Metrics to Understand Future Maintenance Effort Required of a Complicated Source Code

Michael Dorin
mike.dorin@stthomas.edu / Universität Würzburg. Würzburg, Germany

Sergio Montenegro
sergio.montenegro@uni-wuerzburg.de / Universität Würzburg. Germany

Recepción: 17-6-2019 / Aceptación: 9-7-2019

Abstract. An enduring engineering problem is the creation of a source code too complicated for humans to review and understand. A consequence of a complicated source code is that it requires more effort to be implemented and maintained. Exacerbating the problem is a lack of a proper understanding of exactly what the words "complicated" and "complex" mean, as the definitions of these words are often misconstrued. Some systems are indeed inherently complex, but this does not mean they must be complicated. In our research, several open-source projects were evaluated using software metrics to map the complicatedness of a source code with the ongoing effort to sustain the project. The results of our research show that a relationship exists between a complicated source code and the maintenance effort. It is clear that adhering to proper coding practices and avoiding a complicated code can result in a much more manageable future maintenance effort.

Keywords: analysis, metrics, complex, complicated, source code

Métricas para comprender el esfuerzo de mantenimiento futuro requerido de un código fuente complicado

Resumen. Un problema de ingeniería duradero es la creación de código fuente demasiado complicado para que los humanos lo revisen y comprendan. Una consecuencia del código fuente complicado es que requiere más esfuerzo para implementar y mantener. Lo que exacerba el problema es la falta de una comprensión adecuada de lo que significan exactamente las palabras "complicado" y "complejo", ya que estas definiciones a menudo se interpretan mal. Algunos sistemas son realmente intrínsecamente complejos, pero esto no significa que deban ser complicados. En nuestra investigación, se evaluaron varios proyectos de código abierto utilizando métricas de software para mapear la complejidad del código fuente con el esfuerzo continuo para mantener el proyecto. Los resultados de nuestra investigación muestran que existe una relación entre el código fuente complicado y el esfuerzo de mantenimiento. Está claro que adherirse a las prácticas de codificación adecuadas y evitar el código complicado puede resultar en un esfuerzo de mantenimiento futuro mucho más manejable.

Palabras clave: análisis, métricas, código fuente complejo, complicado

1. INTRODUCTION

Life on earth presents humans with some very complex and complicated concepts. It is easy for people to suppose that cancer and quantum mechanics are both complex and complicated, as they are defined like that by nature. The best humans can do is to make simplified model representations of complex physical phenomena. Software development is unique in the world of science and math, such that developers can choose if our work is going to be complicated.

Our research begins by examining and affirming the definitions of *complicated* and *complex*. Afterward, the consequences of *complicated* are measured with a thorough examination of the bug reports and the source code of prominent software projects. The analysis focused on the components of a *LAMP* server (Linux, Apache, MySQL, PHP). The results of this work show that when a source code becomes complicated, it requires more effort to sustain. By establishing the indicators of *complicated* and *complex*, a protocol can be defined to reduce the development of complicated software. As mentioned before, regardless of problem complexity, developers can choose whether their code will be complicated.

2. BACKGROUND

Complicated and Complex

The words *complicated* and *complex* are often misconstrued, and it is possible to have an understanding that the two words mean the same thing. This fact becomes evident when we review standard definitions of the words, as it can be challenging to notice the differences. For example, a standard definition of *complicated* indicates that something complicated is composed of interconnected parts and is difficult to analyze (Mish, 2004). Similarly, something complex is "composed of many interconnected parts; compound; composite" (Mish, 2004). These definitions almost sound identical.

The etymology of the word *complicated* shows the Latin root "plic," which means "to fold". The word *complex* has the Latin root "plex," which means "to weave". Something that is woven has many strands, but the strands can be well organized. When "fold" is used as a suffix, it means a "multiple". Something complicated can have fragments folded in, making it conceptually tricky to understand (Lissack, 1999).

Finally, when we consider whether something is complicated, we must acknowledge that, many times, something is complicated when it is not understood. Upon understanding a topic, it is often no longer complicated. With this in mind, consider that the number of people involved also plays a role in creating complicatedness. Having more people involved means that it is more likely that a single person will not understand it.

Git

The use of features found in Git was an essential part of this research. Git is a prevalent and heavily utilized version control system developed in 2005 by Linus Torvalds (Spinellis, 2012). Logs created by Git were used extensively and were examined using the Git log command. A python script was used to parse the logs which focused on Git commits. A Git commit means that a file was created or updated and stored in a local Git repository (Torek, 2019).

