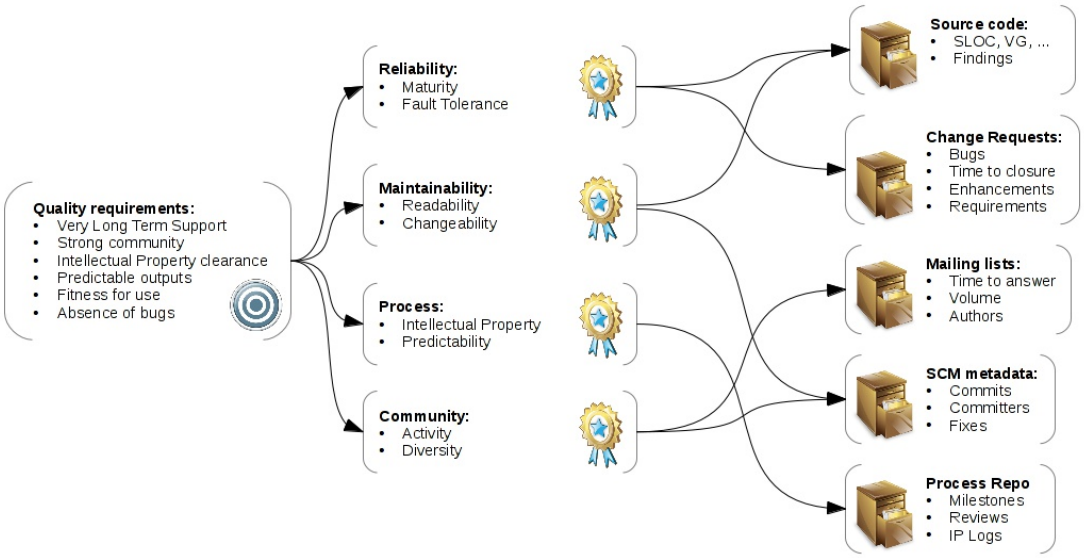


Figure 1: From quality definition to repository metrics.

and nature are bound together and form a unified, consistent view.

3.3 Identify available metrics

Once we precisely known what quality characteristics we are looking for, we have to identify measures that reflect this information need. Data retrieval is a fragile step of the mining process. Depending on the information we are looking for, various artefact types and measures may be available: one has to select them carefully according to their intended purpose. The different repositories available for the projects being analysed should be listed, with the measures that may be retrieved from them. Selected metrics have to be stable and reliable (i.e. their meaning to users must remain constant over time and usage), and their retrieval automated (i.e. no human intervention is required).

This step also defines how the metrics are aggregated up to the top quality characteristics. Since there is no universally recognised agreement on these relationships one has to rely on local understanding and conventions. All actors, or at least a vast majority of them, should agree on the meaning of the selected metrics and the links to the quality attributes.

3.4 Implementation

The mining process must be fully automated, from data retrieval to results presentation, and transparently published. Automation allows to reliably and regularly collect the information, even when people are not available[3] or not willing to do it[4]. From the historical perspective, missing data poses a serious threat to the consistency of results and to the algorithms used to analyse them – information is often easier to get at the moment than afterwards. The publishing of the entire process also helps people understand what quality is in this context and how it is measured (i.e. no magic), making them more confident in the process.

[3]Nobody has time for data retrieval before a release or during holidays.

[4]Kaner cites in [22] the interesting case of testers waiting for the release before submitting new bugs, pinning them on the wall for the time being.

3.5 Presentation of results

Visualisation of results is of primary importance for the efficiency of the information we want to transmit. In some cases (e.g. for incremental improvement of quality) a short list of action items will be enough because the human mind feels more comfortable correcting a few warnings than hundreds of them. But if our goal is to assess the technical debt of the product, it would be better to list them all to get a good idea of the actual status of the product's maintainability. Pictures and plots also show to be very useful to illustrate ideas and are sometimes worth a thousand words. If we want to highlight unstable files, a plot showing the number of recent modifications would immediately spot the most unstable of them and show how much volatile they are compared to the average.

Literate data analysis is the application of Donald Knuth's literate programming principles to the analysis of data. Reproducible research is thus made possible through tools like Sweave [13] or Knitr [35], which allow to generate custom reports from R data analysis scripts, with dynamically generated graphics and text.

4. PRACTICAL MINING

4.1 Topology of a software project

Repositories hold all of the assets and information available for a software project, from code to review reports and discussions. In the mining process it is necessary to find a common base of mandatory repositories and measures for all projects to be analysed: e.g. configuration management, change management (bugs and change requests), communication (e.g. forums or mailing lists) and publication (web-

site, wiki). We list thereafter some common repositories and how they can be used.

It is useful to retrieve information on different time frames (e.g. yesterday, last week, last month, last three months) to better grasp the dynamics of the measures. Evolution is an important aspect of the analysis since it allows users to understand the link between measures, practices and quality attributes. It also makes them realise how better or worse they do with time.

4.2 Source code

Source code is generally extracted from the configuration management repository at a specific date and for a specific version (or branch) of the product. Source releases as published by open source projects can be used as well but may heavily differ from direct extracts, because they represent a subset of the full project repository and may have been modified during the release process.

Static code analysis [24] tools usually provide *metrics* and *findings*. *Measures* target intrinsic characteristics of the code, like its size or the number of nested loops. *Findings* are occurrences of non-conformities to some standard rules or good practices. Tools like Checkstyle [8] or PMD [10] check for anti-patterns or violations of conventions, and thus provide valuable information on acquired development practices [33]. Dynamic analysis gives more information on the product performance and behaviour [11, 32], but needs compiled, executable products – which is difficult to achieve automatically on a large scale. Dynamic and static analyses are complementary techniques on completeness, scope, and precision [4].

4.3 Configuration management

A configuration management system allows to record and restitute all changes on versioned artefacts, with some useful information about who did it, when, and (to some extent) why. It brings useful information on artefacts' successive modifications and on the activity and diversity of actors in the project.

The primary source of data is the verbose log of the repository branch or trunk, as provided in various formats by all configuration management tools. The interpretation of metrics heavily depends on the configuration management tool in use and its associated workflow: as an example, commits on a centralised subversion repository do not have the same meaning than on a Git distributed repository because of the branching system and philosophy. A Subversion repository with hundreds of branches is probably the sign of a bad usage of the branching system, while it can be considered normal with Git. In such a context it may be useful to setup some scales to adapt ranges and compare tools together. The positioning of the analysis in the configuration management branches also influences its meaning: working on maintenance-only branches (i.e. with many bug fixes and few new features) or on the trunk (next release, with mainly new features and potentially large refactoring or big-bang changes) does not yield the same results.

Example of metrics that can be gathered from there are the number of commits, committers or fix-related commits on a given period of time. As a note, we defined in our context a fix-related commit as having one of the terms *fix*, *bug*, *issue* or *error* in the message associated to the revision. Depending on the project tooling and conventions, more sophisticated methods may either be set up to establish the link between bugs and commits [2].

4.4 Change management

A tracker, or bug tracking system, allows to record any type of items with a defined set of associated attributes. They are typically used to track bugs and enhancements requests, but may as well be used for requirements or support requests. The comments posted on issues offer a bunch of useful information regarding the bug itself and people's behaviour; some studies even treat them as a communication channel.

A bias may be introduced by different tools and workflows, or even different interpretations of a same status. A common work-around is to map actual states to a minimalist set of useful canonical states: e.g. open, working, verifying, closed. Almost all life cycles can be mapped to these simple steps without twisting them too much.

Examples of metrics that can be retrieved from change management systems include the time to closure, number of comments, or votes, depending on the system's features.

4.5 Communication channels

Actors of a software project need to communicate to coordinate efforts, ensure some consistency in coding practices, or simply get help [29]. This communication may flow through different means: mailing lists, forums, or news servers. Every project is supposed to have at least two communication media, one for contributors to exchange on the product development (the developers mailing list) and another one for the product usage (the users mailing list).

Local customs may impact the analysis of communication channels. In some cases low-activity projects send commits or bugs notification to the developer mailing list as a convenience to follow repositories activity, or contributors may use different email addresses or logins when posting – which makes it difficult, if not impossible, to link their communications to activity in other repositories. The different communication media can all be parsed to retrieve a common set of metrics like the number of threads, mails, authors or response ratio, and time data like the median time to first response.

4.6 Publication

A project has to make the final product available to its users along with a number of artefacts such as documentation, FAQ or How-to's, or project-related information like team members, history of the project or advancement of the next release. A user-edited wiki is a more sophisticated communication channel often used by open source projects. The analysis of these repositories may be difficult because of the variety of publishing engines available: some of them display the time of last modification and who did it, while other projects use static pages with almost no associated meta data. As an example user-edited wikis are quite easy to parse because they display a whole bouquet of information about their history, but may be very time-consuming to parse because of their intrinsic changeability.

Metrics that can be gathered from these repositories are commonly linked to the product documentation and community wealth characteristics. Examples include the number of pages, recent modifications or entries in the FAQ, the number of different authors and readers, or the age of entries

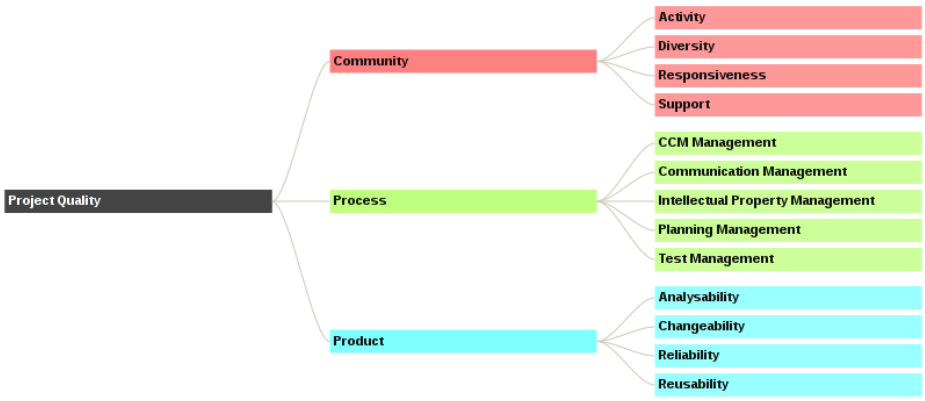


Figure 2: Proposed Eclipse quality model

(which may denote obsolete pages).

5. THE ECLIPSE CASE

5.1 Context of the quality assessment program

The Eclipse foundation is an independent, non-profit corporation founded in 2004 to support the development of the Eclipse IDE. It is one of the most well-known open source success stories around, gathering a "vibrant, vendor-neutral and transparent community" of both open source developers and industry actors. As a result Eclipse is the most used IDE in the world with tens of millions of downloads per year.

The Polarsys working group was launched in 2011 to foster embedded systems development and ecosystem. Its goals are to provide mature tools, with long-term support (as long as 75 years in space or aeronautics) and compliance with safety-critical certification processes. The Maturity Assessment task force was initiated in the beginning of 2013 to address quality concerns and provide some guarantees on the components embedded in the Polarsys stack of software.

5.2 Declaration of intent

The objectives of the conducted program have been identified as follows:

- *Assess projects maturity* as defined by the Polarsys working group stated goals. Is the project ready for large-scale deployments in high-constraints software organisations? Does it conform to the Eclipse foundation recommendations?
- *Help teams develope better software* regarding the quality requirements defined. Propose guidance to improve the management, process and product of projects.
- *Establish the foundation for a global agreement on quality* conforming to the Eclipse way. A framework to collaborate on the semantics of quality is the first step to a better understanding and awareness of these concerns in the Eclipse community.

5.3 Quality requirements

Because of their open-source nature, Eclipse components have strong maintainability concerns, which are actually reenforced for the Polarsys long-term support requirements. In the spirit of the ISO/IEC 9126 standard[18] these concerns are decomposed in **Analysability**, **Changeability** and **Reusability**. Another quality requirement is **Reliability**: Eclipse components are meant to be used in bundles or stacks of software, and the failure of a single component may threaten the whole application. For an industrial deployment over thousands of people in worldwide locations this may have serious repercussions.

The Eclipse foundation has a strong interest in IP management and predictability of outputs. Projects are classified into phases, from proposal (defining the project) to incubating (growing the project) and mature (ensuring project maintenance and vitality). Different constraints are imposed on each phase: e.g. setting up reviews and milestones, publishing a roadmap, or documenting APIs backward compatibility. In the first iteration of the quality model only **IP management** and **Planning management** are addressed.

Communities really lie at the heart of the Eclipse way. The Eclipse manifesto defines three communities: developers, adopters, and users. Projects must constitute, then grow and nurture their communities regarding their **activity**, **diversity**, **responsiveness** and **support** capability. In the context of our program we will mainly focus on developer and user communities.

5.4 Metrics identification

Source code.

Source code is extracted from Subversion or Git repositories and analysed with SQuORE [3]. Static analysis was preferred because automatically compiling the different projects was unsafe and unreliable. Further analyses may be integrated with continuous build servers to run dynamic analyses as well.

Metrics include common established measures like line-counting (comment lines, source lines, effective lines, statements), control-flow complexity (cyclomatic complexity, maximum level of nesting, number of paths, number of control-flow tokens) or Halstead's software science metrics (number of distinct and total operators and operands).

Findings include violations from SQuORE, Checkstyle and PMD. All of these have established lists of rules that are mapped to practices and classified according to their impact (e.g. reliability, readability, fault tolerance).

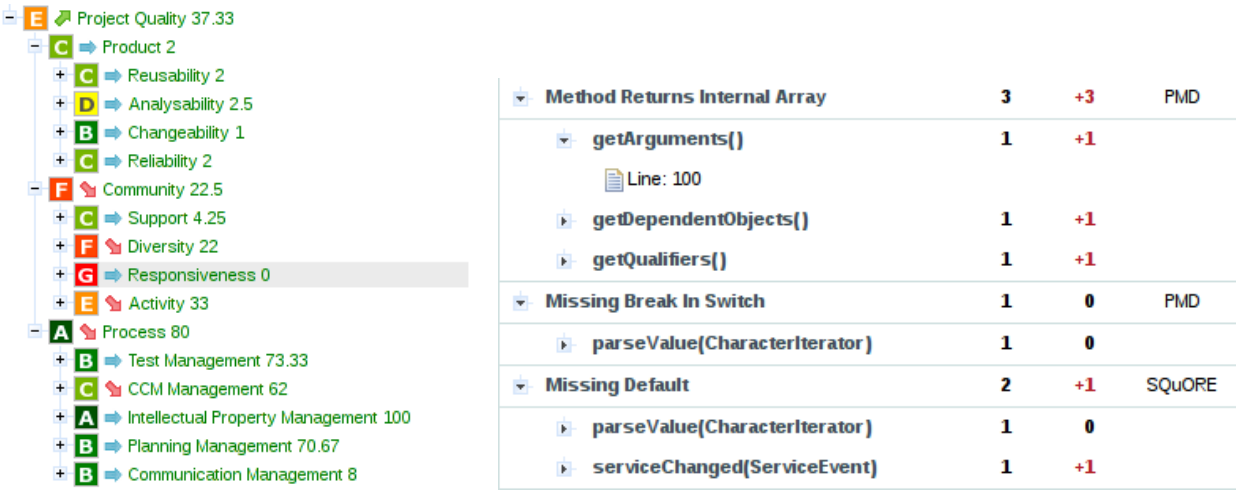


Figure 3: Examples of results for the Polarsys mining program.

