Measuring Quality Improvements After Stimulating Software Quality Awareness Among Developers

Master's Thesis

Remy Alidarso

Measuring Quality Improvements After Stimulating Software Quality Awareness Among Developers

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Remy Alidarso
born in Leidschendam, the Netherlands



Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl



KPMG
Pr. Catharina-Amaliastr. 5
The Hague, the Netherlands
www.kpmg.nl

© 2011 - 2012 Remy Alidarso.

Measuring Quality Improvements After Stimulating Software Quality Awareness Among Developers

Author: Remy Alidarso
Student id: 1221736
Email: r.alidarso@gmail.com

Abstract

Software systems are getting larger and more complex. It takes therefore more time and money to maintain these systems. The maintenance effort is strongly related to the quality of the implementation during the development phase. Providing qualitative numbers to developers about their previous implementations could help increase the quality of their next implementation. In this thesis an approach is presented both for gathering internal software quality metrics that are related to a system's maintainability and also for extracting information from these metrics. The extracted information is then returned as feedback to the development team to give them the ability to improve their source code. This in turn will increase the virtuous circle of improved maintainability which again results in better software quality overal. This is done via a self-made feedback reporting tool which is described in detail. Three projects have been followed both before and after developers got access to our feedback mechanism. Afterwards, we evaluate the situations.

Thesis Committee:

Chair:	Prof. Dr. A. van Deursen, Faculty EEMCS, TU Delft
University supervisor:	Dr. A. Zaidman, Faculty EEMCS, TU Delft
Company supervisor:	Drs. E. van den Brom MIM CISA, KPMG
Committee Member:	Dr. Z. Al-Ars, Faculty EEMCS, TU Delft

Preface

This project was done at KPMG, to whom I am grateful for the opportunity to write my thesis in an inspiring and challenging environment with great colleagues. In particular I would like to thank my supervisor Erik van den Brom, for always being able to challenge me in my work and for asking the right questions. I would also like to thank my supervisor from Delft University of Technology Andy Zaidman for his endless patience and for being such a dedicated researcher who kept me focussing on the things that matter.

For proofreading concept versions of my thesis I would like to express my gratitude and appreciation to Cor-Paul Bezemer, Rali Genova, and Jasper Oosterman.

I would finally like to thank my parents for their unconditional support throughout my entire life.

Remy Alidarso
Delft, the Netherlands
September 18, 2012

Contents

Preface iii

Contents v

List of Figures vii

List of Tables ix

1 Introduction 1
- 1.1 Problem Statement . . . 1
- 1.2 Research Questions . . . 2
- 1.3 Approach . . . 3
- 1.4 Related Work . . . 4

2 Software Quality 7
- 2.1 Quality Model . . . 10
- 2.2 Internal Quality . . . 12
- 2.3 Metrics . . . 12
- 2.4 Remarks . . . 17

3 Tooling Infrastructure 19
- 3.1 Requirements . . . 19
- 3.2 Continuous Integration . . . 20
- 3.3 Source Code Analysis Tools . . . 27
- 3.4 Sonar . . . 31
- 3.5 Miscellaneous . . . 34

4 Research Approach 35
- 4.1 Awareness . . . 35
- 4.2 Data . . . 36

5 Results **45**
5.1 Introduction 45
5.2 Project A 46
5.3 Project B 53
5.4 Project C 62
5.5 Developer Interviews 67
5.6 Threats to Validity 68
5.7 Discussion 70

6 Conclusions and Future Work **73**
6.1 Conclusions 73
6.2 Contributions 75
6.3 Future work 76

Bibliography **77**

A Terminology **81**

B Interview with Joost Koedijk on January 26th 2012 **83**

C Interview with Anton Gochev on February 9th 2012 **85**

List of Figures

2.1 Quality model of Cavano and McCall, as proposed in their paper "A Framework for the Measurement of Software Quality" [4] 9
2.2 Quality in the lifecycle of a software product [12]. 13

4.1 Schematic picture of the feedback mechanism internals. 40
4.2 An image of the standard deviation of a normal distribution with confidence intervals. 41
4.3 An example graph depicting the fact that a recent change in the amount of duplicated lines has been detected. 42
4.4 Schematic picture of the feedback mechanism as "black box" in the development process. 43

5.1 Graph of historic numbers for project A with Lines of Code, Files, Classes plotted against each other. 48
5.2 Graph of historic numbers for project A with Blocker Violations, Critical Violations and Major Violations plotted against each other. 48
5.3 Graph of historic numbers for project A with Lines of Code, Complexity and Violations plotted against each other. 49
5.4 Graph of historic numbers for project A with Lines of Code, Methods and Percentage of Duplicated Lines plotted against each other. 49
5.5 Graph of historic numbers for the business logic of project B with Lines of Code, Files, Classes plotted against each other. 54
5.6 Graph of historic numbers for the business logic of project B with Blocker Violations, Critical Violations and Major Violations plotted against each other. 55
5.7 Graph of historic numbers for the business logic of project B with Lines of Code, Complexity and Violations plotted against each other. 56
5.8 Graph of historic numbers for the business logic of project B with Lines of Code, Methods and Percentage of Duplicated Lines plotted against each other. 57

5.9 Graph of historic numbers for project B with Lines of Code, Files, Classes plotted against each other. 58
5.10 Graph of historic numbers for project B with Lines of Code, Complexity and Violations plotted against each other. 59
5.11 Graph of historic numbers for project B with Blocker Violations, Critical Violations and Major Violations plotted against each other. 59
5.12 Graph of historic numbers for project B with Lines of Code, Methods and Percentage of Duplicated Lines plotted against each other. 60
5.13 Graph of historic numbers for project C with Lines of Code, Files, Classes plotted against each other. 63
5.14 Graph of historic numbers for project C with Lines of Code, Methods and Percentage of Duplicated Lines plotted against each other. 63
5.15 Graph of historic numbers for project C with Lines of Code, Complexity and Violations plotted against each other. 64
5.16 Graph of historic numbers for project C with Blocker Violations, Critical Violations and Major Violations plotted against each other. 65

List of Tables

3.1 Comparison of proprietary Continuous Integration suites as well as plugin support. 22
3.2 Comparison of builders the different Continuous Integration suites support. 25
3.3 Comparison of Continuous Integration suites with Software Configuration Management support. 25
3.4 Comparison of Continuous Integration suites with Issue-Tracking Systems support. [36] 26
3.5 Definition of lines of code. [23] 28

5.1 Table with historic data for project A 47
5.2 Table with violations data for project A. 50
5.3 Table with historic data for the business logic of project B. 54
5.4 Table with violations data for the business logic for project B. 57
5.5 Table with historic data of project B. 58
5.6 Table with violations data for project B. 59
5.7 Table with historic data for project C. 64
5.8 Table with violations data for Project C. 64

Chapter 1

Introduction

This introduction chapter is meant to give a motivation for this research, to give the problem statement and to give accompanying research questions. A research approach is given to solve these problems.

This section continues as follows: First section 1.1 will describe the problems addressed in this research. Section 1.2 gives an overview of the research questions this thesis answers. Finally, a small section is dedicated to describe the approach in which we try to overcome these problems.

1.1 Problem Statement

In this research we focus on the idea that software quality can be improved by stimulating developers' awareness towards the current state of software quality. In other words: "Can we observe a (positive) change in software quality when making developers more aware of software quality?"

Software is always changing, sometimes because new features have to be implemented and sometimes due to maintenance on existing code. The state of software quality is therefore also always changing.

This is important because software engineering is expensive due to the custom work accompanying it. Unlike large physical engineering projects which can be built (possibly only) with the help of machinery, large software engineering applications are still dependent on manual labour. These costs can be optimized for two situations, either for the short term or for the long term. These correspond to the two ways in which developers can implement some functionality. They can do this in a quick, but messy way, or in a clean way which will take longer [13]. There is always future work to be done on software products. The effort this work takes is called *Technical Debt*. Ward Cunningham invented this term and he describes it as the costs associated with maintaining and enhancing a platform where shortcuts were taken in order to meet financial or time constraints [8]. This all holds for software products that were taken into production. But how about software products still under development?

Even today the newspapers remind us that large IT projects fail [9]. Many reasons can be named for the failure of IT projects. Software quality encompasses an extensive area because software engineering has many different facets, all of which can go wrong. We focus on the development process. This research is directed towards improving the internal software quality, which is one of the elements of software quality. Software testing - which is often associated with improving software quality - is also part of software development. Software testing has - as part of quality assurance - the goal to detect defects in a software application. This is done for instance with unit tests. But these unit tests approach the software quality from the "outside" inwards, or done from a users' perspective. Charette et al. blame the failure of IT projects among others on "sloppy development practices" [5]. These "sloppy development practices" can mean two things, either that developers are sloppy in following a certain process, or in the way they produce source code. We focus on the latter. The way in which developers produce source code is directly related to the internal software quality. Therefore, it can be deducted that we can show an improvement in software quality by enhancing the way developers produce source code.

1.2 Research Questions

The experiments done in this research are meant to improve the internal software quality, and with this, the overal software quality. Multiple definitions of internal software quality exist, of which some can be found in chapter 2. Every stakeholder involved in a software engineering project has his own definition for software quality. Because we like to use an objective way of determining software quality, we make use of some parts of the widely known ISO 9126 standard for software quality. The improvement of software quality is achieved either by creating or stimulating developers' awareness for the current state of the software quality. The idea behind this is that although developers know how to implement a certain functionality, they are not all familiar with factors that matter for the internal software quality. To be able to *improve* the internal software quality, one first has to be able to *measure* this internal software quality. In this thesis, we will address the problem of measuring internal software quality. In particular, we will focus on the following research questions:

- **RQ1** What is software quality?
- **RQ2** Can we observe improvement in software quality when awareness of software quality is stimulated among developers?
- **RQ3** How can software quality be best improved by solely stimulating developer awareness towards the state of software quality?

In order to answer these questions, we have divided each of the questions above in multiple subquestions. The first topic we cover is software quality, of which internal software quality is a part of. Since there are many publications about software quality

and how it is defined [3][11][4][12], we need to clarify the definition we use. Because software cannot be touched, we need to know how we can take measurements of intangible objects. The following questions help us to understand this.

- **RQ1.1** How do we define software quality?
- **RQ1.2** How do we measure software quality?
- **RQ1.3** How can measurements of software quality be compared?

The second research question requires again additional subquestions to answer.

- **RQ2.1** Is developer awareness with respect to software quality correlated with the actual state of software quality?
- **RQ2.2** Can software quality be improved by only making developers (more) aware of software quality?

In the final research question the involvement of humans is central. The outcome of the experiments should be handled with care because humans introduce a subjective factor. The following subquestions should be answered to be able to give objective statements.

- **RQ3.1** In which way can developer awareness for software quality be best stimulated?
- **RQ3.2** Which information is most important for stimulating developer awareness of software quality?
- **RQ3.3** How should this information be disclosed to developers?

1.3 Approach

The goal of this research is to see if the internal software quality can be improved by making developers more aware of the internal software quality and how this can be achieved. We have mentioned that we first need to have the ability to measure the current state of (internal) software quality before we can compare measurements. To measure the current state of the internal software quality, multiple tools exist that can be used free of charge. The drawback of these tools is that they do not filter any of the data they gather. Another drawback of these tools is that they store their data differently, making direct comparisons difficult. To be able to store this data in a homogeneous way, we will make use of an umbrella tool that gathers the data of all these small tools and stores it. With this data stored in a centralized place in a homogenic way, we are able to extract useful data from it in a unified way. Only then we are able to compare measurements. But the mass amount of information still is a problem. For the purpose of generating a general sense of the (internal) software

quality, we have designed and implemented a software quality feedback mechanism that is able to interpret the gathered data. This feedback mechanism does not use all the available data but rather a subset that forms a good base for denoting the state of the internal software quality.

1.4 Related Work

The experiments done in this research do not stand on their own. The ideas for this research came from earlier research projects. We have seen that much effort has been put into research concerning software quality and ways to improve this. Also much research has been done in the past to find software quality metrics. On these topics much has been written in papers and books, of which we will now name some. Not much related work has been found on trying to make developers more aware of internal software quality.

Software Quality

Much has been written about software quality. Much earlier work about software quality or quality in general has been crystallized and has been bundled in these publications.

- The International Organization for Standardization has published its ISO 9126 standard describing software quality in depth [12]. This is still an authority when it comes to objectively publishing about software quality.
- Juran's book consists of multiple chapters where each chapter describes quality in a different field [17]. A dedicated chapter about software quality is included in the book.

Software Quality Improvement

Many publications can be found on ways to improve software quality. These ways range from changing the software development processes, to using new kinds of programming language constructs that decrease the likelihood of making much made mistakes.

- Michele Lanza and Radu Marinescu wrote a book about object-oriented metrics, and how they are used in practise [19]. It can be used to improve the overal software quality by improving the design of a software project.
- John Ferguson Smart wrote the book "Jenkins - The Definitive Guide" [32]. In this book not only Jenkins itself is addressed, but also the idea is given to use Jenkins to give developers insight in their software quality. In essence this is the same idea as used in this research project. The difference is that we use a more abstract insight to make developers aware of their produced software quality instead of only serving highly detailed information.

Software Quality Metrics

From the early 1970's much research has been done to be able to evaluate software on

a certain scale. Metrics were invented to be able to take measurements of software. Since that time, programming languages have evolved a lot, and new metrics had to be invented that reflect these changes.

- Mark Lorenz and Jeff Kidd wrote the book "Object-Oriented Software Metrics", a book about metrics specifically aimed for projects built with object-oriented programming languages [25].
- Henderson-Sellers also wrote a book specifically about object-oriented metrics, which is called "Object-Oriented metrics: measures of complexity" [14].

Chapter 2

Software Quality

Engineering projects - that includes software engineering projects - are generally subjected to extensive tests as part of a quality assurance programme to be able to guarantee a certain level of correct functioning before they are taken into production. The quality of an engineering project cannot be defined with a single factor. Software engineering also differs from other engineering fields in the way its quality level is established, not in the first place because a software product is intangible. Software products also differ from other engineering products because the software products should always be evolving during their life time to be successful [19]. Another important fact about software engineering is that it is still manual labour; This means that many "solutions" exist for the same problem, there is no single "best" solution, and all of these solutions are made by hand. This is why it is hard to determine the quality of a software engineering product.

Let us first look at the regular definition of quality. Many definitions for quality exist. Many quality forerunners have given short sentences to define the word "quality". [21] States that Joseph Juran uses the definition "Fitness for use" [17], while Phil Crosby uses the more usable definition "Conformance to Requirements" [7]. In his book, Juran continues by stating that quality entails more than can be captured with such short statements. The definitions stated above can be used in any situation, for almost any engineering product. This means we must try to find a more suitable definition that is specific for software. For our research we make a distinction between code quality and software quality. As Juran states in his book, quality is stakeholder dependent [17]. If software quality means quality perception from every stakeholder, code quality means quality perception from a developer point of view. As we will see later, the code quality influences the way in which other people will perceive the software product, and thus the software quality itself.

Establishing the quality state of a product is done by letting the product undergo certain tests. Testing software products can be done from the outside inwards, or from the inside outwards. The former usually consists of testing for broken functionality while the latter usually consists of implementation errors which may (or may not!) lead to system failure. Testing of defects might seem to have a significant importance,

but Edsger Dijkstra, a well-known Dutch computer scientist once said: "Testing can only show the presence of bugs not their absence". With this he meant that only known defects can be spotted, it does not guarantee that the source code is free from any defects.

Related with testing for failure rates is taking measurements of system properties. Identical to other engineering products, we should take measurements on which we are able to make confident statements about the state of quality. But as stated in the introduction, software quality is difficult to measure because software is intangible. It does not have any physical properties which can be measured. In the late 1970s McCall and Cavano said the following: "A major difficulty in dealing with software is that there are no quantitative measures of the quality of a software product." [4]. The two proposed a measurement model which contains a hierarchical definition of software quality. This model consists of multiple factors, which in turn depend on characteristics, each again with their own sub-characteristics. Examples of factors on which software quality is based on could be for instance: reliability, scalability, maintainability, testability etc. Characteristics could be for instance complexity, duplication, volume, coupling, cohesion etc. Some of these characteristics or sub-characteristics can be measured, thus making certain parts measurable.

The idea of Cavano and McCall has been embraced by the international communities and the International Organization for Standardization (ISO) has therefore developed a standard in cooperation with the International Electrotechnical Commission (IEC) [12] for software product quality named the ISO/IEC 9126 standard. As many software quality standards, the ISO/IEC 9126 standard is based on the quality model first introduced by McCall and Cavano in 1978 when they developed a framework to measure software quality [4].

Development on the standard continued, and combined with ISO/IEC 14598 (the standard for software product evaluation) a new series of standards were created, which resulted in the ISO/IEC 25000 standard. ISO/IEC 25000 was named SQuaRE which stands for **S**oftware **Qua**lity **R**equirements and **E**valuation [3].

Both the ISO/IEC 9126 and ISO/IEC 25000 standards describe a quality model consisting of a structured set of factors with underlaying characteristics with accompanying sub-characteristics similar to the McCall and Cavano model.

SQuaRE consists of eight characteristics, whereas the older ISO/IEC 9126 standard consists of only six software quality factors or characteristics.

Because of the ever-changing nature of the entire computer science field, standards or methods to assess the quality should evolve equally. After the introduction of the SQuaRE standard, a French branch of the Norwegian independent foundation called "Det Norske Veritas"[1] developed a method to assess software quality which they named *SQALE*, or **S**oftware **Q**uality **A**ssessment based on **L**ifecycle **E**xpectations. This model is different from other models in the fact that SQALE emphasizes the fact that maintainability is necessary throughout its entire lifetime.

[1] http://www.dnv.com

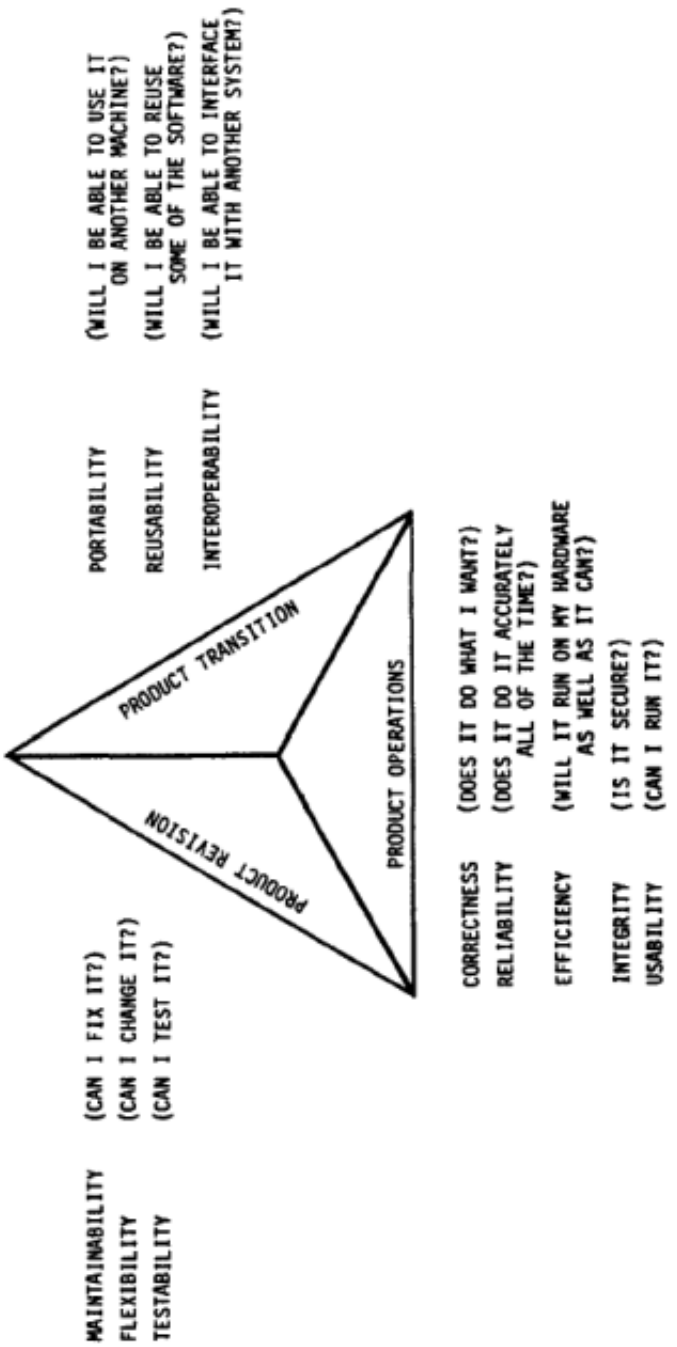


Figure 2.1: Quality model of Cavano and McCall, as proposed in their paper “A Framework for the Measurement of Software Quality” [4]

It focuses on making technical debt clear. The term "technical debt" describes how much money would be involved to maintain a certain software project.

Since many of the more recent models are based on the model as used in the ISO/IEC 9126 standard, we use this particular model to explain some of its inner working.

