Visualizing Software Metrics for increased Refactoring

Paul Steneram Bibby
ada08pst@student.lu.se

Fredrik Palmquist
dat11fpa@student.lu.se

March 7, 2016

Keywords: Agile Development, Code Complexity, Refactoring

Abstract

Refactoring is a practice which is supposed to reduce code complexity in software, but lack of motivation or knowledge can cause developers to skip it in favor of "more important" tasks. Code complexity is a problem in any large scale software development and the goal of this study is to see if a visualization of code complexity can inspire developers to refactor more. The main code complexity metric we'll be focusing on will be the cyclomatic complexity, and the goal will be to keep it as low as possible. At the end of the course *EDA260 Software Development in Teams*, the team this study was performed on had the second lowest cyclomatic complexity of all teams in the course by a slight margin.

1 Introduction

Refactoring is a practice which is supposed to reduce code complexity in software, but lack of motivation or knowledge can cause developers to skip it in favor of "more important" tasks. An attitude of "If it's not broken, don't fix it", where they don't see the need to refactor code if it's working as intended, is not unheard of either.

Code complexity is a problem in any large scale software development. Changes made upon bug fixes made upon other changes can cause the code to deteriorate over time. Although high complexity in code that is rarely changed might not be a problem, it can cost a lot of time in areas where changes are made frequently.

The goal of this study is to see if a visualization of code complexity can inspire developers to refactor more. Measuring code coverage during testing can motivate developers to write more tests in order to increase the coverage. Not only can it be satisfactory to see the coverage percentage increase, it can also point out what parts of the code has not yet been covered by tests. We would like to extends this idea to code complexity, where the visualization can help indicate problem areas in the code, and motivate the developers to try and keep it as low as possible by refactoring the problem areas.

This study is done during the course *EDA270 Coaching of Programming Teams* which runs in parallel with the course *EDA260 Software Development in*

Teams at LTH. Teams of IT students will be tasked with developing a timing system for a motor bike race called Enduro. The code will be written in Java, and developed in the environment called Eclipse. Eclipse is easily customizable by the use of downloadable plugins, and the developers in our team will be instructed to download and install a plugin called CodeCity, which will be used to visualize the complexity in the code. We will then compare the complexity in our code with the complexity in the other teams code to see if the tool has helped keep the complexity down.

In the Background section, we explain the complexity metrics measured by our tool, refactoring as well as our hypothesis. The section Methods describes the tool we've chosen, followed by the Results section where we document our finding. After that we discuss and reflect on our results in the Discussion section, followed by a Conclusion where we summarize our findings. Lastly there is also a short section about recommendations for future research.

2 Background

In this section we will discuss the background for our study. First we will explain what the Enduro project is followed by a section about code complexity. We will also go into more details about the three code complexity metrics we've chosen to measure using our tool. There will also be a section about refactoring, and finally a description about our hypothesis.

2.1 The Enduro Project

Each year at LTH the course EDA260 aims to teach software development in teams. Undergraduate students are put together in teams of roughly eight(8) to twelve(12) developers. They are given the task of developing a time measuring program for the dirt bike race known as Enduro. During this time they will have at least three(3) releases, but most teams will do four(4). Each team are given two(2) senior students that will act as coaches to help them during their six(6) weeks long development cycle. The senior students are taking a course in coaching for software development in a team, EDA270.

Each week consists of a two hour planning session and an eight hour development session. Besides the scheduled time the developers also have four hours of spike time, two of which should be dedicated to an XP focus and the other two hours should be used for something beneficial to the team.

During the project the team will have a contact person that act as a customer for the project. The customers are fictive and acted by various professors at the faculty of computer science at LTH.

2.2 Code Complexity

We had initially planned to use the CK Metric for measuring the code complexity of our code. The CK metric was developed by Shyam Chidamber and Chris Kemerer in 1993, and is a suit of six different metrics [6]. It takes into consideration number of methods in a class, the depth of an inheritance tree, the number of children of a class, the coupling between objects, the response for

a class, and the cohesion between methods. Details about the CK metric can be read in Section 8.1 in the Appendix.

We found the CK metrics attractive due to it being able to measure different forms of complexities. Complexity does not take one single form that can be solved by measuring one metric. However we had to scrap this idea due us not being able to find any plugin with the ability to measure the CK metric that was compatible with the version of Eclipse found on the computers at LTH. Instead we chose to measure the cyclomatic complexity, duplicated code and lines of code per class.

2.2.1 Cyclomatic Complexity

The cyclomatic complexity is the number of linearly independent paths through a function. An if-statement introduces two possible paths: one where the predicate is true, and one where the predicate is false. Cyclomatic complexity are usually only used to measure the complexity of functions. Using it on a whole program is unfeasible as there could be thousands upon thousands of independent paths through the code.

