Bachelor's Programme in Science and Technology

Software quality tracking with static analysis

Utilization and deployment of static software metrics in software projects

Santeri Suitiala

BACHELOR'S THESIS

Aalto University
BACHELOR'S THESIS 2018

Software quality tracking with static analysis

Utilization and deployment of static software metrics in software projects

Santeri Suitiala

Thesis submitted in partial fulfillment of the requirements for the degree of Bachelor of Science in Technology.
Otaniemi, November 13, 2018

Supervisor (ABB): Jorma Keronen
Supervisor (Aalto): Marko Hinkkanen
Advisor: Markus Turunen

Aalto University
School of Electrical Engineering
Bachelor's Programme in Science and Technology

Author
Santeri Suitiala

Title
Software quality tracking with static analysis

School School of Electrical Engineering

Degree programme Bachelor's Programme in Science and Technology

Major Electronics and Electrical Engineering **Code** ELEC3013

Supervisor (ABB) Jorma Keronen

Supervisor (Aalto) Marko Hinkkanen

Advisor Markus Turunen

Level Bachelor's thesis **Date** November 13, 2018 **Pages** 15 **Language** English

Abstract

Lorem ipsum.

Keywords software metrics, software quality

urn https://aaltodoc.aalto.fi

Tiivistelmä

Lorem ipsum.

Avainsanat ohjelmistometriikat, ohjelmiston laatu

urn https://aaltodoc.aalto.fi

Contents

1. Introduction

"The notion of 'software engineering' was first proposed in 1968" (Sommerville, 2011). Ever since the profession started, software projects have become bigger and more complex. This is because the hardware has been growing even faster and so software engineers have been having a hard time keeping the increasing phase up (Brooks & Kugler, 1987). The increasing code complexity raises new problems with understanding existing code and increases probability of software flaws. Software complexity can be later decreased by refactoring code. Refactoring can mean e.g. removing duplicate code by abstracting, renaming and commenting the code to make it more readable.

Software metric is a quantitative value calculated from a piece of a code, a whole software project or even from a software process. Software metrics are used to track software code and process attributes to determine whether the software has improved or not. This thesis focuses on static product metrics. Knowing the change of the software quality over time makes it possible to plan and allocate resources to fix software with poor quality.

ABB Drives manufactures industrial drives which are controlled by software. Industrial drives are made to last for a long period of time and the same goes for the software inside the machine. Also new bugs are found and customers demand new features, which makes the software evolve rapidly over time. The same effect that Brooks discovered over 30 years ago is something that still exists in modern companies: software size and complexity increases over time. Without refactoring, the software may become ambiguous and difficult to modify for the developers. If the project source code is too complicated, modifying might become a high risk factor. On the other hand, if too much time is spent refactoring and improving the code, no new feature gets developed and customers are left unsatisfied. In

a big company, it is difficult to perceive the whole picture and so the golden mean of internal improving and feature development can be hard to find.

This thesis introduces the underlying theory of software metrics. Furthermore a part of the task is also to determine the core metrics by getting familiar with the most commonly used metrics, try selected metrics in the real world and make conclusions. In a long term, this thesis aims to improve the understanding of software quality with the help of systematic and automatic daily static software analysis. Analysis calculates different core software metrics and ideally represents a long-term graph to determine whether the quality and other attributes of the software is increasing or decreasing and how rapid is the change.

The ultimate goal of this thesis is to help the company's software development teams and management to decide when to refactor code and improve tools and dependencies and how much time should be used for it.

2. Theory behind software metrics

Software measurement is a quantitative value calculated from a software process or system. A software metric can either be a software measurement or a function of many different software measurements. Software metrics are used to track e.g software quality or cost estimation. Tracking the quality of software makes it easier and more efficient to see whether software system or process has improved or not. Knowing the direction of the software quality makes it possible to plan and give resources to fix software or process with poor quality or efficiency.

2.1 Classification of software metrics

Software metrics can be divided into product, process and project metrics. Project metrics (Thakur, n.d.) help project managers to estimate projects by tracking for example amount of developers or delivered lines of code. This helps to estimate similar projects in the future. Process metrics measures values concerning software development processes. Thakur lists e.g. number of errors found before the software release and conformity to schedule to be process metrics. These metrics help project stakeholder to perceive how the project is proceeding. Product metrics gives measurements from the developed software product (Sommerville, 2011). A simple example of a process metric is source lines of code (SLOC) which simply tells the number of lines of code the whole program has altogether (Nguyen, Deeds-Rubin, Tan, & Boehm, 2007). Figure 2.1 present and help to understand the division of different kind of software metrics. This thesis focuses on product metrics.