A quality model consists of several characteristics and sub-characteristics. These characteristics should be based on something that can be measured. Because software cannot be measured physically, the only way to take measurements is directly from the source code. These measurements are established by metrics. Underlying metrics define the degree to which these characteristics and sub-characteristics are followed.

The ISO/IEC 9126 standard distinguishes three kinds of metrics; internal, external and quality-in-use metrics. Each of the metrics contribute to a certain extent the degree to which the characteristics and sub-characteristics are followed. Because this research focusses on the improvement of internal metrics, the external and quality-in-use metrics are omitted.

This chapter explains how to define software quality through a top-down approach, therefore the use of quality models is first explained in section 2.1. The quality model is divided into several subsections of which only the section about internal quality will be described in 2.2. Section 2.3 explains which metrics are considered to be of importance for this particular research. Finally, a section with important remarks about the usage of these metrics will follow.

2.1 Quality Model

Determining the quality of a certain product is a structured process. One cannot simply take a few measurements of some tangible object and conclude that its quality is sufficient for its intended use. The object needs to be subjected to different tests that check if the intended job can indeed be done and if it can be done in a minimum amount of time before breaking (Ikea in-store "stress" tests for example).

The same applies for establishing the quality level of a software product. It is insufficient to simply measure the amount of lines of code or the format in which it was delivered for example. It is as if one would base a quality report for a building on the amount of bricks used, without calculating for instance the weight it can carry.

The only way to inspect software products is to submit them to tests that check if they indeed "do what they should do" (if the output is indeed the expected output for instance) and if the underlying mechanics are robust and safe to use. But these questions raise more questions. To test whether a certain software product "does what it should do", a certain perspective or intended use has to be taken into account. This is usually seen from the end-user's perspective. For instance if a customer expects a software product to scale proportional with the hardware it is run on, he may be very disappointed when the software does not scale that well. The other way around holds as well; What if a specialized cash register application is tested and one of the findings is that the software does not scale? The customer may be very happy

with the product although it will not score a high mark for scalability in some quality model.

Scalability and other such higher level functional requirements are all captured in a quality model's factors. The ISO/IEC 9126 quality model consists of six quality factors; six factors of which is believed that they (combined) give an accurate grade of the quality of a software product, seen from the user's perspective.

The six ISO/IEC 9126 factors are the following:

- functionality
- reliability
- usability
- efficiency
- maintainability
- portability

Of all these factors, only maintainability will be discussed, as this one encompasses the internal quality of a software product.

Maintainability

The maintainability factor gives insight in the possibility to modify the software product. This can include improvements (extended functionality, introduce new functionality), but also corrections (correct defects). Each factor in a quality model consists of multiple characteristics. The characteristics of maintainability are listed below followed by the key question they answer.

- **Analysability** Is it possible to diagnose any deficiencies?
- **Changeability** Can a certain modification be implemented?
- **Stability** Does the software product keep being functional in a stable way after it has been modified?
- **Testability** Can the modified product be tested in some way?
- **Maintainability Compliance** Does the product adhere to standards and regulations regarding maintainability?

The characteristics are meant as a guideline for reports on the state of the software quality of some product. The underlying numbers are leading to these characteristics. Reliability for instance can be measured from the exterior of the system by checking the number of failures in a certain amount of time. This can also be measured internally by checking for defects in the source code that can lead to failures.

Such internal and external aspects are defined with metrics. The numbers gathered from internal attributes should give users, testers, and developers confidence that the product is of high quality, which means that the application fullfils the needs of the users in a correct, robust and safe way.

A quality model such as the one found in the ISO/IEC9126 standard consists of multiple parts. The part in which we are interested in particular is the internal quality model because this is the part where influences of developers are reflected in. The other models included in the ISO/IEC9126 standard are excluded from this study since they are not related to the topic.

2.2 Internal Quality

As already stated, the internal quality of a software product has our main attention. The internal quality is represented by internal metrics. The purpose of these internal metrics is to ensure that external quality and quality-in-use is reached. This is because the internal quality influences the external quality, which in turn influences the quality in use.This is also depicted in figure 2.2. The idea behind this approach is that potential defects surface in an earlier stage of the development process, thus ensuring the end product contains less defects that may cause failures.

> When developing a software product the intermediate products should be evaluated using internal metrics which measure intrinsic properties, including those which can be derived from simulated behaviour [12].

The text above is quoted directly from the ISO/IEC9126 standard. It gives strenght to our motivation to attempt to improve the internal software quality while the software product is under development. Early discovery of faults in any of the characteristics can contribute to a successful end-product.

Internal metrics are mostly given as numbers or frequencies of small parts of source code such as statements or comments. Most of the internal metrics can be gathered with the static analysis tools that will be discussed in chapter 3.3.

2.3 Metrics

Now that the different quality categories have been explained, we can go a level of abstraction deeper into a list of internal quality metrics we would like to use in our experiment. As one can imagine, no single metric exists that is able to represent the software quality on its own. Therefore several metrics have been selected to capture a general sense of the categories listed below. Note that not every metric from the list below can be obtained because our tooling infrastructure does not support the gathering of every metric for every programming language. More on this in chapter 3.

If we look deeper into the area where internal quality is influenced, we can identify multiple sections for the metrics. It is important to make a distinction between metrics, so we can select a properly distributed set of metrics that equally represent each section. The following sections are being distinguished:

- **Coupling** “Object X is coupled to object Y if if and only if X sends a message to Y”[1]. A high coupling factor indicates that individual modules might not exist or they cannot be used in another software project. It also indicates that the underlying elements are grouped too much together. A low coupling factor (loose coupling) indicates that the underlying software elements can easily be separated and as such be reused in other projects. The looser the coupling, the less information is being exchanged between the different parts of the software product.

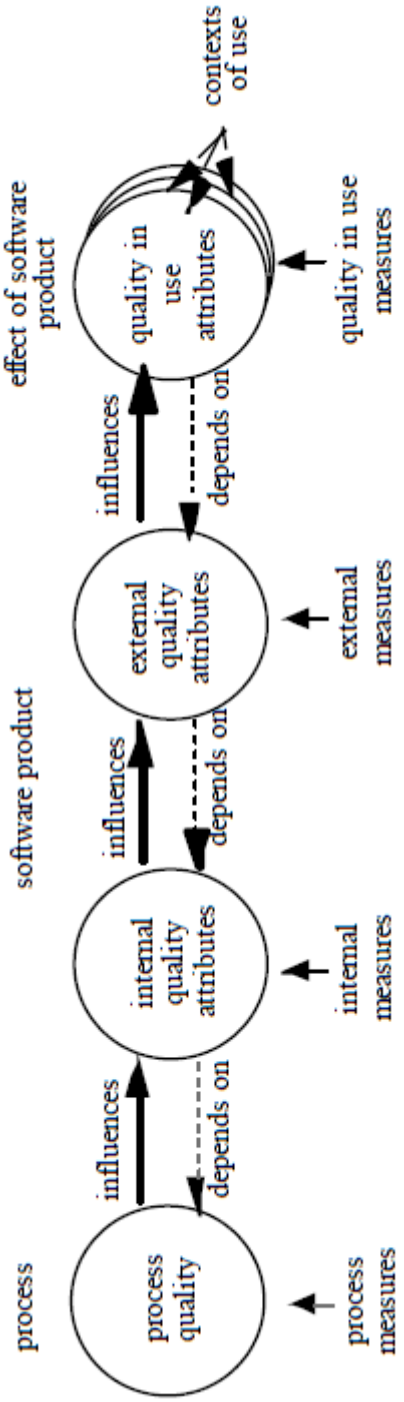


Figure 2.2: Quality in the lifecycle of a software product [12].

- **Cohesion** "The degree to which the methods within a class are related to one another" [1]. The higher the degree, the higher the amount of information that is shared and being acted on.

- **Size and Complexity** metrics are used to denote the size and complexity of the software product.

- **Inheritance** "A relationship among classes, wherein one class shares the structure or methods defined in one other class (for single inheritance) or in more than one other class (for multiple inheritance)" [1]. A well-layered software product separates for instance a data layer from the interface layer.

The granularity of these metrics can also be adjusted. Some metrics give details on the system level while others give details on the function level. Because the investigated projects are all created in an object-oriented way, the following levels are considered:

- System / Project
- Package
- File
- Class
- Method

This will be made clear in the descriptions below. Some of the metrics found below have evolved since their initial formulation because people were convinced adjustments were needed to reflect changes found in newer technologies. We will use the original definitions of the metrics, unless stated otherwise.

A few people have formulated metrics that still form the basis in establishing a quality level today. Thomas J. McCabe founded the notion "complexity" in his paper "A Complexity Measure" from December 1976 [26]. Chidamber and Kemerer have formulated six metrics including "depth of inheritance tree", "coupling between object classes", "number of children" and more in their "A Metrics Suite for Object-Oriented Design." from June 1994[6].

The following list of metrics have been selected to be general enough to give proper insight in the software quality.

Coupling:

- Coupling Between Object classes (CBO) - Defined as a count of the number of other classes to which it is coupled. The inventors Chidamber and Kemerer stated the following: "In order to improve modularity and promote encapsulation, inter-object class couples should be kept to a minimum. The larger the number of couples, the higher the sensitivity to changes in other parts of the design, and therefore maintenance is more difficult." Also: "A measure of coupling is useful to determine how complex the testing of various parts of a design are likely to be. The higher the inter-object class coupling, the more rigorous the testing needs to be." [6]. There is no target number and developers should judge each project on its own. The CBO metric is defined on class level.

- Coupling Factor (CF) - Denotes the ratio of actual couplings in relation to the total amount of possible couplings. A couple is defined as a class that calls a function in another class or when it accesses a variable in a class other than itself. Coupling is subject to directionality; When class A calls a function in class B, only class A is considered to be coupled with class B. A higher coupling factor means higher complexity and lower encapsulation, but coupling is necessary for any project to function. Again, there is no magic number, so development teams should decide for each project what percentage is acceptable. The CF metric is defined on system level.
- Afferent Coupling (CA) - Denotes the amount of incoming coupling links for each class. This metric should not be a benchmark on its own, but rather in combination with individual quality factors of the class. Many classes with high afferent coupling could mean insufficient segregation of information encapsulation. Having only one class with a high afferent coupling could indicate the existence of a "God-class" in the system; One class on which many other classes rely, creating a single-point of failure location in the system. This metric is defined on class level, but could be aggregated to form a metric on package level or even on system level.
- Efferent Coupling (CE) - Denotes for each class the amount of outgoing coupling links. The CE metric is defined on the same level(s) as the CA metric.

Cohesion:

- Lack of Cohesion in Methods (LCOM) - Denotes the cooperation of methods in a class. Currently there are four alternatives specified, but the original version was suggested by Chidamber and Kemerer in 1994. It is specifically formulated as follows:

$$\begin{aligned} LCOM = |P| - |Q| \qquad & \text{if } |P| > |Q| \\ = 0 \qquad & \text{Otherwise} \end{aligned}$$

  Where P represents the set of methods that do not share any instance variables with each other. Q denotes the set of methods where some instance variables are shared. The higher the LCOM mark, the higher the null intersections in the class, and thus the more estranged the methods are from each other. [6]. Again, no magic number is known because every project should have its own equilibrium for the LCOM metric. The LCOM metric is defined on class level.

Size and Complexity:

- Lines of Code (LOC) - Simply defined as being the number of non-white, non-comment lines of source code. Also called SLOC, for Source Lines of Code or NCLOC for Non Commenting Lines Of Code. This metric is included to ensure people can make informed decisions with respect to other metrics. It is important to have some idea of the size of a project. This metric can be defined on any level.
- Cyclomatic Complexity Number (CCN) - This metric has grown to be the *de facto* standard to define software complexity. Sometimes named McCabe's

Complexity after its inventor Thomas J. McCabe who invented this metric in 1976[26]. It is important to know because it indicates the degree to which the source code can be perceived as too complex. It is also an indicator for the amount of effort that should be put into testing. The Cyclomatic Complexity counts the number of decision paths in a particular method, this can be calculated when projecting each method as a graph. The Cyclomatic Complexity is then defined as:

$$v(G) = e - n + 2 * p$$

Where e is the amount of edges in the graph, n the amount of vertices and p the number of connected components. The mathematical approach is included for clarity purposes only since it can be calculated fairly easily. Since the Cyclomatic Complexity metric was invented in a period where procedural programming was most common, Object-Oriented decision paths were not included in the original design. New static analysis tools count occurences of any of the following list of programming events to calculate the Cyclomatic Complexity:

- ?
- ||
- &&
- or
- and
- xor
- case
- catch
- if
- elseif
- for
- foreach
- while

The Cyclomatic Complexity Number is defined on method level, but can easily be summed to form the CCN for a class, file, package or system. It is hard to define a target number for the CCN. The original paper uses 10 as threshold [26], but other sources state that 10 is an unrealistic number for current Object Oriented programming languages[24]. Because of this we use the originally proposed threshold of 10.

- Weighted Methods per Class (WMC) - Another metric defined by Chidamber and Kemerer in their 1994 paper "A Metrics Suite for Object-Oriented Design" [6]. Defined as:

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

Where n is the amount of methods in the class, and $c_1, \ldots, c_n$ is the complexity of the methods. This metric is important because it gives insight in the

complexity on a more abstract level, per class instead of per method. There is no target number but this metric helps identify classes with disproportionate complexity.

- Depth of Inheritance Tree (DIT) - Defined as the distance of the class to the root of the inheritance tree. The higher this number, the more complex the system is thought to be. As Chidamber and Kemerer describe it in their 1994 paper: "DIT is a measure of how many ancestor classes can potentially affect this class"[6]. This is important, because it increases the chances of breaking functionality when customizing ancestor classes. Again, defining a target number is hard, but this should give some insights in the design and architecture of the system. This metric is defined on the system level.

Inheritance:

- Number of Children (NOC) - Denotes the amount of classes affected when this class is changed. Only direct descendants are counted, no second or later generations. [6]. This metric is related to the Depth of Inheritance Tree metric. NOC is important to know on a more abstract level to see how many classes are dependent on some classes. This metric is defined on class level. No exact target number exists because it is dependent on the project.

2.4 Remarks

Due to tooling infrastructure (Chapter 3), not all metrics we would like to have, can actually be gathered. We have selected alternative metrics that differ as little as possible from the ones described above. When no alternative metrics could be found, others were selected based on their existance in the same area as described in 2.3.

Finally, we have to mention that software quality is something subjective. One can deliver the same product to multiple clients, but they will not be equally happy with it. This is due to the fact that software quality entails more than just code quality. It should always be seen in a certain context, and improving software quality means that it should always have some reference point.

Chapter 3

Tooling Infrastructure

Since awareness is a subjective term, we need to map it to some measureable value in order to make some careful (objective) statements. It is not the objective of this research to measure the amount of awareness, but rather the difference in software quality the awareness causes. It is hard to prove that there is a causal relationship between the two, but an attempt is made here to prove a loose connection between awareness of software quality and software quality itself. And for that we need to have insight in both.

To conduct this research some requirements and constraints have to be taken into account. We rely on other applications and tooling to gather data. The requirements to stimulate or to create this awareness in software quality are discussed in section 3.1. Section 3.2 is about Continuous Integration which we need as a foundation for our software quality platform. On top of the Continuous Integration foundation is a layer of source code analysis tools, which are discussed in section 3.3. On top lays Sonar, a metrics aggregation tool. This is discussed in section 3.4.

3.1 Requirements

Before one is able to analyze and compare any data, some tooling is necessary to retrieve such data. This section will describe the requirements for such tooling. We need tooling that is able to retrieve certain metrics from existing source code. This is easier said than done. To accomplish this, we need to make a distinction between a centralized way of compiling the complete source code and the actual retrieving of metrics. The centralized compiling of the complete source code is done by a Continuous Integration (CI) server, otherwise known as build server. The details of this build server are discussed in section 3.2. This build server is only meant to act as a centralized compiling mechanism. There is still no metric data available. Fortunately most CI servers can be extended to run additional source code analysis tools, which extract metric data from the source code. These additional analysis tools are discussed in section 3.3. Metric data is now available, but because several independent, non-collaborative analysis tools are used, it still is scattered over different files and

stored in heterogenic ways. The results of these analysis tools are aggregated and combined by a software product named Sonar. Sonar is described in depth in section 3.4.

Having described the preceding tools, we have the ability to gather metric data and store it in a homogenic way. Via Sonar we are able to extract metric data again from its database. Because we do certain analyses over the metric data, we cannot just show the data via the Sonar web interface. A tool has been developed that filters out specific metric data. It does this by analyzing metadata from the metric data. More information about this specific tool will be discussed in chapter 4.

3.2 Continuous Integration

Continuous Integration is a very important step in the attempt to improve software quality by making developers aware of it. Continuous Integration can simply be described as being a continuous compilation tool. Developers should be familiar with this concept because it is part of almost every modern Integrated Development Environment. Because projects grow enormously and developers often just work on small parts of (extensions of) a project, the integration of all of these small parts should be tested as a whole. This is where CI suites come into place. CI suites give developers insight into current issues with the complete source code. Continuous Integration suites are specifically built for this functionality.

Several of these CI suites are available. Since these suites are all developed for the same purpose - providing an almost-instant software quality feedback mechanism - all of these suites share certain common functionality. Some also have exclusive features that others do not have. A typical functionality is that the Continuous Integration suite is coupled with one or multiple SCM server(s). This SCM server is polled periodically for updated source code. When updated source code has been detected, the source code will be copied to a local directory on the build server. Then it gets compiled or in some other way interpreted. This integration with SCMs, combined with the frequent check on renewed source code, plús the compilation of these sources are the basis for the concept of continuously building an entire software project.

Furthermore, the support in CI suites for source code languages has to be investigated. This is important because not every CI suite supports every so-called builder which are used to compile source code. Builders are discussed in section 3.2.3.

To be able to make a fair and well-informed decision it is important to compare these features. For future re-evaluations of this research it is also preferable to have a clear sight on what is possible at this moment with each of these suites. When we have this knowledge, we are able to choose the best matching suite for our situation. Comparisons in this section are partially based on several feature matrices that can be found in [35] and [34]. Updated information has been looked up on the publishers' websites. A selection based on apparent usage from scientific articles was made out

of several CI suites [3] [33]. The following CI tools are being considered to support this research and are thus compared in this chapter:

- Apache Continuum[1]
- Bamboo[2]
- Buildbot[3]
- CruiseControl[4]
- CruiseControl.NET[5]
- Hudson / Jenkins[6]
- QuickBuild[7]
- Team Foundation Server [8]
- Vulcan[9]

As already stated in the introduction of this chapter, some CI suites incorporate certain functionality while others lack this functionality. An example of functionality that only a handful CI suites have (out-of-the-box), is for instance the inclusion of specialized issue tracking capabilities or project management tools.

This section is further subdivided as follows. In the first subsection the key constraints for the choice of a Continuous Integration suite are described. In the second subsection a comparison of supported builders is given. A comparison of several supported Software Configuration Management tools is described in section 3.2.4. Next the supported Issue Tracking Systems are described. After this subsection a new one will describe miscellaneous possibilities these Continuous Integration tools might offer. A decision is made in the final subsection.

3.2.1 Constraints

Several constraints are applicable for this research. An important thing to keep in mind is the environment in which the research should be conducted. For this research we have been given a server with certain - possibly theoretical - limitations which we have to keep in mind. Also, KPMG develops software projects in only a few languages, thus making the support for these specific languages important. This subsection will describe these key constraints in several subsections on their own.

Licensing
Although many software packages can be used free of charge at first sight, the publishers may have set (additional) usage limitations in their license file when used

[1] http://continuum.apache.org
[2] http://www.atlassian.com/software/bamboo/overview
[3] http://trac.buildbot.net
[4] http://cruisecontrol.sourceforge.net
[5] http://www.cruisecontrolnet.org
[6] http://jenkins-ci.org
[7] http://www.pmease.com
[8] http://www.microsoft.com/visualstudio/en-us/products/2010-editions/team-foundation-server/overview
[9] http://code.google.com/p/vulcan

by corporate businesses. A constraint for our research is that the Continuous Integration suite can be used free of charge, even in a corporate business environment. Also, some licenses incorporate a section about the mandatory release of the software package when additional code has been written or adjustments were made to the original code. Since this research does not incorporate such source code changes, these license limitations do not have to be taken into account. Table 3.1 below gives a general comparison on proprietary licenses.

Table 3.1: Comparison of proprietary Continuous Integration suites as well as plugin support.

CI Suite	Proprietary	Plugin support
Apache Continuum	-	-
Bamboo	✓	✓
Buildbot	-	-
CruiseControl	-	✓
CruiseControl.NET	-	✓
Hudson / Jenkins	-	✓
QuickBuild	✓	✓
Team Foundation Server	✓	✓
Vulcan	-	-

Given the hard constraint that we would only like to use free software, we can already conclude that we cannot make use of Bamboo, QuickBuild and Team Foundation Server. Team Foundation Server is the exception here, because there is already a license present to use it. To give a complete comparison for a possible future reconsideration of dismissing proprietary software, the previously named Continuous Integration suites are still represented in the comparisons below.

