Linköping University | IDA
Master Thesis | Computer Science
Spring 2016 | LIU-IDA/LITH-EX-A--16/047—SE

IMPLEMENTATION AND EVALUATION OF A CONTINUOUS CODE INSPECTION PLATFORM

Tomas Melin

Handledare/Tutor, Cyrille Berger, Wang Tiantian, Christian Svedin, Magnus Grimsell
Examinator, Kristian Sandahl

LINKÖPINGS UNIVERSITET

硕士学位论文

Dissertation for Master's Degree

(工程硕士)

(Master of Engineering)

持续的代码审查平台的实现与评价

IMPLEMENTATION AND EVALUATION OF A CONTINUOUS CODE INSPECTION PLATFORM

王维

哈爾濱工業大學 *Linköping University*

2016 年 9 月

国内图书分类号：TP311　　学校代码：10213
国际图书分类号：681　　密级：公开

工程硕士学位论文
Dissertation for the Master's Degree in Engineering
(工程硕士)
(Master of Engineering)

持续的代码审查平台的实现与评价

IMPLEMENTATION AND EVALUATION OF A CONTINUOUS CODE INSPECTION PLATFORM

硕 士 研 究 生： 你的姓名
导　　师： HIT 王甜甜　副教授
副　导　师： LiU　导师姓名、职称
实习单位导师： 实习单位导师姓名、职称
申 请 学 位： 工程硕士
学　　科： 软件工程
所 在 单 位： 软件学院
答 辩 日 期： 2016 年 9 月
授予学位单位： 哈尔滨工业大学

Classified Index: TP311
U.D.C: 681

Dissertation for the Master's Degree in Engineering

IMPLEMENTATION AND EVALUATION OF A CONTINUOUS INSPECTION PLATFORM

Candidate: Tomas Melin
Supervisor: Prof. Wang Tiantian
Associate Supervisors: Prof. Kristian Sandahl, Cyrille Berger
Industrial Supervisors: Christian Svedin, Magnus Grimsell
Academic Degree Applied for: Master of Science
Speciality: Software Engineering
Affiliation: School of Software
Date of Defense: September, 2016
Degree-Conferring-Institution: Harbin Institute of Technology

摘　要

建立和保持高水平的软件质量可以带来经济利益等诸多好处，然而这是一项很困难的任务。其中一种防止软件项目质量下降的方法是通过跟踪项目的度量值和某些属性，来查看项目的属性的变化情况。通过引入持续的代码审查和应用静态代码分析方法可以实现这种方法。然而，在人们的印象中，这类工具往往具有较高的误检，因此需要进一步调查实际情况、研究其可行性，这是本文的初始研究目标。本文在瑞典林雪平的 Ida Infront AB 公司开展了案例研究，调研了该公司开发人员的意见，并通过访问开发人员，确定持续的代码审查平台 SonarQube 的性能。作者对持续的代码审查环境进行了配置，分析了公司的部分产品，进而确定哪些规则适用于该公司。调查结果表明该工具是高质量并且准确的，还提供了持续监测代码来观察度量值的趋势和进展等先进功能，例如通过监测环路复杂度和重复代码等度量值，来防止复杂度和重复代码的增加。通过组合误检压缩、对 pull requests 的瞬间分析反馈、以及分解和建立给定的条件等特征，使得所实现的环境成为一种可以降低软件质量保障难度的方式。

关键词：静态代码分析，持续代码审查，SonarQube，软件质量

Abstract

Establishing and preserving a high level of software quality is a not a trivial task, although the benefits of succeeding with this task has been proven profitable and advantageous. An approach to mitigate the decreasing quality of a project is to track metrics and certain properties of the project, in order to view the progression of the project's properties. This approach may be carried out by introducing continuous code inspection with the application of static code analysis. However, as the initial common opinion is that these type of tools produce a too high number of false positives, there is a need to investigate what the actual case is. This is the origin for the investigation and case study performed in this paper. The case study is performed at Ida Infront AB in Linköping, Sweden and involves interviews with developers to determine the performance of the continuous inspection platform SonarQube, in addition to examine the general opinion among developers at the company. The author executes the implementation and configuration of a continuous inspection environment to analyze a partition of the company's product and determine what rules that are appropriate to apply in the company's context. The results from the investigation indicate the high quality and accuracy of the tool, in addition to the advantageous functionality of continuously monitoring the code to observe trends and the progression of metrics such as cyclomatic complexity and duplicated code, with the goal of preventing the constant increase of complex and duplicated code. Combining this with features such as false positive suppression, instant analysis feedback in pull requests and the possibility to break the build given specified conditions, suggests that the implemented environment is a way to mitigate software quality difficulties.

Keywords: Static Code Analysis, Continuous Code Inspection, SonarQube, Software Quality

Acknowledgment

For a start, the author would like to thank his supervisors; Kristian Sandahl, Cyrille Berger and Wang Tiantian for their guidance during the author's work, it has been invaluable. Equally important, the author would like to acknowledge Christian Svedin and Magnus Grimsell at Ida Infront AB for the grateful opportunity of performing the master thesis project at their company. It has been an incredible eye opening experience in being a part of the company during this project. In addition, the author would like to communicate his appreciation to the interviewees at Ida Infront AB who were willing to contribute to this paper by being interviewed and observed as they performed code reviews in rather spartan conditions. In order for these interviewees to remain anonymous, their names will not be listed.

The author would also like to express his gratitude towards his companions, Daniel Andersson and Robert Krogh, who have guided the author during the progression of this project in regards to brainstorming and solving problems related to both the theoretical work and the practical assignment executed at Ida Infront AB.

Table of Contents

Table of Figures

Table of Tables

Chapter 1 Introduction

Maintaining a high quality software is an objective in many software projects [1], but the amount of resources that are allocated to achieve this objective may differ. Software quality is defined as the degree that the software meets the specified requirements [2]. Where software quality may be further defined using quality attributes, such as usability or maintainability. To allow quantitative measurement, quality metrics have also been declared. These metrics determines the level of the specific quality attribute that has been fulfilled.

Studies have confirmed that a higher software quality has a positive effect on the overall maintenance costs [3]. The quality of the code has in many cases been approved by the passing of test cases. While this implies that the code performs all the necessary tasks, the passing of certain tests does not certify the quality in terms of code conventions and other types of faults which can escape the conventional testing procedure.

Applying static code analysis tools to a code base can be performed in various ways, where the most common is for developers to have a command, which they run from their terminal or IDE to control that they follow their pre-decided code conventions. This approach may seem sufficient to the specific developer and his contributions, however, given a team of developers whom all contribute to the same project, the complexity of coordinating the code quality is increased and should be handled using a different approach. Since the functionality of each static code analysis tool varies, it is important to be cautious when selecting a tool to deploy for your setting. The reason for this fact is that defects exist even in thoroughly tested software written by experienced developers and that it does not require a tremendous amount of effort to perform an automatic static analysis control to identify these software anomalies. The source of these issues or defects may be misunderstood concepts or functionalities in the programming language which may not be detected in conventional testing [4].

However, solving these bugs in a convenient and productive way is a far more delicate issue. To solve this issue, a solution is to use a continuous code inspection platform to coordinate several different static code analysis tools. By using continuous inspection the metrics collected by several static code analysis tools will be presented

in one location where they can be evaluated and compared with previous values making the software quality more comprehendible and the monitoring becomes manageable to overview [5]. This is demonstrated in Figure 1-1 that contains graphs of the duplicated code and lines of code metrics, combined with latest modification affect.

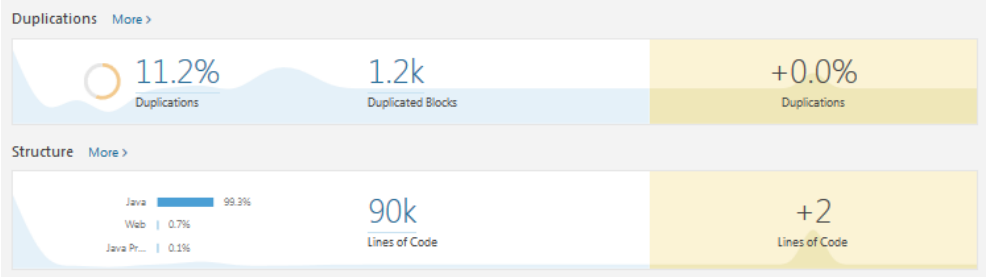

Figure 1-1: Demonstrative example of how the monitoring of metrics may look.

1.1 Background

This report contains a case study executed in the software development industry at the company Ida Infront AB. This chapter is intended to introduce the readers to the company and the context of which the case study is conducted. This master thesis project is a part of a Double-degree agreement between Harbin Institute of Technology(HIT) in Harbin, China and Linköping University(LiU) in Linköping, Sweden. The author has studied one semester at HIT followed by performing the master thesis project satisfying the requirements for both universities. Supervisors from both universities have been included in the thesis process, in addition to supervisors at Ida Infront AB.

1.1.1 About the Company

Ida Infront AB is a well-established company with many years of experience in case management, digital archiving and secure communication. The company was founded in 1984 and has their headquarters in Linköping, Sweden. The customers of Ida Infront are primarily found within the public sector. Ida Infront helps their customers to solve their needs by implementing solutions based on their own product family, iipax. The company has offices in Sweden (Stockholm, Linköping), Norway (Oslo) and India (Thane). Ida Infront has around 70 employees and is a part of Addnode Group and this project will be conducted at their office in Linköping, Sweden. In this thesis, Ida Infront AB will be referred to as the internship company.

1.1.2 Context

The internship company has investigated the opportunities of implementing static code analysis in their development process but has not found the generated feedback to be sufficiently comprehensible. They also considered there was a high number of false positives presented among the anomalies, causing the code inspection process to require more time and resources than what was initially allocated. Resulting in the lack of the essential benefit from the static code analysis. Further investigation is needed to determine the possibilities of implementing static code analysis in their development process. The code base to be used in the experiments has been in development for more than fifteen years, which has a tendency to result in a certain amount of legacy code. The code base is constructed as a plugin-based framework in order to make the software easy to adapt according to specific customer requirements.

1.2 Motivation

Ensuring that code is of excellent quality is an activity that is complicated to execute since there are various ways to perform these controls of quality. A well-known approach is the conventional code review that is executed by a physical person studying and analyzing the work by another person. Panichella et al. [6] perform a study where they investigate whether a code review would be improved by the addition of a static code analysis tool. The results from the study display how the warnings found in the source code are only reduced slightly for each code review and the overall percentage of removed warnings were between 6% and 22%. According to the authors [6], they found that the developers have a tendency to target a certain type of problems which results in the deletion of between 50% and 100% of these problems. As humans are not able to investigate a code base in the same sense as a computerized tool, resulting in the focus of one area or another.

There are several methods for performing manual code reviews. Likewise, there are also a high number of automatic tools to assist the reviewers. The results of tool supported code reviews have also been proven to find higher numbers and a more varying number of anomalies [7], [8].

While it may seem tempting to apply tools in this context to solve the human errors completely, it is not certain that the tools applied will perform the task as intended. If the tools are not properly configured, the results may be misleading.

Although, given the correct configuration the output from the tools may indeed be very useful [6].

Another valuable contribution made by Panichella et al. [6] was the conclusion that a higher number of warnings were fixed using static code analysis tools compared to projects not taking advantage of these tools. Automated static code analysis has also been proven to be very useful for detecting software anomalies in early phases of software development [9]. And by using an automatic static analysis tool which detects and lists anomalies according to a preset prioritization technique, the developers may focus their anomaly inspection of the areas who they are interested in [9].

1.3 Purpose and Aim

The concept of code reviews is an important step in software development as a step to verify the code quality while sharing experiences and knowledge among the employees [10]. To investigate this area further, the author has, in agreement with the internship company, decided to evaluate and implement a continuous code inspection environment using static code analysis tools. The evaluation will be conducted in terms of assessing the accuracy of the produced issues of the continuous inspection environment. Additionally, the author has been assigned the task to investigate how feedback from the continuous inspection environment may be used to improve the architecture and design of the code base, in addition to provide support during code reviews.

1.4 Research Questions

To fathom the generated output from a static code analysis environment the values produced should be evaluated and weighed, to enable the determination of the usefulness of this output. This is the reason for RQ1(Research Question 1) and RQ2. To investigate the difference between the implementation of several static code analysis tools and how they may be implemented in a continuous code inspection environment, RQ3 were constructed.

RQ1. How can the design and architecture of a code base be improved using output from static code analysis?

RQ2. How may static code analysis be used in order to find defects in the code?

RQ3. How may a continuous code inspection platform be used in an agile environment to find defects in the code?

In the following chapters and sections, the research questions will be referenced using the RQX format, where X is the number referencing to a research question.

1.5 Delimitations

This project is limited to investigating how the continuous code inspection tool SonarQube [11] may be applied to find faults in a software development project, not focusing on comparing this tool to other continuous inspection tools in similar contexts, but instead perform an evaluative investigation of the performance of these tools.

The focus of the evaluation resides in the resulting output, in terms of produced recommendations and specified anomalies rather than an evaluation of the qualitative aspects of the SonarQube software as a product.

The material used in this study is provided by the internship company, resulting in a highly specific context that the configuration is adapted to. Properties that applies in this context may not be applicable to other scenarios where the code based is constructed differently, such as written in another programming language than Java.

1.6 Approach

During the initial phase of this project, the author produced a planning report that included a time plan in the form of a Gantt-chart to be used as a continuously updated planning chart with the purpose of monitoring the status of the writing of this report in addition to the project executed at the internship company. The planning of this project included the research presented in this report and the project at Ida Infront, since these projects were planned as two separate but related tasks with a number of dependencies, the scheduling of the assignments had to be carefully considered in order to prevent accidental halts during the progress of the project.

1.6.1 Literature Study

Previous to the implementation and configuration phase, the author was required to obtain further knowledge to gain a broader and deeper understanding of the static code analysis and continuous inspection area. Another essential aim of the literature study was to educate the author of the available static code analysis and continuous inspection tools to allow the author to elect the most appropriate tool for the project along with the most contributing aim of the study. The author considered whether to continue on previous studies performed by other researchers in similar contexts or

pivot from an existing paper's conclusion to investigate new possibilities. The resulting approach was somewhat of a combination, the author chose to perform an evaluative approach to investigate the usefulness of a static code analysis and continuous inspection in practice in the setting of the internship company.

To provide the reader with support to replicate the results of this study, the method phases of this project will be introduced and described.

1.6.2 Setup

The initial phase for this project was the configuration and setup of the SCM server, automation server and continuous inspection server. In addition to configuring the internal settings for each entity, the communication between these three entities has to work properly. Details for this setup are described further in Chapter 5.

1.6.3 Rule Configuration

The rules that are to be applied in the continuous inspection have to be configured in the SonarQube interface. There were several steps taken by the author to adapt the rules, which SonarQube should use to monitor the code base and detect issues to improve the quality of the code base. The initial step was to perform an analysis on the entire code base provided. However, due to the large number of alerts detected, the code base had to be divided into smaller divisions. One of the sources of this large number of alerts is that the code base provided was huge, containing about two million lines of code. Another factor, which influences the large amount of found alerts by the continuous inspection, was that the only previous static code analysis tool, which had been applied previously in the development process at the internship company, was Checkstyle [12]. This tool had been configured to control syntactical and esthetical rules during development. However, the most significant source to the large number of alerts is the setting of the SonarQube tool, i.e. the configuration being set to its default values. Using the default settings of a continuous inspection tool may be appropriate from start, although, it is highly recommended to configure the platform according to the context in addition to what type of alerts that are desirable to detect, in order to make the most out of the platform.

In order to configure the rules to produce alerts in this setting, the author investigated the packages in the code base. The packages that had the highest number of alerts were the targets of this investigation. An alternative approach would have

been to choose a module randomly or selected by consultancy by the author's supervisor but this approach was deemed less appropriate in the sense of improving the software quality in this investigation. The modules which contain the highest number of alerts were selected and presented to the supervisor who provided feedback regarding the selection and prioritization of packages. They were investigated further in order to accomplish as valuable and interesting result as possible. The packages found to be the most interesting were discussed with the author's supervisor and manager. The purpose of this consultation was to be confident that the most appropriate packages were chosen to be further investigated. These packages would also be used to perform the rule configuration on. Once the most interesting package had been decided and selected to perform the rule configuration on, the author performed an initial rule configuration investigation in order to examine how the tool operated and what functionalities were available. This investigation consisted of analyzing the violated rules in SonarQube and what type of issues that were detected. Since some properties of the code base may be unique and rather specific to the context, this investigation was followed by an additional investigation performed with the assistance of the supervisor of this project to ensure that the rule configuration was executed as accurate as possible to match the actual setting of the code base. An additional motivation for performing this second investigation in collaboration with the author's supervisor, was to be able to adapt the rules to the most accurate setting. The investigation began with monitoring what alerts were detected, starting with the alerts ranked as the most severe type by SonarQube [11] and the highest frequency. For each rule that produced alerts, the alerts were investigated by the author with the consultancy of his supervisor. To control whether the rule was applicable in its context, due to either differentiating coding conventions or properties of the code base which does not collaborate well with the rules stated in SonarQube.

First, the rule was inspected to check if it was relevant and useful for the development setting of the company, followed by being estimated whether it would produce a significant number of false positives. Second, the alerts produced by the rule were studied and analyzed by determining the conformity of the rule and the produced alerts. If the rule was deemed useful before beginning the study of the alerts, in addition to the majority of the alerts were deemed true positives by the author and his supervisor – the rule was decided to be applicable to the code base. However, if the usefulness of the rule was uncertain, extra caution was used during the investigation of alerts to

ensure that the decision whether the rule should be applied or not was carefully considered.

Once all alerts that were detected had been dealt with, each rule was altered to be either a Blocker or a Major, depending on their severity for the code base, the phase of evaluating the found anomalies by using an alert oracle, as introduced by Heckman et al. [13] and described in this work in Section 2.2.3.1. Where Blocker and Major are severity rankings in SonarQube and in the configured environment, Blocker are issues that would fail the build if contained in the contributed code. While Major issues are issues that would act as warnings to the developers. In this project the alert oracles have been in the form of developers at the internship company, by performing interviews using a subset of the alerts found in the static code analysis results. By selecting a number of alerts that represent blocker and major issues combined with prioritizing the number of alerts that were more frequent in the analysis, a representative set of alerts has been established. By applying the FAULTBENCH process, described in Section 2.2.3.1, to this set of alerts, an evaluation of how well the static code analysis has performed may be indicated by applying the precision, recall and accuracy metrics, introduced in Section 2.2.3.1. Next, the alignment and order of the content in this thesis will be presented.

1.7 Main content and Organization of the Thesis

As this chapter has introduced to the reader to the origin and aim of this thesis, this section is intended to guide the reader to this document to enhance the experience of studying this paper.

Subsequent to this chapter, Chapter 2 Theoretical Framework will present the fundamental research that is essential to this topic. The chapter may be divided into three major topics, software quality metrics, static code analysis and continuous inspection in addition to the presentation of current research that this project is built upon in addition to compare this paper's contribution with similar work.