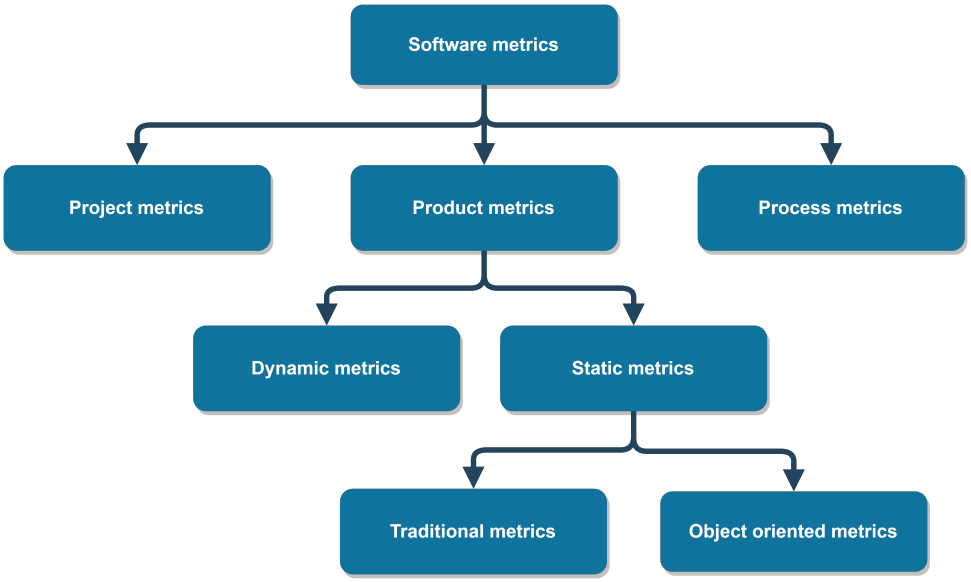

Figure 2.1. Software metric types

2.1.1 Product metrics

Many defined sets of product metrics exists. One defined set is e.g Halstead metrics (Al Qutaish & Abran, 2005). It is also possible to adjust an existing metric for specified software or implement a completely new metric. A product metric can either be dynamic or static (Sommerville, 2011). Dynamic metrics are measured when a program is executing and static metrics can be measured e.g by reading the source code or documentation of a program. This thesis focuses on static code analysis. Dynamic metric such as server uptime is easy to interpret as 100% would be the wanted outcome. Static metrics has the problem that those can often be ambiguous and easily to be misinterpreted. For example SLOC can be 5000 in one product and one million in other. That doesn't mean that other is better quality software than the other. This thesis study static product metrics. Dynamic metrics are also very important for projects but the company is already using

2.1.2 Static metrics

The magnitude of a static product metric does not matter. We are more interested about the change of the metric over a defined amount of time. Other approaching method for static metrics is comparing different results inside the project that is being analyzed. If a part of a software project has anomalous attributes, it should be investigated further.

2.1.3 Traditional and object-oriented metrics

Traditional metrics can be calculated from any kind of software even if the source code is object-oriented. These metrics can be categorized as "non-object-oriented" metrics. Traditional metrics present size of the program and help stake holders to perceive a more accurate picture about what kind of program is being developed. Traditional static metrics are for example SLOC and cyclomatic complexity (Fenton & Bieman, 1997).

Object-oriented metrics (OOM) are specific for object-oriented programming languages. OOM measurements present structure of a project and tells for example if classes are abstract enough for the software to be maintainable. A study has estimated corrective maintenance cost saving to be 42% when OOM are being used (Sarker, 2005). A good example of OOM is Chidamber and Kemerer Metrics of then referred as CK Metrics (Chidamber & Kemerer, 1994).

2.2 Software quality

As bigger and more complex software systems is being developed, the need for quality in these systems has raised. In the 1960s the growth of software projects led to projects lacking maintainability, reusability and reliability (Sommerville, 2011). This lead to different approaches to control software quality in order to make software with less defects and more profit. Defining software quality can be problematic because there exists no one single truth for software quality. Quality has many different attributes such as reliability, security and portability (Sommerville, 2011). Every quality attribute can not be taken into account at the same time as improving one attribute of the software can have a decreasing effect on other attribute. Only some of the attributes can be choosed to define if a project has good quality or not. This can lead to conflicts between stakeholders as they can have differentiating points of view and quality attribute preferences about what defines the quality of a software project. Quality management is particularly important for large software systems that takes years to develop. On smaller systems, quality management can be useful but not as mandatory.

2.2.1 Software complexity and maintainaility

Lorem ipsum.

2.2.2 Software modularity and reusability

Lorem ipsum.

