UNIVERSITY OF TWENTE.


# On the Quality of Quality Models

MASTER THESIS

J.H. Hegeman

**Author**

Erik Hegeman
MSc. student Computer Science,
track Information Systems Engineering
Dept. of EEMCS, University of Twente
Student number 0086215
j.h.hegeman@alumnus.utwente.nl
+31 647848024

**Graduation Committee**

University members

Pascal van Eck
Information Systems chair, dept. of EEMCS, University of Twente
p.a.t.vaneck@utwente.nl
+31 534894648

Mariëlle Stoelinga
Formal Methods & Tools chair, dept. of EEMCS, University of Twente
m.a.i.stoelinga@utwente.nl
+31 534893773

External members

Nico Nijenhuis (assignment)
Managed IT Services, Info Support BV
nicon@infosupport.com
+31 647504890

Marco Pil (technical supervision)
Knowledge Center, Info Support BV
marcop@infosupport.com
+31 645478915

Process support

Barbara Engel
HRM, Info Support BV
barbarae@infosupport.com
+31 652671371

**Document availability** Unrestricted

**Revision number** 3.71

**Release date** July 4, 2011

“Measure what is measurable, and make measurable what is not so”

Galileo Galilei (1564-1642)

# Management Summary

This project investigates the usability of the SQALE method at software company Info Support BV. This method allows Info Support to measure the quality of Java and C# source code. We learned that SQALE, as implemented in the Sonar tool, provides a workable method to perform this quality assessment for projects of both languages.

Reason for Info Support to start this project was a wish to be able to assess the quality of customers' projects before the Managed IT Services department contracts customers for management services. The ability allows Info Support to decide which services can be delivered for which price.

The investigation was performed by correlating SQALE quality judgments of 9 Info Support managed software projects with two types of validation data:

- A survey was conducted, in which 11 experts (Info Support Managed IT Services employees) rated the sample projects they had knowledge of. This resulted in 22 project gradings.
- An investigation was conducted of the time spent on resolving incidents and problems in the sample projects in 2010. This resulted in a value (in hours) for each project, which we divided by the project size (in KLOC) for scaling. The resulting value is an overall quality indicator.

In a proof of concept, the Sonar tool and SQALE method were setup and used to assess the source code of the 9 sample projects. We calculated the correlation of the SQALE measurements with the expert opinions and financial quality indicators. We performed this analysis with an initial (default) SQALE configuration as well as with a calibrated SQALE configuration in which Info Support programming rules were used. The observed Pearson correlation values are displayed in Table 1. In all cases, we expected a value equal to or larger than +0.30.

| | Initial configuration | Calibrated |
|---|---|---|
| Sonar results vs. Survey results | +0.41 | +0.50 |
| Sonar results vs. Financial quality | +0.34 | +0.36 |

**Table 1 Main Pearson correlation coefficients: Sonar measurements vs. validation data**

This leads to the conclusion that in general, the methods provides quality measurements that are valid. A number of side nodes should be made:

1. The method configuration is very flexible, and configuring is therefore a difficult task. This thesis suggests a number of options to enhance the configuration and possibly further increase correlation with validation data;
2. A higher correlation coefficient was found to not always imply a better configuration quality. A starting-point configuration, consistent with the Endeavour rule set, and the use of a continuous improvement procedure is suggested.

The choice for the SQALE model and Sonar tool followed from a literature review and free search on the internet, respectively. The method allows Info Support to not only determine the overall quality, but also provides a rating for four important quality aspects, consistent with ISO 9126:

- Analyzability (referred to as Maintainability in SQALE): about the readability and understandability of application source code;
- Changeability: about the effort needed to change an application;

- Reliability: about the robustness of an application, amongst which exception handling, input error detection and thread handling;
- Testability: about the effort needed to test changes in a system.

As a by-product of this project, a number of recommendations for Info Support was formulated:

1. Use existing tools and do not attempt to imitate them by in-house development, as the latter option is more expensive and has more risks;
2. Keep an eye open for better tools since the SQALE method is supported by multiple tools and the tool market is young and developing; better options than Sonar may become available;
3. Use appropriate (virtual) hardware to run software analysis on. The analysis process requires quite a lot of computer power. Therefore, appropriate hardware is a requirement for effective use of the method, especially when the method becomes part of a non-incidental process (i.e. nightly builds). Bottlenecks are, in order of priority, CPU, disk I/O and RAM;
4. Integrate the method in PDC nightly builds: to support the alignment of the Professional Development Center and Managed IT Services Department, the method can be made part of the build cycle of PDC projects. This allows developers to better assure maintainability of projects that are being developed and are to become managed projects of MITS;
5. Sell Quality Assessment as a Service: quality assessments created by SQALE can be made 'SIG-compatible', which provides the business opportunity to Info Support to sell software source code quality assessments as a service to customers. Resulting quality judgments will be comparable to SIG audit results;
6. Assign method responsibility and authority: the use of SQALE requires some knowledge, and also an authority that is capable of taking non-trivial decisions mainly concerning the configuration of the quality model. Therefore, it is recommended to assign the responsibility for and authority of the use of SQALE at Info Support to a specific 'project owner';
7. Improve the MITS incident registration procedure. Details of this recommendation are confidential.

Additionally, two proposals for SQALE model extensions were formulated:

1. A proposal for introducing the concept of rule violation severeness in the quality model;
2. A proposal for a method to balance SQALE indexes over quality characteristics and languages.

To further elaborate upon the topic of the quality of quality models, a number of ideas that are deemed interesting but lay beyond the scope of this project were formulated:

1. Investigate the possibility to create a 'benchmarking repository' that can be used for the calibration of the quality model. Such a repository is also used by the Software Improvement Group. The question, in this case, would be if and how such a repository can contribute to the quality of the SQALE configuration calibration procedure, specifically for Info Support or in general.
2. Further Investigate the nature and characteristics of the mathematical relationship between Lines of Code in a software project and its number of Function Points.

# Preface

In 2004, I came to the University of Twente to study Business Information Technology. It would take some time for me to realize that at that point I didn't actually have any clue what it meant to be a University student. Although my studies went quite well and I passed all my first year courses, it was only at the end of my second year when I moved from Enter to Enschede and started to participate in extracurricular activities. I also started a second Bachelor's program in Computer Science. These steps turned out to have a major impact on my life as a student.

The next few years would become a heavy but pleasant mixture of courses, projects, meetings, reading and writing enormous amounts of documents and e-mails and of course more than occasional drinks. I managed to combine studying and other things I did quite well, so six and a half years after coming to the University, I found myself having finished all courses. It was time for me to graduate. I wanted to do a graduation assignment outside the University, since I didn't have any 'external' experience yet.

Directly after my final exams, I started my graduation project at Info Support. I decided not to move to the Veenendaal area, since travel times were acceptable and my social life was still in Enschede. It was hard for me to get used to spending days reading, writing and thinking, since I missed the action and variety of the life I had had for the past few years. I'm not someone who likes to work alone. It seemed to me to be quite inefficient to spent 40 hours a week on just one research project, especially since it involves a lot of thinking that I would normally do while doing 'actual work' that preferably provides some kind of tangible output at the end of the day. Fortunately, the project went quite well and I was able to more or less ignore this mismatch between the way of working and the way I prefer to work. I also was quite satisfied with the facilities and support during the project. These things allowed me to stay on schedule. After 3 months, I was able to draw the first conclusions from my research and to elaborate upon them, which announced a project phase that I found far more interesting than the data collection I had had to do in the previous two months. I enjoyed discussions about interpretation and implications of results and was able to finish the project soon afterwards.

Aside from capabilities and knowledge, the last few years have brought me lots of good memories. I especially enjoyed my board and committee functions, in which I got to know many different people and worked together with them to accomplish a diversity of things and also had a lot of fun. My advice to any student would be to choose things to do besides studying that are both pleasant and useful for the academic community and personal development.

Last but not least, I would like to thank my family and friends who have supported my throughout my studies. Without their help I would not have been able to accomplish the things I have in the last years. I also would like to thank the members of my graduation committee, who provided useful guidance and feedback, contributing to a graduation process that I experienced as quite smooth.

Erik Hegeman
Enschede, June 21, 2011

# Table of Contents

# 1. Introduction

In this project, we attempted to answer the research question "*how do Sonar SQALE quality assessment results of projects correlate to Info Support experiences and expectations?*" This introductory chapter provides information on the problem, scope and shows how this research question was formulated.

## 1.1. Problem Statement

The problem that was reason for Info Support - see background information section 2.1 for more information about the company - to formulate the initial research assignment, is a lack of insight in what is or is not a suitable way of dealing with automated software quality assessment.

The assignment was formulated by the Managed IT Services – 'MITS' – department of Info Support, which is responsible for delivering software management services (defined in paragraph 1.4.1) to customers as well as Info Support itself. Software, possibly developed outside Info Support, is at some point in time offered to MITS to be managed. To be able to assess this software and determine which management services can be offered for which price, a method is needed to qualify relevant quality aspects during the intake procedure. This does not mean, however, that the results of this research are only relevant for MITS itself. Monitoring of quality, specifically the maintainability aspect, is also relevant during the software development phase. The assessment method therefore is also intended to eventually be used at the Professional Development Center – 'PDC' – at Info Support, to increase the maintainability of software and allow for easy management after the initial development phase has finished, saving resources in the long term.

A specific requirement of the quality assessment method to be performed is that, for any given project, analysis results in at least an index for 'quality' and 'maintainability'. This index is a value on a certain scale, The quality index should incorporate at least a score for unit tests, unit test coverage, standard compliance and complexity. The maintainability index should incorporate at least analyzability, changeability, stability and testability. In the literature, maintainability is usually considered to be a specific aspect of quality, as will be explained later. A definition of quality will be chosen that incorporates this requirements but also conforms to ISO 9126, see definition paragraph 1.4.1.

## 1.2. Goal statement

The goal of this project follows from the problem statement, and is to develop and validate a method, applicable in the software management environment of Info Support, to automatically and quantitatively determine the quality and maintainability of software, given the software's source code. Research questions that need to be answered in order to reach this goal are defined in paragraph 1.4. The design of the research that will allow us to reach this goal is defined in chapter 3.

## 1.3. Scope

The description of the research presented thus far is still quite broad, therefore a number of constraints has been identified, i.e. limitations by impossibilities, and defined, i.e. by choices made to

scope the research. This way, the focus of this research has been narrowed down to something new, relevant and realizable. The following constraints apply:

- The set of software source code languages is limited to Java and C#, the languages used in software developed or maintained at Info Support. Tools used in this research need to support these languages.
- This research is not be about defining our own quality standard or model; we use the ISO 9126 standard and existing quality models, used by existing tools, as a foundation. This ISO-standard is explained in the Background section. Using this standard, we try to identify tools that conform to both the standard, the language support constraint, and are mature enough to be used in a production environment. Although this implies that we will not write our own software, we leave the possibility open that, as a result of this research, we will recommend to do so.
- Specifically, we focus on the 'maintainability' aspect of the ISO-standard. Reason for this is that this aspect is considered most relevant in the context of the research assignment as formulated by Info Support. The definition we use for 'quality' in this research expresses this and can be found in paragraph 1.4.1.
- The number of tools used in the Proof of Concept phase is limited to one, due to resource (time) constraints and the lack of available alternatives to the selected tool. Background information about tools can be found in paragraph 2.6.

## 1.4. Research Question

### 1.4.1. Definitions

To be able to understand and interpret the research questions and their relations as formulated in paragraph 1.4.2, we first need to define the terminology used in the formulation. The following terms are used in the research questions and throughout the rest of this thesis using a specific meaning defined in this section. Terms are presented in alphabetical order.

| Term | Definition |
|---|---|
| ***Expectations*** | Expectations are quantified formulations of what is expected to be the value of an element from the quality information tree (see 'Quality Information'), from a specific perspective. For example, the results of the second research phase (financial investigations) is referred to as *expectations*. |
| ***Experiences*** | Experiences are opinions of *Professionals* on the value of elements from the quality information tree. Specifically, we use *experiences* to refer to the survey results of the first phase of this research. |
| ***Professionals*** | Professionals are employees of Info Support that are directly involved in software management, i.e. employees of the Managed IT services department. |
| ***Quality information*** | Quality information is quantitatively expressed information about the quality of software source code, expressed in quality indicators values calculated from source code metric measurements using a quality model (see 'Quality model'). Quality indicators used in this research conform to ISO 9126 and are a sub tree of the indicators defined in this standard, defined in consultation with Info |

| | |
|---|---|
| | Support supervision. Elements from the tree are also referred to as 'quality aspects'.<br><br>○ Maintainability<br>▪ Analyzability<br>▪ Changeability<br>▪ Stability<br>▪ Testability<br>• Unit Test run result<br>• Unit Test Coverage<br>• Complexity<br>▪ Maintainability Compliance<br><br>('maintainability compliance' indicates the extent to which software meets programming guidelines and conventions. This is also referred to as 'standard compliance') |
| ***Quality model*** | A Quality model is a mathematical model that determines the value of quality indicators (see 'Quality Information'), by mapping and aggregating source code measurements to quality indicators. Quality models can implement a standard, for example the ISO 9126 standard. |
| ***Software Management*** | The set of activities conducted by the Info Support Managed IT Services Department, which includes, for example, hosting, guaranteeing availability, incident management, and performing updates, repairs and modifications. |

**Table 2 Research Question Terminology Table**

### 1.4.2. Question formulation

Based upon the research goal and scope, the main research question was initially formulated as follows:

*To which extend can software tools, incorporating quality models, provide quality information that matches Info Support experiences and expectations?*

1. *How is software quality being experienced by Info Support professionals?*
2. *How can software quality be expressed in financial terms, given historic data?*
3. *How can software tools be used to assess software source code quality?*

The main research question contains the phrase 'to which extend'. To make the answer measurable, we use quantitative analysis of the result of all sub questions. We now flash-forward to the theoretical background and research design sections and narrow down the research questions. We use the following information:

- The software tool selection phase identified Sonar to be the tool of choice (2.6.2)
- The quality model selection phase identified SQALE to be the model of choice (2.4.6)
- The 'extend of the match' is calculated by correlations (3.5.1)
- In the survey to be conducted we ask experts to rate projects on SQALE characteristics (3.2)
- In the financial investigation phase we determine a financial quality indicator values (3.3)
- Hypotheses about the correlations between the results of the sub questions are formulated in research design section 3.5

This information leads to the following reformulation:

***How do Sonar SQALE quality assessment results of projects correlate to Info Support experiences and expectations?***

1. ***How do Info Support experts rate the sample projects on relevant SQALE characteristics?***
2. ***What is the financial quality, expressed as hours/KLOC, of the sample projects?***
3. ***How are the sample projects rated by a Sonar SQALE setup?***
4. ***Which methods to improve the quality of the quality model configuration exist?***

## 1.5. Document Structure

Globally, this thesis has the following structure:

- Background information about Info Support, quality models & tools is provided (ch. 2, p19);
- The research design is defined (ch. 3, p43);
- Results of the process of gaining validation data as well as the validation data itself are reported (ch. 4 and 5, p61 onwards) ;
- Results of the phase in which we setup and configured the tooling and attempted to use it to assess the quality of projects are described (ch. 6, p64);
- An analysis if the findings is presented (ch. 7, p74);
- Recommendations for Info Support are presented (ch. 8, p88);
- Discussion topics and suggestions for future research are provided (chapters 9 and 10, p91);
- Conclusions are presented (ch. 11, p101).

## 1.6. Related Work

This section mentions some other relevant work performed in the area of software quality assessment and assessment methodology validation.

The IEEE attempted to empirically validate the suitability of object-oriented design metrics as quality indicators (Basili 1996) by assessing eight comparable projects. Examples of metrics used in this context are the inheritance tree depth, methods per class, and number of children of a class, i.e. all metrics related to object oriented design. Specifically, the set of metrics from the Chidamber & Kemerer 'metrics suite' are used (Chidamber 1994). Validation data consists of error data from the testing phase of the applications. For each metric, an hypothesis is formulated about the relationship between the metric value and 'quality', where more errors means less quality. This hypothesis was found to be true in five out of six cases; results for the sixth case were insignificant. This results demonstrates that most metrics from (Chidamber 1994) can be validated to provide information that is an indication of quality.

This study is complementary to (Li 1993) in which the same metrics are used to estimate the maintenance frequency of classes in a system. Li concludes that there is a strong relationship between metrics and maintenance effort in object oriented systems and that maintenance effort can be predicted from combinations of metrics collected from source code. The conclusions of both studies are consistent.

Although no earlier work on the validation of the SQALE method was found, the underlying concept of remediation cost, or technical debt (Cunningham 1992), is often discussed. Recently, the second international workshop on managing technical debt has been held in Honolulu, Hawaii. The program can be found at the website (Techdebt 2011), proceedings are not yet available at time of writing of this thesis. Work is presented about, for example, prioritizing design debt investment opportunities, models for economic trade-off decision making and an empirical model of technical debt and interest.

An empirical study of quality models in Object Oriented systems is described in (Briand 2002). This study lists a number of 'correlational studies' in which metric values of a some data set are validated by, for example, numbers of defects or expert opinions. Usually, these studies use a 'set of metrics' rather than an actual quality model. Data sets usually consists of a very limited number of projects, often just one, which is identified as a shortcoming of many of the projects. Some of the studies use expert opinions to gain validation data. For example, (Chen 1993) uses expert opinions to assess the validity of a single newly invented metric.

The use of quality standards is not limited to software engineering, but applied in other sectors as well. For example, organizational procedures can be 'ISO 9000'-certified (Guler 2002) and philosophies like 'Total Quality Management' focus on the continuous improvement of both products and processes (Daft 2003) . Also, in the food industry, many quality standards are used, sometimes inspired by incidents impacting public safety (Trienekens 2008).

## 1.7. Conclusion

We conclude this chapter by providing a schematic overview of the project. The diagram in Figure 1 shows the three research phases (left, right and bottom-center), as well as the conclusion phase (top-center). These phases are separated by striped lines. Phases I through IV were carried out consecutively. In each phase, a number of entities, indicated by boxes, perform actions or provide information, indicated by arrows. Red questions are answered in the research design. The goal is to answer the blue questions, which are the questions of this research. Arrows indicate flows of information, while boxes indicate sources and processes.

The research design, that explains the details of the overview displayed in this figure, is fully described in chapter 3.

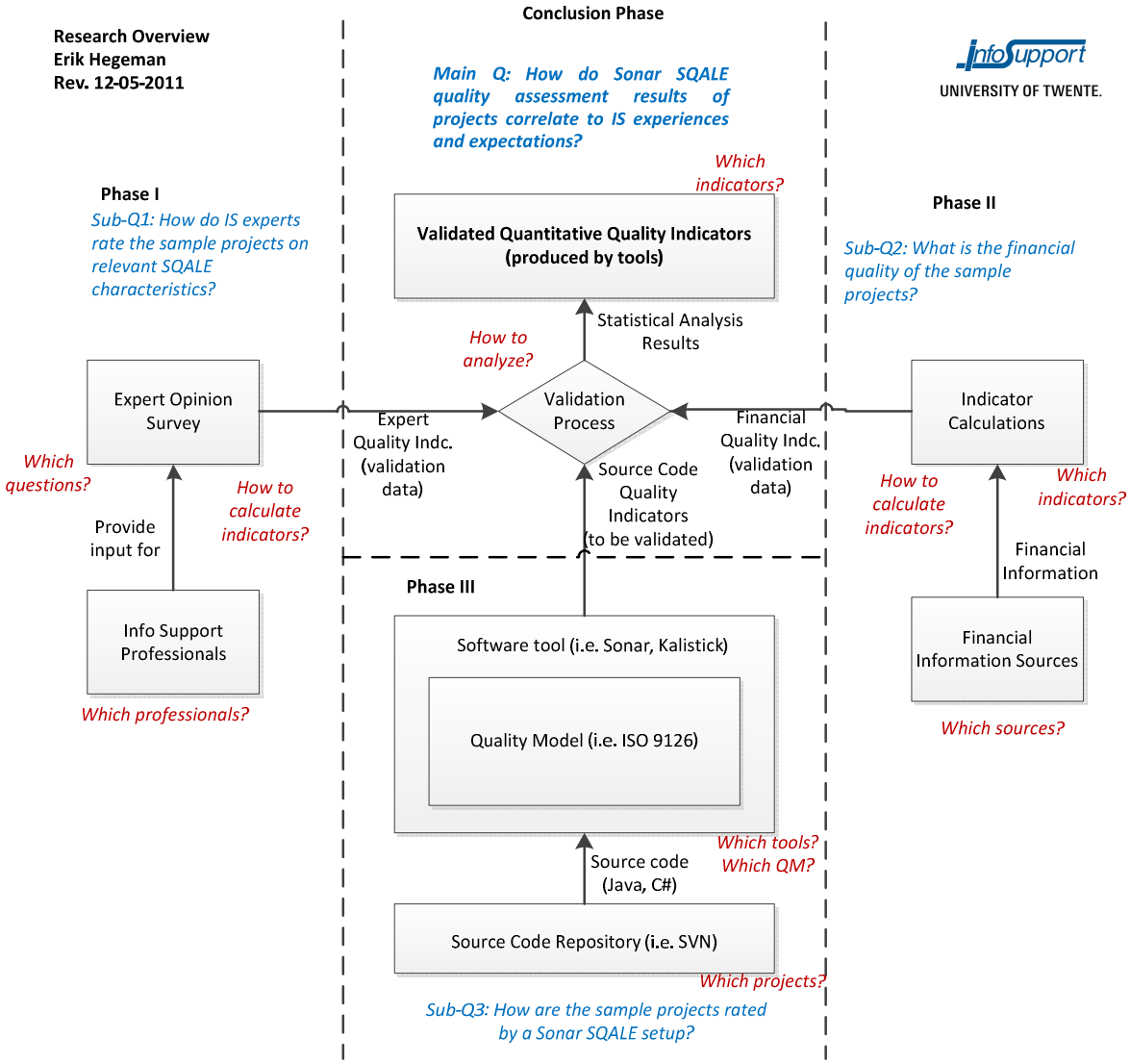



**Figure 1 Research Design Overview**

# 2. Theoretical Background

This chapter provides to the reader background information that is relevant in the context of this research. The information is obtained from a literature review as well as the documentation of various tools and quality models available.

The definition of the term 'quality' can be an item of discussion. Although it is widely recognized that context-specific definitions can be useful, the wish to have more general definitions remains unfulfilled (Jørgenson, 1999). The search for a universal definition has not succeeded, and it is claimed that such a global definition does not exist (Reeves, 1994). Also, different stakeholders may have different opinions on what defines quality of a given type of artifact, potentially obscuring communications. That is why we must make unambiguous context-specific definitions.

In an Information Technology context, the need for quality is broadly acknowledged. Software development projects still often encounter cost overruns and/or time overruns, and often do not conform to specifications or expectations (i.e. Standish, 2001). According to the Project Management Diamond, as displayed in Figure 2. (Haughey 2010), steering factors time, cost, scope and quality should be balanced to make an information system live up to its expectations. But balancing is only possible if we can measure all four factors, including quality.

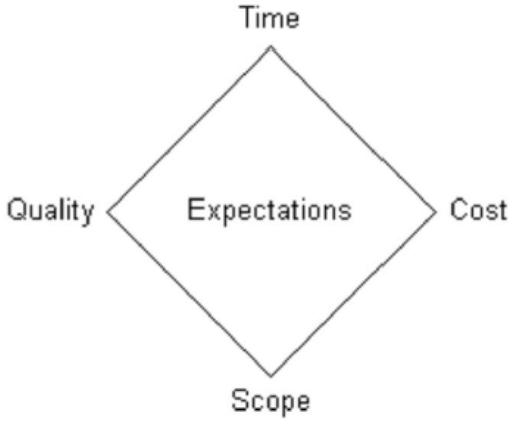



Figure 2 Project Management Diamond

An IT-specific definition of quality has been developed in the ISO 9126 standard. This standard identifies a number of high-level quality characteristics, namely Functionality, Reliability, Usability, Efficiency, Maintainability and Portability (Jung e.a. 2004) and also software metrics that can contribute to determining a score for these aspects. Chapter 2, on theoretical background information, further elaborates upon ISO 9126.

An important asset of a software project is its source code. The source code of a software project determines what the software actually does, and how, and therefore plays a crucial role in the realization of quality (Luijten 2010).

## 2.1. Info Support Context

### 2.1.1. On Software Engineering and Management at Info Support

This research is conducted at Info Support BV in Veenendaal, the Netherlands. Info Support develops and manages software for customers in four sectors: healthcare, industry & trading, finance and the public sector. Distinguishing characteristics of Info Support, amongst others, are very strictly defined and knowledge-intensive processes and procedures, used to ensure solidness and quality of delivered products and services. Methods used for this purpose are, for example, test-driven development, SCRUM (Abrahamsson 2002), code conventions and version control check-in policies. The internally developed software development methodology is called 'Endeavour', which is maintained by the Professional Development Center (PDC). The Managed IT Services (MITS) department takes care of managing production software, developed either internally or externally.

More information on Info Support can be found at its website (Info Support, 2011). Figure 3 shows a simplified overview of PDC and MITS relation. In the application lifecycle, Info Support can either do only development, only management, or (preferably) both. Cases in which Info Support does neither development or management are not relevant.

| | Internal development | External development |
|---|---|---|
| Internal management | Development & mgmt (preferred) | Management only |
| External management | Development only | Irrelevant |

Figure 3 Info Support PDC and MITS relations

### 2.1.2. Earlier Work at Info Support

The problem defined in paragraph 1.1 has already led to some research initiatives that have been conducted at Info Sport. In earlier research, it has been attempted to identify and test maintainability metrics, but this research did not provide results that could be used in practice at Info Support (Woolderink 2007). Another, non-academic but relevant, project is a quickscan of the possibility to use the Sonar tool for quality monitoring and enhancement. Results of this quickscan were promising, leading to a constraint on this research that the Sonar tool is to be one of the tools to be looked at. This requirement is taken into account in paragraph 2.6.2 which identifies tools that can be used to set up the proof of concept as defined in research design section 3.4.

## 2.2. Quality Model Terminology

A number of essential concepts need to be described before additional background information is introduced. These concepts play a crucial role in quality models. First, *metrics* measure a specific aspect of source code (Fenton 1997). Examples are cyclomatic complexity of a method, unit test coverage of a class and the depth of a set of nested if-statements. Input to metrics is source code, output is a value (i.e. numerical, percentage or Boolean). Second, *Quality Models* are mathematical models that translate *metric* values into higher-level quality indicators. Since the late seventies, different models have been developed, as will be described in paragraphs following. In some cases, the quality model is also referred to as 'method'. In 1991, the International Standards Organization developed a *standard*, the original ISO 9126, later to be replaced by ISO 25000, as described in the next paragraph. This standard was inspired by older quality models (i.e. McCal). Later models (i.e. SQALE) may implement this standard. Software *tools* that assess quality implement a quality model.

## 2.3. On the ISO 25000 standard

In this research, we use the ISO 25000 standard as a foundation for software quality measurement. Defined in 2005, this standard is a follow-up of two relevant predecessor standards, namely ISO 9126 on software product quality, and

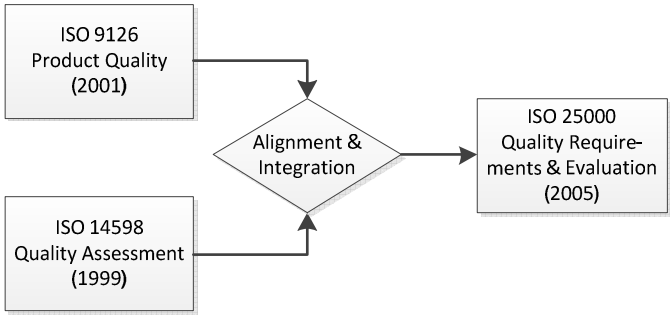



Figure 4 Integration of ISO 9126 and ISO14598 into ISO25000

ISO 14598 on the software quality assessment process. Figure 4 shows a visual representation of the integration and alignment process that led to the definition of ISO 25000 (Zubrow 2004).

### 2.3.1. The ISO 9126 component

Introduced in 1991 and developed by the International Standards Organization and the International Electrical technical Commission, the ISO/IEC 9126 standard is an international standard for defining and measuring software product quality. This original version of the standard includes six high level quality characteristics and their definition (Jung 2004). It was claimed that these six quality characteristics were sufficient to represent any aspect of software quality (Burris, 2004). Sub characteristics and metrics, however, were not part of the standard, making the standard difficult to apply.

The standard has evolved over the years, which has led to the development of ISO/IEC 9126-1 through-4 (written in 2001-2004)[1]. The standard now includes not only the original quality model (9126-1), but also external and internal metrics (9126-2 and 9126-3) and metrics for quality of use (9126-4) (Jung 2004). External metrics assess the behavior of software in a simulated environment from an interface-view, while internal metrics do not rely on execution but look at the insides of software, i.e. the source code. Usability metrics (referred to as 'Quality-of-use metrics') assess software from a user point-of-view. For the purpose of this research, both the general standard and internal metrics are relevant. We will see that these are the aspects of the ISO standard incorporated in quality models used in software tools, since this allows for automated code quality assessment.

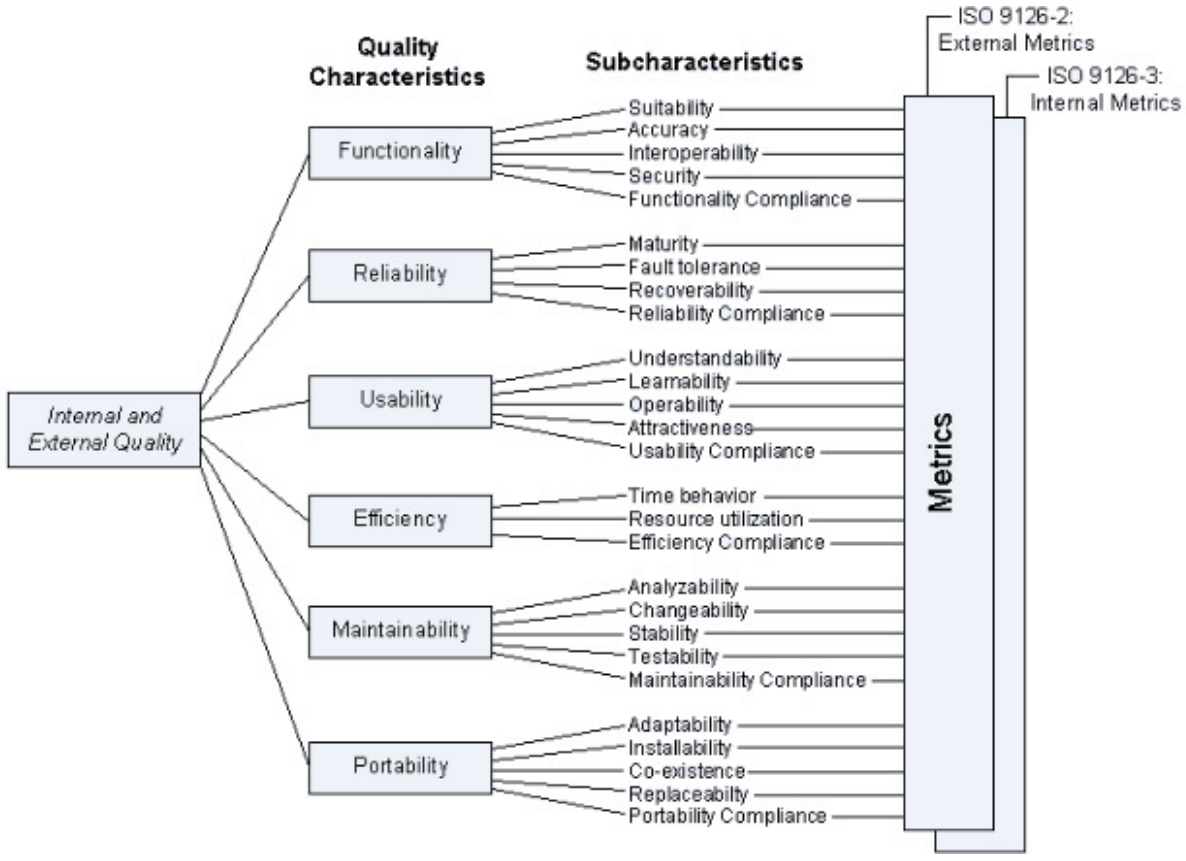



Figure 5 Hierarchical view of the ISO/IEC 9126 quality model

[1] The full ISO standards documentation is available from www.iso.org

The standard proposed a hierarchy, denoting the term 'quality' as the root of the hierarchy three and the six quality characteristics as the first level nodes. Each of these nodes is split up in several sub-characteristics. Internal and external metrics can be used to determine a value for a sub characteristics. Figure 5 gives a visual representation of this tree structure. As stated in the scope paragraph 1.3, in this research we focus on the 'maintainability' aspect of quality. The maintainability subtree of the ISO standard is consistent with our definition of quality from section 1.4.1.

For the aspects we look at, we provide a brief description of their meaning (Tavaf 2010):

- Maintainability is the ability to find and fix a fault in a software system.
    - Analyzability characterizes the ability to identify the root cause of a failure within the software.
    - Changeability characterizes the amount of effort needed to change a system.
    - Testability characterizes the effort needed to verify (i.e. test) a system change.
        - Unit test run results: the amount of errors encountered while running the unit tests of a system.
        - Unit test coverage: the percentage of source code covered by the unit tests.
        - Complexity: complexity (i.e. cyclomatic complexity) of the source code. This is related to testability, because more complex source code is more difficult to test and therefore has a negative impact on testability.
    - Stability characterizes the sensitivity to change of a given system that is the negative impact that may be caused by system changes.
    - Compliance: Where appropriate certain industry or government laws and guidelines need to be complied with. This sub characteristic addresses the compliant capability of software.

Note that these definitions are not necessarily consistent with the characteristics descriptions of the SQALE quality model (2.4.6), which will be used in the Proof of Concept of this project (3.4). They can, however, be mapped, at will also be shown (6.1).

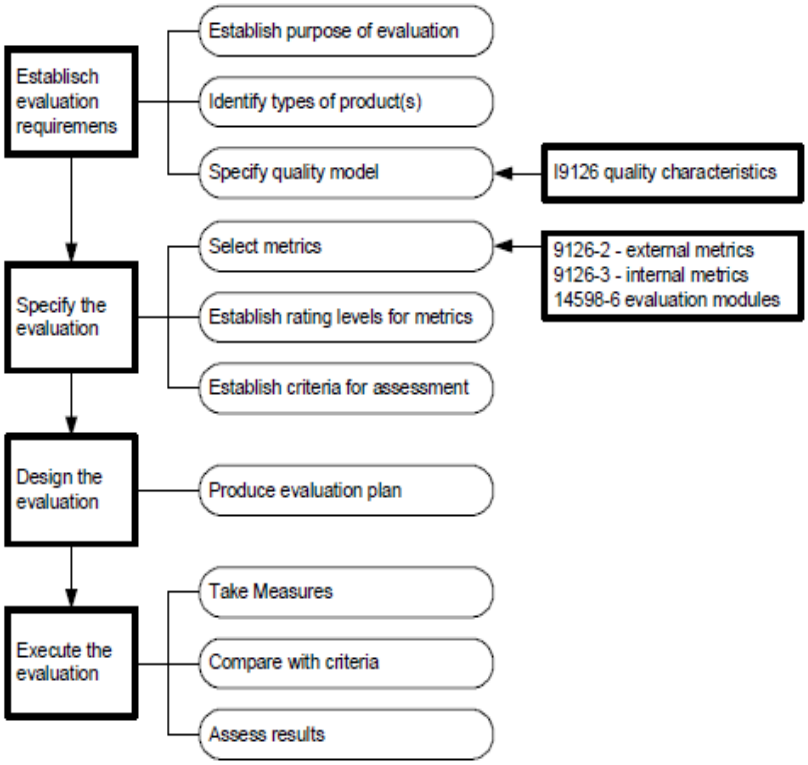



Figure 6 ISO 14598 Quality Evaluation Model

### 2.3.2. The ISO 14598 component

The ISO 14598 component does not focus on software quality, but on the process of assessing quality. This research project incorporates an assessment structure that is based upon ISO 14598. The standard provides an abstract proces of how a software quality assessment process should look like. The process is depicted in Figure 6 (source: (Gruber 2007)). The relationship between ISO 14598 and ISO9126 is that the first can be applied to the second, meaning that the process is used to execute the model. Note that the process is generic in the sense that it can be applied to any quality model. Also, other processes may provide suitable ways to execute the model.

## 2.4. On Quality Models

As mentioned, the ISO 25000 standard did not just emerge. Quality models have been defined for decades. The purpose of a quality model, in this context, is to transform metrics to high-level quality indicators. Usually, the quality model allows for some tweaking to conform to business needs. Figure 7 displays a simple overview of the concept of a quality model. This paragraph provides a short description of a number of models and their relation to the ISO standard.

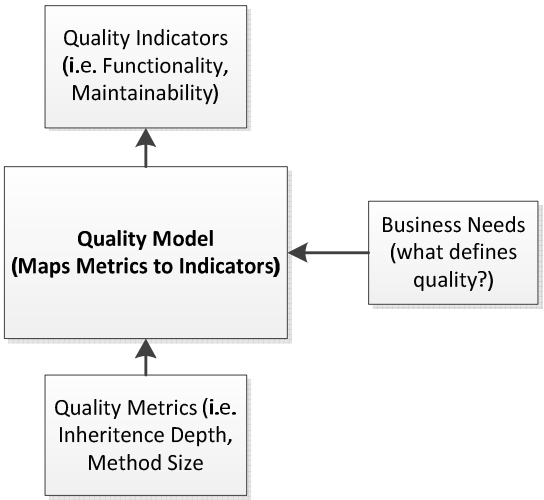



Figure 7 Quality Model Concept. Arrows show input and output

### 2.4.1. McCal Model

As a foundation for what would later become the ISO standard, McCal introduced the concept of the hierarchical combination of metrics into higher-level quality attributes as early is 1978 (McCal 1978). Combination is performed by addition, where each metric value is associated with a regression coefficient, based upon the established importance of the metric value.

Consider the situation in which we have four metrics with the following regression coefficients:

- Cyclomatic complexity: -0.5
- Program length: -0.1
- If-statement nesting depth -0.5

The model allows us to define negative coefficients for metrics for which a higher measurement value means lower quality. Suppose that the software artifact under consideration has a cyclomatic complexity of 30 paths, a program length of 660 lines and a maximum if-statement nesting depth of 5.

Suppose this set of metrics is attached to one quality factor, namely 'maintainability'. The maintainability factor would be -0.5*30 + -0.1 * 660 – 0.5*5 = -83,5. An important limitation of the model is that it is only suitable for metric-quality correlations with a linear relation.

Elements of the McCal model are still visible in today's ISO standard. For example, McCal introduces, amongst others, maintainability, testability, portability and reusability as high-level quality indicators.

### 2.4.2. Boehm Model

The Boehm model (Boehm, 1999), also referred to as COQUALMO ('Constructive Quality Model'), focuses on the introduction and removal of defects in a software system, where in a source code context a defect is a programming error leading to incorrect software behavior. Goal of the model is to predict the number of residual defects per unit of size, i.e. thousands of source lines of code or function points. An practical description of the use of the model can be found in (Madachy 2008). The mathematics in the model are as follows. Consider the following formula:

$$\text{Estimated Cost of Introduced Coding Defects} = A * \text{Size}^b * \prod_{i=1}^{21} D_i$$

In which:

- A is the scalar of coding defects, used to be able to combine the result with requirements and design defects (which are also part of the model, consider A=1 for this example)
- Size is the size of the project in KLOC
- b is used to account for economies of scale if needed. No proof for a value other than 1 is found, so by default it is set to 1 (Boehm 2000).
- $D_i$ is a Defect Introduction Driver value, of which 21 have been defined in the model. These drivers identify reasons for defects to occur.

Example of drivers are team cohesion en programmer capability. This means that source code quality is determined based on information that is not necessarily directly source-code related. This paradigm is mainly applicable for development processes.

A predecessor of COQUALMO, called COCOMO ('Constructive Cost Model'), is thought to have been the most cited, best known and most plausible of all traditional cost prediction models (Atterzadeh 2010). In the late nineties, however, the model no longer fitted the modern development environment and was succeeded by COCOMO II. Reasons for this were the need for support for application composition (i.e. object orientation) and the need to be able to determine costs in earlier stages of the development process, The focus on quality was introduced in COQUALMO, which can be seen as an extension to COCOMO II, which was still mainly a cost model. The model is not explicitly related to the ISO standard.

### 2.4.3. Dromey Model

As an implementation of the original 1991 ISO9126 standard, Dromey (Dromey 1995) introduced the concept of 'Quality-carrying properties' of source code to provide a definition of what the high-level quality indicators from the original ISO standard mean in a practical situation. A quality-carrying property is associated with one or more high-level indicators from the ISO standard and with specific aspects of software source code, meaning that building these properties into software contributes to reaching a high-level quality indicator. Dromey believes that it is impossible to build high-level quality attributes into products. Instead, developers must build components that have properties that result in the manifestation of quality (Kitchenham 1996). This can be considered a bottom-up approach. Consider Figure 8, a redraw of an image from (Dromey 1995). The source code artifact of type 'expression' can, in the model, have four 'quality carrying properties'. How it is determined whether or not the expression carries these

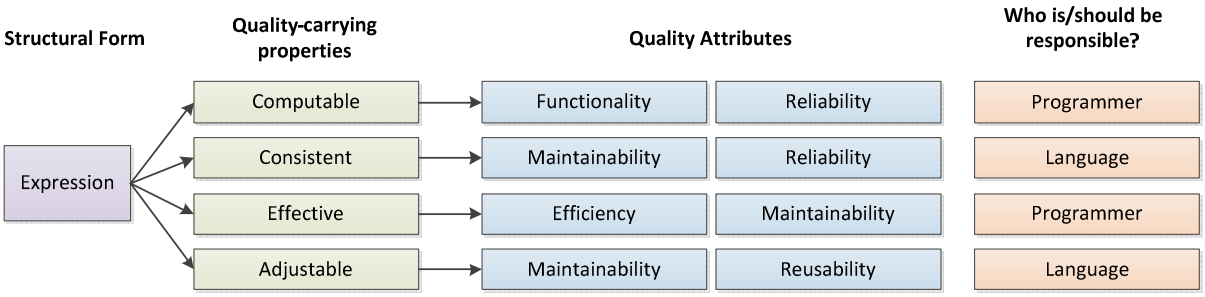



Figure 8 Dromey hierarchy example

properties is left to the user of the model, but the outcome is binary. This means that an expression either has or does not have each of the quality-carrying properties. The properties are mapped to the ISO-based quality characteristics and sub characteristics. The amount of properties artifacts have and have not are an indicator for quality. Also note that properties are not necessarily the responsibility of the programmer; some properties are inherent to design choices such as the programming languages.

### 2.4.4. SIG model

#### *2.4.4.1 General model description*

The Software Improvement Group (SIG) uses an ISO9126-based quality model, with a focus on maintainability, for professional software quality assessments (Heitlager 2007). In this model, a maintainability index on a scale from 1 to 5 is calculated using the four aspects of analyzability, changeability, stability and testability, consistent with the sub characteristics of maintainability of ISO

9126 as well as the definition of quality in this research. The SIG model itself it public, but it relies on a large 'benchmarking repository' for calibration. For this purpose of this research, a SIG audit has been provided to the author of this work as reference material. Since this is a confidential report, we cannot go into details. The way in which assessment results are presented indicate that the method used by SIG is based upon quality models and tool results that are very similar to the ones used in this research.

#### 2.4.4.2 The Sonar SIG Maintainability Model Plugin

Note that for the Sonar tool (see paragraph 2.6.2.2), a SIG plugin exists (Sonar SIG 2011). This plugin implements a subset of the SIG quality model using five metrics: Lines of Code, Duplication, Coverage, Complexity and Unit Size with a 5-point scale to calculate, using a many-to-many table (see Figure 9) averages the four quality indicators analyzability, changeability, stability and testability, which are the aspects of maintainability as defined in ISO 9126. This plugin does not fully imitate the SIG model, as it has no calibration functionality and also does not allow the user to drilldown into origins of problems.

| | Volume | Complexity | Duplications | Unit size | Unit tests |
|---|---|---|---|---|---|
| analysability | ✓ | | ✓ | ✓ | ✓ |
| changeability | | ✓ | ✓ | | |
| stability | | | | | ✓ |
| testability | | ✓ | | ✓ | ✓ |

Figure 9 Sonar SIG model plugin: metric-indicator mappings

#### 2.4.4.3 The SIG paradigm

A number of relevant aspects of the ideas behind the SIG quality model should be emphasized. This information was largely obtained from a visit to SIG during the course of this project, where Joost Visser, head of research, presented the model.

- The set of metrics used in the SIG model is limited. SIG states that metrics should not overlap (i.e. each used metrics should identify a different aspect of source code quality) and should be non-controversial, meaning that developers should know what to do in case a metric value does not meet requirements.

