Using Software Metrics to Better Understand Complexity Growth during Software Evolution

Olaf Haalstra
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
o.n.r.haalstra@student.utwente.nl

ABSTRACT
Many successful software projects evolve all the time. New features are added and bugs are fixed. All these continuous changes to software projects tend to increase the complexity. In this paper, we propose an approach to study complexity growth within software projects with aid of software metrics. We evaluated currently existing software metrics and their applicability on evolving software projects. We found that specific software metrics can be used to identify evolution categories of classes. This paper focuses on better understanding how software metrics can identify the cause of increasing complexity in software projects.

Keywords
Software Metrics, Software Evolution, Software Complexity

1. INTRODUCTION
As a software project evolves, it usually becomes harder to maintain. Each new version of the software introduces new features and bugfixes that often increase the complexity of the software. Introducing these new features or bugfixes might break down the modularity of the project. Over time all these changes can significantly increase the complexity of the project.

"Unchecked complexity growth is likely to become a deterrent of further evolution and may trigger the need for substantial software restructuring and refactoring"[3], meaning when nothing is done to reduce the continuously increasing complexity, the software will become more and more difficult to maintain.

In order to reduce complexity growth, the source of this growth must be identified. To this end, we will use software metrics. Software metrics have been commonly used to measure the complexity of software projects [9]. To better understand how software metrics can be used to identify the source of complexity, we studied which software metrics can effectively indicate complexity growth in the context of software project evolution. Then we assessed the evolution of complexity by using these metrics. Our study evaluated the complexity growth of two software projects. In this experiment, we identified the different causes of the complexity growth during these software projects' evolution. We were able to categorize these causes into multiple evolution categories that stereotype the complexity growth of classes within the software project.

The rest of the paper is organized as follows. In section 2 we discuss the identified software metrics and the software evolution analysis. In Section 3 we will describe how we obtained our results and in Section 4 we will describe our experiment and results. Section 5 discusses related work. The results will be discussed in section 6, we also give recommendations for future work in this section. Finally, we conclude in section 7.

1.1 Problem statement
The objective of this research is to determine how software metrics can be used to identify the causes of increasing complexity during a software project’s evolution. In order to be able to study the increasing complexity, first a way to accurately determine software complexity is defined. Then existing software projects are evaluated in order to see what type of changes can be identified. We attempt to better understand the applicability of software metrics to determine what causes increasing complexity within the selected projects. Finally, we want to do this in such a way that our approach can be repeated for any software project.

1.2 Research questions
The following research questions will be addressed, starting with the main question:

- How can software metrics be used to identify the cause of increasing complexity in a software project’s evolution?

In order to answer the main questions it has been divided into three subquestions:

1. What software metrics can effectively indicate complexity growth in the context of software evolution?
2. What type of changes in the software that results in increased complexity can be identified and how?
3. How can our approach be automated to evaluate any software project?

2. BACKGROUND AND TERMINOLOGY
To determine which software metrics can effectively indicate complexity growth in the context of software evolution background information about software metrics and software evolution was collected first.
2.1 Software metrics

In the article of Varela [12], a mapping study was conducted. Almost 300 source code metrics were identified in this study. Based on the categories and descriptions in their work we identified several metrics that can effectively indicate complexity growth for software evolution. Additionally, we will describe the number of occurrences the indicated software metric has across all of the mapped papers by Varela.

2.1.1 LOC - Lines Of Code

The first metric is the most trivial one. It counts the lines of code for every method. The LOC ignores comments and counts actual statements. This way lines of code that are split are counted as one. LOC is a popular metric, with 79 occurrences in the study of Varela [12]. We use this metric in order to be able to compare software projects and put complexity growth into perspective with project size.

2.1.2 Cyclomatic complexity

Cyclomatic complexity is a software metric to indicate the complexity of a method. It measures the number of linearly-independent paths through the source code [11]. It is a popular metric according to the article of Varela, with 55 occurrences [12]. This metric is very suitable for our research since it directly describes the complexity of the project.

2.1.3 WMC - Weighted Methods per Class

The WMC measures the complexity of an individual class by summing the weight of each method [6]. To measure the weight of each method we used the cyclomatic complexity metric, this approach has been used in the context of studying software evolution with software metrics [8]. WMC is the most popular metric within the papers mapped by Varela, with 89 occurrences [12]. WMC with cyclomatic complexity is our most important metric because it has been so widely studied and has been used in the context of software evolution before.