Off-site storage
Due to company regulations, no data may be stored off-site (online for instance). For this reason, certain SCM servers have to be disregarded. These services are only offered online and can not be downloaded and installed on a company's own server.

3.2.2 Plugins

Since all the projects in our research all make either use of .NET, Java or PHP technology, the CI software should be able to build at least these source languages. Some of the investigated CI suites do not *natively* support all of these languages, but there is a solution for this. Some of these CI suites offer the possibility to write additional features, or plugins to be used from within the CI suite. Since this ability can change the key features of such CI suites, we have incorporated this notion in this section. Some additions that have been made are released as proprietary software and thus should be paid. These plugins are left untouched due to our limitation not to use

proprietary software. The support for plugins of each CI suite can be seen in table 3.1.

3.2.3 Builders

An important aspect of Continuous Integration suites is which builders are used to (among others) compile the source code. Builders can be explained as being the executables which read from buildscripts that define source dependencies and compilation order. They also describe which tools should be executed after compilation, for instance code analysis, testing tools and metrics tools. A few well known builders exist today. The most noteworthy are Make[10], Apache Ant[11], NAnt[12], Apache Maven[13], and Microsoft MsBuild[14]. Each of these have their own properties and strengths but also their limitations. This section is dedicated to describe these properties, strengths and limitations to be able to choose the best builder for our situation. Because Make was the first buildtool that was widely used, we use Make to demonstrate the use and possibilities of buildtools in general.

Make
Make can be described as being the first tool which enabled complex (C /C++) build routines to be combined in a single buildscript. A simple buildscript called a "Makefile" for Make, is given below.

```
# The variable CC denotes which compiler should be used.
CC=g++
# CFLAGS will be the options for the compiler.
CFLAGS=-c -Wall

all: hello

hello: hello.o
$(CC) hello.o -o hello

hello.o: hello.cpp
$(CC) $(CFLAGS) hello.cpp

clean:
rm -rf *.o hello
```

The shown buildscript is able to build an executable file, and is also able to remove all files except the original source files. As can be imagined, large buildscripts

[10] http://www.gnu.org/software/make/
[11] http://ant.apache.org/
[12] http://nant.sf.net/
[13] http://maven.apache.org
[14] http://msdn.microsoft.com/en-us/library/wea2sca5(v=vs.90).aspx

can be created that incorporate automatic testing of dependencies, or the execution of various analysis tools. This makes it much easier to execute certain tasks which otherwise would take many singular commands.

Ant
Apache Ant can be described as the Java equivalent of Make. While Make uses Makefile's with their own syntax, Ant uses XML to describe build targets. Ant was built in Java and was focused to support Java programs. Later support was added for numerous other tasks. Today Ant is still a very powerful builder that is able to execute compilers or interpreters for many programming languages.

NAnt
NAnt is almost the same as Ant, but it is written in C# and it is targeted towards usage of C# and other .Net languages. It also uses XML to describe the configuration for the build process.

Maven
Apache Maven is the spiritual successor to Apache Ant. Although they are different programs with separate code base, because they are made for the same purpose and because Maven is technically more advanced than Ant, it is believed Maven is the successor to Ant. The usage of Project Object Model files (abbreviated as POM-files) is a key factor of Maven that makes it the next iteration in the builder evolution. A POM-file is an XML-structured file that describes the name of a package, its version, a location to where it can be found, and possible other dependencies it relies on. Maven is able to read these POM-files and if needed download any missing dependencies (including their own POM-files with more dependencies) from a global repository. This greatly improves and eases the use of builders.

MsBuild
Microsoft's MsBuild is integrated in the Visual Studio IDE suite. It supports all of the .Net framework languages. It uses an XML-structured build file to describe compile order and also dependencies.

Builder support
The comparison of the CI suites under consideration with the support for the builders above result in the following table. For completeness the possibility to use standalone shell scripts is also included.

As can be seen from the table above many CI suites support most builders. This should not come as a surprise because this implicitly defines the supported source languages.

3.2.4 Software Configuration Management

Another important aspect of the CI suite is which SCM System is supported (again, without the need for proprietary plugins). The SCM systems which are being used at KPMG are Visual Source Safe and Subversion. If a CI suite can be extended by a

Table 3.2: Comparison of builders the different Continuous Integration suites support.

CI Suite	Ant	NAnt	Maven	MsBuild	Shell scripts
Apache Continuum	✓	-	✓	-	✓
Bamboo	✓	✓	✓	✓	✓
Buildbot	✓	✓	✓	✓	✓
CruiseControl	✓	✓	✓	✓	✓
CruiseControl.NET	-	✓	-	✓	-
Hudson / Jenkins	✓	✓	✓	✓	✓
QuickBuild	✓	✓	✓	✓	✓
Team Foundation Server	-	-	-	✓	-
Vulcan	✓	✓	✓	✓	✓

A comparison of Continuous Integration suites by supported build tools. Note that "Make" is absent from the table above, this is because it can be installed on any computer after which it can be executed through shell scripts. It is not incorporated in CI suites due to its age and the availability of newer and more feature-rich alternatives.

plugin that gives the possibility to use this SCM, we count that as a supported SCM system.

Table 3.3: Comparison of Continuous Integration suites with Software Configuration Management support.

CI Suite	Visual Source Safe	Subversion
Apache Continuum	✓	✓
Bamboo	-	✓
Buildbot	-	✓
CruiseControl	✓	✓
CruiseControl.NET	✓	✓
Hudson / Jenkins	✓	✓
QuickBuild	✓	✓
Team Foundation Server	-	-
Vulcan	-	✓

A comparison for Continuous Integration suites with SCM support for Visual Source Safe and Subversion.

3.2.5 Issue Trackers

Several tools exist today that are used to keep track of bugs in software systems. It would be pleasant if the CI suite would support the Issue Tracking System that is used at KPMG, which is Jira. At table 3.4 a list of supported Issue Tracking Systems

per Continuous Integration suite is given. This includes the use of natively supported ITSs as well as ITSs which are only supported with the additional use of plugins. The letters that represent the different Issue Tracking Systems are explained below the table.

Table 3.4: Comparison of Continuous Integration suites with Issue-Tracking Systems support. [36]

CI Suite	B	G	J	L	M	R	T
Apache Continuum	-	-	-	-	-	-	-
Bamboo	-	-	✓	-	-	-	-
Buildbot	-	-	-	-	-	-	-
CruiseControl	-	-	-	-	-	-	-
CruiseControl.NET	-	-	-	-	-	-	-
Hudson / Jenkins	✓	✓	✓	-	✓	-	✓
QuickBuild	✓	-	✓	-	-	✓	✓
Team Foundation Server	-	-	-	-	-	-	-
Vulcan	✓	✓	✓	✓	✓	✓	✓

- **B**ugzilla[15]
- **G**oogle Code[16]
- **J**ira[17]
- **L**aunchpad[18]
- **M**antis[19]
- **R**edmine[20]
- **T**rac[21]

It has to be said that a couple of these SCMs have to be discarded because they only offer online storage, or off-site storage. This is not permitted due to company regulations.

3.2.6 Tool of choice

When properties of different CI tools are compared we can conclude the following:

Some Continuous Integration servers have to be discarded because they do not satisfy the constraints we set ourselves for this research.

All CI suites under consideration were able to build source code both at predefined timeslots and whenever a change in SCM was detected. A discrepancy can be found in the amount of supported SCMs. Of course, not all CI tools supported

[15] http://www.bugzilla.org
[16] http://code.google.com
[17] http://www.atlassian.com/software/jira
[18] https://launchpad.net
[19] http://www.mantisbt.org
[20] http://www.redmine.org
[21] http://trac.edgewall.org

every build tool, some supported “Ant” and “Maven”, while others only supported “MsBuild”.

Given the fact that most CI suites are very extensible, one can choose to make use of a certain CI suite which has a lot of plugins, or one can choose to make the preferred plugins themselves.

Integration with Issue Tracking Systems is preferred, but since only one is in use at KPMG, only CI suites that support it should be further considered.

Based on the considerations of above properties we have decided to make use of the Jenkins Continuous Integration Suite.

3.3 Source Code Analysis Tools

While the major software tools described above are for a great deal in accordance with our requirements, there are still some smaller - but equally important - tools which help to maintain or improve software quality as well. These source code analysis tools are described below. Afterwards, an important subsection is given about Sonar, an umbrella analysis tool that combines the gathered data from the different static and dynamic analysis tools. It then stores this data in a database which can be reached through a web site and through a web service API. It also makes this information clear by providing dashboards that can be viewed in a regular web browser. A full analysis of Sonar is given in section 3.4.

Before going through each analysis tool, a distinction between static analysis tools and dynamic analysis tools has to be made first.

Static analysis tools are tools that analyze source code without actually executing it. Tools such as these analyze for instance the design of a software product, or the complexity of its implementation. Gathering the amount of lines of code, or the percentage of comment lines with respect to the total amount of lines of code is also typical information.

Dynamic analysis tools are tools that analyze the quality of the software while actually executing it. Other information will be gathered when running the software. Memory usage (memory footprint, memory leaks), memory input / output ratio per file or per class are all metrics that can be found with dynamic analysis. Test coverage is also something that is determined with dynamic analysis.

3.3.1 Checkstyle / CodeSniffer

Checkstyle is a tool that verifies if the source files provided adhere to preselected coding standards. It can be incorporated in the Jenkins pipeline for easy execution. The coding standard to which Checkstyle checks the source files can be defined by the user. Every standard is made up of many different rules. Despite having a predefined ruleset, every company should consider if they would like to have their own ruleset. CodeSniffer[22] is the PHP version of Checkstyle, which only supports Java.

[22] http://pear.php.net/PHP_CodeSniffer

Checkstyle or CodeSniffer report their findings in a large XML file. The XML file is filled with coding standard violation entries like:

```
<error line="ww" column="xx" severity="error" message="Line
indented incorrectly; expected at least yy spaces, found
zz" source="..." />

<error line="ww" column="xx" severity="error" message="Method
name '....' is not in camel caps format" source="..." />
```

As one can see, the errors Checkstyle or CodeSniffer return are not meant to tell a developer that his code does not work. It does however tell the developer that his code is not in line with company regulations and for the sake of maintainability this should be adjusted.

3.3.2 PDepend

PDepend[23] is short for PHP_Depend. It is based on the popular JDepend[24] static analysis tool for Java. It is just like JDepend, Checkstyle and also CodeSniffer a static analysis tool. It is able to calculate the Cyclomatic Complexity Number, the amount of possible execution paths (NPath Complexity) and more for all possible scopes of a program. This package is very helpful for the aggregation of metric data into Sonar.

PDepend calculates for every method, class, file and package the total lines of code, the non-commenting lines of code (ncloc), the comment lines of code (cloc), the effective lines of code (eloc) (denoted by the amount of lines of code which have one or more actual statements), the logical lines of code (lloc). The following table with pseudocode presents an overview:

Table 3.5: Definition of lines of code. [23]

Source code line	LOC	eLOC	lLOC	Comment	Blank
if (x<10) // test range	✓	✓	-	✓	-
{	✓	-	-	-	-
// update y coordinate	-	-	-	✓	-
	-	-	-	-	✓
y = x + 1;	✓	✓	✓	-	-
}	✓	-	-	-	-

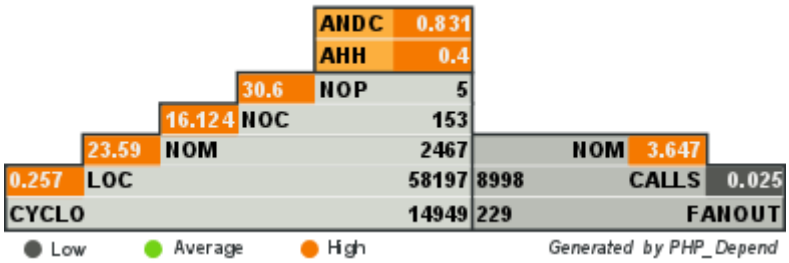
PDepend is also able to calculate many more metrics for instance the Cyclomatic Complexity Number (CCN, otherwise known as McCabe's Complexity), Afferent Coupling (ca), Efferent Coupling (ce), Coupling Between Object Classes (cbo), Depth of Inheritance Tree (dit), NPath Complexity, and more. Unfortunately, Sonar is only

[23] http://pdepend.org

[24] http://clarkware.com/software/JDepend.html

able to retrieve the Cyclomatic Complexity Number from source files. We have asked the developers of Sonar to update their retrieval mechanism and this is set to be fixed for the next release of Sonar.

PDepend is also able to generate a so-called overview pyramid, in which a lot of implicit information is stored. A sample of such an overview pyramid is given below.



The overview pyramid is not included in this research for the reason that much of the implicitly stored information, is difficult to extract with a single view.

3.3.3 PMD / PHPMD

PMD is not an abbreviation, but the community came up with several catchy "backronyms" like *Project Meets Deadline* or *Project Mess Detector*. PMD / PHPMD scans Java / PHP source files to find potential sub-optimal code. For clarity, this is subdivided into four categories, each with their own rules. The four categories are:

- **Code Size** Coding size rules are not violations literaly about the actual size of the source code, but rather about excessive method, class and parameter lengths, excessive Cyclomatic and NPath Complexity, excessive amount of methods, fields, etc.

- **Design** Rules about design include excessive amount of children (noc), depth of inheritance (dit), and coupling between object classes (cbo). It also warns the developer when exit expressions are used, which cannot be tested.

- **Naming** Naming rules can be placed in the same category as Checkstyle. They do not detect any programming errors, but they detect bad habits. The rules check for variablenames and methodnames that are either too long or too short.

- **Unused Code** This category also does not detect any programming errors, but it really keeps the code clean and clear. It raises warnings when fields, methods or local variables are not being used.

Just like Checkstyle, the patterns and rules that need to be checked can be fully configured. For ease of use, PMD has been made available for many IDEs via several plugins.

3.3.4 PMD-CPD

CPD is an add-on for the PMD package above. It stands for Copy-Paste Detection and detects if parts of the source code have been copied and pasted elsewhere. Consensus has not been reached yet whether having copied pieces of source code is bad or not[15]; Conservatives say no code must be copied because in case of a bug it has to be fixed at multiple locations which makes it error-prone[20][16], others say copied text has already been checked multiple times making sure it does exactly what needs to be done[30]. Because copied source code might also give an overal sense of the program architecture, the copy/paste detection is included nonetheless.

3.3.5 JUnit

JUnit is used as unit test framework. Combined with test coverage tools such as Cobertura[25], Clover[26] or Emma[27] it is not only possible to test the software, but also on a more abstract level, it is also possible to check the source code coverage as the unit tests have been executed. This also gives a general sense about the percentage of source code statements that were executed during the unit tests.

3.3.6 NUnit

NUnit[28] is a .Net unit test framework and coverage tracker. Although developers have the capability to use unit testing directly from Visual Studio, NUnit enables us to incorporate this in our Jenkins setup. NUnit is already being used by a small .Net development team within KPMG in combination with a Team Foundation Server setup.

3.3.7 FxCop

FxCop[29] is a static analysis tool from Microsoft which started as a standalone program but has already made its way into the Visual Studio IDE itself. The underlying mechanism works the same way as Checkstyle; It has rules to which the source code should adhere to. Although being directly incorporated in the Visual Studio IDE, it also has a standalone graphical user interface. In this graphical user interface the activation of rules can be configured and rules can be created or deleted. To be able to run FxCop with our Jenkins solution, we need a commandline utility which Jenkins is able to execute. Fortunately, FxCop comes standard both with a graphical user interface, as well as a commandline tool. We can execute this commandline tool directly from Jenkins.

[25] http://cobertura.sourceforge.net
[26] http://www.atlassian.com/software/clover
[27] http://emma.sourceforge.net
[28] http://www.nunit.org
[29] http://http://msdn.microsoft.com/en-us/library/bb429476(v=vs.80).aspx

For the problem of having to build a static analysis program for each of the .Net languages, Microsoft invented with a few other companies an intermediate language, which they call CLI, or Common Language Infrastructure. For each of the .Net languages, Microsoft built a compiler to CLI. This is relevant for our research, because FxCop does not check the source languages, but the transformed source code of this intermediate language CLI. Because CLI does not make use of any of the high level language constructions of C# for instance, this will result in warnings and errors the user might not be responsible for. FxCop has a long list of sections for which it looks for possible errors. This list includes performance, security, reliability, portability, maintainability, etc. Because Microsoft invented both the .Net framework, and FxCop, it is able to check the - albeit transformed - source code with respect to its intended use.

3.3.8 StyleCop

StyleCop[30] is a coding standard tool for .Net languages. StyleCop takes care of the compliance to styling rules from the coding standards, just like Checkstyle.

3.4 Sonar

Sonar is used as an aggregation tool to gather metrics that have been calculated by all the tools mentioned above. Because every tool uses its own bug or rule violation qualification or categorization, Sonar tries to level these different categorizations by using one classification scale. All violations are classified in one of several categories. These are named BLOCKER, CRITICAL, MAJOR, MINOR, and INFO. An example of a rule with a 'CRITICAL' priority is: "Class cyclomatic complexity exceed maximum". As can be imagined, this might be a real threat to the maintainability of such a program. As such, it is an important characteristic for the global software quality. On the opposite side of such an important rule, an example of a rule with an 'INFO' priority: "Scope indent incorrect". This last example shows that also rules can be applied that check the layout of source code (coding standards). As one can imagine from this example, this rule is not considered very important for every company.

Sonar reads the produced files from the analysis tools and qualifies each violation with respect to each other. After reading every violation and qualifying it with respect to all other violations, it is stored in a homogenic way in a database. Although Sonar offers a rich web front-end where one is able to lose oneself, it also offers a web service API. This web service API will be used in chapter 4 to discuss the feedback mechanism to be able to stimulate the awareness among developers in a targeted way.

[30] http://stylecop.codeplex.com

3.4.1 Sonar Web Service

The main idea behind the feedback mechanism is to gather as much metric data as possible and to plot this in a graph so developers will get a general sense of the software quality. Much of the data Sonar stores in its database is accessible through their web service API. The main web service calls we use to be able to access the data we want is done via a request to:

```
public function resourcesRequest() {
$resources = $this->rest->get('resources/', array(), 'json');
}
```

The pseudocode above will return the resources (projects) that Sonar has been analyzing. With this information, we can get detailed violations information about this resource by requesting:

```
public function violationsRequest($resourceID) {
$violations = $this->rest->get('violations/', array(
'resource' => $resourceID,
'depth' => '-1',
'verbose' => 'true'
), 'json');
}
```

This will return all current violations of the resource which' ID we entered as parameter. Although this might seem to be information we would like to return as feedback to developers, it is too detailed to give them a general sense of the current software quality.

To be able to give that general sense of the software quality, we also need to have a track of history of the same information. Because old information is not interesting to see, the information should be relatively new, but it should not go too far back in time. A nice compromise was to limit the information to a maximum of thirty days. To request all the metric data of all builds up to and including thirty days back is done by the following pseudocode:

```
public function timemachineRequest($resourceID, $metrics) {
$violations = $this->rest->get('timemachine/', array(
'resource' => $resourceID,
'metrics' => $metrics,
'fromDateTime' => Date(date('c', strtotime('-30 days'))),
'toDateTime' => Date(date('c'))
), 'json');
}
```

This will return all historic data for the selected metrics for the last thirty days. With this information we are able to construct graphs that show the development for each

metric. We are also able to calculate the average of all data values for every metric, which is also incorporated.

3.4.2 Sonar Violations

As already stated in chapter 2, software metrics are used to quantify software quality. A lot of research has been done to find useful metrics. As can be imagined, unfortunately not all metrics can be retrieved by the analysis tools of our choice. The solution is as simple as effective, we look for metrics that the analysis tools are able to extract from the source code, and we select the ones we selected earlier in section 2.3. After that, we cross check both lists of metrics.

In the list below both metrics as well as violations can be found. Metric numbers should be interpreted as just numbers. Violations should be explained a bit more here. As already discussed in the previous section, violations are bound to metrics. For instance, there is a metric that calculates the Cyclomatic Complexity of a class. This will result in a certain metric value. There is also a rule bound to this metric, that tests if the Cyclomatic Complexity of a class never rises above a certain threshold. If the threshold is passed, the rule is violated, which results in an increase in the correct violation category according to the rule severity.

The following metrics and violations can be extracted using Sonar extended with the analysis tools described above.

- **classes** is the amount of classes in the system. This is only used to give a sense of size.
- **ce** or Efferent Coupling, denotes the amount of outgoing dependencies a certain class has.
- **ca** or Afferent Coupling, denotes the amount of incoming dependencies for a certain class.
- **lcom4** or Lack of Cohesion of Methods. Already discussed in depth in chapter 2.3.
- **ncloc** or number of lines of code. This is defined as the number of physical number of lines of code - number of blank lines - number of comment lines - number of header file comments - commented-out lines of code.
- **complexity** McCabe's Cyclomatic Complexity number of methods. Discussed in depth in chapter 2.3.
- **class_complexity** denotes the average complexity per class.
- **dit** or Depth of Inheritance Tree. Also discussed in chapter 2.3.
- **violations** denotes the amount of times a rule has been violated.
- **blocker_violations** denotes the amount of time a rule with a "BLOCKER" severity has been violated.
- **critical_violations** denotes the amount of time a rule with a "CRITICAL" severity has been violated.
- **major_violations** denotes the amount of time a rule with a "MAJOR" severity has been violated.