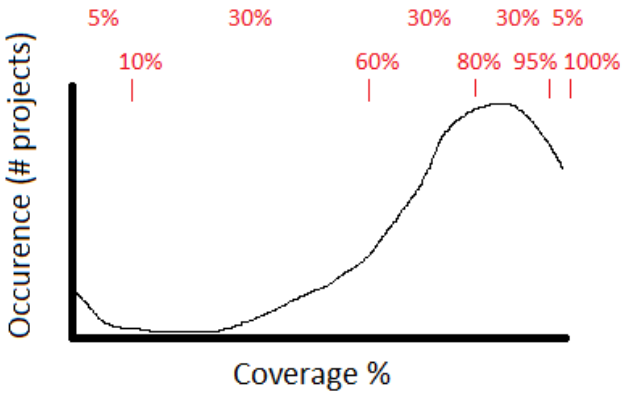



Figure 10 Benchmark example – Unit Test Coverage distribution

- SIG explicitly does not define what is 'good' and 'bad', but only defines 'better' and 'worse'. This is accomplished by a benchmarking principle. Consider the following example: suppose 'unit test line coverage' is a used metric. Also, supposed that we want to assign the worst 5% of projects a '1 star' rating and the best 5% of projects a '5 star' rating, and split up to 90% of projects in between into 3 segments of 30% with a 2, 3 or 4 star rating. Also, we assume that

a larger coverage value is always better than a lower one. Figure 10 shows an example distribution of coverage percentage over projects (the artifact size may be a class or a method as well). The two scales above the image show the five selected sections of the x value range (upper) and the x values at the boundaries of the sections (lower). We see that 5% of projects have a coverage level of 10% or lower; the next 30% has a coverage between 10% and 60%, etcetera. This results in the following rankings:

  - Rating 1: <= 10% coverage
  - Rating 2: 10% < coverage <= 60%
  - Rating 3: 60% < coverage <= 80%
  - Rating 4: 80% < coverage <= 95%
  - Rating 5: coverage >95%

- A major advantage of this approach is that a quality judgment is always relative to other systems and no explicit definition of good and bad is needed. Also, by adding newly assessed projects to the repository and removing old ones (i.e. older than 3 years), the calibration process automatically stays up to date. In practice, a slight but structural increase of software quality in general is observed by this method. Major disadvantage of the method is that it needs a large database of sample projects for calibration, which makes the model less usable in situations where this 'benchmarking repository' is not available. More background information on the benchmarking principle is available from (Correia 2008). Aggregation of ratings per metric into higher-value ratings is performed by a method that is similar to, but somewhat more complex than displayed in Figure 9; i.e. a many-to-many mapping from metric ratings to ISO factors. For each occupied cell in the matrix, a weight factor is used.
- Validation of the model configuration was found to be difficult due to the lack of sources of validation data. A positive correlation, however, was established between software artifact quality as measured by SIG and the time needed to repair incidents in these projects (Luijten 2010).

### 2.4.5. Qualixo Model

The Qualixo quality model (Laval 2008) defines four elements with different granularity levels:

- *Metrics* are low-level measurements computed directly from source code, consistent with the definition in the terminology section 2.2.
- *Practices* assess one quality aspect of a model and are associated with one or more *metrics*. The value for a *practice* always lies between 0 and 3, where 3 is the best possible value and 0 is the worst possible value. These values are obtained by weighting and scaling associated *metric* values, and scaling this down to the 0-3 range. This downscaling can take place either by a continuous function or by mapping discrete values 0 through 3 to a range of metric values. As an example, consider a source code artifact with the following metrics, all associated to one *practice*:

| Metric | Value | Mapping function | Scaled value | Metric Weight |
|---|---|---|---|---|
| **Cyclomatic Complexity** | 7 | <=8 → 3<br>9-12 → 2<br>13-16 → 1<br>>=17 → 0 | 3 | 3 |
| **Unit Test Coverage** | 80% | 3 * value | 2.4 | 5 |
| **Class Length** | 426 lines | <=100 → 3<br>101-300 → 2<br>301-500 → 1<br>501+ → 0 | 1 | 1 |
| **If statement nesting depth** | 4 | <=2 → 3<br>3-4 → 2<br>5-6 → 1<br>7+ → 0 | 2 | 2 |

Tabel 3 Qualixo model calculation example: metrics to practices

The 'Mapping' column defines how metric values are translated to the [0..3]-scale. Unit Test Coverage is translated by a continuous function, while the other metrics are translated by a discrete mapping. The 'Scaled value' column shows the resulting quality judgment for each metric. The total score for this *practice* is the weighted average of scaled values:

(3x3 + 2.4x5 + 1x1 + 2*2) / (3+5+1+2) = 2.36

- *Criteria* assess one principle of software quality by taking a weighted average of a nonempty set of practices. For each practice, the value is calculated as described earlier, and a weight is used to indicate the importance of the practice in the set. This calculation again results in a value between 0 and 3.
- *Factors* represent the highest quality assessment and are again computed over a set of weighted criteria. Figure 11 gives an overview of the Qualixo quality model.

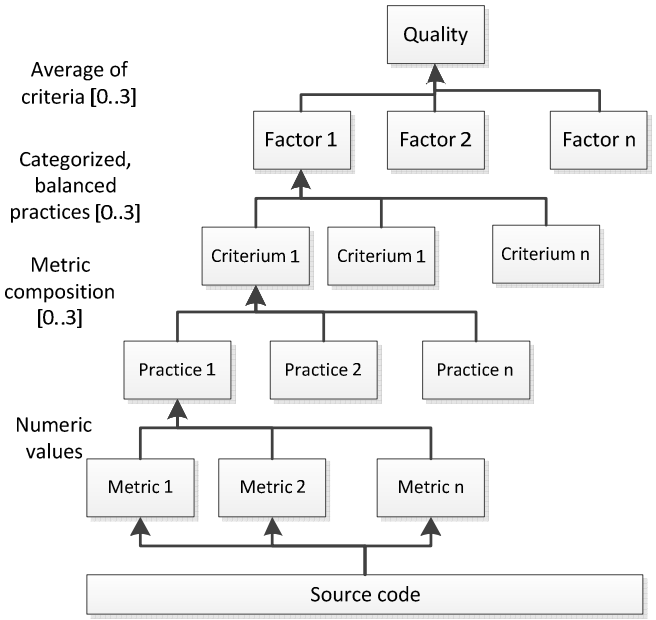



Figure 11 Qualixo Quality Model

A fixed interpretation is set for the scores in the [0-3] scale, as displayed in Table 4.

| Qualixo Score | Interpretation |
|---|---|
| [0-1) | Failure in meeting the quality objective |
| [1-2) | Achieved with reserve |
| [2-3] | Achieved |

Table 4 Qualixo Score Interpretations

The set of Factors for the Qualixo model is fixed. These factors are Functional Capacity, Architecture, Maintainability, Capacity to Evolve, Capacity to re-use and Reliability. This partially matches the ISO 9126 set as van be seen in the comparison table at the end of this paragraph. A drawback of the model is that it is quite abstract and does not provide guidelines for the weighing that occurs on multiple levels on the model.

### 2.4.6. SQALE Model

The SQALE quality model ('Software Quality Assessment based on Life Cycle Expectations') (Sqale 2011)(Letouzey 2009), also referred to as 'method' instead of model, is a newer, language and tool independent method for quality assessment. It can be applied to different types of artifcats, and software source code is one of them. It is based upon a 'remediation cost' paradigm, in which high-level quality indicator values indicate the amount of time or financial resources needed to repair all issues. These 'issues' are violations of *rules*, which are pre-defined metrics with metric value threshold that define what is allowed and what is not (see 2.4.6.2) Remediation costs are compared to 'total costs', which are defined as the total estimated amount of time or financial resources invested in developing the project. Total costs may be calculated by multiplying the size of a project (i.e. in thousands of lines of source code) by the estimated average amount of hours needed to develop one thousand lines of code. A discussion about this paradigm can be found in section 9.4. Since it is incorporated in many tools today, we will elaborate on this model. First, we will elaborate on the structure of the model. Second, we will provide an example. Back.

#### *2.4.6.1 Model Structure*

The model defines three levels of hierarchy:

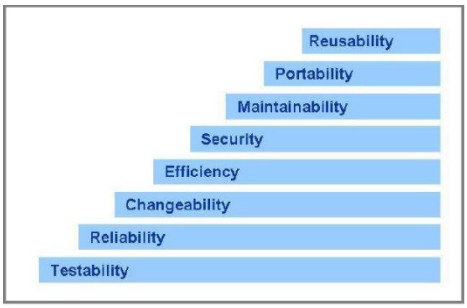



Figure 12 SQALE quailty characteristics

- The upper level in the hierarchy are the SQALE Characteristics, as shown in Figure 12, which are a 'projection of ISO 9126 model on the chronology of a software application's lifecycle'. This means that the characteristics of the model have a chronological sequence of importance, and failure on one level implies failure on all levels above, since each level conforms to a phase in the software lifecycle. For example, if maintainability is low, portability and reusability are also compromised, since these characteristics depend on maintainability. Depending on the context in which the model is used, some characteristics may be left out. Efficiency, Security, Maintainability, Portability and Reusability are considered 'optional' in the definition of the model.

- Below the top level is the level of sub characteristics. These sub characteristics have a lower level of abstraction than characteristics. Each sub characteristics is attached to one characteristic, namely the lowest one applicable in the 'staircase' as depicted in Figure 12, consistent with the characteristics concept. There is no fixed set of sub characteristics to use, and the SQALE user is free to choose his or her own sub characteristics and their mapping to characteristics. Table 5 show this mapping as implemented as the default model configuration in the Sonar SQALE plugin. Note that the 'Reusability' characteristic is left out of this implementation.
- Source code requirements. In SQALE, these are called 'rules'. Rules are measurable quality-related aspects of sources code, again linked to the lowest possible sub characteristic of the model. Rules can be violated, and the number of violations is essential in determining high-level quality indicators and remediation costs.

| Portability | Maintain-ability | Security | Efficiency | Changeability | Reliability | Testability |
|---|---|---|---|---|---|---|
| Compiler Related | Readability | API Abuse | Memory use | Architecture Related | Architecture Related | Integration level |
| Hardware Related | Under-standability | Errors | Processor use | Data related | Data related | Unit level |
| Langauge Related | | Input Validation & Representation | | Logic related | Exception Handling | |
| OS Related | | Security Features | | | Fault tolerance | |
| Software Related | | | | | Instruction related | |
| Time zone related | | | | | Logic related | |
| | | | | | Synchroni-zation related | |
| | | | | | Unit Tests | |

**Table 5 SQALE mapping of Sub Characteristics to Characteristics (Sonar plugin default model)**

#### 2.4.6.2 *Rules versus Metrics*

While other models may have source code metrics as their lowest-level quality hierarchy components, SQALE uses rules. To be able to comprehend the model, it is important to understand the relation between rules and metrics, since these are not equivalent. A metric, in the context of software source code, is a method to measure a specific aspect of the code. The input is source code and the output is a value (i.e. a number or percentage). A rule, on the other hand, not only measures some aspect of source code, but also decides whether or not the measurement result is to be considered 'good' or 'bad'. When a result is 'bad', this is considered to be a 'rule violation'. To accomplish this, rules incorporate not only a metric, but also parameters that define what is good and bad. Additionally, a parameter exists that defines the remediation cost in units of time per violation. So, a rule is not equivalent to a metric, but has a 'has-a' relation to one (and only one) metric. The following two examples clarify this.

- Cyclomatic complexity is a metric that measures the number of linearly independent paths through software source code (Fenton 1997). When incorporated in a SQALE rule, parameters would be added that define that the maximum allowable cyclomatic complexity is, for example, 8 different execution paths per method, and remediation costs for a violation are 45 minutes. Each method in the source code that has a cyclomatic complexity of 9 or higher would be considered a

violation of this rule and the total remediation costs are 45 minutes times the number of violations.

- Unit Test line coverage is a metric that measures the percentages of lines of code that are covered by unit tests (Williams 2001). When incorporated in a SQALE rule, parameters would be added that define that the minimum allowable unit test line coverage is, for example, 70% of source code lines without comments and whitelines per class and remediation costs for a violation are 30 minutes. Each class that has a coverage below 70% would be a violation and the total remediation costs are 30 minutes times the number of violations.

The parameter that define good and bad can be a single threshold value (i.e. a minimum or maximum) or can be a function for cases when there is no clear distinction between good and bad. In this case, the function is required to be monotonic on the interval(s) corresponding to non-conforming values, and constant or equal to 0 on conforming intervals. Remediation costs can be specified either per violation or per source code file containing one or more violations. Note that a linear relationship between cost, hours of work and lines of code is assumed, so these are considered equivalent and the terms are sometimes used interchangeably.

#### 2.4.6.3 Aggregation of Metrics

All aggregation in SQALE is performed by addition. This way, the SQALE remediation costs for each sub characteristic can be calculated by summing up the remediation costs of violations of all associated rules. This sum of remediation costs is referred to as 'index'. The index of a characteristic is simply the sum of indices of associated sub characteristics. The overall SQALE quality index is the sum of all characteristic indices. Also, a 'consolidated index' is defined for each level of the characteristics 'staircase', which is always the sum of indices of the specific level and all levels below.

| Rating | Range | Color |
|---|---|---|
| A | <0.2 | |
| B | 0.2-0.4 | |
| C | 0.4-0.6 | |
| D | 0.6-0.8 | |
| E | >0.8 | |

Figure 13 an example SQALE score-rating-color mapping

#### 2.4.6.4 Calculating Ratings

By dividing indices by the size of the artifact that is being analyses, we define the set of 'density indices'. Since the indices indicate the amount of man-hours needed to repair all issues, the size of the artifact is also expressed in man-hours, namely the amount of man-hours needed to develop the artifact. Out of the SQALE density indices, SQALE ratings can be calculated using a mapping of the continuous range of density indices to a discrete scale of a number of ratings. What is considered 'good' and 'bad' is left to the user of the model.

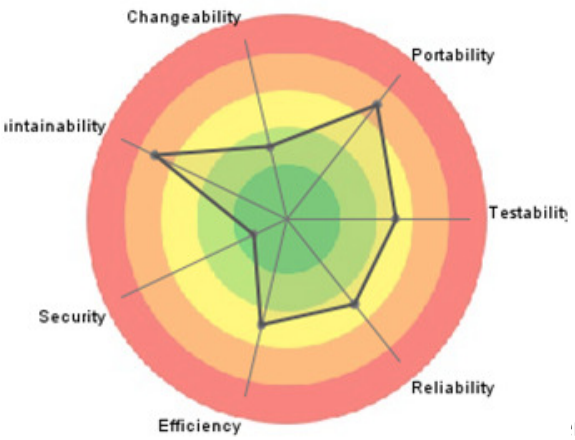



Figure 14 Sonar SQALE Kiviat example (screenshot)

This information also allows us to define a Kiviat. Figure 13 displays a mapping from indices to five ratings, and Figure 14 shows a Kiviat example from the Sonar tool that uses the SQALE model. These ratings provide us the 'indices' as defined in the requirements of this research, namely a discrete value on a defined scale. Also note that in this Kiviat diagram, 'inner' values, indicated by a point in the green circle, are better than values in 'outer'

(red) circles, as oposed to the Kiviat used by Kalistick as described in 2.6.2.3. Reason for this is that in the remediation cost paradigm, less is better.

The diagram on the following page shows a full example of a SQALE quality calculation using, from left to right, rules (incorporating metrics), sub characteristics, characteristics and an overall quality judgment. A legend, project metadata and index-rating mapping are defined on top of the diagram.

Additional background information can be found in two conference papers (Letouzey 2010a)(Letouzey 2010b).

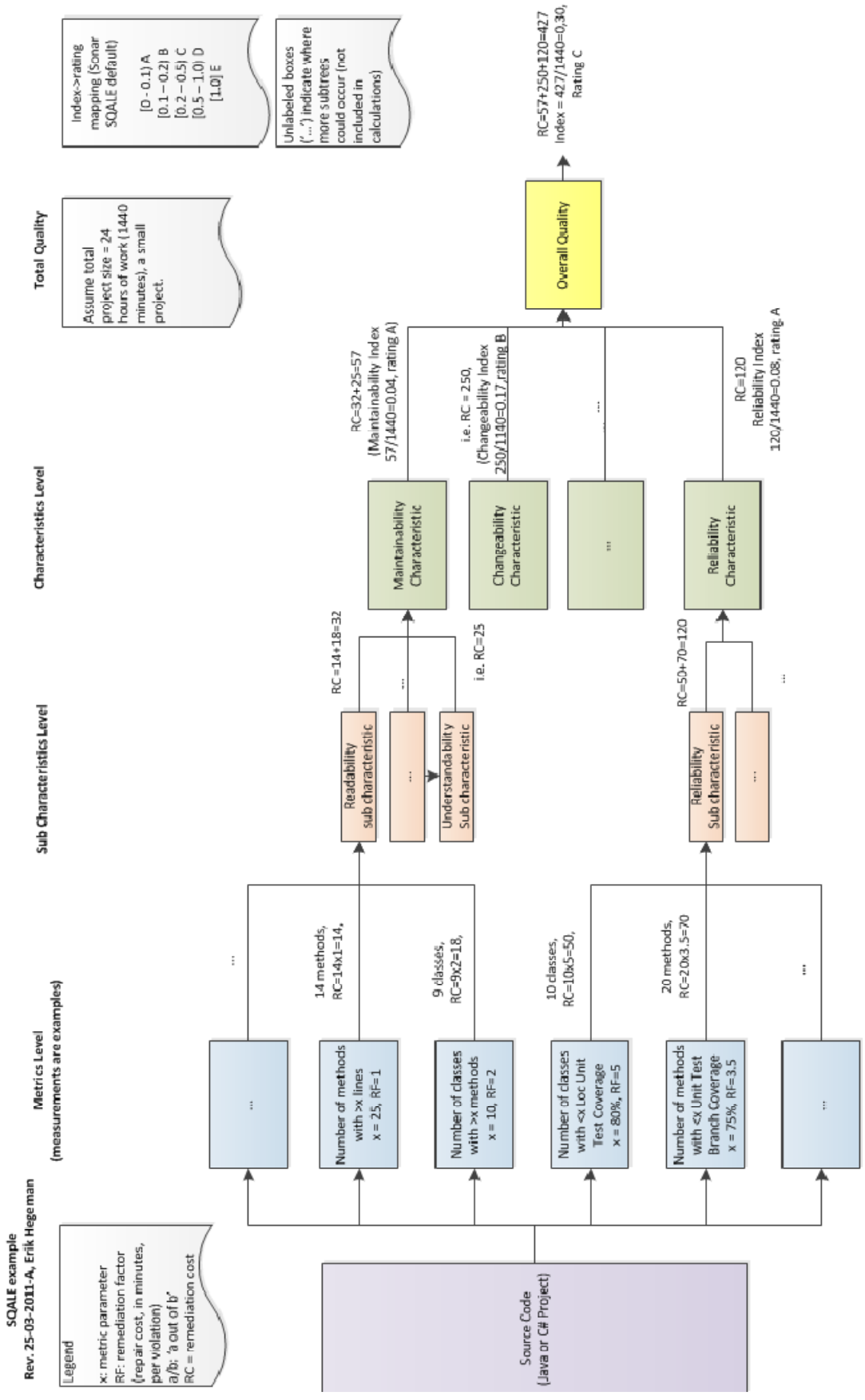



Figure 15 Full SQALE example

#### 2.4.6.5 Calculation Example

We illustrate the use of SQALE by means of an example. Consider the example of a SQALE quality assessment displayed in Figure 16. This assessment was performed using our Proof of Concept setup (6.2). The assessed project has 62 KLOC of code. SQALE has been configured so that a LOC costs 0.06 man-days to develop, which is the default value. Total development cost, therefore, is 62*0.06*1000=3720 days. The index-rating mapping is defined as follows: A: <0.01, B: 0.01-0.04, C: 0.04-0.16, D: 0.16-0.64, E: >0.64.

The overall SQALE rating, therefore, is 392 days / 3720 days = 0,105, which gets rating 'C'. This calculation can be performed for each of the characteristics, which has also been done to draw the kiviat. This yields the following scores:

- Changeability:
  10.1 / 3720 = 0.0027 → Rating 'A'
- Maintainability:
  225.7 / 3720 = 0.061 → Rating 'C'
- Reliability:
  113.1 / 3720 = 0.030 → Rating 'B'
- Testability:
  43.2 / 3720 = 0.012 → Rating 'B'

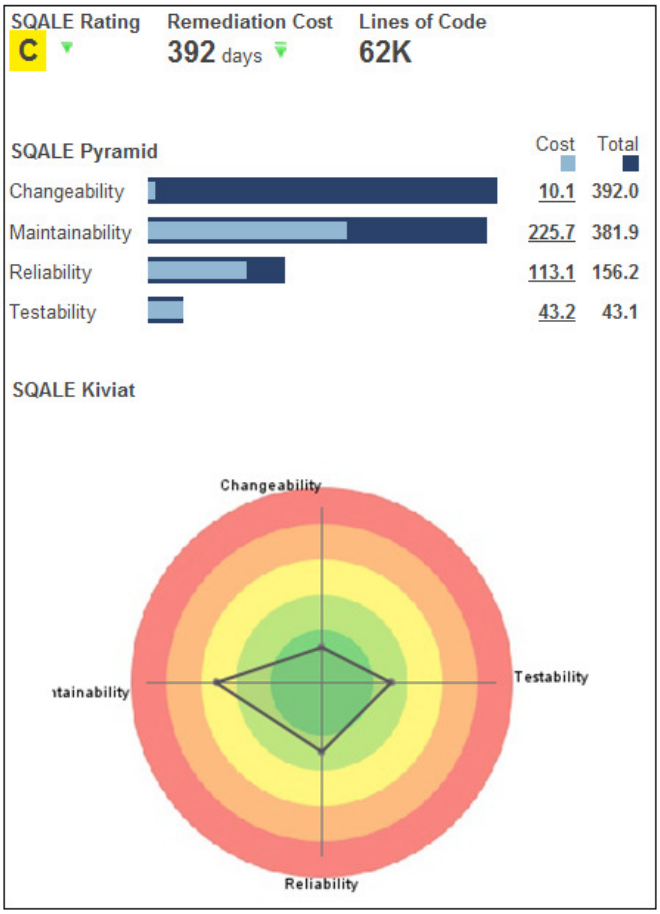



Figure 16 SQALE calculation example

## 2.5. Discussion

Summarizing, we discussed the following models:

- The McCal model, which introduces a hierarchy to translate metrics to higher-level quality indicators and uses linear regression coefficients to determine the effect of metric values on quality;
- The Boehm model, which predicts the number of defects in software based on 21 indicators relation to different aspects of the development processes;
- The Dromey model, introducing quality-carrying properties of source code artifacts which are asociated to high-level indicators to determine quality
- The SIG model, which is used in a commercial service which focusses on the maintainability aspects of ISO9126 and provides scores on a [1..5] scale for these aspects, based on source code analysis.
- The SQALE model that we explained in detail. It uses the concept of *rules* to determine remediation costs for source code artifacts, and translates these to high-level indicators by a

mapping of rules to sub characteristics, sub characteristics to characteristic and a mapping of remediation costs divided by development costs to a [A..E] scale.

Now that we have elaborated upon quality models, a few general remarks of discussion should be made. First, it should be emphasized that the implementation of the ISO standard by models, as well as the implementation of the models by tools, is not always strict. Model and tool developers usually appear to be using free interpretations of the entities they are implementing to suit there own needs or preferences. To give an example, the SQALE characteristics 'look like' the ISO9126 characteristics, but are not a strict subset (see Table 6 on page 40 for a comparison of ISO9126 to model implementations used in tools.). Also, the SQALE model specifices a number of characteristics to by mandatory in the model, while Sonar SQALE still allows the user to remove these.

Second, all models contain parameters that may be set by the user to conform the business needs or wishes. Most tools have 'default settings', so that they run out of the box. In this research, we use expert opinions to validate measurements. We choose to initially run tools (and thus models) with their default settings, because if we were to tweak the models to conform to Info Support wishes, we would need experts to define these settings. This would have an impact on our validation attempt, since the data source of the validation data, the experts, would also be one of the sources of the data to be validated. On the other hand, it will be interesting to see of model calibration by experts will increase the correlation of tool-based quality measurements with expert opinions. This makes it a logical research design choice to perform both an initial 'uncalibrated' measurement and a measurement using a quality model configuration calibrated by experts. We will incorporate this thought in the research design as defined in chapter 3.

## 2.6. On Quality Assessment Tools

To be able to accomplish the task of software quality assessment, we use software tools based upon software source code quality models. In the initial research phase of this project, tools have been selected to use in the Proof of Concept phase as defined in paragraph 3.1.3. In this section, we report on this selection process by identifying available tools and compare them using several relevant criteria. First, these criteria will be defined and motivated. Second, all identified tools will briefly be described. Third, a comparison of the available tools will be presented, using the criteria identified earlier. Finally, conclusions will be drawn from the comparison. Note that a maximum of two tools will be used for the Proof of Concept phase of this research, due to resource constraints, as mentioned in section 1.5.

### 2.6.1. Comparison Criteria

Before we can judge about tools we identify, we need to define a number of criteria. Some criteria are 'kickout criteria', meaning that meeting these state an absolute requirement. The following criteria have been formulated, and will be elaborated upon in the next sections:

1. Source and License Type
2. Quality Model & Standard Compliance
3. Languages Support
4. Business Model
5. Cost
6. Tool Architecture

#### 2.6.1.1 Source and License type

Although a preference for open source tools is superimposed by Info Support, we did not ignore all closed source solutions as of yet, since we may miss interesting tools that have advantages that compensate their closed source. Commercially available products will be investigated, provided enough information can be obtained to be able to assess their suitability.

The license type is relevant since we want to be able to incorporate tools in a professional, for-profit production environment. The tool license should allow this, because if it does not, the tool is not usable in the context of the business aspect of this research. Therefore, this is a kick-out criterion.

#### 2.6.1.2 Quality Model & Standard Compliance

The tool we select is supposed to be using a well-defined quality model. The concept of quality models was explained in section 2.4. Since we have explicitly chosen to use the ISO9126 standard as a foundation for quality assessment, selected tools should be using a quality model that is based upon this standard as well.

We define this as a kickout criterion. An initial quickscan, however, suggests that many tools may use an adapted version of the standard, often documented by terms like 'inspired by' or 'based upon'. We consider this to be allowable, since a requirement of strict standard compliance will kick out too many, or possibly all, potentially interesting tools.

#### 2.6.1.3 Language Support

Tools can support any number of target programming languages. A constraints of this research is that tools are required to at least support both Java and C#. This is therefore a kickout criterion.

#### 2.6.1.4 Business Model

Using third-party tools in a production environment implies that the production environment becomes, to a certain extent, dependent on that third party. This makes is preferably that this third party can continuously provide support, for example by providing software updates and bug fixes. We therefore define a criterion 'Business Model', and state the we prefer a model focused on continuity, i.e. a commercial product, or an open source project with sufficient community support, over a model that is less certain to persist, i.e. a single-person initiative or a purely academic research initiative.

#### 2.6.1.5 Cost

Cost may be involved in using third-party tools, for example license or support costs. Also, cost will be involved in the actual implementation of a tool in the software management process at Info Support, but for the purpose of this research we focus on the long term and only consider structural cost.

#### 2.6.1.6 Tool Architecture

Available tools differ in the global architecture they use. Examples are a standalone architecture, in which the tool runs on a standalone computer, or a server-client-architecture in which a central server may perform the actual analysis. Also, some tools are 'software as a service (SaaS)', in which the analysis is performed on a computer in a 'cloud'. We prefer not to use a SaaS-solution, since this would imply sending source code to a third party, which is especially difficult when Info Support is not yet the owner of the source code under consideration, i.e. when performing analyzes to

determine the management service that can be offered to a potential new customer. This is, therefore, a knockout criterion.

### 2.6.2. Tools

This section described the identified tools and assesses them using the defined criteria. Tools have been identified by searching literature and by using a free search on the Internet, using search terms like 'software quality tools', 'software quality assessment' and 'software metric tools'. Internet forums have been helpful, since answers to questions on the availability of software quality assessment tools can be found there. Although a thorough search has been conducted, we do not claim that this set includes all possible relevant tools, since we have no method of proving such a claim. Found tools were added to the set if the first impression was positive or neutral. To prevent 'false negatives', only tools that were evidently unsuitable have been left out. 'Evidently unsuitable' can mean, for example, that the tool does not support Java or C#, even using a plugin, is not actively maintained or is not supported by any organization. Some tools, or 'frameworks', appear to be very interesting but only seem to exist on paper, which clearly also indicates unsuitability. Also, many tools operate on a low level of abstraction, applying metrics to take code measurements but not using any quality model to judge quality. Most tools found do not meet the basic requirements. A small number of them do, these are described in the next paragraphs.

First, we will present a textual survey of all identified tools that are considered potentially suitable, and their capabilities and properties, after which a comparison of all tools will be provided, using the criteria defined earlier

#### *2.6.2.1 Squale*

Squale (Squale 2011) stands for Software QUALity Enhancement and is a 'qualimetry platform that allows both developers and managers to gain insights in the quality of software being developed'. It uses the Qualixo quality model to aggregate software metrics into higher level quality indicators (Laval 2008). As of 2009, it is a M€3.1 project with an effort of about 25 person-years (Bergel 2009). Main goals of Squale is to enhance the existing software quality approach in a number of areas, including enhancement of the quality model, defining dashboards and disseminating acquired knowledge. Squale is being used in a test-case production-environment at Air France and PSA Peugeot-Citroën.

The tool is distributed under the terms of the GNU Lesser General Public License (LGPL) version 3 and can be freely downloaded from the project website. A standalone version for testing purposes is available.

A major drawback of the current implementation is that C# support is not yet available. Although it is planned to be implemented, the current version only supports Java, C/C++ and Cobol. Although Squale is considered an interesting tool, this drawback makes it unsuitable for the Proof of Concept in this research at this point in time.

#### *2.6.2.2 Sonar*

Sonar is an 'open source platform to manage code quality' (Sonar, 2011). Since recently, Sonar incorporates the SQALE quality model, not to be confused with the Squale tool described earlier.

Sonar is published under the LGPL version 3 and freely available for download from it's website. It requires Java Development Kit version 5 or newer and depends on Maven for source code access.

Additionally, a database is required to store measured data. Options include MySQL, Oracle, PostgreSQL and MS SQL Server. Sonar inherently supports Java, while C# support is available using a plugin which enables simple support for maven in Visual Studio. In turn, this plugin requires a maven-dotnet plugin, which is also freely available.

A drawback of Sonar is that the .net-plugin does not support 100% of the functionality provided for Java by the built-in Java support. Improvements are being made, but since Sonar is originally only focused on Java it is to a certain extend tailored to a Java-way of doing things, making it difficult to achieve full functionality in any other language (Sonar .net 2011).

Although Sonar is an open source project, support for the SQALE quality model is provided by a plugin that costs k€2.7 per year per instance of sonar, including upgrade, maintenance and support. The plugin provides six additional 'dashboard widgets' that provide high-level quality indicators. Without this plugin, the information provided by Sonar has a lower level of abstraction (i.e. metric-level) and is less usable for managers. For the purpose is this research, the SQALE plugin is required. It has been verified that the plugins for SQALE and .Net can be combined, although the mapping from rules to quality characteristics has to be manually defined in this case as there is no default mapping available.

A public demo of Sonar, called 'Nemo', is available at  (Sonar Nemo 2011). This demo contains a database of a large number of open source projects. The demo environment is equipped with both .Net support plugins and the SQALE plugins. A drawback of the demo environment is that is cannot be fully customized, since no administrative privileges are available to visitors.

#### 2.6.2.3 Kalistick

Other than Squale and Sonar, Kalistick is a commercial product. It focuses on 'continuously delivering working software' by 'enhancing collaboration between developers, testers, Quality Assurance and operations (Kalistick 2011). It claims to be mainly suitable for agile development processes, the 'agile' product costing approximately k€0.5 per month. A 30-day trial version is freely available. Kalistick has native support for both Java and C#, which is an advantage over Sonar which depends on a plugin for C#-support.


**Figure 17 Kalistick high-level quality indicator example (screenshot)**

The software explicitly uses an implementation of the ISO 9126 standard, similar to SQALE. The set of high-level quality aspects, in Kalistick named 'factors' or 'quality themes', contains the items maintainability, capacity to evolve, efficiency, portability, security and reliability. Figure 17 shows an example of the high-level quality overview. Each factor is rated on a scale from 0, in the center of the image, to five, in the outer ring. An objective can be formulated for each factor. Apparently, when a factor value becomes higher than the objective, it is rounded down to the objective value.

Kalisticks quality model uses metrics and 'threshold values', allowing the model to identify all 'violations'. A violation is a case in which a specific metric exceeds the threshold value at a specific

location in the source code. Threshold values can be dynamically configured. All types of possible violations belong to one of the following quality domains: implementation, structure, test, architecture, documentation, duplication. Also, violations all have a 'severeness', in order of severeness: to avoid, disadvised, highly disadvised and forbidden. The quality factor values are calculated from the amount and severeness of the violations.

A drawback of Kalistick is that it is 'Software as a Service' (SaaS). While a part of the software runs on the client, the actual code analysis is performed in a cloud. Although a secure connection is used and non-disclosure is guaranteed, users may not want to send their source code to a third party for analysis. Info Support has indicated a SaaS solution to be 'not preferable', especially due to the fact that assessment at MITS takes place before Info Support becomes owner of the code. Since we have defined SaaS as a knockout criterion, we cannot use it in our Proof of Concept.

#### *2.6.2.4 CAST Application Intelligence Platform*

The CAST Application Intelligence Platform (Cast 2011) is an 'automated system to measure the structural quality of software and the performance of development teams'. It is a commercial product, that uses a quality model based upon a number of characteristics that is similar to, but not explicitly based upon, ISO 9126. Standard indicators are transferability, changeability, robustness, performance, security and maintainability index. CAST supports both Java and .NET source code natively.

Not much information about the internal working and actual functionality of CAST is available online, as its available documentation and website are very business-oriented. Also, no trial version is available, and no pricing information is available. Requests for more information have not been fulfilled.

#### *2.6.2.5 Metrixware*

Like CAST, Metrixware (Metrixware 2011) is a commercial product. It explicitly uses ISO 9126 and strictly follows this standard, using exactly the quality aspect set (called 'Code Health Factors') from the standard. It natively supports both Java and C#.

Unfortunately, no trial version or pricing information is available, and documents (i.e. whitepapers) that can be requested to be sent by e-mail were never received. Also, no reply was received to explicit information requests by e-mail. The websites of Metrixware and SQALE, however, identify Metricware as a SQALE-based SaaS-solution. Since SaaS is defined as a knockout criterion, we cannot use Metrixware in our Proof of Concept.

#### *2.6.2.6 SQuORING*

SQuORING is a new tool that has not yet been released. It explicitly uses the SQALE quality model and natively supports bot Java and C# (Squoring 2011). It's business model is not yet known, but the product is being developed by a privately-held company. The application uses a client-server model and is non-SaaS. Although the basic requirements are met, the fact that the software is not yet available makes it impossible to use in the PoC. It is, however, a promising projects that may become interesting in the feature.

### 2.6.3. Comparison & Conclusion

Using the criteria defined earlier, we will now compare the identified tools. The comparison of criteria values is listed in Table 7. Bold table cell entries indicate violations of knockout criteria. Also,

Table 6 shows a comparison of the quality indicators as used by the different tools, and the original ISO 9126 standard.

| As defined by ISO standard | Qualixo | SQALE | 'Kalistick-model' | 'CAST model' | 'Metrixware model' |
|---|---|---|---|---|---|
| Functionality | Functional Capacity | | | | Functionality |
| Reliability | Reliability | Reliability | Reliability | Robustness | Reliability |
| Maintainability | Maintainability | Maintainability | Maintainability | Maintainability | Maintainability |
| Efficiency | | Efficiency | Efficiency | | Efficiency |
| Usability | | | | | Usability |
| Portability | | Portability | Portability | Transferability | Portability |
| | Architecture | | | | |
| | Capacity to re-use | Reusability | | | |
| | Capacity to evolve | Changeability | Capacity to evolve | Changeability | |
| | | Testability | | | |
| | | Security | Security | Security | |
| | | | | Performance | |

Table 6 Quality Aspect Comparison

| Criterion | Sonar | Squale | Kalistick | CAST | Metrixware | SQuORING |
|---|---|---|---|---|---|---|
| **Source & License** | Open, LGPL v3 | Open, LGPL v3 | Closed, app-specific | Closed, app-specific | Closed, app-specific | **Unknown** |
| **Java/C# Support** | Yes* | **No (Java only)** | Yes | Yes | Yes | Yes |
| **Business Model** | Non-profit (non-academic) | Non-profit (academic) | Commercial | Commercial | Commercial | Unknown (Privately held) |
| **Cost** | k€2.7/year ** | Free | k€6/year | Unknown | Unknown | **Unknown** |
| **Quality Model** | SQALE | Qualixo | SQALE-like | **Unidentified** | ISO 9126 | SQALE |
| **Tool Arch.** | Client-server | Client-server | **SaaS** | Unknown | **SaaS** | Client-server |

Table 7 Software Tool Comparison

* C# support for Sonar is only available using a plugin

** Sonar itself is freely available, but the plugin required to use the SQALE quality model is a commercial product.

From the comparison, we can conclude that Squale is not a logical option to include in the Proof of Concept, since it does not support C# yet while alternatives do. C# support is a requirement for Info Support. We therefore choose to omit Squale from the remainder of this research.

While the functionality of Sonar is impressive, the C# support through a plugin may not be very solid. It is hard to determine to which extend this may lead to problems without a more practical test. Since it is the only open source solution that is considered potentially feasible, it seems logical to include Sonar in the third phase of the research to be able to learn more.

Kalistick appears to be a promising solution, natively supporting both Java and C#. The major drawback are the costs and the violation of Info Support's preference for an open source and non-SaaS solution. These aspects cause Kalistick to be unsuitable for our proof of concept.

CAST and Metrixware are, in many ways, comparable to Kalistick. Its scientific background could, however, not be validated and not all essential information is available. Inclusion in our proof of concept is therefore not an option.

This analysis shows only Sonar to be potentially applicable for our Proof of Concept. As can be read in section 6.2.1, however, this tool is potentially not optimally suited for implementation in a business context. We will use Sonar in the proof of concept and, based upon our experience, will advise on the question of whether or not to use Sonar as a standard component of Info Support procedures.

## 2.7. On Financial Indicators for Software Quality

In the second phase of this research, conforming to the overview diagram on page 18, we will use financial indicators to assess software quality, as to obtain data that can be used to validate software quality measurements made by sourcde code analysis tools. This paragraph provides some background information about financial software quality indicators.

(Slaughter 1998) states that software quality improvement should be seen as an investment. The study shows that software defect density improved with each software quality initiative, but at a decreasing rate. This implies that quality initiatives can (and should) pay off, but a conclusion is also that it is possible to invest too much in software quality.

Quantification of financial quality is a complex matter. Multiple methods to assess the cost involved in IT exist, but their focus is usually on operational cost or investment decision making. For example, the COCOMO model, that was a predecessor of the COQUALMO model, focusses on the financial impact of defects in software (Boehm 2000). Other examples are (Benaroch 2002) that presents an approach for managing IT investment risk and (Verhoef 2005) provides an example of a quantitative investment decision assessment. Sometimes, economical models used in other sectors are adapted to work for IT projects (Slot 2010).

The link, however, between financial results of software management and source code quality, appears to be an uninvestigated area of science, as can be concluded from the lack of publications on this specific topic.

A general conclusion that can be drawn from literature relating IT to finance, is that a lot of financial resources are spend on 'challenged' information technology solutions. Recent estimations indicate that a annually a budget of 290 billion US dollars is spend on challenged IT, as indicated by in magazine article (Verhoef 2006). Common types of problems in IT are budget overruns, time overruns or inadequate functionality (Standish, 2001); these types are also used as a definition of 'challenged IT'. As Verhoef indicates, reasons for these failures can often be found in new requirements being introduced in later stages of the development process ('requirement creep') or attempts to do too many things in an amount of time that is too small ('time compression'). The reasons for failure found in the Standish report concern three of the four project steering parameters from Figure 2 on page 19: cost, scope and time.

A potentially relevant detail from the literature is the following. Verhoef (Verhoef 2005) states that the amount of resources needed per function point of a software system increases with the total number of function points. The curve that is associated with this statement is displayed in Figure 18. This curve is mainly associated with development costs, i.e. the amount of resources needed to 'add' functionality. Given the fact that software management also involves working on software on a source code level, i.e. when fixing bugs ore implementing change requests, it seems reasonable to assume that the relation is applicable there as well.

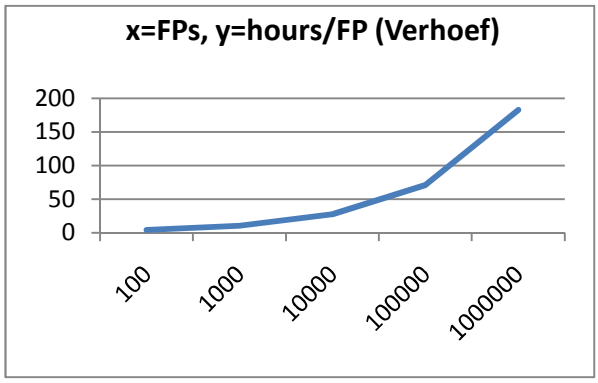



Figure 18 FPs and hours per FP according to Verhoef

An important indicator of maintainability is the amount of hours spend on a project during the maintenance phase, per time period. In a product with a high maintainability, it is easier to accomplish a task (i.e. find a bug) than in a product with a low maintainability (Woolderink 2007). Based upon (Verhoef, 2005), however, it can be assumed that given a small and a large project with an 'equal' maintainability, tasks are still easier to accomplish in the small project than in the larger project. A conclusion that can be drawn from this is that maintainability becomes more important once a software system becomes larger.

Readily usable and suitable indicators for financial quality that are applicable in this research do not seem to be available, since the models mainly focus on investment and have a business perspective, using paradigms like the Total Cost of Ownership. Also, this research calls for simple and transparant indicators, since we need to prevent force-fitting potentially unvalidated financial quality models to validate a software source code quality model. Therefore, we have chosen to define our own indicator, based on available financial data and project management data. Since this information cannot be considered 'background', it is mentioned in the research design section of this thesis (see paragraph 3.3).

# 3. Research Design

In the introductory chapter, the goals of this research have been defined. To accomplish these goals, a research structure has been defined, which is presented in this chapter.

The diagram in Figure 19 provides an global overview of the research steps and the sequence in which they are carried out. The Proof of Concept phase incorporates an explicit iteration possibility. All steps are explained in this chapter. Steps indicated in blue have already been carried out in the research design phase, while green steps concern research conduction. The 'Preliminary Research Phase' consists of the research on tools, models and other background information, as presented in chapter 2.

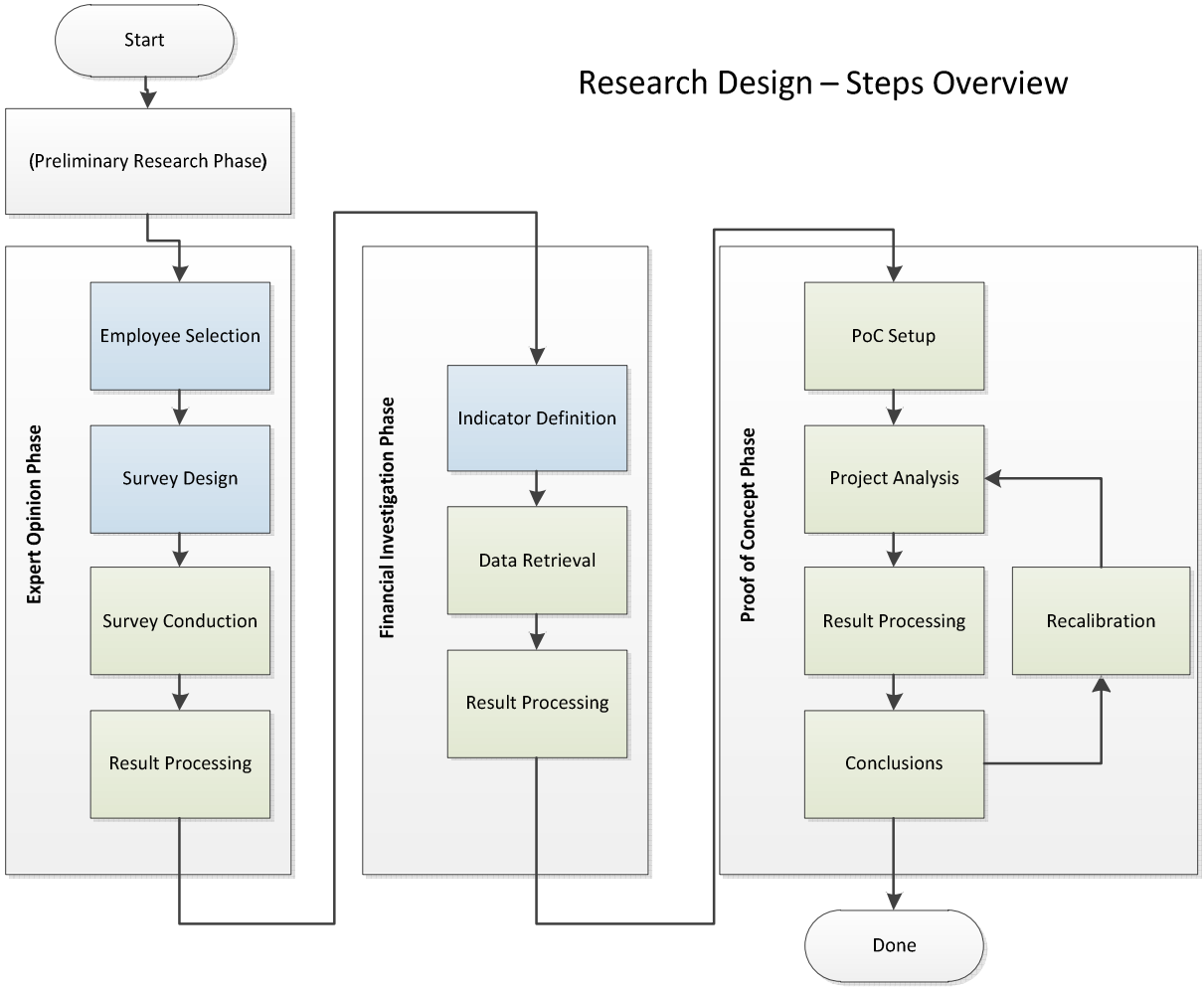



Figure 19 Research Design - Steps Overview

## 3.1. Project Subset Selection

Since all research steps involve the use of actual software projects, a subset of these relevant software projects at Info Support has been selected to use as sample data in all phases of the research. We choose to select a subset of MITS projects that is as large as possible, to maximize statistical significance of the research.