Static Analysis

Static analysis examines a program source code without actually executing the program. One of the first tools for performing static analysis was *Lint*, created in 1978 (Bardas, 2010). Static analysis has improved considerably since then and bugs flagged by the original Lint are now included in the compiler output. Many static analysis tools exist at this time, and the various tools specialize in different areas (Bardas, 2010). This study used the Cppcheck tool (Marjamäki, 2019) as it looks for multiple types of errors and is easy to run immediately without integrating it into a functioning build.

Measurements and Metrics

Attempting to understand complications and *complexity* is not a new concept, and several metrics have been invented throughout the decades to quantify these subjects. The first metric would merely be counting lines of code and reporting using units such as LOC (lines of code) and KLOC (thousands of lines of code). In 1976, Thomas McCabe developed the cyclomatic complexity metric, which is a measure of the number of linearly independent paths through a program (McCabe 1976). In 1977, Maurice Halstead published complexity metrics based on distinct operators, distinct operands, the total number of operators, and the total number of operands (Halstead, 1977). Using Halstead attributes, a value for difficulty and effort is made. Both Halstead and McCabe metrics have been both praised and criticized, but they are relatively simple and practical contributions to the software engineering community. More advanced metrics exist, but as this research is focused on identifying complexity and complicatedness, we have decided to use uncomplex and uncomplicated metrics as part of our work.

3. RELATED WORK

Existing Literature

Several research studies that consider the human aspects of a complicated source code have been conducted. Daniel Sturtevant has done an excellent work in his thesis for MIT describing the impact of software architectural design, and connecting it to employee productivity and staff turnover. He worked with a professional organization, and included a case study of a development project which shows that the structure of a software system does impact productivity and employee retention. Sturtevant points out that further research on different types of projects is warranted, and that there are many opportunities for this (Sturtevant, 2013).

Human considerations are further explored in the paper from Foucault et al. (2015) researching the impact that developer turnover has on software quality. This paper covers open-source software, and demonstrates a link between developer turnover and the number of bugs found in the project (Foucault, 2015). Interestingly, Foucault and Sturtevant do not entirely agree on the aspect of whether poorly designed software drives developers away or whether the turnover creates a poorly designed software.

The turnover of developers might also be a reason for code churn as described in the paper *Use of Relative Code Churn Measurements to Predict System Defect Density* (Nagappan. 2005). Code churn is a metric based on how much the code is changed, and this metric has been shown to be an indicator of faults. It is suggested that more recent code is faultier than the original code (Nagappan, 2015).

Other papers analyzed measurable aspects of software itself, rather than human influence. In the article *Reexamining the Fault Density - Component Size Connection*, Les Hatton discusses the relationship with the size of the program and the number of faults. Lines of code (LOC) has always been a good predictor of fault opportunities, and it is intuitively apparent why (Hatton, 1997). A four-line "Hello World" program will have fewer opportunities for faults than a four thousand-line program. Having more lines offers more opportunities for mistakes. Though LOC is a good indicator of bug opportunities, it also does not help you narrow down specific locations where faults may be present.

Previous Project Work

In a previous work on this project, we determined which programming practices software was the most difficult for engineers. We discovered what engineers misinterpret and what software engineers find displeasing to review. Also, a consistent unpleasant to review code had a higher Halstead and cyclomatic complexity. Stylistic constructs that programmers generally have difficulty evaluating are shown in Table 1 (Dorin, 2018).

Table 1
Unpleasant Styles (Dorin, 2018)

Style Name
Avoid too deep blocks
Do not write over 120 columns per line
Matching braces should be in the same column
Use less than five parameters
Use braces even for one statement
Indent blocks inside a function

For measuring project conformance with stylistic rules, the nsiqcppstyle tool (Yoon, 2014) was used. Using this previously developed knowledge in a complex and complicated code, we are now expanding the work to develop indicators to relate a complicated source code to this ongoing effort.

4. METHODOLOGY

Existing literature covers a lot of important subjects, but many papers seem to be specific to particular cases and specific environments. In this project, we reviewed several active open-source projects maintained by Git. Popular projects were preferred as they gave the most opportunities for users to report bugs. GitHub.com was used as a source for all the projects. The four cornerstone projects selected were Linux, Apache, MySQL, and PHP. The remaining projects were found based on a report of active projects on GitHub. Once projects were found, the following steps were performed:

1. *Calculation of thousands of lines of code (KLOC) for C, C++, and headers*. The amount of information given to a human to process is a factor in how much can be understood. For each project, for each source related file, we used the Lizard tool (Yin, 2019) to calculate the lines of code in both source files and headers.
2. *Calculation of Halstead for C, C++, and headers.* The Halstead metric, which evaluates complexity based on distinct operators, distinct operands, the total number of operators, and the total number of operands, was also used in this study. For each project, we used a program called commented code detector (Borowiec, 2014) to measure the Halstead metrics.