2.1.4 LCOM - Lack of Cohesion in Methods

Another popular choice within the work of Varela with 86 occurrences [12] is LCOM. This metric refers to how closely the operations in a class are related to each other. The larger the number of similar methods, the more cohesive the class [6]. The lack of this cohesion causes greater rework and greater design effort [5] and is an indicator of complexity [7]. We are interested in this metric because of the indication of complexity. However, this metric has been discarded at a later stage motivated in section 3.1.3.

2.1.5 NOC - Depth Number of Children of Class

The NOC is defined as the number of immediate subclasses of each class. Classes with a large number of children are considered to be more complex and fault-prone [7]. NOC is also a popular choice in the work of Varela with 77 occurrence [12]. This metric was selected for its ability to flag complex classes but was also discarded at a later stage motivated in section 3.1.3.

2.1.6 Cognitive complexity

Cognitive complexity is based upon the principles of cyclomatic complexity but has been formulated to be more accurate to assess the understandability of code. Cognitive complexity counts the number of breaks in the linear flow of the code. When flow-breaking structures are nested the count increments for each level of nesting [2]. Campbell states that cognitive complexity is superior for its distinction between a domain class, one with a large number of simple getters and setters, and one that contains a complex control flow where cyclomatic complexity fails to distinguish those. The term cognitive complexity can also be found in the work of Varela, however, those metrics are all calculated differently. This specific cognitive complexity metric can only be found in the technical report of SonarSource 1. Nevertheless, this metric is interesting for our research because of the claimed superiority to cyclomatic complexity but was discarded at a later stage motivated in section 3.1.3.

2.2 Software evolution

In the last decade, the evolution of software projects has been extensively studied [4]. The foundations for this area have been laid by M. M. Lehman who defined the Lehman’s laws of software evolution. We are particularly interested in rule 2: "As an evolving program is continuously changed, its complexity increases unless work is done to maintain it or reduce it.” [10].

Interesting insights related to better understanding software evolution with aid of software metrics are found in the work of Lanza [9]. They categorize the different behavior of a class in an evolving software project. To identify the types of changes in the software that contribute to complexity, we distinguish between the following evolution categories:

- **Supernova**
  A supernova is a class which suddenly explodes in size. Common reasons for this are:
  - Major refactorings of the system which have caused a massive shift of functionality towards a class.
  - Data holder classes which mainly define attributes whose values can be accessed. Due to the simple structure of such classes, it is easy to make such a class grow rapidly.
  - Sleeper classes. A class that has been defined a long time ago but is waiting to be filled with functionality. Once the moment comes the developers may already be certain about the functionality to be introduced and do so in a short time.

  Supernova classes should be examined closely as their accelerated growth rate may be a sign of unclean design or introduce new bugs into the system.

- **Pulsar**
  A pulsar class grows and shrinks repeatedly during its lifetime. The growth phases are due to additions of functionality while shrinking phases are mostly probably due to refactorings and restructurings of the methods in the class. Note that a refactoring might also make a class grow, for example when a long method is broken into many shorter methods. In that case, the WMC stays the same.

- **Red Giant**
  A red giant class can be seen as a permanent god class, which over several versions keeps on being very large. God classes tend to implement too much functionality and are quite difficult to refactor.

3. AUTOMATED EXTRACTION

To better understand how metrics can be used to identify the source of complexity growth within evolving software projects, we want to analyze existing software projects. Because of the iterative nature of these projects, we want to extract information about each iteration automatically. We chose to analyze projects from GitHub since this is a very popular hosting service for Git repositories. Git is a very popular option for handling software projects and PMD has the functionality to exactly write file;  

Algorithm 1: The cyclomatic complexity is calculated for every method. And summed for the WMC of every class. Done for every version

We chose to use Git tags since they are named by semantic versioning. Meaning each tag will at least incorporate bugfixes, added functionality or major project changes. The data from the file contains the classes’ WMC of every version in a comma separated file. We can open it with spreadsheet programs in order to create graphs and analyze the data.