Before making the selection, we define the following constraints:

- The subset will only contain projects that are actively maintained at the time of this research. By doing this, we omit older projects that were developed or maintained using methodologies no longer commonly used at Info Support, and we also ensure that the entire subset consists of project that use a consistent management methodology. Also, we reduce the risk that we are unable to conduct surveys due to the non-availability of professionals involved in the project.
- The subset will only contain projects that have a 'critical mass'. We ignore small, internal projects that exist only to aid in non-primary processes. The reason for this is that for projects of those kinds quality is not as important as for projects that Info Support uses to earn money, so quality standards do not necessarily apply and little expertise and incident data may be available.
- To be able to perform the research, a number of resources need to be available, being:
    - A number of employees to be surveyed on project quality;
    - Financial data providing indicators for project quality to be used in the Financial Investigation Phase (see section 3.3)
    - Project source code (in Java or C#) to be able to perform the Proof of Concept (see section 3.4)

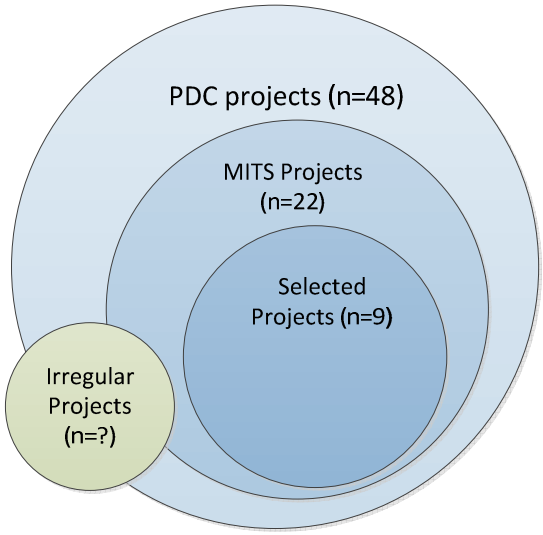


Figure 20 Project Subset Selection Venn diagram

As mentioned in paragraph 2.1.1, Info Support both develops and manages software. Management activities also include additions and modifications, for which software development process Endeavour is used. Endeavour is also used for software projects under development that have not yet taken into use and therefore are not 'managed'. Because of this, the set of projects managed by MITS is a proper subset of the set of projects of the Professional Development Center. Since this research assignment is performed at MITS, and we prefer to have actively maintained projects we choose a subset of MITS projects. Out of the 22 MITS projects we selected, 9 projects that are deemed suitable. Figure 20 shows a Venn diagram representing the subset selection. As denoted by the irregularly colored circle, 'irregular' projects may exist at Info Support (i.e. projects that are not managed using Endeavour or project that are not managed using regular MITS procedures). The exact size of this set is unknown. This set has been left out of this research.

Based upon this, we came to a selection of projects as listed in (confidential) appendix K. We will denote the nine selected projects with identifiers A through I in the remainder of this thesis. The

selection process is performed by taking the list of all MITS project and removing those entries that are not suitable for the purpose of this research. There were basically three reasons to consider projects unsuitable:

- The project was no longer actively maintained (i.e. a contract had ended);
- The project was an internal, minor project that is not to be considered a serious application;
- The project was irregular in some way (i.e. not maintained using MITS standard policies)

Information on project was obtained using information on the internal MITS project portal at Info Support and in consultation with MITS management.

## 3.2. Expert Opinion Phase

To obtain information on how software quality is perceived by IT professionals, a survey will be conducted with professionals at Info Support who are directly involved in the management of software (i.e. MITS employees). For this purpose the following steps will steps will be taken:

### 3.2.1. Employee selection

A subset of relevant employees of Info Support will be selected. Relevant employees are those who are directly involved in software management, namely employees from the Managed IT Services department. This set of employees includes those who have knowledge of the projects on a source-code level.

Since the survey contains only closed question (i.e. scales) and does not have complex branching; see the survey design section 3.2.2, there is no need to 'physically' conduct interviews. Participants can fill out a survey that is send to them by e-mail. This allows us to define a larger group of survey participants and gain results that have a higher statistical significance.

Because of this, we choose to ask all MITS employees, approximately 30, to fill out the survey. It must be noted that only approximately half of the potential respondents has direct knowledge of the source code of MITS project. Others are not expected or required to participate in the survey. The maximum response rate, therefore, is approximately 16 experts.

### 3.2.2. Survey Design

The survey has been designed to be able to obtain relevant information from the aforementioned employees. The questions asked were be designed as to aid us in answering the research question of this section: *How do Info Support experts rate the sample projects on relevant SQALE characteristics?* We will use closed questions so that statistical analysis of the results is possible. Also, we will use questions with a particularly high level of abstraction, specifically the level of abstraction conforming to the characteristics of the SQALE quality model that is used by the Sonar tool in our Proof of Concept phase. Reason for this is that in this way, we get clear results that indicates how experts perceive quality, based on their experience and intuition. If we would use low-level metrics in the expert survey and transform the result to general quality characteristics, we would basically be manually applying a quality model, similar to a model used in tools. This would mean that the outcomes of calculations made by quality models will be validated by calculations made by other (non-validated) quality models, which is something we want to prevent.

A number of relevant constraints for the survey has been identified:

- To quantify survey results, we will not use open questions but use scaled questions instead. Research suggests that an odd number of items between 5 and 9 is recommendable (Cox 1980). We will use a nine-point scale to maximize the granularity of the results.
- People with different roles may have more or less knowledge about projects concerned. We therefore introduce a 'confidence factor', which participants can use to judge their own confidence in their ratings. We can use this confidence factor afterward to calculate weighted averages of given scores.
- We present the scale to participants as an 'agreement scale', ranging from 'totally disagree' to 'totally agree' instead of a numerical scale. The reason for this is that a higher spread in answers is expected using this method. Rationale behind this assumption is that in a 1-9 scale, participants are expected to give mainly scores in the 6-8 range, since a quality aspect of a software system is never expected to be perfect nor so bad that deserves the equivalent of a grading for a failed exam. It should be noted that no literature was found that either confirmed or invalidated this assumption, so we rely on our assumption-making capabilities here. For the results to be processed, we will translate the provides answeres back to a nine-point scale.

The constraints above allow us to define the survey in the form of a set of matrices. Each matrix is about one of the selected projects. The horizontal axis contains the nine-point scale, while the vertical axis contains the quality indicators from our definition of quality. An example of what a survey looks like can be seen in appendix E.

After the survey has been conducted, we calculate our validation information as follows: for each project from the subset of possible projects as defined in 3.1 and for each quality characteristic as defined in the survey of appendix E, we take the weighted sum of score of all respondents. The weights of the scores are the confidence levels on a discrete scale from 0 to 2, while the scores are the values entered in the survey on a discrete scale from 0 to 8. We reverse this scale so that 0 is the best score and 8 is the worst score, so that it is aligned with the remediation cost paradigm using in the SQALE quality model (Sqale 2011) that is used in the Proof of Concept. A description of the remediation cost paradigm can be found in the background information section, 2.4.6. The scale is defined as follows:

| Dutch text, used in Survey | English translation | Numerical value |
|---|---|---|
| **Volledig mee eens** | Totally agree | 0 |
| **Sterk mee eens** | Strongly agree | 1 |
| **Mee eens** | Agree | 2 |
| **Enigszins mee eens** | Slightly agree | 3 |
| **Neutraal** | Neutral | 4 |
| **Enigszins mee oneens** | Slightly disagree | 5 |
| **Mee oneens** | Disagree | 6 |
| **Sterk mee oneens** | Strongly disagree | 7 |
| **Volledig mee oneens** | Totally disagree | 8 |

Table 8 Survey scale definition

The questions to be answered have a form that is equivalent to 'Project <name> is <characteristic>' , for example 'Project 'TicTacToe' is Reliable'. See appendix E for the exact design of the survey.

We choose to include only a subset of the SQALE characteristics, as listed in Table 9. The reason to not include some characteristics, is that they are irrelevant in our definition of maintainability as stated in paragraph 1.4.1. Note that the definition of the 'maintainability' characteristic in SQALE is much more narrow than in our own definition (see Table 5 on page 30; the SQALE definition consists of just understandability and readability, while we include analyzability, stability, changeability and reliability), which is the reason why we cannot just only include the SQALE 'maintainability' characteristic. Also, note that the SQALE definition document considers changeability, reliability and testability to be mandatory elements when defining a SQALE subset to be used in a specific context (Sqale 2011). The Reusability of the SQALE definition is not mandatory, and also not implemented in the Sonar SQALE plugin, so we have no other option than to leave this one out.

| Characteristic | Include in survey? |
|---|---|
| **Reusability** | No (not in SQALE) |
| **Portability** | No |
| **Maintainability** | Yes |
| **Security** | No |
| **Efficiency** | No |
| **Changeability** | Yes (mandatory) |
| **Reliability** | Yes (mandatory) |
| **Testability** | Yes (mandatory) |

Table 9 SQALE characteristics subset selection

It should be explained why we do not use exactly the quality indicators from definition 1.4.1 in our survey, since these are quality indicators we eventually want to be able to calculate values for. The reason for this is that the mapping between this definition and the SQALE characteristics is particularly non-trivial. By using the same characteristics in the survey and the tool assessment, no mapping is needed to calculate correlations, so we avoid the risk that an ill-defined mapping causes us to be unable to find a correlation, or to incorrectly find a correlation. Mapping validated tool characteristic values to our definition of quality will be performed in the final phases of this research. If we are unable to define a good mapping, we will have to conclude that it is advisable to express quality in terms of the SQALE characteristics instead of the maintainability aspects from definition 1.4.1.

The calculation of project quality indicators from the survey results yields for each project a score for each of the selected quality characteristics, on a scale from 1 to 9. We presume this scale to be equivalent to the SQALE Kiviat value for the individual characteristics, which has a scale from 1 to 5. The transformation of values from one scale to the other can be performed by simple scaling. The correlation between survey results and tool measurements is one of the expectations of this research (see 3.5). The results of the survey will be used to validate the outcomes of the tool-based assessment phase as described in section 3.4.

### 3.2.3. Survey Conduction

The described survey has been conducted by asking by e-mail the Info Support employees of the Managed IT services department to participate. In an ideal situation we will have a 100% response rate, we do not consider this to be a necessity. We do however, define a minimum response rate of 67% of all potential respondents, and will increase the response period and send reminders if this rate is not met at the end of the initial participation term. The survey answers the research sub question: *how do Info Support experts rate the sample projects on relevant SQALE characteristics?*

### 3.2.4. Survey Conclusions

Conclusions will be drawn from the survey results. An answer to the research question will be formulated. This also provides us with the project quality assessment results as described in the Survey Design section, providing validation data for the tool-based quality assessment.

Of course, it should be emphasized that expert opinions are opinions, and therefore inherently subjective. It is, however, assumed that these opinions on quality reflect the 'actual' quality of software. This assumption, which is supported by the fact that we are dealing with experts that work on the projects on a daily based as well as the fact that we introduced confidence scores in the survey, is needed to be able to use the survey results as validation data. We thus define the survey results to provide an adequate taxonomy of the quality of the projects.

## 3.3. Financial Investigation Phase

To be able to objectively assess the quality of software in relation to financial results, we will conduct an investigation of financial project results. Goal of this investigation is to determine the financial quality for each element of the selected set of projects, so that these can be compared with quality indicators resulting from source code analyses and expert opinions. The following steps will be taken.

### 3.3.1. Financial Indicator Calculations

A Financial indicator will be defined and calculated for each of the projects. As can be read in the theoretical background section 2.7, literature research did not provide sufficiently suitable information to construct a quality indicator. Therefore, we must define this indicators ourselves. Consistent with the expert opinion phase, we choose to limit the number of transformations applied to financial data to calculate the indicator, since we want to prevent introducing our own non-validated 'financial quality model' to generate validation data. Indicators therefore will be as basic and transparent as possible.

#### *3.3.1.1 Cost Types*

In financial terms, hours spend on a project can be expressed in currency, by multiplicating the amount of hours by the (average) salary per hour. For the purpose of this research, we neglect indirect costs and only consider man-hour costs involved in management or development of projects, which is a choice consistent with existing cost models (i.e. Atterzadeh 2010). Examples of ommitted indirect costs are costs of facilities needed to manage software, amongst which buildings, computer hardware and support personell, so costs that cannot be directly related to one project. At Info Support, man hours are registered using codes for different types of activities. We choose to only include the 'incident' and 'problem' codes, as these are the categories that are related to maintainability in a way that can indicate quality. We explicitly exclude hours booked due to customer requests (i.e. 'change request', ' service request' or 'release' codes). The mentioned codes fully cover the set of available codes.

A small investigation shows that the following information is obtainable:

- For each project, the actual number of hours spend on that project during a specific time-period (i.e. a year), per type of cost (i.e. incident, problem, service request)
- The size of each project.

#### 3.3.1.2 Function Points vs. Lines of Code

We prefer to look at a project size in Kilolines of Code rather than function points, since KLOCs are better measurable and measurements have a better availability KLOC measurements are available from the tool used in the Proof of Concept. Also, evidence exists for a relationship between a project size in lines of code and the number of function point that is approximately linear (Dolado 1997)(Caldiera 1998). The exact relationships depend on the definition of the function points and lines of code, but in general the relationship has the form:

**S = a * F + C**

Where:
- S = the size of the project (the number of Lines of Code)
- a = the scalar (additional LoC's per FP)
- F = the number of Function Points in a system
- C = a constant to account for overhead

An Info Support sample project indicates values of approximately C = 20.000 and a = 100. Further analysis could be used to narrow down these values and calculate their reliability. We consider this a suggestion for future research, especially due to the fact that no research that explicitly focuses on this issue appears to have been conducted, but for the purpose of this research, the knowledge that we can consider the relationship between function points and lines of code to be linear is sufficient, because it allows is to consider FP's and LOC's to be equivalent.

#### 3.3.1.3 Counting Lines of Code

While counting code lines, we chose to use the project size measurements as obtained by Sonar, the tool used in the Proof of Concept (see 2.6.2.2 on Sonar and 3.4 on the PoC). This indicator ignores white lines and comment lines. By using this Sonar measurement, we assure a constant definition of 'line of code' over the projects. Some issues:

- A slight difference in the amount of lines of code needed to program specific functions or constructions may exist between Java and C# project. We assume this difference to be neglectable and will ignore this. This is a reasonable assumptions, because:
    - Java and C# are syntactically similar object-oriented languages;
    - Strong Info Support coding rules imply that code follows conventions that are similar for both languages.

  A practical comparison of Java and C# can be found in (Chandra 2005).
- The Project may contain generated code. For example, this may be code that is generated by a GUI designer. Sonar automatically skips generated code when it is labeled as such, but we have no way of knowing for sure that all generated code, and only generated code has this label. This will not affect resulting quality indications, as long as the amount of generated code either:
    - Correctly automatically skipped, and/or
    - Is neglectable, and/or
    - Constitutes a constant percentage of the total project's code.

  No measures to verify or falsify these possibilities are available. We therefore assume some combination of the possibilities be true, which implies that we do not consider generated code to pose a problem. If during the Proof of Concept phase we get reasons to believe this assumption is not valid, we will deal accordingly.

#### 3.3.1.4 Financial Quality Indicator

We have established that the amount of effort put into a managed project during a specific time period is an indication for quality. The relationship between hours and money is linear in the context of Info Support, by definition. We also have established the equivalence of Function Points and Lines of Code. Figure 21 displays the established relations.

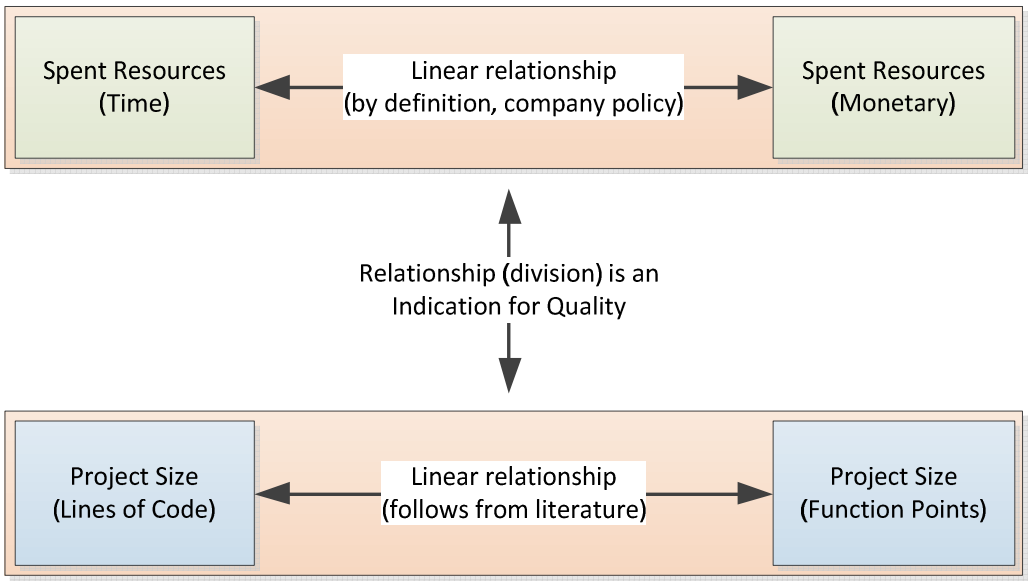



Figure 21 Relationship between time, money, LoC and FP in projects

The relationship between the 'upper' and 'lower' part of the diagram in Figure 21 is considered to be linear (Albrecht 1983). As stated in the background information section 2.7, however, indications exist, that the amount of work per function point increases linearly with the size of the product (Verhoef 2005). This would imply that the size of the projects needs to be exponentially accounted for. From the established relations, we derive the following indicator for quality:

$$\mathbf{Q = R / S^F}$$

Where:
- Q = the quality indicator (in €/KLOC)
- R = the actual amount of hours spent on a project in 2010
- S = the size of the project in KLOCS
- F = the size weight exponent

For now, we assume F = 1, and reconsider this choice if research results call for it. Included in the labour costs are all man-hours assigned to a specific project in 2010, but since we focus on maintainability in this research, only hours spend on software management (assigned to category 'incident' or 'problem') will be included, as mentioned earlier.

We selected 2010 as the period to look at. Several issues need to be taken into account when making this selection. We assume the choice of the full year 2010 to be the right amount and period of time considering the following factors:

- We cannot select a period that is very short; since the number of reported incidents would be very low and potentially non-representative for longer periods.
- We cannot select a period that is very long; since projects change during time; they are added or removed to the MITS portfolio and projects in the portfolio are modified and improved to increase quality and suit customer needs.
- Since we use recent source code, we cannot use data that is too old; since issues would concern much older versions of software that we analyze. Also, expert opinions retrieved are about the status quo of the projects.

This indicator thus calculates the 'management time investment per kiloline of code in a specific time period', i.e. the amount of effort spent on a project, during a fixed time periode, per kiloline of code. We assume the project size to be constant during the period we looks at. This is considered a reasonable assumption, since the software development phase has already finished before MITS starts managing projects. The only factors influences source code size are fixes and extensions, which are assumed to have only a limited impact on the change in project size.

As mentioned in the background information section, the amount of effort needed to perform a task increases with the size of the project (Verhoef 2005). It was, at time of defining the research structure, not yet clear if this should be taken into account in the calculations of this research. This is due to the fact that investigated projects have a modalar structure, and we do not know as of yet if this could compensate for the mentioned effect. If the modular structure does not compensate for the mentioned issue, we should transform the quality indicator formula by increasing the exponent to the size variable with a value conforming to the curve in Figure 18 on page 42 to compensate for this.

#### *3.3.1.5 Financial Investigation Conclusions*

Conclusions will be drawn from the calculations, and expectations for the outcome of the automated source code assessment will be formulated: *What is the financial quality, expressed as hours/KLOC, of the sample projects?* Also, this step will provide us with an indication of project quality that will be used as validation.

## 3.4. Proof of Concept Phase

In this phase, we develop a Proof of Concept in which we use software tools to assess code quality, providing us with quality indicators. We then compare these indicators to the financial indicators and survey results obtained in the earlier phases of this research. The process is based upon ISO14598 as described in theoretical background section 2.3.2. The following steps will be taken.

### 3.4.1. Proof of Concept Setup

Setup up a proof of concept using the tools mentioned in this proposal and the set of software projects identified. In the setup step, we setup up the tools on a computer system that has access to the source code of identified projects so that we are able to perform analysis. A log has been kept during the setup procedure and added to this thesis (see appendix H), so that it procedure can be repeated.

### 3.4.2. Software Quality Assessment

Assessment of software quality using this proof of concept. For each of the software projects identified in the project subset identification, we will determine the value of a selected number of quality characteristics using the selected software tools.

### 3.4.3. Assessment Result Analysis

Now that we have expert opinions, financial indicators and quality indicators, we can answer research sub question #3: *How are the sample projects rated by a Sonar SQALE setup?* Further steps will be taken to yield more analysis results if the initial results call for this.

## 3.5. Research Results Phase

### 3.5.1. Correlation Calculations

Finally we will summarize the results and answer the main research question: *To which extend can software tools, incorporating quality models, provide quality information that matches Info Support experiences and expectations?*

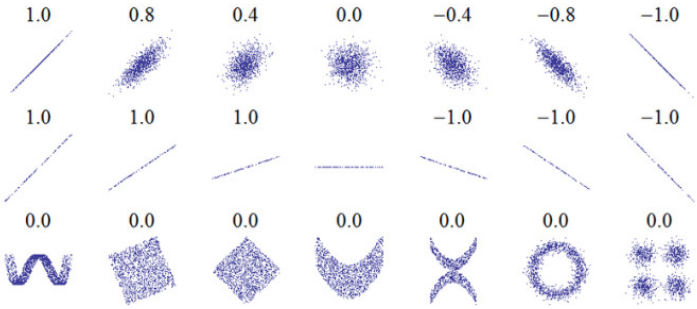



Figure 22 Examples of linear relations and their Pearson coefficient values

In the conclusion phase, we will apply statistical analysis to the quantitative results of the earlier research phases, to be able to draw conclusions about correlations. Also, we will elaborate on the generalizability of research results to a context that is broader than Info Support.

Since at this point we will have completed the three research phases, we have the following information available:

- For each selected project, for each selected quality characteristic: a score on a scale from 0 to 9, indicating the weighted average expert opinion, to be used as validation data. Confidence scores (0, 1 or 2) are used as weights.
- For each selected project, a general financial indicator indicating financial project success, to be used as validation data.

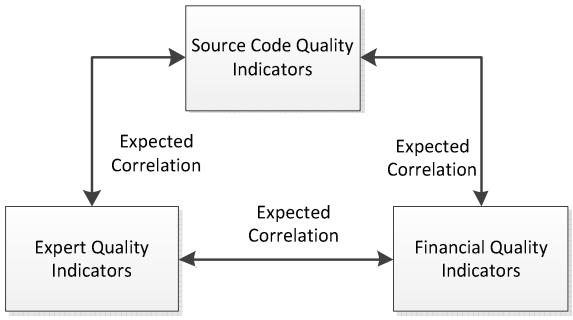



Figure 23 Expected quality indicator relations

- For each selected project, for each quality indicator from our quality definition, for Sonar: a score on a scale from 1 to 9, indicating the quality as measured by the tool, to be validated using the validation data.

This quantitative information will be the input for our statistical analysis. Specifically, we expect a linear dependency between all three datasets, and therefore will calculate the Pearson Product Moment Correlation Coefficient (Rodgers 1988) for each of the combinations of the three data

sources. The reason to choose this coefficient is that it is suitable for linear dependence and non-sensitive for scaling, which makes is suitable for the use in this project. It also

This correlation coefficient provides a value between -1.0 and +1.0. A higher absolute value of the coefficient implies a stronger correlation, either negative or positive. Figure 22 shows examples of scatter plots of the values of two variables and the Pearson value of their correlation. For all three relations between the datasets, we expect to find a correlation, as indicated in Figure 23.

Interpretation of the actual value depends highly on the context of the research (Cohen 1988); there is no universal definition of 'high' or 'low' values. In the context of this research, due to the subjective nature of the expert opinions and our unfamiliarity with the correlation between financial quality and source code quality, we do not expect to correlation values near 1 or near -1. Therefore, we define the following qualifications:

| Abs. value | Qualification |
|---|---|
| [0.0 – 0.1) | Insignificant |
| [0.1 – 0.3) | Low |
| [0.3 – 0.5) | Medium |
| [0.5 – 0.7) | High |
| [0.7 – 1.0] | Very high |

Table 10 Correlation Coefficient Qualifiactions

We define the following expectations for the relations. In all cases, the direction of the correlation will be so that a higher quality on one scale correlates with a higher quality on another scale, although a higher quality may be indicated by a lower value (which is that case with the financial indicator).

- At least a 'medium' correlation between source code QI and expert QI
- At least a 'low' correlation between expert QI and financial QI
- At least a 'medium' correlation between source code QI and financial QI.

The reason to expect merely a 'low' correlation between expert QI and financial QI is that we use these two data sources to validate results of the source code QI and not to validate each other. We do not need to draw conclusions from this correlation to be able to answer the research questions. Nonexistence of a correlation, however, could be qualified as 'strange', since this would imply that we validate values using two datasets that are themselves apparently uncorrelated or almost uncorrelated. A significant negative correlation would be even more strange and is also unexpected, since this would imply that projects which are considered 'good' be experts actually have a low maintainability based upon financial data and vice versa.

We also have reasons to believe that no very high correlations between Sonar SQALE measurements and validation data can be seen. These reasons are seemingly reasonable assumptions that cannot be falsified, rather than provable facts. An example is the inherent subjectiveness of opinions and the sensitivity for circumstances. Since this is a topic of discussion, more information is provided in the discussion section of this thesis (chapter 9.2).

### 3.5.2. Statistical Significance

The fact that we have a relatively small project sample size (n=9) implies that we either have to reach high correlation values or accept a lower confidence when making statements about the statistical

significance of research results. The critical values for the Pearson Correlation Coefficient that apply in our one-tail test are displayed in Table 11 (Moore 2006). In this case, *DF*, the number of 'free degrees', equals 9 – 2 = 7.

| Confidence | 0.25 | 0.10 | 0.05 | 0.25 | 0.01 | 0.005 |
|---|---|---|---|---|---|---|
| Value | 0.2596 | 0.4716 | 0.5822 | .6664 | .7498 | .7977 |

Table 11 Critical Pearson values (DF=7)

This table should be read as follows: in the context of this research, to be able to state that a positive correlation exists with 90% certainty (1 minus 0.10), the Pearson correlation coefficient needs to have a value of at least 0.4716.

### 3.5.3. Procedure

Figure 24 shows a flowchart of the procedure of calculating correlations and reaching conclusions. It shows that we will consider validation data to be true, and modify the tool's quality model configuration to increase the correlation if possible. This way, we also answer the fourth research sub question: *Which methods to improve the quality of the quality model configuration exist?*

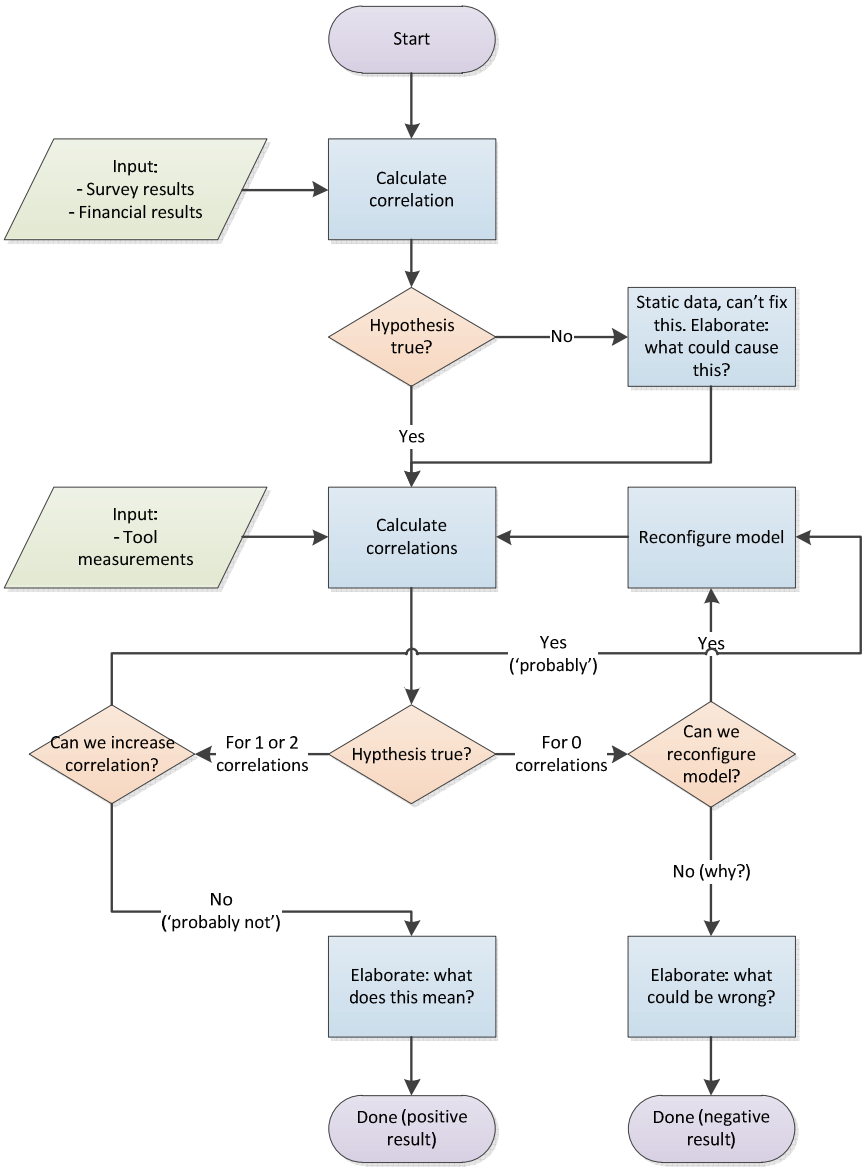



Figure 24 Correlation calculation & conclusion flowchart

The diagram in Figure 24 requires some elaborated. At the top of the diagram, we see the start of the process. The first step is to calculate the correlation between the two sets of validation data. This data is fixed, so independent from the actual correlation we continue, but it the correlation is not what we expect, we elaborate upon why that may be the case. After this, we calculate correlations between validation data and Sonar measurements. We then enter a loop in which we recalibrate the model and recalculate correlations as long as we see possibilities to increase them. Finally, we come to a conclusion.

### 3.5.4. Calibrating the Quality Model Configuration

Initially, the quality model is configured by enabling all possible rules available in SQALE with default parameters and applying a default mapping. If no default mapping is available, we will manually map rules to seemingly appropriate characteristics.

There is no reason to assume that using the initial quality model configurations, optimal correlations of measurements results and validation data will be obtained. After the initial correlations have been

calculated, therefore, we will attempt to improve the correlation by reconfiguring the quality model. This reconfiguration procedure can be seen as a calibration attempt, in which we try to make modifications to the quality model configuration that make it better reflect Info Support coding standards and policies. There is a number of possible ways to do this, which can be combined in a calibration procedure.

#### 3.5.4.1 Calibration Methods: selecting rules

The following list sums up a number of methods that can be used to calibrate the quality model. These methods are intended to be used to decide whether or not to include rules in the configuration and to configure individual rules.

1. Iterate over all existing rules (approximately 1350) and for each rule, decide whether or not it should be enabled. Use the provided functions to initially enable all rules, and during the iterations disable the individual rules that are not applicable for the context (in case of the Proof of Concept aspect of this research, these are rules that are not a coding standards at Info Support) No knowledge of the quality model is needed to perform this task. Of course, general knowledge of Java or C# is needed to be able to determine whether or not to disable rules, as well as knowledge of context-specific coding rules.
2. Perform the same iteration, and make an estimation of the remediation cost per violation for each of the enabled rules. By doing this, the remediation costs of the rules better approximate the actual situation. The estimation process could be performed by multiple people, where the average result is used as output value. By making the estimation with more people, the quality of the estimation is assumed to increase, and with it the quality of the quality model configuration.
3. The iteration can be performed by more than one person. For example, instead of one person (a Java specialist for the Java rules and a C# specialist for the C# rules), a small group of specialists for one of the languages could iterate the rules together, i.e. using a beamer, and discuss the relevance and remediation cost for each of the rules. This approach is thought to provide a higher decision-quality, since consensus must be reached among the participants. Also, this approach is thought to be more fun than a single-person approach, since it will incorporate interesting discussions about what would normally be a very procedural task. It is recommended that these sessions do not last for many hours, but are split up in multiple smaller sessions due to the required attention and concentration.
4. Remove duplicate rules (which may exist if multiple metric tools used by Sonar incorporate the same metrics) from the rule set. The effect of duplicate rules is that violations are counted twice, which causes remediation costs to be reported to be too high.
5. Remove ‘unmatched rules’, i.e. rules that only exist for either Java or C#. If a rule does not exist for both languages, the violation will not be considered an issue in one of the languages, which leads to inconsistencies in quality assessments. An approach to implement this suggestion would be to take a list of all Java rules (‘list 1’) and a list of all C# rules (‘list 2’), and for each element on list 1 find an equivalent counterpart on list 2. If this is not found, remove the rule from list 1. When list 1 is completed, repeat the procedure with swapped lists. Of course, rules that are important but language-specific in the sense that the languages differ so that there can be no equivalent for a rule, should be kept in the model. Reason for this is that the coverage of potential problems is more important than the balance between C# and java assessments.

6. Reverse-engineer the assessment results by eliminating rules that have too much impact on the results. If a single rule constitutes a significant faction of the total remediation cost for a project, this could be due to the fact that the rule is not applicable. For example, a rule called 'Tabs must not be used' will have an extremely large number of violations if tabs are actually used by default for indenting source files and is irrelevant in that context. Analysis will show large remediation costs for unnecessary rules, which can then be eliminated.
7. Calculate the 'optimal' rule set by iterating all possible configurations. An approach would be to run analysis with all rules enabled, and register the remediation costs per rule per project in a to-be-written utility program that iterations the possible combinations of rules, calculates total remediation costs and the correlations with validation data, searching for the maximum correlation. Main problem with this approach is it's computational complexity, which is exponential in the number of rules. The number of violated rules is over 250, making it impossible to iterate all combinations in a reasonable amount of time. It is possible to approach the optimal configuration by ignoring all rules with total violation costs below a certain threshold, or by selecting the top-20 or top-30 of violated rules.

Note that some suggestions are quite labor-intensive to implement. For example, the removal of duplicates or unmatched rules have a quadratic complexity in the number of rules to consider, and the worst part is that it can only be performed manually. See Tabel 12 for an overview.

| Method | Worst case complexity (n=#rules, m = #characteristics) | Auto/manual |
|---|---|---|
| **Enable/disable rules** | O(n) | Manual |
| **Estimate Remediation cost** | O(n) | Manual |
| **Evaluate mapping** | O(nm) | Manual |
| **Remove duplicates** | $O(n^2)$ | Manual |
| **Remove unmatched rules** | $O(n^2)$ | Manual |
| **Reverse-engineer** | - | Manual |
| **Calculate optimum** | $2^n$ | Automatic |

**Tabel 12 Calibration method summary**

#### *3.5.4.2 Calibration methods: rules – sub characteristic mapping*

Aside from deciding which rules are important and which are not, an appropriate mapping between rules and sub characteristics should be configured. Since SQALE ratings are calculated per characteristic, it is more relevant to associate rules with the correct characteristics than which sub characteristic to use (actually, the concept of sub characteristics does not appear to have much purpose in SQALE). When defining the mapping, it is important to note the hierarchical concept of the characteristics as described in paragraph 2.4.6. It is recommended to perform the 'rule selection' phase of the calibration process before the 'rule mapping' phase. The following method applies:

Iterate all enabled rules, and per rule decide on the characteristic to map to by iterating the characteristics in inversed hierarchical order. For each characteristic, decide whether or not the rule influences the quality of that aspect of a software artifact. If so, map the rule. If not, go to the next characteristic. If a rule does not seem to apply for any characteristic, reconsider whether or not it should be enabled at all, since an enabled rule must be mapped to one characteristic.

To clarify this method, consider the following example: suppose we have chosen to include a rule named 'Missing Switch Default'. This Java rule is violated if a switch statement is used that has no

default section. To map this metric to a characteristic, we follow the following steps as depicted in Figure 25. The sequence of characteristics to iterate is consistent with the SQALE hierarchy (see Figure 12 on page 29). Also, the definitions of the characteristics need to be kept in mind (see appendix E). The flow through the flow chart is, in this case, as follows:

- A missing switch default does not impact the testability. Motivation: the effort needed to write (unit) tests does not increase or decrease when violating this rule.
- A missing switch default does impact reliability. Motivation: a switch statement is defined because some variable-value-dependent work needs to be done. A switch statement without default is likely not to perform any work on some variable values. Essential steps in the program flow may not be performed. This makes the software unreliable in case of unexpected input. Changes in the software that cause the input to the switch statement to change can especially make the system unreliable.

Note that the motivation aspect of each choice is inherently subjective; no determinate procedure to define the mapping is available. In general, be sure to use motivations that are reasonably convincing and not too far-fetched. Reason for this is that practically any rule could be claimed to indirectly impact testability, but allowing this to influence the mapping is unlikely to result in a balanced distribution of rules of characteristics.

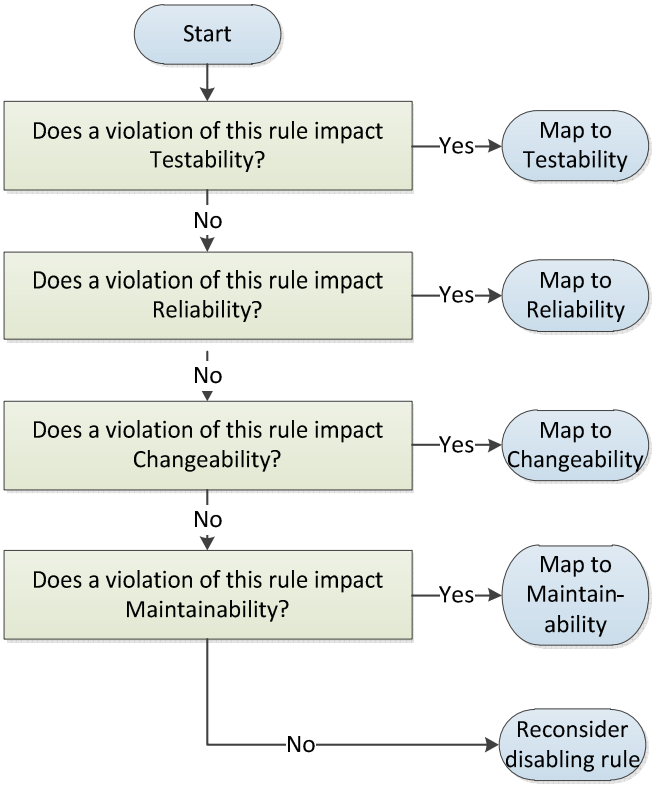



Figure 25 Rule - characteristic mapping flowchart

A different approach, which can be used with an existing, unbalanced configuration as a starting point, is the following:

- Given the list of rules associated to a characteristic that has relatively many rules, decide for each rule if it may be brought down one level in the hierarchy. For example, for all rules associated to the 'Reliability' Characteristic, decide whether or not the rule may also impact 'Testability'. In the configuration depicted in Figure 26, this will have a positive impact on the balance if this is the case for 1 or more rules, which seems likely.

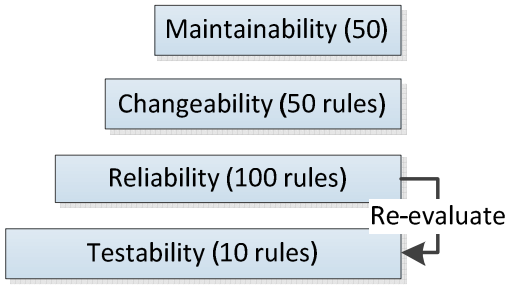



Figure 26 Re-evaluation example diagram

This approach has potential in cases in which characteristic definitions were interpreted too strictly (i.e. a too strong motivation is required before a rule is accepted to be associated with a characteristic), causing rules to be associated with characteristics that are higher in the hierarchy than needed. Loosening up the mapping using this method can have a positive impact on balancing, provided the investigated characteristic has a much higher rule count than the one below it in the hierarchy.

#### 3.5.4.3 Applied Configuration Procedure

For the purpose of this research, due to resource constraints we choose to perform a relatively simple calibration and use the results to formulate suggestions for further calibration. The calibration procedure is designed as follows:

1. Our starting point will a configuration with as many rules enabled as possible, and a default rule characteristic mapping.
2. Initially, we will use the reverse-engineering method to identify rules or rule parameters that are not applicable at Info Support, as described in the method description in the previous section. This should eliminate the worst invalid violations.
3. Using the resulting configuration, we will analyze the project set and present and analyze the results.
4. Further calibration will be performed by using Info Support knowledge to enable, disable and configure rules. We prefer to use configuration files for tools used by Sonar for this purpose.
5. Afterwards, a new analysis of the project set will be performed and again the results will be analyzed. From this, conclusions will be drawn about the calibration procedure, potentially leading to new ideas for further calibration.
6. Now that the rule set has been optimized, the next step is to optimize the rule-characteristic mapping. We will perform the second method described earlier to accomplish this., re-evaluating the mapping of rules associated with characteristics that have too many.

# 4. Validation Data Collection Results: Expert Opinions

## 4.1. Data Collection Process

Initially, the survey was sent by e-mail to 32 MITS employees. Approximately 50% of these employees had source code level knowledge of one or more of the selected projects. 11 employees filled out the survey. Additionally, a number of employees indicated, by e-mail or face to face that they were unable to answer any of the questions due to lack of source code level knowledge of all of the projects. In total, 22 project ratings where received. This is an average of 2.0 projects per employee who did return a survey form, and an average of 2.44 gradings per project. There was some variation in the number of responses per project. For an overview of raw survey results, see (confidential) Appendix F. Figure 27 shows a diagram with global statistical information. For each project, two values are listed:

- The number of respondents that rated the project, indicated in blue (left column for each project). Each of these respondents provided 4 ratings; one for each characteristic.
- The average variance in answers per characteristic, indicated in red (right column for each project). Note that for project H, the average variance is undefined due to the fact that there was only one respondent. The average variance of all other projects is 1.1.

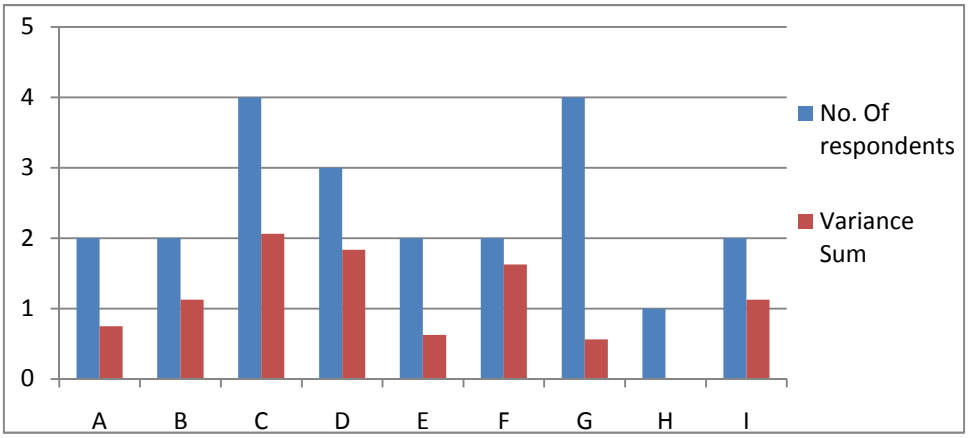



Figure 27 Response frequencies and answer variance per project

Since it is inherent to the MITS structure that for each project only a small number of specialized employees exist, the average of 2.44 gradings per project conforms to expectations. It has been verified that for the project with only one respondent, no other potential respondents exist.

