Bachelor's Programme in Electrical Engineering

Software quality tracking using static product metrics

Utilization and deployment of static software metrics in software projects

Santeri Suitiala

Software quality tracking using static product metrics

Utilization and deployment of static software metrics in software projects

Santeri Suitiala

Author
Santeri Suitiala

Title
Software quality tracking using static product metrics

School School of Electrical Engineering

Degree programme Bachelor's Programme in Electrical Engineering

Major Electronics and Electrical Engineering   **Code** ELEC3013

Supervisor (ABB) Jorma Keronen

Supervisor (Aalto) Marko Hinkkanen

Advisor Markus Turunen

Level Bachelor's thesis   **Date** December 15, 2018   **Pages** 19   **Language** English

Abstract

Evaluating software quality is not an easy task especially for large projects. This thesis tries to improve software quality tracking by implementing a static software product metric analysis. Wanted metrics depend on the wanted quality attributes that are chosen for the project. This means that a new set of metrics must be chosen for each project. Six chosen metrics will be used in order to evaluate software quality with better accuracy.

Keywords software metrics, software quality

urn https://aaltodoc.aalto.fi

Tiivistelmä

Ohjelmistoprojektit kasvavat ja monimutkaistuvat vuosi vuodelta. Ohjelmiston koon kasvaessa, sen laatu heikkenee välttämättä. Laadun heikkeneminen vaikeuttaa lähdekoodin ymmärrettävyyttä ja voi altistaa ohjelmiston virheelliselle toiminnalle. Ohjelmiston laatu on tärkeä tekijä varsinkin suurissa ja pitkäikäisissä projekteissa, joissa kirjoitettua lähdekoodia saatetaan muokata vielä monien vuosien kuluttua. Jos laatu heikkenee liikaa, ohjelmiston ylläpidettävyydestä voi tulla erittäin haastavaa. Laatua voidaan parantaa projektin myöhemmässä vaiheessa lähdekoodin uudelleenjäsentelyllä eli re-faktoroimalla (engl. refactoring). ABB Drives tuottaa taajuusmuuttajia, joita ohjataan laitteen sisäisellä ohjelmistolla. Taajuusmuuttajat valmistetaan kestämään vuosikymmeniä ja ohjelmistoja kehitetään ja päivitetään jatkuvasti.

Lähdekoodille voidaan suorittaa mittauksia joiden tuloksena saadaan kvantitatiivinen luku, jota kutsutaan ohjelmistometriikaksi. Ohjelmistosta voidaan mitata esimerkiksi monimutkaisuutta tai modulaarisuutta. Työn tavoitteena on tutkia, mitkä ohjelmistometriikat voisivat olla yritykselle hyödyllisiä. Lisäksi tarkoituksena on toteuttaa staattinen ohjelmistometriikka-analyysi, joka laskee päivittäin valitut ohjelmistometriikat uusimmasta integroidusta ohjelmistosta.

Metriikoiden valitsemiseen ei ole olemassa valmiita ohjeita tai metriikkakokonaisuuksia, ja valinta on aina projektikohtainen. Metriikat täytyy valita kirjallisuuden ja omien johtopäätösten avulla. Työssä valittiin perustellusti kuusi metriikkaa jotka mittaavat sekä lähdekoodin monimutkaisuutta että modulaarisuutta. Metriikat ovat itsessään vain numeroita ja ne voivat olla hyvinkin epäselviä ilman aiempaa tietämystä. Metriikoita voidaan tulkita kahdella tavalla: vertaamalla tuloksia keskenään projektin eri osa-alueisiin tai tutkimalla pitkällä aikavälillä, kasvaako vai pieneneekö metriikka. Metriikkaa tulkitsevien ohjelmoijien ja päättäjien tulee tietää tarkasti, mitä metriikka tarkoittaa ja minkälaisia tuloksia on odotettavissa koska metriikkaa on erittäin helppo tulkita väärin.