Additionally, Chapter 3 System Requirement Analysis introduces the process of designing and defining the requirements of the implemented system, in terms of architecture and functionality.

Next, Chapter 4 Design and Development of the System that is intended to, in detail, describe the components that are used to compose the built system, in addition to describing the process of evaluating the constructed system.

Chapter 5 Case Study is destined to describe the context of this project, i.e. introduce the internship company, Ida Infront AB, in addition to the material that will be subject of the analysis during this project.

Furthermore, Chapter 6 Resulting System and Evaluation contains the detected results and evaluates these results as described in Chapter 4. In addition to the resulting system, Chapter 6 also describes the most important functionalities and features of the implemented environment that are vital components in order to find defects in the code.

Chapter 7 Discussion discusses the previously presented methods and results to highlight the benefits and drawbacks of the implemented system and the found results in the analysis partition of the system. This chapter also contains the discussion of the work in a wider context.

Finally, the Conclusions chapter defines and summarizes the aim and research objective to at last, state the outcomes and contributions of this work followed by describing future work approaches.

Chapter 2 Theoretical Framework

This chapter will describe the contents of the theoretical foundation applied in this thesis report and set the level of knowledge required to grasp the contents of this thesis.

As defined by the IEEE Standards Association [14], the concept of software quality may be described as the capability of a software artifact to comply with stated and required needs once used in a certain setting. Maintaining a high software quality in projects is a requirement rather than an option to achieve success in software development projects. The level of software quality also affects what customers that are able to keep and attract to a business [15]. Given good software quality, maintenance activities in most projects cost a significant amount of resources [16]. This results in the opportunity to reduce these costs, as found by Emam [3]. Emam states that there are a number of evidences which shows that a higher software quality reduces the maintenance costs during the entire product lifecycle [3].

2.1 Metrics

There are a number of metrics, which have been used to determine the quality of code bases, however, the focus in this section will reside on the metrics which are applied and discussed at a later stage in this thesis.

2.1.1 Complexity

The concept *complexity* is in most occurrences used in terms of an external characteristic, thus including the concept of describing a system as being psychological complex [17] and measuring a system's control complexity [18]. This meaning of the word have influenced the software complexity research to the extent that the research is implicitly or explicitly aimed towards this focus [17].

In order to solve the issues with extensive time and costs being spent on maintaining and testing software systems, McCabe attempted to develop a mathematical approach to resolve the issues with software having a too high number of control paths [19]. The approach involved dividing the program into vertices and edges, where vertices are code blocks and edges are branches. The *cyclomatic number*

$V(G)$ of a graph G with n vertices, e edges and p strongly connected components can be defined as:

$$V(G) = e - n + 2p. \quad (2\text{-}1)$$

A connected component is defined as a component where every vertex is reachable from every other vertex and a strongly connected component is a connected component with the addition that the graph is directed such that if we add any vertices or edges to the graph, it is not a connected component anymore [20].

Using Equation (2-1) the following theorem may be stated:

"Theorem 1: In a strongly connected graph G, the cyclomatic number is equal to the maximum number of linearly independent circuits." [19]

By applying this theorem to a program and associating it with a directed graph with unique entry and exit nodes, a graph can be constructed to illustrate the cyclomatic complexity properties. Each code block in the program will be illustrated as a node and each arc will be represented branches in the program [19]. By constructing two smaller examples of program control graphs, which may be viewed in Figure 2-1 and Figure 2-2, the relationship between the control path and cyclomatic complexity is easy to detect [19].

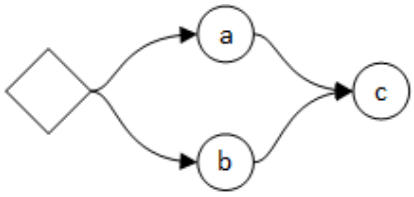

Figure 2-1: Program control graph for a simple if-then-else-case.

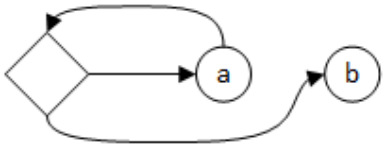

Figure 2-2: Program control graph for a simple while-loop case.

The cyclomatic complexity of the program control graphs in Figure 2-1 and Figure 2-2 may be calculated using Equation (2-1) [19]:

Figure 2-1: $V_1 = 4 - 4 + 2 = 2$

Figure 2-2: $V_2 = 3 - 3 + 2 = 2$

The graphs that have been constructed are also known as the program control graphs and it is assumed that each node can be reached from the initial node and that

each node may also reach the exit node. The complexity of a program may be estimated by computing the number of linearly independent paths [19].

Figure 2-3 is an illustrative example of how cyclomatic complexity may be calculated for code, the figure depicts how cyclomatic complexity may be calculated for Java code using SonarQube's metric definition for cyclomatic complexity, as defined by Racodon [21].

```
public void process(Car myCar){          // +1
    if(myCar.isNotMine()){               // +1
        return;                          // +1
    }
    car.paint("red");
    car.changeWheel();
    while(car.hasGazol() && car.getDriver().isNotStressed()){   // +2
        car.drive();
    }
    return;
}
```

Figure 2-3: Demonstrative example how to calculate the cyclomatic complexity using SonarQube's guidelines.

As exhibited in Figure 2-3, the cyclomatic complexity for the Java method **process(Car myCar)** is five. This is the result of incrementing the keywords **if**, **return**, **while** and **&&**, joint with the fact that each method complexity is initialized to one. Worth noting though, is that the last return statement does not result in the increase of cyclomatic complexity, this is not an error but a property of the metric.

There is no definitive limit for when a system's complexity increases to the point where it becomes too obscure, however, there are several recommendations to adhere to. As stated by Fenton et al. [18] when the cyclomatic complexity exceeds ten in any module, it is probable that problems may occur which implies that the module in question should be refactored to lower the complexity. There are no thresholds for when complexity is deemed a too high number for a function, file or class. According Campbell et al. the complexity of a file should not exceed 60, while the complexity for methods should not exceed seven in order to keep the code understandable and maintainable [22, pp. 96–112].

Criticism against the cyclomatic complexity metric has been raised by arguing that although the complexity measurement constructed by McCabe measures the complexity of a program, the metric fails to differentiate between the complexities

of simple cases where single conditions are used instead of multiple conditions in conditional statements [23].

Similarly, according to Vinju et al. [24], the cyclomatic complexity metric should be cautiously interpreted, as described in their work:

"[...] when applied to judge a single method on understandability, must be taken with a grain of salt."

Vinju et al. have collected empirical data from eight open source Java projects, that establishes how the metric often may underestimate and overestimate the understandability of methods.

2.1.2 Size

Measuring the size of a code unit may be performed in several ways, in most cases the metrics used are lines of code (LOC), number of statements and the number of blank lines [25].

2.1.3 Technical Debt

There are always several approaches to extend the functionality of a system; approaches that require less effort and thought in the moment but might result in difficulties later on when extending the package or class in question. On the other hand, there are approaches that require more energy and struggle as of now but will result in a cleaner and significantly more adaptable design. To aid developers to handle this issue, the metric of technical debt[1], was constructed by applying the metaphor of assimilating technical debt to financial debt, where interest payments are incurred in the form of performing additional effort in future development due to choosing to do inexpensive and unclean design choices [26]. As in the financial world, certain opportunities have to be taken, thus risking resources – similar opportunities may be taken in software development, e.g. to hit an important deadline or deliver a certain feature in time. However, unlike the financial variant, technical debt is challenging to measure effectively – causing the effect of technical debt to be concealed [26].

[1] Technical debt was first introduced by Ward Cunningham (**http://c2.com/doc/oopsla92.html**).

2.2 Static Code Analysis

The process of running an analysis on code without executing the code is known as static code analysis. Compared to conventional testing, this analysis can be performed without the need to design and construct test cases. In this sense static code analysis can be viewed as a conventional code review with the modification that the reviewer (which in most common cases is human) is replaced by a number of tools using statistic data to evaluate whether the code is containing malformed statements or breaking conventions stated by rules in the tools. This makes the tools very useful to apply during the implementation phase to scan the source and byte code for patterns and anomalies. They also allow the static analysis tools to search through the code base independently to find hidden backdoors or other errors which are difficult to detect manually [27].

By using static code analysis tools, the hidden errors can be discovered in the implementation even before the software has arrived at testing or production [7], [8], which is very valuable since errors detected earlier in the development process are less expensive to fix. If defects can be found during the development phase, less effort has to be put in the testing phase in addition to the system becoming increasingly more maintainable and the amount of operations are minimized [7]. Static code analysis tools can also be helpful to discover security problems, however, one should be cautious to replace the manual code review completely with tool supported code review, since both kinds of code review find different types of defects. As these tools use rules and patterns decided by humans, their result should never be viewed as giving the final answer [7].

Determining the software quality of a module is not always a straightforward procedure since software quality comes in many different shapes. By using static code analysis tools to distinguish the difference in software quality between components, this problem can become significantly easier to handle [8].

2.2.1 Static Code Analysis Techniques

There are several techniques and methods, which may be applied using static code analysis tools. To introduce the reader to the various types of methods that are being applied, the following sections will introduce the most common techniques.

2.2.2 Control Flow Analysis

Several aspects of the code may be investigated by executing an analysis using tools or manually at several levels of abstraction, such as modules or nodes [28]:

- The execution sequence may be verified to be correct.
- The organization and structure of the code.
- Code statements who are not syntactically reachable.
- Occurrences in the code which requires to be further investigated to insert required termination statements.

The output of control flow analysis may produce results in the form of visual and graphical representations [28].

2.2.2.1 Data Flow Analysis

Accessing variables that have not been set to a value could result in bugs, which are difficult to find. Data flow analysis investigates whether there are any execution paths in the software that could retrieve the value of a variable which have not been initialized [28]. This type of tools often uses the result of the control flow analysis in addition to read/write access to the variables. As global variables may be accessed from anywhere, this activity may in some cases become rather complex. Another example of what types of detections this technique may discover is the act of multiple writes without intervening reads [28].

2.2.2.2 Information Flow Analysis

Information flow analysis may be used to analyze how the execution of a unit of code generates dependencies between the input and output of this unit [28]. By comparing and verifying the dependencies in the specification to the generated dependencies, the opportunity to analyze and trace the output to the input emerges. This traceability may be very precious in cases where critical output is generated and the source to that output has to be investigated all the way back to the input from the software or hardware interface. German [28] states how information flow analysis may be improved using annotations, i.e. stylized comments to provide documentation

regarding assumptions about functions, variables, parameters and types. By introducing these annotations, the analysis's efficiency may be enhanced since it is given supplementary data related to that portion of the code.

2.2.2.3 Path Function Analysis

Path function analysis may be applied to verify certain properties of a program [28]. Path function analysis will perform an algebraic manipulation of the source text without the requirement of a formal specification. By checking the semantics of each path through a program section or procedure, the analysis produces the relationship between the input and output of a specific program section and some sophisticated tools may even produce expressions, which describe the mathematical relationship between the input and output. The analysis is executed by iterating through the code by assigning expressions instead of values to each variable, thus converting the sequential logic into a set of parallel assignments where the output values are expressed in the form of input values, making the output easier to interpret. For every path consisting of the conditions that cause the path to be executed, the tools will produce an output in addition to the result of executing that path. Path function analysis is also known as semantic analysis or compliance analysis, where semantic analysis may be described as revealing exactly what the code does in all known scenarios for the whole range of input variables for every program section. Although, the need for human involvement is consistently significant in this technique in comparing the tool's output with the specification [28].

2.2.2.4 Byte Code Analysis

In addition to the static code analysis tools analyzing the source code there are also tools that analyze the compiled byte code. While compilers optimize code, the byte code may not mirror the source code, however, working on bytecode is significantly faster which will have a huge impact when having a large code base [27].

Furthermore, the detected anomalies are not certain to be faults, but rather true or false detections, which will be referred to as *alerts*.

2.2.3 Alerts

An important aspect of using static code analysis tools to improve the code base is how the result is presented to the users, which in many cases are the developers,

and by introducing the issues and suggested improvements in a structured and organized way. The risk by not applying this approach is that the feedback from the continuous code inspection will be too overwhelming for the users due to the high number of anomalies found. A related but not equal source which may be the reason why developers are not using static code analysis tools may be the risk of experiencing a too high number of false positives, i.e. the tool found an anomaly which is not an error or a fault [29]. This may result in distrust from the developers to the static analysis tool that may, given enough time, lead to the developers ignoring the output of the static analysis tool. Another possible reason why developers may avoid or simply ignore static analysis tools can be due to being overloaded with tasks and assignments, which may cause them to deprioritize the process of solving issues found by the static analysis tools by considering if the code passes the tests, the code quality is sufficient.

2.2.3.1 FAULTBENCH Benchmark

Heckman et al. has defined a benchmark named FAULTBENCH to be used for evaluating the output from static code analysis tools, by prioritizing and classifying the alerts [13]. The benchmark is created to be used when adaptively evaluating false positive mitigation techniques, and as stated by Heckman et al. [13] adaptive false positive mitigation techniques requires the state of the alerts to be recorded after each inspection. Whereas non-adaptive false positive mitigation techniques would only require the evaluation of prioritized or classified alerts without fixing or suppressing the alerts. The FAULTBENCH process contains an entity that is named *alert oracle* which is the entity considered to have the correct answer whether the alert is a true or false positive and the process is described as follows:

1. Run a static analysis tool against a clean version of the program.
2. Record the original state of the alert set.
3. Prioritize or classify the generated alerts using a false positive mitigation technique.
4. Either by starting from the top of the prioritized list or randomly electing an alert classified as important, examine each alert,
    a. if the alert oracle considers the alert to be an anomaly – fix the alert with the specified change. Rerun the static analysis tool if needed.

b. if the alert oracle states that the alert is a false positive – suppress the alert.

5. After each alert inspection, record the state of the alert set.
6. Once all alerts have been inspected, evaluate the results using the alert classification technique.

The next step of the FAULTBENCH benchmark is to predict whether the alerts are true positives (TP) or false positives (FP). If an alert is classified as a TP when the alert is a TP, the classification is named a *true positive classification (TP_C)*. In the same way, if an alert is classified as a FP when the alert in fact is an indication of an anomaly, the classification is correct and a *true negative classification (TN_C)* has been identified. Similarly, a *false positive classification (FP_C)* is the event where the model predicts that an alert is a TP while the alert in fact is not an anomaly, i.e. not an error in the code. And, lastly, a *false negative classification (FN_C)* is when the model suggests that an alert is a FP when the alert actually is an anomaly [30].

Table 2-1: Classification table slightly altered from Zimmerman et al.

		Anomalies are observed.		
		True	False	
Model predicts alerts.	Positive	True Positive (TP_C)	False Positive (FP_C)	**Precision**
	Negative	False Negative (FN_C)	True Negative (TN_C)	
		Recall		**Accuracy**

To judge the quality of the classification model, Zimmerman et al. [31] recommends the use of the metrics precision, recall and accuracy, as adopted by Heckman et al. [13] as well and illustrated in Table 2-1, in addition to the following definitions:

- **Precision**: defined as the amount of correctly classified anomalies (TP_C) out of all alerts predicted as anomalies ($TP_C + FP_C$), resulting in the following equation:

$$precision = \frac{TP_C}{TP_C + FP_C} \quad (2\text{-}2)$$

The desired value for precision is close to one since it would imply that every detected anomaly actually was anomalies [31].

- **Recall**: defined as the amount of correctly classified anomalies (TP_C) out of all possible anomalies $(TP_C + FN_C)$, leading to the Equation (2-3):

$$recall = \frac{TP_C}{TP_C + FN_C} \quad (2\text{-}3)$$

  As with the desired value for precision, the desired value for recall is also close to one, since it would suggest that the detected anomalies are anomalies [31].
- **Accuracy**: defined as the number of accurate classifications out of all classifications, resulting in the following expression:

$$accuracy = \frac{TP_C + FN_C}{TP_C + TN_C + FP_C + FN_C} \quad (2\text{-}4)$$

  The value of accuracy to strive for is one, which would state that the classified model is perfect and that not a single mistake was made during the classification [31].

In order to perform a correct interpretation of the measurements, the percentage of files, which have defects, has to be known. An example, made by Zimmerman et al. [31] to illustrate the relationship between these measurements, is the case where 80% of the files contains defects and the model classifies 100% of the files to contain defects. In this scenario, the model has a precision of 80%, recall of 100% and accuracy of 80% resulting in a model that is not optimal to predict defects, since two out of the three values are not relatively close to one or, in this scenario 100%. In the study performed by Zimmerman et al. these three measurements are applied to a project at file level and package level, resulting in the precision value slightly above 60% in most cases and low recall values (between 18.5% and 33%), at file level, indicating that only a few of the files containing defects were detected. Although, the precision values are above 60% in most of the cases, implying the correctness of the analysis, i.e. that there are only few false positives.

Dealing with this type of errors may not be straightforward, especially since these numbers of found alerts may be huge, and as the code bases increase in size and complexity the desire for a solution is growing [5].

2.2.4 Tools

As described in previous sections, there are several techniques to analyze code and this section is intended to briefly introduce some of the most common static analysis tools. Some of these will be applied or mentioned in this paper.

- **Checkstyle:** Checkstyle is an open source, development and static analysis tool that attempts to assist the developer to follow a certain code standard or convention during development. Thus, Checkstyle focuses on the style of your code, rather than finding the most critical bug [12].
- **FindBugs:** FindBugs is a static analysis tool that analyzes a project's byte code to find bug patterns that may be defined as a code idiom that in most cases is an error [32]. FindBugs is written in Java and is open source. According to the developers of the FindBugs tool, less than 50% of all alerts are false warnings.
- **PMD**: PMD is an open source, static code analyzer that examines Java code for issues such as: possible bugs, dead code, suboptimal code, overcomplicated expressions and duplicate code [33].

As shown by Hovemeyer et al. PMD and Checkstyle focuses on style issues, causing them to generate a larger number of alerts compared to FindBugs, that is aimed to find "real" bugs [4].

2.3 Continuous Inspection

As stated by Weimer et al. [5]:

"[...] the desire for a silver bullet is as strong as ever."

Weimer et al. [5] uses the representation of silver bullet that symbolizes the solution to the rising problem with code bases increasing in size, complexity accumulatively increasing, product cycle times are reducing resulting in a large portion of software development projects being clogged and having serious issues with the code quality. The search for a solution for these matters has increased during recent years and Weimer et al. suggests a candidate to mitigate the previously mentioned issues, *continuous code inspection*. There is a common belief that by testing a piece of code, provides the assurance that the code is of high quality, which is not true. However, testing is essential to verify functionality of a system but there are some important aspects of testing that states its inefficiency:

- Testing a too complex code base is arduous and in some cases unfeasible.

- The cost of detecting defects using testing is expensive since several iterations of locating and mitigating the defects often has to be executed.
- Verifying functionality using testing is challenging, especially when the functionality to be tested is cluttered behind structural defects.