SCM metadata.

A data provider was developed to retrieve software configuration metadata from the two software configuration management systems used inside Eclipse, Subversion and Git. The metrics identified are the following: number of commits, committers, committed files, and fix-related commits. All measures are computed for the last week, last month and last three months as well as in total. Doing so we get the ability to catch recent evolutions of the project by giving more weight to the most recent modifications.

Communication.

Communication channels used at Eclipse are NNTP news, mailing lists and web forums. All of these can be mapped and converted to the mbox format, which we then parse to extract the metrics we need. Metrics retrieved are the number of threads, distinct authors, the response ratio and median time to first answer. Metrics are gathered for both user and developer communities, on different time frames to better grasp the dynamics of the project evolution (one week, one month, three months).

Process repository.

The Eclipse foundation has recently started an initiative to centralise and publish real-time process-related information. This information is available through a public API returning JSON and XML data about every Eclipse component. Metrics defined are: number and status of milestones and reviews, and coverage of intellectual property logs.

5.5 Results

Following the quality requirements established in step 3, we proposed and documented a quality model to the working group. People could better visualise the decomposition of quality and relationships between quality attributes and metrics. This led to new ideas and constructive feedback and after some minor changes and improvements the model represented in figure 2 was officially published on the working group wiki.

The prototype for this first iteration of the quality program has been implemented with SQuORE[3] and provides several means to deliver the information:

- A *quality tree* that lists the quality attributes and shows the conformance to identified requirements (figure 3, left side). The evolution compared to the last analysis is also depicted.
- *Action lists*, classified according to the different concerns identified: overly complex or untestable files, naming conventions violations, or a refactoring wish list. An example is shown on the right of figure 3.
- *Nice, coloured graphics* that immediately illustrate specific concepts. This proves to be especially useful when the characteristics of an incriminated file make it significantly different than the average so it can be identified at first sight, as e.g. for highly complex files.

Finally the goals, definitions, metrics and resulting quality model were presented at the EclipseCon France conference held in May, 2013 in Toulouse and received good and constructive feedback.

6. CONCLUSION

In this study we propose an integrated approach to software project management, assessment and improvement through data mining. Building upon measurement programs experience and good practices, software data mining unveils new horizons of information, along with new methods and tools to extract useful knowledge from it.

The proposed approach focuses on a few key concepts that aim to gather all actors around a common understanding of the program's objectives to produce a consensual evaluation of the project characteristics and pragmatic recommendations for its improvement. When software engineering fails to bring a unanimously accepted definition of some concepts we rely on local agreement and understanding to institute specific conventions. Successive iterations of the procedure and quality model will improve upon the experience and feedback of users to correct identified issues, address new concerns, and provide new recommendation types.

We also detail the information available in common software repositories, give practical advice as how to extract it,

and list some of the pitfalls that practitioners may encounter during data retrieval. The application of the proposed approach to a large-scale mining program with industrial participants illustrates how we could get to a common local agreement on quality definition and improvement and produce a fully-featured prototype.

Forges are great environments for data mining programs, because they propose a restricted set of tools and ease the integration of information across them. The development of collaborative frameworks such as the Open Services for Lifecycle Collaboration (OSLC) [6] will also lead to tighter integration of information coming from different sources. Working with real-world data and situations also makes a great difference. We encourage future research work to setup software projects mining in production environments to ensure their usability and relevance to pragmatic concerns. Open source software forges have a wealth of information available that can be transformed into useful knowledge, and numerous studies from the data mining field could benefit from this application, increasing their impact on software engineering knowledge and practices.

7. REFERENCES

[1] M. Al-Zoubi. An effective clustering-based approach for outlier detection. *European Journal of Scientific Research*, 2009.

[2] A. Bachmann, C. Bird, F. Rahman, P. Devanbu, and A. Bernstein. The Missing Links: Bugs and Bug-fix Commits. 2010.

[3] B. Baldassari. SQuORE: A new approach to software project quality measurement. In *International Conference on Software & Systems Engineering and their Applications*, 2012.

[4] T. Ball. The Concept of Dynamic Analysis. In *Software Engineering—ESEC/FSE'99*, pages 216–234, 1999.

[5] V. R. Basili, G. Caldiera, and H. D. Rombach. The Goal Question Metric approach, 1994.

[6] O. Berger, S. Labbene, M. Dhar, and C. Bac. Introducing OSLC, an open standard for interoperability of open source development tools. In *International Conference on Software & Systems Engineering and their Applications*, pages 1–8, 2011.

[7] M. Bruch and M. Mezini. Improving Code Recommender Systems using Boolean Factor Analysis and Graphical Models. 2010.

[8] O. Burn. Checkstyle, 2001.

[9] P. B. Crosby. *Quality is free: The art of making quality certain*, volume 94. McGraw-Hill New York, 1979.

[10] D. Dixon-Peugh. PMD, 2003.

[11] T. Eisenbarth. Aiding program comprehension by static and dynamic feature analysis. In *International Conference on Software Maintenance*, 2001.

[12] N. Fenton. Software Measurement: a Necessary Scientific Basis. *IEEE Transactions on Software Engineering*, 20(3):199–206, Mar. 1994.

[13] Friedrich Leisch. Sweave. Dynamic generation of statistical reports using literate data analysis. Technical Report 69, SFB Adaptive Information Systems and Modelling in Economics and Management Science, WU Vienna University of Economics and Business, Vienna, 2002.

[14] A. Gopal, M. Krishnan, T. Mukhopadhyay, and D. Goldenson. Measurement programs in software development: determinants of success. *IEEE Transactions on Software Engineering*, 28(9):863–875, Sept. 2002.

[15] H.-J. Happel and W. Maalej. Potentials and challenges of recommendation systems for software development. In *Proceedings of the 2008 international workshop on Recommendation systems for software engineering*, pages 11–15. ACM, 2008.

[16] H. Hemmati, S. Nadi, O. Baysal, O. Kononenko, W. Wang, R. Holmes, and M. W. Godfrey. The MSR Cookbook. In *10th International Workshop on Mining Software Repositories*, pages 343–352, 2013.

[17] H. Ircdenkscn, I. Vcj, and J. Iversen. Implementing Software Metrics Programs: A Survey of Lessons and Approaches. *Information Technology and Organizations: Trends, Issues, Challenges and Solutions*, 1:197–201, 2003.

[18] ISO. ISO/IEC 9126 Software Engineering - Product Quality - Parts 1-4. Technical report, ISO/IEC, 2005.

[19] J. Iversen and L. Mathiassen. Lessons from implementing a software metrics program. In *Proceedings of the 33rd Annual Hawaii International Conference on System Sciences*, pages 1–11. IEEE, 2000.

[20] G. S. D. S. Jayakumar and B. J. Thomas. A New Procedure of Clustering Based on Multivariate Outlier Detection. *Journal of Data Science*, 11:69–84, 2013.

[21] J. M. Juran, A. B. Godfrey, R. E. Hoogstoel, and E. G. Schilling. *Juran's Quality Handbook*, volume 2. McGraw Hill New York, 1999.

[22] C. Kaner and W. P. Bond. Software engineering metrics: What do they measure and how do we know? In *10th International Software Metrics Symposium, METRICS 2004*, pages 1–12, 2004.

[23] B. Kitchenham and S. Pfleeger. Software quality: the elusive target. *IEEE Software*, 13(1):12–21, 1996.

[24] P. Louridas. Static Code Analysis. *IEEE Software*, 23(4):58–61, 2006.

[25] M. R. Marri, S. Thummalapenta, and T. Xie. Improving Software Quality via Code Searching and Mining. *Source*, pages 33–36, 2009.

[26] T. Menzies, C. Bird, and T. Zimmermann. The inductive software engineering manifesto: Principles for industrial data mining. In *Proceedings of the International Workshop on Machine Learning Technologies in Software Engineering*, Lawrence, Kansas, USA, 2011.

[27] B. B. Naib. An Improved Clustering Approach for Software Quality Analysis. *International Journal of Engineering, Applied and Management Sciences Pradigms*, 05(01):96–100, 2013.

[28] M. S. D. Pachgade and M. S. S. Dhande. Outlier Detection over Data Set Using Cluster-Based and Distance-Based Approach. *International Journal of Advanced Research in Computer Science and Software Engineering*, 2(6):12–16, 2012.

[29] E. Raymond. The cathedral and the bazaar. *Knowledge, Technology & Policy*, 1999.

[30] M. Robillard and R. Walker. Recommendation

systems for software engineering. *IEEE Software*, 27(4):80–86, 2010.

[31] G. Singh and V. Kumar. An Efficient Clustering and Distance Based Approach for Outlier Detection. *International Journal of Computer Trends and Technology*, 4(7):2067–2072, 2013.

[32] M.-A. Storey. Theories, methods and tools in program comprehension: Past, present and future. In *13th International Workshop on Program Comprehension (IWPC 2005)*. IEEE Computer Society, 2005.

[33] S. Wagner, F. Deissenboeck, M. Aichner, J. Wimmer, and M. Schwalb. An Evaluation of Two Bug Pattern Tools for Java. In *2008 International Conference on Software Testing, Verification, and Validation*, pages 248–257. Ieee, Apr. 2008.

[34] L. Westfall and C. Road. 12 Steps to Useful Software Metrics. *Proceedings of the Seventeenth Annual Pacific Northwest Software Quality Conference*, 57 Suppl 1(May 2006):S40–3, 2005.

[35] Y. Xie. Knitr: A general-purpose package for dynamic report generation in R. Technical report, 2013.

[36] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, 31(6):429–445, 2005.

Understanding software evolution: The Maisqual Ant data set

Boris Baldassari
SQuORING Technologies
Toulouse, France
Email: boris.baldassari@squoring.com

Philippe Preux
SequeL, LIFL, CNRS, INRIA
Université de Lille
Email: philippe.preux@inria.fr

***Abstract*—Software engineering is a maturing discipline which has seen many drastic advances in the last years. However, some studies still point to the lack of rigorous and mathematically grounded methods to raise the field to a new emerging science, with proper and reproducible foundations to build upon. Indeed, mathematicians and statisticians do not necessarily have software engineering knowledge, while software engineers and practitioners do not necessarily have a mathematical background.**

The Maisqual research project intends to fill the gap between both fields by proposing a controlled and peer-reviewed data set series ready to use and study. These data sets feature metrics from different repositories, from source code to mail activity and configuration management meta data. Metrics are described and commented, and all steps followed for their extraction and treatment are described with contextual information about the data and its meaning.

This article introduces the Apache Ant weekly data set, featuring 636 extracts of the project over 12 years at different levels of artefacts – application, files, functions. By associating community and process related information to code extracts, this data set unveils interesting perspectives on the evolution of one of the great success stories of open source.

***Index Terms*—Data mining Software Engineering Software Metrics**

I. Introduction

In the last 30 years, software has become ubiquitous both in the industry (from Internet to business intelligence to supply chain automation) and in our daily lives. As a consequence, characteristics of software such as reliability, performance or maintainability have become increasingly important – either for stakeholders, developers, or end-users. Research on software metrics was introduced a long time ago to help and support the field, but despite some remarkable advances there are still many critics (most notably from Fenton [1] and Kaner [2]) as to the scientific approach and overall mathematical rigour needed to build scientific methods.

Most of the studies that cover software engineering concerns use their own retrieval process and work on unpublished, non-verifiable data. This fact definitely influences the credibility of studies, and lends credence to criticism about the relevance, reproducibility and usability of their conclusions. The proposed data set intends to establish a bedrock for upcoming studies by providing a consistent and peer-reviewed set of measures associated to various and unusual characteristics extracted from heterogeneous sources: mails, configuration management and coding rules. It provides a set of metrics gathered on a long-running, real-world project, and states their definitions and requirements. It is designed to be easily imported with any statistical program.

In section II, we describe the structure and contents of the data set, and give information on the project history. Section III enumerates the various types of measures retrieved and how they are integrated, and section IV lists the coding rules checked on code. Section V proposes a few examples of usage for the data set. Finally, we state our on-going and future work regarding these concerns in section VI.

II. Data set description

A. The Ant project

The early history of Ant begins in the late nineties with the donation of the Tomcat software from Sun to Apache. From a specific build tool, it evolved steadily through Tomcat contributions to be more generic and usable. James Duncan Davidson announced the creation of the Ant project on the 13 January 2000, with its own mailing lists, source repository and issue tracking.

TABLE I: Major releases of Ant.

Date	Version	SLOC	Files	Functions
2000-07-18	1.1	9671	87	876
2000-10-24	1.2	18864	171	1809
2001-03-02	1.3	33347	385	3332
2001-09-03	1.4	43599	425	4277
2002-07-15	1.5	72315	716	6782
2003-12-18	1.6	97925	906	9453
2006-12-19	1.7	115973	1113	12036
2010-02-08	1.8	126230	1173	12964

There have been many versions since then: 8 major releases and 15 updates (minor releases). The data set ends in July 2012, and the last version officially released at that time is 1.8.4. Table I lists major releases of Ant with some characteristics of official builds as published. It should be noted that these characteristics may show inconsistencies

with the data set, since the build process extracts and transforms a subset of the actual repository content.

Ant is arguably one of the most relevant examples of a successful open source project: from 2000 to 2003, the project attracted more than 30 developers whose efforts contributed to nominations for awards and to its recognition as a reliable, extendable and well-supported build standard for both the industry and the open source community.

An interesting aspect of the Ant project is the amount of information available on the lifespan of a project: from its early beginnings in 2000, activity had its climax around 2002-2003 and then decreased steadily. Although the project is actively maintained and still brings regular releases the list of new features is decreasing with the years. It is still hosted by the Apache Foundation, which is known to have a high interest in software product and process quality.