To demonstrate we'll give a quick example. In Figure 1 you can see a flow graph for a function. You could imagine it's a function that prints a header based on a boolean, and then iterates through a list of drivers. Calculating it's cyclomatic complexity is easy. The formula is as such:

$$
\begin{aligned}
E &= \text{number of edges} \\
N &= \text{number of nodes} \\
M &= E - N + 2
\end{aligned}
$$

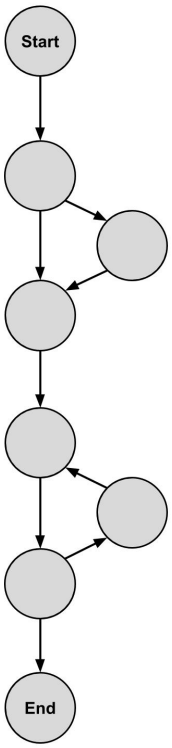

Figure 1: Function graph

The variable M represents the cyclomatic complexity. In the graph in Figure 1 we count the number of edges to nine(9), and the number of nodes to eight(8). Using the formula we can calculate the cyclomatic complexity to:

$$M = 9 - 8 + 2 = 3$$

Which means that there are three(3) independent paths through the graph. This value can also be used to determine the *minimum* amount of tests needed to fully test the function. However this is not part of the study.

2.2.2 Duplicated Code

Duplicated code is when a sequence of code appears multiple times. This is usually caused by developers copying and pasting code from one place to another due to laziness or bad code practice. Duplicated code should be avoided as it causes double maintenance; changes made to one of the copies will have to be applied to the others, and they need to be applied exactly the same way.

2.2.3 Lines of Code

Lines of code is simply the number of lines of code in a class. Huge classes does not necessarily have to mean it's complex, however avoiding large classes is usually recommended.

2.3 Refactoring

"Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure. It is a disciplined way to clean up code that minimizes the chances of introducing bugs. In essence when you refactor you are improving the design of the code after it has been written." [3]

If the team remembers to do small refactorings before and after each single task, the need for major refactorings is often non-existent in PVG projects. However, very few (no?) teams do that, so after a couple of iterations the code is so "patchy" that a major clean-up is need to be able to move ahead.

Refactoring aims to bring order from chaos. A lot of projects stop being engineering jobs and start being hacks. If this happens that is usually a very good indication that your project is in need of a major refactoring.

2.4 Hypothesis

Code complexity and refactoring are closely associated with each other. The need for refactoring is always present but becomes more and more relevant the more complex your program becomes. We will investigate if by visualizing the complexity for the PVG-team will encourage small refactoring, to keep the complexity low and thereby eliminate the need to big refactoring altogether.

3 Method

To visualize different software metrics to the team we went with a plugin for eclipse called CodeCity. Each developer have access to produce and analyze the graphs at any time. But during each iteration we have a stand up meeting to discuss the graphs together with the whole team and arrive at conclusions about what possible problem areas are, and where they need to look out for code that is in need of refactoring.

After each release, the code produced by all the nine(9) teams was downloaded to be used for analysis. This gives us three(3) data points for each project to use when comparing the projects software metrics to each other.

For refactoring the teams was encouraged to read a book on refactoring [3]. They recently went through a course which aimed to teach them proper

software patterns. So the book was aimed as a reminder about how to apply that knowledge.

3.1 Tool

The goal of our study was to see if the use of visualized code complexity metrics would encourage the developers to refactor in order to reduce said complexity. The developers in our team were using Eclipse, which is a development environment for Java which can easily be customized by the use of plugins. We wanted to find a plugin that could calculate the complexity for classes and methods and then display it for the developers. There are several different tools that can be downloaded as plugins to Eclipse. For most of them, calculating code complexity is only a part of it's full functionality. One will have to be chosen to be used during the EDA260 course, as using more than one feels excessive. Comparing the tools aren't part of this deep study.

We initially chose to use a plugin called Eclipse Metrics which was supposed to calculate various metrics for the code and print warnings in the Problems View in Eclipse. Unfortunately we were unable to successfully install it because of the version of Eclipse available at LTH:s computers. Instead we chose to use a plugin called CodeCity.

3.1.1 Codecity

CodeCity is an integrated environment for software analysis developed by Richard Wettel and is limited to non-commercial use [8]. We chose CodeCity because it was able to visualize different metrics in one simple graph. We thought if it was kept simple it would be easier to motivate the developers to use it. If the it was required to first create a release, then run it through some external software and then visualize the results by some other tool, it is unlikely anyone would have been bothered to do it at all. It's hard enough to get them to run the unit tests before pushing, if measuring the code complexity required that many steps it would have been skipped for more important things, such as more stories.