However, one could question the reason for the code inspection to be continuous, instead of conventional code inspection. Weimer et al. [34] states the drawbacks of conventional code inspection in five descriptive points:

- There is a lack of measurable benefit – it is perceived as discussion-forums causing the contribution to be difficult to quantify.
- There is a tendency for comments to be ignored and modification to be resisted due to arguments that it compiles and passes the tests, such as unit tests.
- Defining rules that are interpreted and followed correctly by all individuals may be challenging.
- Conventional code reviews have a risk of becoming too emotive and confrontational, which could result in reduced productivity of the team.
- It is common for code reviews to end up focusing on irrelevant issues, instead of the crucial aspects of the code.

As a solution to the state of a software development project that has evolved to a project difficult to maintain and extend, Aguiar et al. [35] suggests the continuous code inspection pattern. The continuous inspection approach is supposed to assist the team by detecting problems early in the development process in addition to probe whether the new code complies with the intended architecture and design restrictions set by the team.

There are two main aspects of continuous inspection – inspection moment and inspection type [35], as illustrated in Figure 2-4. The left container represents various inspection types that may be applied in a continuous inspection approach to investigate certain properties of the code base and the current quality of the code, while the right container represents types of inspection moments that the continuous inspection procedure may apply to collect the information to monitor the code base. Metrics generation is one of the most commonly applied inspection types; it extracts various metrics from the source and byte code. By setting thresholds for the metrics for different levels of modules (packages, classes, methods), the measurements may be used as indicators of when the code has to be refactored. The process of

constructing coding rules that are used to manage the code base, also known as *code smells* detection, may also be applied in this stage. Detecting security flaws in the form of SQL injection or cross-site scripting is the focus of other inspection types, called application security checks, which focuses on discovering security vulnerabilities in the code. Architectural conformance involves inspecting the code for patterns that violate the set design and architecture rules or bad dependencies.

By introducing this concept in addition to a continuous inspection tool, reports may be generated to analyze the project's health and draw attention to any alerts that are detected by the rules. Tools of this kind may be executed locally on a developer's machine or run on a continuous integration server that builds the code at specific time intervals or on each code commit [35]. To adopt this approach, the requirement of having a knowledgeable individual to maintain the rules as a part of the process in addition to describing the intended architecture that the rules will uphold, this approach is illustrated in Figure 2-5. There are various ways to present the generated analysis report; several tools provide a dashboard to monitor the status of the code and by using a server to maintain the continuous inspection, the server and the build server may communicate to allow the build to be marked as failed in the event of e.g. issues thresholds being exceeded. Handling the alerts generated from the continuous inspection tool may be dealt with using several tactics. Some teams embrace the tactic of fixing all alerts for the code to be thought of as complete, while an alternative approach is to rank the alerts in categories, according to their consequence, thus allow the adoption of the zero-alerts-policy for only the *worst* type of category.

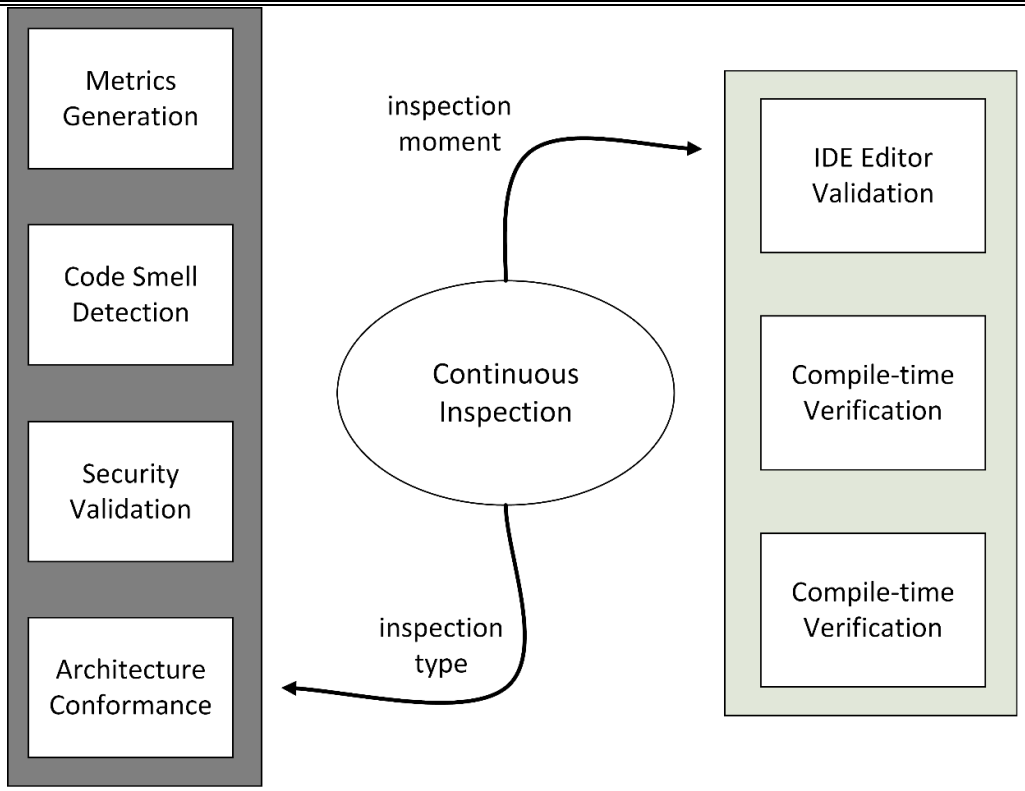

Figure 2-4: The two major aspects of continuous inspection.

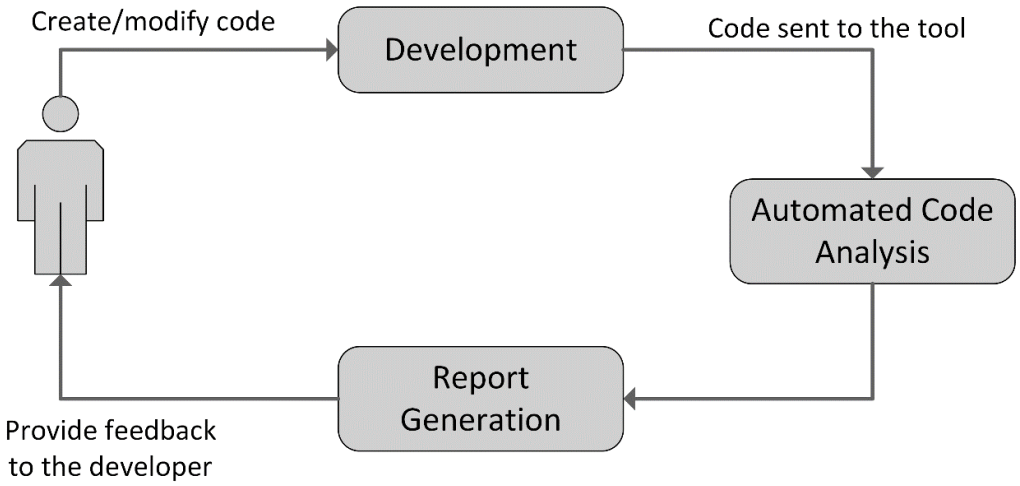

Figure 2-5: Simplified illustration of the continuous inspection procedure.

2.3.1 SonarQube

An example of a continuous code inspection platform is SonarQube which is a web-based application that handles rule alerts, thresholds, exclusions and settings [11]. SonarQube is open source and is marketed as a quality management platform.

The overlaying structure of SonarQube may be described as four main components:

1. SonarQube Server – responsible for starting three major processes:
    a. A Web Server for developers and managers to browse quality snapshots of the code base and configure the SonarQube instance.
    b. A Search Server based on Elasticsearch[2] to enable searching from the user interface. Elasticsearch is a search server and may be used to search in all types of documents. It provides scalable search combined with near real-time search.
    c. A Compute Engine Server to process the produced code analysis reports and storing these in the SonarQube Database.
2. SonarQube Database – used to store the configuration of the specific SonarQube instance, such as security, plugins and settings, and the quality snapshots of projects, views, etc.
3. SonarQube Plugin(s) – to allow certain language features, such as SCM, integration or authentication properties.
4. SonarQube Scanner(s) – to analyze the projects using a build or continuous integration server.

As may be seen in Figure 2-6, which illustrates the architecture of the SonarQube platform [11], the relationship between the components 1-4 are visualized.

[2] www.elastic.co

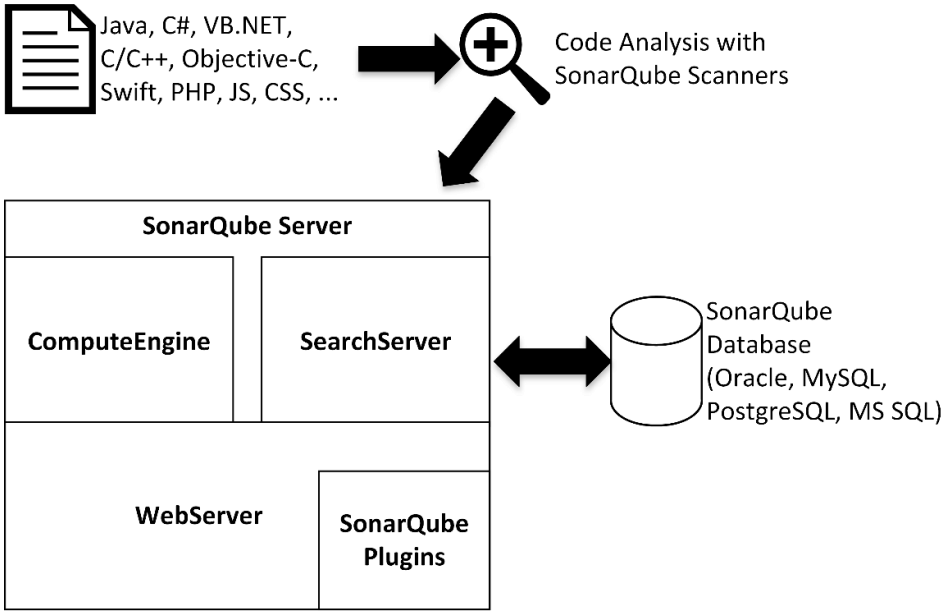

Figure 2-6: The architecture of SonarQube.

2.4 The Status of Related Research

To introduce the stage of the current research in this context this chapter will present what each study has concluded in addition to state the fundamental ideas that are the origin of this paper.

2.4.1 Static Code Analysis Tools

Researchers are aware of the fact that static code analysis tools are valuable and are able to find bugs in different contexts, this has been stated in several papers [4], [6], [7], [27], [28], [36]–[38]. The compared static analysis tools include evaluations and comparisons of PMD, Checkstyle, FindBugs, Coverity, FlexeLint, Squale, Klocwork, CodePro, AppPerfect, ECS/Java2, Fortify, Splint/LCLint, Gendarme, SonarQube and StyleCop among other tools. The evaluations vary in both thoroughness and level of detail, while some studies focus on investigating and comparing the functionalities and capabilities for each tool [7], [27], [28], [36], [38], other studies focus on performing evaluative experiments [4], [6] or retrieving user's opinion about static analysis tools [29]. While these studies cover an expansive range of tools, the common conclusion is that the most appropriate tools recommend the use of PMD, FindBugs, Checkstyle, SonarQube and Squale in various combinations

depending on the goal of the tools and application. Even though the evaluations are thoroughly executed and reported in an adequate sense, the issue of performing the necessary changes in order to improve the code base remains.

During recent years, there has also been shift of the general opinion of static code analysis tools, where the reaction no longer is uncharismatic and negative, it has changed to the far more positive [39].

Given the extensive positive research of static code analysis and its motivation why it should be implemented in every development process, there are also aspects that needs to be considered when introducing static analysis tools. The most important aspect to consider, is the extensive amount of false positives which appear during analysis and that the resources available during development is not sufficient to spend time on correcting static code analysis alerts [29]. This characteristic is also included in this study, i.e. investigating the extensiveness of false positives produced from static code analysis tools in the context, in which this project is performed.

2.4.2 Continuous Code Inspection

Current research in the static code analysis area have developed several high quality tools that may be used to execute continuous code quality control to avoid a decreasing quality level of the code base[40][40][40][40][40][40][40][40][40]. Examples of these results are the tools ConQAT, Teamscale and SonarQube. Steidl et al. [40] have also experienced that many companies have included the process of applying quality measurements to their code. However, due to the pressure of implementing additional features, the act of maintaining the code according to the tools, is not prioritized and becomes forgotten [40]. In order to avoid this scenario, the same authors constructed an enhanced quality control process with the important modification of requiring more manual operation than previously described [41]. The result of their study was the conclusion that software quality analysis cannot be entirely based on automatic measurements, but that the process of analyzing the software requires the significant addition of human evaluation and interaction. By applying this approach to a large number of industry projects there were evidently measurable, long-term quality improvements [41].

Merson et al. [35] performed a thorough investigation of how continuous inspection should be performed and how it should be used with the example of SonarQube's features. Evidently, the paper by Merson et al. is related to the investigation performed in this paper, with the differences that this paper focuses on applying that to an existing software development process and code base, thus making this paper a type of realization of the paper by Merson et al. Their paper includes a comprehensive list of advantages and drawbacks of applying this pattern, which will be valuable for this study to take into account. Although, what their paper does not contain are detailed instructions of how the feedback from the pattern in question may be applied to improve the code quality.

This paper will provide the community with an updated and practical contribution of how static code analysis tools perform in combination with continuous inspection.

Chapter 3 System Requirement Analysis

Provided the knowledge presented in Chapter 2, the structure of the system that are to be implemented will be introduced in this chapter, in addition to the architecture and supportive tools that are required to apply in order to perform the necessary tasks. Another essential subject in this chapter is the description of the requirements.

3.1 The Goal of the System

The main task to be executed by the system is to perform the static code analysis in a continuous inspection context, with the goal of improving the code quality and providing a higher level of quality in future development. This being said, there are several assignments and tasks that need to be considered, such as:

(1) Assist the developers when taking architectural decisions.

(2) Acting as a supportive tool to find, understand and solve defects in the code base, during code review and regular development.

3.2 Requirements Design Process

With the goal of producing clear and informative requirements, the main assignment was initially analyzed thoroughly in order for the author to view the problem from as many approaches as possible before selecting the one to adapt in this project. The first considered approach was use to a number of separate static code analysis tools to perform static code analysis. These tools would analyze the code in different aspects and would allow a practical comparison of the results of the tools.

The second approach was to use a continuous code inspection tool to allow the execution of one the two options:

i. Using several tools in combination to perform an extensive analysis by combining the results from all tools.

ii. Using an advanced tool containing several static code analysis tools to adapt according to the code base and goals of the analysis.

These two approaches were the most promising approaches of static code analysis. By performing a careful comparison of these two approaches, in addition to comparing both alternative options of the second approach, the author concluded that the best approach was to choose the continuous code inspection with multiple static code

analysis tools applied in the analysis approach. The motivation for deciding to adapt this approach originates from current research, that states how the development of the chosen system has become increasingly active and the tools are becoming progressively more sophisticated [40]. The selected continuous code inspection platform also supports the collaboration with an automation server and SCM server, which is very convenient for this system since it will allow the environment development to progress faster.

Once this approach has been set, the requirements may be collected and defined in further detail.

3.3 Requirements Gathering and Analysis Process

To achieve an overview of the environment to be constructed and implemented two diagrams have been constructed. Figure 3-1 describes a high-level view of the setup from the user's point of view and Figure 3-2 is a process diagram to represent and describe the environment from a developer perspective.

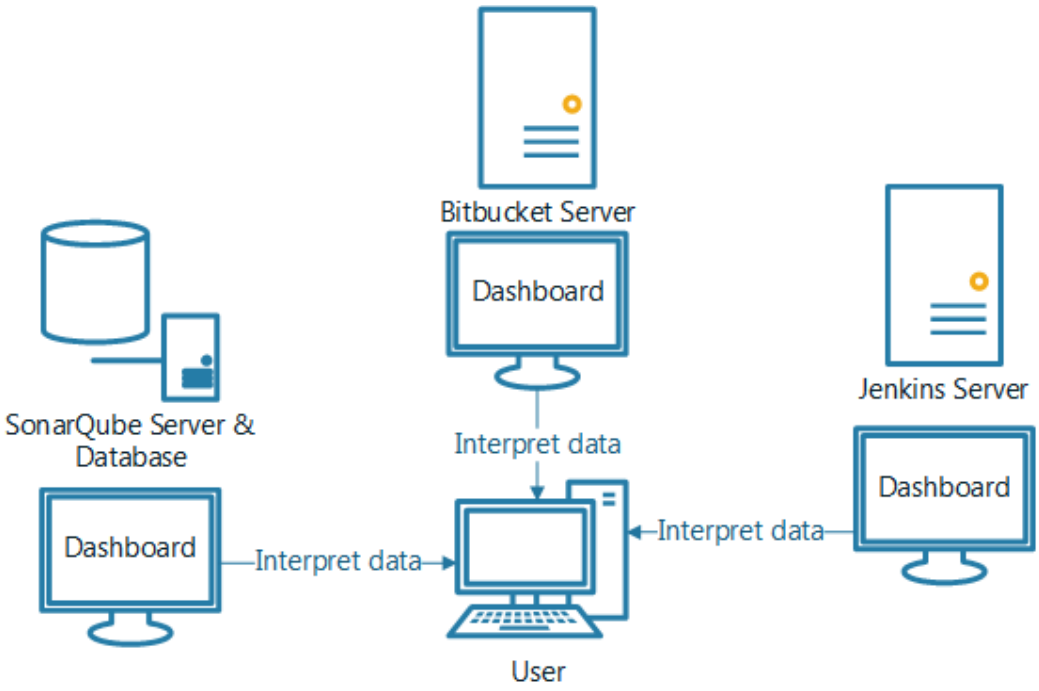

Figure 3-1: High-level view of the user perspective in the development setup.

Figure 3-2 is a modified version of the Enhanced Quality Control Process designed by Steidl et al. [41] and differs by the removal of the project manager and quality engineer. The motivation for this adjustment is that this is the case for the internship company since they do not have a designated person for those assignments and will not involve the project manager at this level of detail at this time. Starting from the upper left corner, the developer will perform an implementation in the system; this system will act as the input to the tool, which performs the analysis of the system. This tool generates the feedback that is presented at the dashboard of the continuous inspection platform. The developers interpret this feedback during the development to provide them with information of how their code base has changed during the sprints, giving them a historical feedback review. By using Quality Goals which defines what aims and achievements to stride for during the projects lifetime will increase the transparency of the process [41].

