International Journal of Software Engineering.
ISSN 0974-3162 Volume 4, Number 2 (2013), pp. 27-34
© International Research Publication House
http://www.irphouse.com

Investigating Methods for Appraising Software Complexity

Sudhir Kumar Singh

Research Scholar,
CMJ University, Shilong,
Meghalaya, India

Abstract

Software complexity is widely regarded as an important determinant of software maintenance costs.
Increased software complexity means that maintenance and enhancement projects will take longer, will cost more, and will result in more errors. In this paper we initiate our work from Sikka Broadband Services (SBS). The purpose of our assignment at SBS is to investigate methods for measuring the complexity and Risk Analysis of software development projects.

Keywords: Software complexity, Complexity measurement, software development.

1. Introduction

One way to start the quest for a model of software complexity is to examine the origins of the concept. The intention is to be able to define what we mean by software complexity by examining where it comes from. To solve this problem we set out by looking at the different phases of the software product life cycle and try to derive different types of complexity from these stages.

Although there is very little written about the origin and nature of software complexity, some suggestions of a definition can be found. Most of them are based on the notion that software complexity is the degree of difficulty in analyzing, maintaining, testing, designing and modifying software. Our approach to the concept of software complexity, in this master thesis, will be similar to the one of these researchers. With this basis it could be expected that different types of complexity are created during each stage of the product life cycle. Hence, we are suggesting a twofold classification of complexity:

- Complexity of the problem, which is the inherent complexity, created during the requirements phase, and;
- Complexity of the solution (also referred to as added complexity), which is the complexity being attached to the complexity of the problem. This type of complexity is added during the development stages following the requirements phase, primarily during the designing and coding phases.

As we aim to confine our work at Sikka Broadband Services(SBS), Lucknow, the problem of this research work consists of three logically related parts. By answering the questions of these three problem areas, the purpose of this research work can be reached. Firstly, since SBS is interested in measuring software complexity, it is necessary to define this concept, with regard to its sources, nature and consequences. Specifically, we are interested in its relationship with other attributes of the software product and process, mainly productivity and quality. This leads us to the formulation of our first problem:

- How can software complexity be defined?
- Where does software complexity come from and what does it lead to?
- How is software complexity related to software productivity and quality?

Secondly, our interest is directed to available measures of software complexity. This means that SBS needs a comprehensive survey of which methods that are possible to apply to the projects at SBS. Therefore, our second problem is:

- How can software complexity be measured?
- Which methods and measurements are available for capturing different aspects of software complexity?

Thirdly, SBS has requested an analysis of which method or measurement that is best suited for SBS's purpose. The last part of our mission is therefore to create the basis for a choice of a software complexity measure that can be used for productivity and quality comparisons between projects. The third problem analyzed in this report is formulated as follows:

- Which method or measurement of software complexity should be chosen with regard to SBS's need of a comparative measure of project productivity and quality?
- How is this method or measure going to be implemented in SBS's system development process?

Our expectations of this field were that we would find a rather intricate picture of software complexity, with many different views of how it is influencing project productivity and product quality. We also suspect that there would be a wide range of methods available for measuring software complexity, but that each of them measured only some aspect of software complexity. Thus, the prospect of finding one method or measure that encompasses all parts of this concept was not so bright. The choice of one method would therefore be a question of priorities, and these priorities had to be based on the prerequisites at SBS.