B. Structure of the data set

The complete data set is a consistent mix of different levels of information corresponding to the application, files and functions artefact types. Hence three different subsets are provided. The **application** data set has 159 measures composed of 66 metrics and 93 rules extracted from source code, configuration management and communication channels. Each record is an application version. The **files** data set has 123 measures composed of 30 metrics and 93 rules extracted from source code and configuration management. Each record is a Java file with the .java extension. The **functions** data set has 117 measures composed of 24 metrics and 93 rules extracted from source code only, of which each record is a java function with its arguments.

TABLE II: Sizing information for the CSV data sets.

	App	File	Func
Size of flat files	312KB	232MB	2.4GB
Size of compressed files	68KB	12MB	89MB
Number of records	636	654 696	6 887 473

Each data set is composed of 636 exports of the Ant project, extracted on the Monday of every week since the beginning of the project until end of July, 2012. The format of the files is plain text CSV and the separator used for all data sets is `!` (exclamation mark). Some key sizing information is provided in table II.

C. Retrieval process

Data is retrieved from the project's official repositories:

- Source code is extracted from the Subversion repository's trunk at specific dates. Only files with a `.java` extension have been analysed. *Code metrics* are computed using SQuORE [3], and *rules violations* are extracted from SQuORE, Checkstyle [4], [5] and PMD [6], [7].
- Configuration management metadata is extracted from Subversion's `svn log -v` command and parsed with custom scripts.
- Communication measures are computed from the mailing lists' archives in `mbox` format.

To ensure consistency between all artefact measures we rely on SQuORE, a professional tool for software project quality evaluation and business intelligence [3]. It features a parser, which builds a tree of artefacts (application, files, functions) and an engine that associates measures to each node and aggregates data to upper levels. Users can write custom parsers to analyse specific types of data sources, which we did for the analysis of configuration management and communication channels.

III. Metrics

The measures presented here are intended as real-world data: although they have been cross-checked for errors or inconsistencies, no transformation has been applied on values. As an example, the evolution of line counting metrics shows a huge peak around the beginning of 2002 due to some configuration management large-scale actions which impacted many metrics. We deliberately kept raw data because this *was* actually the state of the subversion repository at that time.

Migrations between tools often make it difficult to rely on a continuous measure of the characteristics. Some information of the former repository may not have been included or wrongly migrated, and the meaning of metadata may have heavily differed depending on the tool. An example of such issues lies in erroneous dates of migrated code in the new configuration management system.

Another point is there may be a huge difference between the source code of an official release and the actual configuration management state at the release time. The build and release process often extracts and packages source code, skipping some files and delivering other (potentially dynamically generated) information and artefacts. As an example, the common metrics shown in table I for Ant official releases cannot be confirmed by the repository information available in the data set.

The set of metrics for each artefact type is shown in tables III, IV and V. Some metrics are available only at specific levels – e.g. distinct number of operands in a function, while others are available on more than one level – e.g. SCM Commits. Please note that the relationship between levels varies: summing line counts on files gives the line count at the application level, which is not true for commits – since a commit often includes several files. Practioners should check ecological inference[8] to reduce bias when playing with the different levels of information.

A. Source code

Most source code measures are borrowed from the literature: artefact- and line-counting metrics have their usual

definition[1], VG is from McCabe [9], DOPD, DOPT, TOPD, TOPT are from Halstead [10]. LADD, LMOD and LREM are differential measures that respectively count the number of lines added, modified and removed since last analysis.

Some of the metrics considered have computational relationships among themselves. They are:

$$\begin{aligned} \text{SLOC} &= \text{ELOC} + \text{BRAC} \\ \text{LC} &= \text{SLOC} + \text{BLAN} + \text{CLOC} - \text{MLOC} \\ \text{LC} &= (\text{ELOC} + \text{BRAC}) + \text{BLAN} + \text{CLOC} - \text{MLOC} \\ \text{COMR} &= ((\text{CLOC} + \text{MLOC}) \times 100)/(\text{ELOC} + \text{CLOC})) \end{aligned}$$

TABLE III: Source code metrics.

Metric name	Mnemo	App	File	Func
Blank lines	BLAN	X	X	X
Braces lines	BRAC	X	X	X
Control flow tokens	CFT	X	X	X
Number of classes	CLAS	X	X	X
Comment lines of code	CLOC	X	X	X
Comment rate	COMR	X	X	X
Depth of Inheritance Tree	DITM	X		
Distinct operands	DOPD			X
Distinct operators	DOPT			X
Effective lines of code	ELOC	X	X	X
Number of files	FILE	X		
Number of functions	FUNC	X	X	
Lines added	LADD	X	X	X
Line count	LC	X	X	X
Lines modified	LMOD	X	X	X
Lines removed	LREM	X	X	X
Mixed lines of code	MLOC	X	X	X
Non Conformities	NCC	X	X	X
Maximum nesting	NEST			X
Number of parameters	NOP			X
Number of paths	NPAT			X
Acquired practices	ROKR	X	X	X
Source lines of code	SLOC	X	X	X
Number of statements	STAT	X	X	X
Number of operands	TOPD			X
Number of operators	TOPT			X
Cyclomatic number	VG	X	X	X

B. Configuration management

All modern configuration management tools propose a log retrieval facility to dig into a file history. In order to work on several different projects, we needed to go one step further and define a limited set of basic information that we could extract from all major tools (e.g. CVS, Subversion, Git): number of commits, committers, and files committed. The SCM_FIXES measure counts the number of commits that have one of *fix*, *issue*, *problem* or *error* keywords in their associated commit message.

Measures are proposed in four time frames, by counting events that occured during last week (SCM_*_1W), during last month (SCM_*_1M), during last three months (SCM_*_3M), and since the beginning of the project (SCM_*_TOTAL). These enable users to better grasp recent variations in the measures, and give different perspectives on its evolution. Configuration management metrics are listed in table IV.

[1]A complete definition of the metrics is available on the Maisqual web site: maisqual.squoring.com/wiki/index.php/Data Sets.

TABLE IV: SCM metrics.

Metric name	Mnemo	App	File	Func
SCM Fixes	SCM_FIXES	X	X	
SCM Commits	SCM_COMMITS	X	X	
SCM Committers	SCM_COMMITTERS	X	X	
SCM Committed files	SCM_COMMIT_FILES	X		

C. Communication channels

Open-source projects usually have at least two mailing lists: one for technical questions about the product's development itself (i.e. the developer mailing list) and another one for questions relative to the product's usage (i.e. the user mailing list). Historical data was extracted from old mbox archives, all of which are available on the project web site.

TABLE V: Communication metrics.

Metric name	App	File	Func
Number of authors in developer ML	X		
Median response time in developer ML	X		
Volume of mails in developer ML	X		
Number of threads in developer ML	X		
Number of authors in user ML	X		
Median response time in user ML	X		
Volume of mails in user ML	X		
Number of threads in user ML	X		

Available measures are the number of distinct authors, the volume of mails exchanged on the mailing list, the number of different threads, and the median time between a question and its first answer on the considered time frame. These measures are proposed in three time frames spanning on last week (COM_*_1W), last month (COM_*_1M) and the last three months (COM_*_3M) of activity. Communication metrics are listed in table V.

IV. Rules

Rules are associated to coding conventions and practices. They deliver substantial information on the local customs in use in the project, and are usually linked to specific characteristics of quality. In the data sets, conformity to rules is displayed as a number of violations of the rule (non-conformity count or NCC) for the given artefact.

Many of these rules are linked to coding conventions published by standardisation organisms like CERT (Carnegie Mellon's secure coding instance), which usually give a ranking on the remediation cost and the severity of the rule. There are also language-specific coding conventions, as is the case with Sun's coding conventions for the Java programming language [11].

A. SQuORE rules

We identified 21 rules from SQuORE 2013-C, targeting the most common and harmful coding errors. Examples of checked rules include fall-through in switch cases, missing default, backwards goto, and assignment in condition. The following families of rules are defined: fault tolerance (2 rules), analysability (7 rules), maturity (1 rule), stability (10 rules), changeability (12 rules) and testability (13 rules). The full rule set is described on the Maisqual project wiki[2].

B. Checkstyle rules

We identified 39 rules from the Checkstyle 5.6 rule set, corresponding to useful practices generally well adopted by the community. The quality attributes impacted by these rules are: analysability (23 rules), reusability (11 rules), reliability (5 rules), efficiency (5 rules), testability (3 rules), robustness (2 rules) and portability (1 rule). All rules are described on the Checkstyle web site[3].

C. PMD rules

We selected 58 rules from the PMD 5.0.5 rule set. These are related to the following quality attributes: analysability (26 rules), maturity (31 rules), testability (13 rules), changeability (5 rules), and efficiency (5 rules). The full rule set is documented on the PMD web site[4].

V. Possible uses of the data set

The introduction of data mining techniques in software engineering is quite recent, and there is still a vast field of possibilities to explore. Data may be analysed from an evolutional or static perspective by considering either a time range or a single version. Since the different levels of data (application, file and function) are altogether consistent, one may as well consider studying relationships between them, or even the evolution of these relationships with time.

Communication and configuration management metrics give precious insights into the community's activity and process-related behaviour in development. Time-related measures (LADD, LMOD, LREM, *_1W, *_1M, *_3M) are useful to grasp the dynamics of the project's evolution. Since rule violations are representative of coding practices, one may consider the links between the development practices and their impact on attributes of software.

The Maisqual project itself investigates the application of statistical techniques to software engineering data and relies on this data set series. Examples of usages are provided on the Maisqual web site, including evolution of metrics with time (e.g. time series analysis), analysis of coding rule violations, and basic exploration of a single version of the project (e.g. clustering of artefacts, principal components analysis).

[2]See http://maisqual.squoring.com/wiki/index.php/Rules.
[3]See http://checkstyle.sourceforge.net/config.html.
[4]See http://pmd.sourceforge.net/pmd-5.0.5/rules/.

We recommend the use of literate analysis tools like Sweave [12] and Knitr [13], which allow practitioners to embed R code chunks into LaTeX documents. These dynamic documents can then be safely applied to many sets of data with a similar structure to easily reproduce results of an analysis on a large amount of data. Another added value of literate data analysis is to situate results in a semantic context, thus helping practitioners and end-users to understand both the computation and its results.

VI. Summary and future work

The contribution of the data set presented here is twofold: firstly the long time range (12 years) spans from the very beginning of the project to an apogee of activity and to a stable state of maturity. Secondly, the introduction of unusal metrics (rule violations, configuration management, mailing lists) at different levels opens new perspectives on the evolution of software, the dynamics of its community, the coding and configuration management practices.

Next versions of this data set will include new metrics, gathered on new sources (e.g. bug tracking system), with new data types (e.g. boolean, categorical) to foster usage of different algorithms working on non-numerical data. New projects will also be added from the open-source community (GCC, JMeter).

References

[1] N. Fenton, "Software Measurement: a Necessary Scientific Basis," *IEEE Transactions on Software Engineering*, vol. 20, no. 3, pp. 199–206, Mar. 1994.

[2] C. Kaner and W. P. Bond, "Software engineering metrics: What do they measure and how do we know?" in *10th International Software Metrics Symposium, METRICS 2004*, 2004, pp. 1–12.

[3] B. Baldassari, "SQuORE: A new approach to software project quality measurement," in *International Conference on Software & Systems Engineering and their Applications*, Paris, France, 2012.

[4] P. Louridas, "Static Code Analysis," *IEEE Software*, vol. 23, no. 4, pp. 58–61, 2006.

[5] O. Burn, "Checkstyle," 2001. [Online]. Available: http://checkstyle.sf.net

[6] N. Ayewah, W. Pugh, D. Hovemeyer, J. D. Morgenthaler, and J. Penix, "Using static analysis to find bugs," *IEEE Software*, vol. 25, no. 5, pp. 22–29, 2008.

[7] D. Dixon-Peugh, "PMD," 2003. [Online]. Available: http://pmd.sf.net

[8] D. Posnett, V. Filkov, and P. Devanbu, "Ecological inference in empirical software engineering," in *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, Nov. 2011, pp. 362–371.

[9] T. McCabe, "A complexity measure," *Software Engineering, IEEE Transactions on*, no. 4, pp. 308–320, 1976.

[10] M. H. Halstead., *Elements of Software Science*. Elsevier Science Inc., 1977.

[11] Sun, "Code Conventions for the Java Programming Language," Tech. Rep., 1999.

[12] Friedrich Leisch, "Sweave. Dynamic generation of statistical reports using literate data analysis." SFB Adaptive Information Systems and Modelling in Economics and Management Science, WU Vienna University of Economics and Business, Vienna, Tech. Rep. 69, 2002.

[13] Y. Xie, "Knitr: A general-purpose package for dynamic report generation in R," Tech. Rep., 2013.

Appendix B

Data sets

B.1 Apache Ant

The following table shows the releases considered for the Apache Ant project, and their date of publication.

Date	Version	SLOC	Files	Functions	
1.1	2000-07-18	1.5.4	2003-08-12	1.7.1	2008-06-27
1.2	2000-10-24	1.6.0	2003-12-18	1.8.0	2010-02-08
1.3	2001-03-02	1.6.1	2004-02-12	1.8.1	2010-05-07
1.4	2001-09-03	1.6.2	2004-07-16	1.8.2	2020-12-27
1.5.0	2002-07-15	1.6.3	2005-04-28	1.8.3	2012-02-29
1.5.1	2002-10-03	1.6.4	2005-05-19	1.8.4	2012-05-23
1.5.2	2003-03-03	1.6.5	2005-06-02		
1.5.3	2003-04-09	1.7.0	2006-12-19		

B.2 Apache httpd

The following table shows the releases considered for the Apache httpd project, and their date of publication.

Version	Date	Version	Date	Version	Date
2.0.35	2002-04-05	2.0.46	2003-05-28	2.0.54	2005-04-11
2.0.36	2002-05-01	2.0.47	2003-07-09	2.0.55	2005-10-10
2.0.39	2002-06-18	2.0.48	2003-10-24	2.0.58	2006-04-24
2.0.40	2002-08-09	2.0.49	2004-03-18	2.0.59	2006-07-27
2.0.42	2002-09-19	2.0.50	2004-06-29	2.0.61	2007-09-04
2.0.43	2002-10-03	2.0.51	2004-09-15	2.0.63	2008-12-26
2.0.44	2003-01-18	2.0.52	2004-09-23	2.0.64	2010-10-14
2.0.45	2003-03-31	2.0.53	2005-02-05	2.0.65	2013-06-28

Version	Date	Version	Date	Version	Date
2.2.0	2005-11-29	2.2.10	2008-10-07	2.2.17	2010-10-14
2.2.2	2006-04-22	2.2.11	2008-12-06	2.2.18	2011-05-08
2.2.3	2006-07-27	2.2.12	2009-07-2	2.2.19	2011-05-20
2.2.4	2007-01-06	2.2.13	2009-08-06	2.2.20	2011-08-30
2.2.6	2007-09-04	2.2.14	2009-09-24	2.2.21	2011-09-09
2.2.8	2008-01-10	2.2.15	2010-03-02	2.2.22	2012-01-25
2.2.9	2008-06-10	2.2.16	2010-07-21		

B.3 Apache JMeter

The following table shows the releases considered for the Apache JMeter project, and their date of publication.