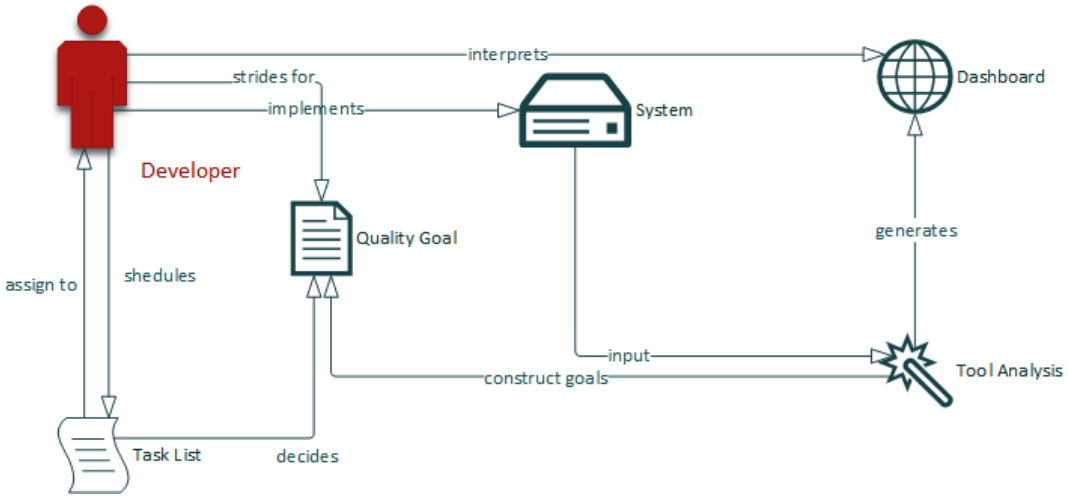

Figure 3-2: Process diagram of the quality control process.

To clarify the usages of the system further, two use cases have been constructed, which may be seen in Figure 3-3 and Figure 3-4. The first use case illustrates the functionality for the user, which in this context is the developer. Figure 3-3 illustrates the actions that may be taken by the developer during the continuous inspection process. The first and probably the most obvious action is the event of performing implementations, i.e. modifying the code base in any way. Next, once this modification has been stored in the SCM, SonarQube will allow the developer to interpret the SonarQube dashboards to view the analysis results. These dashboard interpretations will allow developers to strive for quality goals in their teams or projects, resulting in

distinct objects to aim for during development. The use of task lists would also allow the developers to assign specific tasks to developers who are the most appropriate to solve those kind of issues.

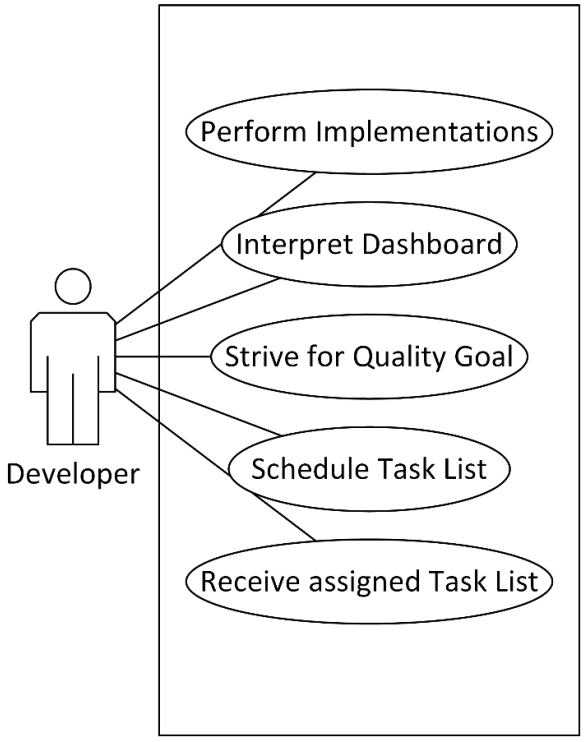

Figure 3-3: Use case diagram from a developer point of view.

The second use case, Figure 3-4, demonstrates the major functionalities provided by SonarQube. SonarQube may generate analysis reports that are presented to developers on dashboards. These reports allow developers to continuously monitor and maintain the quality of the code base. Combining this with the use of quality goals, as mentioned previously, the developers are given the opportunity to track the quality of the code base. Another action that SonarQube provides is the retrieval of the code base from the SCM, i.e. the automatically fetching of the code from, in this case, Bitbucket server.

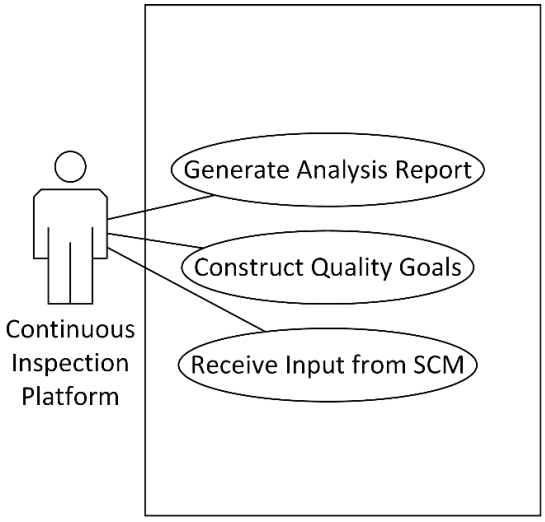

Figure 3-4: Use case diagram from the continuous inspection platform point of view.

Using these diagrams and information provided by the internship company, several functional and non-functional requirements may be stated.

3.4 Functional Requirements

FR1. The environment shall present areas where architecture and design improvements shall be necessary.

FR2. The environment shall allow the developers to detect and study issues, which are found during the analysis.

FR3. If the requirements, in terms of number of Blocker and Major issues for the analysis are not met by the code base, the build triggered by the automation server shall be marked as failed.

FR4. The static code analysis shall only be triggered after a successful build.

FR5. At the event of viewing a pull request in Bitbucket, data from SonarQube analysis from the branches in the context shall be displayed.

3.5 Non-functional Requirements

NFR1. The implemented continuous code inspection environment shall be portable using the virtual machine and require minimal setup settings to transfer from one machine to another in order to demonstrate the environment.

NFR2. The environment must be able to analyze code written in Java.

NFR3. The continuous code inspection environment shall be adapted to require minimum effort during the development process.

NFR4. The feedback from the analysis shall be comprehendible and feasible to implement, given the authors documentation of how to interpret the results.

3.6 Brief summary

This chapter has introduced the main goals of the system, followed by describing the design and gathering process of the system requirements. This chapter has also illustrated the structure and process of the environment to be constructed. As the underlying structure and specifications were presented, use cases were constructed to illustrate the functionality from the user and platform's point of view. Finally, the functional and non-functional requirements can be established. Next, the design and development of the system will be described in detail.

Chapter 4 Design and Development of the System

Within this chapter, the author will describe the design choices and artifacts that are part of the implemented system. In addition to software that are used by the solution.

4.1 General Development Decision and Approaches

To introduce the reader of this document to the conditions related to the technical and experimental circumstances in this project, the next sections will describe the environments, in terms of tools and items, that the project will use.

4.1.1 Technical Condition

These are the static code analysis tools, which will be used to perform the static code analysis:

1. SonarQube (Version 5.4), previously known as Sonar [11] – The continuous code inspection platform which hosts the actual static code analysis tools such as the above mentioned tools.
    a) SonarQube server,
    b) SonarQube Scanner also known as SonarQube Runner,
    c) Plugins:
        i. FindBugs (Version 3.3),
        ii. Git (Version 1.1),
        iii. Java (Version 3.13),
        iv. Java Properties (Version 1.5) and
        v. SVN (Version 1.2)
2. FindBugs [32] – a static code analysis tool that will be implemented in SonarQube by installing the FindBugs plugin.

4.1.2 Experiment Condition

In order for this project to be successful, the implemented environment has to be able to perform the necessary tasks, which are required in the development process. This section contains the key technological components, which are vital to the implementation and the setup for the continuous code inspection environment. Each component will be listed and described briefly.

1. Atlassian Bitbucket server (Version 4.3.2), previously known as Stash [42] – is a web-based hosting service aimed for projects which uses the Git revision control

system.

a) Sonar for Bitbucket server (Version 1.6.0), previously known as Sonar for Stash [43] – is a plugin for the above mentioned web-based hosting service which provides the integration of SonarQube and Bitbucket server.

b) Bitbucket Server Webhook to Jenkins (Version 3.0.1) [44] – is a plugin for Bitbucket server to allow the Bitbucket server to communicate with a Jenkins server.

2. Jenkins automation server (Version 1.651) [45] – is a continuous integration and continuous delivery application which can be used to build and test software projects.

a) SonarQube Plugin (Version 2.3),

b) Git Plugin (Version 2.4.2) and

c) Maven Integration Plugin (Version 2.12.1).

3. Eclipse IDE [46].

4. Oracle VM Virtual Box (Version 5.0.14r105127) – is a virtualization engine for x68 hardware.

a) Ubuntu Desktop (Version 15.10).

5. iipax – Code base provided by Ida Infront where one package will be subject of further investigation, named Objectbase.

4.2 Key Techniques

In order to assemble the environment to perform the static code analysis, several supportive components have to be implemented and configured. The following sections will describe these supportive components.

- **SCM server:** To maintain a historical record of the changes made to the code base and configuration files in the project an SCM server will be constructed.
- **Automation server:** Once continuous code inspection is implemented in a production environment the static code analysis should be run as a part of the software development lifecycle, and should require minimal effort to run once it has been configured properly. This motivation is the origin to use an automation server to call the static analysis from.
- **Continuous code inspection server:** In order to implement the continuous code inspection environment that performs as the company desires, the rules used by the static analysis plugin tools in SonarQube require modifications. The extensiveness of these modifications is difficult to estimate at this time, since they have not been performed at this stage of the project. However, due

to the complexity and the high number of rules, it is likely that this will be a challenging issue. This is due to the need to adapt the entire library of rules to the company's code base in addition to the creation of new rules. E.g., what should be allowed but in a certain way and what should not be allowed in their code base. This is a critical step in this project due to the feedback from the continuous inspection environment. If the feedback is not accurate and recommends modifications that are not feasible for the current code base and the developer is aware of this fact, the environment will lose credibility amongst the developers. Which in turn, may cause the developer to ignore the feedback from the environment once these events have occurred a number of times.

- **Server Hosting:** To enable sufficient portability in the generated environment, the author will use the common approach of having a virtual machine to host the three servers on, which will allow the author to deploy the resulting system with all the configured settings adequately.

4.3 Evaluation Approach

With the purpose of investigating the results from the static code analysis performed on the code base iipax and its packages, the method presented by Plösch et al. [47] will be adapted and applied in this study. Plösch et al. [47] have constructed a method for measuring the quality of static code analysis tools in addition to applying this method to compare a number of these tools. This method may be seen in Figure 4-1. By making some modifications to their approach, the author has constructed his own version of this method.

The initial step of the method is to decide what rules to be applied in the analysis. The rules which will be used in the analysis are the rules from the SonarQube Java [48] and the FindBugs [32] plugins. Since the SonarQube community is currently working with replacing the FindBugs rules using rules from their own Java plugin, some are deprecated at this stage, these have been excluded from the analysis by the author in order to avoid duplicate results.

The next step is to perform the static code analysis on the code base iipax and its packages. In this context, this includes uploading the code base to the Bitbucket server, building the project in Jenkins automation server, which sends the result of the SonarQube scan in a report to the SonarQube server.

By extracting the analysis report and performing an automatic rating on the severity of each rule violation, the next step can be executed. The issues are also presented in the web interface of the SonarQube server. The severity levels are divided into five categories [11], where Blocker is the most severe type of issue:

- Blocker – bug likely to affect the performance of the application.
- Critical – bug that is not as likely to occur which will influence the behavior of the application or an issue, which characterizes a security weakness.
- Major – quality weakness that may have significant impact on the developer productivity.
- Minor – quality weakness that may have a slight impact on the developer productivity.
- Info – a finding that is not a bug nor a quality flaw, simply useful information.

Once the issues have been presented in the web interface, the manual classification may be initiated. This phase consists of determining whether the found issues should be resolved or not by marking them as true or false positive.

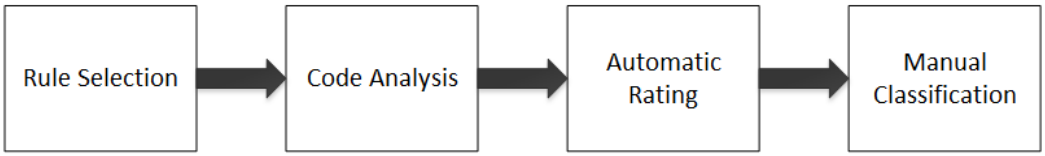

Figure 4-1: Flow diagram illustrating the evaluation method.

However, as stated in the requirements, the issues will be classified using another scale in this work – ranked as Blocker or Major.

- Blocker – critical issue or bug that needs to be fixed before, will fail the build.
- Major – potential issue or anomaly that should be dealt with. Major-ranked issues will not block the build.

If the issue is considered not to be a defect, it will be deactivated.

4.4 Brief summary

This chapter has specified the design and development choices that have been made during the creation of this system. Each of the components and plugins are briefly introduced in order to allow the recreation of this project for interested peers. The evaluation approach is also defined in this chapter to describe the process that will be used to determine the performance of this system.

Chapter 5 Case Study

This chapter is designated for the case study, which was performed at Ida Infront AB in Linköping, Sweden. It will present the methodology of the case study in addition to the results that were found from the data collection techniques. The case study originates from RQ3 and involves investigating how a continuous inspection environment should be integrated into a software development process. What aspects of the introduction which are especially important to keep in mind, and what expectations the developers may have for this new entity in the development process.

As the given code base was huge, in order to be able to go into further detail, the author was required to decrease the scope of the analyzed code base. This was performed by, in collaboration with the author's supervisor and manager, selecting a package that would be interesting to investigate further. The package that was chosen was Objectbase.

5.1 Objectbase

This package consists of 89,756 lines of code and contains code that has been in development since the year 2004, implying that this package may contain legacy code. As most of the other content in iipax, it is mainly written in Java. The functionality implemented in Objectbase is very similar to an object-relational mapping, i.e. the technique to convert data between incompatible type systems in object-oriented programming languages by creating a virtual object database [49]. As compared with alternative popular database products, e.g. SQL DBMS, which can only store and manipulate scalar values organized in tables [49].

5.2 Data Collection Techniques

The initial step of collecting material for the case study was to have discussions, to investigate if and how the static analysis processes are performed. This included having conversations with the author's supervisor and other members of the team to gain a sufficient understanding of the processes in this team.

To be confident that the author had interpreted and translated each interviewee correctly, since each interview was performed in Swedish, the collected material from each interview was validated. The validation was executed by summarizing the

contents of the interview and sending this material to each interviewee that was questioned, to give feedback and comments whether their statements were correctly interpreted.

5.2.1 Interviews

Once the author had gained an initial understanding of the level of knowledge and to what extent static code analysis is used today – he consulted his supervisor and manager to discuss what employees would be suitable to interview. The criterias' for the interviewees were flexible, stating that they should have been involved in implementing the code base that the author would analyze – qualifying a number of developers to be interviewed at the internship company. Once three developers had been recommended for interviews, they were contacted to ask if they would be interested to participate in this study. Fortunately, all recommended interviewees were able to attain the interview and two hours were dedicated for each interview. The interviewees were all male, system developers at the company who were working in the same team.

The interviews contained a simplistic type of manual code inspection, which was a difficult procedure to estimate, since it was dependent on the individual and the case, resulting in the creation of a large number of code samples that could be used during the interviews. 32 code samples were constructed that could be used as interview material, giving the opportunity to perform as many cases as possible during two hours, although all cases that were discussed and were used to draw conclusions from, had to be introduced to all three interviewees.

The interview plan was to present each case to the interviewee and ask him to perform a code inspection that is normally performed once another developer has committed and created a pull request to contribute to the code base. This way the author would be able to compare what is found during a manual code inspection and what a tool based code inspection could find.

The tools available to the interviewees during their code inspection were very simplistic – merely a piece of paper with the code including file name and file path. This humble setup was intended to allow the author to observe the interviewee during the inspection process thoroughly, since the interviewee would be very exposed without the tools available in an integrated development environment. Once the interviewee had been given each case, the author waited and noted the reactions and

stated thoughts by the interviewee, since he had been asked to describe his thoughts during the interview to ease for the authors note taking.

When the interviewee was satisfied with the inspection, he could state the anomalies he had found during the inspection, which was noted by the author. Then, the author presented what anomalies were found by the tool SonarQube during the code analysis that was made before the interviews. Next, the interviewee was asked what he thought of these anomalies found by the tool. Whether he agreed, thus making the anomaly a true positive, or disagreed, making the anomaly a false positive. If the interviewee did not comprehend the issue, by asking about the problem, the author described the problem at hand using the rule description provided by the SonarQube interface, in order to create an as realistic setting as possible. Regardless what the interviewee thought of the anomaly, they were also questioned about the severity of the rule, i.e. whether the anomaly should be a

i. Blocker – very severe issue and will cause the build to fail,
ii. Major – indication of code that should be changed but is not critical or
iii. Nothing – a pattern that unnecessary or will not improve the quality of the code.

In addition to this severity rating, the interviewee was also asked to speculate whether the rule was applicable in other contexts, in order to gather as much information as possible about each possible issue and related rule. During each interview, there were also final questions about the interviewee's code inspection approach. The final questions involved the interviewee describing how he studied the code and what properties were investigated of each case. These questions were expressed using semi-structured questions, as defined by Merton et al. [50].

5.3 Cases

To introduce each case that was treated during all of the interviews, this section will describe the topic of each case and issue to give the reader a comprehension of what was this case study included.

As described in previous sections, the author initially constructed 32 interview cases since the time that it takes developers to execute a code inspection was, according to the author, difficult to estimate. Thus, the author constructed a large number using the motivation that it is better to prepare too many cases compared to constructing a too small number. The samples were composed using the mentality that high severity

in combination with high frequency of occurrence were the most important cases to investigate whether they were accurate or not. The resulting cases that will be presented in this section are the same cases that have been the source to the conclusions, which will be mentioned later in this chapter. Moreover, the rules that were affected by this case study will be described.

5.3.1 Rules

The relationship between each case and issue in this case study may be observed in Table 5-1, in addition to the code snippet that SonarQube detects as the issue. The intention of inserting these rather small code snippets is to introduce the context of the found issue. By combining the information in Table 5-1 with Table 5-2 give detailed data of what rule is related to what issue may be obtained. Each of the rules originates from the rule set of SonarQube and of the ones that were originally configured and kept during the rule investigation performed by the author in collaboration with his supervisor described in Section 1.6.3.

The first rule R1, stated in Table 5-2, endorses the preservation of the original caught exception by logging the original exceptions message and stack trace, alternatively passing it forward [48]. This rule was highest representative among the issues and rules since it was also the most frequent of the findings in the analysis of the code base from start, thus making it captivating to investigate whether it was an applicable rule or not.

The following rule, R2, checks whether any of the deprecated classes of the Java API are being used, such as **`Vector`**, **`HashTable`**, **`Stack`** or **`StringBuffer`**. The recommended alternative is to use **`ArrayList`**, **`Deque`**, **`HashMap`** or **`StringBuilder`**, respectively. These classes were made synchronized in order to provide thread-safety, however, synchronization has a significant negative impact on performance, even when using these classes from a single thread [48]. This rule was interesting to include for two major reasons, it was the second most frequent violated rule among the rules and the fact that Objectbase contains a quite large amount of legacy code, making this type of property interesting to investigate since this rule suggest replacing old Java API classes with different, newer classes.