3. *Calculation of McCabe cyclomatic complexity for C, C++, and headers.* The McCabe cyclomatic complexity metric is based on the number of linearly independent paths through code. The cyclomatic complexity was used because more paths require more effort to understand. The Lizard tool (Yin, 2019) was also used to gather information on cyclomatic complexity for each project.
4. *Determination of "bug"-sized changes.* Several small utilities were written to analyze the logs maintained by Git. When searching for keywords, we determined the typical number of lines of code found in bugs and thus classified bugs by size. We classified tiny bugs as "gnats," as they are important for consideration since numerous small bugs take resources away from developing essential features. The gnat-sized bugs constitute 50% of the Git check-ins, which is a good indication of how bothersome they are.
5. *Measurement of conformance to stylistic rules*. From our previous work, we determined what was essential and what rules had the most impact on programmer understanding. As with our past projects, we used the nsiqcppstyle tool to measure stylistic rule conformance.
6. Measurement of basic coupling between modules. Though many utilities exist for measuring coupling, they were not practical for our use. For coupling, we created a straightforward metric which we called *Sheficom*. The Sheficom metric computes how many external modules are coupled using a count of the number of headers in the module. Sheficom is based on the assumption that, if a module is not coupled, it would not need to include a header.
7. *Measurement of the percentage of bug-related changes.* The goal of this effort was to measure how much work in a project was dedicated to bug fixes. By using the technique outlined in Item 4 to identify bugs, the percentage of bug-related changes was measured. We counted the aggregate number of complete bugs as well as the total number of lines changed for fixing bugs.
8. *Count of the number of authors.* As mentioned, one of the natures of being complicated is the absence of understanding. More programmers mean more people are required to understand the source code. Depending on the communication paths and capabilities of the authors, understanding could be impacted.
9. *Performance of static analysis of modules.* The Cppcheck tool (Marjamäki, 2019) was used to perform static analysis on the source code. This tool was chosen as it does not require performing any special actions or creating special configurations to analyze the source code, but it will provide an estimate of errors which exist in the project.

Table 2

Metric	Software Complexity	Human Complication
Size		X
Style		X
Sheficom (coupling)	X	
Author Turnover		X
Halstead	X	
Cyclomatic	X	

These metrics were used to get a picture of the effort needed for ongoing maintenance work. All of the metrics, in one way or another, contribute to impacting effort. However, some metrics have a more significant influence on what humans perceive as complicated (see Table 1). For example, in the new Sheficom metric, tightly coupled modules take a human longer to review and therefore more time to understand. It is plausible that a human, with enough time, could understand what is going on. At that point, though it remains complex, the code is no longer complicated. Author turnover was also essential to measure, as new authors require time to build up an understanding of the existing code.

5. RESULTS

Linux

Linux is a very well-established operating system and, for this study, we performed measurements every year for ten years, starting in 2008. See Table 3 and Figure 2 for a summary of the results. When examining the coding style, it seems to conform less year by year (see Figure 2). Improvements in module coupling and cyclomatic complexity are apparent. However, the size of the project grows linearly over time, and the number of static analysis detected bugs per snapshot also increases over time. More than forty percent of the effort to continue Linux seems to be changes related to bugs, as shown in Table 3. One more thing to bear in mind is that Linux has a lot of authors working on the project, which will contribute to the project being more complicated. It seems that a lot of authors and a lot of code lead to misunderstandings, and a lot of effort is spent fixing bugs rather than adding new functionality.