CodeCity visualizes metrics by the use of bars, see Figure 2 for Team 08's third release. It's not hard to figure out why it's called Code*city*. These bars are able to show three types of metrics at the same time by their width, height and color, and there are several different options available to select from. The obvious choice for us was cyclomatic complexity, as it's a fairly good metric for measuring code complexity. However, selecting the other two options were not as obvious. The list was sadly lacking most, if not all, of the metrics from the CK metric suit. Some options didn't really feel relevant or important, and some of them we didn't understand. Our goal was also to keep it simple, so researching and then explaining some of the metrics to the developers didn't feel like a good idea. In the end we chose *duplicated code*, as it was simple to understand and easy to correct, and finally *lines of code* as avoiding large classes is also a simple way of reducing complexity. See Table 3.1.1.

Drawbacks to this plugin is that it requires the developers to actively generate the graphs and look at them. Compared to the JUnit suit were you get instant feedback after the push of a button this requires more thought and intent from the developers.

Height	Lines of Code
Width	Dublicated Code
Color	Cyclomatic Complexity

Table 1: Codecity metrics.

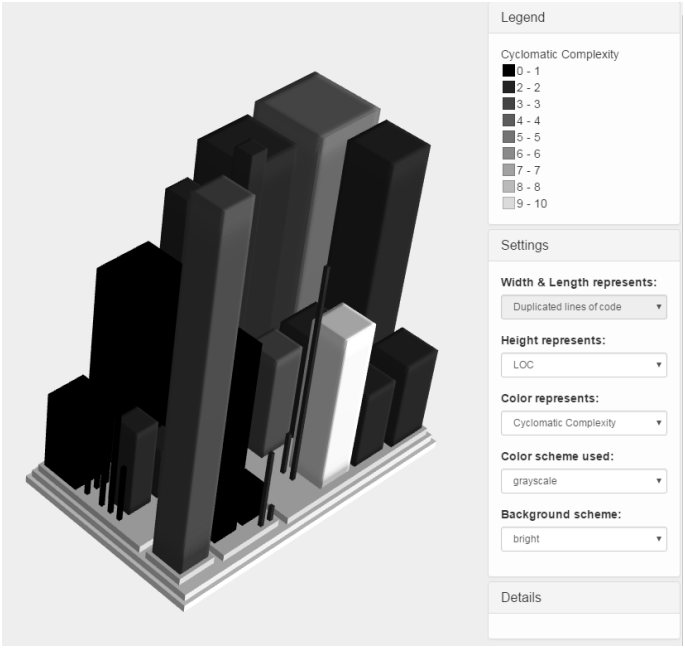

Figure 2: CodeCity for Team 08.

Team	Release 1b	Release 2	Release 3
1	14	14	17
2	13	44	60
3	8	19	10
4	No Ant	14	17
5	Won't compile	17	15
6	8	21	17
7	12	33	44
8	4	13	11
9	5	38	40

Table 2: Highest cyclomatic complexity for each team.

4 Results

Here we present the results from our studies. Data was collected from all teams: cyclomatic complexity was measured using a program called cyvis, and a form about how many stories were implemented and what architecture they used was sent out and answered by the most of the team coaches.

4.1 Cyclomatic Complexity

Table 2 shows the highest cyclomatic complexity for each of the teams during this course. To calculate the cyclomatic complexity a program called cyvis [9] was used. Each release was downloaded and the complexity calculated and then put into the table. Test classes were excluded in the check. A few of the programs were excluded in the calculations due to either not compiling or there being no build script available for the program. This was however only a problem during the first release, after that all programs could be built without any problems.

Table 3 shows number of classes with cyclomatic complexity >= 7 divided by the total number of classes in the program. Test classes were excluded during the count. Since a low or high complexity is not the only thing that shows how complex a program is. If a lot of classes with semi high complexity is spread out in the program, it might be just as hard to understand as a program with one class with super high complexity. Due to the realization that this metric could be useful first after the first release we don't have any metrics to present for that release.

Some noteworthy cases are team 2, 7 and 9. Their complexity seem to indicate that they have implemented some of the later stories such as the network solution. Since the developers have not yet read any courses in real time programming there solutions are probably more complex then they needed to be.

5 Discussion

By just looking at the numbers the team that used a tool to keep their complexity down managed to keep it at a reasonable level. They were not the team

Team	Release 1b	Release 2	Release 3
1	-	1/15	8/36
2	-	4/20	6/19
3	-	2/35	3/35
4	-	8/22	10/28
5	-	5/21	7/24
6	-	8/20	8/23
7	-	6/23	7/23
8	-	1/18	3/18
9	-	6/19	6/20

Table 3: # of classes with CC >= 7 divided by the total # of classes.

with the lowest cyclomatic complexity. To our knowledge other teams did not use any program to check the complexity of their programs but this might very well be the case that they did through spike time or during each iteration.