## 4.2. Expert Project Ratings

To calculate project characteristic and overall ratings, we applied the calculation method as described in the research design. We use the confidence rating as a weighing factor (value 1 or 2) for a weighted average of the ratings per characteristic, and take the sum of the four ratings that are calculated this way, we get the project ratings. The scale for these ratings is [1..9], consistent with the survey definition. Note that the scale is 'reversed' to conform to the remediation cost paradigm; see research design section 3.2.2. The 'Overall' column shows the sum of the ratings of the four characteristics, again consistent with the remediation cost paradigm. We explicitly chose not to take the average rating, since this would make the format of the data inconsistent with SQALE

measurements and more difficult to compare, since SQALE also uses addition of remediation costs per characteristic to calculate overall quality (see 2.4.6).

| Project | Analyzability | Changeability | Reliability | Testability | Overall |
|---|---|---|---|---|---|
| A | 1.3 | 3.7 | 1.3 | 1.0 | **7.3** |
| B | 3.7 | 3.3 | 4.0 | 5.7 | **16.7** |
| C | 4.0 | 3.2 | 2.4 | 3.6 | **13.2** |
| D | 3.5 | 2.0 | 2.3 | 5.3 | **13.0** |
| E | 2.0 | 3.0 | 2.5 | 4.0 | **11.5** |
| F | 2.7 | 2.7 | 4.3 | 2.7 | **12.3** |
| G | 2.5 | 2.5 | 1.8 | 3.0 | **9.8** |
| H | 2.0 | 1.0 | 2.0 | 3.0 | **8.0** |
| I | 0.7 | 0.3 | 1.7 | 0.0 | **2.7** |

Table 13 Initial average project ratings from survey

## 4.3. Characteristic Correlations

We also investigated the correlation between the ratings provided by experts for all combinations of 2 out of 4 characteristics. The results are displayed in Table 14. It is interesting to see that all correlations are positive and have a value that we qualify as 'medium' to 'very high' according to research design section 3.5.1. This implies that if an expert rates a project high on one quality aspect, he/she is likely to provide a similar rating for other aspects. It is not necessarily the case that a project with a good rating for one characteristic will also receive a good rating for another characteristic when analyzed by tools. This could mean a few things:

- Experts are simply right and a project with a good score on one characteristic is likely to receive a good score on another characteristic as well. We should be able to test this by performing the same characteristic correlation analysis on quality measurements by Sonar.
- Experts give individual characteristic ratings that are implicitly based upon a general opinion on projects. For example, an expert may consider a project to be 'bad' in general, and when filling out the characteristic ratings may just indicate which aspect is even worse than another. If this is the case, and if project characteristic scores should not actually correlate, we will see this also in the Sonar measurement results.
- Another explanation is a conceptual overlap between characteristics. Experts may conceptually perceive the characteristic definitions to be non-distinct, or at least the boundaries to be unclear. This would imply a theoretical expectation of correlations between the characteristics.

| Characteristic 1 | Characteristic 2 | Pearson correlation | Interpretation |
|---|---|---|---|
| **Analyzability** | Testability | 0,84 | Positive, very high |
| **Analyzability** | Reliability | 0,52 | Positive, high |
| **Reliability** | Testability | 0,51 | Positive, high |
| **Analyzability** | Changeability | 0,47 | Positive, medium |
| **Changeability** | Testability | 0,38 | Positive, medium |
| **Changeability** | Reliability | 0,30 | Positive, medium |
| **(average)** | | 0,50 | Positive, high |

Table 14 Expert Characteristic Score correlations

# 5. Validation Data Collection Results: Financial Indicators

In this research phase, we have collected financial data to determine a financial quality indicator for each of the projects we have investigated. We have used the indicator as defined in paragraph 3.3.1.

A number of issues was encountered while attempting to retrieve the necessary data. Details of these issues are left out of the unrestricted version of this thesis.

For these (confidential) reasons, we should keep in mind the possibility that the reliability of this validation data low. Within the context of this research, no possibilities to increase the reliability were identified. Due to the redundant nature of tool assessment validation (using both expert opinions and financial data), we choose not to redesign this research aspect to overcome this potential lack of reliability.

Using the methods above, we managed to retrieve a quality indicator value for each project. The amount of hours is the sum of the registered management hours in the categories 'problems' and 'incidents', in 2010. For Project I, no incidents were registered, so the value is 0.0. Note that due to the linear relationship between hours and money, these should be considered equivalent and we do not have to perform transformation calculations.

| Project | Project size (LOC) | Total relevant mgmt. hours (x) | No. of incidents | x/KLOC |
|---|---|---|---|---|
| A | 51,888 | 60 | 21 | 1,127 |
| B | 83,313 | 31 | 7 | 0,377 |
| C | 62,666 | 36 | 19 | 0,574 |
| D | 62,293 | 7 | 5 | 0,057 |
| E | 128,664 | 172 | 17 | 1,336 |
| F | 51,629 | 85 | 11 | 1,646 |
| G | 64,189 | 276 | 25 | 4,300 |
| H | 12,035 | 15 | 1 | 1,246 |
| I | 2,377 | 0 | 0 | 0,000 |

Table 15 Financial Quality per Project

# 6. Proof of Concept

This chapter shows how an attempt to setup Sonar, with all required tooling to assess both Java and C# projects using the SQALE method, was successfully setup.

## 6.1. SQALE implementation in Sonar

Some implementation-specific remarks on the SQALE plugin for Sonar are to be noted. Additionally, some bookkeeping activities need to be performed in order to effectively use the setup.

### 6.1.1. Implementation characteristics

The Sonar tool uses a plugin to implement the SQALE model. This plugin conforms to the SQALE description as provided in background information section 2.4.6 and described in (Sqale 2011) with the following implementation-specific remarks:

- The implementation does not implement the 'Reusability' characteristic. This presents no problem, since the selected subset of characteristics to use does not include 'Reusability' (see research design section 3.2.2);
- The implementations allows the user to remove all of the characteristics individually, while the standard considers some of them to be mandatory;
- The implementations remediation function options are limited to a constant value (expressed in units of time) per violation or per file containing one or more violations, while the standard allows for more complex functions to be used.

The SQALE definition provides a small set of Conformity Criteria. It has been verified that the implementation meets these criteria.

### 6.1.2. Mapping of SQALE characteristics to quality definition

Before we can use our proof of concept setup, some bookkeeping needs to be performed. The quality indicators used in the quality model of the tools as well as in the expert opinion survey do not fully match the definition of quality as presented in paragraph 1.4.1. This discrepancy exists due to the fact that a definition of quality was part of the initial research assignment, while during the initial research phase it was found that tools and models do not fully conform to this definition. Table 16 presents the two definitions. The definition of ISO 9126 characteristics can be found in (Chamillard 2005). The SQALE quality model definition document (Sqale 2011) does not provide a clear definition of the model's characteristics in text. For this reason, we will need to use information on the default sets of assigned sub characteristics to assess this definition. In Table 16, the last column lists these sub characteristics and names, as an example, some assigned metrics or metric categories.

| **Quality (by definition of this research)** | **Definition:** *the capability of the software product…* | **Quality (as used in tools based upon SQALE)** | **Assigned sub characteristics** |
|---|---|---|---|
| **Analyzability** | … to be diagnosed for deficiencies or causes of failures in the software, or for the parts to be modified to be identified. | Maintainability | Readability, understandability (i.e. conventions for naming, lengths and code layout) |
| **Changeability** | … to enable a specified modification to be | Changeability | Architecture, data and logic related changeability (i.e. metrics |

| | | | |
|---|---|---|---|
| | implemented. | | concerning coupling, visibility, abstraction, structure of if-statements) |
| **Stability** | … to avoid unexpected effects from modifications of the software | Reliability | Architecture, data, instruction, logic and synchronized related reliability, exception handling, fault tolerance, unit tests (i.e. metrics concerning bad coding practices, unit test coverage) |
| **Testability** | … to enable modified software to be validated | Testability | Integration level and unit level testability (i.e. parameter number, cyclomatic complexity |
| **Maintainability Compliance** | … to adhere to standards or conventions relating to maintainability | | |

Table 16 Qualiy Definitions Overview

A mapping between these two sets of characteristics is displayed in Figure 28. This mapping is 'own work' and based upon our experience with ISO 9126 and SQALE acquired in this project. As with any mapping, a few comments should be made:

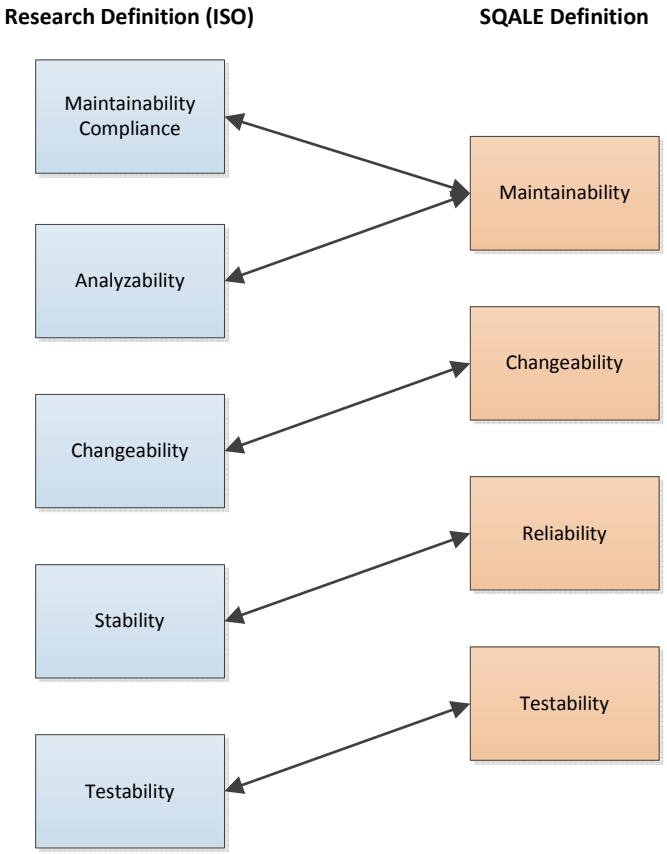



Figure 28 Mapping of defined quality aspects to SQALE characteristics

- The 'maintainability compliance' aspect is not mapped to its 'own' SQALE characteristic. When looking at the metrics assigned to the 'maintainability' characteristic of SQALE, we see that metrics concerning the naming and length of field and methods are covered there. This indicates that the intention of the 'maintainability compliance' aspect is, as well as 'analyzability', covered by the 'maintainability' characteristic. We therefore choose a 2-to-1 mapping here.
- The idea behind the 'testability' aspects seems to differ slightly between the standards. In the ISO definition, testability is about the ability to test, while in SQALE, testability appears to be about the ability to *develop* tests. A result of this is, for example that 'unit test coverage' is part of the Reliability characteristic instead of Testability. Since in SQALE, reliability implies testability (see 2.4.6), we do not consider this to be a problem. If we did, a solution could be to use the flexibility of SQALE to reassign the coverage metrics to a testability sub characteristics.

- The fact that SQALE has a ‘maintainability’ characteristic could be confusing. The definition of this characteristic is more narrow that the name might imply, which allows us to map it to our ‘analyzability’ aspect of the ISO standard, which is in itself a sub aspect of our definition of maintainability as defined in paragraph 1.4.1. This way, we have made the narrowly defined SQALE-maintainability characteristic an aspect of our broadly-defined ISO maintainability. All left-column items in Figure 28 constitute maintainability according to our own definition.

### 6.1.3. Relationship with ISO 9126

The relation between the ISO 9126 quality characteristics and SQALE is as follows and depicted in Figure 29 (page 67):

The maintainability characteristic of the ISO standard is the root of the SQALE tree. The full subtree of ISO maintainability is also part of the SQALE tree, where it should be noted that ‘Maintainability compliance’ is covered by the analyzability sub characteristic. Not that in SQALE, the term ‘Maintainability’ is used for what ISO calls ‘Analyzability’. ISO characteristics Functionality and Usability cannot be covered by source code analysis, since these concern the extent to which software meets requirements and the extent to which users can efficiently use this functionality through the user interface. The Reliability aspect of ISO is covered by the SQALE’s Reliability aspect. On a source code level, ‘stability’ can be considered equivalent to ‘reliability’, which is the reason why these are also linked. Efficiency and Portability can optionally be enabled as part of the SQALE standard, but are not part of the ‘maintainability index’ as defined in the original assignment (see 1.1).

Figure 29 depicts the mapping between ISO 9126 and SQALE. Red lines show equivalence relations. Note that in a few occasions there is a many-to-one relation. This is due to the fact that when we limit ourselves to source-code based analysis, some concepts become equivalent. Green colors indicate mapped characteristics, yellow colors indicate optionally mapped characteristics and red characteristics cannot be mapped due to reasons stated earlier.

Note that no explicit mapping is available from literature. The mapping presented here is ´own work´, derived from the definitions of both the ISO standard and SQALE and our experience with both as obtained in this project.

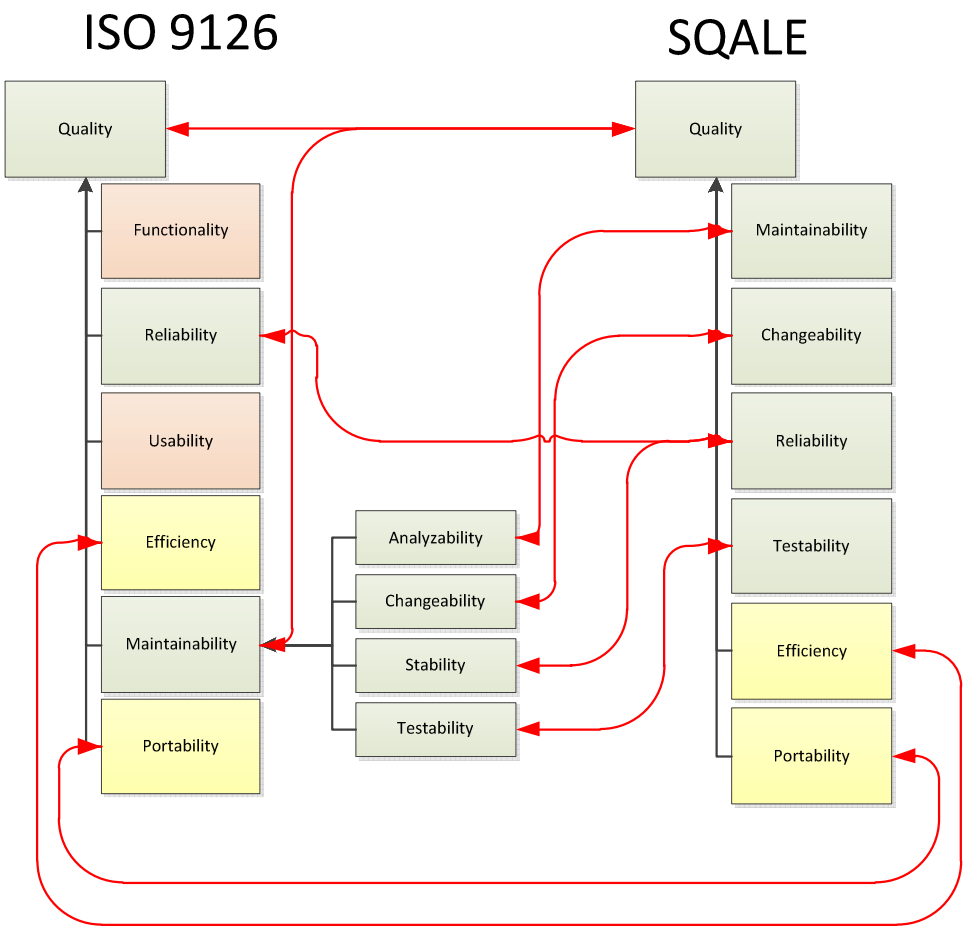



Figure 29 ISO 9126 – SQALE characteristic equivalence relations.

## 6.2. Setting up the Proof of Concept

We configured the Proof of Concept and identified a number of issues that was encountered and the solution to them. This basically concern a number of rules that can only work under conditions that were not met in the provided context. Also, we make some first remarks on experiences with SQALE. Although a number of shortcomings is identified, in general SQALE works for its purpose. The setup results in a working SQALE environment that can be used to assess Java and C# projects, which has successfully been performed for the sample projects.

### 6.2.1. Initial Sonar Setup attempt

In an early stage of this research, an attempt was made to setup the Proof of Concept for the Sonar tool, as defined in the research design. The reason to perform this attempt is that the implications of the non-native support for .Net in sonar could not be clearly seen by a pure theoretical approach, as indicated in background section 2.6.2.2. Full technical documentation of the steps taken in the attempt can be found in appendix F. A graphical overview of the technical setup used for the Proof of Concept can be found in appendix H. This setup constitutes for an important part the method that is the goal of this research from the Info Support perspective.

For this purpose, we used a clean installation of Microsoft Windows XP SP3, with all updates applied, on a Virtual PC (Info Support unattended install as of March 7, 2011). Although the general setup of

Sonar (including Sonar, a JDK, a MySQL database and Maven and connecting them together) went quite well, a number of issues were encountered while attempting to realize .Net support:

- A substantial number of plugins and utilities is required in order for .Net support to be enabled. An overview can be found in the installation notes in appendix G.
- In some cases, dependencies caused version-issues. Some versions of plugins may have compatibility issues with other versions of plugins. The plugin website states a recommended combinations of versions.
- Although the SQALE quality model plugin and .net plugin cooperate, no default quality model parameter set is available. This configuration needs to be done manually. To obtain measurements results that are similar for java and .net projects, the java quality model parameters used need to be imitated as closely as possible in the .Net parameter set.

Although it is possible to create a working setup in which Sonar can evaluate .Net products at this point in time, a risk is involved in using a complex structure of dependencies. For example, a dependency may seize to be developed, supported or available or may not be developed fast enough to allow for the newest .Net techniques to be used. Therefore, .Net support in Sonar is, at this point in time, not considered to provide an optimal solution for a business context. However, as the investigation has shown, there are no alternatives at this time given the constraints. We will therefore use Sonar with .Net support in our proof of concept, and elaborate on the applicability in a production environment based upon our experience.

### 6.2.2. Sonar SQALE Quality Model Settings

Although the SQALE quality model plugin and the .NET plugin for sonar cooperate without problems, there is no default mapping between C# source code metrics and SQALE characteristics. An investigation of this issue led to the conclusion that it is not possible to fully imitate the Java mapping for C#, due to the fact the Sonar internally uses different tools to perform source code measurements for both languages. Table 17 provides an overview of metric tools used by Sonar. The Java tools are a standard component of the default Sonar installation. The C# tools are dependencies of the Sonar .NET plugin. An exception is the SQALE plugin, which is optional for both Java and C#. For Java, Sonar has added eleven rules to the ones found in the tools. The values behind tool names in the table indicate the number of metrics provided by that tool.

| Java tools (745 rules) | C# tools (604 rules) |
|---|---|
| Checkstyle (122) | FxCop (240) |
| Findbugs (384) | Gendarme (216) |
| PMD (224) | Stylecop (144) |
| SQALE plugin rule set (4) | SQALE plugin rule set (4) |
| Sonar (native rule set) (11) | |

Table 17 Metric tools used by Sonar

To be able to use a quality model that consistently assesses both Java and c# projects, we have the following two basic options to choose from:

- Start from scratch. This means that for each of the existing metrics for both languages, we decide 1) if it is relevant and 2) should and does it have an equivalent in the other language. If the answer to both questions is 'yes', for both languages the metrics can be assigned to a selected quality model sub characteristic. Drawbacks of this approach are the non-trivial nature of the metric comparison; their names and descriptions do not allow for 1-to-1 mapping and equivalence for languages is not easily established. Also, the large number of available metrics makes this a very labor-intensive approach.
- Use an existing configuration as a basis and adapt it if needed. In this case, we 'just analyze' projects using an existing configuration. We can assume the validation data to be true, and adapt to model to see if we can make it produce output for which the correlation expectations from section 3.5 can be verified to be met. Main drawback is this approach is that the existing configuration may significantly differ from what we actually want, which causes the amount of effort needed to make modifications ('calibration cost') to become high. Also, we were able only to obtain one existing configuration that incorporates both Java and C# metrics. It was obtained from the author of the .Net plugin for Sonar.

The estimated efficiency of the second approach is thought to be highest. The configuration aims, according to the authors, at making comparable assessments of java and c# projects, which is what we need. We were able to install the obtained configuration without significant difficulties.

### 6.2.3. .Net Project Setup

The process of making .Net projects suitable for analysis by Sonar requires some extra steps to be taken. .Net project may consist of a number of Visual Studio Solutions and for each Visual Studio Solution, a definition file (.sln) exists. The Info Support projects used in the Proof of Concept consist, on average, of about 8 solutions (estimate). Sonar cannot aggregate quality measurements of projects spread over multiple solution definitions. Fortunately, a tool was found that can be used to

merge solution definitions into one definition file before starting Sonar analysis. This tool was found at Google Code and is used in the Proof of Concept. The tool is also part of the method begin developed in this research. See the Digital Resources section in appendix C for more information.

During the use of this tool, no issues were encountered that were reason to believe that it did not operate as it should. It has been verified that the merging process includes all components of multi-solution .Net projects, which is its main purpose. Also, source code of the tool is available, so that further functionality verification, or modification, is possible.

### 6.2.4. Initial Quality Model Configuration Calibration & Setup Test

#### *6.2.4.1 Basic Configuration*

Initially, all rules have been enabled. A full list of available rules is provided as a digital resource, see appendix C. Some of them, however, caused problems when attempting to run the analyses. These rules have been disabled. Additionally, some rules have been disabled or modified because they had an impact of analysis results that was out of proportion to the actual problem (i.e. an increase of remediation cost in orders of magnitude) or had duplicates. These are considered 'trivial configuration enhancements'. All changes to the initial configuration are listed in Table 18.

| Rule name | Reason for disabling/modifying |
|---|---|
| **Avoid too complex class (Java)** | Cause NullPointerExceptions, replaced by Cyclomatic Complexity Rule. Known bug, registered under number SONAR-2301 created March 25, 2011. |
| **Avoid too complex method (Java)** | |
| **Avoid too deep inheritance tree (Java)** | Causes NullPointerException, no direct replacement identified. Reported to Sonar developers. |
| **Header (Java)** | These rules can be used to match specific (regular) expressions in headers or bodies of files. They need to be configured before they can be used. A violation can be defined for cases in which a file does or does not contain the header or regular expression. We have no use for these rules at this moment. |
| **Regexp (Java)** | |
| **Regexp Header (Java)** | |
| **Regexp Multiline (Java)** | |
| **Regexp Singleline (Java)** | |
| **Regexp Singleline Java (Java)** | |
| **XPath rule template (Java)** | Not an actual rule but a template for custom rules, cannot just be enabled. |
| **AvoidCodeDuplicateInSame-ClassRule (C#)** | Too rigid to be workable; single-line copies are already a violation. Also: insufficient memory to perform analyses and no Java alternative. Replaced by a.o. a SQALE rules that works both for Java and C# ('Duplicated Blocks' rule). |
| **AvoidCodeDuplicateInSibling-ClassesRule (C#)** | |
| **Tabs must not be used (C#)** | Constitutes a disproportional share of remediation costs (over 1/3), while Tabs are no problem at Info Support. Also, there is no Java equivalent. |
| **DesignForExtension (Java)** | This is a very interesting rule that requires nonprivate nonstatic methods of nonfinal classes (practically most methods) to be either abstract, final, or have an empty implementation. Documentation states that *this API design style protects superclasses against being broken by subclasses. The downside is that subclasses are limited in their flexibility, in particular they cannot prevent execution of code in the superclass, but that also means that subclasses cannot corrupt the state of the superclass by forgetting to call the super method.* Whether or not this is a 'good' rule could be a paper subject itself. At |

| | |
|---|---|
| | Info Support, this programming rule is not applied. Violations constitute a large part of remediation costs (upto about 1/3) only for Java projects. We choose here to disable this rule. |
| **Strict Duplicate Code (Java)** | This rule has a duplicate, namely the Duplicated Blocks rule from the SQALE plugin. |
| **Dataflow Anomaly Analysis (Java)** | Has too much impact on the quality readings due to too many violations of type 'a recently defined variable is redefined'. Documentation states that this does not have to be a bug. This rule is not applied at Info Support. |
| **Method Argument Could be Final (Java)** | This rule covers a subset of 'Final Parameter' and is therefore a duplicate. |
| **Local Variable could be final (Java)** | This rule covers a subset of 'Final Parameter' and is therefore a duplicate. Is also a copy of 'final local variable' |
| **Final local variable (Java)** | This rule covers a subset of 'Final Parameter' and is therefore a duplicate. Is also a copy of 'local variable could be final' |
| **Final Parameter (Java)** | This rule has a major impact on the remediation costs; it is apparently not an Info Support quality policy. |
| **Prefix local calls with 'this' (C#)** | No java equivalent available, major remediation cost impact. |
| **Identifiers should be spelled correctly (C#)** | Some of the sample projects use Dutch names for classes, methods and variables while this rule only recognizes English names. It is possible to add a Dutch dictionary, but no dictionary could be found. Also, no Java equivalent was identified. We therefore chose to disable this rule. |
| **Identifiers should be cased correctly (C#)** | Causes a violation for each occurrence for each incorrectly cased variable. Due to source code refactoring possibilities this yield unrealistic remediation costs. The remediation costs per violation for this rule have been changed from 0.01 days to 0.001 days. |
| **Hidden field (Java)** | This rule checks that a local variable or a parameter does not shadow a field that is defined in the same class. This rule has not been removed, but a parameter has been set to disable if for setters and constructor parameters, because method shadowing is done by default in these situations in Info Support java projects. |
| **Elements must be documented (C#) / Javadoc (java)** | Remediation costs changed from 0.03 days to 0.01 days. The amount of time needed to document a local variable, the most common violation, is far less than 14 minutes. The original remediation cost had too much impact on measurements. |
| **Curly Brackets must not be omitted (C#)** | No policy and no Java equivalent. |
| **Unit Test Coverage (both Java and C#)** | Some sample projects use a very deprecated NUnit test frame work not supported by Sonar. To get consistent measurements, we choose to disable the coverage rule. |

Table 18 Modified rule list

Disabling the Unit Test Coverage rule was needed due to the face that some (n=2) of the .Net project use a (very) deprecated NUnit framework which is not supported by Sonar. For consistency reasons, we chose to disable to rule for all measurements. It should be noted, however, that if measured for other projects coverage percentage was found not to present a problem. Info Support already monitors coverage and has strict regulations concerning minimum coverage. This is reason to believe that disabling the rule does not impact research results.

### 6.2.4.2 Model Calibration

To calibrate the model, we follow the steps as indicated in research design paragraph 3.5.4. The number of rules attached to each characteristic for both Java and C# are listed in Table 19.

| Characteristic | Java | C# | (total) |
|---|---|---|---|
| **Maintainability** | 205 | 251 | 456 |
| **Changeability** | 36 | 13 | 49 |
| **Reliability** | 332 | 170 | 502 |
| **Testability** | 13 | 9 | 22 |
| (total) | 586 | 443 | 1029 |

Table 19 Rules per language per selected characteristic

This configuration is reflected in the Sonar interface, as can be seen in the screenshot displayed in Figure 30. Note that the interface shows all rules that have been attached to a quality characteristic. A number of them, in particular rules that have been listed as disabled according to Table 18, are not used in the quality calculations. Formally, this can be put as follows:

1. The set of *enabled rules* is a subset of the total set of *available rules*
2. The set of rules that are attached to a SQALE characteristic (‘*attached rules*’) if a subset of the total set of available rules
3. The intersection of the set of *enabled rules* and *attached rules* is the set of rules that is used in quality assessment (‘*used rules*’).

The conforming Venn diagram is displayed in Figure 31. Additionally, Figure 30 displays the hierarchy of attached rules in the current configuration.

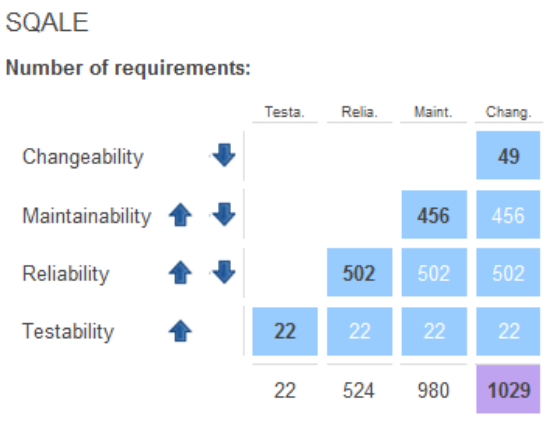



Figure 30 Sonar SQALE configuration screenshot

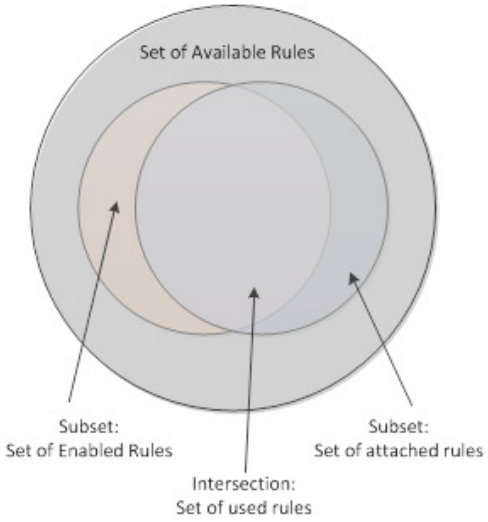



Figure 31 Venn diagram of Sonar Rules

A full overview of the configuration can be found in appendix I. This appendix displays the full tree structure in in the configuration state described.

### 6.2.4.3 SQALE indexes

Using the described setup, we performed an initial analysis of the sample project. This resulted in the SQALE indexes (remediation cost per characteristic and in total divided by total development cost, see 2.4.6) as displayed in Table 20.

| Project | Analyzability | Changeability | Reliability | Testability | Total |
|---|---|---|---|---|---|
| A | 0.07 | 0.01 | 0.05 | 0.03 | 0.16 |
| B | 0.07 | 0.00 | 0.12 | 0.00 | 0.19 |
| C | 0.09 | 0.02 | 0.06 | 0.05 | 0.23 |
| D | 0.03 | 0.00 | 0.03 | 0.01 | 0.07 |
| E | 0.05 | 0.01 | 0.04 | 0.04 | 0.13 |
| F | 0.07 | 0.00 | 0.02 | 0.01 | 0.10 |
| G | 0.08 | 0.07 | 0.03 | 0.02 | 0.20 |
| H | 0.05 | 0.01 | 0.02 | 0.02 | 0.10 |
| I | 0.03 | 0.04 | 0.01 | 0.01 | 0.09 |

Table 20 Initial Sonar measurements

#### 6.2.4.4 SQALE experiences

Using this configuration and a customized Sonar Dashboard of which an example can be found in appendix D, initial analysis attempts have been conducted. Based upon the results of these attempts, the following experiences and shortcoming of the SQALE model were identified.

- As can be read in the quality model description (2.4.6), SQALE ratings are calculated by dividing the remediation cost by the total development cost. The result of these divisions ('indices') are mapped to a discrete 5-point scale ('ratings') that defines the final judgment. The number of rules per characteristic varies. A logical consequence is that the remediation costs for a characteristic with many rules will usually be higher than the remediation cost for a characteristic with not many rules. Because only one mapping is used, this results in higher ratings for characteristics with a lower rule count. For the initial analysis results, in seemed impossible to define a mapping that yields 'nice' scores for all characteristics, nice meaning that for no characteristics most of the projects score either 1 or 5. We overcome this issue by using the ' raw' value for remediation cost divided by total cost in the analysis, not applying this mapping.
- The overall rating is defined by the addition of the remediation costs of all enabled characteristics, divided by the total development cost. The resulting index is mapped to a rating using the same mapping as is used for individual characteristic. The consequence of this is that the overall quality rating, on a 1-5 scale, is always equal to, or worse than, the worst quality characteristic rating. For example, a project with a rating of '2' on all characteristic, may have an overall rating of '4'. This may not be what one would probably expect. A possible solution is to just ignore the overall rating, and only look at the characteristic-specific ratings instead. An average of the ratings instead of a summation could be used as a replacement for the overall characteristic.
- Although Sonar associates a severeness property with each rule, severeness of violations is not taken into account when calculating SQALE ratings. This conforms to the SQALE definition, but can be considered a lack of information. This issue is addressed in a proposal for an extension of SQALE, see paragraph 7.5.2.2.

# 7. Analysis & Optimization

This chapter provides a more in-depth analysis of the results obtained, conforming to research design section 3.5. The analysis shows that SQALE measurements, as obtained in chapter 6, correlate with validation data – expert opinions and financial quality indicators – to an acceptable extend. The correlation can be enhanced by optimizing the SQALE configuration to match Info Support development policies. Major side nodes concern the difficulty of determining what is a 'good' configuration. We also state that a higher correlation value does not necessarily imply a higher quality of the quality model configuration. We elaborate upon these summarized findings in the following sections.

## 7.1. Validating the Validation data

Before presenting an analysis of the correlation of SQALE measurements with validation data, it should be noted that the survey results and financial quality indicators together have no significant correlation. This finding is illustrated in Figure 32.

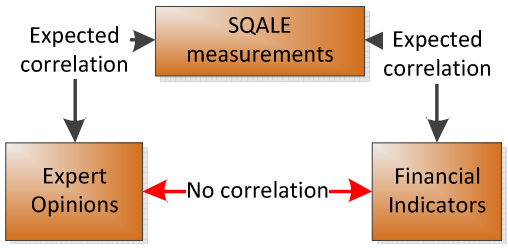



Figure 32 Validation data correlation

This correlation can easily be calculated from the overall quality ratings of the sample projects as provided by experts (section 4.2) and the financial indicators (chapter 5). Correlating the data in Table 13 on page 62 and Table 15 on page 63 provides the following information:

| | Pearson value |
|---|---|
| Survey results vs. financial quality | -0.03 |

Tabel 21 Validation data correlation

The value for this correlation, -0,03, is to be interpreted as 'insignificant' in accordance with research design paragraph 3.5.1. From this, we must draw the following conclusion:

- At least one of the two sets of validation data (expert opinions or financial indicators) is apparently unreliable, and cannot be used to assure the validity of quality assessment by Sonar SQALE.
- Due to the different nature of the sets of validation data and the problems that arose when obtaining the data needed to calculate financial indicators, as indicated in chapter 5, it seems reasonable to assume that the financial indicator values are the ones being unreliable. If we assumed those were reliable, it would follow that the expert opinions are unreliable, which is considered less likely.

In the remainder of the statistical analyses, we will calculate correlations of SQALE measurements with both expert opinions and financial indicators, but we will keep in mind that the financial indicator values are to be considered unreliable.

Note that intuitively, one may expect that given datasets A, B and C, a positive correlation between A and B and a positive correlation between B and C together imply a positive correlation between A and C, that is, that correlation is transitive. This is, however, a common misconception (Sotos 2007).

## 7.2. Initial Validation Results

### 7.2.1. Calculating Correlations

Using the Proof of Concept setup from chapter 6.2 and the validation data from chapters 4 and 5, an initial attempt to establish correlations has been performed. The results of this attempt are described in Table 22. This table has the following columns:

- Correlation: the name of the correlation we calculated. This takes the form 'A vs B', meaning that the correlation of A and B is calculated. Correlations 1, 6 and 7 concern the total data sets of Sonar (SQALE) results, survey results and financial results, while correlations 2 through 5 'zoom in' on specific quality characteristics and indicate the correlations between Sonar results and survey results for those individual characteristics.
- Value: the Pearson correlation coefficient values
- Expected: the expected correlation value ranges as defined in research design section 3.5.1.
- Conclusion/remarks: remarks about the findings, which will be elaborated upon later.

| Correlation | Value | Expected | Conclusion/remarks |
|---|---|---|---|
| **Sonar results vs. Survey results (overall)** | **0,41** | **>= 0,30 (medium)** | **As expected** |
| - Sonar results vs. Survey results (analyzability) | 0,45 | >= 0,30 (medium) | |
| - Sonar results vs. Survey results (changeability) | -0,28 | >= 0,30 (medium) | Language inbalance |
| - Sonar results vs. Survey results (reliability) | 0,46 | >= 0,30 (medium) | |
| - Sonar results vs. Survey results (testability) | -0,03 | >= 0,30 (medium) | Not enough rules |
| **Sonar Results vs. Financial results (overall)** | **0,34** | **>= 0,30 (medium)** | **As expected** |
| **Survey results vs. Financial results (overall)** | **-0,03** | **>= 0,10 (low)** | **Not as expected** |

Table 22 Initial correlation values

A visualization of the correlations is displayed in Figure 33 using scatter plots. The following legend applies to this figure:

- EQI = Expert Quality Indicator (Survey results). The scale indicates the sum of remediation cost of the four quality characteristics (horizontal axes in all charts except top-center)
- FQI = Financial Quality Indicator (Financial investigation results). The scale indicates the amount of hours spent per KLOC (horizontal axis in top-center diagram, vertical axis in top-right diagram)
- TQI1 – Tool Quality Indicator 1 (Sonar results with initial configuration). The scale indicates the remediation cost divided by total cost (vertical axis in all charts except top-right).

The upper diagrams show the global correlations between the three data sets. The lower four diagram show the correlations between survey results and Sonar measurements for each characteristic

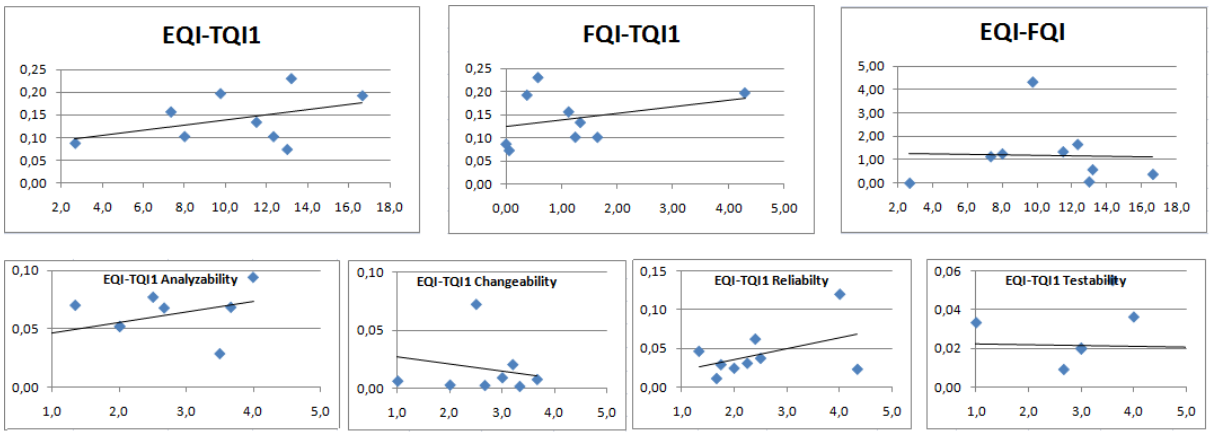



Figure 33 Initial Correlation Scatterplots

From this table and figure, the following initial observations can be made:

- The expectation that there is a 'medium' correlation between expert opinions and Sonar assessment results is true.
    - The result vary per characteristic. The analyzability and reliability characteristics yield acceptable correlation values.
    - For Testability, there is no correlation. This issue is addressed to the low number of rules assigned to this SQALE characteristic, which causes the occurrence of violations to become somewhat coincidental. This issue can be addressed in attempts to optimize the configuration.
    - The Changeability character yields a negative correlation. This can presumably be explained by the assumption that using the current configuration, Sonar is 'very tough' for Java projects, which are ranked high by expert. This is supported by the fact that the correlation is very sensitive to the Java projects and increases from -0.28 to +0.25 by just removing the Java projects from the data set. This issue can be addressed in attempts to optimize the configuration.
- Despite the low quality of the financial data retrieved, the hypothesis that the correlation between this data and Sonar results is at least 0.30 is still true.

Note that at least a +0.26 value is needed for a statistically significant positive correlation with $p < 0.25$, while at least a +0.47 value is needed for a statistically significant positive correlation with $p < 0.10$, in accordance with paragraph 3.5.2. This means that we can state that most correlations exist with a certainty of over least 75%.

### 7.2.2. Sensitivity

In general, the diagrams show the correlations to be quite sensitive; the removal of only one project can have a major impact on the correlation coefficient. To make the sensitivity measurable, we recalculated all correlations 9 times and each time removed one of the nine projects from the sample set, i.e. we tried all proper subsets with (n=8) as a sample data set. The results have been visualized in a graph, which is displayed in Figure 34. The vertical axis indicates the Pearson value range, while the horizontal axis displays all projects. A value on the horizontal axis indicates the subset of sample projects in which that project is omitted, i.e. value 'D' indicates the set {A..I} \ {D}. The values in the legend of the diagram are the correlation coefficients when using the total project set (n=9).

The more horizontal a line is, the more stable it is. Peaks in the graph (either negative or positive) are caused by projects that have a high impact on the correlation value. These are usually projects that lie far away from the trend lines in the scatter plots (Figure 33). For example, the 'Reliability' line (orange) is heavily impacted by the project B (in a positive way) and Project F (in a negative way). Removal of these lead to a negative and positive peak in the graph, respectively.

Note that the relationship between projects and impacted correlations appears to be quite random; there is no single project that has a major impact on many correlations in the same direction.

The average variance for all correlations is 0,026. After calibration of the model has been completed, we expect to see less sensitive relations.

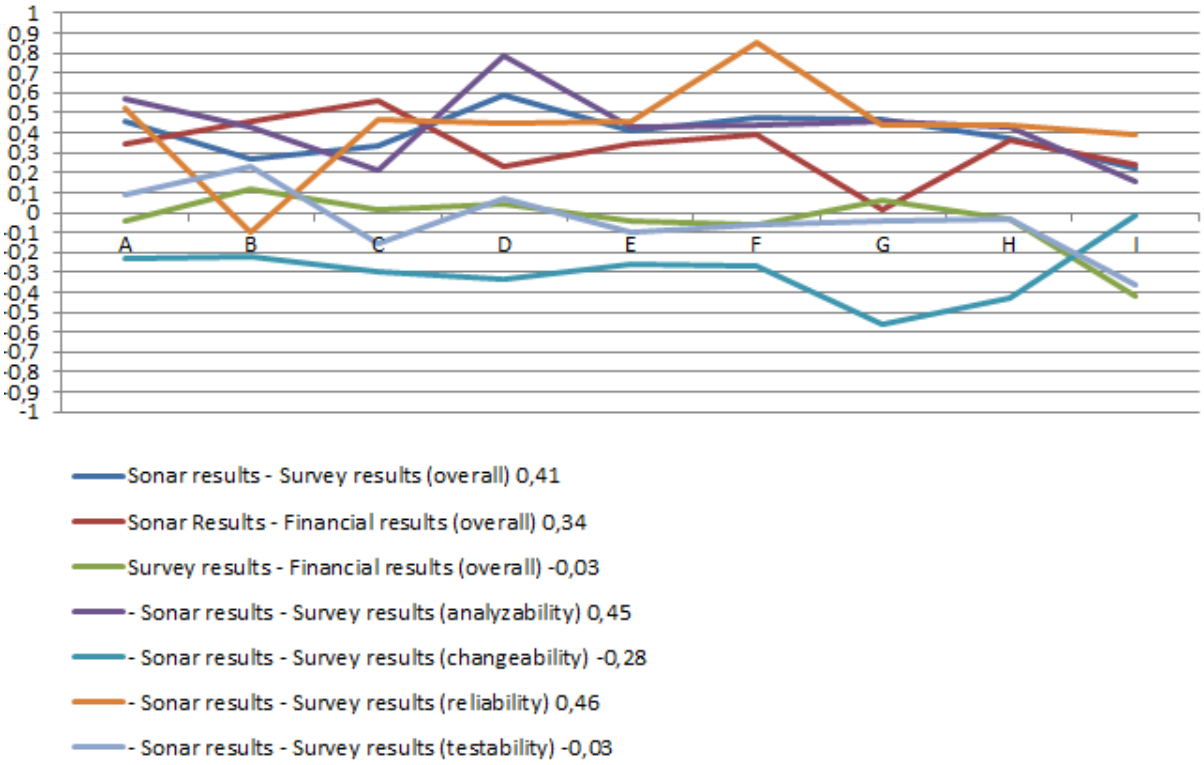



Figure 34 Sensitivity of initial findings

### 7.2.3. Correlations of Characteristics

As a last step of the initial analysis, we calculated the correlation between the scores given by Sonar for different characteristics, just as we did with the expert survey results (see 4.1). This yields the information displayed in Table 23.

| Characteristic 1 | Characteristic 2 | Pearson correlation | Interpretation |
|---|---|---|---|
| **Analyzability** | Changeability | 0,11 | Positive, low |
| **Analyzability** | Reliability | 0,45 | Positive, high |
| **Analyzability** | Testability | 0,57 | Positive, high |
| **Changeability** | Reliability | -0.33 | Negative, medium |
| **Changeability** | Testability | 0,00 | Insignificant |
| **Reliability** | Testability | 0,02 | Insignificant |
| **(average)** | | 0,14 | |

Table 23 Characteristic correlations for initial Sonar run

We see that the correlations vary a lot, which is another sign of unbalance in the model configuration. The average correlation is positive, but low. We expect to see higher correlations between scores for individual characteristics, like the ones we have found in the expert opinions (see 4.1). We will reflect on this issue in the reflection section (7.4).