The next rule R3 is rather trivial, it states how uncommented code should not be present in the committed code base.

Moreover, rule R4 is violated when conditional statements are found to be unconditionally true or false, this would result in the code being always or never executed, either way is most probably not what the developer of this code intended [48].

Rule R5 is intended to track either forgotten or overlooked TODO-tags in the code. While this is not an error per se, it is not the intended use of TODO-tags [48].

Additionally, R6 tracks the use of **`Throwable.printStackTrace(…)`** and recommends the replacement of Loggers, due to two major advantages, Loggers enables the users to easy retrieve the logs, and the format of the log messages remain uniform [48].

Another essential rule that is included is R7, declaring that a reference to null should never be dereferenced or accessed, since this will invoke a **`NullPointerException`** [48]. The consequences of such an exception would, in the best case, result in abrupt program termination, and in the worst case, could result in debugging information being exposed [48].

Next, R8 defines the prohibition of returning, breaking etc. from a **`finally`**-block since it would suppress the propagation of any unhandled **`Throwable`** that was thrown in the preceding **`try`**- or **`catch`**-block [48].

Additionally, R9 outlines the exploration of **`if/else if`**-statements that have the same conditions that could lead to dead code [48]. This issue is likely to occur when copy/pasting code and might, in the worst case scenario, lead to unexpected behavior in the program, while in the best case it would simply induces dead code [48]. Equally important is R10 that detects what is referred to as *dead stores*, which may be alternatively described as useless assignments i.e. assign a value to a variable followed by an additional assignment, resulting in the first value never to be read [48].

The final rule, R11, defines the maximum number of cyclomatic complexity for each method using the motivation that complex code may perform poorly and is far more difficult to understand and maintain [48]. The default value for this rule is the cyclomatic complexity of 10, which has been applied in this case study.

Table 5-1: Table containing the cases with code that SonarQube found to be issues in a rather narrow scope.

Case	Issue	Code
1	1	`catch (ConstraintViolationRuntimeException e) {…}`
	2	`catch (ObjectbaseOperationException f) {…}`
2	3	`private void toStringBuffer(StringBuffer sb){…}`
	4	`catch (IllegalStateException e) { sb.append("<not set>"); }`
3	5	`//sb.append(" ( objectid BIGINT)") PRIMARY KEY )");`
4	6	`if (tx != null){…}`
5	7	`// TODO Auto-generated catch block`
5	8	`e1.printStackTrace();`
6	9	`plugin.setPartition(partition);`
7	10	`throw new CommandFailureException("Unable to close and unlock!");`
8	11	`else if (tableInfo.isWideTable())`
9	12	`String fileTable = null;`
10	13	`public void onInstallation(){…}`

Table 5-2: Table briefly stating the relationship between each issue and each rule.

Issue(s)	Rule#	Rule
1,2,4	R1	Exception handlers should preserve the original exception.
3	R2	Synchronized classes Vector, Hashtable, Stack and StringBuffer should not be used.
5	R3	Sections of code should not be "commented out".
6	R4	Conditions should not unconditionally evaluate to "TRUE" or to "FALSE".
7	R5	"TODO" tags should be handled.
8	R6	Throwable.printStackTrace(...) should not be called.
9	R7	Null pointers should not be dereferenced.
10	R8	"return" statements should not occur in "finally" blocks.
11	R9	Related "if/else if" statements should not have the same condition.
12	R10	Dead stores should be removed.
13	R11	Methods should not be too complex.

5.4 Results

Three interviews were held and the resulting number of cases that were dealt with was 10, containing 13 issues and involving 11 code rules. The personal development experience among the interviewees spanned between 20-30 years. The professional time span that the interviewees had been involved in development in the industry spanned between 11-27 years. None of the interviewees had used static code analysis tools frequently, other than minor uses within their team. All of the interviewees had a positive attitude to the concept of using static code analysis tools as a supportive tool in code inspection in addition to assist the developer to detect defects in the code. Furthermore, the interviewees were asked to determine the classification and ranking of the issues detected by SonarQube.

5.4.1 Issue Determination

Once the interviewees were satisfied reviewing the given case, they were given the opportunity to present their findings. Once they had presented their findings, the author stated what SonarQube had found to be defects in that specific case and giving the interviewee the opportunity to classify and rank the stated defect. If the interviewee had found the same defect as SonarQube, the finding value was set to Found (F), otherwise Not Found (NF).

- **Classification:** Each interviewee was given the choice whether to classify the detected alert as either a TP or FP, resulting in evident results stating whether they agree to the issue at hand. The results from the interviews may be observed in Table 5-3 and as the reader may note, interviewee A and interviewee B consider all found issues to be TPs. Compared to interviewee C that considers all issues except 1, 4 and 10 to be TPs.

Table 5-3: Classification table.

Issue	Interviewee A	Interviewee B	Interviewee C
1	TP	TP	**FP**
2	TP	TP	TP
3	TP	TP	TP
4	TP	TP	**FP**
5	TP	TP	TP
6	TP	TP	TP
7	TP	TP	TP
8	TP	TP	TP
9	TP	TP	TP
10	TP	TP	**FP**
11	TP	TP	TP
12	TP	TP	TP
13	TP	TP	TP

- **Ranking:** Each interviewee was also asked to rank the rule, based on the issue and the properties of the rule to set a recommendation of how severe these type of issues are. The interviewee were able to choose between Blocker (B), Major (W) and to deactivate the rule (X) where the letter in parenthesis presents the letter that represents the rank in Table 5-4.

Table 5-4: Ranking table.

Issue	Interviewee A	Interviewee B	Interviewee C
1	W	W	W
2	W	W	W
3	W	W	B
4	W	W	X
5	W	W	W
6	W	W	W
7	B	B	W
8	B	B	B
9	B	B	B
10	W	B	W
11	B	B	B
12	W	W	B
13	W	W	W

By studying Table 5-4, there are several remarks to note:

- The interviewees agree on eight of the 13 issue what rank the rule should have.
- Only one occurrence of the rules from all three interviewees that was ranked as X, i.e. to deactivate the rule.

What should be clarified is that the occurrence of when the interviewee has stated that he considers the issue to be a FP in addition to claiming that a rule is either a Blocker or Major; this combination may at first glance appear strange. However, what makes this combination possible is the fact that the interviewee may consider the specific occurrence of the rule to be a FP while the interviewee considers the rule to be of valid content and that it should be applied.

Another essential aspect of these results are whether the issues that were detected by SonarQube were also discovered by the interviewees.

Finding: As SonarQube would be used as a supportive tool to find potential defects in the code, SonarQube should detect patterns that may not be detected by developers. As may be seen in Table 5-5, four of the 13 issues were found by all interviewees, whereas two of the issues were not detected by any of the interviewees.

Table 5-5: Findings table.

Issue	Interviewee A	Interviewee B	Interviewee C
1	NF	F	NF
2	NF	F	F
3	F	NF	NF
4	F	F	NF
5	NF	F	NF
6	F	NF	NF
7	F	F	F
8	F	F	F
9	F	F	F
10	NF	NF	NF
11	F	F	F
12	F	F	NF
13	NF	NF	NF

Next, results from the final questions with the interviewees will be presented for each interviewee.

5.4.2 Final Questions

Interviewee A considers the idea of static code analysis tools to be useful, while the difficulties to define and rank the issues in categories is obvious. The interviewee has merely used static code analysis within the team. He also considers that the warnings detected by static code analysis tools have to be able to be switch off by using annotations, since there will always be cases where a certain rule is applicable to 90% to all cases but has to be able to be suppressed if required. The fact that these warnings have to be handled in a certain way in order to keep the development functional is concerning the interviewee. The interviewee thinks that it would be very useful if the static code analysis tools would assist the developers in finding problems in the functional part of the code to verify that the code is working as intended. He also states how interesting it would be if the code analysis tool could assist when taking design and architecture decisions as well. After this interviewee had performed the interview, he remained to have a positive attitude for static code analysis tools since he realized the potential in these tools.

Interviewee B has used SonarQube for a short period of time, he experienced that there were a lot of noise when using it to analyze the source code. During his experimentation with SonarQube, he found that it was difficult to integrate with the build system. He has also used FindBugs to a small extent. The interviewee thinks that the feature in SonarQube that provides the historical aspect of the code analysis is interesting, since it provides a useful view to monitor the code modifications over time to prevent continuously making the code worse. Interviewee B also considers that the focus of the analysis should be on the new code to be added to the code base, since there are simply not any resources to spend on the old code that already does as it was intended. The interviewee used the parable of a black box that delivers the required output from the required input to represent the legacy code. He also considers the use of static code analysis to be useful when applied in the development process.

Interviewee C has also used static code analysis tools in his team but not in any large extent and he is positive towards static code analysis. The interviewee also states how several cases that have been discussed during the interview, had not been considered detectable by static code analysis tools and how he has learned that several aspects that he had not thought could be inspected, is actually detectable.

Chapter 6 Resulting System and Evaluation

Once the case study has been performed in addition to the implementation being completed, the author will present and describe the results of the project. The resulting system consists of Bitbucket, Jenkins and the most significant entity, SonarQube. The system fulfills the specified requirements stated in Chapter 3 and features functionalities such as configured quality gates, leaks, build breaking, pull request view and suppression of false positives. The following sections will describe these functionalities as well as the results from the research by the author that define approaches to configure and design such a system. In addition to these features, the resulting system is configured according to the results produced from the case study.

6.1 Rules

In order to construct the rules in SonarQube to fit the setting of the internship company's setting, three phases were conducted to perform the configuration: an initial investigation, investigation in collaboration with the supervisor and alert oracle configuration. The results from the initial investigation will not be presented since it was intended to educate the author of what rules existed and the capabilities of the environment. The result of the rule configuration is stored in a Quality profile named iipax-product Profile, where a Quality profile is a set of rules that may be applied to several projects in the same SonarQube instance. Currently, there are no other sources of quality profiles other than profiles provided by SonarQube [51].

Furthermore, the results from the subsequent investigations will be presented in the following sections. The results that will be described will be referred to in the succeeding sections are based on the same data used in the case study in Chapter 5, i.e. the package Objectbase.

6.1.1 Supervised Configuration

As the default setting of SonarQube applied the alert ranking described in Section 4.3, which differed from the requested ranking. One of the tasks during this phase was to rank or deactivate each rule to the ranking requested by the internship company, i.e. as Blocker or Major in addition to deactivating the rule due to its inapplicability to the code base. Table 6-1 presents the results from the rule investigation, seven rules ranked

as Blockers, 25 rules ranked as Major and 15 deactivated rules. The raw data for each rule, including the rule id to provide traceability to SonarQube's rule database, may be found in Appendix A. The reason for ranking these rules as described was based on their applicability in this context, i.e. for the package Objectbase. Although that package was used as a representative sample the entire code base, named iipax product, it was carefully selected to be able to demonstrate the properties of the whole project. The incentive for ranking these rules as described was to minimize the number of FPs, in addition to adapt the set of rules according to certain properties since Objectbase consist of a vast amount of legacy code. Furthermore, some properties that are recommended by the SonarQube tool are not applicable in the context of this instance of code, since the legacy code contains the use of many old Java classes that have been replaced by new classes with better performance, but not necessarily making the current code faulty, but improvable. Another example is the rule stating how "Values passed to SQL commands should be sanitized", which had the default ranking Critical but was deactivated since, according to the author's supervisor, there were a vast amount of occurrences in the code that were not able to do so. If this rule had been kept, a large number of false positives would have been produced. An additional example to exemplify what type of discussions occurred during this investigation was the rule stating how "Fields in a "Serializable" class should either be transient or serializable". Similar reasoning was used when altering the ranking of this rule from Critical to Deactivated. That it would produce a large number of positives since there are a large number of occurrences in the code base that violate this rule, that is not relevant to check in this partition of the code base, since the effects of this particular issue would not be severe, according to the author's supervisor.

Table 6-1: Results from the supervisor rule investigation.

Default Ranking	Current Ranking	Nr
Info	Blocker	1
Minor	Deactivated	8
Minor	Major	15
Critical	Deactivated	5
Critical	Major	8
Critical	Blocker	4
Major	Deactivated	2
Major	Blocker	2
Blocker	Major	2

The single rule that was modified from Info to Blocker was the rule regarding TODO-tags, named "'TODO" tags should be handled", describing how TODO-tags should not exist in the code since, conventionally, TODO-tags are intended for reminding the developer, who wrote the TODO-tag, to implement further functionality or edit certain properties. However, TODO-tags should not remain in the production version of the code, according to the author's supervisor. This characteristic was deemed especially important since a TODO-tag indicates that the code is not complete, thus altering the ranking from Info to Blocker.

Prior to this investigation the default mode for all rules were set which resulted in a large number of issues being produced, as shown in Table 6-2. Once the rules had been configured in SonarQube, the results appeared in the form of the total number of detected issues being reduced by 834.

Table 6-2: The number of issues prior to the first investigation.

Ranking	#
Blocker	112
Critical	746
Major	3046
Minor	1110
Info	30

Table 6-3: The number of issues past to the first investigation.

Ranking	#
Blocker	605
Major	3605

Next, the results from the case study described in Chapter 5 will be used to perform the ranking of the rules.

6.1.2 Alert Oracle Configuration

The data will be used to modify the ranking of the rules is found in Table 5-4, i.e. the ranking data from the interviews, where the interviewees act as alert oracles. The summarized results may be found in Table 6-4. Since issue 1, 2 and 4 involve the same rule and the result form the interviews are of equal results for all occurrences except one where it is marked as 'X', the author deems this rule to be ranked as Major by adhering to the majority of the interviewees' judgement. In the event of the interviewees stating different rankings on the same issue, the ranking that has a majority has been chosen, as may be seen in the column 'Majority agreed'. Additionally, the column 'Complete agreement' states the number of rules that all three interviewees completely agreed on.

Table 6-4: Summarized results from the case study.

Ranking	Complete agreement	Majority agreed
(B) Blocker	3	1
(W) Major	4	3
(X) Deactivated	0	0

In turn, this data results in the rules being configured to the corresponding ranking that may be found in Table 6-5.

Table 6-5: Ranking for each specific rule.

Rule Name	Ranking
Exception handlers should preserve the original exception.	W
Synchronized classes Vector, Hashtable, Stack and StringBuffer should not be used.	W
Sections of code should not be "commented out".	W
Conditions should not unconditionally evaluate to "TRUE" or to "FALSE".	W
"TODO" tags should be handled.	B
Throwable.printStackTrace(...) should not be called.	B
Null pointers should not be dereferenced.	B
"return" statements should not occur in "finally" blocks.	W
Related "if/else if" statements should not have the same condition.	B
Dead stores should be removed.	W
Methods should not be too complex.	W

As a result, the number of detected issues has changed accordingly to the resulting numbers presented by Table 6-6. The change involves 547 issues to be ranked as Majors instead of Blockers.

Table 6-6: The resulting number of issues of past the alert oracle configuration.

Ranking	#
Blocker	58
Major	4,152

Next, the introduction of quality gates, including their usages.

6.2 Quality Gates

Quality gates are defined by sets of Boolean conditions that are defined by measure thresholds. If these conditions are evaluated to false, the gate will be marked as failed. Examples of these conditions are [11]:

- No new Blocker issues.
- The number of Major issues are less than 100.

Conditions may be based on metrics from categories such as complexity, documentation, duplication, issues, management, size, technical debt and tests. Thus, quality gates are able to set thresholds for almost any property of the code base.

When setting the threshold, it is also possible to mark a warning threshold, which will generate a warning when the value reaches this, in addition to the error threshold that will violate the quality gate if exceeded [11]. It may seem like an appropriate setting to create one quality gate for all projects. Though, since numerous quality gates may be constructed and configured accordingly, the gates should be configured to the appropriate extent of each project [11]. The default quality gate 'SonarQube way' is provided and activated by default.

Though, as of writing this report, the SonarQube report processing on the SonarQube server is not able to process two conditions defined for the same metric in the quality gate. This error is identified as the issue SONAR-7276 in the public repository of SonarQube[3] and its estimated release date is in June 3, 2016.

A highly useful feature in this context is the leak property, i.e. to compare the current value of a property with a previous version of the software.

[3] Accessible at **https://jira.sonarsource.com/browse/SONAR-7276**.

6.3 Leaks

Leaks may be assimilated to the choice of performing a quick fix or find the source of the issue. The SonarQube platform attempts to solve the analogous problems in software development where developers are not aware of what measures are changing as they perform their modifications [11]. By combining this leak with the previously described Quality gates, the possibility to fail your quality gate if your code base had a constant increase in e.g. complexity or lines of code.

The leak period, i.e. the period to be compared with the current status of the code – may be set in various ways [11]:

- Number of days before analysis. Resulting in the selection of the first available snapshot in that time range.
- A specific date. Also resulting in the selection of the first available snapshot in that time range.
- `'previous_analysis'` to compare to previous analysis.
- `'previous_version'` to compare to the previous version in the project history.
- A specific version, e.g. '1.2' or 'BASELINE'.

The concept of leaks may also be applied in other contexts, such as affecting your development process if certain conditions are not passed.

6.4 Breaking the Build

As requested and defined in FR3, that if not the required number of blocker and major issues are met, the build should be marked as failed in Jenkins. This is made possible by the installation and configuration of the Build breaker plugin [52].
Build breaker will mark the build as failed if the project either fails its quality gate or uses a forbidden configuration. The control whether the build will be passed or failed is performed by the plugin by communicating to Jenkins that the SonarQube server is currently analyzing that specific build and will poll again in a fixed period of time (which may be configured). Thus, this plugin does not restrict other builds from being sent to the same SonarQube server, allowing several builds to be processed. The plugin uses the SonarQube web service API to first find the analysis id and depending on the status, either, break the build or wait a pre-set amount of time (assuming the build status is pending or in progress) [52]. This step is repeated until either the build is marked as successful, failed or the threshold of maximum query attempts is exceeded,

resulting in a failed build. In the documentation of SonarQube, these parameters are referred to as

- `sonar.buildbreaker.queryInterval` and
- `sonar.buildbreaker.queryMaxAttempts`.

These may be altered in the Admin-settings of the SonarQube server. By installing this plugin into SonarQube, the build may be failed in the necessary circumstances. It may be useful to increase the query interval or the maximum query attempts when analyzing larger projects. For the entire iipax product code base, the maximum query attempts may be set to 90 to be sufficient. This value was determined by initially using the default value, resulting in not being appropriate since the analysis took too long to execute. Resulting in the experimental increase of maximum query attempts until the build was successful.