- **minor_violations** denotes the amount of time a rule with a "MINOR" severity has been violated.
- **info_violations** denotes the amount of time a rule with a "INFO" severity has been violated.
- **duplicated_lines_density** denotes the percentage of duplicated lines of code with respect to all physical lines of code.
- **noc** or Number of Children denotes the amount of direct and indirect descendants of this class.

3.5 Miscellaneous

3.5.1 Hardware

We need a reasonably fast server that is able to run a few software tools described in this chapter. Hardware-wise we have an Intel Quad Core processor available in our server with 4GB of memory. This machine is equipped with a harddrive of 160GB of storage memory. This will most likely be more than sufficient for all of our needs. The server is running Microsoft Windows 7 Enterprise Edition.

3.5.2 All-in-one packages

Configuring and especially keeping the tools above up-to-date will require a lot of attention. Other possibilities exist that help in reducing these necessary efforts. Both Coveros[31] and CIFactory[32] offer all-in-one packages. The upside of using these all-in-one packages is that one does not need to install the described tools individually. They are also pre-configured to be able to work together. The downside of this approach is that one still needs to configure the individual packages after updating them, and that often older packages are being used. Updating these individual packages is unfortunately also not an easy task.

[31] *http://www.coveros.com/research/research_stack.php*
[32] *http://cifactory.org/*

Chapter 4

Research Approach

Previous chapters gave a good insight in software quality and discussed the tooling infrastructure which we have at our disposal. In this chapter we explain how we try to improve software quality with the aforementioned tools. As the following quote illustrates, the information itself is equally important as the way it is presented. "Proper interpretation of metric values is essential to characterize, evaluate and improve the design of software systems." [10].

As we try to reach the improvement in software quality by stimulating awareness, we first explain the term awareness itself. This is done in section 4.1. The next section (4.2) explains which information we are able to gather and how we are going to transform it in order to get a general sense of the current software quality.

4.1 Awareness

Awareness is literally "having knowledge of", according to several dictionaries[1,2,3,4]. However, awareness itself has never been the goal of a campaign, rather used as a means to an end. It is not any different in our research. Awareness is used in our research as a means to put extra emphasis on software quality during software development. This is done by giving developers a general sense of the current state of the software quality. You could compare this to awareness programmes to make people (more) aware of signs of breast cancer or AIDS. The targeted group does not (nor need to!) know anything about carcinomas or how cells react to AIDS. This is all too detailed knowledge for a campaign to be effective[28] [27] [2]. Our research is to find out if the same holds for software quality. Although our target group (software developers) knows (much) more about software quality than people in general know about carcinomas or AIDS, it does not do any harm to raise extra awareness on an important subject.

[1] http://www.thefreedictionary.com/awareness

[2] http://dictionary.reference.com/browse/awareness

[3] http://www.macmillandictionary.com/dictionary/british/awareness

[4] http://oald8.oxfordlearnersdictionaries.com/dictionary/awareness

While awareness in itself is not our focus, the effect it causes is our main interest. As such, we not only try to measure the outcome, but we should also document the way in which we try to raise this awareness. When dates and metrics data is documented, a possible connection between the two variables could be shown; A possible trend (relation) could be identified between the beginning of raising awareness and an actual difference in metric values. This can be compared to an advertisement campaign where its result is being tracked. We are however unable to state that the awareness programme and the improvement of internal software quality are causally connected because we need more identical environments each with one different variable (project itself, project duration, programming language, amount of team members, team composition, the way in which the awareness is raised, etc.) to prove this.

4.2 Data

The idea behind raising awareness in order to improve software quality is to first gather data with different tools discussed in chapter 3. Because this data is very fine grained and we want to try to create or stimulate (general) awareness towards software quality, we cannot just return this data "as is" to the development team. We have to transform this data into something that is able to give the development team a general sense. This will be discussed in section 4.2.2. First let us describe the projects with which we are working.

4.2.1 Projects

An important fact to know in this research is the different projects that are being used. Below, for each project the most important details are given. Important details are for instance the project duration, project size, project team size, etc. Each of the projects are being discussed in their own subsection below.

One observation that may stand out is the fact that team composition changes quite often over time. This is because a senior developer is not always necessary and may be assigned to another project.

Project A
Project A involves a tool that can be used to build and share teaching materials. It is implemented as a web application that runs on PHP technology. The project was not originally developed by KPMG, but was taken into maintenance in August 2010. Until early July 2012 KPMG did change-requests with new features or changed behavior. Many different people have worked on it, but never more than five at the same time. The project team's composition changed quite a few times, but experienced developers were always part of this development team. The source code is around 15.000 LOC, which includes unit test files.

Project B
Project B was designed to give civilians the ability to measure the performance of

their municipal services. The project was started at the end of February 2012. The project is a web application with all kinds of web services attached to it at the backend and middleware running in between. The web application itself is built with PHP technology and the middleware and web services are both built with Java. The total size is around 6.000 LOC. This project was developed by four developers.

Project C
Project C is meant to offer municipalities an architecture to improve their digital serviceability to their residents. This is again a web-based application built with PHP technology and consists of about 2.500 LOC. Three developers have worked on this project so far.

4.2.2 Inferred Data

Sonar acts as an umbrella application that runs the static analysis tools described in chapter 3 and aggregates data from the generated logfiles and stores them in a normalized way in its database. To stay as generic as possible to be able to support as much programming languages as possible, Sonar uses generic rulesets, with generic violations. Some static analysis tools report violations only because the default coding standards are not followed. Because every company has its own custom coding standards and because not every project lends itself for the use of generic rulesets, a new ruleset has to be made for every company, and possibly even for every project. This will consume a lot of time and is therefore an expense that cannot be justified for every company or project.

Because Sonar only implements metrics that can be gathered for the majority of source code languages, not all metrics described in section 2.3 can be obtained. Therefore, the metrics described in section 3.4.2 are used as alternative.

Developers have multiple projects they work on during a workweek. The associated context switching can cause developers to have problems getting familiar with each project. It is therefore essential that receiving feedback on the work they and their peers have done takes no more than a few moments of their time. Since the static analysis tools described in section 3.3 generate a lot of data, giving developers access to this unprocessed data will get them lost in the amount of information. Giving developers access to Sonar itself, and thus giving them access to many graphs and interesting statistics, will not give developers a general sense of the internal software quality in one view. To be able to give developers a general sense of the software they are building, in the least amount of time, the amount of information has to be drastically filtered.

Figure 4.1 shows the schematic process of the way in which the feedback report is generated. Because a compromise had to be made between having too much information (giving developers access to Sonar itself) and not having the ability to get more information than having the graphs, the feedback mechanism knows two functionalities. The second functionality besides the graphs is the ability to retrieve all violations. A filter to only view violations of a certain category is also available. The graph functionality is described below.

First, we must filter the information we would like to return as feedback. In section 3.4.2 we selected a subset of metrics out of the entire set of metrics. Additionally, Sonar also gives the ability to return all violations, that include coding standard violations. We chose not to include this information at the most detailed granularity, but rather at a more abstract level. We included the total amount of violations, and the amount of violations in each of the five categories separately. With this, we are not only able to detect changes in the total amount of violations, but also if there was a large discrepancy in the amount of violation in a particular category. Because Sonar is an umbrella tool capturing all kinds of quality-related information, we also did not include build results and unit test results on purpose.

Second, we want to include only "local" data. With that we mean currently relevant data. When in the past an outlier has been detected in some metric, we assume this has been corrected by the development team in one of the next builds [18]. If not, the development team agreed on it not being a situation that has to be fixed. This is why we chose to include only builds from the last 30 days into our feedback mechanism.

Thirdly, we filter the amount of information developers get to see from these metrics. The idea behind this is to filter out as much irrelevant data as possible. We are looking for outliers in a certain metric pattern. Some metrics (e.g. the SLOC metric) should increase over time, whereas others (e.g. the comment lines density) should stay around the same value. It is unwanted behavior that regular patterns such as the given examples are included in our feedback mechanism.

To detect irregular metric behaviour we chose the standard deviation of past values. This because of its definition: The variation with respect to the average. The standard deviation is an excellent method to discover real outliers because of the confidence intervals surrounding the average. This way minor differences do not let the feedback mechanism be triggered unnecessarily. Since the selected metrics all have the tendency to stay around the same level, or slightly increase or decrease, we can use the standard deviation in combination with a normal distribution. This is depicted in figure 4.2. We consider a value being an outlier if it fails to fall within the 68% confidence interval, or one standard deviation.

Finally, we display these metrics in such a way, that a single glance at a graph will give an immediate general sense of the direction in which the metric is going. When all the graphs of metrics that have exceeded the standard deviation interval are shown in a single place, the development team should be able to get a general sense of how their project is coming along and which issues should be resolved. An example of such a graph is shown in figure 4.3.

An overview of the data being processed is depicted in figure 4.4.

4.2.3 Ways of giving feedback

The information that is to be returned, and with that, **RQ3.2**, is clear, but the way in which this information is returned (**RQ3.3**) is also a factor that has to be taken into account. Many options have been considered of which many were dropped again

because the working environment made implementation hard. Developers have a development machine separate from their corporate laptop, where their email client and office communicator are installed. The fact that there are two separate machines limits the possibilities we have to return the analyses back to the developers.

The most elegant and possibly best solution would be to have a special plugin for the IDE that would notify the developer when a new analysis was available. This solution would be the most elegant, because it would mean the least developer adaptability since only their IDE should be adjusted by incorporating some plugin. Secondly, it would be a best solution because it would notify developers in an active way. This would not be a sustainable solution because developers are free to choose their own integrated development environments. Currently, developers make use of either Visual Studio, Eclipse or Netbeans, but since the usage is not strictly limited to these three, more plugins would possibly have to be made.

Another solution would be an auto email system that would notify each developer when a build was finished. This way the developers again get the active feedback when something has changed, but due to the fact that the email clients are solely installed on the corporate laptops and not on the development machines - which means the screens are often switched off due to inactivity - the attention is not that often focused on the corporate laptop.

Yet another solution would be to use the office communicator to actively notify developers that a new analysis is available. Again, this solution has to be dismissed because developers are paying more attention to their development machine than to their corporate laptop. Another solution would be to build a website which developers could visit on their own initiative. This website would have the most recent data available a few minutes after a build has been initiated by a SCM checkin because the system needs some time to process the information. A drawback of this solution is that developers are not actively being reminded that a new analysis report is available, and they have to get out of their integrated development environment to obtain the processed information.

Although this solution has some drawbacks, we chose to implement the feedback mechanism in the form of a website because this was the most practical solution. Developers will not be notified from the fact that a new analysis report is ready. This is circumvented by attempting to incorporate it in their workflow. An email was sent to developers with an introduction for the feedback mechanism and how the information should be read. Developers were asked to visit the feedback mechanism 15 minutes after a checkin into the SCM was done to give the system an appropriate processing time. Implementation details can be found in the next section.

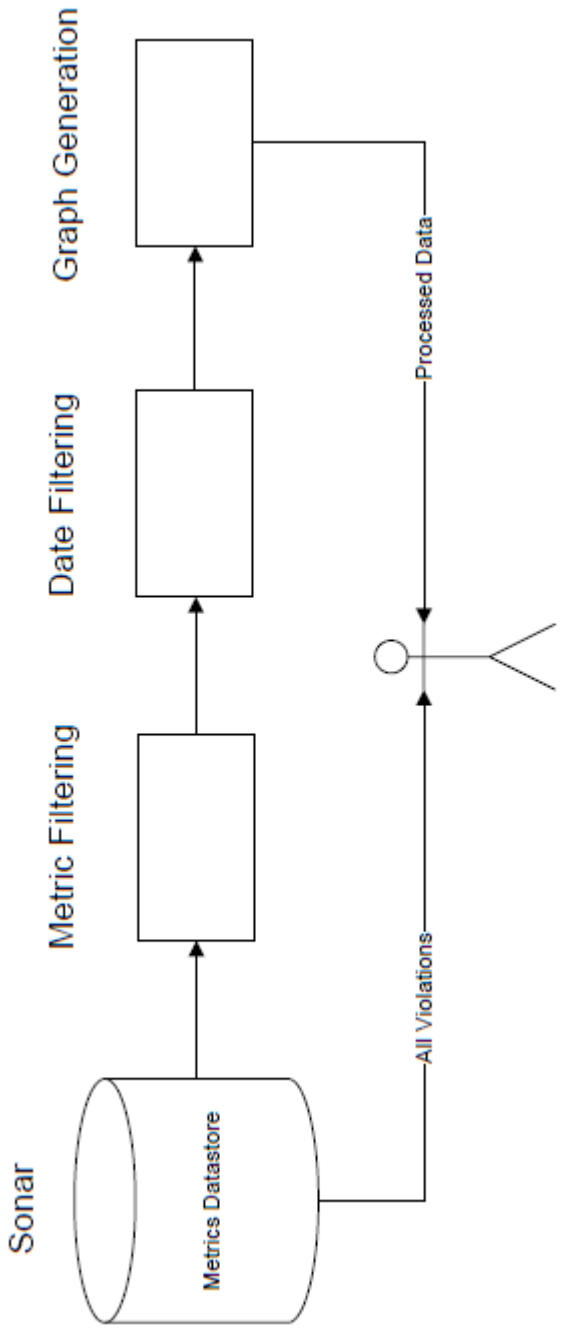


Figure 4.1: Schematic picture of the feedback mechanism internals.

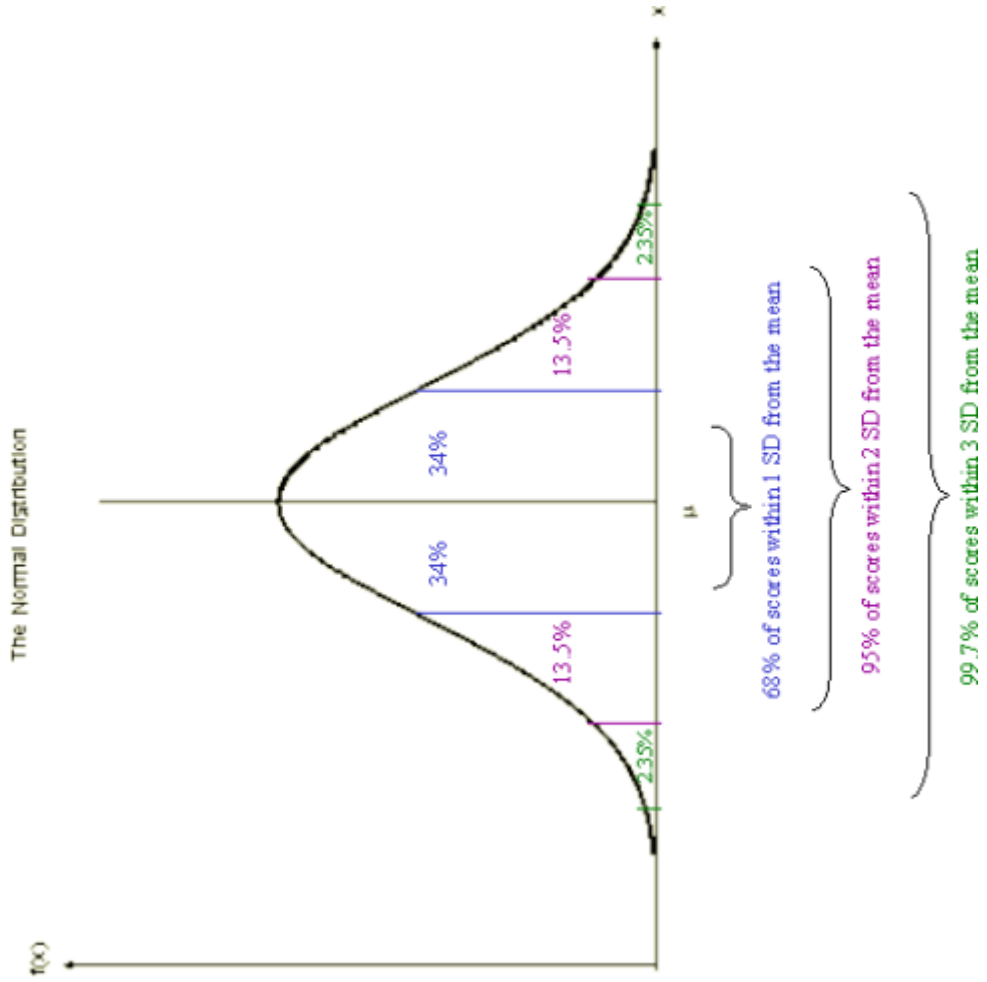


Figure 4.2: An image of the standard deviation of a normal distribution with confidence intervals.

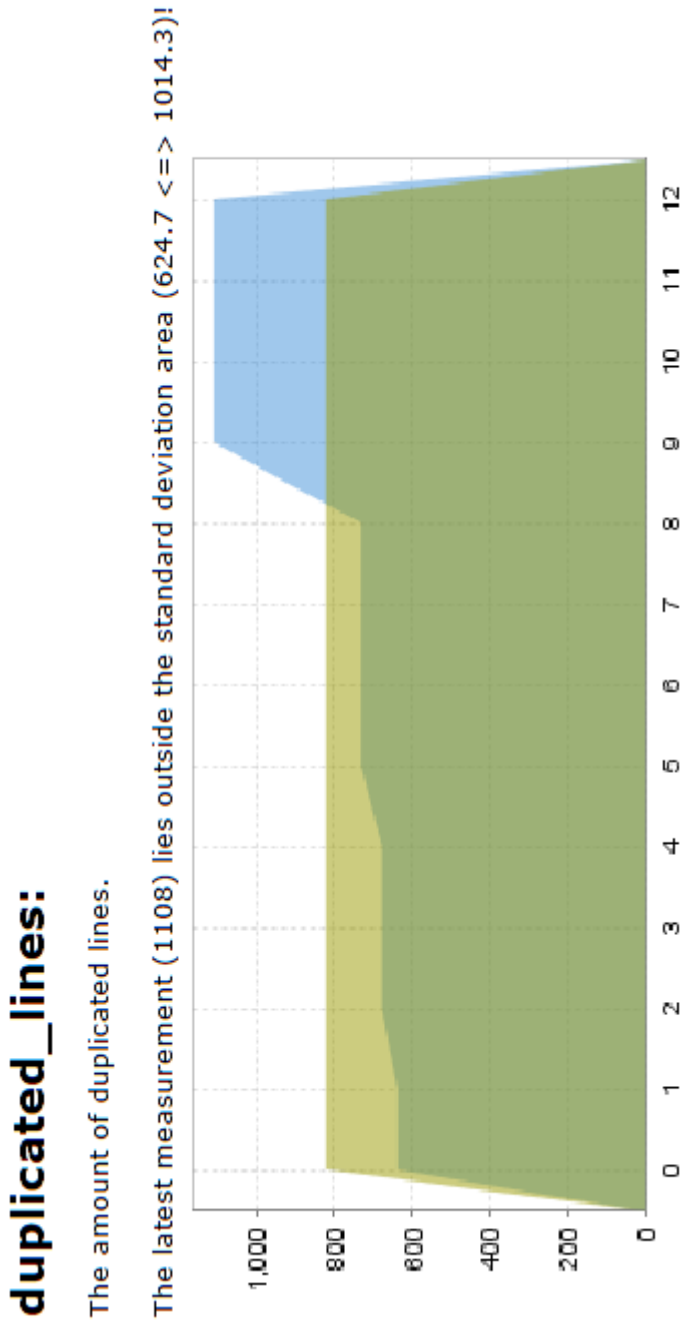


Figure 4.3: An example graph depicting the fact that a recent change in the amount of duplicated lines has been detected.