We observed that the team did not use the tool as much as we had anticipated that they would. Generating the graphs was a time consuming and repetitive job, so if we had adhered to the XP principles we should have automated this task a lot more then it was and thus the developers might have been more inclined to use the tool.

The numbers would suggest that the tool was useful and helped the developers to keep the program less complex. However it's worth noting that cyclomatic complexity does not take into account all areas of well written code. Even if the complexity is low the algorithms could still be poorly constructed or bad naming conventions could also keep the code from being easy to understand or modify. Which would be cause for major refactoring in the future.

The team had to go through one major refactoring of the code. This seems to be the cause of two factors. One we introduced the tool fairly late to not clog down the developers with to much new information in the beginning. The second reason it was only during the end of the second iteration that all developers started to have a firm grasp of what the program was actually supposed to do and what its intended purpose was. which caused the programs architecture to change direction a few times without actually refactoring the code.

It's also hard to asses what changes were done as a result of them using the tool. Due to developer inexperience the architecture of the first version of the program was very dubious. This inexperience might also have prevented them from effectively identifying problem areas of the code. As well as not being able to identify the correct way to refactor this problems when found. A bad code smell that was very prevalent during most of the iterations was shotgun surgery, it was only after a lot of spike time on refactoring that the team successfully found the code smell and managed to correct it through refactoring.

6 Conclusion

Visualizing code complexity appears to motivate developers in trying to reduce it. Our team had the second lowest cyclomatic complexity of all teams, only Team 3 had a lower score but only with one point. It is unclear whether they

also visualized the code complexity in some way.

CodeCity was able to create an easy to understand graph of the code complexity for the whole system. However, because it required effort to use it was probably not used to the extent that we'd hoped. Concern for the code complexity was not as high as the concern for implementing more stores, and most developers felt it more important to start on a new task than worry about code complexity.

7 Future Research

This research could be conducted again put from a longer time perspective to get a more clear result if helps with cyclomatic complexity. Due to developers inexperience with developing software it's hard to gain evidence in such a short amount of time. It would also be interesting to see if it possible to add hooks to Git that prevents complex code and duplicated code from being pushed to the repository. This leaves the developers no choice but to refactor the code in order to commit their changes.

References

[1] A. Watson, T. McCabe, (1996), *Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric*

[2] K. Stroggylos, D. Spinellis, (2007), *Refactoring – Does it improve software quality?*

[3] M. Fowler, (2002), *Refactoring, Improving the Design of Existing Code*

[4] M. C. Robert, (2003), *Agile Software Development - Principles, Patterns, and Practices*

[5] R. Subramanyam, M. S. Krishnan, (2003), *Empirical analysis of CK metrics for object-oriented design complexity: implications for software defects*

[6] S. Chidamber, C. Kemerer, (1993), *A Metrics Suite For Object Oriented Design*

[7] S. A. Mohammed, T. M. Yahya, H. A. Wegdan, A. N. Mohammed, Y. Jararweh, (2015), *Accumulated Cognitive Approach to Measure Software Complexity*

[8] R. Wettel, (2015), *CodeCity*, `http://wettel.github.io/`

[9] Leepoint, (2007), *Java: Computing Cyclomatic Complexity* `http://www.leepoint.net/principles_and_practices/complexity/complexity-java-method.html`

8 Appendix

8.1 CK Metric

The CK metric is a metric suit which was developed by Shyam Chidamber and Chris Kemerer. The suit consists of six classes of metrics: WMC, DIT, NOC, CBO, RFC and LCOM1 [6].

WMC Weighted Methods per Class

DIT Depth of Inheritance Tree

NOC Number of Children

CBO Coupling Between Objects

RFC Response For a Class

LCOM Lack of Cohesion in Methods

Weighted Methods per Class

The WMC is calculated by multiplying the individual complexity of every method in a class. If all the method complexities are unity, then the WMC is equal to the number of methods in the class [6].

Depth of Inheritance Tree

The DIT is calculated by measuring the depth of inheritance of a class. If the class has multiple classes extending it, then the DIT takes the value of the longest line of inheritance [6].

Number of Children

The NOC is calculated by counting the number immediate sub-classes to a class [6]. It differs from DIT as it measures the breadth of the inheritance tree.

Coupling Between Objects

The CBO is calculated by counting the number for classes a class is coupled with [6]. A class is considered coupled with another class if it uses one of it's methods or attributes.

Response for a Class

The RFC is measured by counting the number of methods in the class and the number of remote methods directly called by methods of said class [6].

Lack of Cohesion in Methods

The LCOM is calculated by counting the number of pairs of methods in a class that don't have any fields in common [6].