Criticism for breaking the build when using SonarQube has been described by the creators of SonarQube on their blog [53] and state how the use of Quality gates (without breaking the build) is recommended. Their arguments for this recommendation is the new layout of SonarQube (since SonarQube 5.2) that separates the analyzer and the database causing the analyzer to scan the code and the server-side analyzing the generated results. This is done separately without any communication between the server and analyzer except the polling to control whether the analysis is completed [53]. What the Build breaker plugin does, is to recreate this communication, resulting in the attempted transformation of an asynchronous continuous integration job into a synchronous job. An additional motivation to apply SonarQube's approach would be to minimize the number of ways the Jenkins server may break the build since there already are a high number of reasons why the Jenkins build may fail [53].

Moreover, the process of keeping track of several branches and being able to access the analysis data in other places than the SonarQube interface will be presented.

6.5 Pull Request View

In order to make the development and integration work with the introduced continuous inspection the plugin Sonar for Bitbucket Server [43] was installed in the Bitbucket server. Sonar for Bitbucket allows the inspection of SonarQube's metrics and hunting services in the Bitbucket server pull request-view in addition to the opportunity to view code violations and duplicate code lines directly in the differential view [43]. The information flow may be visualized as seen in Figure 6-1, where the

Jenkins server still is included in the setup but since it is not relevant in this scenario, it is ignored in the figure and in this contextual explanation.

The pull request view applies the information gathered from SonarQube's quality gates and controls whether the to-be-merged code contains any issues detected by SonarQube, including whether these issues violate the quality gates, such as if there are any blocker issues introduced, which will, using this configuration, fail the build.

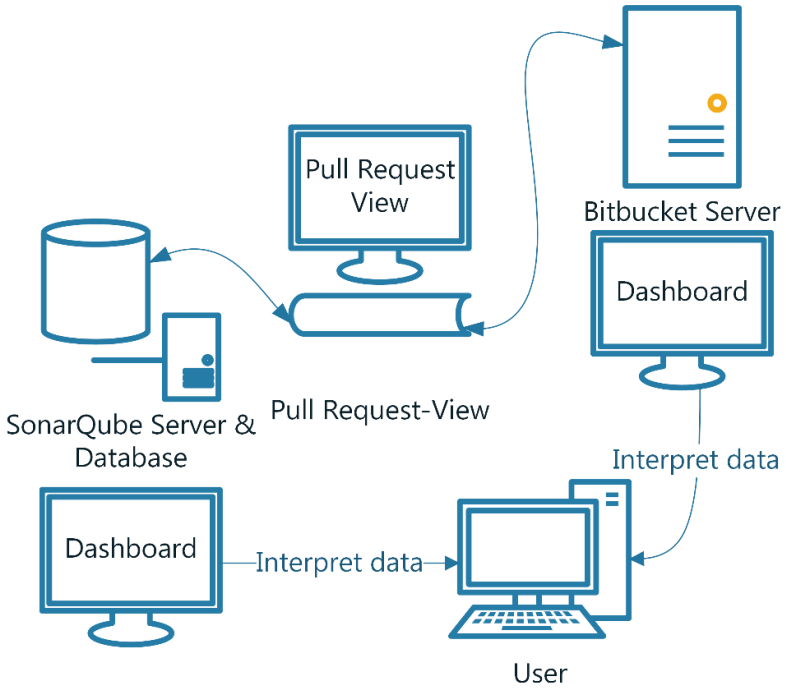

Figure 6-1: Figure representing the pull request view.

For instance, the set quality profile contains the Boolean condition:

Blocker issues	value	is greater than	0 **(X)**

Where **(X)** implies that, the metric measured is an error threshold, resulting in a failed build if this condition is not met, illustrated in Figure 6-2.

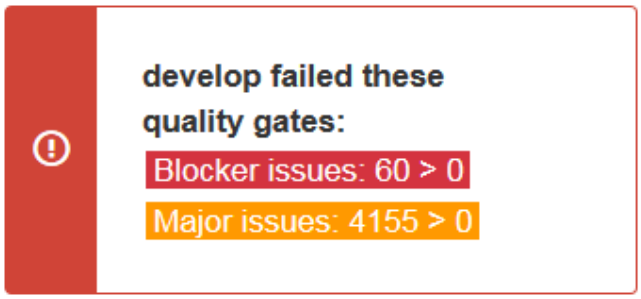

Figure 6-2: Pull request-view of the branch develop that has failed the quality gate. Containing demonstrative data, not related to previous mentioned numbers.

The alternative approach would have been to mark it as **(!)** which would instead have resulted in a warning instead of a failed build, e.g.:

Blocker issues	value	is greater than	0 **(!)**

Which would result in the outcome presented by Figure 6-3.

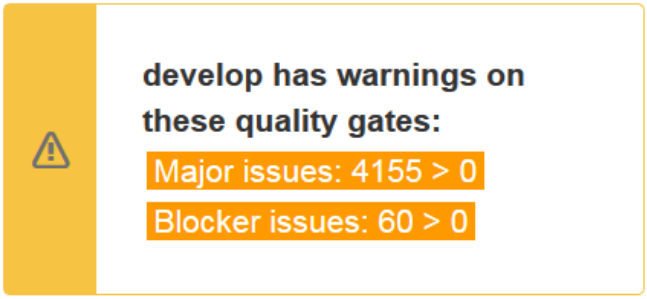

Figure 6-3: Pull request-view of the branch develop that passed the quality gate, with warnings. Containing demonstrative data, not related to previous mentioned numbers.

Equally important to viewing the pull request view is the act of suppressing the alerts that are false, i.e. false positives.

6.6 Suppressing False Positives

The act of suppressing alerts is vital to a tool's appliance since one of the major reasons the use of static code analysis tools are not applied is the high number of false positives [29]. Thus the procedure of marking alerts as false positive has to be available and be within close range during development, otherwise the threat to the tool being ignored is evident, due to the common overload for the developers [29]. In SonarQube, suppressing false positives may be done in various ways; the following are possible commands to use to suppress alerts in code written in Java:

(1) `//NOSONAR` will suppress the warnings on that specific line.

(2) `@SuppressWarning("RULEID")` will suppress one rule or `@SuppressWarning({"RULEID:1"},{"RULEID:2"})` to suppress several rules for a method, class, variable declaration or parameters. `@SuppressWarning("all")` may also be used to suppress all rules the above-mentioned context. This procedure of suppressing false positives is not possible to perform at file level, consequently, if a file is to be ignored entirely, the options of excluding the entire file from analysis should be considered which might both be performed in the SonarQube graphical interface and in the `sonar-project.properties`-file, by using the property `sonar.exclusions=path/to/file/File.java`.

(3) Mark the issue as 'False Positive' in the SonarQube graphical user interface. This approach will not be transferred between branches, thus making option (1) and (2) the more preferable options.

What the author has found during his experimentation and testing is that at least one rule is not possible to suppress. That rule is '*Source files should not have any duplicated blocks*' that detects and notes duplicate blocks of code in the same file or even in different files. Currently, this rule is not possible to suppress, however, it is planned to be fixed in SonarQube version 5.6 that is due to be released in June, 2016 [54].

Next, the historical perspective will be introduced how it may be used to find defects in the code.

6.7 Historical and Trend Information

The ability to track the measurements over time is one of the major benefits of SonarQube since it offers the ability to monitor the trend of the code base [22]. The opportunity to measure the trends of various metrics of your code base should not be overlooked. There are several features available in SonarQube to perform this type of metric collection, firstly, in the project space view where the key metric is plotted in a time line graph to illustrate the rate that the metric resides in. An example of this feature is visible in Figure 6-4 where the duplication and lines of code metric are plotted. The yellow-shaded partition of Figure 6-4 represents the leak-values, i.e. the difference between the version of the code to be compared with (the previous version

is the default choice) and the current version. In this scenario, there have not been any new duplicate code introduced and two new lines of code has been added.

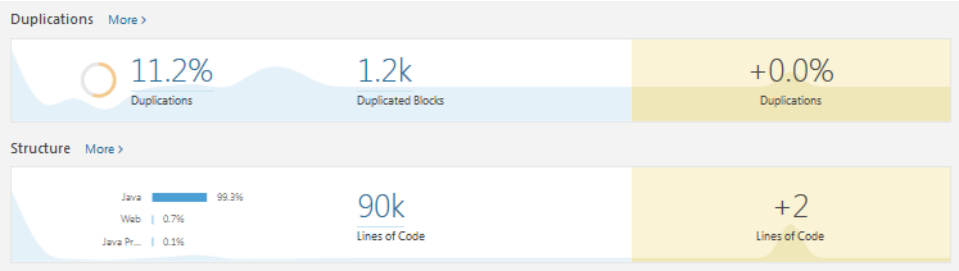

Figure 6-4: Example image of timelines of duplications and lines of code metrics.

Secondly, the time line widget allows for tracking of several metrics combined over time, and by hovering the mouse pointer over the graph, the version number, in this case 1.1, will be presented below the date.

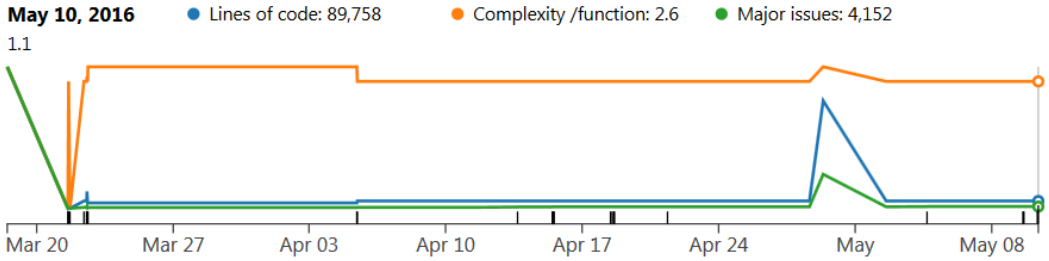

Figure 6-5: Time line graph containing three metrics.

Lastly, to track the individual progression of each metric combined with the comparison of the previous versions the history table feature is available [11]. It also plots the progression of each metric to allow rapid monitoring to detect if the metric has only increased for all previous versions.

	16 May 2009	04 Aug 2011 2.10-SNAPSHOT	15 Sep 2011 2.11-SNAPSHOT	03 Oct 2011 2.12-SNAPSHOT	
Complexity	4,351	11,080	11,598	11,637	
Coverage	62.8%	68.2%	68.8%	68.8%	

Figure 6-6: History table.

The last two of these features are customizable widgets that may be configured to display the appropriate information and scope for each project. These listed features are features that are required to provide the needed traceability to satisfy the goals of advanced code inspection and create the conditions to improve the code continuously [5]. Another essential property in quality measurement systems is the ability to monitor the trend of the analysis, for high-level data of entire projects in addition to detailed

metrics for functions and methods. This feedback, in the code implementation and inspection processes, will provide the opportunity to detect and perform further refinements of the code in order to maintain the code quality.

Next, the author will describe the data flow that occurs during system usage.

6.8 Key System Flow Charts

The implemented system is intended to replicate the development process at the internship company and provide a demonstrative example of how the setup may function. The intention of this example is to provide the maximum support for developers to monitor the code base in as many phases of the development as possible. The data flow diagram, which can be seen in Figure 6-7, represents the environment implemented by the author. The circles represent processes and the boxes represent entities, which in this context are the three server entities: Bitbucket, Jenkins and SonarQube. The initial step of this data flow diagram is that the code is modified, followed by a commit and push to the Bitbucket server. Next, a build is scheduled, Jenkins retrieves the code from the Bitbucket server and builds the project. If the project is successfully built (do note that no analysis has been run on the code yet) Jenkins calls the SonarQube runner that analyzes the code, generates and compresses an analysis report. The report is uploaded to the SonarQube server that extracts and computes the quality gate measures. Depending on the result of the quality gate measures, the build will be failed or successful. This is possible due to the Build breaker plugin in SonarQube that sets Jenkins in a state of polling the SonarQube server for a status regarding the quality gate measurement. When the timer in this plugin expires, or Jenkins receives a response whether the gate is passed or failed, the build will be marked either as failed or successful.

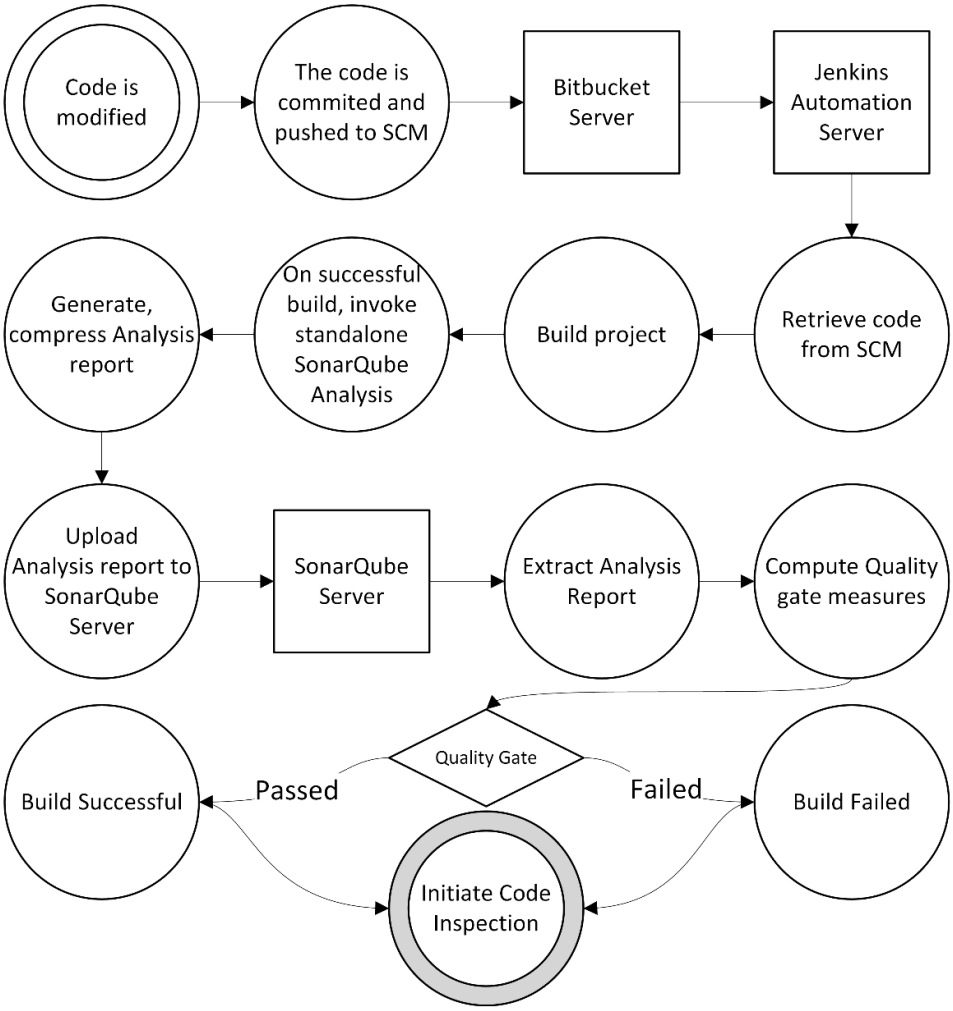

Figure 6-7: Data flow diagram.

Furthermore, to answer the research questions the author has performed an extensive study by reviewing existing research and online content. The results of this investigation will be presented in the following section.

6.9 Analysis Results

This section will describe the content found by the author that will contribute to answering the research questions to investigate how detects may be found by using static code analysis and continuous inspection.

6.9.1 Complexity and Duplication

As described in Section 2.1.1, that introduces the concept of using complexity as a metric to measure the understandability of software. The recommended threshold for the cyclomatic complexity was stated to be 10 [18], as motivated in Section 2.1.1. Equally important is the duplicated code measurement that measures the amount of redundant code, since code that is identical should not be written in more than one location, instead it should be generalized to be reused [55].

During the author's research he also found other approaches to deal with complexity. One additional approach is constructed by Alves et al. [56] that describes and demonstrates a methodology to derive software metric thresholds form a set of systems. This approach may be summarized as to first analyze the statistical distributions of the raw metrics among the different systems and then determine the thresholds based on the variability between the systems, resulting in the opportunity to identify the rarest cases that are of higher risk [56].

In turn, this methodology was applied by Baggen et al. that describe the approach created by the Software Improvement Group(SIG) that is intended to be used to improve the maintainability of software [55]. The defined methodology uses a standardized measurement model that is based on the International Organization for Standardization(ISO)/International Electrotechnical Commission(IEC) 9126 definition of maintainability and source code metrics [55]. The benchmark repository, i.e. the origin of the quality model, contains code from 45 different computer languages with Java, C, COBOL, C#, C++ and ABAP as the most main contributors, in terms of lines of code.

The result of the methodology is a layered quality model with the intention of measuring and rating the technical quality of a software system using the quality characteristics of ISO/IEC 9126 [55]. In this quality model, detailed information about source code measurements. are introduced for code duplications and cyclomatic complexity. The presented results are found in Table 6-7 that contains the threshold for code duplication. The left column defines the rating of the code duplication (the larger number of stars defines a better quality) and the right column defines the maximum number of percentage of code duplication that the code may have to be classified as the corresponding rating.

Table 6-7: Code duplication in the entire system.

Rating	Duplication
★★★★★	3%
★★★★	5%
★★★	10%
★★	20%
★	-

As seen in Table 6-7 that contains the threshold for code duplication where the left column defines the rating of these (the larger number of stars defines a better quality) and the right column defines the maximum number of percentage of code duplication that the code may have to be classified as the corresponding rating. As stated by Juergens et al. [57], strong evidence proves that inconsistent code clones create a major source of faults. This signifies that cloning may be a substantial problem during development and maintenance unless special caution is taken to track and monitor the existing and emerging code clones. One more motivation to adapt the supervision of code clones, is made by Rattan et al. [58] expresses the fact that if clones are detected and removed in earlier phases of software development, maintenance costs will be reduced in the delivered product. Although, properties of certain languages may also prevent the efficiency of adapting code clone classification [58].

Furthermore, Baggen et al. [55] also defined thresholds for cyclomatic complexity in the scope of a code unit that may be defined as a unit that is the smallest module that may be tested individually, e.g. a Java method or a C function. The resulting thresholds are found in Table 6-8, where the left column defines the cyclomatic complexity quantity and the right column defines what risk category is at stake.

Table 6-8: Cyclomatic complexity risk categories for a code unit.

Cyclomatic complexity	Risk category
1-10	Low risk
11-20	Moderate risk
21-50	High risk
>50	Very high risk

An important point made by Baggen et al. [55] is that metric thresholds could in fact be determined by experts and knowledgeable peers, however, the act of calibrating metrics against a large set of real-world systems has some advantages, such as:

(1) The process is more objective.
(2) The process may be executed almost automatically, thus allowing the easy update of the metrics.
(3) Calibration is realistic since it is created to represent the entire spectrum of quality achieved by real-world systems.

Hence, when introducing a metric that will be used to measure quality, it is recommended to adhere to the above-mentioned methodology.

Moreover, as stated by Bouwers et al. [59] the use of software metrics are convenient for developers and project managers. Bouwers et al. also concludes that it is important to attach a meaning to each metric to clearly define the meaning and purpose of having that metric, combined with averting from making the metric the goal. The act of utilizing several metrics to track various aspects and dimension of your goal is highly recommended, however, Bouwers et al. states the importance of not having too many metrics to track simultaneously. The reason for this is to avoid demotivating the team.