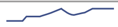
Metric	Total Change	2008	2009	2010	2011	2012	2013	2014	2015	2016	2017	2018
Style		43 %	43 %	43 %	42 %	42 %	41 %	40 %	40 %	40 %	40 %	39 %
Sheficom		12.55	12.36	12.6	12.78	12.58	12.42	12.26	12.16	12.1	12.15	11.94
Halstead		51.46	51.06	50.48	51.3	51.54	52.12	54.24	55.19	55.72	55.2	57.96
Cyclomatic		4.5	4.5	4.4	4.4	4.4	4.3	4.3	4.3	4.3	4.3	4.3
KLOC		5646	7045	79.66	8673	9521	10459	11090	11941	12938	14094	14519
Existing Authors		48 %	51 %	54 %	55 %	58 %	60 %	57 %	59 %	62 %	61 %	64 %
Bug Changes		51 %	41 %	41 %	43 %	46 %	52 %	47 %	40 %	46 %	55 %	69 %
Bug Commits		45 %	43 %	42 %	39 %	39 %	39 %	40 %	41 %	43 %	43 %	43 %
Cppcheck		6247	6219	10371	12889	14406	15551	17127	16530	18317	19979	21494

Figure 1. Linux Year-by-Year Results

Apache

The results of the analysis of Apache are listed in Figure 3 and Table 4. As far as significant projects go, Apache does an outstanding job with the level of effort dedicated to bug fixes versus improvements (~32% for bug fixes). A positive aspect of Apache is that many authors remain from release to release (~85%), which means understanding can remain high, thus reducing the impact of a complicated code. We can also see improvements to cyclomatic complexity. Though Halstead measurements remain pretty steady, the coupling is pretty constant, and the style is pretty steady. Even the Cppcheck static analysis snapshot remains steady.

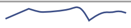
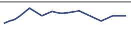
Metric	Total Change	2008	2009	2010	2011	2012	2013	2014	2015	2016	2017	2018
Style		33.3 %	31.6 %	31.0 %	32.3 %	32.3 %	32.7 %	31.9 %	32.6 %	32.1 %	31.8 %	31.0 %
Sheficom		10.7	10.65	10.74	10.99	10.92	11	11.27	11.56	11.88	11.81	12
Halstead		61.47	62.42	63.06	62.94	62.63	63.4	63.67	61.14	62.28	62.76	62.2
Cyclomatic		7.57	6.74	6.75	6.68	6.68	6.63	6.64	6.5	6.44	6.37	6.23
KLOC		143	125	131	139	151	162	170	178	189	196	209
Existing Authors		81 %	85 %	85 %	91 %	75 %	92 %	78 %	77 %	86 %	94 %	93 %
Bug Commits		14 %	43 %	37 %	41 %	98 %	28 %	9 %	18 %	21 %	26 %	26 %
Bug Changes		25 %	30 %	37 %	31 %	35 %	32 %	34 %	30 %	27 %	32 %	31 %
Cppcheck		60	68	58	61	58	64	65	64	64	67	68

Figure 2. Apache Year-by-Year Results

MySQL

The results for MySQL are listed in Figure 4 and Table 5. MySQL has dramatically increased in the number of lines of code. Note the codebase growth between 2015 and 2018, where a large amount of code was added. Along with the dramatic increase in the number of lines of code, the coupling (Sheficom) has also increased. Also, the problems predicted by static analysis (Cppcheck) have increased, and most importantly, the amount of effort spent on fixing bugs has also increased over time.

Metric	Total Change	2008	2009	2010	2011	2012	2013	2014	2015	2016	2017	2018
Style		39 %	38 %	38 %	39 %	40 %	41 %	41 %	44 %	45 %	48 %	42 %
Sheficom		6.02	6.07	6.79	7.04	7.76	8.06	8.18	8.66	9.1	13.07	13.67
Halstead		71.08	71.4	71.3	70.6	72.12	69.78	69.13	66.12	64.21	62.15	62.48
Cyclomatic		4.54	4.55	4.54	4.38	4.11	4.09	4.03	3.9	3.6	3.52	3.68
KLOC		904	923	1160	1204	1493	1571	1674	1833	2066	2581	2715
Existing Authors		46 %	58 %	66 %	71 %	66 %	73 %	73 %	67 %	82 %	77 %	92 %
Bug Commits		33 %	37 %	34 %	30 %	35 %	36 %	40 %	37 %	41 %	42 %	42 %
Bug Changes		60 %	50 %	41 %	66 %	59 %	66 %	61 %	65 %	65 %	70 %	95 %
Cppcheck		116	119	124	130	143	224	259	278	312	422	518

Figure 3. MySQL Year-by-Year Results

PHP

PHP has conditions showing a more complicated code in several areas, as shown in Figure 5. Style conformance has dropped while the number of lines of code has grown. The number of veteran authors has fluctuated, contributing to only a few people having an ongoing understanding of the code. Halstead and cyclomatic complexity both indicate a complicated code in this product, which will also negatively impact new authors joining the project. PHP is spending about half of their time fixing issues rather than adding new features.