Staattinen metriikka-analyysi toteutettiin yhdelle ohjelmistokehitystiimille. Analyysi suoritetaan päivittäin automaattisesti ohjelmistolle, jota valikoitu ryhmä kehittää. Metriikoiden laskemiseen löytyy monta käytännöllistä aputyökalua mikä helpottaa automatisoidun analyysin toteuttamista. Analyysin tulos on kehittäjille helposti saatavilla työpaikan sisäisessä verkossa ja kuka vain voi mennä tarkastelemaan tuloksia.

Ohjelmistometriikan seuraamisen tärkeys on tullut selväksi yrityksessä. Metriikat ovat hyvin käytännöllisiä ja havainnollistavia varsinkin isojen projektien osalta. Metriikkaa on helppo tulkita väärin ja väärin tulkittuna niistä voikin olla enemmän haittaa kuin hyötyä. Valitun ohjelmistometriikan hyöty on vielä epäselvä ja metriikkaa tulisikin arvioida uudelleen heti kun se on mahdollista. Tulevaisuudessa testiryhmässä hyväksi havaitusta metriikasta voidaan luoda yrityksen ohjelmistokehitykselle yleinen laatustandardi koska monet projektit ovat hyvin samankaltaisia. Analyysi on tehty niin, että se on helppo toteuttaa myös muille yrityksen ohjelmistokehitystiimeille. Jos analyysi todetaan hyödylliseksi, seuraava askel on toteuttaa samankaltaisia analyysejä myös muille projekteille.

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 complexity has been growing faster and faster and 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 for example 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 applies for the software inside the drive. 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 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. 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. In a long term, this thesis aims to improve the understanding of software quality with the help of systematic and automatic daily static software metric 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 are increasing or decreasing and how rapid the change is.

By achieving these goals, this thesis could increase the knowledge about software quality and help the company's software development teams and management to decide when to refactor code and improve tools and dependencies. This way, the development teams can estimate better, how much time and effort should be used for improving software quality.

2. Theory behind software metrics

Software measurement is a process that outputs 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 for example 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 development makes it possible to plan and give resources to fix the 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, 2014) help project managers to evaluate and plan projects by tracking for example the 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 for example 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 product 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 presents and helps to understand the division of different kinds of software metrics. This thesis focuses on product metrics.
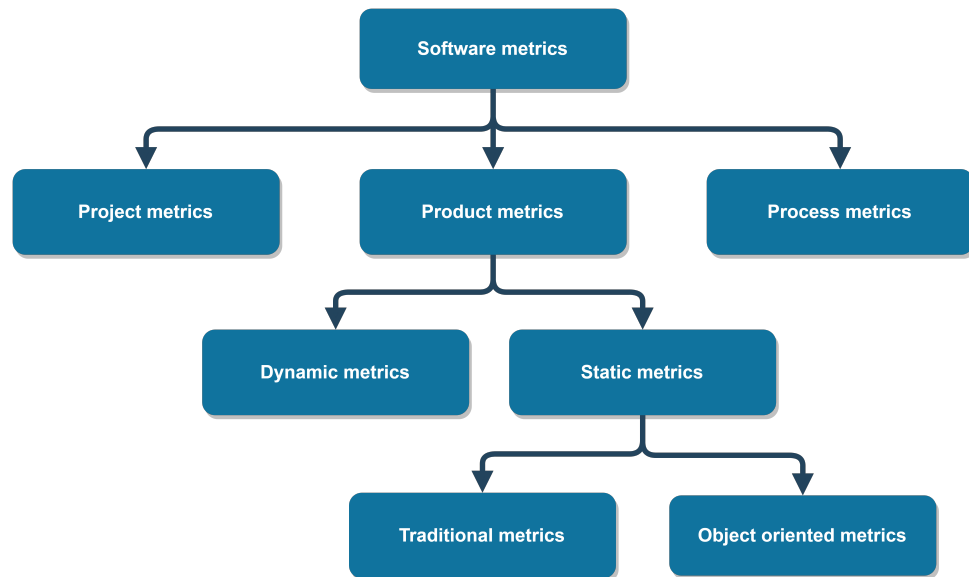
Figure 2.1. Software metric types