4. METRICS EXPERIMENTS

After executing the experiment as described in section 3.2, we analyzed the data. In order to better understand how software metrics can be used to identify the source of complexity, we are looking for the evolution categories we described in section 2.2. These categories give the data meaning: complexity growth can be related to particular files and these files can be marked as a subject for further inspection.

In our experiment, we looked at two software projects. We studied OkHttp and Spring Framework. In order to make a fair comparison we selected them on LOC, these projects currently have a fairly similar amount of LOC: 8000 and 7000 respectively. In the graphs we show the Total WMC, meaning the WMC summed for each class in the project. The growth of LOC and Total WMC of the Spring Framework can be seen in figure 1. The second rule of Lehman states that the complexity of the continuously changing project increases, unless work is done to maintain or reduce it. The reduction of complexity can be contributed change in the LOC, the correlation between Total WMC and LOC, in this case, is almost 100%.

3.1 Tooling

In order to analyze the extracted data with Git, we looked into tools that were suitable for analyzing software projects with the use of software metrics. We came across SonarQube and PMD.

3.1.1 SonarQube

SonarQube analyses software projects, measuring quality and providing reports. Enabling developers to find styling errors, potential bugs, code defects, design inefficiencies, code duplication, lack of test coverage, and excess complexity. For our research, we are interested in the possibility to find excess complexity.

3.1.2 PMD

We discovered that SonarQube uses PMD as a plugin in order to collect software project data with software metrics. PMD is a source code analyzer, it finds common programming flaws. It uses rule-sets to define when a piece of code is erroneous, for our research we are interested in the 'Code Size Rules', containing LOC and cyclomatic complexity.

3.1.3 Tooling discussion

We noted that SonarQube and PMD do not support any software metrics that we found in section 2.1, apart from cyclomatic complexity, cognitive complexity, and LOC. The developer team of SonarQube decided to remove the support for the LCOM metric because they found it was difficult to compute it correctly and therefore difficult to use it correctly, they advise against using this metric to analyze your software project.

Further, it has been stated by Aniche that reporting problematic classes with the aid of software metrics is not a straightforward task. Because of the lack of support and the difficulty to utilize multiple software metrics in the context of software project’s complexity, we decided to execute our experiments with only the aid of the LOC and the cyclomatic complexity software metrics.

Cognitive complexity has been discarded because SonarQube is not informative enough for our research. When diving deeper into the SonarQube tool we discovered that SonarQube is very good at analyzing the current state of the code. It shows some history but lacks the functionality of analyzing previous iterations.

PMD is a command line tool that analyzes the current state of the project. PMD has the functionality to exactly specify which information you want to be extracted. This is why we chose to use PMD to execute our study, in order to be able to extract the information from evolving software projects automatically we created an algorithm.

$$\text{if startTag < tag then} \quad \begin{cases} \text{break; } & \text{for finalTag == tag then} \end{cases}$$

$$\text{end}$$

$$\text{run PMD on directory; for class in PMD.Classes do} \quad \begin{cases} \text{WMC = 0; for method in class.Methods do} \end{cases}$$

$$\text{WMC += method.cyclomaticComplexity;} \quad \text{end}$$

$$\text{file += class, WMC;} \quad \text{end}$$

$$\text{if finalTag == tag then} \quad \begin{cases} \text{break; } & \text{write file;} \end{cases}$$

$$\text{end}$$

$$\text{Algorithm 1: The cyclomatic complexity is calculated for every method. And summed for the WMC of every class. Done for every version}$$

3.2 Method

The algorithm we used first extracts that project from GitHub using Git. The cyclomatic complexity of each method is measured, in order to calculate the WMC of each class. We used PMD, the extraction method can be found in algorithm 1.