## 7.3. Calibrating the configuration: applying Info Support rule set

We were able to optimize the quality configuration by applying Info Support programming policies. This attempt led to an increase of correlation values and stability. This section describes the process and results.

### 7.3.1. Reconfiguring the rule set

As a next step in the calibration process, we chose to limit the set of rules used in the SQALE configuration by applying only those rules that are used at Info Support. In the Endeavour environment, Info Support uses CheckStyle (Java), FindBugs (Java) and FxCop (C#) rules. Conveniently, this is a proper subset of tools internally used by Sonar.

For CheckStyle and FindBugs, configuration files were obtained from Info Support. For FxCop, information was obtained indicating that all rules should be enabled. The only exception are spelling rules that only support the English language. All rules of 'rule engines' other than CheckStyle, FindBugs and Fxcop were disabled.

This lead to the SQALE configuration as displayed in Table 24. The rule-characteristic mapping is still configured as it was in the initial validation attempt, since Info Support does not have any mapping information available. Clearly, this is not a balanced configuration. The distribution of rules over characteristics, as well as the balance between languages, is not likely to result in a balanced judgments of all quality aspects of assessed projects.

| Characteristic | Java | C# | (total) |
|---|---|---|---|
| **Maintainability** | 73 | 60 | 133 |
| **Changeability** | 7 | 3 | 10 |
| **Reliability** | 125 | 69 | 194 |
| **Testability** | 3 | 1 | 4 |
| (total) | 208 | 133 | 341 |

Table 24 SQALE configuration using Info Support ruleset

### 7.3.2. Reconfiguring the rule – characteristic mapping

To overcome the issue of lack of balance between characteristics, we attempted to redistribute the rules before performing analysis. An issue that arose in this attempt, was that the natural distribution of rules of characteristics is inherently unbalanced, the natural distribution being defined as a logical choice for a characteristic for each rule, as assessed by the author of this work. So, for one characteristic there exists many more logically associated rules than for another. This has the following consequences:

- A redistribution of rules over characteristics that results in a balanced configuration will have a less logical mapping than the initial configuration. This blurs the distinction between

characteristics and reduces the quality of the provided ratings themselves, which is an unwanted effect.

- Balance could be accomplished by removing rules from characteristics that have relatively many of them, but this would imply removing potentially relevant rules from the set of used rules which has a negative impact on the quality of quality readings as well.

We were, however, able to balance the model to a limited extend using the second method described in 3.5.4.2; i.e. by re-evaluating the 'overflowing' characteristics and deciding whether characteristics should be moved down the hierarchy or not. In general, we discovered the following:

- Rules assigned to the sub characteristic 'Reliability: Exception Handling' could be reassigned to the Testability characteristic. This is due to the fact that bad exception handling influences testability; if exceptions aren't handled in the correct way, errors can remain undetected and a software project is assumed to be lest testable.
- Rules assigned to the sub characteristic 'Maintainability: Understandability' could be reassigned to the Changeability characteristic. This is due to the fact that a lack of understandability inherently has a negative impact on changeability: if a developers doesn't understand source code, he/she will have a hard time changing it to fix bugs or building new features.

The result of this change is displayed in Table 26.

| Characteristic | Java | C# | (total) |
|---|---|---|---|
| **Maintainability** | 27 | 7 | 34 |
| **Changeability** | 53 | 56 | 109 |
| **Reliability** | 120 | 61 | 181 |
| **Testability** | 8 | 9 | 17 |
| (total) | 208 | 133 | 341 |

Table 25 SQALE configuration with Info Support ruleset - after balancing

The following table and graphs again show the correlations, correlation scatter plots and sensitivity for the new configuration. The table and graphs are to be interpreted like the previous ones described in 7.2.

| Correlation | Value | Expected | Conclusion/remarks |
|---|---|---|---|
| **Sonar results vs. Survey results (overall)** | **0,5** | **>= 0,30 (medium)** | **As expected** |
| - Sonar results vs. Survey results (analyzability) | 0,57 | >= 0,30 (medium) | |
| - Sonar results vs. Survey results (changeability) | 0,36 | >= 0,30 (medium) | |
| - Sonar results vs. Survey results (reliability) | 0,19 | >= 0,30 (medium) | |
| - Sonar results vs. Survey results (testability) | 0,01 | >= 0,30 (medium) | Not enough rules |
| **Sonar Results vs. Financial results (overall)** | **0,36** | **>= 0,30 (medium)** | **As expected** |
| **Survey results vs. Financial results (overall)** | **-0,03** | **>= 0,10 (low)** | **Not as expected** |

Table 26 Correlations after calibration

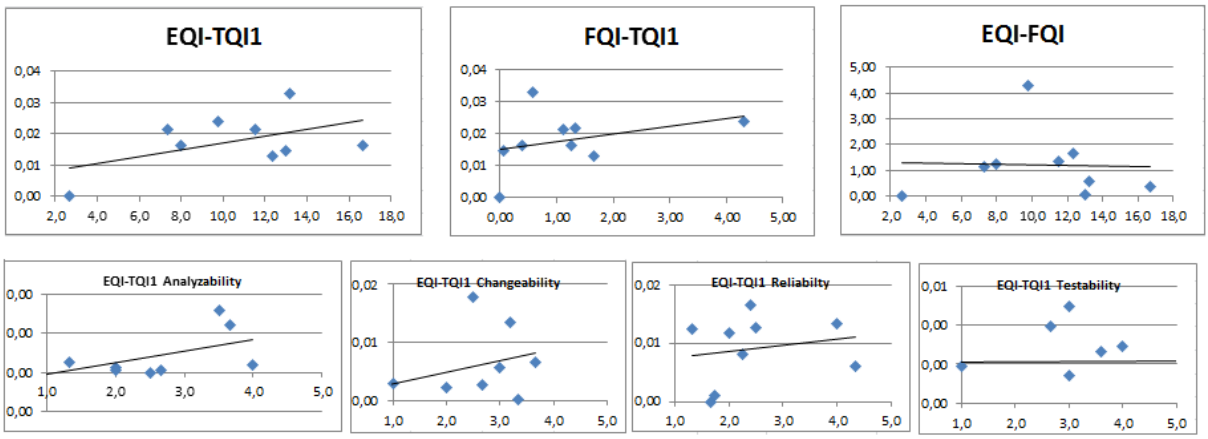



Figure 35 Correlation scatterplot after calibration

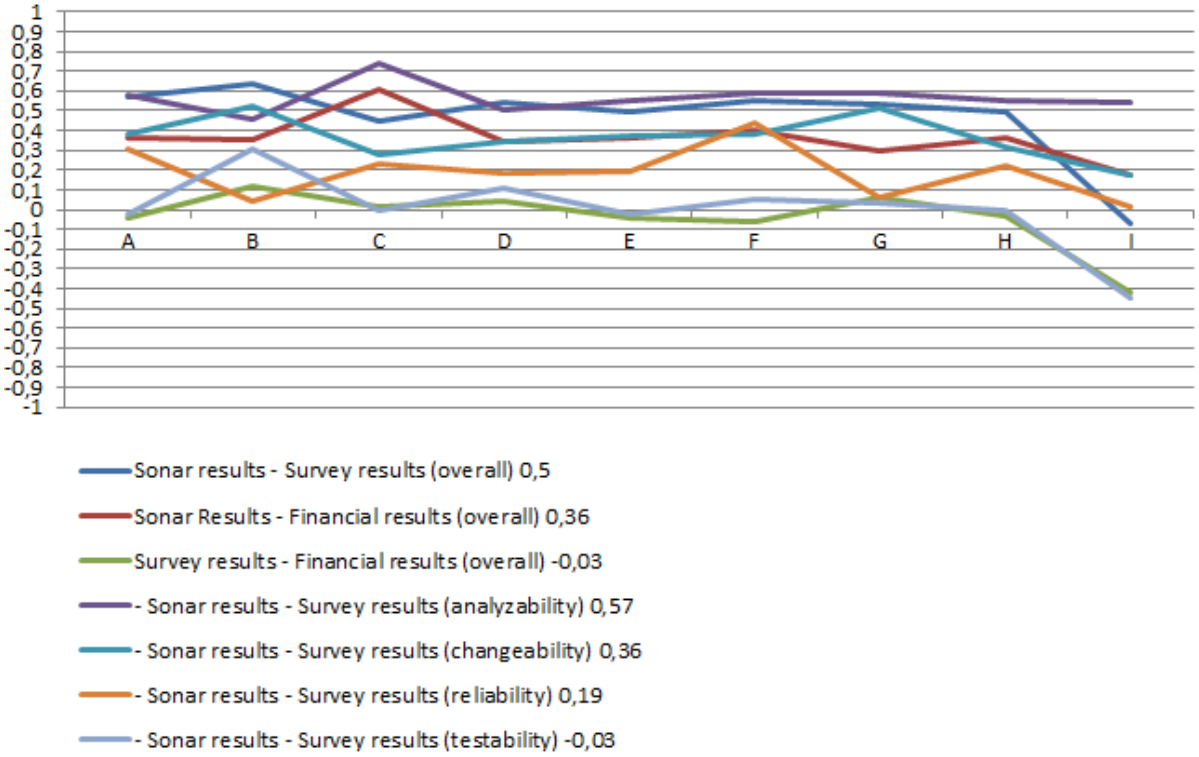



Figure 36 Correlation sensitivity after calibration

A few remarks on these figures:

- Although the correlation values are higher, the scatter plots still show a spread for individual characteristics that is very large and also much larger than the spread for the aggregated survey-tool result correlation. This implies that either the distribution of rules over characteristics is still not optimal, or that we are unable to correlate on this level of abstraction due to a difference in perception of what the characteristics mean.

- The correlation for ‘reliability’ decreased under the modifications made since the initial validation attempt. Since the associated set of rules is a subset of the original set, we may conclude that too many rules have been left out or were transferred to the ‘testability’ characteristic.
- In the sensitivity diagram, the average variance for all correlations is 0.020 (was 0.026), a decrease of 23%.
- We see that most correlations depend heavily on project I in the sample set. Removing this entry is the only action that causes significant negative correlations to show up.

## 7.4. Reflection

Now that we have performed a number of attempts to correlate Sonar measurements to validation data and to increase the value and quality of these correlations, it is time for some reflection on results so far.

### 7.4.1. What is an optimal configuration?

So far, we validated attempts to optimize the configuration by verifying that these optimizations increased the correlation with validation data. Due to the flexibility of the quality model, however, it should be possible to reach a correlation approaching +1.0. This can, for example, be accomplished by following the following steps:

- For each project, identify a rule that causes violations in only that project.
- Disable all rules but the 9 identified in the previous steps
- For each of the identified rules, set remediation costs so that the total remediation costs for the project will have a value that corresponds to expert opinion.

The resulting configuration will probably not be usable in any practical context, since it uses a rather random rule set. Also, considering the inherent subjectiveness and suboptimal reliability of the validation data, there exists a certain practical maximum correlation. This leads to the following statement: for already significant correlation values, an even higher correlation of Sonar measurement results with validation data does not necessarily imply a better configuration. In the context of this statement, ‘better’ is defined as ‘more suited to the needs of the user’, in this case Info Support. So, we may be able to achieve correlations of values approach +1.0 as long as we accept making changes to the quality model configuration purely for the purpose of increasing this correlation, but this is of no use, since it does not increase the quality of the quality model configuration.

We redefine the concept of an optimal configuration as follows:

An optimal SQALE configuration is a configuration that:

1. Uses an available rule if and only if it is deemed relevant in the context;
2. Has each of its enabled rules mapped to the characteristic that is a logical choice considering the SQALE specification;
3. Has, for each enabled rule, a remediation cost setting based on an estimate.
4. Is continuously evaluated and adapted if reasons arise to do so.

This definition contains subjective elements, since no absolute optimum exists. This implies that in the context in which the method will operate, an authority and decision-making procedure are needed to fill in the subjective elements of the definition. This could, for example, be one expert, or a set of people who attempt to reach consensus on items of discussion, i.e. as proposed in 3.5.4.1. If this authority exists, a configuration can be verified to meet the mentioned requirements. For completeness, we could formulate this as an additional requirement:

5. Is supported by an authority that makes choices were needed.

### 7.4.2. Why do correlations on characteristic-level remain low?

Although the overall correlation of survey results with Sonar measurement is good, the correlations for individual characteristics remain quite low and unstable. This is strange, because the overall readings are, both in the case of the survey and Sonar, composed of the four scores on individual characteristics.

It has been argued that the Sonar configuration is not optimally balanced, but attempts to enhance this balance result in the conclusion that this can only be accomplished 'by force', i.e. by making illogical choices in the rule – character mapping. Simply put, we are unable to optimize the balancing. But maybe 'optimizing' is not a correct term. The fact that the correlation of scores on individual characteristics are higher in the survey than in sonar results implies some kind of discrepancy in interpretation of the meaning of characteristics, as indicated in Section 4.1

This would mean that both the survey results and Sonar results are an immutable fact and the lack of high correlation values on this level of abstraction is caused by interpretation differences. This conclusion would be consistent with observations, and also implies that it is of no use to attempt to increase correlations at this level any further

### 7.4.3. On the Suitability of Sonar

During the phase in which we investigated tools and quality models to use, we had some doubts about applying Sonar in a business context (see 6.2.1). After using Sonar, in combination with a number of required plugins, we can report on our experience with it.

We did not encounter any problems that cause us to believe the setup is unsuitable for use in a business context. While practical problems can arise when using the setup, these have found not to be unresolvable. This helpful error reporting functionality of Sonar aided us in solving practical issues using the assessments. This observation confirms the presumed practical suitability of the setup.

## 7.5. SQALE Extension Proposals

In this course of this research, a number of opportunities of improvement of the SQALE model has been identified. These concern the balancing of SQALE ratings to overcome the issue of having to use one index-rating mapping for all characteristics and the lack of the concept of 'severeness'. (see 6.2.4.4) This paragraph elaborates upon these improvement opportunities and proposes model extensions for them.

### 7.5.1. Balancing the Ratings

In the Proof of Concept of this research, we identified the need for a balanced quality model configuration, in the sense that the number of rules assigned to each SQALE characteristic should roughly be comparable for all characteristics and for all languages. This is due to the fact that only one mapping of remediation costs to SQALE ratings is available for the full model. The requirement of balance can be considered a shortcoming of the SQALE model itself. We therefore propose a small extension to the model, that compensates for the lack of balance when calculating quality ratings. Using this extension, it is possible to use SQALE with a less balanced configurations, as we will show.

Consider the data in Table 27. In the second and third columns, labeled 'Java' and 'C', it shows an unbalanced configuration of four characteristics listed in the first column, labeled 'Char'. The configuration is similar to the initial configuration used in the Proof of Concept of this research. Columns 5-8 show two balancing factors, which are defined as follows:

- Interlanguage-balancing is done by calculating a scaling factor for the remediation costs, which is the total number of rules over all languages for one characteristic divided by the number of rules assigned to that characteristic for a specific language, to the power of 0.5 (square root). For example, the Java Analizability Interlanguage balancing factor is √(456/205)=1.49
- Intercharacteristic-balancing is done by calculating a scaling factor for the remediation costs, which is the total number of rules for a specific language divided by the number of rules of that language belonging to a specific characteristic. For example, the Java Analyzability balancing factor is √(586/205)=1.69
- The overall factor is calculated by multiplying the two established factors. For example, in the case of Java Analyzability, the factor is 1.49 * 1.69 = 2.52. For each language, a factor is calculated for each characteristics. These factors are used as a multiplier for the remediation costs per language per characteristic.

| | **No. Of Rules** | | | **Interlang** | | **Interchar** | | **Result** | |
|---|---|---|---|---|---|---|---|---|---|
| Char. | Java | C# | **Totals** | **Java** | C# | **Java** | C# | **Java** | C# |
| Analysability | 205 | 251 | **456** | 1,49 | 1,35 | 1,69 | 1,33 | 2,52 | 1,79 |
| Changeability | 36 | 13 | **49** | 1,17 | 1,94 | 4,03 | 5,84 | 4,71 | 11,33 |
| Reliability | 332 | 170 | **502** | 1,23 | 1,72 | 1,33 | 1,61 | 1,63 | 2,77 |
| Testability | 13 | 9 | **22** | 1,30 | 1,56 | 6,71 | 7,02 | 8,73 | 10,97 |
| **Totals** | **586** | **443** | | | | | | | |

Table 27 SQALE Extension Proposal Example

This way, we create an 'inverse weighing factor' for each characteristic-language combination. The reason to include the square root component in the formula, is the following assumption:

- The average relevance of rules assigned to a characteristic decreases when more rules are added to that characteristic.

The rationale for this assumption is that if one bases a quality judgment on just a few rules, these will presumably be well-selected, while for a large set of rules it is not much of a problem if some rules are not very relevant. This implies the need to introduce an exponent in the 0-1 range. Since we have no further information to narrow this down, we select the power of ½.

| Tool Results (scale 0,05-0,10-0,15-0,20) Model Extension ENABLED | | | | | | Tool Results (scale 0,01-0,02-0,03-0,04) Model Extension DISABLED | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Analyze | Change | Reli | Test | Total | | Analyze | Change | Reli | Test | Total |
| 4 | 2 | 3 | 5 | 4 | | 5 | 1 | 5 | 4 | 5 |
| 3 | 1 | 5 | 1 | 5 | | 5 | 1 | 5 | 1 | 5 |
| 5 | 5 | 4 | 5 | 5 | | 5 | 3 | 5 | 5 | 5 |
| 2 | 1 | 2 | 3 | 2 | | 5 | 1 | 4 | 2 | 5 |
| 3 | 3 | 3 | 5 | 4 | | 5 | 1 | 4 | 4 | 5 |
| 3 | 1 | 2 | 2 | 3 | | 5 | 1 | 3 | 1 | 5 |
| 4 | 5 | 1 | 4 | 4 | | 5 | 5 | 3 | 2 | 5 |
| 4 | 2 | 2 | 5 | 3 | | 5 | 1 | 3 | 3 | 5 |
| 2 | 5 | 1 | 1 | 2 | | 3 | 5 | 2 | 1 | 5 |

Table 28 Comparison of Quality Assessment with and without SQALE model extension

The effect of applying this extension, is that it becomes far more likely that one mapping of remediation costs to SQALE ratings is sufficient to establish readable results for all characteristics as well as the overall quality. The result is demonstrated in Table 28. These results are based upon the actual initial SQALE configuration used during the Proof of Concept. Each row represents a project and each cell contains a SQALE rating for that project, on a specific characteristic. For both the 'extension enabled' and 'extension disabled' situation, the mapping has been optimized to get a fair spread of ratings. Of course, since the extension causes remediation costs to be multiplied by a specific factor per characteristic and per language, the boundaries between A-E ratings on the scale have higher values with the model extension enabled.

As can be seen, the distribution of values in the 1-5 range is bad (variance < 0.5) in two of the five columns of the right overview, which is not the case in the left overview (lowest variance = 1.0 for analyzability). The average variance per column is 2.06 in the left section and 1.41 in the right section, which is an increase of 46%, realized by enabling the model extension.

Note that in this example, no scaling is applied to the 'total' score. This causes the overall ratings to no longer be at least as high is the highest characteristic rating.

Unfortunately, this model extension cannot easily be integrated in the solution for Info Support, since it requires a modification of the tool used, of which the SQALE component is not open source. We therefore consider this to be a small contribution to the field of research instead of a component of the method to be developed.

### 7.5.2. Adding Weights to Rule Violations

Due to the use of the remediation cost paradigm, SQALE allows the user to set remediation costs per violation or per file in which one or more violations occurs. The number of violations or the remediation costs, however, do not say anything about the severity of the violations, that is the potential importance of repairing violations. For example, it may be the case that some rule that has a large number of violations and very high remediation costs, is actually not really important. This could, for example, be the case for rules that concern what could be considered cosmetic source code details such as variable name spelling policies, the use of curly braces for single-line if statements or the amount of whitespace between methods. This paragraph discusses a number of approaches to the lack of violation severity.

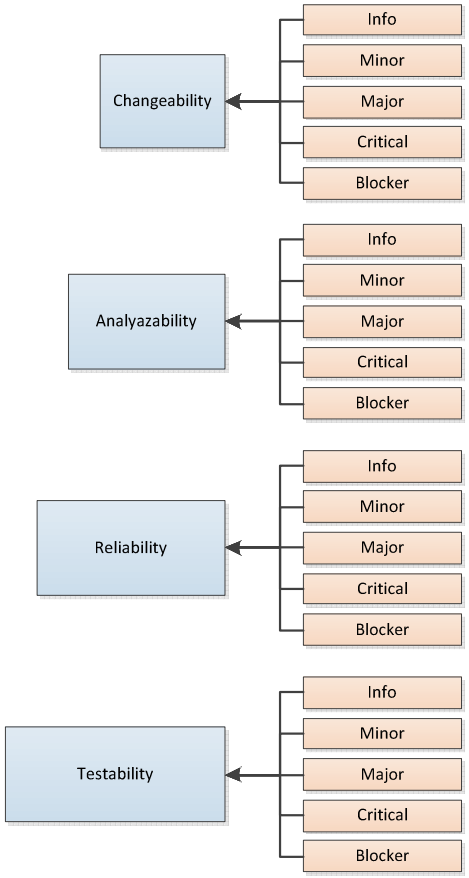



Figure 37 Rule severity approach example

#### *7.5.2.1 Approaches within the current model*

Approach 1: explicitly define severity classes as sub characteristics. Since the model allows the user to freely define sub characteristics, it is possible to define these per severity. For example, the 'Reliability' characteristic could have sub characteristics for severity classes like 'Info', 'Minor', 'Major', 'Critical and 'Blocker. The same sub characteristics could be defined for the other characteristics as well. Each rule would be associated to one of the severity sub characteristics of an appropriate characteristic. Figure 37 shows the first and second level of the SQALE hierarchy when using this approach and the characteristics relevant for this research.

Advantages:

- Can be implemented within the boundaries of the existing model definition;
- Supported by the Sonar tool SQALE plugin;
- Relatively easy to implement, although it will cost some time to reconfigure the model;
- Provides easy insight in severity of violations through the SQALE sunburst diagram (see App. D).

Disadvantages:

- Since the number of levels of hierarchy in SQALE is fixed, one of the hierarchy levels, namely the sub characteristics level, can no longer be used for its original purpose (an abstraction level in between characteristics and rules);
- Although the SQALE definition allows this approach to be implemented, it could be considered as a form of abuse of the intended abstraction hierarchy;
- This approach does not provide a way to take violation severity into account when calculating quality indices and ratings.

Approach 2: use remediation costs as weights. This is a very obvious possibility which has the same advantages as approach 1. It is, however, an explicit violation of the intentions of the SQALE model, since is does not in any way conform to the remediation cost paradigm. When using this approach, SQALE indices would indicate the 'sum of severities of rule violations per characteristic'. This could be considered an indication of quality, but does not say anything about the amount of repair effort required. SQALE ratings would become rather useless, since the total development effort is no longer a relevant measurement. This approach highly disadvised.

#### 7.5.2.2 Model Extension for Rule Severity

Approach 3: use an extension of the model. Although not directly implementable in a practical context using existing tools, we propose an extension to the SQALE model to integrate severity properties of rules. The extension has the following properties:

- The basis is the existing SQALE model as defined in (Sqale 2011).
- Each rule gets an additional property, named 'severity'. This property can have one of the following values: info, minor, major, critical, blocker. It is mandatory to set a value for each rule in the model configuration.
- When executing the model, a new, additional result is provided to the user. This result shows the user not only how much time is needed to repair source code shortcomings, but also shows the sequence in which issues should be addressed. This sequence is calculated based on a priority scheme. In this priority schema, issues are sorted by severity; issues with a higher severity come first. Within one severity group, issues are sorted by remediation costs; items with the lowest remediation costs come first. The result can be displayed in the form of a list. It is also possible to use other priority schemes, such as 'Shortest Job First'. Possible schemes are comparable to those used in process scheduling in operating systems (Silberschatz 2009).
- Next to the SQALE kiviat that shows the SQALE rating per characteristic, a second kiviat is defined that shows the severeness of issues per characteristic. For each severeness category, a weight is set. The default settings are displayed in Table 29. We choose a logarithmic scale, since we assume that a violation in one category is usually considered 'x times as severe' as a violation in a lower category. The weights, however, should be changeable by the user, because users may have a different interpretation of the relative severity of different categories.

| Name | Weight |
|---|---|
| Info | 1 |
| Minor | 5 |
| Major | 25 |
| Critical | 125 |
| Blocker | 625 |

Table 29 Default severity weights for SQALE extension

- The 'Extended SQALE severeness index' for a characteristic is the weighted sum violations associated with that characteristic. The results can be displayed in a Kiviat, along with the original remediation cost kiviat. Consider the example displayed in Figure 38. The original SQALE kiviat (left) and issue severity kiviat (right) should be interpreted together. For example, 'Changeability' (top) has high remediation cost but a low severeness, while Analyzability (left, indicated 'maintainability' in the screenshot) has lower remediation cost

but a higher severity. This information combined, it seems logical to address the analyzability issues before the changeability issues. This insight could not have been obtained using the original SQALE model.

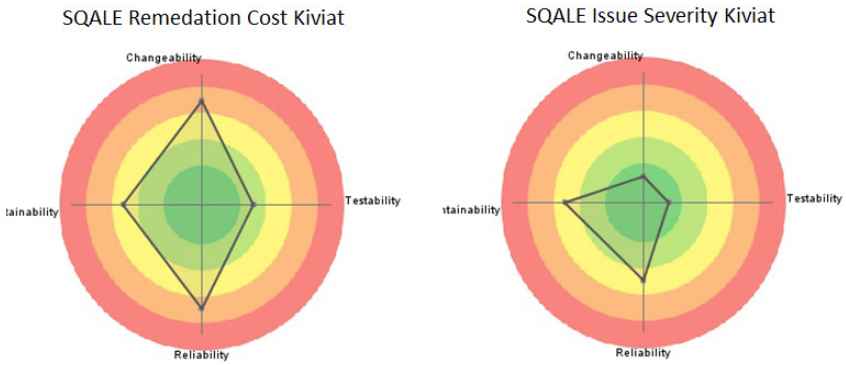



Figure 38 SQALE severity kiviat extension example

The diagrams in Figure 38 could be merged into one diagram, by applying a formula that divides the values for severeness and remediation cost. This indicates the 'amount of reduced problem severeness per unit of time' when working on the artifact. This is no longer a direct indication of quality, but an work scheduling tool. If this is implemented in a fashion that allows the user to 'zoom in', the direct rule violations to start working on can be identified.

Advantages:

- This approach provides useful information not available in the original SQALE model;
- This approach can be used in the process of deciding which issues to address first.
- The extension addresses an import shortcoming of the current model while respecting the current model.

Disadvantages:

- Increases the conceptual difficulty of the model and the time needed for configuration;
- Cannot be used in a practical context as long as no implementation is available.

# 8. Recommendations for Info Support

During this research, a number of ideas about things that, in the opinion of the author of this work, should be done or should not be done have come up. Based upon literature study, the acquisition of knowledge about Info Support as a company and the results of this research, this chapter sums up these recommendations.

## 8.1. Do not try this at home

In the initial research phase of this project, we have looked at various tools (section 2.6, page 35). The amount of effort put into the development of these tools is significant. Also, successful business models assure continuous improvement and friendly prices. Although Info Support has been developing some forms of source code assessment in-house, it is advised not to attempt to develop an equivalent to tools such as Sonar as an internal project. Such project would require a large initial investment as well as a structural investment of resources, while third-party tools such as the ones used in this project can do the same job for a price that is expected to potentially be orders of magnitude smaller. There is more certainty that for a fixed price per time period, the used tools always stays up to date. Also, flexibility of the tools allows for some dynamic configuration, so that specific Info Support requirements can usually still be incorporated when using third-party tools.

## 8.2. Keep an eye open for newer and better tools

In the tool identification phase of this research, it was found that many tools are still under development and are continuously improved. Also, Sonar with .Net support could be called a somewhat improvised solution, due to the non-native support for .Net (6.2.1, page 67). Promising projects, such as Squoring (Squoring 2011) are currently running but do not yet provide solutions that can be used in a business context. It appears that quality-model based source code assessment is still in its childhood. Therefore, it is worthwhile to keep an open eye on the tool market, to seek out opportunities to use the newest tools after they are released.

One might think this recommendation contradicts the first, because why would Info Support not need to develop a tool itself while apparently no optimally-suited tools are available? The answer is that it is expected that 'more suitable' tools become available much faster than Info Support can develop something suitable internally. So, even though Info Support may not be fully satisfied with the tools currently available, it is not possible to decrease the waiting for something better by starting internal development.

## 8.3. Run tools on appropriate hardware

The Proof of Concept phase of this research (section 6.2, page 67) indicated the need for appropriate hardware to run SONAR, since it is a heavy tool. When using a physical machine, the following recommendations apply. When using a virtual machine, an approximation of the physical machine recommendations applies.

1. CPU: this is the primary bottleneck. Faster is better and there is no hard minimum or maximum speed; but due to the increasing price per unit of speed, an affordable CPU in the high-end segment would be recommended.
2. RAM: at least 4 GBytes are needed, at least 6 GBytes is recommended.

3. Disk I/O: analysis involves many read/write operations; it is recommended to use a striped RAID array of fast SATA300 disks (do not use Solid State disks due to expected quick wear caused by many small I/O operations). Also, consider using drives with a large cache.
4. Storage capacity requirements are limited. As a rule-of-thumb, take 30GBytes for the Operating and software, plus one additional GByte for each project to be analyzed.

## 8.4. Integrate analysis in PDC Nightly Builds

It is recommended that the analysis method becomes part of the 'nightly build' process at the Professional Development Center (see paragraph 2.1.1 on page 19) of Info Support. This ensures that developers have access to actual analysis results and can aid them in developing 'maintainable' applications. A safe way to technically integrate the method in the process is by performing the following steps. The analysis server is a separate (physical or virtual) server.

1. Initiate analysis be calling an executable script from the existing build procedure
2. Using this script, copy the source tree (including dependencies, libraries etcetera) to the analysis server
3. For .Net projects:
    a. Merge the solutions the project consists of using merge-solutions (see 6.2.3)
    b. Install a pom.xml project object model with appropriate parameters (can be done using a script)
4. Compile the project, write messages to a location that is accessible by developers
5. Run the analysis, write messages to a location that is accessible by developers

Note that is it is highly recommended to work on a copy of the source code tree, since the Sonar analysis procedure may modify this tree, especially in the case of .Net projects that are to be merged into one solution (see 6.2.3). It may be possible to run the analysis on a source tree without modifying it, but this would require careful configuration.

If the analysis succeeded, the results will be available through the Sonar interface as depicted in appendix D. If the analysis fails, the error logs will indicate which problems to solve. Integration of the display of results in the 'regular' nightly build interface is possible in several ways. The most simple way is by linking to the Sonar interface. Another option is to include and auto-load a custom Sonar dashboard in a frame-like construction.

The described approach is safe since it minimizes the interference with existing procedures, on a source code as well as performance level, which is a recommended approach in the initial phase of using the method in production mode. Drawback of this approach is that it may not be the most efficient procedure in terms of computer resource requirements.

## 8.5. Sell Quality Assessment as a service

Using the results of this research, Info Support could sell software quality assessment as a service to customers. The service would be comparable to that of the Software Improvement Group (SIG), in which a project's source code is analyzed by SIG, resulting in a report (see 2.4.4). The manual work that needs to be done is to write a report around an automated assessment, providing useful information on how to interpret results and to provide recommendations for increasing quality and/or reducing business risks. For these reports, templates can be used to reduce the amount of manual work. The actual quality indices in the reports would be very comparable to SIG audits, since

a comparable quality model is used. Recently, Info Support started an 'Audit Services' projects, in which this suggestion fits quite well. Also, considering the relatively low amount of work needed to perform an automated audit and the 'professionalism' of SIG(-like) audit reports, this opportunity has a low risk and high potential benefits.

To create audit reports that match SIG results as closely as possible, it is possible to 'reverse-engineer' one or more SIG audit reports into a configuration of the SQALE quality model. The configuration of the SIG plugin for Sonar as described in 2.4.4 could be used as a basis. A one-on-one mapping between SIG quality aspects and SQALE characteristics should be defined. while extensions could be made based upon the audit reports. Validation can be done by performing assessments of the projects of which a SIG-audit is available. Note that while it is deemed possible to create the mentioned one-on-one mapping, the paradigms used by the models defer greatly, which may make it impossible to fully imitate SIG. This is primarily caused by the non-existence of a benchmarking repository. The introduction of such a database is a suggestion for future research (see 10.1)

## 8.6. Assign method responsibility and authority

Working with the method at an operational level, and actually understanding it, requires knowledge. The results of applying the method (quality indicator values) are very easy to understand, but the method itself is more complex (see 2.4.6). It is, therefore, recommended to pay attention to the process of gaining this knowledge and storing and transferring it if necessary. One person is recommended to be made responsible for the technical aspect of the method. This person should know the setup of the method and how to use it, and also be able to make modifications if necessary. Knowledge of the method should be shared by a small number of users, i.e. 2 or 3, including the person responsible. This way, there will always be someone who knows how to operate the method and no single point of failure.

Preferably, the persons with knowledge of the method are also familiar with Info Support coding rules for both Java and C#, so that they can enhance the model configuration if needed. This is consistent with the requirement for authority as mentioned in 7.4.10. It is also recommended that the person(s) responsible become a member of the Sonar user mailing list (user@sonar.codehaus.org), which in this project has proven to be a useful resource in case of problems or support questions.

## 8.7. Improve incident registration procedure

Details of this recommendation are left out of the unrestricted version of this thesis.

# 9. Discussion

The results of this project arose a number of items that allow for further discussion. Some of them are elaborated upon in this chapter, namely:

- The fact that the SQALE method is not capable of covering every aspect of software quality assurance;
- The tool and method survey as well as the configuration methods and conclusions are generalizable, while the actual configuration and correlations can be considered context-specific due to the flexibility of the model;
- The remediation cost paradigm is suitable for the intended purpose, but does have shortcomings;
- What the term 'quality' actually means stays context-dependent, stakeholder-dependent, and an ongoing topic of inconclusive debate.

## 9.1. What the Method Does Not Do

The validated software quality assessment method described in this thesis can be used to measure a number of aspects of quality (see 2.4). There are, however, things that are explicitly beyond the scope of the capabilities of the method. For these things, the method cannot and should not be used. This section identifies a number of quality assessment aspects that are not covered by the method, and indicates what could be done to assure that these aspects receive sufficient attention.

### 9.1.1. Functionality Verification

An automated process can only act upon the input it receives. Since the 'intended functionality description' of a software system is no input to the tools discussed in this thesis, the automated quality assessment process cannot verify that software 'does what is should do'. As an extreme example, consider a software system that is supposed to allow a local government to manage a road network maintenance schedule, but has an implementation that is only capable of displaying the text "Hello, world!" to the user. Although it is clear that this piece of software is unsuitable for its intended purpose and does not meet requirements, its quality as measured by software tools may still be very high if the implementation conforms to the requirements as configured in the quality model. This means that in no way can an automated quality assessment process that has only source code and a quality model configuration as input be used to substitute any part of the requirements engineering and validation process; a 'high' source code quality is not an indication whatsoever that a system meets any of its functional requirements. The method, therefore, should be used *in addition* to existing components of the software development and/or maintenance process. See, for example (Lauesen 2002) for information on pre-implementation requirements engineering and functionality verification.

### 9.1.2. Test Quality

While there are rules for both unit test coverage (by line and branch) and unit test results, there is no method of assessing the quality of the tests themselves. For example, if tests are made that call all methods of all classes, but always succeed, Sonar will be unable to notice this. More general, if a developer doesn't 'like' a rule, he/she might be able to fake conformation in the case of some rules. Unit test coverage is one of these.

### 9.1.3. Non-source code components

Projects may contain components that do not consist of Java or C# code. For example, this could be XML schemas and files that are used by the software as a resource. It should be noted that Sonar does not take these files into account, since all available rules concern pure Java or C# code. This is a limitation of the method that becomes more relevant for projects which large amounts of non-regular sources. Additional quality assurance methods may be used to cover these. This makes the method less suitable for, for example, web applications for which specific programming and scripting languages are used to program user interaction, such as html or jsp.

### 9.1.4. Process metrics

The process of software management not only depends on the quality of the software, but also on the quality of the software management or maintenance process. More company-specific that product-specific, this process determines how and how well software management tasks are carried out. Software may be very maintainable, but of not good maintenance procedure exist, the result may still be that inadequate management services are delivered. Examples of aspect of process quality are programmer skills, quality of customer communications, speed of issue resolving and support availability. Formalized quality models for the maintenance process exist or are being developed, i.e. (April 2005), (Kitchenham 1999).

## 9.2. On Correlation-limiting factors

The correlation of SQALE measurements with expert opinions have values that we mostly quantified as 'medium' (3.5.1 and 7.3). We argued that we could further increase correlations by allowing the configuration to be tweaked for the purpose of high correlations only, but that this is not something that we should want to do (7.4). Furthermore, there are possible reasons for these correlations to have a limited value. Verifying these reasons is impossible or beyond the scope of this research, but we mention some of them for the reader to consider.

1. Although it was stated to the expert that the quality assessment concerned source code only, expert may implicitly and/or unconsciously include other aspects of source code in their judgment. This may concern things like documentation, the relationship with the customer, the history of a project, etcetera. These things may influence the opinions of expert, limiting the expected correlation with objective SQALE results
2. Expert opinions are inherently subjective. An enthusiastic expert may, for example, give higher score for a project than a less enthusiastic expert. Also, things like mood, the amount of currently open issues in a project, the weather, the time of the day, the day of the week, the length of unexpected traffic jams and the amount of consumed coffee may play a role. This may sound somewhat far-fetched, but we cannot falsify the hypotheses that the mentioned examples decrease the potential correlation with SQALE measurements.
3. SQALE measurements are taken from a specific version of software. Although we have in the survey referred to specific version numbers, an expert opinions may implicitly have a stronger historic perspective. For example, if two project have an equal objective quality now, but one has always have a high quality and the other has been enhanced after years of issues, experts may consider the first one to be better.

4. In section 4.3 we identified a correlation of ratings for individual quality characteristics by experts, which we did not see in the SQALE measurements (section 7.2). It may be the case that expert are unable to conceptually separate the characteristics, i.e. they don't know the boundaries between reliability, testability etc, while SQALE has very strictly separated rulesets attached to each characteristic.

More reasons could be imagined, all in the form of 'maybe'-like statements that we cannot simply falsify. The point that we make with this issue is that we should be satisfied with correlations with 'medium' values. The same is true for other research projects that use expert opinions in a comparable way.

## 9.3. Generalizability of Research Results

This research was conducted at a specific company, that supplied the sample data used in the validation phase. This paragraph discusses the generalizability of the results of this research, i.e. the likeliness of results being applicable in a broader context. This elaboration will be provided per research aspect.

### 9.3.1. Background information and Tool selection

The background information presented in this thesis, mainly on quality models, is not tied to a company context. It is inherently general and therefore suitable for use in any context. The survey of tools is general in the sense that it objectively describes a number of tools. The selection of tools, however, is based upon requirements that were mainly formulated by Info Support. In a different context, the selection phase may have yielded a different results. The comparison tables could provide the information necessary to make that choice. For example, in a context is which a SaaS solution is not considered a problem, one might prefer Kalistick over Sonar since it natively supports both Java and C#, while Sonar requires a set of plugins to get C# support to work.

It is relevant to mention that it is not necessarily important which tool is used, as long as it uses an appropriate quality model. The tool is merely something that provides us a method to apply a quality model in a practical context. Although a tool selection needs to be made in order to perform the proof of concept, a different tool which uses the same quality model should yield the same research results, as long as there are no major implementation differences.

### 9.3.2. Research Design

The research design section basically describes a sequence of steps, namely 1) retrieve expert opinions on quality, 2) retrieve financial indicators, 3) perform the proof of concept and 4) draw conclusions.

In theory, the research structure could be repeated in any context, the main precondition being the availability of sample project source code, experts and financial data. A problem with in the execution of the research design described in this thesis is the lack of quality of the financial information. When reconducting this research in a different context, the following things could be done to increase the usefulness of this validation data retrieval aspect:

- Only select external projects. This gives a better guarantee for the quality of financial information, since this information is used to bill clients. In this research, it was not possible to apply this constraint due to the low number of actively maintained external projects (n=6).
- Reduce customer diversity. The way customers handle software may influence the number of reported incidents. For example, some customers may provide detailed descriptions of issues with their managed software, while others just call to shout it is broken. Also, some customers may ignore small issues and find workarounds, while others want everything to be fixed. No method to take these things into account is defined in this research, as no possibilities were seen to do so. To reduce the need to address this issue, use a number of project from one customer or a set of similar customers (i.e. governmental organizations).

A suggestion to incorporate these recommendations is to select a set of open source projects of which source code is freely available. Financial quality information can be obtained from the incident registration (ticketing) system. For example, the average time required to close a ticket can be considered a quality indicator (Luijten 2010).

It should be noted that the financial quality indicator defined in the research design could not be validated. Due to the low quality of the data obtained in this research aspect, we cannot easily draw conclusions about the validity of the indicator in this research. This should be taken into account when performing the research in a different context; if a correlation cannot be found, it may be an option to make appropriate modifications to the indicator.

### 9.3.3. Validation Results

In the validation phase of this research, we validated the results of Sonar quality measurements by calculating the correlation with expert opinions and financial indicators. Can we say something about the generalizability of the validation results?

Important in answering this question is the fact that the SQALE model is in fact very flexible; it defines a set of characteristics and sub characteristics and a method of calculating indices and ratings. It allows full freedom when it comes down to defining metrics, configuring remediation costs and assigning metrics to sub characteristics. This flexibility allows the method to be configured to match context-specific coding policies. A validation of measurements, therefore, does not mean that the method will always yield 'correct' results. A better way of putting it would be that the method *can* yield correct results *when properly configured to suit contextual needs*. One might wonder if this is true for any context. Since 'any' is a broad term, this can never be proven. The enormous set of available rules, however, implies that the possibilities to configure SQALE to match contextual needs are quite large. Based on this, it is reasonable to assume that for *most* organizations that develop or manage software using a well-defined set of coding rules and software quality standards, the method will provide adequate quality information.

## 9.4. The Remediation Cost Paradigm

The SQALE quality model is based on the Remediation Cost paradigm (see 2.4.6 on page 5). Basically, for each quality characteristic, the amount of time needed to repair all issues, i.e. to reach a perfect score on this characteristic, is calculated and compared to the total development cost of the project.

The SQALE score is calculated by mapping the divisions of these values to a scale with five discrete levels (see Figure 13 on page 31 for an example). A number of thoughts about this method is addressed here.

- This paradigm does not detect the lack of implementation that 'should be there'. For example, if a software project does not contain any implementation that contributes to the testability of the application, it's 'Testability' remediation costs would be 0 and therefore, by definition, an 'A' rating will be given. This problem can, at least for a part, be addressed by the use of rules that assess the existence of certain types of code (i.e. unit test code coverage). Conforming to what has already been mentioned in 9.1.1, however, the method can do some reflection but cannot look 'beyond the code'. This means that code of which the incorrect nonexistence cannot be detected by the analysis of existing code cannot be taken into account in the quality judgment.
- In the paradigm, each rule is configured with a fixed amount of time to fix one violation. This means that to optimally configure the model, a method to properly estimate these remediation costs needs to be defined. This is very hard, especially when one takes into account the fact that the value is supposed to include post-implementation testing; Also, the fixed value relies on the assumption that there exists a linear relationship between the amount of violations per rule and the time needed to repair all these violations. In some cases, this is not necessarily true. For example, consider the situation in which a variable name does not conform to specified parameters. If the variable is used 25 times in the project, the remediation cost will be 25 times the remediation cost per violation. In reality, however, this issue will be fixed using a refactoring function. The amount of effort needed by the programmer is therefore close to constant and independent of the number of violations. The SQALE implementation in Sonar incorporates a feature that partially fixes this by allowing the user to configure constant remediation costs per file containing one or more violations instead of per violation. The SQALE standard itself, however, allows for more complex functions to be defined. A possible enhancement of the Sonar SQALE plugin would be to implement the possibility to define more complex functions.
- The paradigm does not incorporate the concept of issue severeness. Remediation costs do not necessarily tell the user anything on how bad a rule violation actually is; it only tells the user how much time it is going to cost to fix it. This issue is addressed in a proposal for an extension of the SQALE model (see section 7.5.2.2 on page 86), but this extension is, of course, not incorporated in contemporarily available tools. However, Sonar does allow the user to define the severeness of issues and makes this information available in analysis results. This information, however, is not integrated with the SQALE results, and provided to the user separately.