2.1.1 Product metrics

Many defined sets of product metrics has been declared in the literature. One defined set is for example 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 for example by reading the source code or documentation of a program. This thesis focuses on static metric analysis. Dynamic metrics such as server uptime are often easy to interpret as 100% would be the wanted outcome. However, static metrics can often be ambiguous and easily to be misinterpreted. For example SLOC can be 5000 in one product and one million in another. That does not mean that the other is better quality software than the other. This thesis studies static product metrics. Dynamic metrics are also very important for projects, but the company is already using dynamic testing systematically as a tool. Static software analysis has also already been implemented and solely metrics are analyzed instead of analyses that for example check common coding practices.

2.1.2 Static metrics

Static metrics can be inspected by comparing the measurement from different files and modules or by plotting metric trend data. Measurements can be ambiguous because the outcome depends from many different factors.

For example used programming language is one factor. One interpretation 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. Another method is to define a limit for a good measurement according to literature or just guessing and later adjusting the limit according to own preferences. When inspecting trend, the magnitude of a static product metric does not matter. Trends of metrics tell if the quality of software is increasing or decreasing to give more profound perceiving of a project.

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 a 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 tell, 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 metrics are being used (Sarker, 2005). A good example of OOM metrics are Chidamber and Kemerer Metrics, often referred as CK Metrics (Chidamber & Kemerer, 1994).

2.2 Software quality

As bigger and more complex software systems are 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 caused the initiation of 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 chosen 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

Structural software complexity measurement is considered as an important factor in software maintainability (Banker, Datar, & Zweig, 1989). Software has essential complexity that is needed in order to have a functional system and non-essential complexity that is made by accident and can be removed (CASTsoftware, 2014). This means that software must have some level of complexity, but some of the complexity can be removed with refactoring. Unfortunately, complexity is only a number and the amount of non-essential complexity is undefinable. Nevertheless, complexity measurement can hot spot complex parts of a project and indicate, whether complexity is increasing or decreasing. Complex code should be refactored into smaller tasks to make code more understandable and more easily maintainable. The maintenance of a source code can be two or three times more expensive than the implementation (Banker et al., 1989). Most widely used metric for complexity has been McCabe's cyclomatic complexity (Banker et al., 1989). Code with high cyclomatic complexity has many independent execute paths and is difficult to understand.

2.2.2   Software modularity

Modularity tells how source code is structured. Modularity is split into relationships between the elements inside a module, cohesion, and relationships between modules, coupling (Stevens, Myers, & Constantine, 1999). An example of cohesion and coupling is illustrated in Figure 2.2. For example module 1 would have one cohesion relationship and three coupling relationships from other modules. Modules can be packages, domains, namespaces or something else, depending which programming language is being used (Gupta & Chhabra, 2009).

When structured right, modularity decreases complexity as discussed in subsection 2.2.1. Structured code help developers to change the code
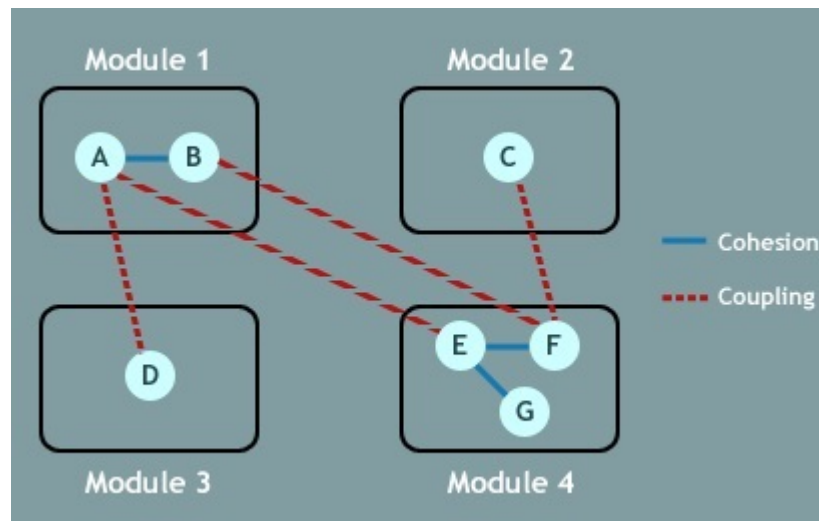
Figure 2.2. Cohesion and coupling (Dhanvani, 2013)