3. Determining useful metrics

Determining the right software metrics is not always a simple task. In this case. the target will be static product metrics (see chapter 2). No predefined package of metrics exists that works for every software project. Every project is unique which means that new set of metrics should be selected for each project. However if two projects have a lot in common, the same metrics can be applied. Even after defining metrics, static measurements can be ambiguous due to the fact that the measurement outputs are just numbers. The real correlation of the measurements to the software quality may be easy to misinterpret. The measurement data must be analyzed in order to be able to understand what attributes, if any, it correlates (Sommerville, 2011).

3.1 Motivation to implement static analysis

Recently, upper management has stated clear instructions that software quality must be improved in the company. A certain part of the working effort is allowed to be used for source code refactoring and internal improving. Static software analysis will be implemented to help stake holders to perceive a picture of different quality attributes of the software. As stated, software complexity is commonly increasing and the target company of this thesis is no different.

3.1.1 Software metrics in agile development process

The software development team that is the target of this thesis is using an agile software development process. Agile is not a software development process but rather a set of rules that agile processes try to follow (authors, 2001).

3.2 Code smells

Static software analysis contains possibility to use code smells (see subsection 4.1.1). Code smell is a small and fast check for any characteristic that might indicate a software problem **??**. A smell does not mean that a problem exists but rather that the source code should be reviewed and, if a problem exists, fixed. Code smell usage might be valuable but in a bigger project, so many smells will be present that the few important smells, that are crucial for the project quality, will not be found. Code smells does not add much value when used in static software analysis and should rather be used to check quality on each commit to check smells for the files to be committed (Tufano et al., 2015). This means that code smells are not implemented to the analysis. If needed, a development team should implement a separate code analysis to analyze each commit to the source code. This could be achieved for example with tools like Microsoft Style Cop[1]. The goal of this thesis is to implement a static software analysis which is run daily and not on each commit. Code smells will not be implemented to the analysis.

3.3 Source code examples

To really test if the metrics are correlating the wanted software quality attributes, we need to first determine low and high quality source code examples (Coleman, Ash, Lowther, & Oman, 1994). These examples should be good or bad quality code according to the whole development teams opinion. After we have determined what good and bad quality concretely means we can start to figure out which metrics correlate to the quality.

3.4 Trial and error approach

As there exists no set of predefined rules that could be used in this context, only logical way is to make a sophisticated guess about what the suitable metrics could be. This can be done iteratively so that chosen metrics are used a defined amount of time and then evaluated again. To do this there must be a way to determine whether the metrics are correlating the wanted attribute or not. One of the goals of this thesis is to implement a first iteration of metrics by choosing metrics according to literature and

[1] https://github.com/StyleCop/StyleCop

guessing. The meaning of each chosen metric must be known. If the quality attribute correspondence of a chosen metric is uncertain, it will be impossible to use correctly and will become a number on the board without meaning.

4. Implementing analysis for a development team

After we are sure that the software metrics really are useful and capable of indicating software quality it is time to make use of the metrics. In this chapter a proof of concept will be implemented and possibly introduced for the software development team's use.

4.1 Essential tools for implementation

For a concrete implementation, some software tools are needed. The preference is that metrics are easily visible and understandable for each developer. Metrics are not meant to be a stressing factor which might happen if the metrics are forced to be seen on a day-to-day basis. Metrics should rather be an auxiliary tool to help developers perceive a bigger picture regarding source code. No comparison was done while choosing the tools and the main idea is to use tools that are easily available or already in use by the company.

4.1.1 Software analysis tool

CppDepend[1] is a commercial tool specifically made to analyze C/C++ code. It provides many of the common software metrics and has a good-looking user interface. There are many other promising tools but investigating available options is out of scope on this thesis. Using this tool gives a good indication if software metrics are useful for the software development team or not. If the tool later turns out to be exchangeable with a more usable one, it can easily be replaced.