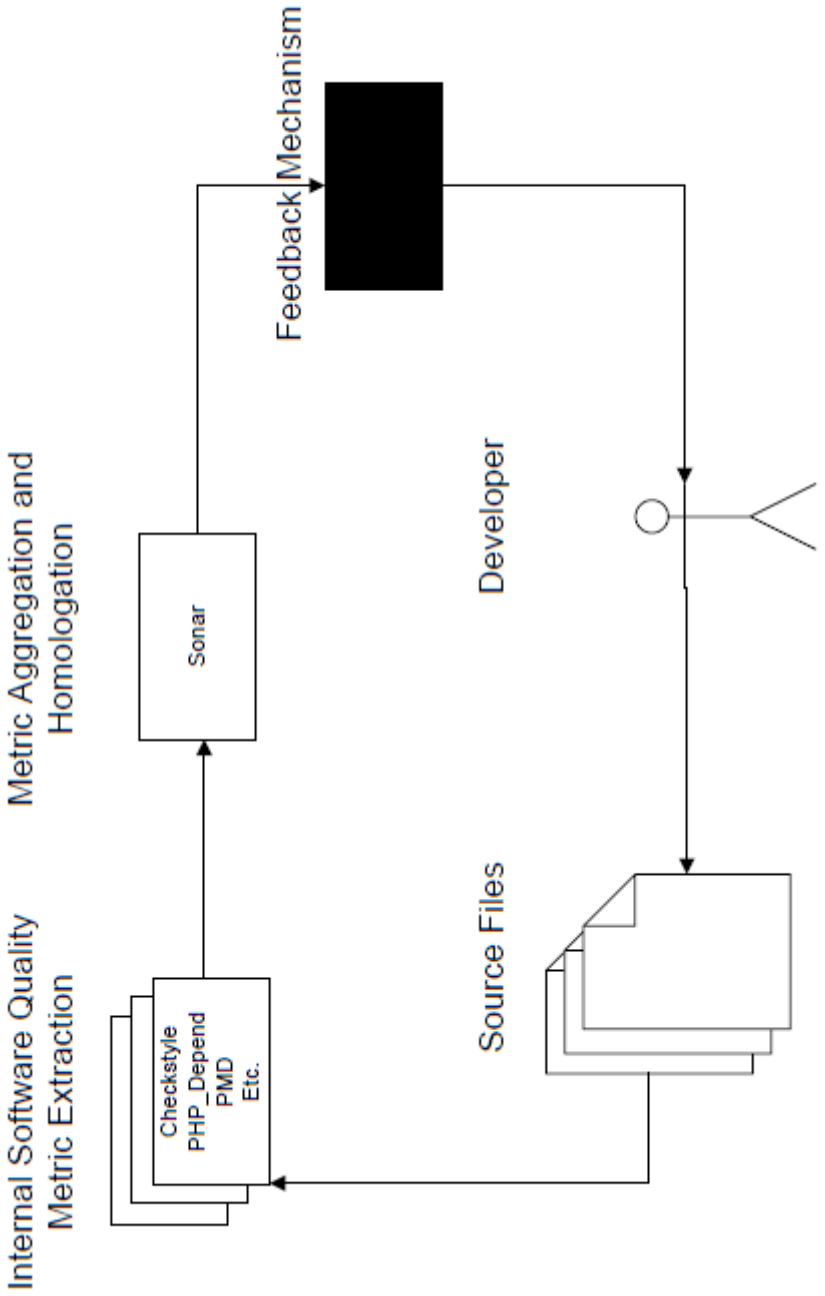


Figure 4.4: Schematic picture of the feedback mechanism as “black box” in the development process.

Chapter 5

Results

This chapter is meant to give the results of the experiments and also to put these results into perspective. After the introduction in 5.1, project A will be discussed in section 5.2, project B in section 5.3 and project C in section 5.4. A general reflection on the research with developer interviews is given in section 5.5. Points of improvement and threats to the validity of this research are given in section 5.6. Finally, some discussion material can be found in section 5.7.

5.1 Introduction

In this chapter the results of the experiments done for this research will be discussed. The reason why the environment is discussed extensively is because the results have to be seen in a clear manner.The results for each of the projects are all presented in the same format. First a small section with a description about the project is given. Project details as size, duration, used technology, project members, a history of how the project came about, and more are given. After that, regular development activities without the feedback mechanism will be described in another section. To obtain objective results, we first monitored each project a certain time before informing developers their project was included in this research. This way we obtained a baseline in which we could see the regular behavior of the metrics before influencing them. Observed trends will be discussed. Reasons for irregular behavior are given as well, if any could be discovered. Observations done after the feedback mechanism has been set to work and developers were able to react on it, will be given in the final section of each project. From the end-results multiple pecularities arose. Causes for these pecularities have been sought for in the interviews that were taken in the end period of the research with the responsible developers and will be mentioned directly in the sections below.

The information shown to developers was based on the data of checkins of the past 30 days. Because we are interested in knowing the effects the raised awareness has on the long term, we include graphs generated by Sonar of the entire duration of us monitoring the project. Each project also features four images with metrics vi-

sualizing the relative comparison they have with each other. The first image depicts the lines of code, the number of files and the number of classes to show the relative evolution. The second image shows the three most important violation severities, namely “blocker”, “critical” and “major”. The third image shows a visualization of the lines of code, and relatively compares them to the amount of cyclomatic complexity sum and the sum of all violations (not just the three major severities). The fourth and final image shows the lines of code in respect to the amount of methods and the percentage of duplicated code.

The tables belonging to these images are spread over two. The first table shows the general historic data and the second shows the distribution of the violations.

5.2 Project A

5.2.1 Project Description

Project A was the first project we incorporated in our tooling infrastructure. As already briefly mentioned in the previous chapter, project A is a sharing tool for teaching material. It enables anyone interested in teaching other people to build their own teaching materials which they can share with the world. For increased accessibility and ease of use, it is implemented as a web application that runs on PHP technology.

Project A was not originally developed by KPMG, but it was taken into maintenance in August 2010. This is very much of influence because developers work with other people’s source code. No matter if the developers doing the maintenance like or dislike the programming style of the source code, their job is to maintain it and deliver service for it. Sometimes this may become an issue; How should a developer implement some new functionality, in the same way as corresponding functionality was implemented, or design-wise the best way possible? It would make sense to reuse the same style, as it will make it easier as opposed to introducing a new way to implement matching functionality.

It has also a strong influence on the way the feedback mechanism will react. Since the feedback mechanism evaluates all source code, it will also evaluate source code which the developers responsible for maintenance did not write. This may result in lost interest in the feedback mechanism altogether, since maintenance developers have to look for comments and critique on their code, which may not even be written in their preferred style, but which may be intentionally written in a matching style for the existing code.

The project has its periods of intensive development, but it has also periods where no changes are made. Since the beginning of us monitoring the project, until early July 2012 KPMG did change-requests which consisted of added functionality or functionality that needed to be changed.

Information about the original development team is hard to obtain. Many source files do not contain comments with author information, and KPMG did not obtain the original source code via SCM access, but rather received a complete package

with complete source code. The development team of KPMG consisted of five team members or less, at any time. One project leader was assigned to this project, to guide developers and to communicate with the customer. The developers' task is to implement the changes as requested with a change request. This entails both resolving bugs and implementing new functionality. Their task is also to test their implementations themselves. No more than three developers have been working simultaneously on this project. The maintenance development team from KPMG changed quite frequently, since people are being assigned to other projects as well. It depends on the planning who gets the project assigned to. Work that has been done previously will be taken into account and developers already familiar with the project will be prioritized when developers are getting assigned to projects. There have never been more than five maintenance developers at the same time.

The source code showed signs of the original development team testing their software because unit tests were included. The entire source code is about 14.500 NCLOC, this includes unit test files.

Data is included from March 6th till July 3rd. As explained earlier, we start with a period not giving developers access to the feedback mechanism, in order to observe trends in regular development activities we refer to as a baseline. We began informing developers to use the feedback mechanism early May.

5.2.2 Observations Before Stimulating Awareness

Development on project A occurs only on client request. This is why there is no constant development activity for project A. Our results will therefore be strongly dependent on the amount of change-requests filed. Some change-requests involve bigger changes than others.

In table 5.1 we have included six snapshots that show some interesting data. The data between the first and second entry should be interpreted as regular way in which the project progresses.

Table 5.1: Table with historic data for project A

Metric	March 6	May 7	May 15	June 7	June 20	July 3
Lines of Code	11645	12555	13130	13844	14597	14453
Files	120	134	140	148	156	156
Classes	74	87	92	98	103	103
Methods	865	943	1002	1051	1098	1099
Duplicated Lines	952	730	1108	1126	1096	1096
Duplicated Lines (%)	5.2%	3.6%	5.2%	5.1%	4.7%	4.7%
Complexity	2565	2749	2884	3013	3208	3213
Complexity / Class	34.6	31.6	31.3	30.7	31.1	31.2
Complexity / File	21.4	20.5	20.6	20.4	20.6	20.6
Violations	801	818	841	235	253	253

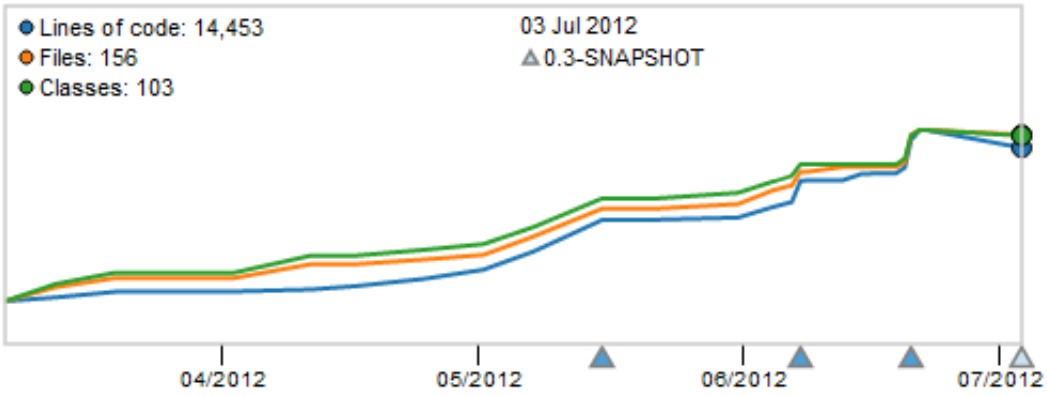


Figure 5.1: Graph of historic numbers for project A with Lines of Code, Files, Classes plotted against each other.

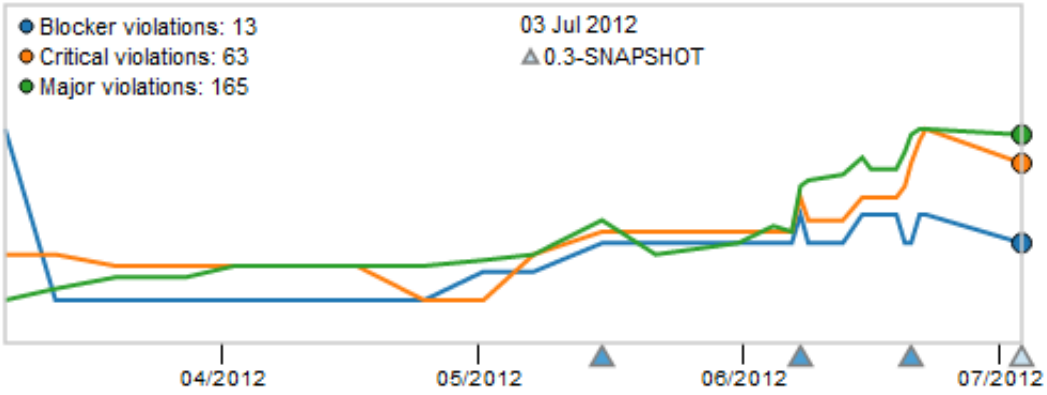


Figure 5.2: Graph of historic numbers for project A with Blocker Violations, Critical Violations and Major Violations plotted against each other.

The first graph (Figure 5.1) shows the development of lines of code over time and is visually compared to the amount of files and classes. The amount of files increased relatively equal to the increase of classes, and the amount of lines of code. This can also be seen from table 5.1. If we look at the first four rows, we observe that with time the amount of lines of code steadily increases. The next three rows also increase in a steady pace. Figure 5.4 shows that the amount of methods indeed (relatively) follows the same trail as the amount of lines of code. This indicates that after stimulating developers' awareness for the internal software quality, no major difference can be seen in the relative amount of lines of code per method, and amount of methods per class and amount of classes per file. Strange behaviour could be for instance a large increase in the amount of lines of code but without a large increase in the amount of

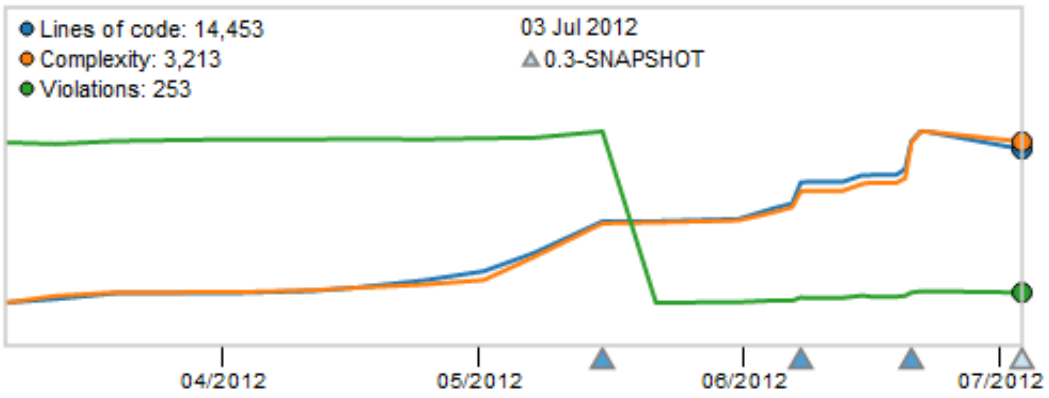


Figure 5.3: Graph of historic numbers for project A with Lines of Code, Complexity and Violations plotted against each other.

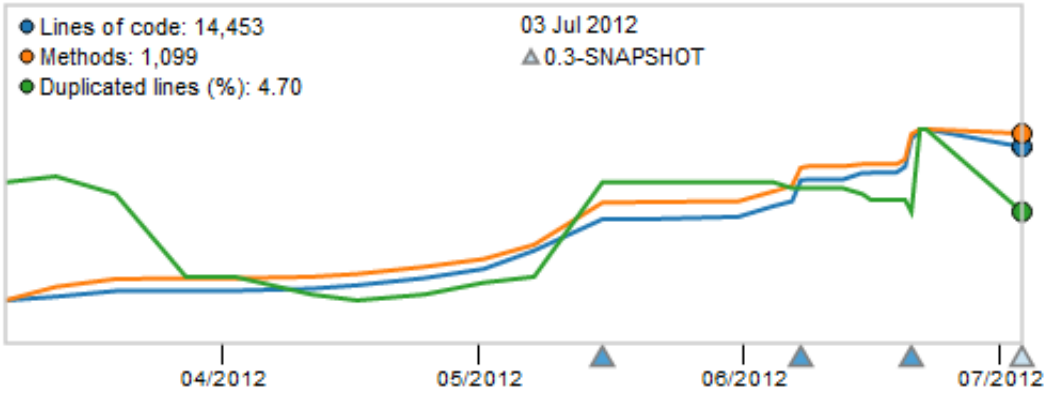


Figure 5.4: Graph of historic numbers for project A with Lines of Code, Methods and Percentage of Duplicated Lines plotted against each other.

methods. This would be evidence that developers were unable to divide functionality over multiple methods. This might be evidence for methods that are responsible for too much functionality, which is error-prone[31].

The rows about complexity from table 5.1 show that the total complexity of the entire source code evolves relatively equal to the amount of lines of code. In section 2.3 we spoke about McCabe's Cyclomatic Complexity Number. This metric is defined on method level. Since we want to give developers a general sense of the source code they produce, we make use of a function included in Sonar. This functionality gives historic values of the total complexity for the project, as well as the average complexity per class and the average complexity per file. We use this to verify that the complexity does not increase in a short amount of time. The total complexity

is defined as the summation of the complexity of all individual methods (McCabe's CCN) of the project. We do realize that taking the standard deviation of the averages is not an ideal way to check for inconsistencies. To overcome this, the total complexity was used to check if the project in its entirety exceeds the standard deviation. Checks per method are also done to see if the complexity is exceeded (original McCabe's complexity). If this is the case this will be included in the list with all violations.

To summarize, the amount of lines of code, classes, files and methods developed all relatively equal to each other. The complexity was also developing in the same rate. Both of these phenomena can be explained by the fact that the source code was not developed in-house and that only change-requests were accepted. These change-requests might include new feature requests or may be actual change-requests on existing functionality. This causes that small portions of the source code are changed relative to the entire source code base.

Other than these normal observations, only two pecularities can be discovered from a date before May 7th. The fact that on March 12th the amount of blocking violations decreased from 17 to 11 and the fact that on March 27th the percentage of duplicated lines of code also decreased, from 5.20% to 3.60%.

The decrease in blocking violations can be explained by the fact that existing code consisted of many exceedances in the maximum method length. Functionality these methods belong to was asked to be changed in that period.

From table 5.1 it can be seen that between March 6th and May 7th the amount of duplicated lines (and thus the percentage of duplicated lines) saw a significant decrease. For the decrease no explanation could be found.

5.2.3 Observations After Stimulating Awareness

If we look at the graphs after the first week of May, we can observe more activity than before. We can also identify more peculiarities.

Table 5.2: Table with violations data for project A.

Metric	March 6	May 7	May 15	June 7	June 20	July 3
Blocker Violations	17	12	13	14	13	13
Critical Violations	55	55	57	60	63	63
Major Violations	136	144	150	156	165	165
Minor Violations	483	499	512	0	0	0
Info Violations	110	108	109	5	12	12
Violations	801	818	841	235	253	253
Weighted Violations	1336	1326	1377	908	940	940

First, from table 5.2, it can be observed that the total amount of "violations" decreased dramatically between May 15th and June 7th. This can also be observed in figure 5.3, a clear decrease can be identified in the green line; More specifically, from table 5.2 it can be seen that the amount of violations with severities "info" and

"minor" decreased almost to zero. This can be explained by the fact that developers saw the result in the feedback mechanism and changed their coding style in their IDE, to match the style Checkstyle uses. The remaining violations were not fixed because it related to code not written by KPMG developers. If we look at figure 5.2, we notice that the trend for violations is to increase over time. Sometimes a decrease can be discovered.

The second is the development of percentage of duplicated lines of code. As already stated, the percentage of duplicated lines first sees a large decrease in the first few weeks, followed by a sharp increase between May 7th and May 15th. If we look at figure 5.4, we can see that this is an increase that is relatively more than the amount of added lines of code. This may be a sign that either the developers did not take notice of the feedback mechanism or they did not care about the increase. Asking the developers about this activity gave the impression that they indeed forgot about the feedback mechanism. On June 20th and June 21st, some new methods were created, but the percentage of duplicated code rose by 1.4%, which is significant with 14.000+ lines of code. Asking developers about this particular phenomenon learnt us that they did notice the sharp increase in the percentage of duplicated lines in the developed feedback mechanism this time. After reviewing their own code they found a more elegant way to solve a certain implementation issue which decreased the amount of duplicated lines again. This corresponds with the findings of Lacoste, where he states that defects are fixed in the following source code commit [18].

5.2.4 Afterthoughts

Looking at these phenomena we can state that if developers pay attention to our feedback mechanism, the internal software quality can certainly improve. The problem here lies in the fact that developers must be pointed to the information to base their decision on to redo the work and they might also be reluctant in changing code which they did not write.

After interviewing developers about these two findings, the common answer was that they were under a lot of time pressure, hence the duplication of code in the first finding. The decrease in the amount of violations can be explained by the fact that they did take notice of the original amount of these violations by the feedback mechanism, and these could be fixed relatively easy by using other coding standard settings in their IDEs for example.

However, the amount of violations with the severities "blocker", "critical" and "major" have all remained at the same level. This might be clarified by the fact that developers treat the severity that Sonar uses for each violation differently. The majority of "blocker" and "critical" violations found their origin in method-length, cyclomatic complexity exceedance and unused local variables. Asking developers why these code snippets were not changed resulted in an answer related to missing incentives and a difference in acknowledging the severity. In the two exceedances of the cyclomatic complexity metric the measured value was 11, where the threshold was set at 10. Deletion of local variables was only done in the case where existing

code had to be adjusted to implement the new feature. An explanation for not removing other unused local variables must be found in the fact that developers might as well do a full source code overhaul. This has not been agreed with the client, so no time will be invested in such activities.

5.3 Project B

5.3.1 Project Description

As already stated in section 4.2.1, project B is an application to give civilians the ability to measure the "performance" of the municipality they live in. A number of (mainly IT) projects should be implemented by each municipality because of federal regulations. The performance is measured both by the amount of projects that have already been successfully implemented and the progress of other projects.

Project B is again a web application that is separated in two sub-projects. The back-end, or business logic, and the "normal" project with front-end. There is also a web service associated, but this is incorporated in the business logic. Projects that have such an architecture usually have a back-end which' functionalities already have been predefined in early stages of the project. After the implementation on the back-end has been finished, it is less likely that changes would have to be made since the back-end only contains core functionality. Only in the case when core functionality has to be added or has to be changed the source code belonging to the back-end will be adjusted. Statistically the front-end will be changed more often than the back-end. But since building the front-end project always triggers the buildprocess for the back-end (but not the other way around), we have (at least) as much data about the back-end as we have of the front-end. But due to the fact that the actual back-end is changed less often, we should see many flat lines in the graphs below.

Because the back-end project serves as core functionality for many projects of which the majority is not being developed by KPMG, it is not an odd sight to see that the front-end consists of a significantly smaller amount of lines of code.

The project was started February 2012 and consists both of PHP as well as java source code. The back-end project consists of 4.200 NCLOC and the front-end consists of 900 NCLOC. Until now, four KPMG developers have worked on this project, of which mainly one developer has been working on the back-end. The developers can be described as very experienced with one developer even having experience with internal software quality metrics. There was also a project leader involved who was responsible for customer relations and developer guidance.