## 9.5. On the Concept of Quality

We have not yet elaborated upon the concept of quality, and the extent to which remediation costs are or are not a suitable paradigm to look at it.

In the introduction section (chapter 1), we already stated that quality has context-specific definitions. An important aspect of the context is the goal. In general, quantitative quality assessment provides a score, on a scale, indicating the level of quality. But what does a higher, or lower, quality mean? An

often-seen term in this context is ‘fitness for use’ (i.e. Tayi 1998), indicating that a higher quality means that the artifact under considerations is more suitable to use for its intended purpose. When talking about source code assessment for maintainability purposes, rating several aspects of maintainability seems consistent with this ‘fitness for use’ concept.

Also, the remediation cost paradigm expressed (lack of) quality in terms of time, or money, required to repair all issues in an artifact. So, quality/maintainability is increased by investing time or money in repairing defects. Examples of these defects are methods that are too long, classes that have insufficient unit test coverage and unsafe exception handling. Indeed, these repairs have a direct positive impact on the maintainability of software and therefore increase the fitness for use.

But in this case we only look at the situation from the perspective of the people that perform software management task. Is the used definition of quality also relevant for other stakeholders? An important stakeholder is the customer, the party actually using the software. A higher maintainability has advantages, like faster issue resolving and lower maintenance costs. It is reasonable to assume that a customer prefers to have more service for less money, so, indirectly, the increase of maintainability also has a positive impact on the quality of service as experienced by the customer. Is this always the case for all stakeholders? Not necessarily. As a counterexample, consider the individual employee that has managed software issue resolvement as his task, together with 20 other employees. Now suppose that the quality of all managed software is increased to its maximum level. Then, due to the fact that the software is better maintainable, the amount of maintenance work is reduced and, unless new customers or projects are attracted, the number of employees may be reduced as well.

So, the definition of quality is not only context-specific, but also stakeholder-specific. Also, this project has shown that ‘context’ does not just mean ‘software development or maintenance’, but is even specific for the company at which this takes place. This supports the claim by (Reeves 1994) that no globally suitable definition of quality can exist.

To put the concept of quality in a broader perspective, we will list a number of views of ‘software quality guru’s, summed up by the Belinge Institute of Technology in (Milicic 2011):

- Armand V. Feugenbaum (1992, American quality control expert and businessman) states that ‘Quality is based upon the customer’s actual experience with the product or service, measured against his or her requirements’;
- Kauro Ishikawa (1915-1986, Japanese professor and quality management innovator) said that quality should be defined according to standards containing shortcomings, and that quality does not reflect constantly changing customer demands;
- Joseph M. Juran (1904-2008, management consultant, author of books on quality and quality management) stated that quality can be those product features which meet the need of customers and thereby provice product satisfaction, as well as freedom from deficiencies;
- Walter A. Shewhart (1891-1967, American physicist, engineer and statistician) stated that quality can either be an objective reality independent of the existence of the customer, or the subjective perspective dependent on individual thoughts, feelings or senses as s result of the objective reality.

We may conclude that due to the many aspects of quality, any attempt to formalize it will capture only a subset of these aspects. And then, we may wonder what is the quality of this subset, and how do we determine that. Over the course of that last century, quality-related publications have not led to a converged, established notion of the concept. Therefore, what quality actually is may remain a topic of interesting, though inconclusive discussion for a long time to come.

# 10. Future Research

In addition to the SQALE extension proposals from chapter 7.5, a number of other suggestions for future research has been formulated.

## 10.1. Benchmark-based calibration

In this project, calibration of the quality model was performed based by applying several methods that in general require the person performing the calibration to define what is good and bad. Since this is an inherently non-trivial and subjective procedure (see paragraph 7.4.1), it would be interesting to attempt to calibrate a SQALE configuration using a benchmarking repository, comparable to the method used by SIG to calibrate their model (see 2.4.4). For example, analyzing a large set of projects with SQALE may provide the following information:

- An empirically established mapping between SQALE indices and ratings. The SQALE method allows the user to customize this mapping, but a large test set would allow us to establish a mapping that would judge assessed projects on their relative quality to test set projects.
- An indication of rules that are often violated or often have high remediation cost. This may indicate that these rules are not very important, that estimated remediation costs are too high or fixing the issue is not considered worthwhile, all of which may be reasons to omit the rule from the test set or to modify parameters.
- Information on rules that are hardly ever violated. These could be left out of the quality model configuration to increase performance.
- If the repository contains multiple versions of the same project, it can be seen which issues remain and which gets fixed. Issues that remain during different versions of a software system may indicate that fixing these was not considered worthwhile, which may indicate that a rule is not very important.

This information will allow us to recommend certain settings of SQALE. The model is, by definition, very flexible. This can be considered an advantage, but also means that the use cannot just start using the model but needs to configure it first. Configuration is hard due to the lack of best practices. This benchmarking method could provide such practices.

## 10.2. Lines of Code versus Function Points

As indicated in background research design section 3.3.1.2, evidence exists for an approximately linear relationship between lines of source code in a project and the number of function point of that some project (Caldiera 1998)(Dolado 1997). Although the nature of the relationship is described in literature, we were unable to find empirical evidence for the actual mathematical characteristics of this relation. This information could be used to fill in variables in the function that describes the relationship, making it more usable in a practical context.

From the literature, it follows that the relationship has the following structure:

S = a * F + C
Where:
- S = the size of the project (the number of Lines of Code)
- a = the scalar (additional LoC's per FP)
- F = the number of Function Points in a system
- C = a constant to account for overhead

Empirical research would allow us to validate the linear nature of the relationship, determine a value for variable a and C and determine under which conditions this formula applies. This would fill up a ‘gap’ in the literature that currently appears to exist.

# 11. Conclusion

## 11.1. Answers to the main Research Questions

In the introduction chapter, we stated the following primary research question:

***How do Sonar SQALE quality assessment results of projects correlate to Info Support experiences and expectations?***

The research provided us an answer to this question: we learned that Sonar, combined with the SQALE quality model, provides a working method to assess the quality of both Java and .Net projects. We saw that the correlation of the values of these measurement with validation data is significant, and can be increased by performing quality configuration calibration methods. This means that we have found and validated a method to perform software source code quality assessment, which was the main goal of this project.

## 11.2. Proof of Concept Setup

We demonstrated the capability of the identified Sonar tool and SQALE quality model to assess both Java and .Net project by setting up the tooling on a virtual machine and performing assessments of sample projects. The procedure to setup the tooling has been described so that it can be repeated. No significant irresolvable problems were encountered when performing this setup.

### 11.2.1. Initial Correlations

Initially, Sonar was configured to use as many rules as possible, using default settings and a rule – characteristic mapping obtained from the developers of Sonar .NET.

The results of the initial measurement answer research sub question “How are the sample projects rated by a Sonar SQALE setup?” These results are displayed in Table 20 on page 73.

We correlated these results with the validation data with the following results:

- Sonar results vs. survey results: +0.41
- Sonar results vs. financial quality: +0.34

### 11.2.2. Optimizing the Configuration

In the research design, we identified a number of possibilities to optimize the configuration. An optimization, in this context, is defined as a change in the quality model configuration that increases the correlation of Sonar measurements with validation data. Ideas for optimization methods are described in the research design section 3.5.4. A number of these methods has been applied to the configuration of the Proof of Concept. By doing this, we answer the research sub question *Which methods to improve the quality of the quality model configuration exist?*

After optimizing the configuration using Info Support coding rules, the correlation with validation data was as follows:

- Sonar results vs. survey results: +0.50
- Sonar results vs. financial quality: +0.36

## 11.3. Obtaining validation data

### 11.3.1. Expert Opinions

We obtained expert opinions by asking experts to rate four quality aspects (analyzability, changeability, reliability and testability) of the sample projects on a nine point scale. We also asked experts to rate their own confidence in the scores they provided.

In total, 11 experts rated 2 projects on average, so 22 project gradings were received, each containing 4 values. Since the sample project set contained 9 projects, this provided us 2.4 gradings per project on average. Average variance in the scores assigned by experts was 1.1. on a nine-point-scale. We calculated final ratings per characteristic per project by taking a weighted sum (by confidence) of provided ratings.

This results in a score for each characteristic for each project, and a total score for each project which is the sum of the scores per characteristic. The calculation method is equivalent to the remediation cost paradigm as used by the SQALE model.

Response statistics are displayed in Figure 27 on page 61. The obtained quality indicators that we use as validation data are displayed in Table 13 on page 62. This information provides an answer to the research sub question *"How do Info Support experts rate the sample projects on relevant SQALE characteristics?"*

### 11.3.2. Financial Investigation

We attempted to obtain financial data on the sample projects. We first defined a financial quality indicator, which we, based on literature and established equivalence relations, defined as the amount of hours spent on problems and incidents in a project, per KLOC (thousand lines of source code) in a specified time period.

Project sizes were obtained by Sonar. Activities were obtained from monthly reports and information from the Incident Monitor information. We set the time period for the to the full year 2010.

We were able to obtain a quality indicator value for all of the projects. Due to a number of reasons, however, the quality of the obtained is disputable. Reasons for this basically come down to problems in the incident registration procedures at Info Support.

We used the obtained data to validate SQALE measurements, but kept in mind that the quality of this selection of validation data may be insufficient to draw conclusions from identified correlations.

The obtained quality indicators that we used as validation data are displayed in Table 15 on page 63. This information provides an answer to research sub question *"What is the financial quality, expressed as hours/KLOC, of the sample projects?".*

## 11.4. Other Findings

This research resulted in a number of findings that are outside the scope of the research design, but still relevant.

### 11.4.1. Recommendations for Info Support

The following recommendations for Info Support were formulated:

1. Do not attempt to develop new tools internally;
2. Keep an eye open for newer and better tools on the market;
3. Run tools on appropriate hardware;
4. Integrate analysis in PDC nightly builds;
5. Sell quality assessment as a service;
6. Assign responsibility for the method to a selected MITS employee;
7. Improve the incident registration procedure.

### 11.4.2. Enhancing the SQALE model

A number of enhancements to the SQALE quality model is proposed:

First, an extension to allow the model to work better with unbalanced configurations is proposed. The extension introduces a balancing factor for each characteristic for each language, to that an uneven distribution of rules over characteristics and rules over languages no longer leads to off-scale ratings caused by the single index-to-rating mapping used by SQALE.

Second, an extension is proposed to incorporate the quality of 'rule violation severeness' in the quality model. With this extension, the quality judgment is no longer limited to the remediation cost paradigm. Aside from being able to see how long it will take to repair issues, the SQALE user will be able to see the severeness of issues. This will aid the user in making an quality improvement plan.

## 11.5. Preliminary Research

Preliminary research provided us information necessary to develop a research design. From a literature review on quality models we learned which mathematical procedures could be used to translate source code metric values into high-level quality indicators. Several models were described. We chose to use the SQALE (Software Quality Enhancement based on Lifecycle Expectations) model in the practical aspect of this project. This model is based on the remediation cost paradigm and provides a rating for several quality characteristics of a project on an A-E rating. These characteristics are, amongst others, reliability, changeability, testability and analyzability.

Also, a survey of available tools was made by conducting a free search on the internet. These tools were evaluated using several requirements, such as support for Java and .Net, quality model support and support by business or community. It was discovered that many tools operate as a SaaS (Software-as-a-Service) solution and many tools, especially commercial ones, use a propriety quality model that does not provide required transparency. Result of this investigation was that, given the context and constraints, Sonar, in addition to a number of plugins to enable .Net support and SQALE support, was decided to be the most suitable tool to incorporate in the quality assessment method to be developed. This also implies that the tool is used in the proof of concept of this project.

## A. Bibliography

References, in alphabetical order.


(Abrahamsson 2002) P. Abrahamsson, O. Salo, J. Ronkainen, J. Warsta – *Agile Software Development Methods* – Review and Analysis. ESPOO 2002, VTT Publications 478, ISBN 951-38-6010-8.

(Al-Kilidar 2005) H. Al-Kilidar, K. Cox, B. Kitchenham – *The Use and Usefulness of the ISO/IEC 9126 Quality Standard*, 2005 International Symposium on Empirical Software Engineering, Nov. 18, 2005, Noosa Heads, Queensland, Australia

(Albrecht 1983) Albrecht, A.J. Gaffney, J.E. jr, *Software Function, Source Lines of Code and Development Effort Prediction: a Software Science Validation*, IEEE Transactions on Software Engineering, Volume SE-9 issue 6, p639-648, 1983

(April 2005) A. April e.a. – Software Maintenance Maturity Model (SM$^{MM}$): the software maintenance process model. Journal of Software Maintenance and Evolution: Research and Practice, vol. 17, issue 3, pp 197-223, May.June 2005.

(Attarzadeh 2010) Atterzadeh, I., Siew Hock Ow – *A Novel Soft Computing Model to Increase the Accuracy of Software Development Cost Estimation*. Proceedings of the 2nd International Conference on Computer and Automation Engineering, Singapore, 2010, p 603-607.

(Basili 1996) V.R. Basili, W.L. Melo, L.C. Brand – A Validation of Object-Oriented Design Metrics as Quality Indicators – IEEE Transactions on Software Engineering, vol. 22, no. 10, October 1996

(Benaroch 2002) Benaroch, M. – *Managing Information Technology Investment Risk – a Real Options Perspective*. Journal of Management Information Systems, vol. 19 isue 2, p 43, 2002.

(Bergel 2009) A. Bergel et al - *SQUALE – Software Quality Enhancement* – Proceedings of the 13th European Conference on Software Maintenance and Reengineering, Kauserslautern, 2009, p285-288

(Boehm 1999) B. Boehm, S. Chulani – *Modeling Software Defect Introduction and Removal – COQUALMO (Constructive QUALity Model)*, USC-CSE Technical Report, 1999

(Boehm 2000) B. Boehm, C. Abts, S. Chulani – *Software Development Cost Estimation Approaches – A Survey*. Annals of Software Engineering, vol. 10, no. 1-4, p177-205, 2000.

(Briand 2002) L.C. Briand, J. Wüst, Empirical Studies of Quality Models in Object Oriented Systems – Advances in Computers, volume 56, 2002

(Burris 2011) E. Burris, *Software Quality Management* (website). University of Missouri, http://www1.sce.umkc.edu/~burrise/pl/software_quality_management/, February 8, 2011.

(Caldiera 1998) G. Caldiera, G. Antoniol, R. Fiutem, C. Lokan - *Definition and Experimental Evaluation of Function Points for Object-Oriented Systems*. Fifth International Symposium on Software Metrics, Maryland, 1998

(Cast 2011) CAST Application Intelligence Platform, website, http://www.castsoftware.com/Product/Application-Intelligence-Platform.aspx, visited Feb 22, 2011

(Chamillard 2005) T. Chamillard, associate professor of CS, University of Colorado at Colorado Springs, *ISO 9126 paper handout*, January 2005 (website, visited March 21 2011), http://www.cs.uccs.edu/~chamillard/cs536/Papers/9126Handout.pdf

(Chandra 2005) Chandra, S.S. and Chandra, K. – *A Comparison of Java and C#*, Journal of Computing Sciences in Colleges, vol. 20 issue 3, Feb. 2005, Consortium for Computing Sciences in Colleges, USA.

(Chen 1993) J.Y. Chen, J.F. Lu – A new Metric for Object Oriented Design – Information and

Software Technology, vol. 35 issue 4, April 1993, pp 232-240.

(Chidamber 1994) S.R. Chidamber, C.F. Kemerer – A Metrics Suite for Object Oriented Design – IEEE Transactions on Software Engineering, vol. 20, no. 6, June 1994, pp 476-493.

(Chulani 1997) S. Chulani - *Results of Delphi for the Defect Introduction Model – Sub Model of the Cost/Quality Model Extension tot COCOMO II*, Technical Report, University of Southern California, CS Department, Center for SE, Los Angeles, USA, 1997.

(Cohen 1988) J. Cohen – *Statistical Power Analysis for the Behavioral Sciences*, second edition, 1988, Lawrence Arlbaum Associates, Inc., USA, ISBN 0-8058-0283-5.

(Correia 2008) Correia, J.P. Visser, J. – *Benchmarking Technical Quality of Software Products*, proceedings of the 15th working conference on reverse engineering, oct 15-18, 2008, Antwerp, Belgium.

(Cox 1980) Eli P. Cox – *The Optimal Number of Response Alternatives for a Scale: a Review* – Journal of Marketing Research, vol. 17, no. 4, 1980, p407-422.

(Cunningham 1992) W. Cunningham - *The WyCash Portfolio Management System* – Proceedings of the 7th annual conference on Object-Oriented Programming Systems, Languages, and Applicastions, Vancouver, British Columbia, Oct 1992, Experience Report

(Daft 2003) R.L. Daft - *Management* - 6th edition, 2003, ISBN 0 03 035138 3, Thomson Education

(Dolado 1997) J.J. Dolado – *a Study of the relationships among Albrecht and Mark II Function Points*, Lines of Code 4GL and Effort, The Journal of Systems and Software, vol. 37, issue 2, page 161 and on, 1997.

(Dromey 1995) R.G. Dromey – *A Model for Software Product Quality*. IEEE Transactions on Software Engineering, vol. 21 issue 2, February 1995.

(Fenton 1997) Norman E. Fenton – *Software Metrics: a Rigorous & Practical Approach*, 2nd edition revised printing, ISBN 0-534-95425-1, Int. Thomson Publishing, 1997

(Gruber 2007) H. Gruber, C. Korner, R. Plosch, S. Schiffer – *Tool Support for ISO 14598 based code quality assessment*. Proceedings of the 6th conference on the Quality of Information and Communications Technology, 2007, Lisbon, p21-29

(Guler 2002) I. Guler et al - *Global Competition, Institutions, and the Diffusion of Organizational Practices: The International Spread of ISO 9000 Quality Certificates*. Administrative Science Quarterly, Vol. 47 issue 2, June 2002, p207-232.

(Haughey 2010) D. Haughey – *An Introduction to Project Management* – ProjectSmart, 2010, http://www.projectsmart.co.uk/pdf/introduction-to-project-management.pdf

(Heitlager 2007) Heitlager, I., Kuipers, T., Visser, J. *A Practical Model for Measuring Maintainability*, Software Improvement Group, 2007, p30-39

(Kitchenham 1999) Kitchenham, B.A. et al – *Towards an Ontology of Software Maintenance* - Journal of Software Maintenance: Research and Practice, vol. 11 issue 6, pp 365-389, Nov/Dev 1999.

(Info Support 2011) Website Info Support BV, Veenendaal, the Netherlands: www.infosupport.com, visited Feb. 15, 2011

(Tavaf 2010) Saeedeh Jadid Tavaf – *Quality Evaluation in Transofrmation of Event Logs into Visual Representations* (MSc. Thesis), may 2009, University of Gothenburg, dept. of Applied Information Technology (report no. 2010:066)

(Kalistic 2011) Kalistic - *Agile Quality for Continuous Delivery* – website http://www.kalistick.com/, visited Feb 16, 2011.

(Laval 2008) J. Laval, A. Bergel, S. Ducasse – *Assessing the Quality of your Software with MoQam – FAMOOS*, 2nd Workshop on FAMIX and Moose in Reengineering (2008), p28-31

(Letouzey 2009) Letouzey, J.L., Coq Th. – *The SQALE method for Assessing the Quality of*

*Software Source Code*, whitepaper, 2009, DNV IT GS France, Paris (www.sqale.org)

(Letouzey 2010a) Letouzey, J.L., Coq Th. – *The SQALE Models for Assessing the Quality of Real Time Source Code*, Proceedings of the Embedded Realtime Software and Systems Symposium, May 2010, Toulouse, France.

(Letouzey 2010b) Letouzey, J.L., Coq Th. – *The SQALE Analysis Model – An analysis model compliant with the representation condition for assessing the Quality of Software Source Code,* Proceedings of the 2nd International Conference on Advances in System Testing and Validation Lifecycle, Nice, France, Aug 2010

(Li 1993) W. Li, S. Henry – Oject Oriented Metrics Which Predict Maintainability – Journal of Systems and Software, vol. 23 issue 2, November 1993, pp 111-122.

(Linke 2008) R. Lincke, J. Lundberg, W. Löwe – *Comparing Software Metrics Tools*, Proceedings of the 2008 International Symposium on Software Testing and Analysis, USA, 2008.

(Luijten 2010) B. Luijten, J. Visser - *Faster Defect Resolution with Higher Technical Quality of Software*. Proceedings of the 4th International Workshop on Software Quality and Maintainability (SQM 2010), March 15, 2010, Madrid, Spain.

(Jørgenson 1999) M. Jørgenson, *Software Quality Measurement*, Advances in Engineering Software 30, 1999, 907-912.

(Jung, 2004) H.W. Jung, S.H. Kim, C.S. Chung – *Measuring Software Product Quality: a Survey of ISO/IEC 9126*, Software (IEEE), vol 21 issue 4, 2004, p88-92.

(Lauesen 2002) Soren Lauesen: *Software Requirements: Styles and Techniques*. Pearson Education Ltd, 2002.

(Madachy 2008) R. Madachy, B. Boehm – *Assessing Quality Processes with ODC COQUALMO, Lecture Notes in Computer Science*, Making Globally Distributed Software Development a Success Story, volume 5007, 2008, p198-209

(McCal 1978) J.A. McCal, J.P. Cavano – *A Framework for the Measurement of Software Quality* – Proceedings of the Software Quality Assurance Workshop on Functional and Performance Issues, ACM, New York, 1978

(Metrixware 2011) Metrixware – website http://www.metrixware.com, visited Feb 22, 2011.

(Milicic 2011 D. Milicic - Software quality models – presentation, Blekinge Institute of Technology, Sweden, visited June 16, 2011 http://www.bth.se/tek/besq.nsf/(WebFiles)/316446EBCD98499CC12570690034683B/$FILE/chapter_1.pdf

(Moore 2006) Moore, D.S., McCabe G.P. *Statistiek in de praktijk*, theorieboek, 5e herziene druk, 2006, ISBN 90-395-2360-6.

(Oppendijk 2008) F. Oppendijk: *Comparison of the SIG Maintainability Model and the Maintainability Index*. Master thesis, Faculty of Science, University of Amsterdam, 2008.

(Ortega 2003) M. Ortega, M. Pérez, T. Rojas – *Construction of a Systemic Quality Model for Evaluating a Software Product*. Software Quality Journal 11, p 219-243, 2003.

(Reeves, 1994) Carol A. Reeves, David A. Bednar: *Defining Quality: Alternatives and Implications*. The Academy of Management Review, Vol. 19 No. 3, Juli 1994.

(Rodgers 1988) J.L. Rodgers, W.A. Nicewander – *Thirteen ways to look at the Correlation Coefficient*. The American Statistician, vol. 42 issue 1, p59-66, Feb. 1988

(Silberschatz 2009) A. Silberschatz – *Operating System Concepts*, 8th edition, 2009, Wiley.

(Slaughter 1998) Slaughter, S.A., Harter, D.E., Krishnan, M.S. – *Evaluating the Cost of Software Quality* – Communications of the ACM, vol. 431 no. 8, Aug 1998.

(Slot 2010) Slot, R. – *A Method for Valuing Enterprise Architecture based Business Transformation and Measuring the Value of Solutions Architecture*, PhD thesis, University of Amsterdam, the Netherlands, 2010.

(Sonar 2011) Sonar – website http://www.sonarsource.org visited Feb. 15, 2011

(Sonar .net 2011) Sonar .NET plugin website http://docs.codehaus.org/display/SONAR/.Net+plugin visited Feb 18, 2011

(Sonar nemo 2011) Sonar 'Nemo' public demo, website, http://nemo.sonarsource.org/, visited June 16, 2011.

(Sonar SIG 2011) Sonar SIG maintainability plugin website, http://docs.codehaus.org/display/SONAR/SIG+Maintainability+Model

(Standish, 2001) The Standish Group International, Inc. - *Extreme Chaos*, 2001, p1-5

(Sqale 2011) SQALE – Method Definition Document, version 0.8, July 2010, http://www.sqale.org/wp-content/uploads/2010/08/SQALE-Method-EN-V0-08.pdf

(Squale 2011) SQualE – Software Quality Enhancement – website http://www.squale.org/ visited Feb. 15, 2011

(Sotos 2007) Sotos, A.E.C. e.a. *The Non-Trasitivity of Peasron's Correlation Coefficient: An Educational Perspective*, International Statistical Institute, 56th session, 2007

(Squoring 2011) SQuORING – A Collaborative Platform for Optimizing Source Quality – website http://www.squoring.org visited Mar. 11, 2011.

(Tayi 1998) G.K. Tayi, D. P. Ballou, *Examining Data Quality* - Communications of the ACM, vol. 41 issue 2, feb. 1998.

(Techdebt 2011) Second International Workshop on Managing Technical Debt, May 21, 2011, Honolulu, Hawaii, USA (website) http://www.sei.cmu.edu/community/td2011/program/index.cfm

(Trienekens 2008) J. Trienekens, P. Zuurbier - Quality and Safety Standards in the Food Industry, developments and challenges. International Journal of Production Economics, vol. 113, issue 1, May 2008, pp107-122

(Verhoef 2005) C. Verhoef – *Quantifying the Value of IT Investments*. Science of Computer Programming, vol. 56, 2005, p315-342.

(Verhoef 2006) C. Verhoef - "IT-hoogleraar rekent af met TCO – Chris Verhoef bedenkt methodiek voor financiële onderbouwing IT", CIO magazine, april 2006, text by Gijs Hillenius.

(Wieringa, 2007) Roel Wieringa: *Writing a Report About Design Research*, 2007. From: Reader 'Problem Analysis and Solution Requirements', dept. of Management and Governance, University of Twente, October 2008, p87 – p96.

(Williams 2001) Williams, T.W., Mercer, M.R., Mucha, J.P., Kapur, R. – *Code Coverage, what does it mean in terms of Quality*? Proceedings of the 2001 Reliability and Maintainability Symposium, Philadelphia, PA, USA, January 2001.

(Woolderink 2007) C.B. Woolderink: *Het bepalen van de onderhoudbaarheid van objectgeorienteerde broncode door middel van metrieken*. Master thesis, Unversity of Amterdam, 2007.

(Zubrow 2004) D. Zubrow – *Software Quality Requirements and Evaluation*, the ISO 25000 Series – PSM Technical Working Group, Feb. 2004, Carnegie Mellon University, Pittsburg, http://www.psmsc.com/Downloads/TWGFeb04/04ZubrowISO25000SWQualityMeasurement.pdf

## B. Lists of tables and figures

### List of Tables

## List of Figures

## C. Digital Resources

This research uses, or has resulted in, a number of artifacts that cannot be published in this document. These are available for download online, at http://www.erikhegeman.nl/research/qqm This appendix describes the digital resources that are available.

### 1. Merge-solutions

Merge-solutions, a command line tool to merge related Visual Studio Solutions, required for aggregation of analysis results: Source: http://code.google.com/p/merge-solutions/

Source code is available through Subversion using the following SVN checkout url:

```
http://merge-solutions.googlecode.com/svn/trunk/merge-solutions-read-only
```

The executable is also available at the download location in tools/merge-solutions.exe

Tool syntax:

```
merge-solutions.exe [/nonstop] [/fix] [/config solutionlist.txt] [/out
merged.sln] [solution1.sln solution2.sln ...]

        /fix: Regenerates duplicate project guids and replaces them in
corresponding project/solution files
              requires write-access to project and solution files

        /config solutionlist.txt: Takes list of new-line separated solution
paths from solutionlist.txt file

        /out merged.sln: path to output solution file. Default is 'merged.sln'

        /nonstop: do not prompt for keypress if there were errors/warnings

        solution?.sln - list of solutions to be merged
```

### 2. Available Rules List

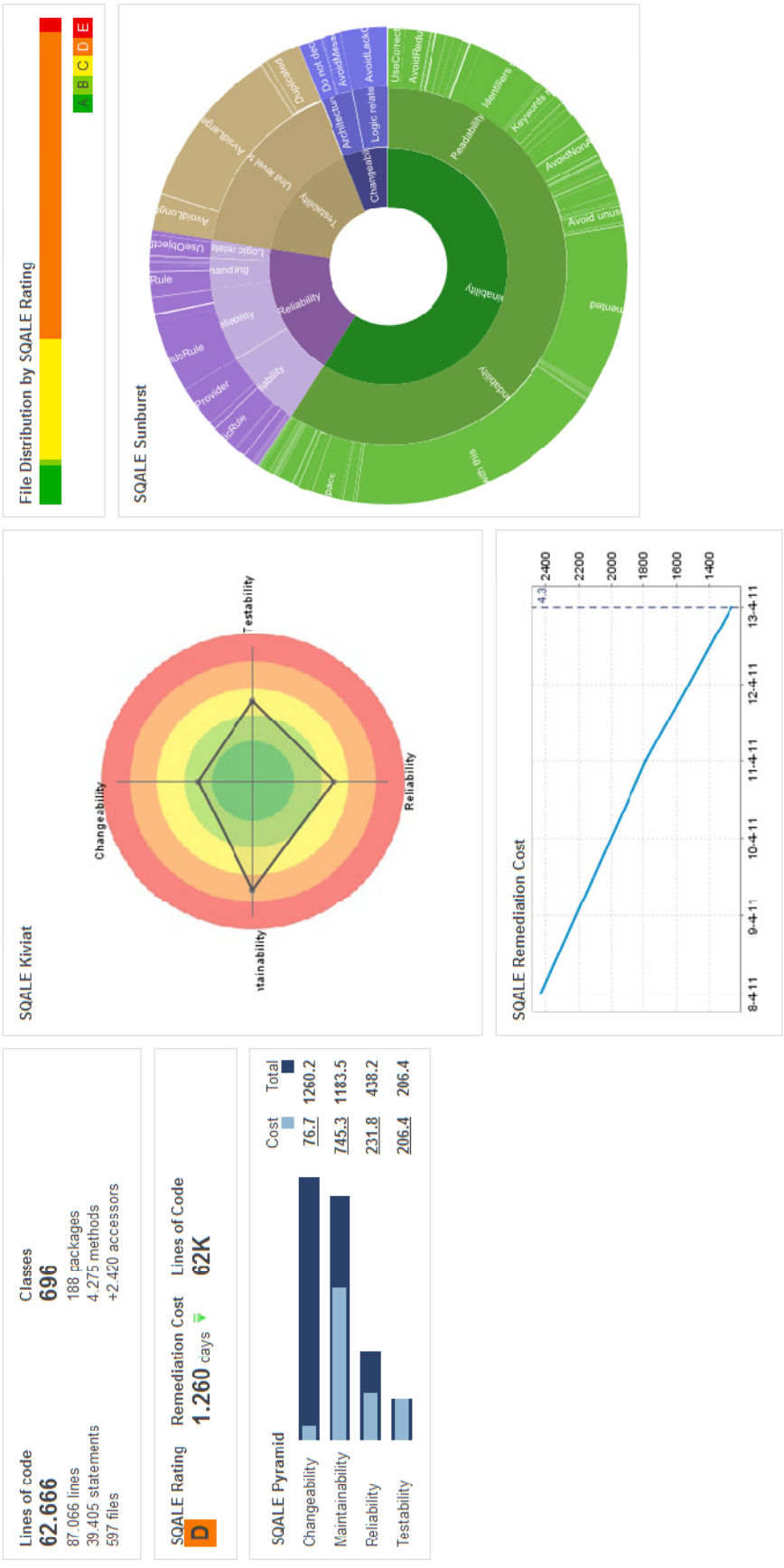
A full list of rules that can be used in Sonar is available as a digital resource. This is a 18-page Excel document, indicating per rule the language (Java or C#) for which it is applicable and the rule engine that provides the metric. The list can be found in the Documents folder as 'Full list of available rules'.

### 3. Configurations

In the course of this research, a number of Sonar configurations were used. These configurations consists of two components, namely a 'quality profile', which is the general Sonar configuration that specifies which rules to use, and a 'SQALE model', which defines the mapping of rules to sub characteristics and the remediation costs of rules. The following versions are available:

- Sonar Quality Profiles and SQALE configurations (available for both Java and C#):
  - 'initial' – the default configuration with all rules enabled
  - 'post-initial-pre-calibration' - the configuration with all rules enabled except those listed in Table 18 on page 71.
  - 'endeavour-calibrated' – the configuration that uses a rule set based upon Info Support PDC rules.

## D. Sonar Dashboard Example



## E. Survey Design

*This appendix contains the actual design of the MITS survey that was conducted. It is in Dutch, since the survey has been conducted in Dutch due to the fact that all MITS employees speak Dutch and for most this is their mother tongue. For all projects that have been selected according to paragraph 3.1 of this thesis, an empty schema conforming to the example is included in the actual survey document.*

---

# Enquete Kwaliteitsperceptie

Deze enquete dient ter verzameling van gegevens over de perceptie van kwaliteit van projecten bij Info Support. De uitkomsten worden gebruikt om resultaten van kwaliteitsmetingen op basis van broncode te valideren. Dit in het kader van een WO-afstudeerproject, met als doel het ontwikkelen van een methode voor automatische codekwaliteitsbeoordeling.

De enquete is verspreid onder alle MITS-medewerkers. Resultaten worden anoniem verwerkt en er zijn geen goede of foute antwoorden. Het invullen van de enquete kan in ca. 10 minuten. Indienen van de ingevulde enquete kan door het ingevulde Word-bestand te mailen naar het mailadres onderaan de ze pagina.

**Lees deze korte instructies voordat u deze enquete invult.**

In deze enquete wordt u verzocht om MITS-projecten te beoordelen op een aantal aspecten. Per project vult u per aspect in, in hoeverre u van mening bent dat het betreffende project voldoet aan het aspect. Op de volgende pagina staat een voorbeeld. Ook vult u in hoe goed u het project kent. Indien u aangeeft geen kennis van het project te hebben, hoeft u het project verder niet te beoordelen.

Over de volgende aspecten wordt u gevraagd per project een beoordeling toe te kennen. De beschrijving is gebaseerd op de documentatie van de ISO-standaard en het SQALE quality model (www.sqale.org)

- ***Analyseerbaarheid*: karakteriseert de leesbaarheid en begrijpbaarheid van de broncode van de applicatie.**
- ***Veranderbaarheid*: karakteriseert de hoeveelheid moeite die moet worden gedaan om een verandering in het systeem te realiseren.**
- ***Betrouwbaarheid*: Karakteriseert de mate waarin het systeem 'tegen een stootje kan' tijdens gebruik, bijv. door correct te reageren op foutieve invoer, correct fouten af te handelen, geen thread-problemen te kennen etc.**
- ***Testbaarheid*: karakteriseert de hoeveelheid moeite die nodig is om een verandering in het systeem te testen.**

Bij voorbaat dank voor uw deelname!

Erik Hegeman
erikh@infosupport.com
Afstudeerder Universiteit Twente

## Voorbeeld

In Tabel 1 ziet u een voorbeeld van een ingevuld schema van het project 'Voorbeeldproject X'. Op elk van de kwaliteitsaspecten is een score ingevoerd op de schaal 'volledig mee oneens' tot 'volledig mee eens'. Tevens schat de deelnemer zijn/haar kennis van het project in als 'beperkt'.

De schema's die u als deelnemer kunt invullen beginnen op de volgende pagina.

| **Voorbeeldproject X** | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Geen | | | Beperkt | | | Veel | | |
| Uw kennis van het project | | | | | X | | | | |
| **Het project is…** | Volledig mee oneens | sterk mee oneens | mee oneens | enigszins mee oneens | Neutraal | enigszins mee eens | mee eens | sterk mee eens | Volledig mee eens |
| Analyseerbaar | | | | | | | X | | |
| Veranderbaar | | | | | X | | | | |
| Betrouwbaar | | | X | | | | | | |
| Testbaar | | | | | | X | | | |

Tabel 1

*(After this example, nine similar boxes for the selected projects follow. These are not included in this thesis appendix)*

## F. Survey Results – raw result overview [CONFIDENTIAL]

The unrestricted version of this thesis does not contain this confidential appendix.

## G. Method Setup notes

### 1. General Setup

**Using a Microsoft Windows XP SP3 Virtual machine (IS Unattended Install)**
**Software: Notepad++, WinSCP, WinRAR, all windows updates**

**- Downloaded Sonar 2.6 to c:\apps\sonar (a newer version of Sonar is currently available. You can try to use the newest version, but this has not been tested).**
**- Downloaded and installed MySQL community server 5.5.9, detailed configuration, used all standard settings. The test server uses root password 'admin'.**
**- Included MySQL bin directory in windows PATH system environment variable**
**- Modified sonar/conf/sonar.properties to use MySQL instead of Derby (comment/uncomment the relevant lines)**
**- Created the database and user on the SQL server:**

```
mysql –u root –p
admin
create database sonar;
create user 'sonar'@'%' identified by 'sonar';
grant all privileges on *.* to 'sonar';
flush privileges;
exit
```

**- Added firewall exception for TCP port 9000**
**- Execute**

```
sonar/bin/windows-x86-32/InstallNTService
```

**- Start service and wait for three minutes for DB auto-initialization**
**- Test: browse to http://localhost:9000 – You should now see the Sonar main interface. Admin login should be possible using credentials admin/admin**
**- Downloaded Maven 3.03 to apps/maven**
**- Inserted this in maven/conf/settings.xml:**

```
<profile>
      <id>sonar</id>
      <activation>
            <activeByDefault>true</activeByDefault>
      </activation>
      <properties>
            <sonar.jdbc.url>
              jdbc:mysql://localhost:3306/sonar?useUnicode=
                  true&amp;characterEncoding=utf8
            </sonar.jdbc.url>
            <sonar.jdbc.driverClassName>
                  com.mysql.jdbc.Driver
            </sonar.jdbc.driverClassName>
            <sonar.jdbc.username>sonar</sonar.jdbc.username>
            <sonar.jdbc.password>sonar</sonar.jdbc.password>
            <sonar.host.url>http://localhost:9000</sonar.host.url>
      </properties>
</profile>
```

**Installed prerequisites for the Maven sonar plugin**
**- .NET SDKs version 2.0, 3.5 and 4.0**
**- Gallio**

**- PartCover**
**- Fxcop (use version 10.0 which is part of Windows SDK 7.1, installer can be found in the Fxcop folder of the installation after installing the SDK)**
**- Gendarme**
**- SourceMonitor**
**- StyleCop**

**Note that it is recommended to install Microsoft Visual Studio and Silverlight libraries on the analysis machine, since .Net projects may depend on them. If a dependency is missing, the analysis logfile will indicate which additional components should be installed. Appendix H shows a list of all software packages installed on the PoC VM.**

- **added to maven/conf/settings.xml:**

```
 <profile>
      <id>dotnet</id>
      <activation>
        <!-- Optional activation by default -->
        <activeByDefault>true</activeByDefault>
      </activation>
      <properties>
      <!--Locations of the .Net installations (pick the one you need)-->
      <!--(below the default values for each dotnet version supported)-->
      <dotnet.2.0.sdk.directory>
            C:/WINDOWS/Microsoft.NET/Framework/v2.0.50727
      </dotnet.2.0.sdk.directory>
<dotnet.3.5.sdk.directory>
            C:/WINDOWS/Microsoft.NET/Framework/v3.5
      </dotnet.3.5.sdk.directory>
      <dotnet.4.0.sdk.directory>
            C:/WINDOWS/Microsoft.NET/Framework/v4.0.30319
      </dotnet.4.0.sdk.directory>
      <!-- Location of the Gallio installation-->
      <gallio.directory>C:/Program Files/Gallio</gallio.directory>

      <!-- Location of FxCop installation-->
      <fxcop.directory>
            C:/Program Files/Microsoft FxCop 10.0
      </fxcop.directory>

      <!-- Location of PartCover installation-->
      <partcover.directory>
            C:/Program Files/PartCover/PartCover .NET 4.0
      </partcover.directory>

      <!-- Location of Source Monitor installation-->
      <sourcemonitor.directory>
            C:/Program Files/SourceMonitor
      </sourcemonitor.directory>

      <!-- Location of Gendarme installation -->
      <gendarme.directory>
            C:/Program Files/gendarme-2.6-bin
      </gendarme.directory>

      <!-- Location of StyleCop installation -->
      <stylecop.directory>
```

```
            C:/Program Files/Microsoft StyleCop 4.4.0.14
        </stylecop.directory>
        </properties>
    </profile>
```

- **Added Maven bin directory to system path**
- **Defined the JAVA_HOME environment variable (set to jdk1.6.0_x folder)**
- **Modified mvn.bat to increase the Java Heap size to 1GB**

```
@set MAVEN_OPTS=-Xmx1024M -Xms1024M
```

- **Downloaded all .jar's listed at http://docs.codehaus.org/display/SONAR/.Net+plugin to apps/sonar/extensions/plugins**
- **Restart sonar service to load plugins**

**The SQALE plugin can simply be added to the apps/sonar/extensions/plugins directory. To configure SQALE, be sure to perform the following steps from the Sonar web interface:**

- **Load the quality profile XML (required for C# support)**
- **Set the global SQALE parameters (you can use the default settings)**
- **Enable all rules for all projects using the Quality dashboard**
- **Configure the dashboard (add SQALE widgets)**

## 2. Setup Test

Run the following command from both a directory containing a Visual Studio Solution (.sln file) of a .Net project and a pom.xml (see below), and a Java project, also defined by a pom.xml. This will result in test project measurements being available through the Sonar web interface.

```
mvn install sonar:sonar
```

## 3. General Usage

To reset the Sonar database:

- **Stop Sonar service**
- **Using mysql command prompt, drop all tables from database 'sonar'**
- **Start Sonar service (automatically reconstructs DB schema)**
- **Wait for a few minutes before attempting to use Sonar**

To increase performance, run Sonar as a Tomcat servlet (optional):

- **Download Apache Tomcat 7 (download from http://tomcat.apache.org)**
- **Perform standard Windows Service installation**
- **Run build-war.bat from sonar/bin**
- **Move sonar.war to Tomcat webapps root folder, autodeploy will start**
- **Start the application from the management app at http://localhost:8080**
- **Add a firewall exception for TCP port 8080 to allow remote access**

To analyaze Java Projects:

From the command line, go to the project directory containing pom.xml and type

```
mvn install sonar:sonar > maven.log
```

After the command has completed, check the error log to see if there are any errors. If there are, try to resolve these and re-run the command. If there are no errors, compilation has succeeded and the project should be visible in the Sonar home screen at http://localhost:9000

To analyze .Net Projects:

These projects are defined as a set of solutions. A tool is needed to merge these solutions into a single artifact. From the root of the directory structure, use the following command sequence:

```
for /R %i in (*.sln) do echo %i >> solutions.txt
merge-solutions /nonstop /fix
           /config solutions.txt /out merged.sln
```

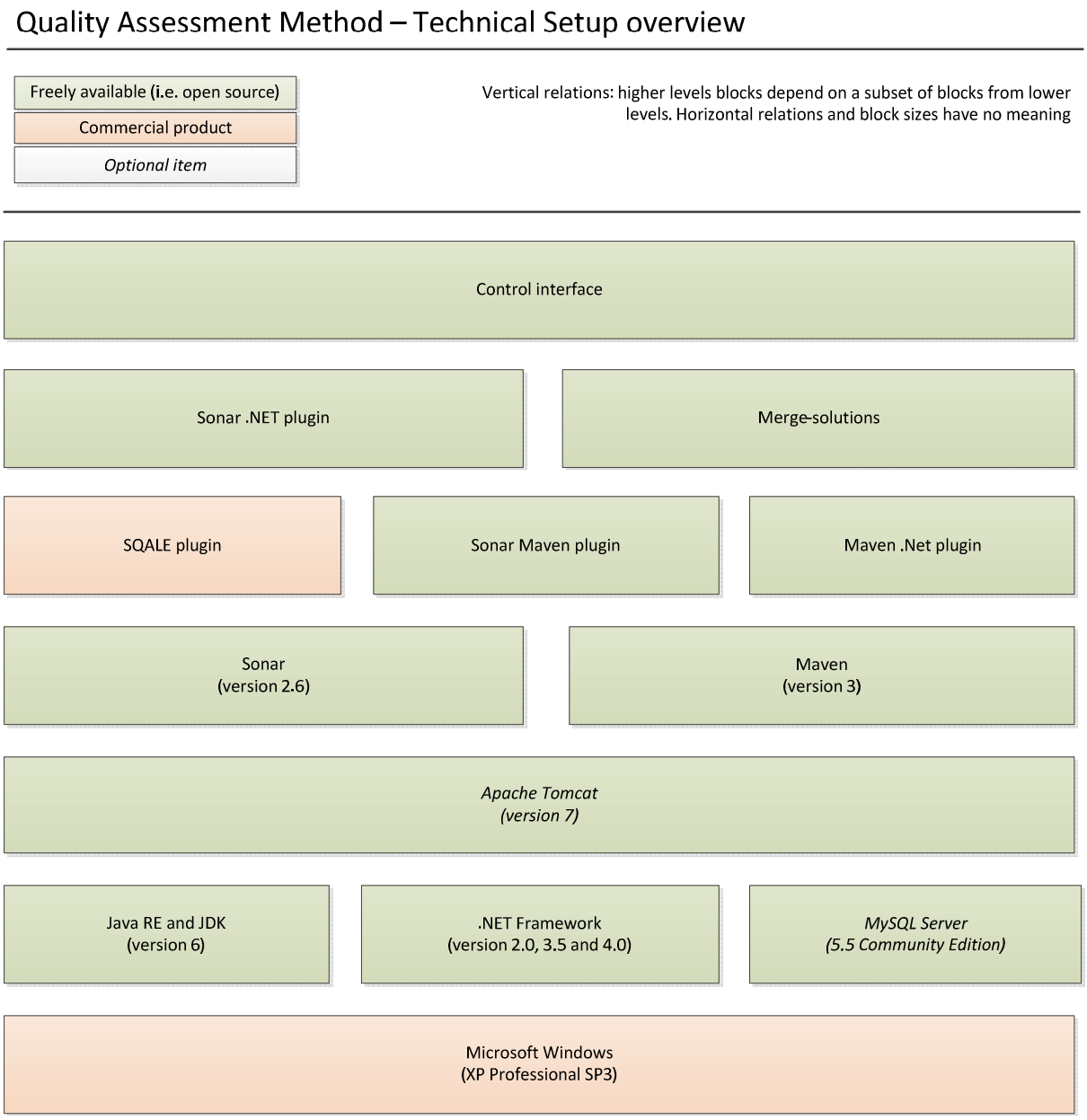
Also, define a pom.xml file based upon the default structure (shown below), fill out the appropriate variables. After that, run

```
mvn install sonar:sonar > maven.log
```