6.9.2 Design and Architecture

As research question RQ1 states, one of the aims of this study was to investigate how SonarQube can provide feedback on design and architecture of a code base. One opportunity to receive feedback in this context is the rule 'Cycles between packages should be removed' that detects cyclic dependencies among packages that affects the maintainability of the modules that are related to these packages [48]. This rule logs a violation on each source file that has an outgoing dependency that requires to be eliminated to break the cycle.

Moreover, during the author's investigation to attempt to use all possible measures to discover all relevant functionalities in SonarQube he received a response on a Stack Overflow post from G. Ann Campbell, one of the writers of the book SonarQube in Action [22]. Campbell stated how the only rule she could think of, that were applicable to the author's desires, were the rule mentioned in the previous paragraph [60]. Although, the author also found a rule in SonarQube, named 'Inheritance tree of classes should not be too deep', that detects the inheritance level

of classes and if the depth of inheritance exceeds the pre-set maximum parameter the rule will mark the inheritance as a violation [48]. The default value of this parameter is five levels of inheritance. According to the rule description, a too deep inheritance tree may cause very complex and unmaintainable source code, while inheritance should not be avoided, it should not be overused either since it can be replaced by composition in certain contexts [48].

Next, the author presents the results from the analysis in regards to continuous inspection.

6.9.3 Continuous Inspection

As described in Section 2.3, there is currently no silver bullet to mitigate the arising difficulty of managing increasingly difficult projects. However, continuous inspection is close [5]. A reason why, is the opportunity to track and coordinate software metrics for various context levels and certain thresholds while another is to provide a centralized tool to coordinate the tracing and definition of code rules. Moreover, instead of using merely snapshots to track the quality of the code base, the continuous inspection approach provides the use of trend and history information that allows the measurement of certain properties over time in addition to the comparison between versions.

6.10 System Evaluation

This section will, using the results from the case study, describe the results of the evaluation of SonarQube performed by the author. The evaluation was performed using a subset of the internship company's code base. For detailed information of the data collection of the case study, the author refers to Chapter 5.

6.10.1 Alert Classification

As stated in Section 2.2.3, alerts may be classified as TP, FP, TN and FN to define their correctness in the analysis. By applying the metrics defined by Zimmerman et al. [31] the opportunity to evaluate the produced output from SonarQube. The metrics that will be applied are precision, recall and accuracy, as defined in Section 2.2.3.1. Similar to the study performed by Heckman et al. [13], this results of this study will also focus on what alerts are detected by the tools, instead of the potential issues that are *not* detected.

Using the results from the case study, described in Section 5.4 the metrics may be applied to the collected material. Since the alert oracle in this setting is actually three interviewees, the author has conducted two versions of each metric to demonstrate the variation when combining the results from each interviewee since each interviewee's output may not be equal to the other two interviewees'.

Approach A consists of combining each classified alert into one response, e.g. if two of the interviewees considered issue X to be a TP while the third interviewee considered issue X to be a FP, approach A will consider the alert oracle to have deemed issue X to be a TP. In comparison to approach B that will take all individual response in to account, i.e. given the previous example the author would consider all responses, in this case resulting in two TPs and one FP.

- **Precision:** Given approach A, the precision p_A may be calculated as:

$$p_A = \frac{TP_C}{TP_C + FP_C} = \frac{13}{13 + 0} = 1.00.$$

Resulting in the precision $p_A = 1.00$, indicating that all detected alerts are defects. In contrast to the calculation of p_B:

$$p_B = \frac{TP_C}{TP_C + FP_C} = \frac{36}{39 + 0} = 0.92.$$

p_B is, as expected, lower than p_A. This result points to a far less accurate analysis, even though still rather accurate. Next, the metric recall will be calculated.

- **Recall:** As the case with precision, the desired value for recall is also as close to one as possible, since that would indicate that all found alerts are defects. The calculation of recall follows:

$$r = r_A = r_B = \frac{TP_C}{TP_C + FN_C} = \frac{39}{39 + 0} = 1.00$$

The results indicate that all found anomalies are defects. However, the fact that this study focuses on the detected anomalies, i.e. the anomalies detected by the tool, the metrics' TN_C and FN_C contribution becomes obliterated. Thus, the recall metric should not be solely used to base any conclusion. Subsequently, the metric accuracy will be calculated next.

- **Accuracy:** Similar to precision and recall, accuracy measures the property of how well the model has classified the alerts and accurately measures the number of correct classifications. Accuracy is calculated in the following expression:

$$a_A = \frac{TP_C + FN_C}{TP_C + TN_C + FP_C + FN_C} = \frac{13 + 0}{13 + 0 + 0 + 0} = 1.00.$$

$$a_B = \frac{TP_C + FN_C}{TP_C + TN_C + FP_C + FN_C} = \frac{36 + 0}{36 + 0 + 3 + 0} = 0.92.$$

The accuracy metric a indicates that all classified alerts are anomalies. However, the fact stated in previous section is also highly topical in this context since there are several metrics that are not retrieved in this formula.

6.11 Brief Summary

As this chapter has shown, the resulting system possess many valuable features and services that may be used to monitor and improve a code base. Comparing the resulting system with the stated requirements in Chapter 3, the observant may notice that all requirements are fulfilled. The resulting system is able to determine according to the set conditions whether to pass or fail the build, in addition to viewing feedback from the SonarQube server, in the pull request view. By combining these properties with the ability to monitor the history and trend of the code base, the metric monitoring of the code base is very likely to be able to improve the quality of the code base.

Chapter 7 Discussion

In this chapter, the author will discuss the possible improvements of the chosen method of approach and the resulting system. The discussions will revolve around what phases of the project that should have been performed in another way, to achieve other results, or to reschedule resources according to assignments in this project.

7.1 Relevance of the Resulting System for the Internship Company

As described previously in this paper, current research states the importance of having a maintainable software, to minimize the maintenance and refactoring expenses. To achieve this in practice, a change has to be made in the development process. The extent of this change depends on the goals determined by the actor. In this case, the actor is the internship company and, as their goals of this modification is primarily to detect defects in the code and to have supportive documentation to initiate design and architecture refactoring; the modification would require adding an additional component to their software development. To demonstrate how and what is needed to be changed in the development process, the system that the author has constructed mimics their current development process setup, by having Bitbucket and Jenkins included in the setup with the corresponding settings that is used by the internship company. The implemented system will act as a template for the internship company when they will implement their own version of this continuous inspection platform, SonarQube. Together with this template, the research made by the author will support the internship company to provide guidelines of how it should be used and what properties that are important to apply during the development process to monitor and measure the evolution of the code base. In detail, this report should guide the internship company to shape SonarQube in terms of what risk level to configure the cyclomatic complexity rule to, in addition to configuring the rules of the entire rule database.

Next, the approaches taken including their result will be discussed and evaluated to highlight what could have been improved in this paper.

7.2 Method

As in almost every project, there are aspects of the chosen approach to solve the problem that would have been altered if the author had performed the same project once more. This section is intended to highlight these aspects of the methods applied in this project. First, the author will discuss the method applied implementation phase followed by the method adapted in the implementation phase.

7.2.1 Implementation

The major entities in the implementations: Jenkins, Bitbucket and SonarQube, were not familiar to the author, making the installation and configuration more time consuming than what it should have. During the initial phase of the project, the author also investigated the use of Docker[4] instead of a VirtualBox in order to make the solution more portable and easier to set up. However, as the images for these three components were deemed immature to use in this context, the author decided to implement the use of a VirtualBox instead. The decision could have been settled sooner in order to maximize the time available for the remaining phases of the project. However, making this kind of decision rapidly could result in unwanted consequences, such as the platform being difficult to port or having other kinds of unwanted properties. This is the motivation for taking the time to decide between these two approaches.

During the beginning of the implementation phase, the author strived towards mimicking the actual development environment at the internship company to the highest extent, by using their existing Jenkins and Bitbucket servers and hosting the SonarQube server in a separate location. This approach ended up being unexecutable, which the author did not discover until a large amount of implementation and testing had been completed. The reason was that, as the network communication was configured at the internship company, a server might not initiate communication to a local machine, i.e. the Jenkins server were not able to communicate to the SonarQube server to initiate an analysis. The consequences of this configuration in the network settings at the internship company resulted in the author installing and implementing his own version of Jenkins and Bitbucket server.

Moreover, the implementation involved a high amount of system integration, in terms of enabling the communication between the three servers. The method applied for the system integration is rather straight forward. Since it more or less consisted of

[4]An open platform for distributed applications for developers and sysadmins. **www.docker.com**

installing the servers and attempting to establish the necessary communication, which according to the author's experience may not be executed in many ways. Rather than to perform the necessary configuration in each entity's settings until each path of communication is functional. The configuration also involved a large amount of debugging during the setup, due to communication difficulties between the components.

Regarding the aspect of replicability for the implementation, the process of installing the described environment is rather straight forward once it is up and running, while the configuration is far more advanced. The required configuration functionality is described in Chapter 6, defining what functionality is required to achieve the same results.

7.2.2 Rule Configuration

The key aspect of the rule configuration phase that could have been improved is to increase the initial base of the rules and to what extent they should have been configured. Doing this would have resulted in a larger set of results to present which would have, in turn resulted in in a more credible study. However, by adapting this approach, prolonged interviewing sessions would have been required to question the interviewees on equivalent material due to the difficulty of estimating the duration of each interviewee's manual code inspection time for each case. Consequences of this approach would have resulted in longer interviews and since it is not certain that the prolonged interviews will result in different results. Chances are that the extended time, including the time to prepare and analyze the results, would not have been worthwhile.

An alternative approach that would have been possible to adapt for the rule configuration would have been for the author to configure the rules using his own knowledge and making assumptions what would have been suitable for the internship company. This would have allowed the configuration of a higher number of rules. Although this approach may sound as a valuable proposition, chances are that the configured rules including their severity may not correlate to the internship company's mindset of what issues are vital and what rules are irrelevant. Considering this possible outcome, the author argues that his executed approach is preferable, since he has performed an investigation of how the rules may be configured and examined.

The rules configured consisted of the FindBugs and SonarQube Java rule bases and as the author found that there are no other sets of rules or quality profiles except

the ones provided by SonarQube [51]. The origin of this subject was a question from the author's manager regarding preset quality profiles that would be recommended for certain types of projects. Although the fundamental concept of having adapted quality profiles for specific types of code bases is good, the actual fulfillment appears troublesome since the rules are required to be adapted to specific properties. Although, SonarSource is currently working with finishing a quality profile refined with certain security rules [51].

By providing each rule with a unique ID in addition to its original rule ID given in the SonarQube database the traceability in this study is provided, in turn this contributes to the reliability.

Following this subsection, the author will discuss the method used during the interviews.

7.2.3 Interviews

The most controversial element of the method used in the interviews performed in this study is the fact that the interviewees, who all were developers, were handed pieces of papers with the code on to review and perform code inspection on. This setting is very likely to have influenced the results of each interviewee's detected anomalies, since the setting that the interviewees were used to, includes the Eclipse IDE and internet access giving them the possibility to rapidly search for properties that they found strange or libraries that they had not been exposed to previously. And by taking away these valuable tools, which are used on daily basis by the developers, they are left with nothing but their current knowledge of the code.

An alternative approach to this method would have been to provide the interviewees with their everyday setup to perform the code inspection in. This would have resulted in a far more natural and realistic setting for the interviewee. However, it would also have resulted in a context which may not have been equal for all employees, since some of the interviewees may use special tools or techniques to perform code inspection that are highly preferable, resulting in an advantage to the interviewees using this technique or tool. This approach would also have resulted in the evaluation of the code inspection setup rather than the code inspection performed by the interviewee. Additionally, the time span of the code inspection for each case would have been affected, as the author already had set the two-hour limit for each interviewee, there is a risk by adapting this approach. The risk is that each interviewee

would have used more time since they had the tools as they have in their day-to-day profession. Although, given these tools, the degree of limitation is stricter, resulting in a more limited environment for the interviewee. In turn, this results in a limited degree of inspection that may be executed by the interviewee.

Another essential point is the fact that the interviewees were not entirely honest during the interview. Scenarios that may have occurred is how they have had a poor experience with other static code analysis tools that have resulted in a bad attitude to other types of tools. However, this is a situation the author should consider but have no power to control other than to act neutral when asking questions to influence the interviewee as little as possible.

Moreover, the notion of requesting the interviewees to perform a conventional code inspection in the previously mentioned context may seem rather theoretical since the interviewees are asked to identify any anomalies they detect in each case and imagine that it is the same code inspection process that is performed daily. In comparison to the actual code inspection that is truly accomplished each day, e.g. during each pull request, this code inspection is somewhat unrealistic.

By performing a validation of the results of each interview, as described in Section 5.2, the results from the interviews may be considered approved by each interviewee. Next, the discussion about the analysis that has been performed to investigate how the feedback from static code analysis and continuous inspection may be applied to find defects in the code.

7.2.4 Analysis

After examining the core functionality of SonarQube and its plugins, the author had gained an extensive understanding of what aspects of feedback that SonarQube focuses on. Using this understanding, the author investigated what current research has found of the core functionalities of SonarQube, such as code duplication and various metrics. As this approach is based on the functionality of SonarQube, rather than a complete open investigation of how code may be improved, the approach is restricted from start to this functionality. In some sense, this is a drawback since some results are not able to be included. However, as the purpose of this paper is to evaluate the performance and result of SonarQube's functionalities and abilities, the restrictions are tolerated.

7.2.5 References

During the implementation of this system, as the author experienced issues, he posted questions on the internet forum Stack Overflow to find answers to problems that he encountered in addition to explore whether certain features that were available or whether they would be available in the future. As the author posted questions, he received responses from the crew at SonarSource, the developers of SonarQube and the languages product owner at SonarSource, who is also the author of SonarQube in Action [22]. This feedback from the Stack Overflow community has been very valuable during the development and debugging.

Furthermore, the majority of the references in this paper are scientific articles that are considered trustworthy and credible. In addition to these articles there are also web sources revolving around SonarQube and its rules. These sources are not necessarily as credible and as qualitative. These sources are required to describe the rules that are used in the development of the implemented system and during the rule configuration.

Following this section, the discussion of the results of this study.

7.3 Results

With the intention of highlighting specific results that are prominent, the results will be discussed in the order of appearance that correlates to the previous section.

7.3.1 Implementation

As stated in the introduction of this chapter, the implemented environment represents the setting the internship company would use, to introduce continuous code inspection in their development process.

The environment implements the key functionality of observing the history and trending of code metrics, in order to detect and mitigate the increasingly complex code base, before it has resulted in a project that is arduous to maintain. The performance of the implementation is not optimal in terms of analysis speed, but this is not the main focus of this work. Since the essential aim is to investigate how to gain the largest amount of feedback from this type of continuous inspection environment.

Next, the results from the rule configuration will be discussed.

7.3.2 Rule Configuration

As stated in Section 6.1.2, there are 11 rules that have had their ranking confirmed during this process and the resulting number of rules that have been affected of the alert oracle configuration may appear rather low. However, as stated in Section 7.3.2, any alternative approach would not, according to the author, have had a significant result. Furthermore, considering this result the rules have been successfully verified and construct a valuable basic framework to start from, when creating the production version of the continuous code inspection version.

Judging by the results mentioned in Section 6.10.1, from the precision metric both variants p_A and p_B, indicate that the performance of the analysis is very exact. By looking at the values of recall and accuracy as well, they may at the first glance look highly promising, however, as the formulas are investigated further, the terms FN_C and TN_C are zero in all occurrences. This indicates that the results of these formulas should be handled with caution. Nevertheless, the result of precision remains persistent.

Succeeding the results from the rule configuration are the results from the interviews.

7.3.3 Interviews

The results from the interviews in terms of the classification was surprisingly positive results since all except three issues were deemed as TP. This signifies that the interviewing process was well carried out, since it was the same interviewee that classified the FPs. This implies that he had another opinion than the other two interviewees what was a TP or FP, at least on those three issues.

As the results indicate from the ranking made during the interviews, the interviewees agree on eight of thirteen issues what rank the rule for that specific issue should have. While there was only one occurrence where the interviewee judged the rule to be deactivated. These results indicate how the interviewees agree on the majority of the issues at hand. This bodes well since developers in some sense have to agree on the same rules and ranking in order for this type of continuous inspection to work smoothly. Additionally, the results also represent how the interviewees agree to the vast majority that these anomalies are defects.

Finally, the results in regards to findings, whether each interviewee detected the same issues that SonarQube detected. As four of the thirteen issues were found by all interviewees and only two were not found by any interviewee, this also bodes well for

the implementation of SonarQube. The reason why is that SonarQube were able to detect anomalies that the interviewees did not. This indicates that the tool analyzes the code in a different way. Although, this is not a revolutionary statement. Nevertheless, it is extremely valuable results to indicate that the tools are useful as a support to assist the developer.

Following the discussion of the interview results, is the discussion of the analysis results.

7.3.4 Analysis

The results of the analysis partition of this project are focused into three points. Firstly, complexity and duplication. The found results are a part of a quality model that is intended to measure the technical quality of software systems [55]. Even though this data is standardized and general, it should be valid starting points for settings thresholds in SonarQube to monitor the code duplications and complexity during development. An alternative would be to use their method, defined by Alves et al. [56], to construct these metrics, i.e. to collect metric data from several of the internship company's projects to calculate the metrics for the specific context. This approach may seem cumbersome but may be worthwhile to consider if the metrics from the quality model, mentioned in Section 6.9.1, are not applicable.

Regarding the design and architectural aspect, SonarQube seems to fall short in this topic, as described in Section 6.9.2, where there are two rules to focus on this. However, during the author's research he has found that SonarQube has implemented features in order to attempt to support developers in analyzing the design and architecture but they have resulted in being difficult to grasp and generating too many FPs [60]. This indicates the difficulty of implementing functionality, that are designed to analyze the design and architectural aspects, that works well.

Moreover, the following section, including its subsections will discuss this work in a wider context involving ethical and sustainable aspects.

7.4 The Work in a Wider Context

To describe the additional aspects of this work in terms of humane and environmental point of views, the following sections will present the plausible effects that this work could have to each area.

7.4.1 Ethical Aspects

The continuous inspection environment implemented in this project interacts with developers during and after development. This will result in the environment being continuously configured and adapted by the developers. If the environment is not configured properly and critical bugs are not caught, as the system previously was configured to do, could result in the developers being blamed for the error.

A similar scenario might occur in the context where the environment fails to catch the critical defects that are either similar to other defects that it detects, or defects that are unheard of. The consequences of this event are similar to the previously described scenario, that the developers are blamed for the unrecognized defect.

An additional situation that may occur once this environment has been active in the development environment for a period is that the developers have grown accustomed to the continuous inspection and start to rely entirely on the continuous code inspection environment instead of combining manual and automatic code review. Consecutively, this could result in the scenario where developers start to depend on the environment instead of using it as a tool, which is not the intention.