Monitoring the two sub-projects was not started at the same time due to problems with correctly monitoring Java applications. For both of the two projects we have five snapshots in which we found peculiarities. We start with the business logic part.

5.3.2 Observations Business Logic Before Stimulating Awareness

Data of the business logic part of project B stretches from May 1st till June 19th. Developers were informed about this research on May 15th.

If we look at the time before May 15th, we can see in figure 5.5 that the amount of lines of code stays relatively equal to the amount of classes and files. The amount of files cannot be seen directly from the graph because it stays relatively identical to the amount of classes. This can be verified from table 5.3 where the values are

a match. From figure 5.8 it can be observed that the amount of methods also stays relatively equal to the amount of lines of code.

Table 5.3: Table with historic data for the business logic of project B.

Metric	May 1	May 15	May 31	June 13	June 19
Lines of Code	3498	4048	4128	4173	4191
Files	90	103	104	104	105
Classes	90	103	104	104	105
Methods	280	324	329	337	337
Duplicated Lines	258	463	463	435	435
Duplicated Lines (%)	4.1%	6.4%	6.3%	5.9%	5.9%
Complexity	538	636	646	659	659
Complexity / Class	6.0	6.2	6.2	6.3	6.3
Complexity / File	6.0	6.2	6.2	6.3	6.3
Violations	148	185	186	195	195

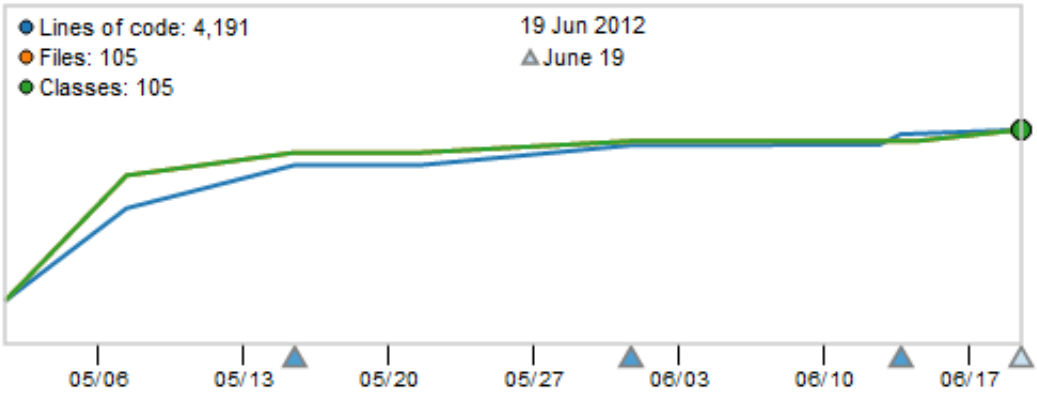


Figure 5.5: Graph of historic numbers for the business logic of project B with Lines of Code, Files, Classes plotted against each other.

In figure 5.7 it can be seen that the complexity stays relative to the amount of lines of code throughout the entire period.

If we look at the amount of violations (Figure 5.6), we observe that this number increases for both the critical as well as the major violations. But if we look at table 5.4, we can see that the increase in critical violations is not that significant; It only increases from 38 to 39 in the given time period. The amount of major violations increased from 58 to 72, which is an increase of 24%. At the same time the amount of lines of code rose by only 15%.

Figure 5.7 shows that the total amount of violations remains to develop relative to the amount of lines of code for the rest of the time.

From the same figure one can observe that the percentage of duplicated lines of code did increase relatively more than the increase of lines of code.

5.3.3 Observations Business Logic After Stimulating Awareness

After the initial period of establishing a baseline, we are unable to see large differences in the graphs and tables. To begin with the percentage of duplicated lines of code, we observe from figure 5.8 that the trend is to convergence in the direction of the lines of code up to a point that it reaches the same relativeness. From table 5.3 we can even see that on June 13th 0.4% of the duplicated lines was removed.

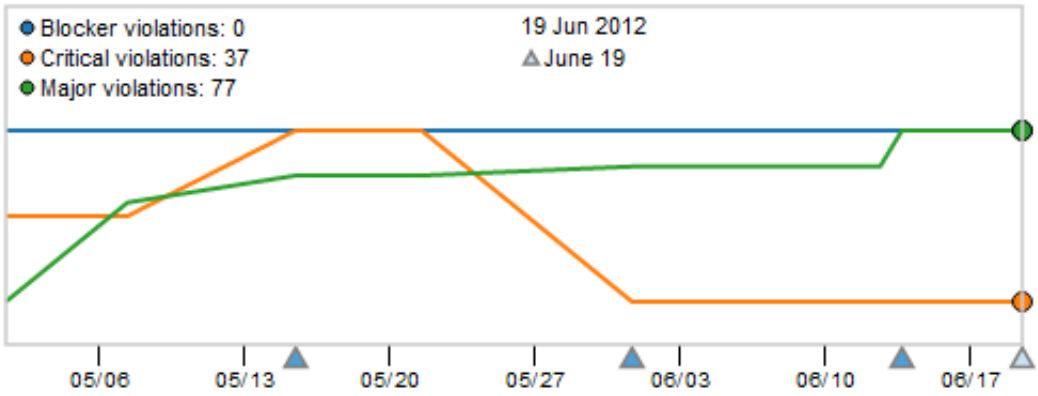


Figure 5.6: Graph of historic numbers for the business logic of project B with Blocker Violations, Critical Violations and Major Violations plotted against each other.

If we take a look at the amount of violations, we see more movement. Figure 5.6 shows a sharp decrease of the amount of critical violations between 21st of May and 31st of May. After an inspection from table 5.3 we see however that it stands in no relation to the total amount of critical violations. The absolute decrease was only two. This is a good example that shows that giving developers only a graph depicting the whole measuring period might give a distorted image. This could be both in a good way, or in a bad way.

From figure 5.7 we do see a high increase in the total amount of violations at June 13. From table 5.3 we see that in 40 changed lines of code nine violations were created. From table 5.4 we can see what kind of violations were introduced. Four are categorized as being major violations (The sharp increase of major violations in figure 5.6). Another four violations were being informative violations and comprised mainly of unused imports or constants not matching a naming pattern.

In the same time period the percentage of duplicated lines decreased with 0.40% as can be seen in figure 5.8. It could be that violations and duplicated code are inversely proportional to each other for this part of the business logic. During the developer interviews it did not become clear why these events occured simultaneously.

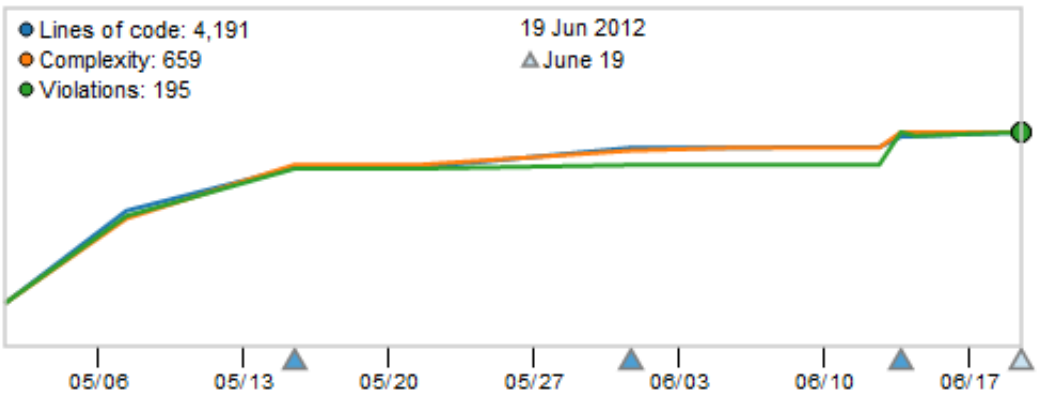


Figure 5.7: Graph of historic numbers for the business logic of project B with Lines of Code, Complexity and Violations plotted against each other.

It must also be taken into account that with this code churn (the amount of changed lines of code), it might be hard to make educated statements.

An interview with the main developer revealed a few noteworthy things. First, the fact that not much comment lines were added to the code was a deliberate choice. The developer believes that code should be self explanatory and therefore does not need any comments. Although this seems a good explanation, we should be wary about this statement because in the future less experienced developers might have to work with this source code. The same developer is also familiar with the static analysis tools that gather metrics for this research, and in his opinion many violation rules are outdated. The specific rule that individual lines may not exceed 80 characters was named for instance. This was a rule meant for older terminals that had a "physical" limit to only show 80 characters per line on their display. At one point, a discussion started between several developers and finally they agreed on using a line limit of 120 characters. The interview also put out a suggestion: When a company really wants people to respect the (albeit old) coding style rules, they could make use of so-called pre-commit hooks. The SCM server checks upon every check-in if the code follows the coding style and it will refuse the check-in if not. The idea was pitched successfully within the company, and pre-commit hooks are now being implemented for every project.

5.3.4 Observations Front-End Before Stimulating Awareness

Although data about this part of project B has been gathered from March 8th, the front-end of project B is a lot smaller than the business logic. This is because the back-end is used by many more applications and this is just one of many front-end applications. Where the business logic lacks a large time frame of us being able to monitor the project, the front-end lacks the amount of lines of code. It is hard to draw

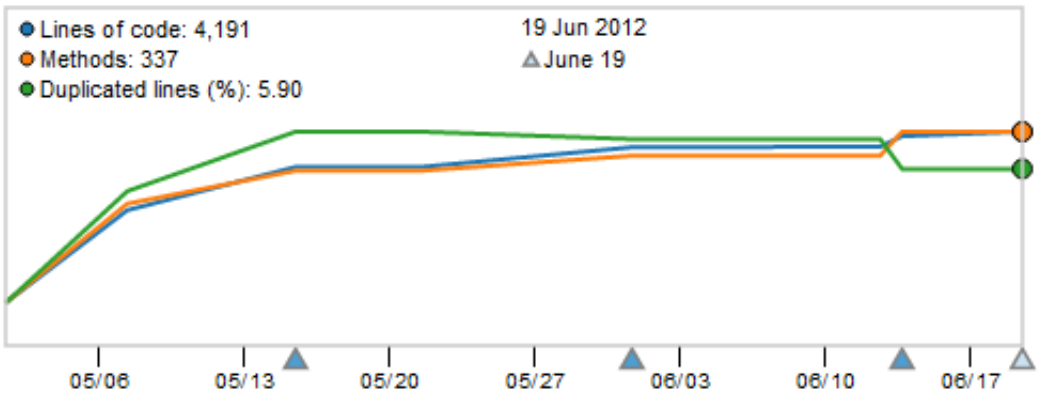


Figure 5.8: Graph of historic numbers for the business logic of project B with Lines of Code, Methods and Percentage of Duplicated Lines plotted against each other.

Table 5.4: Table with violations data for the business logic for project B.

Metric	May 1	May 15	May 31	June 13	June 19
Blocker Violations	0	0	0	0	0
Critical Violations	38	39	37	37	37
Major Violations	58	72	73	77	77
Minor Violations	27	36	37	38	37
Info Violations	25	38	39	43	44
Violations	148	185	186	195	195
Weighted Violations	391	447	441	454	453

conclusions on a small sample space. Nevertheless, even in this small project some oddities can be discovered. Developers were informed about the feedback mechanism on April 24th, when it was just half the size it has at the end of us monitoring the project.

From figure 5.9 it is clear that the distribution of lines of code over classes, and classes over files is evenly divided over the first period. The three tend to increase relatively equal. Later, they tend to diverge from each other. From March 8 till the moment where developers were informed about our feedback mechanism the lines of code and the amount of files are being overtaken relatively by the amount of classes. Figure 5.12 gives visual proof for the fact that the amount of methods also stays relative to the amount of lines of code, up to April 24th.

If we look at the complexity in figure 5.10 for the time period from the start until April 24th, it can be seen that the complexity increases relative to the amount of added lines of code.

The development of the amount of violations on the other hand did see an abnormal increase that does not match the relative increase the lines of code saw. If we

Table 5.5: Table with historic data of project B.

Metric	March 8	April 24	April 30	May 28	July 3
Lines of Code	9	451	749	885	892
Files	1	9	22	26	29
Classes	1	16	21	25	25
Methods	2	51	84	100	100
Duplicated Lines	0	0	0	0	16
Duplicated Lines (%)	0.0%	0.0%	0.0%	0.0%	0.8%
Complexity	2	84	148	176	176
Complexity / Class	2.0	4.8	6.7	6.8	6.8
Complexity / File	2.0	9.3	6.7	6.8	6.8
Violations	0	8	11	8	8

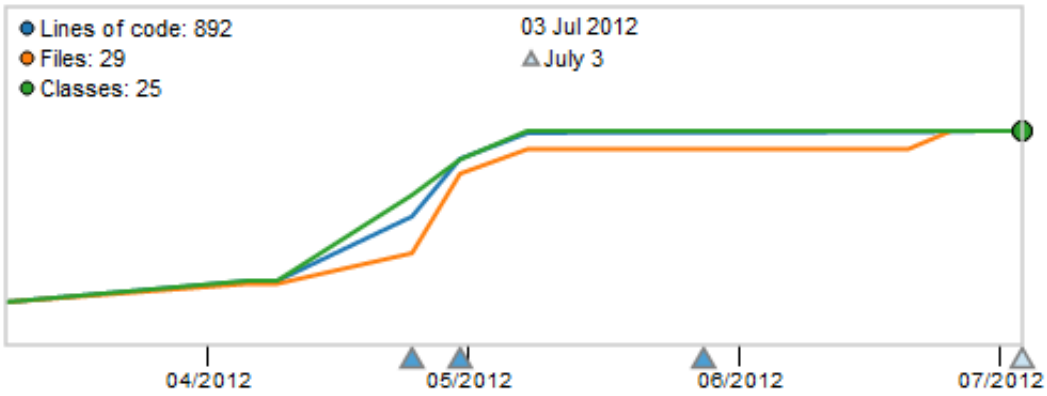


Figure 5.9: Graph of historic numbers for project B with Lines of Code, Files, Classes plotted against each other.

look at table 5.6 to see the violations in more detail, we observe that the total amount of violations is eight. This means that any change will result in a noticeable spike in the graph. From figure 5.11 it became apparent that in the begin period the most changes occurred. A closer look at these violations showed that the critical violation saw its cause in the fact that one method had a cyclomatic complexity of ten, where the default threshold is also set to ten. The major violations consisted of unused local variables. This is strange, because this project was originally developed by KPMG in contrast to project A. A possible explanation could be that the developers already added the local variables for future features, but this does not seem plausible.

From early March until late June no duplicated lines could be found in the project as can be seen from figure 5.12.

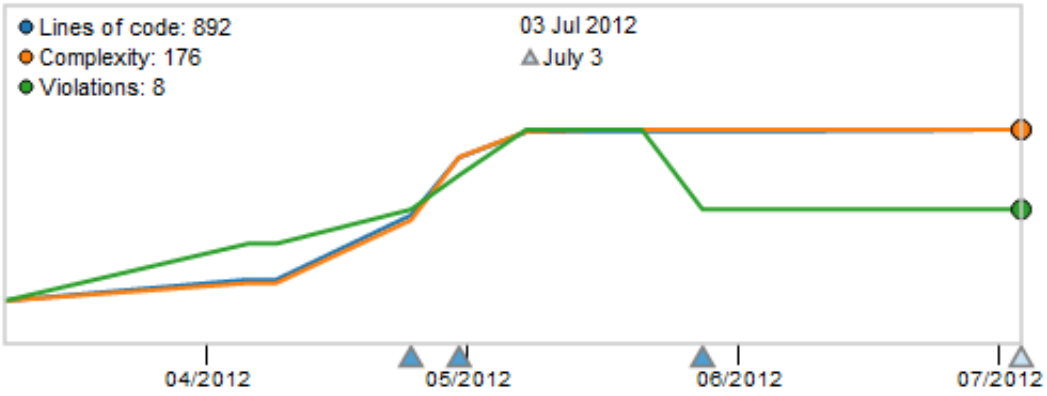


Figure 5.10: Graph of historic numbers for project B with Lines of Code, Complexity and Violations plotted against each other.

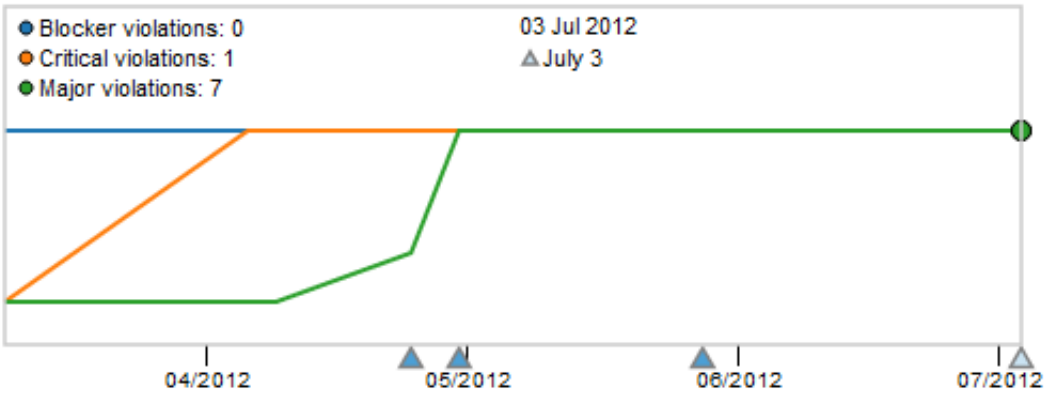


Figure 5.11: Graph of historic numbers for project B with Blocker Violations, Critical Violations and Major Violations plotted against each other.

Table 5.6: Table with violations data for project B.

Metric	March 8	April 24	April 30	May 28	July 3
Blocker Violations	0	0	0	0	0
Critical Violations	0	1	1	1	1
Major Violations	0	2	7	7	7
Minor Violations	0	2	2	0	0
Info Violations	0	3	1	0	0
Violations	0	8	11	8	8
Weighted Violations	0	13	28	26	26

5.3.5 Observations Front-End After Stimulating Awareness

As already said in the previous section, this project is quite little in size. The baseline was set before April 24th, and thereafter still roughly 450 lines of code were added.

If we look at the development of the amount of lines of code, with respect to the amount of files, classes and methods (figures 5.9 and 5.12), we see that the four metrics never stray far from each other.

If we look at the complexity in figure 5.10, it is easy to see that it stays relative with the amount of lines of code for the remainder of the project's duration.

In the same figure it can also be seen that on May 28th, a decrease in the amount of violations occurred. Although the total amount of violations is not very high, it still is a significant change. If we look at table 5.6, we see that only the minor (two) and info (one) violations were resolved. A closer look at the violations that were resolved learned us that they solely existed of invalid line endings and incorrect scope indents. These problems were most likely fixed by using another kind of coding style in developers' IDE.

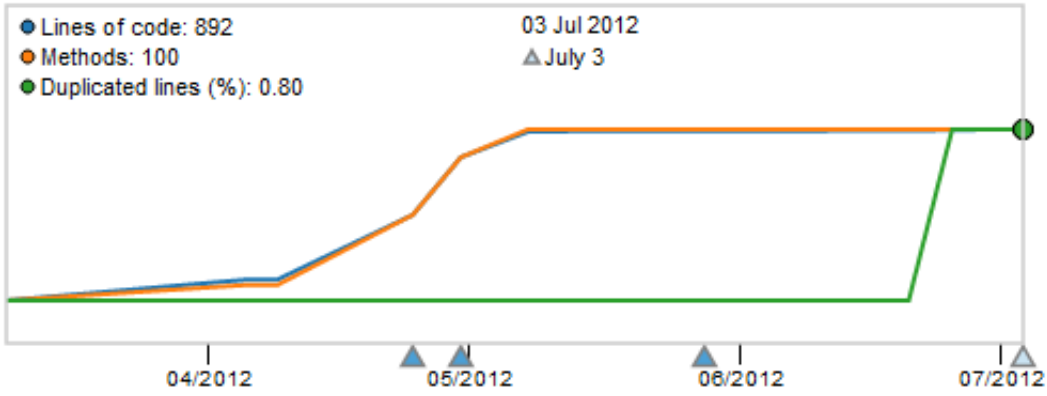


Figure 5.12: Graph of historic numbers for project B with Lines of Code, Methods and Percentage of Duplicated Lines plotted against each other.

Figure 5.12 finally shows the percentage of duplicated lines. At the end of June a spike in the graph can be observed. From table 5.5 it becomes clear that 16 LOC were duplicated, which is insignificant for the 892 LOC totalling the entire project. After inspecting the duplicated code, it became clear that this is in fact a bug in Sonar, as it qualified a section of HTML code in one PHP file as a duplicate of itself.