## 4. Default Pom.XML for .Net projects

```
<!--
        POM.XML template for Sonar dotnet support
        March 29, 2011 - Erik Hegeman, Info Support BV

        Set the following parameters:
        - Group, artifact, version and name
        - SLNFile
-->

<project xmlns=http://maven.apache.org/POM/4.0.0
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
        http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <!-- This block needs to be filled out per instance of the file -->
  <groupId>GroupIdentifier</groupId>
  <artifactId>ArtifactIdentifier</artifactId>
  <version>UndefinedVersion</version>
  <name>Name</name>

  <packaging>sln</packaging>
  <properties>
    <visual.studio.solution>SLNFile</visual.studio.solution> <!--define SLN file here-->
    <visual.test.project.pattern>*.Tests</visual.test.project.pattern>
    <dotnet.tool.version>3.5</dotnet.tool.version>
    <sonar.language>cs</sonar.language>
  </properties>
  <build>
    <plugins>
      <plugin>
        <groupId>org.codehaus.sonar-plugins.dotnet</groupId>
        <artifactId>maven-dotnet-plugin</artifactId>
        <extensions>true</extensions>
       <version>0.5</version>
      </plugin>
      <plugin>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>sonar-maven-plugin</artifactId>
       <version>2.0-beta-2</version>
        <configuration>
          <language>cs</language>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>
```

## H. PoC Technical Setup Overview



# I. SQALE configuration overview

| # | C | Sc | Rule name | Lang | Remediation function (cost/violation in days) |
|---|---|---|---|---|---|
| 1 | Analysability | Readability | Constant Name | java | linear (0.01) |
| 2 | | | Local Final Variable Name | java | linear (0.03) |
| 3 | | | Local Variable Name | java | linear (0.03) |
| 4 | | | Member name | java | linear (0.03) |
| 5 | | | Method Name | java | linear (0.03) |
| 6 | | | Package name | java | linear (0.125) |
| 7 | | | Parameter Name | java | linear (0.01) |
| 8 | | | Static Variable Name | java | linear (0.01) |
| 9 | | | Type Name | java | linear (0.03) |
| 10 | | | Redundant import | java | constant_resource (0.01) |
| 11 | | | Unused Imports | java | constant_resource (0.01) |
| 12 | | | Line Length | java | constant_resource (0.01) |
| 13 | | | Operator Wrap | java | linear (0.03) |
| 14 | | | Paren Pad | java | constant_resource (0.01) |
| 15 | | | Modifier Order | java | linear (0.01) |
| 16 | | | Avoid Nested Blocks | java | linear (0.125) |
| 17 | | | Right Curly | java | linear (0.01) |
| 18 | | | Array Type Style | java | linear (0.01) |
| 19 | | | Upper Ell | java | linear (0.01) |
| 20 | | | Abstract Class Name | java | linear (0.03) |
| 21 | | | Anon Inner Length | java | linear (0.125) |
| 22 | | | Annotation Use Style | java | linear (0.03) |
| 23 | | | Array Trailing Comma | java | linear (0.01) |
| 24 | | | Declaration Order | java | linear (0.01) |
| 25 | | | Generic Whitespace | java | constant_resource (0.01) |
| 26 | | | Import Order | java | constant_resource (0.01) |
| 27 | | | Indentation | java | constant_resource (0.01) |
| 28 | | | Left Curly | java | constant_resource (0.01) |
| 29 | | | Method Param Pad | java | linear (0.01) |
| 30 | | | Multiple Variable Declarations | java | linear (0.03) |
| 31 | | | No Whitespace Before | java | linear (0.01) |
| 32 | | | No Whitespace After | java | constant_resource (0.01) |
| 33 | | | Redundant Modifier | java | linear (0.01) |
| 34 | | | Whitespace After | java | constant_resource (0.01) |
| 35 | | | Whitespace Around | java | constant_resource (0.01) |
| 36 | | | Abstract naming | java | linear (0.03) |
| 37 | | | Avoid Using Octal Values | java | linear (0.125) |
| 38 | | | Boolean Get Method Name | java | linear (0.03) |
| 39 | | | Dont Import Java Lang | java | constant_resource (0.01) |
| 40 | | | Duplicate Imports | java | constant_resource (0.01) |
| 41 | | | For Loops Must Use Braces | java | linear (0.03) |
| 42 | | | For Loop Should Be While Loop | java | linear (0.03) |
| 43 | | | Import From Same Package | java | constant_resource (0.01) |
| 44 | | | Long Variable | java | linear (0.03) |
| 45 | | | Message Driven Bean And Session Bean Naming Convention | java | linear (0.01) |
| 46 | | | Naming - Avoid dollar signs | java | linear (0.01) |
| 47 | | | Naming - Avoid field name matching method name | java | linear (0.01) |
| 48 | | | Naming - Avoid field name matching type name | java | linear (0.01) |
| 49 | | | Naming - Class naming conventions | java | linear (0.01) |
| 50 | | | Naming - Method naming conventions | java | linear (0.01) |

| | | | |
|---|---|---|---|
| 51 | Naming - Method with same name as enclosing class | java | linear (0.01) |
| 52 | Naming - Misleading variable name | java | linear (0.01) |
| 53 | Naming - Short method name | java | linear (0.01) |
| 54 | Naming - Suspicious Hashcode method name | java | linear (0.01) |
| 55 | Naming - Suspicious constant field name | java | linear (0.01) |
| 56 | Naming - Suspicious equals method name | java | linear (0.01) |
| 57 | Naming - Variable naming conventions | java | linear (0.01) |
| 58 | Package case | java | linear (0.03) |
| 59 | Proper Logger | java | linear (0.03) |
| 60 | Remote Interface Naming Convention | java | linear (0.01) |
| 61 | Remote Session Interface Naming Convention | java | linear (0.01) |
| 62 | Short Variable | java | linear (0.03) |
| 63 | String To String | java | linear (0.03) |
| 64 | Too Many Static Imports | java | linear (0.125) |
| 65 | Too few branches for a switch statement | java | linear (0.03) |
| 66 | Unnecessary Return | java | linear (0.03) |
| 67 | Unused imports | java | constant_resource (0.01) |
| 68 | Use Collection Is Empty | java | linear (0.03) |
| 69 | Useless String Value Of | java | linear (0.03) |
| 70 | While Loops Must Use Braces | java | linear (0.03) |
| 71 | Trailing Comment | java | linear (0.01) |
| 72 | Local Home Naming Convention | java | linear (0.01) |
| 73 | Local Interface Session Naming Convention | java | linear (0.01) |
| 74 | Unnecessary Final Modifier | java | linear (0.01) |
| 75 | Unnecessary parentheses | java | linear (0.03) |
| 76 | Unused Modifier | java | linear (0.03) |
| 77 | Use String Buffer Length | java | linear (0.03) |
| 78 | Method names should start with a lower case letter | java | linear (0.03) |
| 79 | Class names should start with an upper case letter | java | linear (0.03) |
| 80 | Dodgy - Class too big for analysis | java | linear (0.375) |
| 81 | Field names should start with a lower case letter | java | linear (0.03) |
| 82 | DoNotPrefixValuesWithEnumNameRule | cs | linear (0.03) |
| 83 | AvoidUnneededCallsOnStringRule | cs | linear (0.03) |
| 84 | PreferEventsOverMethodsRule | cs | linear (0.03) |
| 85 | UseCorrectSuffixRule | cs | linear (0.03) |
| 86 | UseCorrectPrefixRule | cs | linear (0.03) |
| 87 | UseCorrectCasingRule | cs | linear (0.03) |
| 88 | AvoidRedundancyInTypeNameRule | cs | linear (0.03) |
| 89 | AvoidRedundancyInMethodNameRule | cs | linear (0.3) |
| 90 | ObsoleteMessagesShouldNotBeEmptyRule | cs | linear (0.03) |
| 91 | Do not place regions within elements | cs | linear (0.01) |
| 92 | Documentation headers must not contain blank lines | cs | linear (0.01) |
| 93 | Documentation lines must begin with single space | cs | linear (0.01) |
| 94 | Element documentation header must be preceded by blank line | cs | linear (0.01) |
| 95 | Element documentation headers must not be followed by blank line | cs | linear (0.01) |
| 96 | File may only contain a single class | cs | constant_resource (0.03) |
| 97 | Opening attribute brackets must be spaced correctly | cs | constant_resource (0.01) |
| 98 | Opening curly brackets must be spaced correctly | cs | constant_resource (0.01) |
| 99 | Opening curly brackets must not be followed by blank line | cs | constant_resource (0.01) |
| 100 | Opening curly brackets must not be preceded by blank line | cs | constant_resource (0.01) |
| 101 | Opening generic brackets must be spaced correctly | cs | constant_resource (0.01) |
| 102 | Opening parenthesis must be on declaration line | cs | constant_resource (0.01) |
| 103 | Opening parenthesis must be spaced correctly | cs | constant_resource (0.01) |
| 104 | Opening square brackets must be spaced correctly | cs | constant_resource (0.01) |

| | | | |
|---|---|---|---|
| 105 | Parameter list must follow declaration | cs | linear (0.01) |
| 106 | Operator keyword must be followed by space | cs | linear (0.01) |
| 107 | Parameter must follow comma | cs | linear (0.01) |
| 108 | Parameter must not span multiple lines | cs | linear (0.01) |
| 109 | Parameters must be on same line or separate lines | cs | constant_resource (0.01) |
| 110 | Single line comment must be preceded by blank line | cs | constant_resource (0.01) |
| 111 | Single line comments must begin with single space | cs | constant_resource (0.01) |
| 112 | Single line comments must not be followed by blank line | cs | constant_resource (0.01) |
| 113 | Single line comments must not use documentation style slashes | cs | constant_resource (0.01) |
| 114 | Compound words should be cased correctly | cs | linear (0.01) |
| 115 | Identifiers should be cased correctly (FxCop10) | cs | linear (0.001) |
| 116 | Resource string compound words should be cased correctly | cs | linear (0.01) |
| 117 | Code must not contain multiple blank lines in a row | cs | linear (0.01) |
| 118 | Code must not contain multiple statements on one line | cs | linear (0.01) |
| 119 | Code must not contain multiple whitespace in a row | cs | linear (0.01) |
| 120 | Commas must be spaced correctly | cs | linear (0.01) |
| 121 | Field names must not contain underscore | cs | linear (0.01) |
| 122 | Remove unnecessary code | cs | linear (0.03) |
| 123 | Closing attribute brackets must be spaced correctly | cs | linear (0.01) |
| 124 | Closing curly brackets must be spaced correctly | cs | linear (0.01) |
| 125 | Element must begin with upper case letter | cs | linear (0.01) |
| 126 | Element must begin with lower case letter | cs | linear (0.01) |
| 127 | Negative signs must be spaced correctly | cs | linear (0.01) |
| 128 | Positive signs must be spaced correctly | cs | linear (0.01) |
| 129 | Symbols must be spaced correctly | cs | linear (0.01) |
| 130 | While do footer must not be preceded by blank line | cs | linear (0.01) |
| 131 | Protected must come before internal | cs | linear (0.01) |
| 132 | Increment decrement symbols must be spaced correctly | cs | linear (0.01) |
| 133 | Elements must be separated by blank line | cs | linear (0.01) |
| 134 | Const field names must begin with upper case letter | cs | linear (0.01) |
| 135 | Accessible fields must begin with upper case letter | cs | linear (0.01) |
| 136 | Identifiers should be spelled correctly (FxCop10) | cs | linear (0.01) |
| 137 | Identifiers should not contain underscores (FxCop10) | cs | linear (0.01) |
| 138 | Resource strings should be spelled correctly | cs | linear (0.01) |
| 139 | Use preferred terms | cs | linear (0.03) |
| 140 | All accessors must be multi line or single line | cs | linear (0.03) |
| 141 | Closing parenthesis must be on line of opening parenthesis | cs | linear (0.01) |
| 142 | Closing parenthesis must be on line of last parameter | cs | linear (0.01) |
| 143 | Closing generic brackets must be spaced correctly | cs | linear (0.01) |
| 144 | Closing parenthesis must be spaced correctly | cs | linear (0.01) |
| 145 | Closing square brackets must be spaced correctly | cs | linear (0.01) |
| 146 | Colons must be spaced correctly | cs | linear (0.01) |
| 147 | Dereference and access of symbols must be spaced correctly | cs | linear (0.01) |
| 148 | Keywords must be spaced correctly | cs | linear (0.01) |
| 149 | Member access symbols must be spaced correctly | cs | linear (0.01) |
| 150 | Semicolons must be spaced correctly | cs | linear (0.01) |
| 151 | Comma must be on same line as previous parameter | cs | linear (0.01) |
| 152 | Element must not be on single line | cs | linear (0.01) |
| 153 | Field names must not begin with underscore | cs | linear (0.01) |
| 154 | Using directives must be ordered alphabetically by namespace | cs | linear (0.01) |
| 155 | Using alias directives must be ordered alphabetically by alias name | cs | linear (0.01) |
| 156 | Field names must not use hungarian notation | cs | linear (0.01) |
| 157 | Closing curly brackets must not be preceded by blank line | cs | linear (0.01) |
| 158 | Chained statement blocks must not be preceded by blank line | cs | linear (0.01) |

| | | | | |
|---|---|---|---|---|
| 159 | | Closing curly bracket must be followed by blank line | cs | linear (0.01) |
| 160 | | Curly brackets for multi line statements must not share line | cs | linear (0.01) |
| 161 | | Void return value must not be documented | cs | linear (0.01) |
| 162 | | UseSingularNameInEnumsUnlessAreFlagsRule | cs | linear (0.03) |
| 163 | | AvoidConstructorsInStaticTypesRule | cs | linear (0.03) |
| 164 | | AvoidNonAlphanumericIdentifierRule | cs | linear (0.03) |
| 165 | | DoNotPrefixEventsWithAfterOrBeforeRule | cs | linear (0.03) |
| 166 | | DoNotUseEnumIsAssignableFromRule | cs | linear (0.03) |
| 167 | | PreferTryParseRule | cs | linear (0.03) |
| 168 | | CentralizePInvokesIntoNativeMethodsTypeRule | cs | linear (0.125) |
| 169 | | Events should not have before or after prefix | cs | linear (0.03) |
| 170 | | Statement must not use unnecessary parenthesis | cs | linear (0.03) |
| 171 | | Statement must not be on single line | cs | linear (0.03) |
| 172 | | Static elements must appear before instance elements | cs | linear (0.03) |
| 173 | | Remove delegate parenthesis when possible | cs | linear (0.03) |
| 174 | | Preprocessor keywords must not be preceded by space | cs | linear (0.01) |
| 175 | | Nullable type symbols must not be preceded by space | cs | linear (0.01) |
| 176 | | Block statements must not contain embedded comments | cs | linear (0.03) |
| 177 | | Block statements must not contain embedded regions | cs | linear (0.03) |
| 178 | **Understandability** | Avoid Inline Conditionals | java | linear (0.03) |
| 179 | | JavaNCSS | java | linear (0.375) |
| 180 | | Missing Deprecated | java | linear (0.01) |
| 181 | | Package Annotation | java | linear (0.03) |
| 182 | | Uncommented Main | java | linear (0.125) |
| 183 | | Abstract Class Without Abstract Method | java | linear (0.125) |
| 184 | | Accessor Class Generation | java | linear (0.125) |
| 185 | | Assignment In Operand | java | linear (0.03) |
| 186 | | Avoid Multiple Unary Operators | java | linear (0.125) |
| 187 | | Avoid Using Volatile | java | linear (0.125) |
| 188 | | Avoid unnecessary comparisons in boolean expressions | java | linear (0.03) |
| 189 | | Basic - Empty Initializer | java | linear (0.03) |
| 190 | | Class with only private constructors should be final | java | linear (0.03) |
| 191 | | Clone method must implement Cloneable | java | linear (0.03) |
| 192 | | Collapsible If Statements | java | linear (0.03) |
| 193 | | Confusing Ternary | java | linear (0.03) |
| 194 | | Empty Finalizer | java | linear (0.03) |
| 195 | | Empty Finally Block | java | linear (0.03) |
| 196 | | Empty Static Initializer | java | linear (0.03) |
| 197 | | Empty Switch Statements | java | linear (0.03) |
| 198 | | Empty Synchronized Block | java | linear (0.03) |
| 199 | | Empty Try Block | java | linear (0.03) |
| 200 | | Empty While Stmt | java | linear (0.03) |
| 201 | | Excessive Class Length | java | linear (0.125) |
| 202 | | Excessive Method Length | java | linear (0.125) |
| 203 | | Finalize Only Calls Super Finalize | java | linear (0.03) |
| 204 | | Finalize Should Be Protected | java | linear (0.03) |
| 205 | | Immutable Field | java | linear (0.03) |
| 206 | | Local variable could be final | java | linear (0.03) |
| 207 | | More Than One Logger | java | linear (0.03) |
| 208 | | Ncss Constructor Count | java | linear (0.125) |
| 209 | | Return empty array rather than null | java | linear (0.125) |
| 210 | | Signature Declare Throws Exception | java | linear (0.125) |
| 211 | | Simplify boolean returns | java | linear (0.125) |
| 212 | | Singular Field | java | linear (0.03) |

| | | | |
|---|---|---|---|
| 213 | Uncommented Empty Method | java | linear (0.03) |
| 214 | Uncommented Empty Constructor | java | linear (0.03) |
| 215 | Unconditional If Statement | java | linear (0.03) |
| 216 | Unused formal parameter | java | linear (0.03) |
| 217 | Write Tag | java | linear (0.03) |
| 218 | Suppress Warnings | java | linear (0.03) |
| 219 | At Least One Constructor | java | linear (0.03) |
| 220 | Avoid Final Local Variable | java | linear (0.125) |
| 221 | Avoid Instanceof Checks In Catch Clause | java | linear (0.125) |
| 222 | Empty If Stmt | java | linear (0.03) |
| 223 | Excessive Public Count | java | linear (0.125) |
| 224 | Idempotent Operations | java | linear (0.03) |
| 225 | Missing Static Method In Non Instantiatable Class | java | linear (0.125) |
| 226 | Ncss Method Count | java | linear (0.375) |
| 227 | Ncss Type Count | java | linear (0.375) |
| 228 | Simplify Conditional | java | linear (0.125) |
| 229 | Too many methods | java | linear (0.375) |
| 230 | Unnecessary Wrapper Object Creation | java | linear (0.125) |
| 231 | Unnecessary constructor | java | linear (0.03) |
| 232 | Unused Private Field | java | linear (0.03) |
| 233 | Unused local variable | java | linear (0.03) |
| 234 | Unused private method | java | linear (0.125) |
| 235 | Bad practice - Class defines clone() but doesn't implement Cloneable | java | linear (0.125) |
| 236 | Bad practice - Class names shouldn't shadow simple name of implemented interface | java | linear (0.03) |
| 237 | Bad practice - Class names shouldn't shadow simple name of superclass | java | linear (0.03) |
| 238 | Bad practice - Finalizer nulls fields | java | linear (0.03) |
| 239 | Bad practice - Finalizer only nulls fields | java | linear (0.03) |
| 240 | Bad practice - Method doesn't override method in superclass due to wrong package for parameter | java | linear (0.125) |
| 241 | Bad practice - Unchecked type in generic call | java | linear (0.125) |
| 242 | Bad practice - Very confusing method names (but perhaps intentional) | java | linear (0.125) |
| 243 | Correctness - A known null value is checked to see if it is an instance of a type | java | linear (0.03) |
| 244 | Correctness - Covariant equals() method defined, Object.equals(Object) inherited | java | linear (0.125) |
| 245 | Correctness - Covariant equals() method defined for enum | java | linear (0.125) |
| 246 | Correctness - Dead store of class literal | java | linear (0.125) |
| 247 | Correctness - Double assignment of field | java | linear (0.03) |
| 248 | Correctness - Can't use reflection to check for presence of annotation without runtime retention | java | linear (0.125) |
| 249 | Correctness - Method call passes null for nonnull parameter | java | linear (0.03) |
| 250 | Correctness - Method call passes null to a nonnull parameter | java | linear (0.03) |
| 251 | Correctness - Uncallable method defined in anonymous class | java | linear (0.125) |
| 252 | Correctness - Useless assignment in return statement | java | linear (0.03) |
| 253 | Correctness - Useless control flow to next line | java | linear (0.03) |
| 254 | Dodgy - Dead store of null to local variable | java | linear (0.125) |
| 255 | Dodgy - Load of known null value | java | linear (0.125) |
| 256 | AbstractTypesShouldNotHavePublicConstructorsRule | cs | linear (0.125) |
| 257 | PreferXmlAbstractionsRule | cs | linear (0.125) |
| 258 | AvoidUnusedParametersRule | cs | linear (0.03) |
| 259 | AvoidUnusedPrivateFieldsRule | cs | linear (0.03) |
| 260 | AvoidUncalledPrivateCodeRule | cs | linear (0.125) |
| 261 | ConsiderConvertingMethodToPropertyRule | cs | linear (0.125) |

| | | | |
|---|---|---|---|
| 262 | DoNotUseReservedInEnumValueNamesRule | cs | linear (0.03) |
| 263 | RemoveUnusedLocalVariablesRule | cs | linear (0.03) |
| 264 | UseIsOperatorRule | cs | linear (0.03) |
| 265 | AvoidMethodWithUnusedGenericTypeRule | cs | linear (0.125) |
| 266 | Documentation text must not be empty | cs | linear (0.03) |
| 267 | Destructor summary documentation must begin with standard text | cs | linear (0.03) |
| 268 | Do not prefix calls with base unless local implementation exists | cs | linear (0.01) |
| 269 | Documentation must contain valid xml | cs | linear (0.03) |
| 270 | Documentation must meet character percentage | cs | linear (0.03) |
| 271 | Documentation text must contain whitespace | cs | linear (0.03) |
| 272 | Documentation text must meet minimum character length | cs | linear (0.03) |
| 273 | Element documentation must have summary | cs | linear (0.03) |
| 274 | Element documentation must have summary text | cs | linear (0.03) |
| 275 | Element documentation must not be copied and pasted | cs | linear (0.03) |
| 276 | Element documentation must not have default summary | cs | linear (0.03) |
| 277 | File may only contain a single namespace | cs | constant_resource (0.03) |
| 278 | File must have header | cs | constant_resource (0.03) |
| 279 | Comments must contain text | cs | linear (0.03) |
| 280 | Interface names must begin with i | cs | linear (0.01) |
| 281 | Element return value must be documented | cs | linear (0.03) |
| 282 | Elements must be documented | cs | linear (0.01) |
| 283 | Prefix local calls with this | cs | linear (0.03) |
| 284 | Constants must appear before fields | cs | linear (0.01) |
| 285 | Elements must appear in the correct order | cs | linear (0.01) |
| 286 | Elements must be ordered by access | cs | linear (0.01) |
| 287 | File header file name documentation must match file name | cs | linear (0.01) |
| 288 | Field names must begin with lower case letter | cs | linear (0.01) |
| 289 | Element parameter documentation must declare parameter name | cs | linear (0.01) |
| 290 | Generic type parameter documentation must declare parameter name | cs | linear (0.01) |
| 291 | Using directives must be placed within namespace | cs | linear (0.01) |
| 292 | Non private readonly fields must begin with upper case letter | cs | linear (0.01) |
| 293 | Code analysis suppression must have justification | cs | linear (0.03) |
| 294 | Code must not contain empty statements | cs | linear (0.03) |
| 295 | Debug assert must provide message text | cs | linear (0.03) |
| 296 | Debug fail must provide message text | cs | linear (0.03) |
| 297 | Element parameter documentation must have text | cs | linear (0.03) |
| 298 | Generic type parameter documentation must match type parameters | cs | linear (0.01) |
| 299 | Partial elements must be documented | cs | linear (0.03) |
| 300 | Element parameters must be documented | cs | linear (0.03) |
| 301 | Enumeration items must be documented | cs | linear (0.03) |
| 302 | Generic type parameters must be documented | cs | linear (0.03) |
| 303 | Generic type parameters must be documented partial class | cs | linear (0.03) |
| 304 | Element return value documentation must have text | cs | linear (0.03) |
| 305 | Generic type parameter documentation must have text | cs | linear (0.03) |
| 306 | Element parameter documentation must match element parameters | cs | linear (0.01) |
| 307 | Included documentation x path does not exist | cs | linear (0.125) |
| 308 | Partial element documentation must have summary | cs | linear (0.03) |
| 309 | Partial element documentation must have summary text | cs | linear (0.125) |
| 310 | Property summary documentation must match accessors | cs | linear (0.03) |
| 311 | Variable names must not be prefixed | cs | linear (0.01) |
| 312 | AvoidCodeDuplicatedInSameClassRule | cs | linear (0.125) |
| 313 | AvoidRefAndOutParametersRule | cs | linear (0.125) |

| | | | | |
|---|---|---|---|---|
| 314 | | AvoidSmallNamespaceRule | cs | linear (0.125) |
| 315 | | AvoidTypeInterfaceInconsistencyRule | cs | linear (0.125) |
| 316 | | AvoidUninstantiatedInternalClassesRule | cs | linear (0.125) |
| 317 | | DoNotExposeNestedGenericSignaturesRule | cs | linear (0.125) |
| 318 | | ImplementICloneableCorrectlyRule | cs | linear (0.125) |
| 319 | | ProvideTryParseAlternativeRule | cs | linear (0.125) |
| 320 | | UsePluralNameInEnumFlagsRule | cs | linear (0.03) |
| 321 | | UsePreferredTermsRule | cs | linear (0.01) |
| 322 | | AttributeStringLiteralsShouldParseCorrectlyRule | cs | linear (0.03) |
| 323 | | AvoidDeepInheritanceTreeRule | cs | linear (0.375) |
| 324 | | AvoidDeepNamespaceHierarchyRule | cs | linear (0.125) |
| 325 | | AvoidMultidimensionalIndexerRule | cs | linear (0.03) |
| 326 | | AvoidPropertiesWithoutGetAccessorRule | cs | linear (0.03) |
| 327 | | DeclareEventHandlersCorrectlyRule | cs | linear (0.125) |
| 328 | | InternalNamespacesShouldNotExposeTypesRule | cs | linear (0.03) |
| 329 | | MainShouldNotBePublicRule | cs | linear (0.03) |
| 330 | | MissingAttributeUsageOnCustomAttributeRule | cs | linear (0.03) |
| 331 | | OnlyUseDisposeForIDisposableTypesRule | cs | linear (0.03) |
| 332 | | ParameterNamesShouldMatchOverriddenMethodRule | cs | linear (0.01) |
| 333 | | PreferIntegerOrStringForIndexersRule | cs | linear (0.125) |
| 334 | | PreferStringIsNullOrEmptyRule | cs | linear (0.01) |
| 335 | | ReviewUselessControlFlowRule | cs | linear (0.03) |
| 336 | | Using alias directives must be placed after other using directives | cs | linear (0.03) |
| 337 | | System using directives must be placed before other using directives | cs | linear (0.03) |
| 338 | | Split parameters must start on line after declaration | cs | linear (0.03) |
| 339 | | Query clause must follow previous clause | cs | linear (0.03) |
| 340 | | Query clauses must be on separate lines or all on one line | cs | linear (0.03) |
| 341 | | Query clauses spanning multiple lines must begin on own line | cs | linear (0.03) |
| 342 | | Query clause must begin on new line when previous clause spans multiple lines | cs | linear (0.03) |
| 343 | | Property accessors must follow order | cs | linear (0.03) |
| 344 | | Code must not contain space after new keyword in implicitly typed array allocation | cs | linear (0.03) |
| 345 | | Conditional expressions must declare precedence | cs | linear (0.03) |
| 346 | | Declaration keywords must follow order | cs | linear (0.03) |
| 347 | | Curly brackets must not be omitted | cs | linear (0.03) |
| 348 | | Constructor summary documentation must begin with standard text | cs | linear (0.03) |
| 349 | | Event accessors must follow order | cs | linear (0.03) |
| 350 | | Property summary documentation must omit set accessor with restricted access | cs | linear (0.03) |
| 351 | | Use built in type alias | cs | linear (0.03) |
| 352 | | Access modifier must be declared | cs | linear (0.03) |
| 353 | | Arithmetic expressions must declare precedence | cs | linear (0.03) |
| 354 | | AvoidUnnecessarySpecializationRule | cs | linear (0.03) |
| 355 | | ProvideAlternativeNamesForOperatorOverloadsRule | cs | linear (0.03) |
| 356 | | RemoveDependenceOnObsoleteCodeRule | cs | linear (0.125) |
| 357 | | AvoidDeclaringCustomDelegatesRule | cs | linear (0.125) |
| 358 | **Changeability** | Visibility Modifier | java | linear (0.125) |
| 359 | **Architecture related** | Loose coupling | java | linear (0.125) |
| 360 | | Useless Overriding Method | java | linear (0.03) |
| 361 | | Coupling between objects | java | linear (1) |
| 362 | | Class Data Abstraction Coupling | java | linear (1) |
| 363 | | Class Fan Out Complexity | java | linear (1) |
| 364 | | Design For Extension | java | linear (0.125) |

| | | | |
|---|---|---|---|
| 365 | Nested Try Depth | java | linear (0.03) |
| 366 | Abstract class without any methods | java | linear (0.03) |
| 367 | Coupling - excessive imports | java | linear (1) |
| 368 | Too Many Fields | java | linear (0.125) |
| 369 | Avoid Protected Field In Final Class | java | linear (0.125) |
| 370 | Default Package | java | linear (0.125) |
| 371 | Use Array List Instead Of Vector | java | linear (0.125) |
| 372 | Bad practice - Fields of immutable classes should be final | java | linear (0.03) |
| 373 | Bad practice - Superclass uses subclass during initialization | java | linear (0.125) |
| 374 | Correctness - Class defines field that masks a superclass field | java | linear (0.125) |
| 375 | Dodgy - Ambiguous invocation of either an inherited or outer method | java | linear (0.125) |
| 376 | DoNotDeclareProtectedMembersInSealedTypeRule | cs | linear (0.125) |
| 377 | DoNotDeclareVirtualMethodsInSealedTypeRule | cs | linear (0.125) |
| 378 | Do not declare visible instance fields | cs | linear (0.125) |
| 379 | Members should not expose certain concrete types | cs | linear (0.03) |
| 380 | AvoidSpeculativeGeneralityRule | cs | linear (0.375) |
| 381 | TypesShouldBeInsideNamespacesRule | cs | linear (0.03) |
| 382 | ConsiderAddingInterfaceRule | cs | linear (0.03) |
| 383 | PreferGenericsOverRefObjectRule | cs | linear (0.125) |
| 384 | Avoid out parameters | cs | linear (0.03) |
| 385 | Bad practice - Class implements Cloneable but does not define or use clone method | java | linear (0.125) |
| 386 | Bad practice - Class is not derived from an Exception, even though it is named as such | java | linear (0.03) |
| 387 | Bad practice - Confusing method names | java | linear (0.03) |
| 388 | Bad practice - Empty finalizer should be deleted | java | linear (0.125) |
| 389 | Bad practice - Finalizer does nothing but call superclass finalizer | java | linear (0.03) |
| 390 | Bad practice - serialVersionUID isn't long | java | linear (0.03) |
| 391 | Bad practice - serialVersionUID isn't static | java | linear (0.03) |
| 392 | Boolean Expression Complexity | java | linear (0.03) |
| 393 | Correctness - Call to equals() with null argument | java | linear (0.125) |
| 394 | Correctness - Field only ever set to null | java | linear (0.125) |
| 395 | Correctness - Method call passes null for nonnull parameter | java | linear (0.03) |
| 396 | Correctness - Nullcheck of value previously dereferenced | java | linear (0.125) |
| 397 | Correctness - Unnecessary type check done using instanceof operator | java | linear (0.03) |
| 398 | Correctness - Unneeded use of currentThread() call, to call interrupted() | java | linear (0.03) |
| 399 | Correctness - Unwritten field | java | linear (0.03) |
| 400 | Dodgy - Class implements same interface as superclass | java | linear (0.03) |
| 401 | Dodgy - Class is final but declares protected field | java | linear (0.03) |
| 402 | Dodgy - Dead store to local variable | java | linear (0.125) |
| 403 | Dodgy - Exception is caught when Exception is not thrown | java | linear (0.125) |
| 404 | Dodgy - Method checks to see if result of String.indexOf is positive | java | linear (0.125) |
| 405 | Dodgy - Redundant comparison of non-null value to null | java | linear (0.03) |
| 406 | Dodgy - Redundant comparison of two null values | java | linear (0.03) |
| 407 | Dodgy - Redundant nullcheck of value known to be non-null | java | linear (0.03) |
| 408 | Dodgy - Redundant nullcheck of value known to be null | java | linear (0.03) |
| 409 | Dodgy - Unchecked/unconfirmed cast | java | linear (0.03) |
| 410 | Executable Statement Count | java | linear (0.125) |
| 411 | File Length | java | linear (0.375) |
| 412 | Final Class | java | linear (0.01) |
| 413 | Javadoc Type | java | linear (0.01) |
| 414 | Javadoc Style | java | linear (0.01) |
| 415 | Javadoc Package | java | linear (0.01) |

| | | | |
|---|---|---|---|
| 416 | Javadoc Method | java | linear (0.01) |
| 417 | Hide Utility Class Constructor | java | linear (0.03) |
| 418 | Unnecessary Parentheses | java | linear (0.03) |
| 419 | Simplify Boolean Return | java | linear (0.125) |
| 420 | Redundant Throws | java | linear (0.125) |
| 421 | Simplify Boolean Expression | java | linear (0.125) |
| 422 | Performance - Unread field: should this field be static? | java | linear (0.03) |
| 423 | Performance - Unused field | java | linear (0.03) |
| 424 | Performance - Unread field | java | linear (0.03) |
| 425 | Performance - Private method is never called | java | linear (0.03) |
| 426 | Package Declaration | java | linear (0.03) |
| 427 | Method Length | java | linear (0.125) |
| 428 | Javadoc Variable | java | linear (0.01) |
| 429 | Bad practice - serialVersionUID isn't final | java | linear (0.03) |
| 430 | Bad practice - Needless instantiation of class that only supplies static methods | java | linear (0.03) |
| 431 | Abstract types should not have constructors (FxCop10) | cs | linear (0.01) |
| 432 | Assemblies should have valid strong names | cs | linear (0.03) |
| 433 | Attribute string literals should parse correctly (FxCop10) | cs | linear (0.03) |
| 434 | Avoid excessive locals | cs | linear (0.03) |
| 435 | Avoid excessive parameters on generic types | cs | linear (0.03) |
| 436 | Avoid namespaces with few types | cs | linear (0.03) |
| 437 | Avoid non-public fields in COM visible value types (FxCop10) | cs | linear (0.03) |
| 438 | Avoid uncalled private code (FxCop10) | cs | linear (0.125) |
| 439 | Avoid unused private fields (FxCop10) | cs | linear (0.03) |
| 440 | Collections should implement generic interface | cs | linear (0.03) |
| 441 | Do not declare protected members in sealed types | cs | linear (0.03) |
| 442 | Do not declare static members on generic types | cs | linear (0.03) |
| 443 | Do not declare virtual members in sealed types | cs | linear (0.03) |
| 444 | Do not hide base class methods | cs | linear (0.03) |
| 445 | Do not name enum values 'Reserved' | cs | linear (0.03) |
| 446 | Do not nest generic types in member signatures | cs | linear (0.375) |
| 447 | Do not pass types by reference | cs | linear (0.125) |
| 448 | Do not prefix enum values with type name | cs | linear (0.01) |
| 449 | Enumerators should be strongly typed | cs | linear (0.03) |
| 450 | Flags enums should have plural names | cs | linear (0.03) |
| 451 | Generic methods should provide type parameter | cs | linear (0.03) |
| 452 | Identifiers should differ by more than case | cs | linear (0.01) |
| 453 | Identifiers should have correct prefix | cs | linear (0.01) |
| 454 | Identifiers should have correct suffix | cs | linear (0.01) |
| 455 | Identifiers should not contain type names | cs | linear (0.01) |
| 456 | Identifiers should not have incorrect prefix | cs | linear (0.01) |
| 457 | Identifiers should not have incorrect suffix | cs | linear (0.01) |
| 458 | Identifiers should not match keywords | cs | linear (0.01) |
| 459 | Indexers should not be multidimensional | cs | linear (0.125) |
| 460 | Mark all non-serializable fields (FxCop10) | cs | linear (0.03) |
| 461 | Mark attributes with AttributeUsageAttribute (FxCop10) | cs | linear (0.125) |
| 462 | Mark boolean P/Invoke arguments with MarshalAs (FxCop10) | cs | linear (0.03) |
| 463 | Members should differ by more than return type (FxCop10) | cs | linear (0.125) |
| 464 | Nested types should not be visible | cs | linear (0.03) |
| 465 | Normalize strings to uppercase | cs | linear (0.01) |
| 466 | Only FlagsAttribute enums should have plural names | cs | linear (0.03) |
| 467 | Parameter names should match base declaration | cs | linear (0.03) |
| 468 | Parameter names should not match member names | cs | linear (0.03) |
| 469 | Property names should not match get methods | cs | linear (0.03) |

| | | | | |
|---|---|---|---|---|
| 470 | | Provide ObsoleteAttribute message | cs | linear (0.125) |
| 471 | | Remove empty finalizers | cs | linear (0.125) |
| 472 | | Remove unused locals (FxCop10) | cs | linear (0.03) |
| 473 | | Review unused parameters (FxCop10) | cs | linear (0.125) |
| 474 | | Use properties where appropriate | cs | linear (0.125) |
| 475 | | Use params for variable arguments (FxCop10) | cs | linear (0.03) |
| 476 | | Use integral or string argument for indexers | cs | linear (0.125) |
| 477 | | Use generics where appropriate | cs | linear (0.375) |
| 478 | | Use events where appropriate | cs | linear (1) |
| 479 | | Types should not extend certain base types | cs | linear (0.03) |
| 480 | | Type names should not match namespaces | cs | linear (0.03) |
| 481 | | Static holder types should not have constructors | cs | linear (0.03) |
| 482 | | Specify StringComparison | cs | linear (0.03) |
| 483 | | Magic Number | java | linear (0.03) |
| 484 | **Data related** | Interface Is Type | java | linear (0.03) |
| 485 | | Multiple String Literals | java | linear (0.125) |
| 486 | | Avoid Constants Interface | java | linear (0.125) |
| 487 | | Avoid empty interfaces | cs | linear (0.125) |
| 488 | | AvoidEmptyInterfaceRule | cs | linear (0.125) |
| 489 | | AvoidVisibleConstantFieldRule | cs | linear (0.03) |
| 490 | | AvoidMessageChainsRule | cs | linear (0.125) |
| 491 | | Need Braces | java | linear (0.03) |
| 492 | | Default Comes Last | java | linear (0.03) |
| 493 | **Logic related** | Nested If Depth | java | linear (0.125) |
| 494 | | Return Count | java | linear (0.125) |
| 495 | | Strict Duplicate Code | java | linear (0.375) |
| 496 | | Throws Count | java | linear (0.125) |
| 497 | | Avoid Deeply Nested If Stmts | java | linear (0.125) |
| 498 | | Avoid Duplicate Literals | java | linear (0.125) |
| 499 | | If Stmts Must Use Braces | java | linear (0.03) |
| 500 | | If Else Stmts Must Use Braces | java | linear (0.03) |
| 501 | | Only One Return | java | linear (0.125) |
| 502 | | Replace Hashtable With Map | java | linear (0.03) |
| 503 | | Replace Vector With List | java | linear (0.03) |
| 504 | | Switch Density | java | linear (0.125) |
| 505 | | AvoidLackOfCohesionOfMethodsRule | cs | linear (0.375) |
| 506 | **Architecture related** | Avoid Star Import | java | constant_resource (0.01) |
| 507 | | Illegal Import | java | linear (0.125) |
| 508 | | Avoid Static Import | java | linear (0.03) |
| 509 | | Covariant Equals | java | linear (0.125) |
| 510 | | Missing Override | java | linear (0.01) |
| 511 | | Avoid Calling Finalize | java | linear (0.125) |
| 512 | | Avoid StringBuffer field | java | linear (0.125) |
| 513 | | Bean Members Should Serialize | java | linear (0.03) |
| 514 | | Constructor Calls Overridable Method | java | linear (0.375) |
| 515 | | Empty Method In Abstract Class Should Be Abstract | java | linear (0.125) |
| 516 | | Missing Serial Version UID | java | linear (0.03) |
| 517 | | Bad practice - Class is Externalizable but doesn't define a void constructor | java | linear (0.125) |
| 518 | | Bad practice - Class is Serializable but its superclass doesn't define a void constructor | java | linear (0.125) |
| 519 | | Bad practice - Non-serializable class has a serializable inner class | java | linear (0.125) |
| 520 | | Bad practice - Non-serializable value stored into instance field of a serializable class | java | linear (0.125) |

| | | | |
|---|---|---|---|
| 521 | Bad practice - Serializable inner class | java | linear (0.125) |
| 522 | Bad practice - Store of non serializable object into HttpSession | java | linear (0.125) |
| 523 | Bad practice - Usage of GetResource may be unsafe if class is extended | java | linear (0.125) |
| 524 | Bad practice - clone method does not call super.clone() | java | linear (0.125) |
| 525 | Bad practice - equals method fails for subtypes | java | linear (0.125) |
| 526 | Class defines equal(Object); should it be equals(Object)? | java | linear (0.125) |
| 527 | Class defines hashcode(); should it be hashCode()? | java | linear (0.125) |
| 528 | Class defines tostring(); should it be toString()? | java | linear (0.125) |
| 529 | Correctness - Apparent method/constructor confusion | java | linear (0.125) |
| 530 | Correctness - Call to equals() comparing unrelated class and interface | java | linear (0.125) |
| 531 | Correctness - Call to equals() comparing different types | java | linear (0.125) |
| 532 | Correctness - Call to equals() comparing different interface types | java | linear (0.125) |
| 533 | Correctness - Class overrides a method implemented in super class Adapter wrongly | java | linear (0.125) |
| 534 | Correctness - Creation of ScheduledThreadPoolExecutor with zero core threads | java | linear (0.125) |
| 535 | Correctness - Deadly embrace of non-static inner class and thread local | java | linear (0.125) |
| 536 | Correctness - Method doesn't override method in superclass due to wrong package for parameter | java | linear (0.125) |
| 537 | Correctness - Method must be private in order for serialization to work | java | linear (0.125) |
| 538 | Correctness - No relationship between generic parameter and method argument | java | linear (0.125) |
| 539 | Correctness - The readResolve method must not be declared as a static method. | java | linear (0.03) |
| 540 | Correctness - Use of class without a hashCode() method in a hashed data structure | java | linear (0.125) |
| 541 | Correctness - equals() method defined that doesn't override Object.equals(Object) | java | linear (0.125) |
| 542 | Correctness - equals() method defined that doesn't override equals(Object) | java | linear (0.125) |
| 543 | Dodgy - Class doesn't override equals in superclass | java | linear (0.125) |
| 544 | Dodgy - Initialization circularity | java | linear (0.125) |
| 545 | Dodgy - Thread passed where Runnable expected | java | linear (0.125) |
| 546 | UseCorrectSignatureForSerializationMethodsRule | cs | linear (0.125) |
| 547 | MissingSerializationConstructorRule | cs | linear (0.125) |
| 548 | MissingSerializableAttributeOnISerializableTypeRule | cs | linear (0.125) |
| 549 | MarkEnumerationsAsSerializableRule | cs | linear (0.125) |
| 550 | MarkAllNonSerializableFieldsRule | cs | linear (0.125) |
| 551 | ImplementISerializableCorrectlyRule | cs | linear (0.125) |
| 552 | DeserializeOptionalFieldRule | cs | linear (0.125) |
| 553 | CallBaseMethodsOnISerializableTypesRule | cs | linear (0.125) |
| 554 | DoNotForgetNotImplementedMethodsRule | cs | linear (0.125) |
| 555 | Do not call overridable methods in constructors (FxCop10) | cs | linear (0.375) |
| 556 | Avoid static members in COM visible types (FxCop10) | cs | linear (0.375) |
| 557 | Collection properties should be read only (FxCop10) | cs | linear (0.125) |
| 558 | Do not decrease inherited member visibility (FxCop10) | cs | linear (0.03) |
| 559 | Do not expose generic lists | cs | linear (0.125) |
| 560 | Do not overload operator equals on reference types | cs | linear (0.03) |
| 561 | Finalizers should call base class finalizer (FxCop10) | cs | linear (0.125) |
| 562 | Do not mark enums with FlagsAttribute (FxCop10) | cs | linear (0.03) |
| 563 | Do not mark serviced components with WebMethod (FxCop10) | cs | linear (0.125) |
| 564 | Implement IDisposable correctly | cs | linear (0.125) |
| 565 | Non-constant fields should not be visible (FxCop10) | cs | linear (0.375) |
| 566 | Static holder types should be sealed | cs | linear (0.03) |