Now that the ethical aspects have been discussed, the sustainability aspects of this work will be discussed.

7.4.2 Sustainability Aspects

In regards to the sustainability aspects of this work, the goal is to improve the produced software quality of substantially the internship company in addition to investigating this topic from a more general point of view. Given even the slightest frequency improvement in finding defects in the code, would in turn result in a higher software quality. This would allow the developers to spend less time on finding defects to instead implement new features and improve the system's functionality, resulting in enhanced systems and decreases in expenses during development and debugging.

Conclusions

The main purpose of this project was to investigate how the concept of static code analysis may be used as a supportive tool during code reviews in the internship company. The steps taken to perform this investigation were to carry out an evaluation of the continuous inspection platform SonarQube that uses static code analysis to perform its inspection. In addition to the functionality provided by SonarQube, a plugin that is based on PMD and Checkstyle have been included in the implemented environment, as well as the FindBugs plugin that is also based on FindBugs structure and properties. Using this implemented environment, an evaluation was conducted to examine the performance of the environment. The evaluation consisted of performing analysis on a package of the code base provided by the internship company, named Objectbase, to inspect what type of alerts that were detected. These results were then used as material for interviews where the interviewees acted as alert oracles, to empirically determine the severity and legitimate of the found issues and rules. As the determination had been executed and processed the author could compute metrics to be able to assess the performance of the tool.

To regulate the aim and direction of this study, three research questions were composed. RQ1 focuses on the potential improvements that may be detected and issued by the introduction of static code analysis, while RQ2 and RQ3 aim to analyze how static code analysis and continuous code inspection may be applied to find defects to improve the code quality.

The result of the implementation is a continuous inspection environment that mimics the development environment at the internship company by using the identical components by adapting the implementation according to the functional and non-functional requirements.

By using the complexity metrics and thresholds, along with the rules mentioned in Section 6.9.2, implemented in SonarQube the author is confident that the executed static code analysis will be able to assist the developers to detect and improve the design and architecture of the code. Having that said, RQ1 has been answered. Furthermore, the answer to RQ2 is not as straight forward, since the question states the strive to find defects in the code and a static code analysis tool may end up with a large quantity of defects, in the tool's opinion. While the actual number of defects is

significantly smaller. The source of the variation of these values often resides in the fact that the tool generates FPs, thus the answer to this question is related how to differentiate the TPs from the FPs. The solution is to apply an easy-to-adapt rule database to allow users to configure and track the change of the rules, compared to e.g. running on the default set of rules locally. Although, there will always be cases where certain rules do not apply to all detected cases and requires the functionality of suppressing the FPs and as described in Section 6.6, SonarQube provides three ways of suppressing FPs detected. While this way of managing the FPs is not optimal and imperfect, the author considers this functionality satisfactory to achieve this goal since it allows the suppression of certain rules, in specific contexts and the suppression of all rules on specific lines.

Equally important, RQ3 states the query of how to use continuous inspection to find defects in the code. As found by the author in this study, SonarQube provides the ability to combine the execution of static code analysis tools with a centralized rule database to track the rules for entire teams and projects. SonarQube also provides several features to support the introduction of continuous inspection in an agile development process. These include viewing the SonarQube results in the pull request view, blocking the build if the quality gate is failed and allowing the tracking of trends, history of metrics and issues. Moreover, the precision metric indicates how well the performance of SonarQube is, causing the credibility of static code analysis and continuous inspection to increase. Since the data set used to perform the rule configuration represents the most frequent and highest severity of the detected issues, the results of the static analysis should be representable. Combining this with the positive attitude of the interviewees in addition to SonarQube finding defects that were not discovered by the interviewees, the trustworthiness is stalwart.

Therefore, the author is confident in stating that, the introduction of a continuous code inspection environment is profitable, even though, a reasonable amount of configuration is required to be able to gain this turnover.

Future Work

As the aim of this work was to investigate whether the use of a continuous code inspection environment would be able to detect defects and track the software quality, an apparent approach of future work would be to implement this environment, in terms of a production implementation for the internship company. This environment would adapt the stated metric thresholds for the introduced software quality metrics. Combined with performing a thorough rule configuration for their entire iipax product in addition to tuning the rule framework according to certain aspects that are appropriate to the specific project settings.

An additional future work proposition, is the extension of the SonarQube project in terms of developing a plugin to perform measurements that are currently not available using SonarQube but still are valuable metrics. A suggestive proposition for this line of future work, would include the investigation of what type of architectural metrics are the most appropriate for such an implementation followed by the implementation of the plugin to collect and display the metrics.

References

[1] R. Plösch, H. Gruber, C. Korner, and M. Saft, "A Method for Continuous Code Quality Management Using Static Analysis," *2010 Seventh Int. Conf. Qual. Inf. Commun. Technol.*, pp. 370–375, 2010.

[2] Ieee, *IEEE Standard Glossary of Software Engineering Terminology*, vol. 121990, no. 1. 1990.

[3] K. El Emam, *The ROI from Software Quality*. 2005.

[4] D. Hovemeyer and W. Pugh, "Finding bugs is easy," *ACM SIGPLAN Not.*, vol. 39, no. 12, p. 92, 2004.

[5] C. Weimer and F. Bolger, "Continuous Code Inspection - Advancing software quality at source," *PRQA White Pap.*, pp. 1–8, 2013.

[6] S. Panichella, V. Arnaoudova, M. Di Penta, and G. Antoniol, "Would static analysis tools help developers with code reviews?," *2015 IEEE 22nd Int. Conf. Softw. Anal. Evol. Reengineering*, pp. 161–170, 2015.

[7] J. Novak, A. Krajnc, and R. Zontar, "Taxonomy of static code analysis tools," *MIPRO, 2010 Proc. 33rd Int. Conv.*, 2010.

[8] N. Nagappan and T. Ball, "Static analysis tools as early indicators of pre-release defect density," *Proceedings. 27th Int. Conf. Softw. Eng. 2005. ICSE 2005.*, pp. 580–586, 2005.

[9] S. Heckman and L. Williams, "A systematic literature review of actionable alert identification techniques for automated static code analysis," *Inf. Softw. Technol.*, vol. 53, no. 4, pp. 363–387, 2011.

[10] A. Bacchelli and C. Bird, "Expectations, outcomes, and challenges of modern code review," *Proc. - Int. Conf. Softw. Eng.*, pp. 712–721, 2012.

[11] "Documentation - SonarQube-5.4 - SonarQube." [Online]. Available: http://docs.sonarqube.org/display/SONARQUBE54/Documentation. [Accessed: 09-May-2016].

[12] "checkstyle – Checkstyle 6.15." [Online]. Available: http://checkstyle.sourceforge.net/. [Accessed: 18-Feb-2016].

[13] S. Heckman and L. Williams, "On Establishing a Benchmark for Evaluating Static Analysis Alert Prioritization and Classification Techniques," *Proc. Second ACM-IEEE Int. Symp. Empir. Softw. Eng. Meas.*, pp. 41–50, 2008.

[14] IEEE Standards Association, "Systems and software engineering — Vocabulary ISO/IEC/IEEE 24765:2010," *Iso/Iec/Ieee 24765:2010*, pp. 1–418, Dec. 2010.

[15] N. Godbole, "Software quality assurance: Principles and practice," *Alpha Sci. Int'l Ltd.*, 2004.

[16] Standish Group, "Chaos Demographics," *2004 Third Quart. Res. Rep.*, 2004.

[17] J. S. DAVIS and R. J. LEBLANC, "A Study of the Applicability of Complexity Measures," *IEEE Trans. Softw. Eng.*, vol. 14, no. 9, pp. 636–638, 1988.

[18] N. Fenton and J. Bieman, *Software Metrics: A Rigorous and Practical Approach*, Third Edit. CRC Press, 2014.

[19] T. J. McCabe, "A Complexity Measure," *IEEE Trans. Softw. Eng.*, vol. SE-2, no. 4, pp. 308–320, 1976.

[20] K. Ruohonen, "Graph Theory," *Univ. Turku*, p. 108, 2013.

[21] D. Racodon, "Metrics - Complexity," *SonarQube Documentation*, 2015. [Online]. Available: http://docs.sonarqube.org/display/SONAR/Metrics+-+Complexity. [Accessed: 17-Jun-2016].

[22] G. A. Campbell and P. P. Papapetrou, *SonarQube in Action*. Manning Publications Co., 2013.

[23] G. J. Myers, "An Extension to the Cyclomatic Measure of Program Complexity," *SIGPLAN Not.*, vol. 12, no. 10, pp. 61–64, 1977.

[24] J. J. Vinju and M. W. Godfrey, "What does control flow really look like? Eyeballing the cyclomatic complexity metric," *Proc. - 2012 IEEE 12th Int. Work. Conf. Source Code Anal. Manip. SCAM 2012*, pp. 154–163, 2012.

[25] L. Rosenberg and L. Hyatt, "Software quality metrics for object-oriented environments," *Crosstalk Journal, April*, vol. 10, no. 4, pp. 1–6, 1997.

[26] M. Fowler, "TechnicalDebt," 2003. [Online]. Available: http://martinfowler.com/bliki/TechnicalDebt.html. [Accessed: 15-Apr-2016].

[27] A. G. Bardas, "Static code analysis," *J. Inf. Syst. Oper. Manag.*, vol. 5, no. 2, 2011.

[28] A. German, "Software static code analysis lessons learned," *Crosstalk*, no. November, pp. 13–17, 2003.

[29] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why dont software developers use static analysis tools to find bugs?," *Proc. 2013 Int. Conf. Softw. Eng.*, pp. 672–681, 2013.

[30] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J. P. Hudepohl, and M. a S. E. I. T. on Vouk, "On the value of static analysis for fault detection in software," *Softw. Eng. IEEE Trans.*, vol. 32, no. 4, pp. 240–253, 2006.

[31] T. Zimmermann, R. Premraj, and A. Zeller, "Predicting defects for eclipse," *Proc. - ICSE 2007 Work. Third Int. Work. Predict. Model. Softw. Eng. PROMISE'07*, 2007.

[32] "FindBugs™ - Find Bugs in Java Programs." [Online]. Available: http://findbugs.sourceforge.net/. [Accessed: 18-Feb-2016].

[33] "PMD." [Online]. Available: https://pmd.github.io/. [Accessed: 18-Feb-2016].

[34] S. Wagner, F. Deissenboeck, M. Aichner, J. Wimmer, and M. Schwalb, "An evaluation of two bug pattern tools for Java," *Proc. 1st Int. Conf. Softw. Testing, Verif. Validation, ICST 2008*, pp. 248–257, 2008.

[35] E. Guerra, A. Aguiar, P. Merson, and J. Yoder, "Continuous Inspection: A Pattern for Keeping your Code Healthy and Aligned to the Architecture," p. 13, 2013.

[36] K. Daimi, S. Banitaan, and K. Liszka, "Examining the Performance of Java Static Analyzers," *Proc. Int. Conf. Softw. Eng. Res. Pract. (SERP). Steer. Comm. World Congr. Comput. Sci. Comput. Eng. Appl. Comput.*, p. 7, 2003.

[37] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh, "Using Static Analysis to Find Bugs," *IEEE Softw.*, vol. 25, no. 5, pp. 22–29, 2008.

[38] P. Tomas, M. J. Escalona, and M. Mejias, "Open source tools for measuring the Internal Quality of Java software products. A survey," *Comput. Stand. Interfaces*, vol. 36, no. 1, pp. 244–255, 2013.

[39] A. Bessey, D. Engler, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, and S. McPeak, "A few billion lines of code later," *Commun. ACM*, vol. 53, no. 2, pp. 66–75, 2010.

[40] F. Deissenboeck, E. Juergens, B. Hummel, S. Wagner, B. M. y Parareda, and M. Pizka, "Tool Support for Continuous Quality Control," *IEEE Softw.*, vol. 25, no. 5, pp. 60–67, 2008.

[41] D. Steidl, F. Deissenboeck, M. Poehlmann, R. Heinke, and B. Uhink-Mergenthaler, "Continuous Software Quality Control in Practice," in *2014 IEEE International Conference on Software Maintenance and Evolution*, 2014, pp. 561–564.

[42] "Bitbucket — Server." [Online]. Available: https://bitbucket.org/product/server. [Accessed: 18-Feb-2016].

[43] Mibex Software GmbH, "Sonar for Bitbucket Server | Atlassian Marketplace." [Online]. Available: https://marketplace.atlassian.com/plugins/ch.mibex.stash.sonar4stash/server/overview. [Accessed: 18-Feb-2016].

[44] Nerdwin15 LLC, "Bitbucket Webhook to Jenkins | Atlassian Marketplace." [Online]. Available: https://marketplace.atlassian.com/plugins/com.nerdwin15.stash-stash-webhook-jenkins/server/overview. [Accessed: 18-Feb-2016].

[45] "Jenkins." [Online]. Available: https://jenkins-ci.org/. [Accessed: 18-Feb-2016].

[46] "Eclipse desktop & web IDEs." [Online]. Available: https://eclipse.org/ide/. [Accessed: 18-Feb-2016].

[47] R. Plösch, A. Mayr, G. Pomberger, and M. Saft, "An Approach for a Method and a Tool Supporting the Evaluation of the Quality of Static Code Analysis Tools," *Proc. SQMB 2009 Work. held conjunction with SE 2009 Conf. Publ. as Tech. Rep. TUM-I0917 Tech. Univ. Munich*, no. 1, pp. 37–44, 2009.

[48] "Java Plugin - Plugins - SonarQube." [Online]. Available: http://docs.sonarqube.org/display/PLUG/Java+Plugin. [Accessed: 23-Mar-2016].

[49] "What is Object/Relational Mapping? - Hibernate ORM." [Online]. Available: http://hibernate.org/orm/what-is-an-orm/. [Accessed: 29-Apr-2016].

[50] R. K. Merton and P. L. Kendall, "The Focused Interview," *Am. J. Sociol.*, vol. 51, no. 6, pp. 541–557, 1946.

[51] "Are there any default set of rules for SonarQube other than the default rules from the Sonar Java plugin? - Stack Overflow." [Online]. Available: https://stackoverflow.com/questions/36429227/are-there-any-default-set-of-rules-for-sonarqube-other-than-the-default-rules-fr. [Accessed: 18-May-2016].

[52] SonarQubeCommunity, "SonarQube Build Breaker Plugin," 2016.

[Online]. Available: https://github.com/SonarQubeCommunity/sonar-build-breaker. [Accessed: 09-May-2016].

[53] O. Gaudin, "SonarQube™ » Why You Shouldn't Use Build Breaker." [Online]. Available: http://www.sonarqube.org/why-you-shouldnt-use-build-breaker/. [Accessed: 10-May-2016].

[54] J. A. De Oliveira, E. M. Fernandes, and E. Figueiredo, "Evaluation of Duplicated Code Detection Tools in Cross-Project Context."

[55] R. Baggen, J. P. Correia, K. Schill, and J. Visser, "Standardized code quality benchmarking for improving software maintainability," *Softw. Qual. J.*, vol. 20, no. 2, pp. 287–307, 2012.

[56] T. L. Alves, C. Ypma, and J. Visser, "Deriving metric thresholds from benchmark data," *IEEE Int. Conf. Softw. Maintenance, ICSM*, 2010.

[57] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner, "Do code clones matter?," *Proc. - Int. Conf. Softw. Eng.*, pp. 485–495, 2009.

[58] D. Rattan, R. Bhatia, and M. Singh, *Software clone detection: A systematic review*, vol. 55, no. 7. Elsevier B.V., 2013.

[59] E. Bouwers, J. Visser, and A. van Deursen, "Getting what you measure," *Commun. ACM*, vol. 55, no. 7, p. 54, 2012.

[60] "java - Receive feedback from SonarQube in regards to design and architecture - Stack Overflow." [Online]. Available: http://stackoverflow.com/questions/35983244/receive-feedback-from-sonarqube-in-regards-to-design-and-architecture. [Accessed: 12-May-2016].

Appendix A Rule Configuration Tables

Raw results from the supervisor rule investigation.

#	Rule ID	Default Ranking	New Ranking
1	squid:S1135	Info	Blocker
2	squid:S1181	Blocker	Major
3	squid:S2095	Blocker	Major
4	squid:S1166	Critical	Blocker
5	squid:S00112	Critical	Major
6	squid:S2696	Critical	Deactivated.
7	squid:S1948	Critical	Major
8	squid:S2077	Critical	Deactivated
9	squid:S1148	Critical	Blocker
10	squid:S1989	Critical	Major
11	squid:S2142	Critical	Deactivated
12	squid:S2386	Critical	Deactivated
13	squid:S2184	Critical	Major
14	squid:S2885	Critical	Deactivated
15	squid:S1872	Critical	Major
16	squid:S2068	Critical	Major
17	squid:S1163	Critical	Blocker
18	squid:S1862	Critical	Blocker
19	squid:S106	Critical	Deactivated
20	squid:S1149	Major	Blocker
21	squid:CommentedOutCodeLine	Major	Blocker
22	squid:S1186	Major	Deactivated
23	squid:S135	Major	Deactivated
24	squid:S1197	Minor	Major
25	squid:S00117	Minor	Deactivated
26	squid:S1488	Minor	Deactivated
27	squid:S1488	Minor	Deactivated
28	squid:S00122	Minor	Major
29	squid:S1213	Minor	Major

30	squid:S1905	Minor	Major
31	squid:ModifiersOrderCheck	Minor	Major
32	squid:S1214	Minor	Major
33	squid:S1659	Minor	Major
34	squid:UselessImportCheck	Minor	Deactivated
35	squid:S1125	Minor	Major
36	squid:S1170	Minor	Major
37	squid:S00100	Minor	Deactivated
38	squid:S1192	Minor	Major
39	squid:RedundantThrowsDeclarationCheck	Minor	Deactivated
40	squid:S1153	Minor	Major
41	squid:S1301	Minor	Major
42	squid:S2065	Minor	Major
43	squid:S1873	Critical	Major
44	squid:S2275	Critical	Major
45	jproperties:separator-convention	Minor	Deactivated
46	jproperties:key-naming-convention	Minor	Deactivated
47	jproperties:line-length	Minor	Major
48	jproperties:empty-line-end-of-file	Minor	Major

Rule ID-Name Linking table.

Rule ID	Rule name
squid:S1166	Exception handlers should preserve the original exception.
squid:S1149	Synchronized classes Vector, Hashtable, Stack and StringBuffer should not be used.
squid:CommentedOutCodeLine	Sections of code should not be "commented out".
squid:S2583	Conditions should not unconditionally evaluate to "TRUE" or to "FALSE".
squid:S1135	"TODO" tags should be handled.
squid:S1148	Throwable.printStackTrace(...) should not be called.
squid:S2259	Null pointers should not be dereferenced.
squid:S1143	"return" statements should not occur in "finally" blocks.
squid:ModifiersOrderCheck	Related "if/else if" statements should not have the same condition.
squid:S1854	Dead stores should be removed.
squid:MethodCyclomaticComplexity	Methods should not be too complex.