later easier as the code is clearly divided in many different modules. Other aspect of modularity is reusability. Modules should be independent as possible to be reusable, meaning that coupling should be as low as possible. If coupling is low, changing an element inside a module means that no other modules must be changed, making the module more independent and reusable. Cohesion inside a module should be high enough to make a module useful. It makes no sense to make many modules that only contains one element because the idea of modularization is to group similar elements together. If modules are independent from other modules, those can be reused in other projects or in other parts of the project while still preserving maintainability. If modules are not independent, changing a piece of a code might change something from other modules and makes refactoring a much more difficult task.

Coupling is divided into two parts: efferent coupling and afferent coupling (Gupta & Chhabra, 2009). Efferent coupling means, how many elements from other modules are dependent of the elements inside the module under inspection. Afferent coupling is the opposite, meaning how many elements from inside the inspected module is dependent of outside modules. If module has high efferent and low affarent coupling the module is unstable. High affarent and low efferent coupling means that the module is stable. Instability is not a bad attribute but a module should clearly be either stable or unstable to have a clear meaning (Gupta & Chhabra, 2009).

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. 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

Quality is an area that ABB want to invest in more and more. Quality product needs, of course, quality software. A certain part of the working effort is allowed to be used for source code refactoring and internal improving. Software metrics analysis will be implemented to help stakeholders to perceive a picture of different quality attributes of the software. The need for software metrics is unclear, but the underlying literature and theory seem to be promising. The idea is to take metrics into use and determine after a short period of usage, if metrics are valuable or not.

3.2 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 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.

3.2.1   Code smells

Static software analysis contains possibility to use code smells. Code smell is a small and fast check for any characteristic that might indicate a software problem (Fowler, 2006). A smell do 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 daily 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.

3.2.2   Complexity metrics

Three complexity metrics will be selected to analyze maintainability (see: subsection 2.2.1) from different points of view. Cyclomatic complexity is a widely used metric that tells number of ways that a program can be executed. It is a good way to see, for example, if a part of a program has a complex switch sentence or something similar, that is difficult to read. Lines of code metric indicates how much source code the project has. This metric is the most unambiguous metric that is being selected because it is very intuitive to think that big projects are more difficult to maintain. Lines of code helps less experienced programmers to understand, whether

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

Table 3.1. The chosen metrics

Metric name	Quality attribute
Cyclomatic complexity	Complexity
Lines of Code	Complexity
Duplicate code	Complexity
Afferent coupling	Modularity
Efferent coupling	Modularity
Cohesion	Modularity

complexity is increasing or decreasing. Third is duplicate code which tells percentage of code duplications in the code base. If, for example, a method is found in two different places, changing of that method means that the change must be implemented twice. Sometimes a big project with a big lines of code value can be easier to maintain than a small project. This is where cyclomatic complexity and duplicate code percentage should be used to analyze complexity more deeply.

3.2.3 Modularity metrics

Also three metrics that tend to correlate modularity, are being selected. Afferent and efferent coupling (see subsection 2.2.2) will tell how mixed the modules are. Modules should have low coupling and they should be either stable or unstable as possible. Coupling will be inspected in module level but could also be analyzed for class level to see how stable or unstable individual classes are. Class level coupling however, is not selected as core metrics in this case. Cohesion means total coupling between classes inside each module. If cohesion is low, the inspected module might be useless meaning that either more elements must be added or the few elements that the module has, must be moved elsewhere. These three metrics can be very ambiguous if not understood correctly. Modular refactoring can be very laborious and might be almost impossible or too much of a time waste, if the project is big and heavily coupled. However it might offer a big improvement concerning reusability and maintainability. The chosen metrics and their main corresponding attributes are listed in Table 3.1. The metrics may correspond to other quality attributes aswell.