5.3.6 Afterthoughts

If we compare this project with other projects, it is clear that the front-end of project B is not very susceptible to hold complex source code, nor to have a large amount of violations. This may be a result of the small amount of lines of code, or because of the project itself. It might also be the case that our feedback mechanism is unsuitable for

this project. After inspecting the reports of the feedback mechanism, we have to draw the unfortunate conclusion that this is indeed the case. A few reasons can be in place here. The first reason is that the source code did not consist of that many lines of code, thus changing the behavior of some metrics. For some the likelihood to deviate from their average may change (some will decrease while others will increase, metrics based on percentages will lose their likelyhood to change when the amount of lines of code grows larger for instance). A second reason could be that only in the period starting from April 9th till May 7th a lot of lines of code were changed or added for the back-end project. This is the reason that only in this particular time period highly fluctuating metrics could be observed. This might also be the reason that later periods did not show these graphs anymore because the outlier became the average.

These two projects learned us that active attention is necessary, otherwise developers will continue to work as before. This often means too much work in too little time. During interviews with developers it became apparent to make agreements of some sort about the (internal) software quality. All developers agreed on the fact that having an extra set of requirements for their work would help achieving the goal of producing high quality software.

5.4 Project C

5.4.1 Project Description

The last project which was started being monitored was our third project. It is a project focused on offering municipalities an architecture which gives them the opportunity to increase their digital serviceability to residents. It is a good project in combination with project B, to assess the current serviceability of municipalities.

It is implemented as a web application built with PHP technology. A few sidenotes have to be made for this project. First, the project was started only a few weeks before we began to monitor. Since this project would not be very large in size, it might be the case that the majority of the implementation has already been done. We do not expect to see a large difference in lines of code. Because of the fact that the project was anticipated as not being that large, we began giving developers access to our feedback mechanism from the start of us monitoring the project. This does mean however that we have not been able to determine a good baseline for this project, since developers got access to our feedback mechanism from the beginning of us monitoring the project.

Despite the size and age of this project, we were still able to detect some interesting events in which some of the metrics exceeded the standard deviation.

The development team for this project consisted of three experienced developers with two developers having extensive programming experience with multiple languages and one developer who was not very experienced with PHP but who has experience with other programming languages. Again, a dedicated project leader was assigned to guide the developers and to keep contact with the customer throughout the project's duration.

5.4.2 Observations After Stimulating Awareness

The project has been monitored from June 1st till June 19th. Despite only having data from these two and a half weeks, multiple easily detectable events occurred. First, we can see from figure 5.13 that the amount of lines of code evolves again relatively equal to the amount of files and classes. This means that newly added code is neatly being distributed among classes and classes again over files. From figure 5.14 it can also be seen that the amount of methods stays relative to the amount of lines of code.

If we look at the complexity of this project in figure 5.15, we can observe a spike on June 13th. After having a look at table 5.7, we can see that the complexity rose by 50 points at an increase of around 350 in the amount of lines of code. The increase in complexity is hard to explain. A possible explanation might be that the added code consisted of very complicated code, but the overal development of the graph contradicts this, as it remains to stay relative to the amount of lines of code (as can be seen in figure 5.15).

In the same figure it can be seen that the development of the violations follows the same path. But if we look at figure 5.16, we can easily identify a change in the

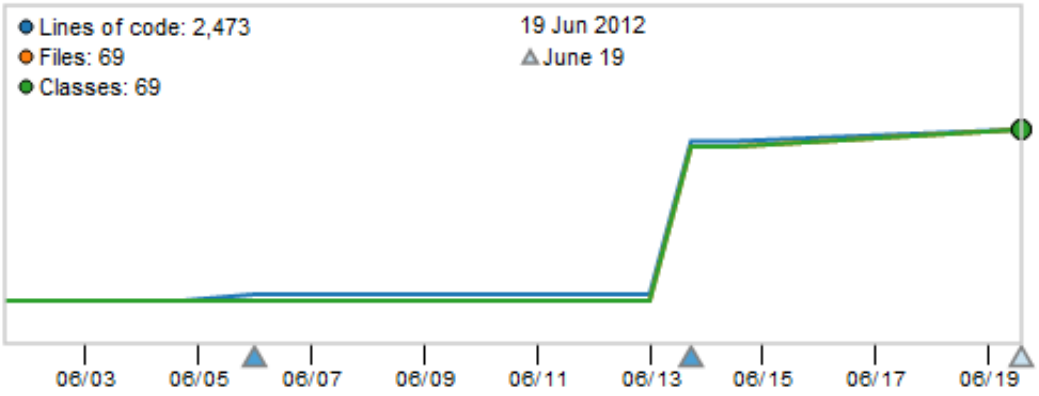


Figure 5.13: Graph of historic numbers for project C with Lines of Code, Files, Classes plotted against each other.

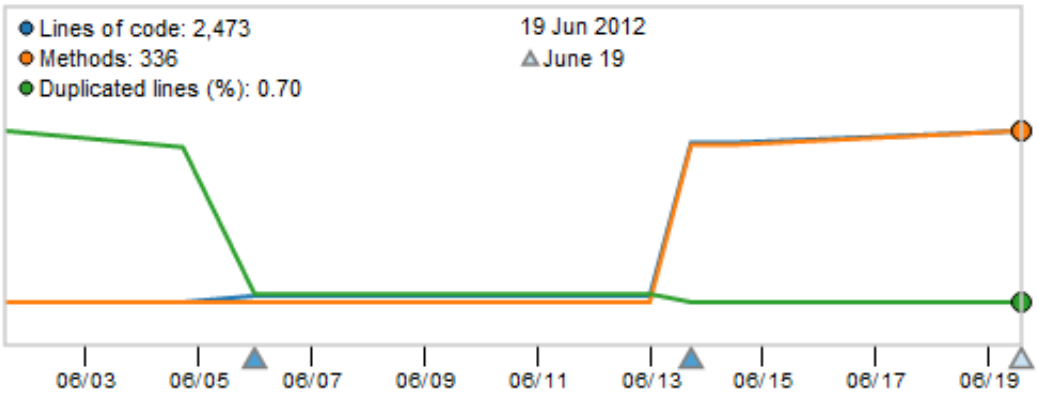


Figure 5.14: Graph of historic numbers for project C with Lines of Code, Methods and Percentage of Duplicated Lines plotted against each other.

amount of critical violations on June 13th. After inspection of the values in table 5.8, we have to conclude that this is again not a significant increase. The threshold for the Cyclomatic Complexity Number was once exceeded by one. The threshold for maximum number of children was once met. The maximum number of methods in a class also exceeded the threshold by one. The fact that these few violations were not fixed can be found in just a few possibilities. Either the developers did not know about these violations because they reasoned that without graphs, everything would be alright. It might also be the case that they did know about the violations but simply could not be bothered fixing them.

Another event occured between June 1st and June 5th. A sharp decrease in the amount of duplicated lines of code can be identified between these dates (Figure

Table 5.7: Table with historic data for project C.

Metric	June 1	June 5	June 13	June 19
Lines of Code	2094	2108	2447	2473
Files	59	59	68	69
Classes	59	59	68	69
Methods	299	299	333	336
Duplicated Lines	82	24	24	24
Duplicated Lines (%)	2.8%	0.8%	0.7%	0.7%
Complexity	358	358	408	410
Complexity / Class	6.1	6.1	6.0	5.9
Complexity / File	6.1	6.1	6.0	5.9
Violations	8	8	9	9

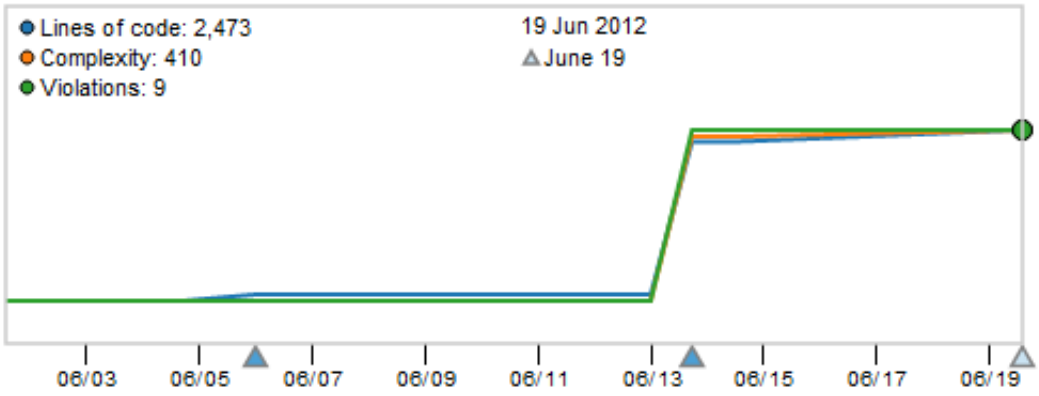


Figure 5.15: Graph of historic numbers for project C with Lines of Code, Complexity and Violations plotted against each other.

Table 5.8: Table with violations data for Project C.

Metric	June 1	June 5	June 13	June 19
Blocker Violations	0	0	0	0
Critical Violations	1	1	2	2
Major Violations	7	7	7	7
Minor Violations	0	0	0	0
Info Violations	0	0	0	0
Violations	8	8	9	9
Weighted Violations	26	26	31	31

5.14). Although only twenty lines of code were *added*, a decrease in the percentage of duplicated lines can clearly be seen. The percentage decreased from 2.80% to 0.80%. From the Sonar dashboard we can see that this is a decrease of 58 duplicated

lines. If we look in table 5.7, we observe that the percentage of duplicated lines did not increase again after this date, it even dropped a little to 0.70%.

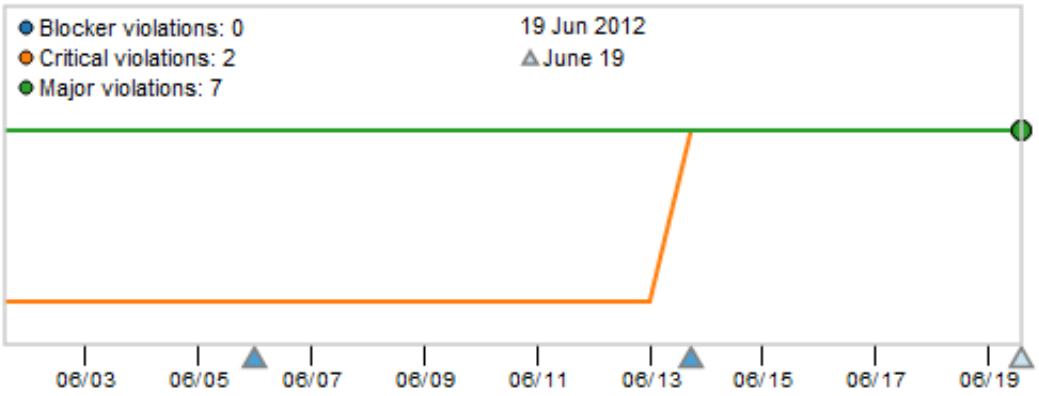


Figure 5.16: Graph of historic numbers for project C with Blocker Violations, Critical Violations and Major Violations plotted against each other.

5.4.3 Afterthoughts

It is difficult to put these numbers into perspective, even more so alongside other projects. Project C has a different character than project A. In project A many different builds could be identified, which might be because of the large differences that were created by many small changes. Project C on the other hand knows only a few relatively large changes.

If we look at the four figures in general, we notice only a few spikes. Of course some activity can be identified both with the amount of violations as with the percentage of duplicated code, but it is hard to base any conclusions on such a small sample space.

Having taken notice of the graphs and the developer's related behavior, it should come as no surprise to conclude that the feedback mechanism does not work well. There is however still something left to investigate, the reports of the feedback mechanism. The reason why no metric went down besides the percentage of duplicated lines of code is much simpler. It seems that no metric fell outside the standard deviation! Therefore no graphs were shown.

After this eye-opener it is not hard to see why the developers did not make use of it. It does beg the question why developers did not fix any violation from the "all violation" list. During the developer interviews it became clear that there were multiple reasons. One of the reasons was that a developer assumed that no violations were found because there were also no graphs. Another reason is that the developers did see the list of violations but did not classify them as very urgent.

This proves that the way in which feedback is returned is just as important as the feedback itself. The classification of violations is also a very important factor since developers might lose interest if violations are not classified like their taste.

5.5 Developer Interviews

After the main period of the experiment was done, five developers were asked to participate in an open interview which started with a few multiple choice questions that were meant to start the interview / discussion. These interviews were done to learn if developers were paying more attention to the internal software quality eventhough this might not be seen directly from the data. Among the group of five developers were three senior developers and two junior developers. All of these developers have knowledge of the Java programming language. One of the junior developers did not have extensive PHP knowledge prior to working on PHP projects but was able to make himself familiar with the language in a few days. Developers were not only responsible for implementing certain functionality, but also to test their implementation. During the interviews also ideas for the future were not being left unspoken.

During the interviews it became clear that each developer was very knowledgeable and that there was often a good explanation for a certain negative trend. Generally speaking, a much heard reason for not doing extensive tests is time-pressure. Testing is an activity that is much overlooked, or something for which too little time is calculated. It might seem that software is not being tested altogether. This is not true. Because development teams work intensively together with customers, functionality is always extensively tested. In other words, the emphasis for testing efforts is placed on the functionality, rather than on the internal software quality. Because development teams work with short development iterations, testing activities sometimes fall through the process. To create a platform to state the importance of testing, developers all agreed that by incorporating the status of these testing activities at intermediate meetings will help to improve their product. When more attention is focused on testing and on internal quality metrics, the overal quality is again able to improve.

Every developer also agreed that more effort should be made towards active quality monitoring. An obvious point of improvement was to incorporate the feedback mechanism in the IDE of the developers' choice. This would make it very accessible. It would also prevent developers from accidently forgetting about the system or ignoring it altogether. For smaller projects it would work well, but it would take too much time for bigger projects to have the results returned from the buildserver. A solution for this problem would be to have the development machines run the static analysis tools themselves, but this would create a situation in which many development machines have to be configured instead of only one. Since configuring the buildserver and maintaining it is hard this is not recommended.

From the interviews it also became clear that a custom set of violation rules should be made for every project. This would take up a lot of time, but it will let developers take the (possibly long) list of violations serious as it (in its default form) often contains violations based on outdated ideas or violations with meaningless severity to make it more generally usable. The extensive list of all violations in a project works demotivating according to some developers.

It was also suggested to use the feedback tool from the earliest stages of the

project in order for developers to be able to switch quickly according to the situation. Since the feedback mechanism uses the standard deviation of past measurements to determine if some metric value should be brought to attention, it is very likely that it erroneously reports several metrics for exceeding their standard deviation. This could very well be ordinary behavior, as the project should be given some time to initialize. Introducing the feedback mechanism too early might let developers lose interest. This might be compensated by raising the threshold value to two standard deviations before including the metric in the feedback report.

Another idea also came along; the usage of pre-commit hooks in the SCM server. When a developer wants to add new code or tries to adjust existing code, the SCM server first runs certain coding style and / or internal software quality tests, to see if the code complies to pre-defined rules. Only if no violations were found, the code will be added to the rest of the source code.

A final idea that came from the interviews was to include social games into the development process. For instance adding "good code" to the project will give developers some credits while breaking the project or adding "bad code" will deduct some credits. Having a scoreboard should increase the productivity and competition between developers. It should be noted that such games might be exploited. A necessary, but also exceptionally large piece of code with high complexity might be cut into pieces to comply to the maximum complexity rule for instance. But this might not do the overal quality any good, since it might never have to be adjusted when it consists of core functionality that never will be adjusted.

5.6 Threats to Validity

The environment in which experiments are done can be of influence for the outcome which can make the obtained results hard to interpret. This is why every research project is performed in a certain (preferably closed and controlled) environment. This chapter describes the environment and other limitations of this research. This chapter consists of three sections that test the validity of this research as recommended by Perry et al. [29] and Yin [37]. The first section states threats to the construct validity. In the second section the internal validity of the research is given and in the final section the external validity is explained.

5.6.1 Construct Validity

Construct validity answers the question whether the variables (both dependent and independent) accurately model the research question. This deals with the question if the right variables have been chosen and if the research question can be answered by having only data of these variables.

The research questions we have defined in section 1.2 all require the fact that we have some sort of benchmark to compare the state of the software projects before and after we have began actively making developers aware of their produced software

quality. **RQ1** let us define our own benchmark. To be able to put everything in perspective, we have to describe the environment as well.

To form a reliable conclusion, the amount of data is very important. The more data there is available, the more reliable the conclusion based on this data is. In this research a few variables are relevant that are begging the question if they are sufficient:

- **Amount of projects** The amount of monitored projects is less than five. This is quite a low amount of projects to base a solid conclusion on. These projects are also quite different from each other, so it is also hard to base a conclusion for a project of a certain type.

- **Short project duration** The duration of the project determines for a great deal the possibility to act on the returned feedback. Project C in particular has not been monitored for a long time. If a project is only being used for a short period of time in this research, the developers may have not been given enough time to work with the feedback mechanism. This can be of influence for the results. It would be nice to have data of a long period of time to see the effects this research has on the long term. This is important, since many software projects have to be maintained for many years after they have been taken into use.

- **Number of metrics** The metrics we use are selected because they all fall in the categories of metrics which are of influence for the current state of the software quality. "we should have at least one coupling, one cohesion, one size, and one inheritance metric included to address the biggest areas of quality-influencing properties" [22]. We should state however, that we were unable to gather data from every one of these categories due to tooling limitations. This is further explained in section 6.1.

If we look at the list of variables above, we can easily identify improvements for future research. In particular the amount of projects might be difficult to defend our conclusions.

5.6.2 Internal Validity

The internal validity questions the dataset. Are the entries under research all equal in size and are timed variables (eg. the time between a and b also equal in length?

We observed several projects with varying sizes which each have also different development times. We recognize the following list of variables:

- **Projects of different sizes** Some projects are bigger than others. The more lines of code a project consists of, the likely it is that more lines of code are edited.

- **Project of different durations** Some projects are worked on for several months to more than a year while others may only know two months of development

time. This will be reflected in the amount of build data that is available. Because data is generated every few weeks (that depends on the predetermined time each sprint takes), a difference in project duration of several months can easily be seen in the amount of build data that is available.

If we look at the list above, it is clear that identical projects could not be selected. If identical projects could be selected, multiple ways to improve the internal software quality could have been used to see if a difference could be spot. This was replaced by the fact that for every project a short amount of time was used to see the natural behavior of the project without having the development team we were monitoring their project. This does mean however, that the kind of changes made in the earliest stages of monitoring might not have been the same as changes made later on in the project.

5.6.3 External Validity

The external validity answers the question whether the outcome of this research can be extrapolated for general cases. As stated in the construct validity and internal validity subsections, the projects we have monitored can by no means stand in for the millions of projects that are or have been in development. The chosen projects are simply too diverse and too custom to be used as common software projects examples. The developers and the way in which we have tried to make them more aware of internal software quality, can however.

Another variable that influences the external validity is the programming language used in the project. The projects we monitored in this research made use of PHP, Java, Javascript and HTML technology. While some programming languages can be compared fairly well, there is a large discrepancy between others which makes a comparison hard.

5.7 Discussion

When we look at the obtained results it becomes clear that it is hard to draw an unambiguous conclusion. Unfortunately, at the last moment we discovered that some of the preferred metrics were not gathered correctly. Some metrics never made it into our feedback report, while we did expect them to. This could mean that the values of the metrics always fell inside the 68% standard deviation interval, so our feedback mechanism did not trigger on these values. A closer look at the values revealed however that there was something wrong with the measurements. Further investigation led to the conclusion that Sonar was unable to parse the lack of cohesion of methods, efferent coupling, afferent coupling, depth of inheritance tree and the number of children from the logfiles. Unfortunately this was discovered in a late stadium of the research thus this could not be fixed. This may have had influence on the results of the experiments, although we still have solid data from other metrics.

We also have to take note of the human part playing a central role in this experiment. There are too many dependencies in place that influence the way in which developers work. As the results from project A show, active monitoring can definitly help to make developers (more) aware of certain unwanted situations. We have seen that this almost immediately led to an improvement in the pieces of code, and thus to higher software quality. Project B developed in another way. No systematic improvement can be shown from any of the graphs, on the contrary, from some graphs it may seem that developers did not care when new violations were introduced. After asking developers about these situations, the general consensus was that it concerned knowingly made rule violations or that the violations were set too tight or were too general. Project C did not contain many data entries and in them not many strange events occurred. A clear improvement in the internal software quality and thus in overal software quality cannot be identified. This might also be a consequence of the fact that the developers learned from projects A and B.

Seeing the above, it might be irresponsible to draw conclusions based on the results. We therefore also held developer interviews that should be able to clear things up. From the interviews it became apparent that this research did make developers aware of the notion software quality again. Due to tight deadlines developers were given insufficient time to pay attention to pursue a certain quality level. All developers were fond of the idea to incorporate software quality findings in every intermediate meeting. Developers were also quite happy with the fact that only important metrics were being shown when rules that belong to the selected metrics were being violated. This way they began to recognize certain unwanted behavior in their source code.

It is also very hard to estimate a minimum amount of improvement; That is too dependent on project specifications.