Version	Date	Version	Date
1.8.1	2003-02-14	2.3.2	2008-06-10
1.9.1	2003-08-07	2.3.3	2009-05-24
2.0.0	2004-04-04	2.3.4	2009-06-21
2.0.1	2004-05-20	2.4	2010-07-14
2.0.2	2004-11-07	2.5	2011-08-17
2.0.3	2005-03-22	2.5.1	2011-10-03
2.1.0	2005-08-19	2.6	2012-02-01
2.1.1	2005-10-02	2.7	2012-05-27
2.2	2006-06-13	2.8	2012-10-06
2.3	2007-07-10	2.9	2013-01-28
2.3.1	2007-11-28	2.10	2013-10-21

B.4 Apache Subversion

The following table shows the releases considered for the Apache Subversion project, and their date of publication.

Version	Date	Version	Date	Version	Date
1.0.0	2004-02-23	1.4.4	2007-05-29	1.6.16	2011-03-03
1.0.1	2004-03-12	1.4.5	2007-08-18	1.6.17	2011-06-02
1.0.2	2004-04-19	1.4.6	2007-12-17	1.6.18	2012-03-23
1.0.3	2004-05-19	1.5.0	2008-06-19	1.6.19	2012-09-12
1.0.4	2004-05-22	1.5.1	2008-07-24	1.6.20	2012-12-27
1.0.5	2004-06-10	1.5.2	2008-08-28	1.6.21	2013-03-29
1.0.6	2004-07-20	1.5.3	2008-10-09	1.6.23	2013-05-23
1.0.7	2004-09-18	1.5.4	2008-10-22	1.7.0	2011-09-09
1.0.8	2004-09-23	1.5.5	2008-12-19	1.7.1	2011-10-20
1.0.9	2004-11-14	1.5.6	2009-02-25	1.7.2	2011-11-29
1.1.0	2004-09-30	1.5.7	2009-08-06	1.7.3	2012-02-10
1.1.1	2004-11-23	1.5.9	2010-12-02	1.7.4	2012-03-02
1.1.2	2004-12-21	1.6.0	2009-03-19	1.7.5	2012-05-10
1.1.3	2005-01-15	1.6.1	2009-04-09	1.7.6	2012-08-28
1.1.4	2005-04-02	1.6.2	2009-05-07	1.7.7	2012-10-04
1.2.0	2005-05-23	1.6.3	2009-06-18	1.7.8	2012-12-10
1.2.1	2005-07-06	1.6.4	2009-08-06	1.7.9	2013-03-29
1.2.3	2005-08-25	1.6.5	2009-08-20	1.7.10	2013-05-23
1.3.0	2006-01-14	1.6.6	2009-10-22	1.8.0	2013-06-13
1.3.1	2006-04-03	1.6.9	2010-01-20	1.8.1	2013-07-24
1.3.2	2006-05-23	1.6.11	2010-04-15	1.8.3	2013-08-30
1.4.0	2006-08-22	1.6.12	2010-06-18	1.8.4	2013-10-22
1.4.2	2006-11-02	1.6.13	2010-09-29	1.8.5	2013-11-25
1.4.3	2007-01-17	1.6.15	2010-11-23		

B.5 Versions data sets

The following table lists the projects' versions generated for the data sets. OO metrics are CLAS (number of classes defined in the artefact) and DITM (Depth of Inheritance Tree). Diff metrics are LADD, LMOD, LREM (Number of lines added, modified and removed since the last analysis. Time metrics for SCM are SCM_*_1W, SCM_*_1M, and SCM_*_3M. Total metrics for SCM are SCM_COMMITS_TOTAL, SCM_COMMITTERS_TOTAL, SCM_COMMITS_FILES_TOTAL and SCM_FIXES_TOTAL. Time metrics for Comm. are COM_*_1W, COM_*_1M, and COM_*_3M.

			Code			SCM		Comm.	Rules		
Project	Lang.	Version	Common	OO	Diff	Time	Total	Time	SQUORE	PMD	Checkstyle
Epsilon	Java	2013-03-05	X	X			X		X	X	X
Epsilon	Java	2013-09-05	X	X			X		X	X	X
Subversion	C	2010-01-04	X				X		X		
Subversion	C	2010-07-05	X				X		X		
Topcased Gendoc	Java	4.0.0	X	X			X		X	X	X
Topcased Gendoc	Java	4.2.0	X	X			X		X	X	X
Topcased Gendoc	Java	4.3.0	X	X			X		X	X	X
Topcased Gendoc	Java	5.0.0	X	X			X		X	X	X
Topcased MM	Java	1.0.0	X	X			X		X	X	X
Topcased MM	Java	2.0.0	X	X			X		X	X	X
Topcased MM	Java	4.0.0	X	X			X		X	X	X
Topcased MM	Java	4.2.0	X	X			X		X	X	X
Topcased gPM	Java	1.2.7	X	X			X		X	X	X
Topcased gPM	Java	1.3.0	X	X			X		X	X	X

Appendix C

Knitr documents

C.1 SQuORE Lab Outliers

This SQuORE Lab `knitr` document was written to investigate various statistical methods for outliers detection and their application to todo lists. See section 8.3.1 for more information. We first show an extract of the `knitr` document featuring graphs and tables of boxplots outliers for the combination of metrics. Some pages extracted from the generated document follow.

```
\paragraph{3 metrics combination}

In the following plot, outliers on SLOC are plotted in dark green, outliers on VG are plotted in blue,
and outliers on NCC are plotted in purple. Intersection between all three variables is plotted in light
red (red3) and union is plotted in dark red (red4).

<<file_out_box_3, results='hide'>>=
outs_sloc <- which(project_data[,c("SLOC")] %in% boxplot.stats(project_data[,c("SLOC")])$out)
outs_vg <- which(project_data[,c("VG")] %in% boxplot.stats(project_data[,c("VG")])$out)
outs_ncc <- which(project_data[,c("NCC")] %in% boxplot.stats(project_data[,c("NCC")])$out)

outs_i <- intersect(outs_sloc, outs_vg)
outs_i <- intersect(outs_i, outs_ncc)
outs_u <- union(outs_sloc, outs_vg)
outs_u <- union(outs_sloc, outs_ncc)

pchs            <- rep(".", nrow(project_data))
pchs[outs_sloc] <- "o"
pchs[outs_vg]   <- "o"
pchs[outs_ncc]  <- "o"
cols            <- rep("black", nrow(project_data))
cols[outs_sloc] <- "darkgreen"
cols[outs_vg]   <- "blue"
cols[outs_ncc]  <- "purple"
cols[outs_i]    <- "red1"
cols[outs_u]    <- "red4"

jpeg(file="figures/file_outliers_box_m3.jpg", width=2000, height=2000,
     quality=100, pointsize=12, res=100)
pairs(project_data[,c("COMR", "SLOC", "VG", "NCC")], pch=pchs, col=cols)
dev.off()
@

\begin{figure}[hbt]
```

```
\begin{center}
\includegraphics[width=\linewidth]{file_outliers_box_m3.jpg}
\end{center}
\caption{3-metrics combination}
\label{fig:file_outliers_box_m3}
\end{figure}

Table \ref{tab:file_outliers_box_sloc_vg_ncc} gives some of the artefacts that are
Outliers both on SLOC and NCC:

\vspace{10pt}
\begin{table}[hbt]
\begin{center}
<<file_outliers_box_table_sloc_vg_ncc, results='asis'>>=

a <- project[outs_i, 3]
a <- as.data.frame(a)
names(a) <- c("File name")

print( xtable(a, align="|r|l|"), tabular.environment="longtable",
       floating=FALSE, size="\\scriptsize"
)

rm(a)
@
\end{center}
\caption{Artefacts with SLOC, VG and NCC outliers -- \texttt{boxplot.stats} method}
\label{tab:file_outliers_box_sloc_ncc}
\end{table}
\vspace{10pt}
```

The following extracts show:

- ⇝ The list of figures and tables generated in the document.
- ⇝ The quick statistics summary at the beginning of the analysis.
- ⇝ Scatterplots of some metrics to visually grasp their shape.
- ⇝ The table of metrics sorted according to their number of outliering values.
- ⇝ The output of a multivariate hierarchical clustering on file artefacts using various linkage methods, with a dendogram and 2-D and 3-D plots.
- ⇝ Another dendogram showing multivariate hierarchical clustering with the Manhattan distance and single linkage.

List of Figures

List of Tables

*Todo list

Metric	Min	Max	Mean	Var	Modes	NAs
BLAN	0	2844	53.612	15598.159	344	0
CFT	0	5825	164.872	526632.995	402	0
CLAS	0	200	2.467	30.326	46	0
CLOC	0	3808	122.48	54217.909	548	0
CLOR	0	100	15.915	1249.654	238	0
COMR	0	100	46.453	422.383	2162	0
ELOC	0	8530	269.594	678703.431	677	0
FUNC	0	747	21.856	3983.681	199	0
GOTO	0	0	0	0	1	0
LC	6	12382	529.108	1742786.111	993	0
NCC	0	9992	263.137	469418.279	760	0
NCC_ANA	0	1589	15.294	4813.389	142	0
NCC_CHAN	0	1529	24.439	4409.003	226	0
NCC_REL	0	732	23.57	4200.059	212	0
NCC_REUS	0	733	28.175	4502.406	240	0
NCC_STAB	0	399	2.09	109.548	57	0
NCC_TEST	0	2608	16.584	7171.917	141	0
NEST	0	0	0	0	1	0
NOP	0	0	0	0	1	0
PATH	0	0	0	0	1	0
RKO	0	44	11.511	43.458	37	0
RKO_ANA	0	3	0.842	0.62	4	0
RKO_CHAN	0	8	2.129	3.317	9	0
RKO_REL	0	4	1.208	0.55	5	0
RKO_REUS	0	8	2.312	0.773	9	0
RKO_STAB	0	6	0.373	0.535	7	0
RKO_TEST	0	8	2.1	4.913	9	0
ROKR	69.231	100	91.915	20.946	52	0
ROKR_ANA	40	100	83.156	247.948	4	0
ROKR_CHAN	38.462	100	83.425	194.418	11	0
ROKR_REL	84.615	100	95.353	8.132	5	0
ROKR_REUS	80	100	94.219	4.83	9	0
ROKR_STAB	33.333	100	95.852	66.09	7	0
ROKR_TEST	50	100	86.723	190.521	10	0
SCM_COMMITS_1M	0	5	0.141	0.216	6	0
SCM_COMMITS_1W	0	3	0.065	0.074	4	0
SCM_COMMITS_3M	0	14	1.588	2.273	13	0
SCM_COMMITS_TOTAL	0	62	8.153	29.418	23	0
SCM_COMMITTERS_1M	0	3	0.107	0.1	4	0
SCM_COMMITTERS_1W	0	2	0.06	0.06	3	0
SCM_COMMITTERS_3M	0	4	1.185	0.506	5	0
SCM_COMMITTERS_TOTAL	0	6	2.156	0.896	7	0
SCM_FIXES_1M	0	5	0.125	0.192	6	0
SCM_FIXES_1W	0	3	0.054	0.056	4	0
SCM_FIXES_3M	0	13	0.632	1.794	11	0
SCM_FIXES_TOTAL	0	52	3.428	18.286	17	0
SLOC	0	10093	355.05	1375443.729	753	0
STAT	0	8763	263.778	1244857.345	526	0
VG	0	2925	96.095	134581.42	345	0

Table 1: File metrics exploration

3 metrics combination In the following plot, outliers on SLOC are plotted in dark green, outliers on VG are plotted in blue, and outliers on NCC are plotted in purple. Intersection between all three variables is plotted in light red (red3) and union is plotted in dark red (red4).

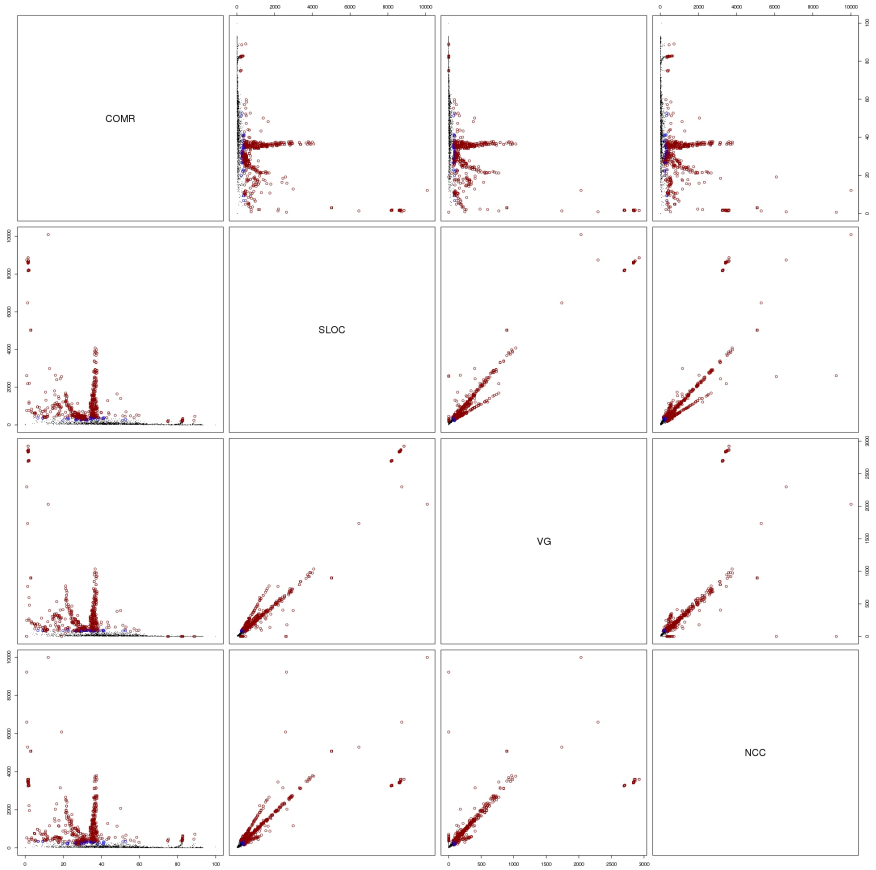


Figure 5: 3-metrics combination

Table **??** gives some of the artefacts that are outliers both on SLOC and NCC:

All-metrics outliers combination The global union and intersection of outliers collected on all individual metrics are plotted in figure 6. The union (2718 elements) is plotted in dark green crosses, while intersection (0 elements) is plotted with red circles. It is a possibility that intersection is empty, considering

2.3 Univariate sorting

The following table 7 gives the number of outliers for each variable with the boxplot method:

	Count of outliers	Percent of whole set
NCC_STAB	1059	5.41
VG	942	4.81
CLOR	933	4.76
STAT	849	4.33
NCC_CHAN	845	4.31
CFT	833	4.25
SLOC	794	4.05
NCC	736	3.76
ELOC	731	3.73
NCC_ANA	718	3.67
BLAN	710	3.62
FUNC	708	3.61
NCC_REUS	639	3.26
NCC_REL	604	3.08
LC	600	3.06
SCM_COMMITS_1M	523	2.67
SCM_COMMITTERS_1M	523	2.67
RKO_STAB	509	2.60
ROKR_STAB	509	2.60
CLOC	497	2.54
NCC_TEST	496	2.53
CLAS	490	2.50
ROKR_CHAN	487	2.49
SCM_FIXES_1M	467	2.38
SCM_COMMITS_3M	440	2.25
SCM_FIXES_3M	352	1.80
SCM_FIXES_TOTAL	333	1.70
SCM_COMMITTERS_TOTAL	298	1.52
SCM_COMMITS_1W	291	1.49
SCM_COMMITTERS_1W	291	1.49
SCM_FIXES_1W	256	1.31
RKO_REUS	231	1.18
ROKR_REUS	231	1.18
RKO_TEST	204	1.04
ROKR_TEST	204	1.04
RKO_ANA	58	0.30
ROKR_ANA	58	0.30
SCM_COMMITS_TOTAL	52	0.27
RKO_CHAN	39	0.20
RKO	16	0.08
ROKR	16	0.08
RKO_REL	5	0.03
ROKR_REL	5	0.03
SCM_COMMITTERS_3M	5	0.03
COMR	1	0.01

Figure 7: Number of outliers with `boxplot.stats`

Next table shows for a few files the metrics that make them outliers.