Metric	Total Change	2008	2009	2010	2011	2012	2013	2014	2015	2016	2017	2018
Style		42 %	43 %	44 %	43 %	43 %	43 %	43 %	43 %	43 %	42 %	41 %
Sheficom		6.47	6.95	6.86	7.25	7.39	7.34	7.31	12.16	7.22	7.18	7.18
Halstead		115.1	134.1	131.6	128.2	131.8	127.5	131.6	55.19	134.0	151.9	151.1
Cyclomatic		7.68	7.65	7.69	7.65	7.74	7.82	7.77	4.3	8.27	8.23	8.19
KLOC		600	789	733	745	812	836	900	11941	1036	1183	1234
Existing Authors		59 %	83 %	80 %	84 %	49 %	45 %	53 %	59 %	42 %	45 %	64 %
Bug Changes		41 %	38 %	31 %	64 %	75 %	65 %	77 %	40 %	50 %	79 %	40 %
Bug Commits		54 %	61 %	59 %	59 %	41 %	36 %	37 %	41 %	30 %	25 %	24 %
Cppcheck		466	562	558	640	663	694	692	16530	257	362	370

Figure 4. PHP Year-by-Year Results

ImageMagic

Finally, the results for the ImageMagic project were also analyzed and are shown in Figure 6. In this project, style conformance is not that much different from other projects. Measurements like Halstead and cyclomatic complexity are not superb either. However, note the strong connection between existing authors and bug changes. After 2014, a decrease in existing authors coincides with a dramatic increase in bug efforts. The amount of energy spent on bugs impacted the amount of effort on new features as new authors arrived.

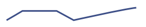
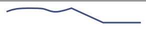
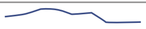
Metric	Total Change	2010	2011	2012	2013	2014	2015	2016	2017	2018
Style		40 %	41 %	41 %	40 %	40 %	40 %	41 %	40 %	40 %
Sheficom		17.24	18.13	17.93	18.77	19.05	19.2	19.96	20.04	19.5
Halstead		12.79	12.08	11.47	11.69	11.51	11.43	11.73	11.71	11.3
Cyclomatic		9.1	9.4	9.4	9.4	9.1	9.2	9.3	9.4	9.5
KLOC		287	297	314	315	335	341	342	346	353
Existing Authors		75 %	100 %	100 %	80%	100 %	44 %	14 %	12 %	11 %
Bug Changes		40 %	0 %	1 %	2%	18 %	11 %	8 %	21 %	38 %
Bug Commits		5 %	3 %	27 %	5%	13 %	17 %	27 %	34 %	47 %
Cppcheck		19	24	35	31	26	24	11	11	11

Figure 5. Image Magic Year-by-Year Results

6. DISCUSSION

This study used several metrics as a means of identifying how complicated a project's source code is. The metrics covered different areas related to a complicated code, including author turnover and module coupling. The most significant metrics seem to be the size of the project and how many human contributors there are. All software engineers have had the experience that something is initially complicated, but as time goes by and more effort is applied, the project gets less complicated. The metrics related to software complexity, as shown in Table 1, seemed to be consistent year over year on most of the projects. More successful projects maintain veteran programmers on their teams, as it would seem that new authors are not as effective. Complexity metrics, as shown in Table 1, are a harbinger of how long it may take new authors to understand the code and be productive. Projects with veteran programmers consistently show that less time and effort is required for addressing bugs in the maintenance process.

Table 3
Average Results

Category	Linux	Apache	MySQL	PHP	ImageMagic
Lines of Code	14,510,383	208,994	2,714,775	1,233,536	353,088
Lines of Code (average by year)	10,354,856	162,894	1,647,628	892,846	325,549
Style Conformance	41%	32%	42.84%	42.57%	40.33%
Cyclomatic Complexity	4.36	6.7	4.1	7.9	9.31
Halstead Difficulty	53.69	62.55	68.20	133.84	11.74
Halstead Effort	19,878.92	24,169.54	39,801	65,720	47,475.5
Coupling (Sheficom)	12.35	11.23	8.58	7.13	18.66
			Authors Year-by-Year		
Average Number	3,336	23	127	88	9
Average Veterans	57%	85%	70%	41.82%	60%
			Bug Effort (commits vs. changes)		
Commit Count	41% bugs	31.22% bugs	37% bugs	47% bugs	19.63% bugs
Changes	48% bugs	32.4% bugs	63% bugs	56% bugs	15.43% bugs
Gnat-Sized Bug	15 lines	12 lines	25 lines	9 lines	7 lines
Static Analysis Error Snapshot	14,675	64	240	500	21

7. RISKS TO VALIDITY

Several difficulties had to be addressed when doing this project. One difficulty was ascertaining the number of bugs in a project. Often developers find and fix bugs without mentioning them in bug tracking systems. It is also possible that a single “Git commit” has multiple bug fixes included. To overcome this, we evaluated bug-sized fixes and compared them to “Git commits” in general. We may be overestimating or underestimating the bug count. Another area of risk is based on the tools selected. These tools were all available open source and easy to find. However, an exhaustive test on the tools was not performed as part of this study.