2. Literature Review

- Tadesse, Milkias (2013) states that software complexity is one aspect that should be raised during the development of software. It is considered as one factor linked with different characteristics of quality in software. Since there are multiple developers located in different places who commit their codes to repositories, there is a need to understand the complexity of OSS before using them. A systematic use of object oriented design metrics can be useful in helping to solve this. In his paper, the complexity of the most popular open source software is investigated by the use of statistical assessment of the metrics. In order to facilitate this, he includes a case study to investigate complexity of ten popular projects that are available sourceforge. His case study has shown that applying software metrics that would measure the different aspects of software would be useful in analyzing, studying and improving the complexity of open source software.
- Syed Muhammad Ali Shah, Maurizio Morisio (2013) clustered the complexity metric values and defects in three categories as high, medium and low. Consequently they observe that for some complexity metrics high complexity results in higher defects. They called these metrics as effective indicators of defects. In the small category of projects they found LCOM as effective indicator, in the medium category of project we found WMC, CBO, RFC, CA, CE, NPM, DAM, MOA, IC, Avg CC as effective indicators of defects and for a large category of projects they found WMC, CBO, RFC, CA, NPM, AMC, Avg CC as effective indicators of defects. The difference shows that complexity metrics relation to defects also varies with the size of projects.
- Sandro Morasca (2013) described the foundations of measurement established by Measurement Theory and show how they can be used in Software Measurement for both internal and external software attributes. They also describe Axiomatic Approaches that have been defined in Software Measurement to capture the properties that measures for various software attributes are required to have. Finally, they show how Measurement Theory and Axiomatic Approaches can be used in an organized process for the definition and validation of measures used for building prediction models.
- Gagandeep Singh, Hardeep Singh (2013) gathered object oriented metrics and studied the changes in the values over different releases of software applications. They found that software tends to become more complex over the series of releases and maintaining them becomes a difficult task. They also investigated the applicability of Lehman's Law of Software Evolution using different metrics and found that Lehman's laws related to increasing complexity and continuous growth are supported by computed metrics.
- Mark A. Jacobs (2013) presented the Generalized Complexity Index (GCI), and illustrates it using publicly available data from the cruise line industry. The GCI employs the product structure diagram to create a geometric structure from which the level of the three dimensions of complexity (multiplicity,

diversity, and interconnectedness) can be computed. The GCI is a function of these three dimensions. A significant advantage of the GCI is that it can be applied at multiple levels of analysis including product, portfolio, and supply chain.

3. Research Methodology

However, to describe how our work with this assignment would evolved through time we would like to start from the beginning with our method description.

Stage 1. Defining the attributes of software complexity

After research on software complexity we would report our results and in collaboration with our superiors choose some of the factors of complexity interesting to focus on with concern to our further research of productivity measures.

Stage 2. Examining and choose tools

In stage 2 we would give a report on existing tools available on the market that can measure complexity and suggest two or three tools suitable for further examination. After more thoroughly examination of these tools we would choose the one SBS should use. The continuing work would be to use this tool on existing code from a completed software project at Sikka.

Stage 3. Complexity formula

From our tests we would have a good foundation for creating a formula that calculated the complexity of software projects at SBS. This could be done simply by using one an existing tool or, if proven to be more accurate, a combination of several tools or other important attributes for software complexity.

Stage 4. Productivity formula

In the final stage we are supposed to report the results of our measures from the completed project and give suggestions of a formula on software productivity. With the given parameters of Lines of Code, error density and man time from SBS, and the complexity metric from our complexity formula, we would create such a productivity formula.

This was the preliminary plan of the method for our work when we started. With increased knowledge of the field we had to adjust our plans to fit in with existing methods of software metrics. Our refined stages turned out to be these:

S*tage 1.Defining the attributes of software complexity*
Stage 2. Examining existing methods
Stage 3.Counting a completed project
Stage 4.Determine suggestions for Sikka

4. Field Tests

As mentioned we separated the project of study in modules, all to all twelve pieces. For reasons of simplicity we will call them A, B, C, D, E, F, G, H, I, J, K and L. In the IWD document module A, B, C and D were treated as one unit. Thus, during the first phase of the tests, we did not count A, B, C and D as separate modules, since that was not possible. However, this does not mean that the sum of FPA and FFP count for these modules in the second round of the counting is comparable to the figure we received after the first phase for A, B, C and D as a unit. This is due to the fact that when applying FPA and FFP some parts of these modules are counted several times, above all the communication internally and externally with other modules.