2.5 Hierarchical clustering – Euclidean distance

Hierarchical clustering relies on distance to find similarities between points and to identify clusters. In figure 25 the euclidean distance is used to find similarities.

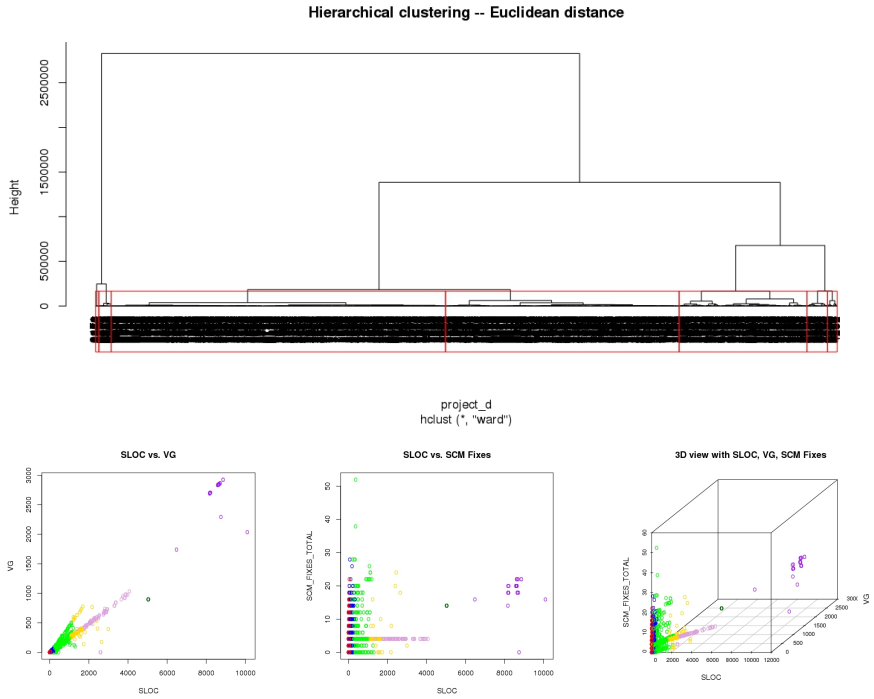


Figure 12: Euclidean distance, Ward method

Number of items in each cluster:

Method used	Cl1	Cl2	Cl3	Cl4	Cl5	Cl6	Cl7	Total
Ward	2260	1579	864	83	139	22	65	5012
Average	4896	1	81	23	1	2	8	5012
Single	4905	1	80	22	1	1	2	5012
Complete	4697	81	182	23	1	2	26	5012
McQuitty	4879	1	80	23	1	2	26	5012
Median	4905	1	80	22	1	1	2	5012
Centroid	4896	1	81	23	1	2	8	5012

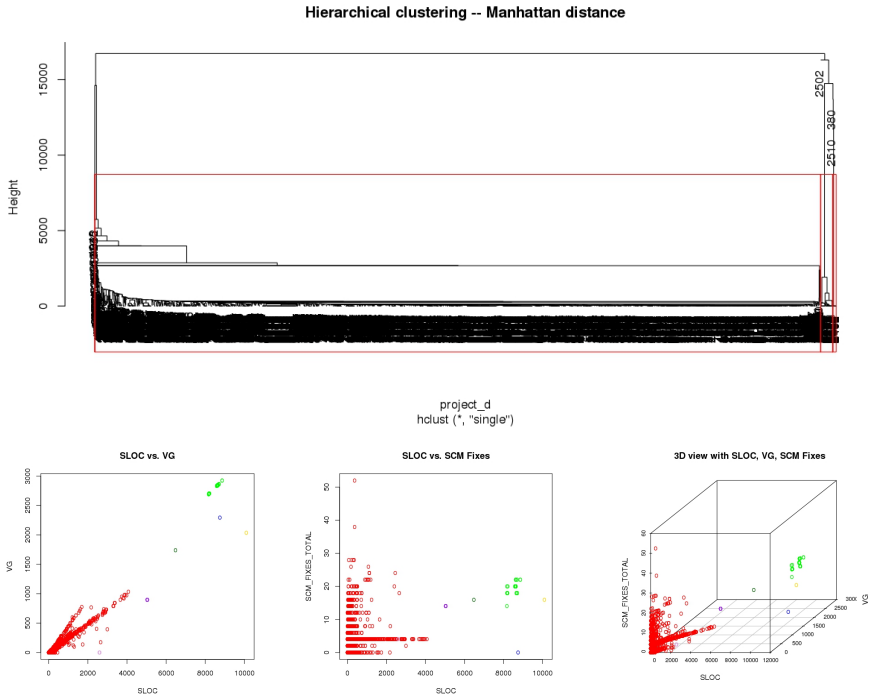


Figure 20: Manhattan distance, Single method

C.2 SQUORE Lab Clustering

In the Clustering SQUORE Lab we needed to investigate clustering methods for automatic categorisation of artefacts. To achieve this we first wrote a generic `knitr` document that applied various clustering methods to file artefacts, trying to find natural clusters in measures. This generic document was then refined in a more specific version to provide consultants with a small and fast report to use during the calibration phase of a quality model.

In the document presented below, each metric from the data set is individually treated to identify the natural threshold values that can be used to configure the SQUORE quality models. This document is entirely automated and relies on some heavy computations to mix R commands with LaTeXformatting directives. An extract of the Knitr source code is provided below.

```
ranges <- as.data.frame( array( c(1,1,2,2,3,3,4,4,5,5,6,6,7,7), dim=c(2,7) ) )
for (j in 1:7) {
    myranges <- range(project_data[project_data_kmeans_7$cluster == j,i])
    ranges[1,j] <- myranges[1]
    ranges[2,j] <- myranges[2]
}
ranges <- ranges[,order(ranges[1,])]
names(ranges) <- c("A", "B", "C", "D", "E", "F", "G")

print( xtable( ranges, digits=0 ),
       include.rownames=FALSE,
       floating=FALSE )

cat( '\\end{center}
     \\vspace{5pt}

     The following scale is proposed:

\\begin{verbatim}
<Scale scaleId="SCALE_NAME">
    <ScaleLevel levelId="LEVELA" bounds="[0;' )
    cat( ranges[2,1] )
    cat( ']" rank="0" />
    <ScaleLevel levelId="LEVELB" bounds="]' )
    cat( ranges[2,1] )
```

The following extracted pages show:

- ⇝ The table of contents with all metrics for which a custom scale is proposed.
- ⇝ Three pages for the COMR, VG, SCM_COMMITS metrics, with the number of artefacts in each clusters, the scale proposed in the format used by SQUORE for its configuration files, and a plot of the repartition of artefacts.

Calibration Wizard

Flavien Huynh, Boris Baldassari

January 20, 2014

Contents

List of Figures

2.5 COMR

Automatic clustering proposes the following repartition of artefacts:

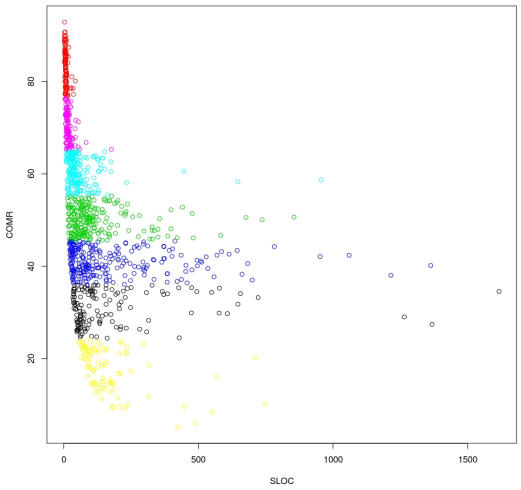
1	2	3	4	5	6	7
138	126	234	247	200	141	90

The associated ranges are the following:

A	B	C	D	E	F	G
5	24	36	46	55	65	77
24	36	45	55	65	76	93

The following scale is proposed:

```
<Scale scaleId="SCALE_NAME">
    <ScaleLevel levelId="LEVELA" bounds="[0;24.03]" rank="0" />
    <ScaleLevel levelId="LEVELB" bounds="]24.03;35.94]" rank="1" />
    <ScaleLevel levelId="LEVELC" bounds="]35.94;45.45]" rank="2" />
    <ScaleLevel levelId="LEVELD" bounds="]45.45;55.16]" rank="4" />
    <ScaleLevel levelId="LEVELE" bounds="]55.16;65]" rank="8" />
    <ScaleLevel levelId="LEVELF" bounds="]65;76.47]" rank="16" />
    <ScaleLevel levelId="LEVELG" bounds="]76.47;[" rank="32" />
</Scale>
```



2.14 VG

Automatic clustering proposes the following repartition of artefacts:

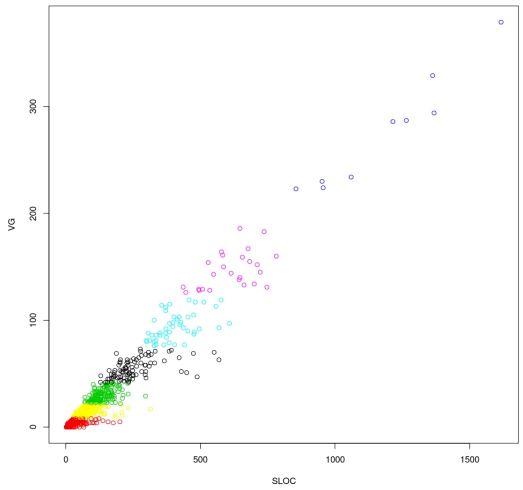
1	2	3	4	5	6	7
92	518	172	9	47	25	313

The associated ranges are the following:

A	B	C	D	E	F	G
0	9	22	42	76	126	223
8	21	41	73	119	186	379

The following scale is proposed:

```
<Scale scaleId="SCALE_NAME">
    <ScaleLevel levelId="LEVELA" bounds="[0;8]" rank="0" />
    <ScaleLevel levelId="LEVELB" bounds="]8;21]" rank="1" />
    <ScaleLevel levelId="LEVELC" bounds="]21;41]" rank="2" />
    <ScaleLevel levelId="LEVELD" bounds="]41;73]" rank="4" />
    <ScaleLevel levelId="LEVELE" bounds="]73;119]" rank="8" />
    <ScaleLevel levelId="LEVELF" bounds="]119;186]" rank="16" />
    <ScaleLevel levelId="LEVELG" bounds="]186;[" rank="32" />
</Scale>
```



2.15 SCM_COMMITS_TOTAL

Automatic clustering proposes the following repartition of artefacts:

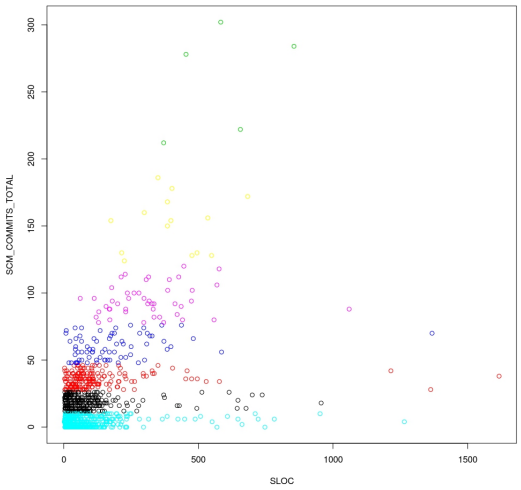
1	2	3	4	5	6	7
279	186	5	73	573	46	14

The associated ranges are the following:

A	B	C	D	E	F	G
0	12	28	48	78	124	212
10	26	46	76	120	186	302

The following scale is proposed:

```
<Scale scaleId="SCALE_NAME">
    <ScaleLevel levelId="LEVELA" bounds="[0;10]" rank="0" />
    <ScaleLevel levelId="LEVELB" bounds="]10;26]" rank="1" />
    <ScaleLevel levelId="LEVELC" bounds="]26;46]" rank="2" />
    <ScaleLevel levelId="LEVELD" bounds="]46;76]" rank="4" />
    <ScaleLevel levelId="LEVELE" bounds="]76;120]" rank="8" />
    <ScaleLevel levelId="LEVELF" bounds="]120;186]" rank="16" />
    <ScaleLevel levelId="LEVELG" bounds="]186;[" rank="32" />
</Scale>
```



C.3 SQUORE Lab Correlations

In the Correlations SQUORE Lab regression methods are applied to various sets of metrics and rules to unveil inter-relationships between them. See chapter 10 page 167 for more information on this work.