3.3 Source code examples

To really test if the metrics are correlating the wanted software quality attributes, low and high quality source code examples are needed to be determined (Coleman, Ash, Lowther, & Oman, 1994). These examples should be good or bad quality code according to the whole development teams opinion. After good and bad quality has been concretely defined, figuring out which metrics correlate to the quality can be initiated.

4. Implementing analysis for a development team

After making 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 use of a software development team.

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. The tool automatically provides code smells as explained in subsection 3.2.1. Code smells will be disabled from CppDepend analysis.
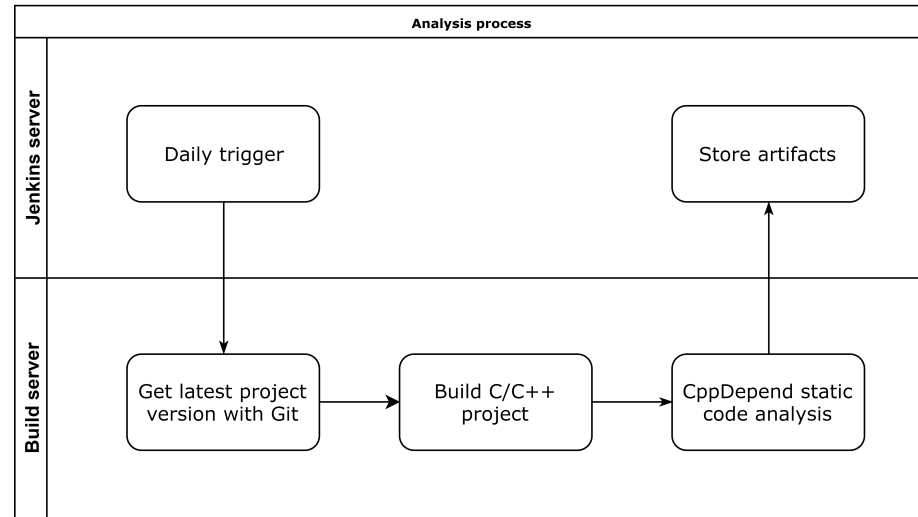
---

[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 all over the world (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 Git before any analysis.

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

13

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 case, 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.

4.3 System outcome

CppDepend does not support duplicate code metrics. Five metrics, cyclomatic complexity, lines of code, cohesion, afferent and efferent coupling, has been analyzed daily during a period of four weeks. Table 4.1 shows the latest results of the analysis. Column "Result 1" shows the results from the first week and the "Result 2" is the results from the fourth week. First clear result is that each of the metrics have changed less than one percent during the inspection period. Most of the metrics have small changes which is very natural for a software project. Second result is that the metrics are as ambiguous as thought. In this case, types mean classes as the programming language used is C/C++.

Complexity metrics are good metrics to watch over time to see if complexity is increasing or decreasing. Total lines of code and cyclomatic complexity have increased by small amount. These complex metrics indicate that something new have been implemented to the source code which has increased the complexity. If new refactoring practices are implemented, these metrics can be used to evaluate if the practices are decreasing complexity growth or not.