$$\text{Result: Cyclomatic complexity of every class}$$

$$\text{git project; for tag in git.Tags do} \quad \begin{cases} \text{if startTag < tag then} \quad \begin{cases} \text{continue; } & \text{run PMD on directory; for class in PMD.Classes do} \end{cases} \end{cases}$$

$$\text{WMC = 0; for method in class.Methods do} \quad \begin{cases} \text{WMC += method.cyclomaticComplexity;} \end{cases}$$

$$\text{end}$$

$$\text{file += class, WMC;} \quad \text{end}$$

$$\text{if finalTag == tag then} \quad \begin{cases} \text{break; } & \text{write file;} \end{cases}$$

$$\text{end}$$

$$\text{Algorithm 1: The cyclomatic complexity is calculated for every method. And summed for the WMC of every class. Done for every version}$$

$$\text{We chose to use Git tags since they are named by semantic versioning. Meaning each tag will at least incorporate bugfixes, added functionality or major project changes. The data from the file contains the classes’ WMC of every version in a comma separated file. We can open it with spreadsheet programs in order to create graphs and analyze the data.}$$
In section 2.2 we described the class evolution categories of Lanza [9] they described pulsars, supernovas, and red giants. We found these evolution categories of classes occurring in the metric experiments:

4.1 Supernova

We evaluated the total increase of complexity each class is responsible for throughout the history of the project. This example shows the four classes that increased most in complexity. In figure 2 can be seen that the complexity rises quickly. The explosion in complexity that can be seen in these classes are also seen in the project total complexity, see figure 1. These files have a huge impact on the overall complexity of the project. We categorize these classes as supernovas.

These classes should be examined closely as their accelerated growth rate may be a sign of unclean design or introduce new bugs into the system [9].

4.2 Pulsar

We tracked the 10 classes with the highest complexities throughout the history of the project. The complexities of these classes were accumulated and compared against the project Total WMC. The black line in figure 4 shows the percentage between the accumulated complexity of these classes and the Total WMC. It starts at 40% and eventually declines to 30%, while these classes only make up for about 5% of the total number of classes. This shows that these classes are responsible for a significant amount of the project Total WMC. We categorize these classes as red giants.

Red giants are probably caused by a large number of complex methods. Lanza stated that red giants tend to implement too much functionality and are quite difficult to refactor [9]. Chidamber stated that classes with large numbers of methods are likely to be more application specific, limiting the possibility of reuse. It also indicates that more time and effort is required to maintain these red giants [6]. This is why it is important that the software project is kept modular and the functionality of red giants are separated among other classes.

5. RELATED WORK

Software evolution has been long studied. M. M. Lehman published a study on software evolution in 1980 defining the laws of software evolution [10]. Different authors have studied the software evolution with aid of software metrics. Lanza created a tool in order to visualize the complexity growth in software evolution. This resulted in defining categories of the evolutionary behavior of classes [9]. Our approach uses these categories in order to identify problematic classes. Lanza did not use metrics specifically meant for indicating software complexity. Our research selected software metrics that can effectively indicate software complexity [6] [7] [11] [12].

Capiluppi analyzed open source software projects to observe their phases of evolution, growth, and maturity. Their
aim was to understand how code changes are achieved, what is their impact on code architecture, and how process issues are related to structural modifications [3]. Capiluppi evaluated only the size of the project, by looking at the LOC, the number of folders, the number of files per folder, and compared the ratio between all of these metrics. They researched how the increased size of the project impacted the architecture. Our research focuses on the source of growth on a more detailed level, finding evolution categories and identifying individual classes.

Johari analyzed two open source software projects in order to measure their change over time with software metrics. Johari did use metrics that are also relevant in our research, cyclomatic complexity [11], and metrics proposed by Chidamber and Kemerer [6]. They used LOC and WMC, methods weighted with cyclomatic complexity, the same as in our paper. Furthermore, they also analyzed the projects with LCOM and NOC, as in our proposed metrics, and others. They found that Lehman’s laws are related to increasing complexity and continuous growth are supported by the data and computed metrics measure [8]. The research of Johari is an empirical study in order to confirm Lehman’s Laws [10]. Our research also studies the evolution of the project with software metrics but is focused solely on complexity. By combining the evolving categories proposed by Lanza, this enriches the understanding how metrics can be used to identify the source of increasing complexity.

Chaikalis evaluated the quality of software libraries by analyzing their evolution with software metrics, they used LCOM, WMC, as in our proposed metrics, and others. With these metrics, a trend over time was created for each library. They found that software libraries can be considered as stable software projects in terms of quality, their quality did not degrade over time [4]. Chaikalis only evaluates the trend of the evolution of software libraries obtained with software metrics from start to end. This trend implies that the software has either improving, neutral, or deteriorating quality over time. In our research, we provide a more detailed overview of the trends seen in software projects and give meaning to these trends with the aid of evolution categories.