The following pages have been extracted from the JMeter analysis document:

- ⇝ The table of the paiwise linear regression on metrics.
- ⇝ The table of the pairwise polynomial regression on metrics.
- ⇝ Three pages of results for the pairwise linear regression of metrics against rules; they represent respectively rules from SQUORE, PMD and Checkstyle.
- ⇝ The table of the pairwise polynomial regression of metrics against rules (SQUORE only).

	BLAN	BRAC	CFT	CLAS	CLOC	COMR	ELOC	FUNC	LC	MLOC	NCC	ROKR	SCM_COMMITS_TOTAL	SCM_COMMITTERS_TOTAL	SCM_FIXES_TOTAL	SLOC	STAT	VG
BLAN	1.00	0.489	0.573		0.459		0.736	0.685	0.776	0.366	0.681	0.427				0.726	0.689	0.664
BRAC	0.49	**1**	0.76		0.482		0.658	0.653	0.768		0.556	0.596				0.771	0.602	0.782
CFT	0.57	0.76	**1**		0.55		0.757	0.608	0.814	0.376	0.635	0.629			0.206	0.808	0.754	**0.924**
CLAS				**1**														
CLOC	0.46	0.482	0.55		**1**		0.455	0.535	0.717	0.248	0.435	0.364				0.491	0.422	0.582
COMR						**1**	0.249					0.313				0.252	0.224	
ELOC	0.74	0.658	0.757		0.455	0.249	**1**	0.62	**0.913**	0.396	0.814	0.596				**0.984**	**0.959**	0.756
FUNC	0.69	0.653	0.608		0.535		0.62	**1**	0.741	0.222	0.666	0.453				0.668	0.538	0.772
LC	0.78	0.768	0.814		0.717		**0.913**	0.741	**1**	0.345	0.787	0.616				**0.94**	0.864	0.846
MLOC	0.37		0.376		0.248		0.396	0.222	0.345	**1**	0.316	0.21				0.367	0.362	0.331
NCC	0.68	0.556	0.635		0.435		0.814	0.666	0.787	0.316	**1**	0.574				0.806	0.769	0.704
ROKR	0.43	0.596	0.629		0.364	0.313	0.596	0.453	0.616	0.21	0.574	**1**				0.635	0.553	0.603
SCM_COMMITS_TOTAL													**1**	0.483	0.801			
SCM_COMMITTERS_TOTAL													0.483	**1**	0.31			
SCM_FIXES_TOTAL			0.206										0.801	0.31	**1**	0.201		
SLOC	0.73	0.771	0.808		0.491	0.252	**0.984**	0.668	**0.94**	0.367	0.806	0.635			0.201	**1**	**0.936**	0.812
STAT	0.69	0.602	0.754		0.422	0.224	**0.959**	0.538	0.864	0.362	0.769	0.553				**0.936**	**1**	0.737
VG	0.66	0.782	**0.924**		0.582		0.756	0.772	0.846	0.331	0.704	0.603				0.812	0.737	**1**

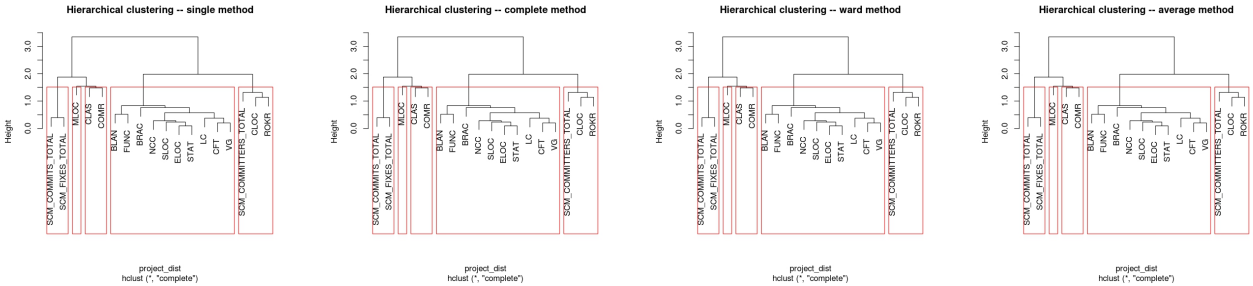


Table 3: Linear regression: adjusted R^2 for file metrics and clustering of results.

	BLAN	BRAC	CFT	CLAS	CLOC	COMR	ELOC	FUNC	LC	MLOC	NCC	ROKR	SCM_COMMITS_TOTAL	SCM_COMMITTERS_TOTAL	SCM_FIXES_TOTAL	SLOC	STAT	VG
BLAN	1.00	0.525	0.582	0.177	0.472	0.153	0.738	0.688	0.776	0.383	0.691	0.429	0.13	0.04	0.162	0.727	0.695	0.666
BRAC	0.53	**1**	0.768	0.182	0.488	0.193	0.668	0.659	0.768	0.184	0.58	0.637	0.178	0.08	0.201	0.773	0.626	0.788
CFT	0.61	0.77	**1**	0.149	0.555	0.153	0.759	0.611	0.814	0.378	0.653	0.678	0.17	0.05	0.208	0.808	0.765	**0.924**
CLAS	0.17	0.16	0.131	**1**	0.086	0.069	0.187	0.163	0.182	0.057	0.143	0.151	0.047	0.016	0.055	0.192	0.162	0.148
CLOC	0.49	0.486	0.55	0.092	**1**	0.028	0.458	0.542	0.719	0.252	0.441	0.401	0.1	0.021	0.111	0.491	0.432	0.583
COMR	0.23	0.3	0.267	0.07		**1**	0.388	0.306	0.255	0.029	0.295	0.446	0.041	0.017	0.06	0.398	0.349	0.279
ELOC	0.76	0.68	0.764	0.197	0.47	0.266	**1**	0.634	**0.913**	0.401	0.824	0.613	0.151	0.047	0.191	**0.984**	**0.961**	0.763
FUNC	0.71	0.678	0.622	0.167	0.535	0.179	0.635	**1**	0.741	0.232	0.681	0.455	0.115	0.042	0.142	0.677	0.566	0.776
LC	0.80	0.782	0.82	0.195	0.725	0.154	**0.915**	0.744	**1**	0.351	0.8	0.643	0.164	0.05	0.196	**0.94**	0.871	0.85
MLOC	0.37	0.188	0.396	0.056	0.252	0.023	0.399	0.225	0.352	**1**	0.318	0.228	0.08	0.019	0.115	0.373	0.362	0.344
NCC	0.69	0.576	0.642	0.146	0.442	0.196	0.814	0.668	0.787	0.317	**1**	0.592	0.105	0.023	0.128	0.807	0.77	0.706
ROKR	0.53	0.658	0.69	0.179	0.388	0.314	0.664	0.505	0.662	0.223	0.67	**1**	0.138	0.036	0.155	0.692	0.637	0.663
SCM_COMMITS_TOTAL	0.15	0.177	0.178	0.064	0.121	0.035	0.16	0.125	0.17	0.098	0.125	0.142	**1**	0.575	0.803	0.17	0.146	0.162
SCM_COMMITTERS_TOTAL	0.02	0.041	0.021	0.014	0.023		0.015	0.014	0.024		0.013	0.016	0.561	**1**	0.326	0.019	0.012	0.019
SCM_FIXES_TOTAL	0.19	0.2	0.213	0.068	0.132	0.049	0.197	0.154	0.2	0.125	0.147	0.157	0.804	0.428	**1**	0.207	0.185	0.197
SLOC	0.75	0.786	0.815	0.206	0.504	0.266	**0.985**	0.681	**0.94**	0.372	0.82	0.658	0.167	0.056	0.205	**1**	**0.941**	0.819
STAT	0.70	0.623	0.759	0.168	0.436	0.243	**0.96**	0.553	0.864	0.368	0.776	0.572	0.132	0.04	0.171	**0.937**	**1**	0.744
VG	0.70	0.798	**0.925**	0.16	0.586	0.163	0.76	0.772	0.846	0.337	0.715	0.633	0.156	0.05	0.191	0.812	0.752	**1**

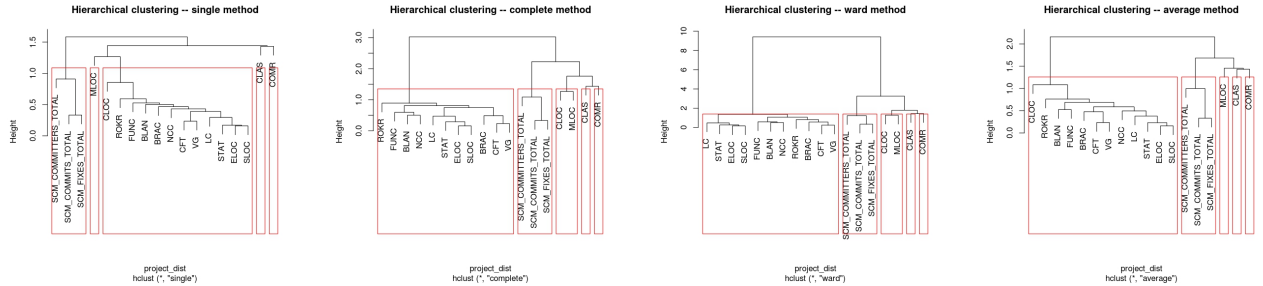


Table 7: Polynomial regression: adjusted R^2 for file metrics and clustering of results

Figure 4: Linear regression of SQuORE rules: lm(metrics ~ rules)

	BLAN	BRAC	CFT	CLAS	CLOC	COMR	ELOC	FUNC	LC	MLOC	NCC	ROKR	SCM_COMMITS_TOTAL	SCM_COMMITTERS_TOTAL	SCM_FIXES_TOTAL	SLOC	STAT	VG
R_NOASGINBOOL																		
R_RETURN	0.12	0.24	0.33	0.02	0.18	0.04	0.16	0.21	0.20	0.06	0.18	0.21	0.05		0.06	0.18	0.16	0.30
R_SGLBRK					0.02							0.01						
R_ELSEFINAL	0.11	0.17	0.22	0.03	0.10	0.03	0.16	0.07	0.17	0.08	0.12	0.18	0.03		0.04	0.17	0.18	0.21
R_COMPOUNDFULL	0.03	0.06	0.06		0.06	0.01	0.05	0.03	0.06		0.09	0.13				0.05	0.05	0.06
R_BRKFINAL			0.01	0.01			0.01		0.01		0.01	0.01				0.01	0.01	0.01
R_NOCLONE_FUNCTION												0.03						
R_DEFAULT	0.02	0.01	0.03	0.01			0.04		0.03	0.02	0.02	0.02	0.01			0.03	0.04	0.02
R_NOASGCOND		0.04	0.04		0.02	0.02	0.04		0.04		0.03	0.04				0.04	0.06	0.04
R_NOCLONE_FILE																		
R_NOFALLTHROUGH																		
R_NOCONT	0.05	0.09	0.14		0.04	0.02	0.09	0.04	0.08	0.04	0.07	0.07	0.01		0.02	0.10	0.09	0.12

Figure 5: Linear regression of PMD rules: lm(metrics ~ rules)

	BLAN	BRAC	CFT	CLAS	CLOC	COMR	ELOC	FUNC	LC	MLOC	NCC	ROKR	SCM_COMMITS_TOTAL	SCM_COMMITTERS_TOTAL	SCM_FIXES_TOTAL	SLOC	STAT	VG
EMPTYFINALLYBLOCK																		
DUPLICATEIMPORTS																		
TOOMANYFIELDS	0.14	0.05	0.053	0.04	0.06	0.01	0.12	0.06	0.108	0.12	0.10	0.06	0.04	0.01	0.04	0.11	0.11	0.064
PRESERVESTACKTRACE	0.02	0.05	0.061		0.04		0.03	0.01	0.041		0.02	0.07	0.02		0.02	0.04	0.03	0.035
ARRAYISSTOREDDIRECTLY			0.011		0.01						0.01	0.01						0.015
UNUSEDLOCALVARIABLE		0.01					0.03		0.018		0.04	0.01				0.02	0.02	
AVOIDCATCHINGTHROWABLE		0.01	0.016									0.01				0.01		
BOOLEANINSTANTIATION																		
CONFUSINGTERNARY	0.15	0.22	0.295	0.02	0.23	0.02	0.26	0.16	0.279	0.10	0.26	0.22	0.04		0.05	0.27	0.26	0.282
EMPTYWHILESTMT																		
SYSTEMPRINTLN			0.01				0.02		0.015		0.06	0.02				0.02	0.03	
EXCESSIVEIMPORTS	0.18	0.20	0.289	0.07	0.16	0.01	0.26	0.13	0.25	0.26	0.18	0.21	0.09		0.12	0.27	0.23	0.237
AVOIDTHROWINGNULLPOINTEREXCEPTION																		
SWITCHDENSITY																		
USENOTIFYALLINSTEADOFNOTIFY																		
NONCASELABELINSWITCHSTATEMENT																		
SIMPLIFYCONDITIONAL			0.011	0.01													0.01	
UNUSEDPRIVATEMETHOD												0.01						
DONOTTHROWEXCEPTIONINFINALLY																		
TOOMANYSTATICIMPORTS																		
METHODRETURNSINTERNALARRAY	0.03	0.03	0.029		0.04		0.02	0.05	0.031	0.01	0.03	0.03				0.02	0.02	0.04
AVOIDTHREADGROUP																		
COUPLINGBETWEENOBJECTS																		
AVOIDUSINGHARDCODEDIP																		
CLOSERESOURCE													0.01					
AVOIDRETHROWINGEXCEPTION												0.01			0.01			
EMPTYCATCHBLOCK		0.01	0.02			0.02	0.02	0.01	0.012		0.02	0.02				0.02	0.01	
EMPTYIFSTMT		0.03	0.016	0.03	0.01		0.02		0.022		0.03	0.03				0.03	0.02	0.014
IMPORTFROMSAMEPACKAGE																		
AVOIDDEEPLYNESTEDIFSTMTS	0.01	0.02	0.028	0.01	0.02		0.02		0.023		0.01	0.04				0.02	0.02	0.026
USESTRINGBUFFERLENGTH																		
EMPTYSTATEMENTNOTINLOOP																		
AVOIDTHROWINGRAWEXCEPTIONTYPES	0.06	0.07	0.08		0.10		0.04	0.09	0.074	0.02	0.05	0.07	0.02		0.01	0.05	0.03	0.08
ASSIGNMENTTONONFINALSTATIC												0.01						
GODCLASS	0.34	0.49	**0.611**	0.06	0.36	0.04	0.46	0.40	**0.513**	0.18	0.43	0.38	0.10	0.01	0.13	0.50	0.47	**0.607**
UNCONDITIONALIFSTATEMENT																		
RETURNEMPTYARRAYRATHERTHANNULL				0.01				0.01										
UNUSEDPRIVATEFIELD												0.01						
DONTIMPORTJAVALANG																		
EXCESSIVEPUBLICCOUNT	0.18	0.16	0.149	0.03	0.24		0.11	0.33	0.188	0.08	0.17	0.06				0.13	0.08	0.215
UNNECESSARYCASECHANGE																		
NONTHREADSAFESINGLETON		0.01	0.01									0.01						
POSITIONLITERALSFIRSTINCOMPARISONS		0.02	0.019		0.01		0.01		0.017		0.02	0.03				0.02	0.01	0.017
LOOSECOUPLING	0.03	0.07	0.074		0.07		0.04	0.04	0.059	0.01	0.03	0.06				0.04	0.04	0.055
AVOIDCATCHINGNPE		0.03	0.034		0.02		0.02	0.02	0.024		0.02	0.04				0.03	0.02	0.021