Chapter 6

Conclusions and Future Work

This chapter provides the conclusions that are based on the research done for this thesis. Future work and the contributions this research has done are also included in this chapter. The conclusions can be found in section 6.1. Contributions can be found in section 6.2, and future work that can be seen as a continuation of this research or work that can be done to improve this research is given in section 6.3.

6.1 Conclusions

This conclusion is constructed as follows: First, answers to the research questions which were introduced in section 1.2 will be given. Then some peculiarities found during the experiments follow.

To be able to answer our research questions, we formulated multiple underlying research questions. The majority of these questions could be answered after evaluating the outcome of our experiments.

RQ2 (the question if an improvement in software quality can be observed when awareness of software quality is stimulated among developers) cannot be answered without first answering **RQ1** (the question what software quality actually is). It also needs the support of **RQ3** (the question in which specific way awareness should be stimulated among developers in order for them to improve the software quality). To begin with **RQ1**, "what is software quality?". Software quality, and more specifically the internal software quality, has played a fundamental role in this research. Methods and tools have been explained that are able to gather information about the internal software quality. Many of such software tools exist today and some are able to perform static analysis on source code. From interviews with several people it became clear that software quality in itself cannot be expressed on a single scale or ladder. Different perspectives have to be considered when writing a report to determine the software quality from some application. Multiple software quality models are in existence today, but many are made to look at the software from all possible viewpoints, thus being negatively biased when only looking at the software from a certain perspective.

This is why we chose to solely look at the internal software quality. It is closest related to the state of the source code and it can be quantified while still taking the environment into account. This is important, because the state of the source code becomes relevant when maintenance of software systems begins. The better the state of the source code is, the easier maintaining it will become, which in turn makes future maintenance costs clear, and probably making it (more) affordable because during the development the maintenance phase was already taken into account.

The internal software quality is being quantified by making use of metrics of the source code. It is noteworthy to mention that of the metrics mentioned in 3.4.2, the following metrics were returned most of the time by our feedback mechanism:

- Complexity
- Duplicated Lines Density
- Comment Lines Density
- Non-Commenting Lines Of Code
- Classes
- Coding Style Violations

Other metrics occured occasionally in the feedback mechanism:

- Function Complexity
- File Complexity
- Class Complexity
- Blocker Violations
- Critical Violations
- Major Violations
- Minor Violations
- Info Violations
- Weighted Violations
- Violations Density

It stands out that two major types of metrics are present in the list above. These can all be explained by the fact that their umbrella metric (Complexity and Violations respectively) often appeared in the feedback mechanism, but the actual metric alternated between a few sub-metrics. The "violations" metric for example, appeared quite often in the feedback mechanism, but sometimes the underlying metric "major violations" appeared in the feedback mechanism as well, and sometimes the "critical violations".

The notable absentees that never made it into the feedback mechanism are:

- Lack of Cohesion Of Methods
- Efferent Coupling
- Afferent Coupling
- Depth of Inheritance Tree
- Number of Children

To be able to answer **RQ3** we have considered multiple options but were forced to make use of a web application to stimulate software quality awareness among

developers. In the most ideal situation developers do not have to leave their IDEs in order to receive feedback on their work. We have seen that in some situations developers were inclined to fix a certain unwanted situation. We have also seen situations where developers did not fix undesireable metric values. The cause may be that the developers do not share the high severity in the found anomalies, or the advice of the feedback mechanism was ignored altogether.

We took over the practice of many software quality models to distinguish certain aspects of the internal software quality. We distinguish the following aspects:

- Coupling
- Cohesion
- Size and Complexity
- Inheritance

For each of the aspects above we have selected metrics which our tooling infrastructure was able to gather. As can be seen from our answer to **RQ1**, we were unfortunately not able to gather all metrics, but we are certain every aspect has been covered sufficiently with the metrics we were able to gather.

As for **RQ2**, we have seen that people (developers) are certainly susceptible to influences from their (working) environment, but we have also seen that this has to be done in an active way in order for them to keep being focused. This proves us that the current state of the (internal) software quality is indeed related to awareness of this matter. Some developers were not educated about aspects that have influence on the software quality, other than to correctly implement certain design patterns. If developers know the idea behind certain internal software quality aspects and the related metrics, they may use this insight in their daily work.

One other important fact is that the information that is used to give feedback on developers' work is equally important as the way in which this information is being returned. We have selected to only incorporate the most important information to avoid making it too complicated. Unfortunately due to infrastructure difficulties we were not able to implement an active feedback mechanism.

6.2 Contributions

The contributions of this research can be formulated as follows:

- **Shown that a relation exists between awareness about software quality and the state of software quality.** We have shown that a relation exists between knowing the state of software quality and efforts to improve the state of software quality. During interviews developers acknowledged that feedback on their work would be appreciated. Some developers did not know all aspects of software quality and liked the idea of having this knowledge. This knowledge would then be used to improve their work.

- **Creating a feedback mechanism which is able to detect anomalies in recent metric values.** A feedback mechanism has been created to show the

current state of the internal software quality to developers. Because regular information has no significant value for developers, we only return feedback when there is something irregular to be seen in the feedback report. This way developers will always know that they should look at the feedback report when available.

- **Shown that our feedback mechanism is indeed able to improve the internal software quality.** We have identified events in which the use of our feedback mechanism contributed to the improvement of certain internal software quality metrics. Developers not only know which aspect of their work could use some improvement, but our feedback mechanism tells us how the improvement can be achieved.

6.3 Future work

This research has shown that future work on this topic has great potential. During this research multiple decisions had to be taken. Made decisions were not right or wrong, but testing the other possibilities might give new insights into the topic.
Information Contents
We have shown that developers are susceptible for improving their work if they have the right information. This research shows that returning some basic information works, but future research should evaluate if returning other, or more information works as well.
Information Structure
We have also seen that the information should be transformed into clear visuals. Long lists of violations can be useful, but only at a later stadium. Developers would like to see what is wrong in a single view.
Information Publication
During this research we had to make our feedback mechanism as a web application. An improvement that will contribute to establish a more solid conclusion is the use of active feedback in the form of IDE plugins that are able to return feedback as developers are working. A few problems can already be identified. For instance the static analysis tools have to be installed and kept up-to-date at each developer machine. Because some projects are quite large, it still might take some time for the active feedback mechanism can return some feedback. This might result in a perceived "delayed" instant feedback mechanism.
Environmental Indepedence
Future work does not only exist of using other information or some other way to give feedback to developers. Future work also entails testing the same way as was done in this research, but with other developers, and in other projects. This will create more support for the findings of our experiments.

Bibliography

[1] C. Archer and M. Stinson. *Object-oriented Software Measures*. Technical report. Carnegie Mellon University, Software Engineering Institute, 1995.

[2] Adesoji Adeola Arinola and Olalekan Abraham Adekunjo. Analysis of hiv/aids information awareness and effectiveness among artisans in ogbomoso, oyo state, nigeria. http://digitalcommons.unl.edu/cgi/viewcontent.cgi?article=1770&context=libphilprac, January 2012.

[3] M Azuma. SQuaRE: The next generation of ISO/IEC 9126 and 14598 international standards series on software product quality. In *Proceedings of the European Software Control and Metrics Conference (ESCOMŠ01*, pages 337–346, 2001.

[4] Joseph P. Cavano and James A. McCall. A framework for the measurement of software quality. *SIGMETRICS Perform. Eval. Rev.*, 7:133–139, January 1978.

[5] Robert N Charette. Why software fails. *Ieee Spectrum*, 3:1–10, 2006.

[6] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, 20(6):476–493, June 1994.

[7] P.B. Crosby. *Quality Without Tears: The Art of Hassle-Free Management*. McGraw-Hill, 1995.

[8] Ward Cunningham. Debt metaphor. http://www.youtube.com/watch?v=pqeJFYwnkjE, 2009.

[9] Het Financieele Dagblad. Maak een eind aan de digitale verspilling. http://fd.nl/Print/krant/Pagina/Opinie/445090-1206/maak-een-eind-aan-de-digitale-verspilling_bron_fd_krant, June 26 2012.

[10] Kecia A.M. Ferreira, Mariza A.S. Bigonha, Roberto S. Bigonha, Luiz F.O. Mendes, and Heitor C. Almeida. Identifying thresholds for object-oriented software metrics. *Journal of Systems and Software*, 85(2):244 – 257, 2012. Special

issue with selected papers from the 23rd Brazilian Symposium on Software Engineering.

[11] Ronan Fitzpatrick. Software quality: Definitions and strategic issues. `http://arrow.dit.ie/scschcomrep/1`, 1996.

[12] International Organization for Standardization / International Electrotechnical Commission. *ISO/IEC 9126. Software engineering – Product quality*. ISO/IEC, 2001.

[13] Martin Fowler. Technical debt. `http://martinfowler.com/bliki/TechnicalDebt.html`, 2004.

[14] Brian Henderson-Sellers. *Object-oriented metrics: measures of complexity*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[15] Wiebe Hordijk, Maria Laura Ponisio, and Roel Wieringa. Harmfulness of code duplication - a structured review of the evidence. In *Proceedings of the 13th International Conference on Evaluation and Assessment in Software Engineering (EASE)*, Swindon, UK, April 2009. British Computer Society.

[16] Elmar Jürgens, Florian Deissenboeck, Benjamin Hummel, and Stefan Wagner. Do code clones matter? In *ICSE'09*, pages 485–495, 2009.

[17] J.M. Juran and A.B. Godfrey. *Juran's Quality Handbook*. McGraw-Hill International Editions. McGraw-Hill, 2000.

[18] Francis J Lacoste. *Killing the Gatekeeper: Introducing a Continuous Integration System*, pages 387–392. IEEE, 2009.

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

[20] Thomas D. LaToza, Gina Venolia, and Robert DeLine. Maintaining mental models: a study of developer work habits. In Leon J. Osterweil, H. Dieter Rombach, and Mary Lou Soffa, editors, *28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20-28, 2006*, pages 492–501. ACM, 2006.

[21] Ribhu Nath Lavania. Software quality basics. *Software Quality Awareness Series-1*, October 2008.

[22] Rüdiger Lincke, Jonas Lundberg, and Welf Löwe. Comparing software metrics tools. In Barbara G. Ryder and Andreas Zeller, editors, *ISSTA*, pages 131–142. ACM, 2008.

[23] M Squared Technologies LLC. Metrics definitions. http://msquaredtechnologies.com/m2rsm/docs/rsm_metrics_narration.htm, 2009.

[24] Habra N. Lopez M. Relevance of the cyclomatic complexity threshold for the java programming language. http://www.fundp.ac.be/recherche/publications/page_view/58160/, 2005.

[25] Mark Lorenz and Jeff Kidd. *Object-Oriented Software Metrics*. Prentice Hall, July 1994.

[26] Thomas J. McCabe. A complexity measure. In *Proceedings of the 2nd international conference on Software engineering*, ICSE '76, pages 407–, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[27] The Norwegian Agency for Development Cooperation NORAD. Effectiveness and sustainability assessment of hiv/aids awareness and preventive project of free pentecostal fellowship in kenya. http://www.norad.no/en/tools-and-publications/publications/publication?key=117500, 2007.

[28] World Health Organization. Breast cancer control. http://www.who.int/cancer/detection/breastcancer/en/index3.html.

[29] Dewayne E. Perry, Adam A. Porter, and Lawrence G. Votta. Empirical studies of software engineering: a roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, pages 345–355, New York, NY, USA, 2000. ACM.

[30] Foyzur Rahman, Christian Bird, and Premkumar T. Devanbu. Clones: What is that smell? In Jim Whitehead and Thomas Zimmermann, editors, *Proceedings of the 7th International Working Conference on Mining Software Repositories, MSR 2010 (Co-located with ICSE), Cape Town, South Africa, May 2-3, 2010, Proceedings*, pages 72–81. IEEE, 2010.

[31] Arthur J. Riel. *Object-Oriented Design Heuristics*. Addison-Wesley, 1st edition, April 1996.

[32] John Ferguson Smart. *Jenkins - The Definitive Guide: Continuos Integration for the Masses: also Covers Hudson.* O'Reilly, 2011.

[33] John S. Smart. Which open source ci tool is best suited for your application's environment? http://javaworld.com/javaworld/jw-11-2006/jw-1101-ci.html.

[34] ThoughtWorks. Ci feature matrix. http://confluence.public.thoughtworks.org/display/CC/CI+Feature+Matrix.

[35] Wikipedia. Comparison of continuous integration software. http://en.wikipedia.org/wiki/Comparison_of_Continuous_Integration_Software.

[36] Wikipedia. Comparison of issue-tracking systems. http://en.wikipedia.org/wiki/Comparison_of_issue_tracking_systems.

[37] R.K. Yin. *Case Study Research: Design and Methods*. Applied Social Research Methods Series. SAGE Publications, 2002.

Appendix A

Terminology

Some explanation about much used technical terms helps to broaden the audience. Throughout this research paper some technical terms and ideas are used a lot. It is therefore essential to understand them in order to fully understand this research paper. Below is an enumeration for these much used terms and ideas.

- **Artifacts** are by-products produced during software development. Examples in this context are files which consist of test suite outcome or files that contain the outcome of several metrics, etc.
- **Continuous Integration** or **CI** is the process of continuously building an entire software project which is still under development. This with the purpose to test if the complete project still operates in the correct way. Some also call it "applying constant quality control on a software project under development". Several tools have been developed to enable this process. Some examples include Jenkins[1] (forked project of a project known as Hudson), CruiseControl(.Net)[2], Apache Continuum[3], Microsoft Team Foundation Server[4] etc. For more information, see chapter 3.2.
- **Software Quality Metrics** or simply **metrics** can be defined as being a measurement of a certain property of software. Simple metrics are for instance the total amount of lines of code, or average amount of bugs per line of code. Metrics are part of the means to define the software quality of software products. The theory behind these software quality metrics is explained in depth in chapter 2.
- **Integrated Development Environment** or **IDE** is a development environment in which various tools are integrated that help the developer in his work. These

[1] http://www.jenkins-ci.org

[2] http://cruisecontrol.sourceforge.net and http://www.cruisecontrolnet.org respectively

[3] http://continuum.apache.org

[4] http://www.microsoft.com/visualstudio/en-us/products/2010-editions/team-foundation-server/overview

tools can be used to build, run, test (part of) the implementation. Some tools are also able to deploy the succesfully built source code. Examples include Eclipse[5], Microsoft Visual Studio[6], Netbeans[7], JDeveloper[8] etc.

- **Software Configuration Management** or **SCM** denotes multiple tools that are used for building the software, configuring the hard- and software of the system, and tracking defects that have been identified in the software. In other words, it deals with everything around the development of a software project.
- **Version Control System** or **VCS** is the group of applications that is capable of storing source code while maintaining a revision list of each of these files to enhance team productivity. Some examples are Subversion[9], Bazaar[10], Mercurial[11] and Git[12].
- **Issue Tracking System** or **ITS** is a system which is used to report and keep track of the status of software bugs. Many different systems exist that provide this functionality for instance Bugzilla[13], Mantis[14], Jira[15], Google Code[16] etc.
- **International Organization for Standardization** or **ISO**[17] for short is the (non-governmental) organization responsible for reaching consensus on developing standards. Because many countries are embodied in this organization the standards are widely used.
- **International Electrotechnical Commission** or **IEC**[18] is a sister organization to ISO. IEC is one of three global non-governmental organizations (next to ITU and the aforementioned ISO) which develop standards. The IEC focuses on electrical, electronic and other related technologies.

[5] http://www.eclipse.org
[6] http://www.microsoft.com/visualstudio/
[7] http://www.netbeans.org
[8] http://www.oracle.com/technetwork/developer-tools/jdev/
[9] http://subversion.tigris.org
[10] http://bazaar.canonical.com
[11] http://mercurial.selenic.com
[12] http://git-scm.org
[13] http://www.bugzilla.org/
[14] http://www.mantisbt.org
[15] http://www.atlassian.com/software/jira/
[16] http://code.google.com
[17] http://www.iso.org
[18] http://www.iec.ch

Appendix B

Interview with Joost Koedijk on January 26th 2012

This interview was originally planned at an earlier date, but due to sudden schedule changes the meeting had to be rescheduled. Joost is partner at KPMG and he holds the position for Software Quality assurance. As a person with many years of experience in large IT projects in several areas, Joost is a very knowledgeable source for questions related with Software Quality. The questions that Joost were asked are therefore not of an introductory nature but rather aimed at the real content of the research.

After having introduced the topic, Joost began to tell a story that he was the technical architect for a project of the government when the Social Security Number was being reformed to the new "Burger Service Nummer" (in Dutch). The government wanted to have the produced software tested by a third party, so the source code was sent to a company called "The Software Improvement Group", SIG for short. SIG told the team leader that they had never seen better code. Joost was asked if he knew why this was the case. Joost said that the things they check are basically a few metrics:

- Cyclomatic Complexity
- Compliance to the Coding Standard
- Amount of Duplicate Code

Another two important aspects were:

- Number of Lines of Code (of each developer)
- Unit Test Coverage >90%

Again, Joost was asked why only these metrics seemed so important. He answered that it was not only because of these metrics, but also because of good architecture. If the architecture was good, the developers should only follow the rules defined by

the architecture in order to deliver a higher quality product than without a proper architecture. If the architecture for instance tells the developers that data manipulation should be done in program code, and the programmers implement data manipulation code in the database (as stored procedures for instance), it might very well be the case that scalability is going to be an issue later on in the project 's lifetime, even though the architects thought of scalability but unfortunately the developers did not. After a longer period of time the application may have a bad response time due to the database being a bottleneck.

The interview went on about the usefulness of using metrics to improve software quality. Joost was convinced that a few are important, but the majority is not. As I was surprised by this answer, I asked to elaborate on the matter. Joost told a story of him refactoring some old code that was run very often. He was able to speed up the execution of the code by an average rate of 300% faster, but nobody noticed any changes because the system was hidden in some middleware but which was frequently used. The functions were using asynchronous calls which also made it hard to notice. The result could probably be seen in the fact that a new machine did not have to be bought anytime soon. In theory, because execution was sped up, the application should be better to scale. Joost then went on telling it was not interesting for him (anymore) that software was written in an efficient manner, rather in an increasing business value manner.

The statement above describes an interesting proposition, namely that it does not matter if only the software quality improves. It actually matters if some other goal is attached to the fact that the software quality improves. For some, the goal is to increase business value by improving software quality in some way, for others, the goal might be to have software that can be adjusted to ever-changing requirements with the least amount of effort. Obtaining a good score in a SIG model could also be a goal to pursue. **This all makes me wonder if the title of my thesis should be further elaborated to cover this idea.**

Joost then told me that he already setup Sonar on several sites, but when the job was done and he left, it never took a long time for people to neglect the Sonar installation or even to forget it altogether. The advice he gave was to make somebody responsible for the Sonar installation and the whole reporting procedure.

As a final advice Joost told me that metrics and characteristics should be divided into groups of different importance. Characteristics as maintainability will not change on a daily basis so this should not be included in a daily report intended for developers. Weekly or monthly reports intended for project managers would benefit from having these statistics because project management might want to intervene if the project threatens to evolve in a wrong way.

Appendix C

Interview with Anton Gochev on February 9th 2012

This interview was - just like Joost's interview - originally planned at an earlier date, but due to sudden schedule changes the meeting had to be rescheduled. The interview was more like a long conversation than a real interview. At first I explained Anton the purpose of my research and the way I was going to obtain a solid and reliable answer to the research question.

From the start Anton went on by stating that there are always excuses for not actively monitoring software quality and acting upon undesired software quality results. The most common excuses are the fact that there is no time for active software quality monitoring (this is equal as not having the proper financial resources) or the fact that no one already has extensive knowledge of this field so a proper software quality process can be implemented.

The discussion went further on about the definition of software quality. I asked Anton if he knew a definition of software quality. Anton reluctantly said: "Well, Software Quality is the Quality of the software product under review." Although true, I asked him if he could somehow elaborate on this. He told me software cannot be touched, so software should be seen as a service one provides. It is the same with the service provided by lawyers. You don't understand anything about their field of operation, but when you hire them for some case, you trust them to handle in your best interest. Of course, some professions need tools, for instance a car garage. The mechanics use tools to fix your car. But how is their provided service experienced? Which garage holder will have the most recurring customers? The ones that had to return to the garage due to incorrect work but while they were treated nice and with honesty? Or the ones whose car was fixed the first time they brought it in but were treated bad. It is not always about the functionality, but a great deal in the satisfaction of the customer is the user-experience he or she has with the service provided.

Another example Anton gave was that some CEO might want to invest in an industry-line machine of $50M. He will not be interested in the software that operates the machine, he trusts the manufacturers of the machine.

Since there is no formal definition for software quality, Anton calls it an agree-

ment between the customer and the producer of the software. In the end it is all about identifying if the software meets the software requirements of the customer. So, as I already found out in the interview with Joost, software quality is hard to perceive in itself, it should always, or it already is connected to some other incentive.

Anton agrees on the point made by Joost in which he told me that it is necessary to make someone responsible for the software quality in a team, otherwise it is easily neglected.