Modularity could not be analyzed for bigger modules than classes. Analyzing classes will be sufficient for evaluating the metrics. Cohesion and

Table 4.1. Analysis results

Metric name	Result 1	Result 2
Average Cyclomatic complexity for types	40.28	40.51
Total Lines of Code	68 185	68 200
Duplicate code	N/A	N/A
Average Afferent coupling for types	0.87	0.87
Average Efferent coupling for types	0.93	0.94
Lack of Cohesion of methods	0.03	0.03

average efferent and afferent coupling for classes are the most difficult and ambiguous metrics in this analysis. The reason that the results are so small values might be due to broken dependencies in the analysis project. This means that the project is not setup completely correctly in the analysis tool. Results can still be used to evaluate whether the metrics are useful or not. These modularity metrics have not changed much. This is understandable because the result values are averages and such small numbers. Average modularity metrics are not very informative because modules can be stable or unstable (see subsection 2.2.2). This means that the quality of modules cannot be evaluated from a average value. Modularity metrics should be calculated for each individual module in order to get more informative data about modularity. This thesis analysed types but a better point of inspection could be namespaces or some other bigger modules.

The analysis tool allows individual classes, methods and namespaces to be inspected. This could be used to further analyse project modularity but should not be included to the analysis. Analysis of each element in code base would fill the analysis full of non-informative data which could make the important data hard to find.

5. Conclusions

Quality is a big concern in large software projects that take long time to develop. It has become clear that software metrics are an important factor in the modern software development business. Software metrics are not very widely studied and the needed metrics differ on each project and company.

5.1 What is needed

The software development department wants to ensure that the achieved software quality will not decrease during the many years of development. It takes time to understand poorly written and structured code. Refactoring can help to make the code base more understandable and maintainable and to ensure that complexity does not get out of control. This is a huge concern, as the software that is being developed is supposed to be supported many years, or even decades, to come. Static software analysis is one way to help developers and management to perceive a better and more clear picture of the code base. This possibly helps to highlight the most poorly structured code that potentially does the most harm when maintainability is concerned.

5.2 What is done

Six core software metrics that could be useful for analysis have been chosen to help analyze complexity and modularity of code base. The metrics are ambiguous in a way that the pure measurement numbers alone do not tell anything. Who ever is reading the analysis, needs a more profound understanding of the collected software metrics in order to not to make false conclusions. Moreover, the analysis has been implemented to the common

infrastructure of the software teams to make the analysis easily accessible. The analysis is scheduled to run daily to make the analysis always have the latest data and to be able to plot metric trends. Other software metrics and trend data is being analyzed as well but the usefulness of those metrics and especially the code smells is not certain and might just confuse developers. This thesis was written in three months, which is a small time for a big company. The usefulness of the chosen metrics and tools could not be tested in such a short period of time. This thesis has however started the usage of clearly structured static product metrics.

5.3   Actions in the future

In the future, the company can analyze output data of the made static software analysis and possibly decide common quality standards for software development. Many projects inside the software development department have similar structures and quality attribute preferences which helps the project teams to compare some of the metrics and, perhaps, imitate processes that are born with the static analysis. The next step could be to implement the static analysis for other development teams as well. This is made easier by making the scripts and Jenkins configurations easy to modify and to implement on different projects. However, The usefulness of the analysis must be tested and suitable metrics must be declared before any further actions can be done.

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.*

Banker, R. D., Datar, S. M., & Zweig, D. (1989). Software complexity and maintainability. In *Icis* (Vol. 89, pp. 247–255).

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

CASTsoftware. (2014). What is software complexity? `https://www.castsoftware.com/glossary/software-complexity`. (Accessed: 14.11.2018)

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.

Dhanvani, P. (2013). Difference between cohesion and coupling. `https://freefeast.info/difference-between/difference-between-cohesion-and-coupling-cohesion-vs-coupling/`. (Accessed: 26.11.2018)

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

Fowler, M. (2006). Codesmell. `https://martinfowler.com/bliki/CodeSmell.html`. (Accessed: 6.11.2018)

Gupta, V., & Chhabra, J. K. (2009). Package coupling measurement in object-oriented software. *Journal of computer science and technology*, *24*(2), 273–283.

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.

Stevens, W. P., Myers, G. J., & Constantine, L. L. (1999). Structured design. *IBM Systems Journal*, *38*(2/3), 231.

Thakur, D. (2014). 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).