Figure 6: Linear regression of Checkstyle rules: lm(metrics ~ rules)

	BLAN	BRAC	CFT	CLAS	CLOC	COMR	ELOC	FUNC	LC	MLOC	NCC	ROKR	SCM_COMMITS_TOTAL	SCM_COMMITTERS_TOTAL	SCM_FIXES_TOTAL	SLOC	STAT	VG
ARRAYTYPESTYLECHECK	0.02	0.03	0.04		0.01	0.01	0.046	0.015	0.034	0.03	0.041	0.05				0.044	0.052	0.038
HIDDENFIELDCHECK	0.06	0.05	0.04		0.05	0.01	0.033	0.104	0.052		0.07	0.06				0.038	0.024	0.077
ILLEGALTHROWSCHECK			0.01															0.012
AVOIDSTARIMPORTCHECK																		
JAVADOCPACKAGECHECK												0.01						
EMPTYBLOCKCHECK	0.01	0.02	0.03	0.01		0.02	0.028	0.017	0.021		0.036	0.04				0.028	0.026	0.015
REDUNDANTMODIFIERCHECK						0.03												
JAVADOCMETHODCHECK	0.48	0.44	0.43	0.08	0.24	0.17	**0.504**	**0.65**	**0.504**	0.15	**0.668**	0.36	0.07		0.08	**0.523**	0.458	**0.544**
UNUSEDIMPORTSCHECK																		
ONESTATEMENTPERLINECHECK	0.03	0.01					0.012	0.02	0.014	0.01	0.029	0.01				0.012	0.011	0.03
NEEDBRACESCHECK	0.03	0.06	0.06		0.06		0.043	0.031	0.059		0.093	0.14				0.048	0.051	0.058
NOWHITESPACEBEFORECHECK																		
EQUALSHASHCODECHECK															0.01			
MULTIPLESTRINGLITERALSCHECK	0.18	0.09	0.09	0.03	0.03	0.11	0.335	0.112	0.214	0.04	0.303	0.10				0.289	0.348	0.086
ANONINNERLENGTHCHECK	0.02	0.04	0.04	0.06	0.02		0.056	0.019	0.046	0.02	0.053	0.03				0.056	0.052	0.033
MISPLACEDNULLCHECK																		
MODIFIERORDERCHECK	0.04	0.06	0.04		0.03		0.04	0.055	0.048	0.02	0.082	0.06	0.02	0.01	0.01	0.047	0.029	0.047
CLASSFANOUTCOMPLEXITYCHECK	0.23	0.26	0.36	0.11	0.19	0.04	0.337	0.19	0.32	0.23	0.24	0.30	0.11	0.01	0.14	0.342	0.306	0.299
REDUNDANTTHROWSCHECK		0.02	0.01			0.02	0.033	0.019	0.026		0.052	0.03				0.032	0.019	0.013
PARAMETERASSIGNMENTCHECK	0.03	0.09	0.11		0.07		0.084	0.054	0.094	0.02	0.1	0.11				0.092	0.082	0.103
REDUNDANTIMPORTCHECK																		
UNNECESSARYPARENTHESESCHECK	0.04	0.04	0.04		0.05		0.041	0.035	0.054		0.056	0.07				0.044	0.043	0.045
CLASSDATAABSTRACTIONCOUPLINGCHECK	0.25	0.17	0.22	0.13	0.13	0.08	0.311	0.153	0.27	0.20	0.203	0.26	0.08	0.01	0.10	0.297	0.279	0.197
INTERFACEISTYPECHECK																		
DESIGNFOREXTENSIONCHECK	0.39	0.37	0.28	0.05	0.31	0.11	0.306	**0.743**	0.397	0.09	0.444	0.22	0.04		0.05	0.34	0.242	0.448
VISIBILITYMODIFIERCHECK	0.03	0.08	0.03	0.02	0.04	0.02	0.041	0.039	0.056		0.066	0.08				0.052	0.035	0.039
LINELENGTHCHECK	0.47	0.27	0.45	0.11	0.32	0.07	**0.617**	0.326	**0.55**	0.32	**0.731**	0.33	0.09		0.11	**0.568**	**0.619**	0.452
AVOIDINLINECONDITIONALSCHECK	0.14	0.07	0.09		0.09		0.081	0.125	0.102	0.06	0.143	0.07	0.01		0.01	0.084	0.069	0.17
BROKENNULLCHECK																		
MAGICNUMBERCHECK	0.17	0.05	0.05	0.04	0.03	0.08	0.192	0.073	0.133	0.07	0.197	0.09			0.01	0.164	0.192	0.064
NEWLINEATENDOFFILECHECK																		
JAVADOCVARIABLECHECK	0.52	0.30	0.38	0.10	0.23	0.09	0.441	0.384	0.424	0.47	0.482	0.30	0.10		0.12	0.437	0.375	0.419
SIMPLIFYBOOLEANRETURNCHECK		0.01	0.01					0.01			0.012	0.01						0.012
RIGHTCURLYCHECK	0.05	0.40	0.14	0.03	0.07	0.05	0.154	0.061	0.173		0.12	0.20	0.07	0.03	0.06	0.208	0.139	0.117
JAVADOCTYPECHECK				0.17		0.06	0.02	0.012			0.018	0.04		0.01		0.018	0.017	
EMPTYSTATEMENTCHECK																		

Figure 13: Polynomial regression of SQuORE rules: plm(metrics rules)

	BLAN	BRAC	CFT	CLAS	CLOC	COMR	ELOC	FUNC	LC	MLOC	NCC	ROKR	SCM_COMMITS_TOTAL	SCM_COMMITTERS_TOTAL	SCM_FIXES_TOTAL	SLOC	STAT	VG
R_NOASGINBOOL																		
R_RETURN	0.19	0.33	0.42	0.04	0.26	0.05	0.23	0.28	0.29	0.10	0.25	0.31	0.07		0.08	0.26	0.22	0.40
R_SGLBRK					0.02							0.01						
R_ELSEFINAL	0.12	0.17	0.23	0.03	0.13	0.03	0.17	0.09	0.18	0.10	0.14	0.21	0.03		0.04	0.18	0.19	0.23
R_COMPOUNDFULL	0.03	0.07	0.07		0.06	0.02	0.05	0.04	0.07		0.09	0.15				0.06	0.06	0.07
R_BRKFINAL	0.01	0.01	0.02	0.02			0.03	0.01	0.02	0.01	0.03	0.02				0.03	0.03	0.02
R_NOCLONE_FUNCTION												0.03						
R_DEFAULT	0.02	0.01	0.03	0.04			0.04		0.03	0.02	0.02	0.02	0.01		0.01	0.03	0.04	0.02
R_NOASGCOND	0.01	0.06	0.07		0.04	0.02	0.05	0.02	0.05	0.01	0.03	0.07	0.02		0.02	0.05	0.06	0.06
R_NOCLONE_FILE																		
R_NOFALLTHROUGH																		
R_NOCONT	0.06	0.11	0.15		0.06	0.02	0.10	0.05	0.10	0.04	0.07	0.08	0.02		0.03	0.11	0.10	0.14

Appendix D

Code samples

D.1 Ant > Javadoc.java > execute()