6. DISCUSSION AND FUTURE WORK
The results show that we are able to identify the evolution categories from Lanza. However, this paper reflects only the particular software projects we analyzed. To be able to draw more solid conclusions a greater variety of software project must be analyzed. In this paper, we proposed an approach to study evolving software projects.

An improvement is to increase the number of software metrics we currently use, there are many other metrics that can be evaluated in the context of software complexity [12]. Our hypothesis is that by using more software metrics, this may lead to more detailed results narrowing down the number of problematic classes.

We already indicated more software metrics that can effectively indicate complexity growth in the context of software evolution in section 2.1. These software metrics must first be implemented within PMD or another tool must be found. More metrics also means more collected data. Before evolving software projects can be analyzed with more software metrics further research is required.

The pulsar category classes grow and shrink repeatedly. This is due refactoring and restructuring the class. Noted that refactoring might make a class grow when it is broken into smaller methods. This is wanted behavior in order to make the software project more modular. In our research, we found that we can indicate the zig-zag figure these pulsars create in the WMC and LOC, see figure 3. However, we want to be able to see if these refactorings positively impact complexity.

Currently breaking methods into smaller pieces does not reduce the cyclomatic complexity, thus does not impact the WMC. If a threshold would be defined, no longer counting low complexity methods, the WMC would shrink when methods are successfully broken down into smaller methods reducing complexity but increasing the LOC. This can then be seen in the WMC and LOC graph as a pulsating LOC but declining WMC, a successful refactor of methods. However it has been stated by Aniche that defining this threshold is not a straightforward task [1], therefore would require more research.

Another approach would be to replace the cyclomatic complexity by cognitive complexity, see section 2.1.6, in order to calculate the complexity of each method. This is likely to be a more accurate measure of complexity since it counts heavier towards nesting and complex structures [2]. If this is the case this could also indicate the successful refactor of methods.

If more software project were to be analyzed with the proposed improvements, this would allow for identifying good and bad trends. With the final goal to be able to compare new software projects against studied trends in order have a better understanding of the source of complexity and how to reduce the complexity growth in these new software projects.

7. CONCLUSIONS
In this paper, we presented the results of a study aiming at the analysis of the evolution of software projects and how to perform this automatically. We aimed to better understand how software metrics can identify the cause of increasing complexity in a software project’s evolution and found the first clues towards this.

First, we studied software metrics. By studying literature a list of software metrics that can effectively indicate complexity within evolving software projects has been described in section 2.1. Then, in section 3.1.3, we motivated why certain software metrics were not considered for our research. We arrived at using LOC and WMC for our research. LOC is used to put complexity growth into perspective with the size of the evolving project. WMC is used for its ability to effectively indicate complexity growth during the evolution of the software project.

Secondly, we were able to identify the type of changes in the software that resulted in the increased complexity. These changes are exploding classes, classes that grow and shrink repeatedly and very complex classes, so called evolution categories. We described these categories in more detail in section 2.2. The different evolution categories all contribute to the complexity growth in different ways, leaving iconic patterns in the data trends. This is how we were able to identify these categories in the data we extracted from two software projects. Where these categories can be found, there are often interesting parts of the software project. These parts can be examined more closely by the developers, because of their impact on the complexity of the project.

Finally, we arrive at our approach. In this paper, we described how the evolution of a software project on GitHub can be automatically extracted, see section 3.2. The data
that is collected this way contains the information about LOC and Total WMC for each version. This results in a graph that can be evaluated as seen in section 4. This approach can be done for any software project that is available on GitHub.

We would like to conclude by stating that we think software metrics can assist in identifying the cause of increasing complexity in a software project’s evolution by creating trends of currently existing software projects. These trends can give useful insights to developers towards the cause of complexity by identifying evolution categories. When developers know which particular classes cause a strong increase of complexity, greatly impacting the software project’s complexity. These classes can be refactored, redesigned or split into other classes in order to keep the project modular and maintainable. The contribution of our work is that we have shown that using software metrics can indeed be used in order to identify evolution categories, that in turn assists in identifying the cause of complexity growth.

8. REFERENCES