Finally, an understanding of the terms *complex* and *complicated* is difficult to arrive at, as humans will all perceive these concepts differently. Something may be thought complicated, but after some time and understanding, the judgment of complicated no longer seems accurate.

8. CONCLUSION

The paper shows that there are consequences to having a complicated code. Many aspects play a part in how complicated a source code is, but the most significant contributors seem to be the size of the project and the number of authors participating. More lines of code mean more to understand. The more authors who participate in a project, the more people who need to understand its code. These factors contribute to an increased amount of time spent fixing bugs as opposed to improving the product.

Simple metrics, as used in this paper, can be employed to identify complicated areas of a project. Areas that are identified as complicated by the metrics imply it will take longer to understand those portions of the code. The longer it takes to understand, the more effort required for bug fixing. If a code is hard to understand, new authors will not be able to come up to speed quickly, and existing authors may not wish to remain on the project. Projects need to maintain loyal authors or spend more time on fixing problems instead of adding features.

Areas for future study include more effort mapping static analysis back to a complicated code and measuring how many identified areas remain in a delivered product. The second area of prospective study would be to focus on projects which follow all the rules established and verify if such projects truly have fewer errors than those which ignore these rules.

REFERENCES

Bardas, A. G. (2010). Static code analysis. *Journal of Information Systems & Operations Management*, *4*(2), 99-107.

Borowiec, D. (2014). Commented Code Detector. Retrieved June 16, 2019, from https:// https://github.com/dborowiec/commentedCodeDetector

Dorin, M. (2018, May). Coding for inspections and reviews. In *Proceedings of the 19th International Conference on Agile Software Development: Companio*n (p. 34). ACM.

Foucault, M., Palyart, M., Blanc, X., Murphy, G. C., & Falleri, J. R. (2015, August). Impact of developer turnover on quality in open-source software. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (pp. 829-841). ACM.

GitHub. (2019). Build software better, together. Retrieved June 16, from https://github.com/

Halstead, M. H. (1977). *Elements of software science* (Vol. 7, p. 127). New York: Elsevier.

Hatton, L. (1997). Reexamining the fault density component size connection. *IEEE software*, *14*(2), 89-97.

Kobashi, Y., Fukuda, M., Yoshida, K., Miyashita, N., Niki, Y., & Oka, M. (2006). Chronic necrotizing pulmonary aspergillosis as a complication of pulmonary Mycobacterium avium complex disease. *Respirology*, *11*(6), 809-813.

Lissack, M., & Roos, J. (1999). *The next common sense: Mastering corporate complexity through coherence*. Nicholas Brealey Publishing.

Manser, M. (2004). *Good Word Guide*. Bloomsbury Publishing UK.

Marjamäki , D. (2019). Cppcheck. Retrieved June 16, 2019, from http://cppcheck.sourceforge.net

McCabe, T. J. (1976). A complexity measure. *IEEE Transactions on software Engineering*, (4), 308-320.

Mish, F. C. (Ed.). (2004). *Merriam-Webster's collegiate dictionary*. Merriam-Webster.

Nagappan, N., & Ball, T. (2005, May). Use of relative code churn measures to predict system defect density. In *Proceedings of the 27th international conference on Software engineering* (pp. 284-292). ACM.

Spinellis, D. (2012). Git. *IEEE software*, *29*(3), 100-101.

Sturtevant, D. J. (2013). *System design and the cost of architectural complexity* (Doctoral dissertation, Massachusetts Institute of Technology).

Torek, A. C. Distributed Version Control With Git And Mercurial. Retrieved June 16, 2019, from http://web.torek.net/torek/tmp/book.pdf

Yin, T. (2019). Lizard. Retrieved June 16, 2019, from https://github.com/terryyin/lizard

Yoon and K. Tyagi. (2014) nsiqcppstyle. Retrieved June 16, 2019, from https://github.com/kunaltyagi/nsiqcppstyle