| | | | | |
|---|---|---|---|---|
| 567 | | ConsiderUsingStopwatchRule | cs | linear (0.03) |
| 568 | | ConstructorShouldNotCallVirtualMethodsRule | cs | linear (0.125) |
| 569 | | EnsureSymmetryForOverloadedOperatorsRule | cs | linear (0.125) |
| 570 | | Do not ship unreleased resource formats (FxCop10) | cs | linear (0.125) |
| 571 | | Implement ISerializable correctly (FxCop10) | cs | linear (0.125) |
| 572 | | Implement serialization methods correctly (FxCop10) | cs | linear (0.125) |
| 573 | | Mark Windows Forms entry points with STAThread (FxCop10) | cs | linear (0.125) |
| 574 | | Avoid duplicate accelerators | cs | linear (0.03) |
| 575 | | Avoid overloads in COM visible interfaces (FxCop10) | cs | linear (0.125) |
| 576 | | COM registration methods should not be visible (FxCop10) | cs | linear (0.03) |
| 577 | | COM visible type base types should be COM visible (FxCop10) | cs | linear (0.125) |
| 578 | | COM visible types should be creatable (FxCop10) | cs | linear (0.03) |
| 579 | | Consider passing base types as parameters | cs | linear (0.03) |
| 580 | | Declare types in namespaces | cs | linear (0.03) |
| 581 | | Mark enums with FlagsAttribute | cs | linear (0.03) |
| 582 | | Fields must be private | cs | linear (0.03) |
| 583 | | ConsiderUsingStaticTypeRule | cs | linear (0.125) |
| 584 | | ImplementGenericCollectionInterfacesRule | cs | linear (0.125) |
| 585 | | ImplementIComparableCorrectlyRule | cs | linear (0.125) |
| 586 | | MarkAssemblyWithCLSCompliantRule | cs | linear (0.125) |
| 587 | | MarkAssemblyWithComVisibleRule | cs | linear (0.125) |
| 588 | | MethodCanBeMadeStaticRule | cs | linear (0.125) |
| 589 | | PreferSafeHandleRule | cs | linear (0.125) |
| 590 | | StaticConstructorsShouldBePrivateRule | cs | linear (0.125) |
| 591 | | UseGenericEventHandlerRule | cs | linear (0.125) |
| 592 | | ICollection implementations have strongly typed members | cs | linear (0.125) |
| 593 | | Operators should have symmetrical overloads (FxCop10) | cs | linear (0.125) |
| 594 | | AttributeArgumentsShouldHaveAccessorsRule | cs | linear (0.03) |
| 595 | | AvoidExtensionMethodOnSystemObjectRule | cs | linear (0.125) |
| 596 | | ProvideCorrectRegexPatternRule | cs | linear (0.375) |
| 597 | | ReviewSelfAssignmentRule | cs | linear (0.03) |
| 598 | | Provide deserialization methods for optional fields (FxCop10) | cs | linear (0.375) |
| 599 | | Types that own disposable fields should be disposable | cs | linear (0.125) |
| 600 | | Types that own native resources should be disposable | cs | linear (0.125) |
| 601 | | Use generic event handler instances | cs | linear (0.125) |
| 602 | | Specify IFormatProvider | cs | linear (0.125) |
| 603 | | Mark ISerializable types with SerializableAttribute (FxCop10) | cs | linear (0.125) |
| 604 | | Mark ComSource interfaces as IDispatch (FxCop10) | cs | linear (0.125) |
| 605 | | FinalizersShouldCallBaseClassFinalizerRule | cs | linear (0.125) |
| 606 | | MarshalBooleansInPInvokeDeclarationsRule | cs | linear (0.125) |
| 607 | | MarshalStringsInPInvokeDeclarationsRule | cs | linear (0.125) |
| 608 | | UseNoInliningWithGetCallingAssemblyRule | cs | linear (0.03) |
| 609 | | GtkSharpExecutableTargetRule | cs | linear (0.03) |
| 610 | **Data related** | Hidden Field | java | linear (0.03) |
| 611 | | Illegal Instantiation | java | linear (0.03) |
| 612 | | Useless Operation On Immutable | java | linear (0.15) |
| 613 | | Explicit Initialization | java | linear (0.03) |
| 614 | | Final Local Variable | java | linear (0.03) |
| 615 | | Final Parameters | java | linear (0.03) |
| 616 | | Missing Constructor | java | linear (0.03) |
| 617 | | Parameter Assignment | java | linear (0.03) |
| 618 | | Assignment To Non Final Static | java | linear (0.03) |
| 619 | | Avoid Decimal Literals In Big Decimal Constructor | java | linear (0.03) |
| 620 | | Avoid Reassigning Parameters | java | linear (0.125) |

| | | | |
|---|---|---|---|
| 621 | Method Argument Could Be Final | java | linear (0.03) |
| 622 | Null Assignment | java | linear (0.03) |
| 623 | Static EJB Field Should Be Final | java | linear (0.03) |
| 624 | Suspicious Octal Escape | java | linear (0.03) |
| 625 | Multithreaded correctness - A volatile reference to an array doesn't treat the array elements as volatile | java | linear (0.125) |
| 626 | Non-transient non-serializable instance field in serializable class | java | linear (0.125) |
| 627 | Bad practice - Equals method should not assume anything about the type of its argument | java | linear (0.125) |
| 628 | Bad practice - Explicit invocation of finalizer | java | linear (0.125) |
| 629 | Bad practice - Static initializer creates instance before all static final fields assigned | java | linear (0.125) |
| 630 | Bad practice - Transient field that isn't set by deserialization. | java | linear (0.125) |
| 631 | Correctness - "." used for regular expression | java | linear (0.125) |
| 632 | Correctness - Field not initialized in constructor | java | linear (0.125) |
| 633 | Correctness - Method defines a variable that obscures a field | java | linear (0.03) |
| 634 | Correctness - Method does not check for null argument | java | linear (0.125) |
| 635 | Correctness - Method may return null, but is declared @NonNull | java | linear (0.125) |
| 636 | Correctness - Method performs math using floating point precision | java | linear (0.125) |
| 637 | Correctness - Non-virtual method call passes null for nonnull parameter | java | linear (0.125) |
| 638 | Correctness - Store of null value into field annotated NonNull | java | linear (0.125) |
| 639 | Correctness - Uninitialized read of field in constructor | java | linear (0.03) |
| 640 | Correctness - Uninitialized read of field method called from constructor of superclass | java | linear (0.03) |
| 641 | Dodgy - Class extends Servlet class and uses instance variables | java | linear (0.125) |
| 642 | Dodgy - Class extends Struts Action class and uses instance variables | java | linear (0.125) |
| 643 | Dodgy - Parameter must be nonnull but is marked as nullable | java | linear (0.125) |
| 644 | DoNotRecurseInEqualityRule | cs | linear (0.125) |
| 645 | AvoidVisibleFieldsRule | cs | linear (0.03) |
| 646 | AvoidVisibleNestedTypesRule | cs | linear (0.03) |
| 647 | AvoidFloatingPointEqualityRule | cs | linear (0.03) |
| 648 | DoNotCompareWithNaNRule | cs | linear (0.125) |
| 649 | Do not hardcode locale specific strings | cs | linear (0.125) |
| 650 | Enums should have zero value | cs | linear (0.03) |
| 651 | Test for NaN correctly (FxCop10) | cs | linear (0.03) |
| 652 | Specify CultureInfo (FxCop10) | cs | linear (0.125) |
| 653 | Set locale for data types | cs | linear (0.125) |
| 654 | Mark assemblies with AssemblyVersionAttribute (FxCop10) | cs | linear (0.03) |
| 655 | AvoidAssemblyVersionMismatchRule | cs | linear (0.125) |
| 656 | EnumsShouldDefineAZeroValueRule | cs | linear (0.03) |
| 657 | EnumsShouldUseInt32Rule | cs | linear (0.125) |
| 658 | FlagsShouldNotDefineAZeroValueRule | cs | linear (0.03) |
| 659 | Define accessors for attribute arguments | cs | linear (0.03) |
| 660 | Do not ignore method results (FxCop10) | cs | linear (0.125) |
| 661 | Interface methods should be callable by child types | cs | linear (0.125) |
| 662 | Avoid Int64 arguments for Visual Basic 6 clients (FxCop10) | cs | linear (0.03) |
| 663 | PreferParamsArrayForVariableArgumentsRule | cs | linear (0.125) |
| 664 | Include node does not contain valid file and path | cs | linear (0.125) |
| 665 | DoNotRoundIntegersRule | cs | linear (0.03) |
| 666 | MarkAssemblyWithAssemblyVersionRule | cs | linear (0.125) |
| 667 | Operations should not overflow (FxCop10) | cs | linear (0.03) |
| 668 | Override methods on comparable types | cs | linear (0.125) |
| 669 | AvoidCodeDuplicatedInSiblingClassesRule | cs | linear (0.125) |

| | | | | |
|---|---|---|---|---|
| 670 | | AvoidReturningArraysOnPropertiesRule | cs | linear (0.125) |
| 671 | | ProvideCorrectArgumentsToFormattingMethodsRule | cs | linear (0.125) |
| 672 | | ReviewDoubleAssignmentRule | cs | linear (0.125) |
| 673 | | ReviewInconsistentIdentityRule | cs | linear (0.375) |
| 674 | | ReviewUseOfInt64BitsToDoubleRule | cs | linear (0.125) |
| 675 | | Use managed equivalents of Win32 API (FxCop10) | cs | linear (0.125) |
| 676 | | UseCorrectDisposeSignaturesRule | cs | linear (0.125) |
| 677 | **Exception handling** | Mutable Exception | java | linear (0.03) |
| 678 | | Avoid Catching NPE | java | linear (0.125) |
| 679 | | Avoid Catching Throwable | java | linear (0.125) |
| 680 | | Avoid Rethrowing Exception | java | linear (0.125) |
| 681 | | Strict Exception - Avoid throwing new instance of same exception | java | linear (0.03) |
| 682 | | Strict Exception - Do not throw exception in finally | java | linear (0.03) |
| 683 | | Use Correct Exception Logging | java | linear (0.03) |
| 684 | | Avoid Print Stack Trace | java | linear (0.03) |
| 685 | | Avoid Throwing Null Pointer Exception | java | linear (0.125) |
| 686 | | Avoid Throwing Raw Exception Types | java | linear (0.125) |
| 687 | | Bad practice - Method may fail to close stream on exception | java | linear (0.125) |
| 688 | | Bad practice - Method might ignore exception | java | linear (0.125) |
| 689 | | Correctness - Exception created and dropped rather than thrown | java | linear (0.125) |
| 690 | | AvoidArgumentExceptionDefaultConstructorRule | cs | linear (0.125) |
| 691 | | MissingExceptionConstructorsRule | cs | linear (0.125) |
| 692 | | InstantiateArgumentExceptionCorrectlyRule | cs | linear (0.125) |
| 693 | | ExceptionShouldBeVisibleRule | cs | linear (0.125) |
| 694 | | DoNotThrowReservedExceptionRule | cs | linear (0.125) |
| 695 | | DoNotSwallowErrorsCatchingNonSpecificExceptionsRule | cs | linear (0.125) |
| 696 | | DelegatesPassedToNativeCodeMustIncludeExceptionHandlingRule | cs | linear (0.125) |
| 697 | | CheckNewExceptionWithoutThrowingRule | cs | linear (0.125) |
| 698 | | AvoidThrowingBasicExceptionsRule | cs | linear (0.125) |
| 699 | | DoNotDestroyStackTraceRule | cs | linear (0.125) |
| 700 | | DoNotThrowInUnexpectedLocationRule | cs | linear (0.125) |
| 701 | | DoNotUseLockedRegionOutsideMethodRule | cs | linear (0.125) |
| 702 | **Fault tolerance** | Empty Catch Block | java | linear (0.125) |
| 703 | | Misplaced Null Check | java | linear (0.03) |
| 704 | | Equals Avoid Null | java | linear (0.03) |
| 705 | | Do Not Extend Java Lang Error | java | linear (0.125) |
| 706 | | PreferEmptyInstanceOverNullRule | cs | linear (0.03) |
| 707 | | GetLastErrorMustBeCalledRightAfterPInvokeRule | cs | linear (0.03) |
| 708 | **Instruction related** | Typecast Paren Pad | java | constant_resource (0.01) |
| 709 | | Equals Hash Code | java | linear (0.125) |
| 710 | | Inner Assignment | java | linear (0.03) |
| 711 | | Class Cast Exception With To Array | java | linear (0.03) |
| 712 | | Equals Null | java | linear (0.125) |
| 713 | | Bad Comparison | java | linear (0.03) |
| 714 | | No Clone | java | linear (0.125) |
| 715 | | String Literal Equality | java | linear (0.03) |
| 716 | | Compare Objects With Equals | java | linear (0.03) |
| 717 | | Empty Statement Not In Loop | java | linear (0.03) |
| 718 | | Finalize Does Not Call Super Finalize | java | linear (0.03) |
| 719 | | Finalize Overloaded | java | linear (0.15) |
| 720 | | Override both equals and hashcode | java | linear (0.125) |
| 721 | | Proper clone implementation | java | linear (0.15) |
| 722 | | String Buffer Instantiation With Char | java | linear (0.03) |
| 723 | | Use Equals To Compare Strings | java | linear (0.03) |

| | | | |
|---|---|---|---|
| 724 | Use Proper Class Loader | java | linear (0.375) |
| 725 | Empty Statement | java | linear (0.01) |
| 726 | Empty For Iterator Pad | java | linear (0.01) |
| 727 | Empty For Initializer Pad | java | linear (0.01) |
| 728 | Multithreaded correctness - A thread was created using the default empty run method | java | linear (0.125) |
| 729 | Performance - Explicit garbage collection; extremely dubious except in benchmarking code | java | linear (0.03) |
| 730 | Bad practice - Abstract class defines covariant compareTo() method | java | linear (0.125) |
| 731 | Bad practice - Abstract class defines covariant equals() method | java | linear (0.125) |
| 732 | Bad practice - Check for sign of bitwise operation | java | linear (0.125) |
| 733 | Bad practice - Class defines compareTo(...) and uses Object.equals() | java | linear (0.125) |
| 734 | Bad practice - Class defines equals() and uses Object.hashCode() | java | linear (0.125) |
| 735 | Bad practice - Class defines equals() but not hashCode() | java | linear (0.125) |
| 736 | Bad practice - Class defines hashCode() and uses Object.equals() | java | linear (0.125) |
| 737 | Bad practice - Class defines hashCode() but not equals() | java | linear (0.125) |
| 738 | Bad practice - Class inherits equals() and uses Object.hashCode() | java | linear (0.125) |
| 739 | Bad practice - Clone method may return null | java | linear (0.125) |
| 740 | Bad practice - Comparator doesn't implement Serializable | java | linear (0.125) |
| 741 | Bad practice - Comparison of String objects using == or != | java | linear (0.125) |
| 742 | Bad practice - Comparison of String parameter using == or != | java | linear (0.125) |
| 743 | Bad practice - Equals checks for noncompatible operand | java | linear (0.125) |
| 744 | Bad practice - Method ignores results of InputStream.read() | java | linear (0.125) |
| 745 | Bad practice - Method ignores results of InputStream.skip() | java | linear (0.125) |
| 746 | Bad practice - Suspicious reference comparison | java | linear (0.125) |
| 747 | Bad practice - The readResolve method must be declared with a return type of Object. | java | linear (0.125) |
| 748 | Correctness - Bad attempt to compute absolute value of signed 32-bit hashcode | java | linear (0.125) |
| 749 | Correctness - Bad attempt to compute absolute value of signed 32-bit random integer | java | linear (0.125) |
| 750 | Correctness - Bad comparison of nonnegative value with negative constant | java | linear (0.125) |
| 751 | Correctness - Bad comparison of signed byte | java | linear (0.125) |
| 752 | Correctness - Bad constant value for month | java | linear (0.125) |
| 753 | Correctness - Collections should not contain themselves | java | linear (0.125) |
| 754 | Correctness - Don't use removeAll to clear a collection | java | linear (0.125) |
| 755 | Correctness - Doomed attempt to append to an object output stream | java | linear (0.125) |
| 756 | Correctness - Doomed test for equality to NaN | java | linear (0.125) |
| 757 | Correctness - Array formatted in useless way using format string | java | linear (0.03) |
| 758 | Correctness - Double.longBitsToDouble invoked on an int | java | linear (0.125) |
| 759 | Correctness - Format string placeholder incompatible with passed argument | java | linear (0.125) |
| 760 | Correctness - Format string references missing argument | java | linear (0.125) |
| 761 | Correctness - Illegal format string | java | linear (0.125) |
| 762 | Correctness - Impossible cast | java | linear (0.03) |
| 763 | Correctness - Impossible downcast | java | linear (0.03) |
| 764 | Correctness - Impossible downcast of toArray() result | java | linear (0.03) |
| 765 | Correctness - Incompatible bit masks | java | linear (0.125) |
| 766 | Correctness - Incompatible bit masks | java | linear (0.125) |
| 767 | Correctness - Integer multiply of result of integer remainder | java | linear (0.125) |
| 768 | Correctness - Integer remainder modulo 1 | java | linear (0.125) |
| 769 | Correctness - Integer shift by an amount not in the range 0..31 | java | linear (0.125) |
| 770 | Correctness - Invalid syntax for regular expression | java | linear (0.125) |

| | | | |
|---|---|---|---|
| 771 | Correctness - Invocation of equals() on an array, which is equivalent to == | java | linear (0.125) |
| 772 | Correctness - Invocation of hashCode on an array | java | linear (0.03) |
| 773 | Correctness - Invocation of toString on an array | java | linear (0.03) |
| 774 | Correctness - Invocation of toString on an array | java | linear (0.03) |
| 775 | Correctness - JUnit assertion in run method will not be noticed by JUnit | java | linear (0.03) |
| 776 | Correctness - MessageFormat supplied where printf style format expected | java | linear (0.03) |
| 777 | Correctness - Method assigns boolean literal in boolean expression | java | linear (0.03) |
| 778 | Correctness - Method attempts to access a prepared statement parameter with index 0 | java | linear (0.03) |
| 779 | Correctness - Method attempts to access a result set field with index 0 | java | linear (0.03) |
| 780 | Correctness - Method ignores return value | java | linear (0.125) |
| 781 | Correctness - Method ignores return value | java | linear (0.125) |
| 782 | Correctness - More arguments are passed that are actually used in the format string | java | linear (0.125) |
| 783 | Correctness - No previous argument for format string | java | linear (0.125) |
| 784 | Correctness - Overwritten increment | java | linear (0.125) |
| 785 | Correctness - Random value from 0 to 1 is coerced to the integer 0 | java | linear (0.125) |
| 786 | Correctness - Suspicious reference comparison of Boolean values | java | linear (0.125) |
| 787 | Correctness - Suspicious reference comparison to constant | java | linear (0.125) |
| 788 | Correctness - The type of a supplied argument doesn't match format specifier | java | linear (0.125) |
| 789 | Correctness - Using pointer equality to compare different types | java | linear (0.125) |
| 790 | Correctness - Vacuous call to collections | java | linear (0.125) |
| 791 | Correctness - close() invoked on a value that is always null | java | linear (0.125) |
| 792 | Correctness - equals() used to compare array and nonarray | java | linear (0.125) |
| 793 | Correctness - equals(...) used to compare incompatible arrays | java | linear (0.125) |
| 794 | Correctness - int value cast to float and then passed to Math.round | java | linear (0.125) |
| 795 | Correctness - int value cast to double and then passed to Math.ceil | java | linear (0.125) |
| 796 | Dodgy - Call to unsupported method | java | linear (0.125) |
| 797 | Dodgy - Check for oddness that won't work for negative numbers | java | linear (0.125) |
| 798 | Dodgy - Consider returning a zero length array rather than null | java | linear (0.125) |
| 799 | Dodgy - Non-Boolean argument formatted using %b format specifier | java | linear (0.125) |
| 800 | Dodgy - Non serializable object written to ObjectOutput | java | linear (0.125) |
| 801 | Dodgy - Questionable use of non-short-circuit logic | java | linear (0.125) |
| 802 | Dodgy - Questionable cast to concrete collection | java | linear (0.125) |
| 803 | Dodgy - Questionable cast to abstract collection | java | linear (0.125) |
| 804 | Dodgy - Remainder of hashCode could be negative | java | linear (0.125) |
| 805 | Dodgy - Result of integer multiplication cast to long | java | linear (0.125) |
| 806 | Dodgy - Test for floating point equality | java | linear (0.125) |
| 807 | Dodgy - int division result cast to double or float | java | linear (0.125) |
| 808 | OverrideEqualsMethodRule | cs | linear (0.125) |
| 809 | OperatorEqualsShouldBeOverloadedRule | cs | linear (0.125) |
| 810 | AvoidCallingProblematicMethodsRule | cs | linear (0.125) |
| 811 | CallingEqualsWithNullArgRule | cs | linear (0.125) |
| 812 | ImplementEqualsAndGetHashCodeInPairRule | cs | linear (0.125) |
| 813 | Initialize value type static fields inline (FxCop10) | cs | linear (0.125) |
| 814 | Operator overloads have named alternates (FxCop10) | cs | linear (0.15) |
| 815 | DoNotUseGetInterfaceToCheckAssignabilityRule | cs | linear (0.03) |
| 816 | Do not use AutoDual ClassInterfaceType (FxCop10) | cs | linear (0.125) |

| | | | | |
|---|---|---|---|---|
| 817 | | Implement serialization constructors (FxCop10) | cs | linear (0.03) |
| 818 | | Call GetLastError immediately after P/Invoke (FxCop10) | cs | linear (0.03) |
| 819 | | Call base class methods on ISerializable types (FxCop10) | cs | linear (0.03) |
| 820 | | Declare P/Invokes correctly (FxCop10) | cs | linear (0.125) |
| 821 | | Declare event handlers correctly | cs | linear (0.125) |
| 822 | | Use ordinal StringComparison | cs | linear (0.03) |
| 823 | | Overload operator equals on overloading add and subtract | cs | linear (0.125) |
| 824 | | Overload operator equals on overriding ValueType.Equals (FxCop10) | cs | linear (0.125) |
| 825 | | Override Equals on overloading operator equals (FxCop10) | cs | linear (0.125) |
| 826 | | Override GetHashCode on overriding Equals (FxCop10) | cs | linear (0.125) |
| 827 | | ProtectCallToEventDelegatesRule | cs | linear (0.03) |
| 828 | | P/Invoke entry points should exist (FxCop10) | cs | linear (0.375) |
| 829 | | ArrayFieldsShouldNotBeReadOnlyRule | cs | linear (0.03) |
| 830 | | Empty Block | java | linear (0.03) |
| 831 | | Missing Switch Default | java | linear (0.03) |
| 832 | | Modified Control Variable | java | linear (0.125) |
| 833 | **Logic related** | Jumbled Incrementer | java | linear (0.125) |
| 834 | | Fall Through | java | linear (0.125) |
| 835 | | No Finalizer | java | linear (0.125) |
| 836 | | Comment pattern matcher | java | linear (0.125) |
| 837 | | Android - call super first | java | linear (0.03) |
| 838 | | Android - call super last | java | linear (0.03) |
| 839 | | Broken Null Check | java | linear (0.03) |
| 840 | | Call Super In Constructor | java | linear (0.03) |
| 841 | | Check ResultSet | java | linear (0.03) |
| 842 | | Dataflow Anomaly Analysis | java | linear (0.375) |
| 843 | | Default label not last in switch statement | java | linear (0.125) |
| 844 | | Missing Break In Switch | java | linear (0.03) |
| 845 | | Non Case Label In Switch Statement | java | linear (0.125) |
| 846 | | Position Literals First In Comparisons | java | linear (0.03) |
| 847 | | Return From Finally Block | java | linear (0.125) |
| 848 | | Switch statements should have default | java | linear (0.125) |
| 849 | | Non Static Initializer | java | linear (0.125) |
| 850 | | Switch statement found where default case is missing | java | linear (0.125) |
| 851 | | Switch statement found where one case falls through to the next case | java | linear (0.125) |
| 852 | | Bad practice - Creates an empty jar file entry | java | linear (0.125) |
| 853 | | Bad practice - Creates an empty zip file entry | java | linear (0.125) |
| 854 | | Bad practice - Covariant equals() method defined | java | linear (0.125) |
| 855 | | Bad practice - Covariant compareTo() method defined | java | linear (0.125) |
| 856 | | Bad practice - Finalizer does not call superclass finalizer | java | linear (0.03) |
| 857 | | Bad practice - Finalizer nullifies superclass finalizer | java | linear (0.03) |
| 858 | | Bad practice - Iterator next() method can't throw NoSuchElementException | java | linear (0.125) |
| 859 | | Bad practice - Method ignores exceptional return value | java | linear (0.125) |
| 860 | | Bad practice - Method may fail to close database resource | java | linear (0.125) |
| 861 | | Bad practice - Method may fail to close stream | java | linear (0.125) |
| 862 | | Bad practice - Method with Boolean return type returns explicit null | java | linear (0.125) |
| 863 | | Bad practice - equals() method does not check for null argument | java | linear (0.125) |
| 864 | | Bad practice - toString method may return null | java | linear (0.125) |
| 865 | | Correctness - A collection is added to itself | java | linear (0.125) |
| 866 | | Correctness - A parameter is dead upon entry to a method but overwritten | java | linear (0.125) |

| | | | |
|---|---|---|---|
| 867 | Correctness - An apparent infinite loop | java | linear (0.125) |
| 868 | Correctness - An apparent infinite recursive loop | java | linear (0.125) |
| 869 | Correctness - Bitwise OR of signed byte value | java | linear (0.125) |
| 870 | Correctness - Bitwise add of signed byte value | java | linear (0.125) |
| 871 | Correctness - Check for sign of bitwise operation | java | linear (0.125) |
| 872 | Correctness - Check to see if ((...) & 0) == 0 | java | linear (0.125) |
| 873 | Correctness - Explicit annotation inconsistent with use | java | linear (0.125) |
| 874 | Correctness - Explicit annotation inconsistent with use | java | linear (0.125) |
| 875 | Correctness - Futile attempt to change max pool size of ScheduledThreadPoolExecutor | java | linear (0.125) |
| 876 | Correctness - Nonsensical self computation involving a field (e.g., x & x) | java | linear (0.125) |
| 877 | Correctness - Nonsensical self computation involving a variable (e.g., x & x) | java | linear (0.125) |
| 878 | Correctness - Null value is guaranteed to be dereferenced | java | linear (0.125) |
| 879 | Correctness - Null pointer dereference in method on exception path | java | linear (0.125) |
| 880 | Correctness - Null pointer dereference | java | linear (0.125) |
| 881 | Correctness - Number of format-string arguments does not correspond to number of placeholders | java | linear (0.125) |
| 882 | Correctness - Possible null pointer dereference | java | linear (0.125) |
| 883 | Correctness - Possible null pointer dereference in method on exception path | java | linear (0.125) |
| 884 | Correctness - Read of unwritten field | java | linear (0.125) |
| 885 | Correctness - Repeated conditional tests | java | linear (0.125) |
| 886 | Correctness - Self comparison of value with itself | java | linear (0.125) |
| 887 | Correctness - Self comparison of field with itself | java | linear (0.125) |
| 888 | Correctness - Self assignment of field | java | linear (0.125) |
| 889 | Correctness - Return value of putIfAbsent ignored, value passed to putIfAbsent reused | java | linear (0.125) |
| 890 | Correctness - Signature declares use of unhashable class in hashed construct | java | linear (0.125) |
| 891 | Correctness - Static Thread.interrupted() method invoked on thread instance | java | linear (0.125) |
| 892 | Correctness - Value annotated as carrying a type qualifier used where a value that must not carry that qualifier is required | java | linear (0.125) |
| 893 | Correctness - Value annotated as never carrying a type qualifier used where value carrying that qualifier is required | java | linear (0.125) |
| 894 | Correctness - Value is null and guaranteed to be dereferenced on exception path | java | linear (0.125) |
| 895 | Correctness - equals method overrides equals in superclass and may not be symmetric | java | linear (0.125) |
| 896 | Correctness - equals method compares class names rather than class objects | java | linear (0.125) |
| 897 | Correctness - equals method always returns true | java | linear (0.125) |
| 898 | Correctness - equals method always returns false | java | linear (0.125) |
| 899 | Correctness - hasNext method invokes next | java | linear (0.125) |
| 900 | Correctness - instanceof will always return false | java | linear (0.125) |
| 901 | Dead store due to switch statement fall through | java | linear (0.125) |
| 902 | Dodgy - Complicated, subtle or wrong increment in for-loop | java | linear (0.125) |
| 903 | Dodgy - Computation of average could overflow | java | linear (0.125) |
| 904 | Dodgy - Dereference of the result of readLine() without nullcheck | java | linear (0.125) |
| 905 | Dodgy - Double assignment of local variable | java | linear (0.125) |
| 906 | Dodgy - Immediate dereference of the result of readLine() | java | linear (0.125) |
| 907 | Dodgy - Method discards result of readLine after checking if it is nonnull | java | linear (0.125) |
| 908 | Dodgy - Method uses the same code for two branches | java | linear (0.125) |
| 909 | Dodgy - Method uses the same code for two switch clauses | java | linear (0.125) |
| 910 | Dodgy - Possible null pointer dereference due to return value of | java | linear (0.125) |

| | | | | |
|---|---|---|---|---|
| | | called method | | |
| 911 | | Dodgy - Possible null pointer dereference on path that might be infeasible | java | linear (0.125) |
| 912 | | Dodgy - Potentially dangerous use of non-short-circuit logic | java | linear (0.125) |
| 913 | | Dodgy - Self assignment of local variable | java | linear (0.03) |
| 914 | | Dodgy - instanceof will always return true | java | linear (0.125) |
| 915 | | DoNotIgnoreMethodResultRule | cs | linear (0.125) |
| 916 | | BadRecursiveInvocationRule | cs | linear (0.125) |
| 917 | | ToStringShouldNotReturnNullRule | cs | linear (0.125) |
| 918 | | CloneMethodShouldNotReturnNullRule | cs | linear (0.125) |
| 919 | | UseObjectDisposedExceptionRule | cs | linear (0.125) |
| 920 | | EqualsShouldHandleNullArgRule | cs | linear (0.125) |
| 921 | | CheckParametersNullityInVisibleMethodsRule | cs | linear (0.03) |
| 922 | | DisposableFieldsShouldBeDisposedRule | cs | linear (0.125) |
| 923 | | DisposableTypesShouldHaveFinalizerRule | cs | linear (0.125) |
| 924 | | GetEntryAssemblyMayReturnNullRule | cs | constant_resource (0.125) |
| 925 | | UseValueInPropertySetterRule | cs | linear (0.03) |
| 926 | | AvoidAlwaysNullFieldRule | cs | linear (0.03) |
| 927 | | ProvideValidXPathExpressionRule | cs | linear (0.125) |
| 928 | | ProvideValidXmlStringRule | cs | linear (0.125) |
| 929 | | ReviewCastOnIntegerMultiplicationRule | cs | linear (0.125) |
| 930 | | ReviewCastOnIntegerDivisionRule | cs | linear (0.125) |
| 931 | | ReviewUseOfModuloOneOnIntegersRule | cs | linear (0.125) |
| 932 | **Synchronization related** | Double Checked Locking | java | linear (0.375) |
| 933 | | Use Notify All Instead Of Notify | java | linear (0.375) |
| 934 | | Avoid Thread Group | java | linear (0.375) |
| 935 | | Close Resource | java | linear (0.125) |
| 936 | | Do Not Use Threads | java | linear (0.125) |
| 937 | | Double checked locking | java | linear (0.375) |
| 938 | | Non Thread Safe Singleton | java | linear (0.375) |
| 939 | | Unsynchronized Static Date Formatter | java | linear (0.125) |
| 940 | | Multithreaded correctness - Call to static Calendar | java | linear (0.125) |
| 941 | | Multithreaded correctness - Call to static DateFormat | java | linear (0.125) |
| 942 | | Multithreaded correctness - Class's readObject() method is synchronized | java | linear (0.125) |
| 943 | | Multithreaded correctness - Class's writeObject() method is synchronized but nothing else is | java | linear (0.125) |
| 944 | | Multithreaded correctness - Condition.await() not in loop | java | linear (0.125) |
| 945 | | Multithreaded correctness - Constructor invokes Thread.start() | java | linear (0.125) |
| 946 | | Multithreaded correctness - Empty synchronized block | java | linear (0.125) |
| 947 | | Multithreaded correctness - Field not guarded against concurrent access | java | linear (0.125) |
| 948 | | Multithreaded correctness - Inconsistent synchronization | java | linear (0.125) |
| 949 | | Multithreaded correctness - Inconsistent synchronization | java | linear (0.125) |
| 950 | | Multithreaded correctness - Incorrect lazy initialization and update of static field | java | linear (0.125) |
| 951 | | Multithreaded correctness - Incorrect lazy initialization of static field | java | linear (0.125) |
| 952 | | Multithreaded correctness - Invokes run on a thread (did you mean to start it instead?) | java | linear (0.125) |
| 953 | | Multithreaded correctness - Method calls Thread.sleep() with a lock held | java | linear (0.125) |
| 954 | | Multithreaded correctness - Method does not release lock on all exception paths | java | linear (0.125) |
| 955 | | Multithreaded correctness - Method does not release lock on all paths | java | linear (0.125) |
| 956 | | Multithreaded correctness - Method spins on field | java | linear (0.125) |

| | | | | |
|---|---|---|---|---|
| 957 | | Multithreaded correctness - Method synchronizes on an updated field | java | linear (0.125) |
| 958 | | Multithreaded correctness - Mismatched notify() | java | linear (0.125) |
| 959 | | Multithreaded correctness - Mismatched wait() | java | linear (0.125) |
| 960 | | Multithreaded correctness - Monitor wait() called on Condition | java | linear (0.125) |
| 961 | | Multithreaded correctness - Mutable servlet field | java | linear (0.125) |
| 962 | | Multithreaded correctness - Naked notify | java | linear (0.125) |
| 963 | | Multithreaded correctness - Possible double check of field | java | linear (0.125) |
| 964 | | Multithreaded correctness - Static Calendar | java | linear (0.125) |
| 965 | | Multithreaded correctness - Static DateFormat | java | linear (0.125) |
| 966 | | Multithreaded correctness - Sychronization on getClass rather than class literal | java | linear (0.125) |
| 967 | | Multithreaded correctness - Wait with two locks held | java | linear (0.125) |
| 968 | | Multithreaded correctness - Wait not in loop | java | linear (0.125) |
| 969 | | Multithreaded correctness - Using notify() rather than notifyAll() | java | linear (0.125) |
| 970 | | Multithreaded correctness - Unsynchronized get method, synchronized set method | java | linear (0.125) |
| 971 | | Multithreaded correctness - Unconditional wait | java | linear (0.125) |
| 972 | | Multithreaded correctness - Synchronize and null check on the same field. | java | linear (0.125) |
| 973 | | Multithreaded correctness - Synchronization performed on java.util.concurrent Lock | java | linear (0.125) |
| 974 | | Multithreaded correctness - Synchronization on interned String could lead to deadlock | java | linear (0.125) |
| 975 | | Multithreaded correctness - Synchronization on field in futile attempt to guard that field | java | linear (0.125) |
| 976 | | Multithreaded correctness - Synchronization on boxed primitive values | java | linear (0.125) |
| 977 | | Multithreaded correctness - Synchronization on boxed primitive could lead to deadlock | java | linear (0.125) |
| 978 | | Multithreaded correctness - Synchronization on Boolean could lead to deadlock | java | linear (0.125) |
| 979 | | Bad practice - Certain swing methods needs to be invoked in Swing thread | java | linear (0.125) |
| 980 | | Dodgy - Class exposes synchronization and semaphores in its public interface | java | linear (0.125) |
| 981 | | DoNotUseThreadStaticWithInstanceFieldsRule | cs | linear (0.125) |
| 982 | | DoNotUseMethodImplOptionsSynchronizedRule | cs | linear (0.125) |
| 983 | | DoubleCheckLockingRule | cs | linear (0.125) |
| 984 | | DoNotLockOnWeakIdentityObjectsRule | cs | linear (0.125) |
| 985 | | DoNotLockOnThisOrTypesRule | cs | linear (0.125) |
| 986 | | ReviewLockUsedOnlyForOperationsOnVariablesRule | cs | linear (0.125) |
| 987 | | UseSTAThreadAttributeOnSWFEntryPointsRule | cs | linear (0.125) |
| 988 | | WriteStaticFieldFromInstanceMethodRule | cs | linear (0.03) |
| 989 | | NonConstantStaticFieldsShouldNotBeVisibleRule | cs | linear (0.125) |
| 990 | | COM registration methods should be matched (FxCop10) | cs | linear (0.125) |
| 991 | | Rethrow to preserve stack details (FxCop10) | cs | linear (0.03) |
| 992 | Integration level | Instantiate argument exceptions correctly (FxCop10) | cs | linear (0.125) |
| 993 | Testability | Implement standard exception constructors | cs | linear (0.125) |
| 994 | | Exceptions should be public | cs | linear (0.125) |
| 995 | | Do not raise reserved exception types (FxCop10) | cs | linear (0.125) |
| 996 | | Do not raise exceptions in unexpected locations | cs | linear (0.125) |
| 997 | | Do not raise exceptions in exception clauses (FxCop10) | cs | linear (0.125) |
| 998 | | Do not catch general exception types | cs | linear (0.125) |
| 999 | | Illegal Throws | java | linear (0.375) |
| 1000 | | Bad practice - Method might drop exception | java | linear (0.125) |
| 1001 | | Bad practice - Method may fail to close database resource on exception | java | linear (0.125) |

| | | | | |
|---|---|---|---|---|
| 1002 | | Bad practice - Dubious catching of IllegalMonitorStateException | java | linear (0.125) |
| 1003 | | Illegal Catch | java | linear (0.125) |
| 1004 | Unit level | Parameter Number | java | linear (0.125) |
| 1005 | | Cyclomatic Complexity | java | linear (0.375) |
| 1006 | | NPath Complexity | java | linear (0.375) |
| 1007 | | Code size - cyclomatic complexity | java | linear (0.375) |
| 1008 | | Exception As Flow Control | java | linear (0.125) |
| 1009 | | Excessive Parameter List | java | linear (0.125) |
| 1010 | | NPath complexity | java | linear (0.125) |
| 1011 | | Correctness - TestCase implements a non-static suite method | java | linear (0.03) |
| 1012 | | Correctness - TestCase has no tests | java | linear (0.03) |
| 1013 | | Correctness - TestCase defines tearDown that doesn't call super.tearDown() | java | linear (0.03) |
| 1014 | | Correctness - TestCase defines setUp that doesn't call super.setUp() | java | linear (0.03) |
| 1015 | | Correctness - TestCase declares a bad suite method | java | linear (0.03) |
| 1016 | | AvoidLongMethodsRule | cs | linear (0.375) |
| 1017 | | AvoidLargeClassesRule | cs | linear (0.375) |
| 1018 | | AvoidLargeNumberOfLocalVariablesRule | cs | linear (0.125) |
| 1019 | | AvoidLargeStructureRule | cs | linear (0.375) |
| 1020 | | AvoidLongParameterListsRule | cs | linear (0.125) |
| 1021 | | AvoidComplexMethodsRule | cs | linear (0.375) |
| 1022 | | Duplicated blocks | java | linear (0.125) |
| 1023 | | Duplicated blocks | cs | linear (0.125) |
| 1024 | | Replace repetitive arguments with params array | cs | linear (0.03) |
| 1025 | | AvoidSwitchStatementsRule | cs | linear (0.375) |
| 1026 | | Insufficient line coverage by unit tests | cs | linear (0.01) |
| 1027 | | Insufficient line coverage by unit tests | java | linear (0.01) |
| 1028 | | Insufficient branch coverage by unit tests | cs | linear (0.05) |
| 1029 | | Insufficient branch coverage by unit tests | java | linear (0.05) |

# J. List of installed software on the PoC VM

- Apache Tomcat 7.0 (remove only)
- Application Verifier
- ASP.NET Security Rules for FxCop
- Crystal Reports Basic for Visual Studio 2008
- Debugging Tools for Windows (x86)
- Dropbox
- Enterprise Library for .NET Framework 2.0 - January 2006
- Gallio 3.2 build 750
- Gendarme 2.10
- Hyper-V Integration Services (version 6.1.7601.17514)
- Java DB 10.6.2.1
- Java(TM) 6 Update 24
- Java(TM) SE Development Kit 6 Update 24
- Microsoft .NET Compact Framework 2.0 SP2
- Microsoft .NET Compact Framework 3.5
- Microsoft .NET Framework 2.0 Service Pack 2
- Microsoft .NET Framework 3.0 Service Pack 2
- Microsoft .NET Framework 3.5 SP1
- Microsoft .NET Framework 4 Client Profile
- Microsoft .NET Framework 4 Extended
- Microsoft .NET Framework 4 Multi-Targeting Pack
- Microsoft BizTalk Server 2009 Developer Edition
- Microsoft CRM SDK
- Microsoft Device Emulator version 3.0 - ENU
- Microsoft Document Explorer 2008
- Microsoft Enterprise Single Sign-On
- Microsoft FxCop 1.36
- Microsoft FxCop 10.0
- Microsoft Help Viewer 1.0
- Microsoft Office 2003 Web Components
- Microsoft Primary Interoperability Assemblies 2005
- Microsoft Report Viewer Redistributable 2005
- Microsoft ReportViewer 2010 Redistributable
- Microsoft SQL Server 2005
- Microsoft SQL Server 2005 Analysis Services ADOMD.NET
- Microsoft SQL Server 2008 Analysis Services ADOMD.NET
- Microsoft SQL Server 2008 Management Objects
- Microsoft SQL Server Compact 3.5 for Devices ENU
- Microsoft SQL Server Compact 3.5 SP1 Design Tools English
- Microsoft SQL Server Compact 3.5 SP1 English
- Microsoft SQL Server Database Publishing Wizard 1.3
- Microsoft SQL Server Native Client
- Microsoft SQL Server Setup Support Files (English)
- Microsoft SQL Server VSS Writer
- Microsoft SQLXML 4.0 SP1
- Microsoft StyleCop 4.4.0.14
- Microsoft Visual C++ Compilers 2010 Standard - enu - x86
- Microsoft Visual C++ 2010 x86 Redistributable - 10.0.30319
- Microsoft Visual Studio 2008 Professional Edition - ENU
- Microsoft Visual Studio Web Authoring Component
- Microsoft Windows Performance Toolkit
- Microsoft Windows SDK for Visual Studio 2008 .NET Framework Tools - enu
- Microsoft Windows SDK for Visual Studio 2008 Headers and Libraries
- Microsoft Windows SDK for Visual Studio 2008 SDK Reference Assemblies and IntelliSense
- Microsoft Windows SDK for Visual Studio 2008 SP1 Tools
- Microsoft Windows SDK for Visual Studio 2008 SP1 Win32 Tools
- Microsoft Windows SDK for Windows 7 (7.1)
- Mozilla Firefox 4.0 (x86 nl)
- MSXML 6.0 Parser
- MySQL Server 5.5
- Notepad++
- PartCover .NET 4.0
- SequoiaView
- SourceMonitor V2.6.3.104
- SQL Server System CLR Types
- Tools for SLN File
- Visual Studio 2005 Tools for Office Second Edition Runtime
- Visual Studio Team System 2008 Database Edition - ENU
- Visual Studio Team System 2008 Database Edition GDR - ENU
- Visual Studio Tools for the Office system 3.0 Runtime
- Windows Installer XML Toolset 3.0
- Windows Internet Explorer 8
- Windows Mobile 5.0 SDK R2 for Pocket PC
- Windows Mobile 5.0 SDK R2 for Smartphone
- Windows Support Tools
- WinRAR
- WinSCP 4.3.2
- XAMPP 1.7.4

## K. Project identifier list [CONFIDENTIAL]

The unrestricted version of this thesis does not contain this confidential appendix.