The `execute()` function is presented in section 6.1.2 page 107 and figure 6.0 page 111 from chapter 5 to illustrate some metrics on real code.

```
%      style=JavaInputStyle]
public void execute() throws BuildException {
    if ("javadoc2".equals(getTaskType())) {
        log("!! javadoc2 is deprecated. Use javadoc instead. !!");
    }

    Vector packagesToDoc = new Vector();
    Path sourceDirs = new Path(getProject());

    if (packageList != null && sourcePath == null) {
        String msg = "sourcePath attribute must be set when "
            + "specifying packagelist.";
        throw new BuildException(msg);
    }

    if (sourcePath != null) {
        sourceDirs.addExisting(sourcePath);
    }

    parsePackages(packagesToDoc, sourceDirs);

    if (packagesToDoc.size() != 0 && sourceDirs.size() == 0) {
        String msg = "sourcePath attribute must be set when "
            + "specifying package names.";
        throw new BuildException(msg);
    }

    Vector sourceFilesToDoc = (Vector) sourceFiles.clone();
    addFileSets(sourceFilesToDoc);

    if (packageList == null && packagesToDoc.size() == 0
        && sourceFilesToDoc.size() == 0) {
        throw new BuildException("No source files and no packages have "
                                 + "been specified.");
    }

    log("Generating Javadoc", Project.MSG_INFO);

    Commandline toExecute = (Commandline) cmd.clone();
    toExecute.setExecutable(JavaEnvUtils.getJdkExecutable("javadoc"));

    // --------------------------------------- general javadoc arguments
    if (doctitle != null) {
        toExecute.createArgument().setValue("-doctitle");
        toExecute.createArgument().setValue(expand(doctitle.getText()));
    }
    if (header != null) {
        toExecute.createArgument().setValue("-header");
        toExecute.createArgument().setValue(expand(header.getText()));
    }
    if (footer != null) {
        toExecute.createArgument().setValue("-footer");
        toExecute.createArgument().setValue(expand(footer.getText()));
```

```
54      }
55      if (bottom != null) {
56          toExecute.createArgument().setValue("-bottom");
57          toExecute.createArgument().setValue(expand(bottom.getText()));
58      }
59
60      if (classpath == null) {
61          classpath = (new Path(getProject())).concatSystemClasspath("last");
62      } else {
63          classpath = classpath.concatSystemClasspath("ignore");
64      }
65
66      if (!javadoc1) {
67          if (classpath.size() > 0) {
68              toExecute.createArgument().setValue("-classpath");
69              toExecute.createArgument().setPath(classpath);
70          }
71          if (sourceDirs.size() > 0) {
72              toExecute.createArgument().setValue("-sourcepath");
73              toExecute.createArgument().setPath(sourceDirs);
74          }
75      } else {
76          sourceDirs.append(classpath);
77          if (sourceDirs.size() > 0) {
78              toExecute.createArgument().setValue("-classpath");
79              toExecute.createArgument().setPath(sourceDirs);
80          }
81      }
82
83      if (version && doclet == null) {
84          toExecute.createArgument().setValue("-version");
85      }
86      if (author && doclet == null) {
87          toExecute.createArgument().setValue("-author");
88      }
89
90      if (javadoc1 || doclet == null) {
91          if (destDir == null) {
92              String msg = "destDir attribute must be set!";
93              throw new BuildException(msg);
94          }
95      }
96
97      // ---------------------------- javadoc2 arguments for default doclet
98
99      if (!javadoc1) {
100         if (doclet != null) {
101             if (doclet.getName() == null) {
102                 throw new BuildException("The doclet name must be "
103                                          + "specified.", getLocation());
104             } else {
105                 toExecute.createArgument().setValue("-doclet");
106                 toExecute.createArgument().setValue(doclet.getName());
107                 if (doclet.getPath() != null) {
108                     Path docletPath
109                         = doclet.getPath().concatSystemClasspath("ignore");
110                     if (docletPath.size() != 0) {
111                         toExecute.createArgument().setValue("-docletpath");
112                         toExecute.createArgument().setPath(docletPath);
113                     }
114                 }
115                 for (Enumeration e = doclet.getParams();
116                      e.hasMoreElements();) {
117                     DocletParam param = (DocletParam) e.nextElement();
118                     if (param.getName() == null) {
119                         throw new BuildException("Doclet parameters must "
120                                                  + "have a name");
121                     }
122
123                     toExecute.createArgument().setValue(param.getName());
124                     if (param.getValue() != null) {
125                         toExecute.createArgument()
126                             .setValue(param.getValue());
127                     }
128                 }
129             }
130         }
131         if (bootclasspath != null && bootclasspath.size() > 0) {
132             toExecute.createArgument().setValue("-bootclasspath");
133             toExecute.createArgument().setPath(bootclasspath);
134         }
135
136         // add the links arguments
137         if (links.size() != 0) {
138             for (Enumeration e = links.elements(); e.hasMoreElements();) {
139                 LinkArgument la = (LinkArgument) e.nextElement();
```

```
140
141                 if (la.getHref() == null || la.getHref().length() == 0) {
142                     log("No href was given for the link - skipping",
143                         Project.MSG_VERBOSE);
144                     continue;
145                 } else {
146                     // is the href a valid URL
147                     try {
148                         URL base = new URL("file://.");
149                         new URL(base, la.getHref());
150                     } catch (MalformedURLException mue) {
151                         // ok - just skip
152                         log("Link href \"" + la.getHref()
153                             + "\" is not a valid url - skipping link",
154                             Project.MSG_WARN);
155                         continue;
156                     }
157                 }
158
159                 if (la.isLinkOffline()) {
160                     File packageListLocation = la.getPackagelistLoc();
161                     if (packageListLocation == null) {
162                         throw new BuildException("The package list "
163                                                  + " location for link " + la.getHref()
164                                                  + " must be provided because the link is "
165                                                  + "offline");
166                     }
167                     File packageListFile =
168                         new File(packageListLocation, "package-list");
169                     if (packageListFile.exists()) {
170                         try {
171                             String packageListURL =
172                                 fileUtils.getFileURL(packageListLocation)
173                                 .toExternalForm();
174                             toExecute.createArgument()
175                                 .setValue("-linkoffline");
176                             toExecute.createArgument()
177                                 .setValue(la.getHref());
178                             toExecute.createArgument()
179                                 .setValue(packageListURL);
180                         } catch (MalformedURLException ex) {
181                             log("Warning: Package list location was "
182                                 + "invalid " + packageListLocation,
183                                 Project.MSG_WARN);
184                         }
185                     } else {
186                         log("Warning: No package list was found at "
187                             + packageListLocation, Project.MSG_VERBOSE);
188                     }
189                 } else {
190                     toExecute.createArgument().setValue("-link");
191                     toExecute.createArgument().setValue(la.getHref());
192                 }
193             }
194         }
195
196         // add the single group arguments
197         // Javadoc 1.2 rules:
198         //   Multiple -group args allowed.
199         //   Each arg includes 3 strings: -group [name] [packagelist].
200         //   Elements in [packagelist] are colon-delimited.
201         //   An element in [packagelist] may end with the * wildcard.
202
203         // Ant javadoc task rules for group attribute:
204         //   Args are comma-delimited.
205         //   Each arg is 2 space-delimited strings.
206         //   E.g., group="XSLT_Packages org.apache.xalan.xslt*,
207         //                XPath_Packages org.apache.xalan.xpath*"
208         if (group != null) {
209             StringTokenizer tok = new StringTokenizer(group, ",", false);
210             while (tok.hasMoreTokens()) {
211                 String grp = tok.nextToken().trim();
212                 int space = grp.indexOf(" ");
213                 if (space > 0) {
214                     String name = grp.substring(0, space);
215                     String pkgList = grp.substring(space + 1);
216                     toExecute.createArgument().setValue("-group");
217                     toExecute.createArgument().setValue(name);
218                     toExecute.createArgument().setValue(pkgList);
219                 }
220             }
221         }
222
223         // add the group arguments
224         if (groups.size() != 0) {
225             for (Enumeration e = groups.elements(); e.hasMoreElements();) {
```

```
226                 GroupArgument ga = (GroupArgument) e.nextElement();
227                 String title = ga.getTitle();
228                 String packages = ga.getPackages();
229                 if (title == null || packages == null) {
230                     throw new BuildException("The title and packages must "
231                                              + "be specified for group "
232                                              + "elements.");
233                 }
234                 toExecute.createArgument().setValue("-group");
235                 toExecute.createArgument().setValue(expand(title));
236                 toExecute.createArgument().setValue(packages);
237             }
238         }
239 
240         // JavaDoc 1.4 parameters
241         if (javadoc4) {
242             for (Enumeration e = tags.elements(); e.hasMoreElements();) {
243                 Object element = e.nextElement();
244                 if (element instanceof TagArgument) {
245                     TagArgument ta = (TagArgument) element;
246                     File tagDir = ta.getDir(getProject());
247                     if (tagDir == null) {
248                         // The tag element is not used as a fileset,
249                         // but specifies the tag directly.
250                         toExecute.createArgument().setValue ("-tag");
251                         toExecute.createArgument().setValue (ta.getParameter());
252                     } else {
253                         // The tag element is used as a fileset. Parse all the files and
254                         // create -tag arguments.
255                         DirectoryScanner tagDefScanner = ta.getDirectoryScanner(getProject());
256                         String[] files = tagDefScanner.getIncludedFiles();
257                         for (int i = 0; i < files.length; i++) {
258                             File tagDefFile = new File(tagDir, files[i]);
259                             try {
260                                 BufferedReader in
261                                     = new BufferedReader(new FileReader(tagDefFile));
262                                 String line = null;
263                                 while ((line = in.readLine()) != null) {
264                                     toExecute.createArgument().setValue ("-tag");
265                                     toExecute.createArgument().setValue (line);
266                                 }
267                                 in.close();
268                             } catch (IOException ioe) {
269                                 throw new BuildException("Couldn't read "
270                                     + " tag file from "
271                                     + tagDefFile.getAbsolutePath(), ioe);
272                             }
273                         }
274                     }
275                 } else {
276                     ExtensionInfo tagletInfo = (ExtensionInfo) element;
277                     toExecute.createArgument().setValue("-taglet");
278                     toExecute.createArgument().setValue(tagletInfo
279                                                         .getName());
280                     if (tagletInfo.getPath() != null) {
281                         Path tagletPath = tagletInfo.getPath()
282                             .concatSystemClasspath("ignore");
283                         if (tagletPath.size() != 0) {
284                             toExecute.createArgument()
285                                 .setValue("-tagletpath");
286                             toExecute.createArgument().setPath(tagletPath);
287                         }
288                     }
289                 }
290             }
291 
292             if (source != null) {
293                 toExecute.createArgument().setValue("-source");
294                 toExecute.createArgument().setValue(source);
295             }
296 
297             if (linksource && doclet == null) {
298                 toExecute.createArgument().setValue("-linksource");
299             }
300             if (breakiterator && doclet == null) {
301                 toExecute.createArgument().setValue("-breakiterator");
302             }
303             if (noqualifier != null && doclet == null) {
304                 toExecute.createArgument().setValue("-noqualifier");
305                 toExecute.createArgument().setValue(noqualifier);
306             }
307         }
308 
309     }
310 
311     File tmpList = null;
```

```
312         PrintWriter srcListWriter = null;
313         try {
314
315             /**
316              * Write sourcefiles and package names to a temporary file
317              * if requested.
318              */
319             if (useExternalFile) {
320                 if (tmpList == null) {
321                     tmpList = fileUtils.createTempFile("javadoc", "", null);
322                     tmpList.deleteOnExit();
323                     toExecute.createArgument()
324                         .setValue("@" + tmpList.getAbsolutePath());
325                 }
326                 srcListWriter = new PrintWriter(
327                                             new FileWriter(tmpList.getAbsolutePath(),
328                                                            true));
329             }
330
331             Enumeration e = packagesToDoc.elements();
332             while (e.hasMoreElements()) {
333                 String packageName = (String) e.nextElement();
334                 if (useExternalFile) {
335                     srcListWriter.println(packageName);
336                 } else {
337                     toExecute.createArgument().setValue(packageName);
338                 }
339             }
340
341             e = sourceFilesToDoc.elements();
342             while (e.hasMoreElements()) {
343                 SourceFile sf = (SourceFile) e.nextElement();
344                 String sourceFileName = sf.getFile().getAbsolutePath();
345                 if (useExternalFile) {
346                     if (javadoc4 && sourceFileName.indexOf(" ") > -1) {
347                         srcListWriter.println("\"" + sourceFileName + "\"");
348                     } else {
349                         srcListWriter.println(sourceFileName);
350                     }
351                 } else {
352                     toExecute.createArgument().setValue(sourceFileName);
353                 }
354             }
355
356         } catch (IOException e) {
357             tmpList.delete();
358             throw new BuildException("Error creating temporary file",
359                                      e, getLocation());
360         } finally {
361             if (srcListWriter != null) {
362                 srcListWriter.close();
363             }
364         }
365
366         if (packageList != null) {
367             toExecute.createArgument().setValue("@" + packageList);
368         }
369         log(toExecute.describeCommand(), Project.MSG_VERBOSE);
370
371         log("Javadoc execution", Project.MSG_INFO);
372
373         JavadocOutputStream out = new JavadocOutputStream(Project.MSG_INFO);
374         JavadocOutputStream err = new JavadocOutputStream(Project.MSG_WARN);
375         Execute exe = new Execute(new PumpStreamHandler(out, err));
376         exe.setAntRun(getProject());
377
378         /*
379          * No reason to change the working directory as all filenames and
380          * path components have been resolved already.
381          *
382          * Avoid problems with command line length in some environments.
383          */
384         exe.setWorkingDirectory(null);
385         try {
386             exe.setCommandline(toExecute.getCommandline());
387             int ret = exe.execute();
388             if (ret != 0 && failOnError) {
389                 throw new BuildException("Javadoc returned " + ret, getLocation());
390             }
391         } catch (IOException e) {
392             throw new BuildException("Javadoc failed: " + e, e, getLocation());
393         } finally {
394             if (tmpList != null) {
395                 tmpList.delete();
396                 tmpList = null;
397             }
```

```
398
399             out.logFlush();
400             err.logFlush();
401             try {
402                 out.close();
403                 err.close();
404             } catch (IOException e) {
405                 // ignore
406             }
407         }
408     }
```

D.2 Agar > sha1.c > SHA1Transform()

This function is detected as hard to read code by the outliers detection mechanism, see section 8.4.1 on page 146.

```
1   %       style=CInputStyle]
2   /*
3    * Hash a single 512-bit block. This is the core of the algorithm.
4    */
5   void
6   AG_SHA1Transform(Uint32 state[5], const Uint8 buffer[AG_SHA1_BLOCK_LENGTH])
7   {
8           Uint32 a, b, c, d, e;
9           Uint8 workspace[AG_SHA1_BLOCK_LENGTH];
10          typedef union {
11                  Uint8 c[64];
12                  Uint32 l[16];
13          } CHAR64LONG16;
14          CHAR64LONG16 *block = (CHAR64LONG16 *)workspace;
15
16          (void)memcpy(block, buffer, AG_SHA1_BLOCK_LENGTH);
17
18          /* Copy context->state[] to working vars */
19          a = state[0];
20          b = state[1];
21          c = state[2];
22          d = state[3];
23          e = state[4];
24
25          /* 4 rounds of 20 operations each. Loop unrolled. */
26          R0(a,b,c,d,e, 0); R0(e,a,b,c,d, 1); R0(d,e,a,b,c, 2); R0(c,d,e,a,b, 3);
27          R0(b,c,d,e,a, 4); R0(a,b,c,d,e, 5); R0(e,a,b,c,d, 6); R0(d,e,a,b,c, 7);
28          R0(c,d,e,a,b, 8); R0(b,c,d,e,a, 9); R0(a,b,c,d,e,10); R0(e,a,b,c,d,11);
29          R0(d,e,a,b,c,12); R0(c,d,e,a,b,13); R0(b,c,d,e,a,14); R0(a,b,c,d,e,15);
30          R1(e,a,b,c,d,16); R1(d,e,a,b,c,17); R1(c,d,e,a,b,18); R1(b,c,d,e,a,19);
31          R2(a,b,c,d,e,20); R2(e,a,b,c,d,21); R2(d,e,a,b,c,22); R2(c,d,e,a,b,23);
32          R2(b,c,d,e,a,24); R2(a,b,c,d,e,25); R2(e,a,b,c,d,26); R2(d,e,a,b,c,27);
33          R2(c,d,e,a,b,28); R2(b,c,d,e,a,29); R2(a,b,c,d,e,30); R2(e,a,b,c,d,31);
34          R2(d,e,a,b,c,32); R2(c,d,e,a,b,33); R2(b,c,d,e,a,34); R2(a,b,c,d,e,35);
35          R2(e,a,b,c,d,36); R2(d,e,a,b,c,37); R2(c,d,e,a,b,38); R2(b,c,d,e,a,39);
36          R3(a,b,c,d,e,40); R3(e,a,b,c,d,41); R3(d,e,a,b,c,42); R3(c,d,e,a,b,43);
37          R3(b,c,d,e,a,44); R3(a,b,c,d,e,45); R3(e,a,b,c,d,46); R3(d,e,a,b,c,47);
38          R3(c,d,e,a,b,48); R3(b,c,d,e,a,49); R3(a,b,c,d,e,50); R3(e,a,b,c,d,51);
39          R3(d,e,a,b,c,52); R3(c,d,e,a,b,53); R3(b,c,d,e,a,54); R3(a,b,c,d,e,55);
40          R3(e,a,b,c,d,56); R3(d,e,a,b,c,57); R3(c,d,e,a,b,58); R3(b,c,d,e,a,59);
41          R4(a,b,c,d,e,60); R4(e,a,b,c,d,61); R4(d,e,a,b,c,62); R4(c,d,e,a,b,63);
42          R4(b,c,d,e,a,64); R4(a,b,c,d,e,65); R4(e,a,b,c,d,66); R4(d,e,a,b,c,67);
43          R4(c,d,e,a,b,68); R4(b,c,d,e,a,69); R4(a,b,c,d,e,70); R4(e,a,b,c,d,71);
44          R4(d,e,a,b,c,72); R4(c,d,e,a,b,73); R4(b,c,d,e,a,74); R4(a,b,c,d,e,75);
45          R4(e,a,b,c,d,76); R4(d,e,a,b,c,77); R4(c,d,e,a,b,78); R4(b,c,d,e,a,79);
46
47          /* Add the working vars back into context.state[] */
48          state[0] += a;
49          state[1] += b;
50          state[2] += c;
51          state[3] += d;
52          state[4] += e;
53
54          /* Wipe variables */
55          a = b = c = d = e = 0;
56  }
```

D.3 R code for hard to read files

```
require('robustbase')
dim_cols <- ncol(project)
dim_rows <- nrow(project)
outs_all <- data.frame(matrix(FALSE, nrow = dim_rows, ncol = 4))
colnames(outs_all) <- c("Name", "SLOC", "Max", "Box")
outs_all$Name <- project$Name
outs_all$SLOC <- project$SLOC
outs_max <- data.frame(matrix(FALSE, nrow = dim_rows, ncol = dim_cols))
colnames(outs_max) <- names(project)
outs_max$Name <- project$Name
outs_max$SLOC <- project$SLOC
outs_box <- data.frame(matrix(FALSE, nrow = dim_rows, ncol = dim_cols))
colnames(outs_box) <- names(project)
outs_box$Name <- project$Name
outs_box$SLOC <- project$SLOC
for (i in 3:6) {
  seuil <- 0.85 * max(project[,i]);
  outs_max[project[,i] >= seuil, i] <- TRUE;
  myouts <- which(project[,i] %in% adjboxStats(project[,i])$out);
  outs_box[myouts, i] <- TRUE
}
for (i in 1:nrow(outs_max)) {
  nb <- 0;
  for (j in 3:6) {
    nb<-nb+(if(outs_max[i,j]==TRUE) 1 else 0) };
  outs_max[i,c("NB")] <- nb
}
max_max <- max(outs_max$NB)
outs_max$Result <- FALSE
outs_max[outs_max$NB > 0, c("Result")] <- TRUE
outs_all$Max <- outs_max$Result
for (i in 1:nrow(outs_box)) {
  nb <- 0;
  for (j in 3:6) {
    nb<-nb+(if(outs_box[i,j]==TRUE) 1 else 0) };
  outs_box[i,c("NB")] <- nb
}
max_box <- max(outs_box$NB)
outs_box$Result <- FALSE
outs_box[outs_box$NB == max_box, c("Result")] <- TRUE
outs_all$Box <- outs_box$Result
for (i in 1:nrow(outs_all)) {
  outs_all[i,c("ResultAND")] <-
    outs_max[i, c("Result")] &&
    outs_box[i, c("Result")];
  outs_all[i,c("ResultOR")] <-
    outs_max[i, c("Result")] ||
    outs_box[i, c("Result")];
```

```
}
outs <- outs_all[outs_all$SLOC > 10 & outs_all$ResultOR == TRUE, c("Name")]
rm(i, j, outs_max, outs_box, outs_all)
```

Index