[1] https://www.cppdepend.com/

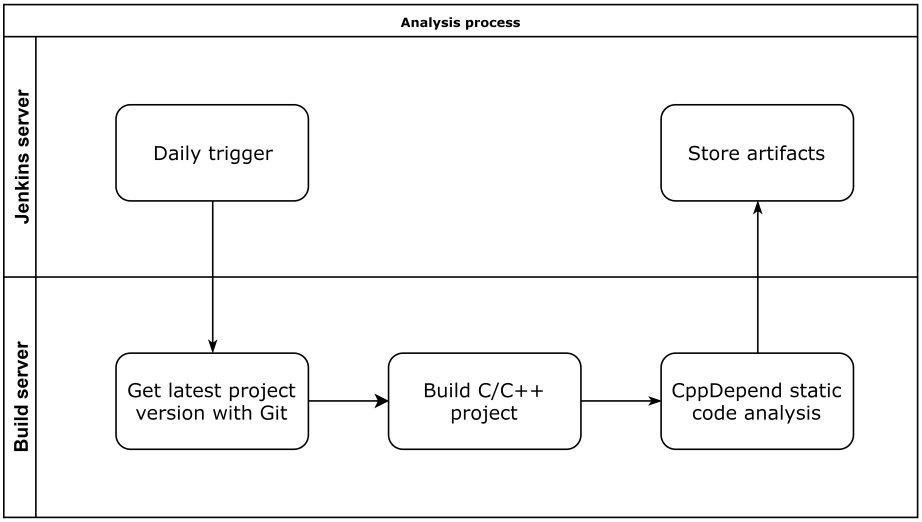

Figure 4.1. System funcionality

4.1.2 Analysis automation tool

Jenkins[2] is an open-source continuous integration tool. Jenkins is a server that can easily be set to trigger for example CppDepend to analyze source code daily. It can also be set to store artifacts which in this case would be the analysis outcome in Hypertext Markup Language (HTML) format. The strength of Jenkins is that anyone in the research and development department who has access to the internet can observe the outcome of the analysis. Source code is updated to the latest version with the help of Git[3]. Git is a distributed version control system that is widely used by globally (Skerrett, 2014).

4.1.3 Build server

Analysis can not be done on any computer. It must be a computer that is constantly powered on and online. This way the analysis will be run constantly and enables continuous data for different graphs. For this purpose a build server will be set up. Jenkins will be connected to this computer and will tell the server to do the required commands. A local copy of the source code is stored in the server and it will always have the latest version of the software available by using Get before any analysis.

[2]https://jenkins.io/
[3]https://git-scm.com/

4.2 System description

The selected tools will form a system that will create desired analysis daily to the Jenkins server where it can be inspected by everyone. The system functionality is illustrated in figure 4.1. Jenkins can be set to periodically call a sequence of commands. In this the period will be one day. The source code will be upgraded to its latest version with version control system, Git. After the source code is upgraded, it will be compiled and analyzed by the CppDepend tools. CppDepend will return analysis artifacts as a HTML file to the Jenkins server, which will be stored and viewable in the Jenkins user interface. This way the process will be transparent and available to the whole software development team.

The whole process takes only about ten minutes to run and is not a strenuous task for the build server. The analysis tool does an incremental analysis, which means that if some part of the source code is not changed, it will be copied from the last report and not analyzed again.

5. Conclusions

It has become clear that software metrics are an important factor in the modern software development business.

Quality is a big concern in large software projects that take long time to develop.

The company can analyze output data of the made static software analysis and possibly decide common quality standards for software development.

The next step is to implement the static analysis for other development teams aswell. This is made easier by making the scripts and Jenkins configurations easy to modify.

Bibliography

Al Qutaish, R. E., & Abran, A. (2005). An analysis of the design and definitions of halstead's metrics. In *15th int. workshop on software measurement (iwsm'2005). shaker-verlag.*

authors, A. M. (2001). Manifesto for agile software development. `http://agilemanifesto.org/`. (Accessed: 28.10.2018)

Brooks, F., & Kugler, H. (1987). *No silver bullet*. University of North Carolina at Chapel Hill.

Chidamber, S. R., & Kemerer, C. F. (1994). A metrics suite for object oriented design. *IEEE Transactions on software engineering*, *20*(6), 476–493.

Coleman, D., Ash, D., Lowther, B., & Oman, P. (1994). Using metrics to evaluate software system maintainability. *Computer*, *27*(8), 44–49.

Fenton, N., & Bieman, J. (1997). *Software metrics: a rigorous and practical approach*. CRC press.

Nguyen, V., Deeds-Rubin, S., Tan, T., & Boehm, B. (2007). A sloc counting standard. In *Cocomo ii forum* (Vol. 2007).

Sarker, M. (2005). An overview of object oriented design metrics. *From Department of Computer Science, Umeå University, Sweden*.

Skerrett, I. (2014). Eclipse community survey 2014 results. `https://ianskerrett.wordpress.com/2014/06/23/eclipse-community-survey-2014-results/`. (Accessed: 26.10.2018)

Sommerville, I. (2011). *Software engineering*. Boston : Pearson.

Thakur, D. (n.d.). Classification of software metrics in software engineering. `http://ecomputernotes.com/software-engineering/classification-of-software-metrics`. (Accessed: 24.10.2018)

Tufano, M., Palomba, F., Bavota, G., Oliveto, R., Di Penta, M., De Lucia, A., & Poshyvanyk, D. (2015). When and why your code starts to smell bad. In *Proceedings of the 37th international conference on software*

engineering-volume 1 (pp. 403–414).