The results that came out of testing the methods on this project are summarized in Table 1. Some important conclusions can be drawn when looking at this table. First of all the modules H, I, J, and perhaps also modules A and K could be regarded as I/O-heavy modules, i.e. modules with a large degree of communication with other parts of the system, but not so much algorithmic complexity. This can be concluded by looking at the quota between the figures for FPA and FFP. If this quota is relatively high the FPA method has a greater impact, and thus it is probable that these modules contains much input and output, processes that are counted high with the FPA method. The modules B, C and D, on the other hand, could be regarded as modules with many sub-processes and complex algorithms, since the FFP method has a greater impact relatively to FPA. Finally, the figures in the two columns for the second round of counting are higher than in the columns representing the first round.

This means that when we increase the level of detail, regarding the documents and code analyzed, we usually find more functionality to take into account. The exceptions are modules H and I. The reason for this is that they are only concerned with input and output, i.e. the functionality is included in the IWD document.

5. Validation of our results

Our main concern was to validate the results we got from our tests from an objective viewpoint. However, a fully objective view of this specific project at SBS was hard to acquire. The approach we chose was therefore to compare our results with the jointed opinion of the system developers involved in the project. When we had concluded our tests, we asked three of the system developers involved in the project to place the modules in order of precedence. We explained that we wanted them to order the modules according to size and complexity, since FPA and FFP combine these factors. The results of our tests were not shown to them until their order of precedence was done. Naturally their knowledge of the modules varied. Some of the modules they had developed themselves, others were examined by them, but there were always one or a few modules that they had very little knowledge of. Thus, we compiled the opinions from these three persons. In this compilation, when a person had greater knowledge of a module than the others, his opinion took precedence over the others regarding this

specific module. However, the opinions were very similar, especially regarding which modules that were the most and the least complex. The order of precedence that became the result of the "opinion compilation" can be seen in Figure 1.

Table 1: Results of counting FPA and FFP.

	Counting based on IWD's		Counting based on IS's and source code	
Module	FPA	FFP	FPA	FFP
A+B+C+D	231	42		
A			455	116
B			296	134.4
C			276	183
D			268	81
E	200	35.6	283	89.6
F	184	54.2	252	79.2
G	153	42.2	201	62.4
H	117	22	121	17
I	91	21	86	16
-	89	18	170	38.4
.	65	13	137	32.8
L	65	13	113	37.2

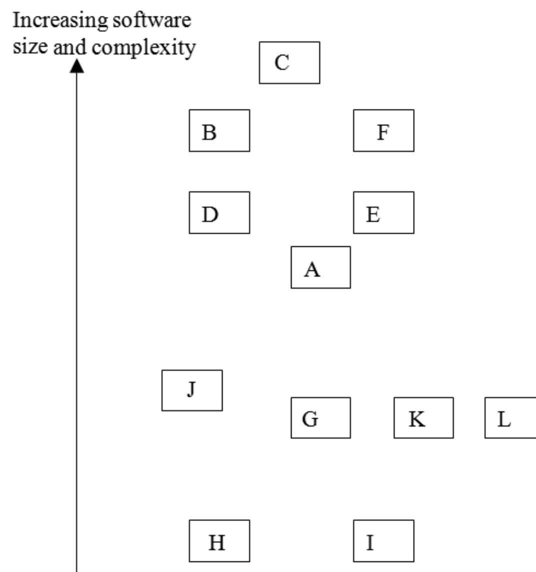

Figure 1: Module complexity according to the system developers.

The first thing to notice is that we can distinguish three groups of modules. The most complex group (C, B, F, D, E and A) consists of modules with many lines of code and many algorithms. The middle group (J, G, K and L) is made up of modules

that contains a certain amount of input and output, but also fragments with algorithms and complex functionality. The last group (H and I) of modules are units with a pure I/O- functionality.

If we compare our tests of the FPA and FFP methods on these modules with the opinion of the system developers, we find that they are rather congruent. The column in Table 1 that best agrees with Figure 1 is without doubt, the counting of FFP on IS's and source code (4th column). The order of precedence there is C, B, A, E, D, F, G, J, L, K, H and I. The explanation for the exceptions (A and G is counted higher with FFP than according to the system developers, and F is counted lower) can be found in the system developer's perception of the problem. We suspect that they interpreted the order of precedence only as a software complexity issue and disregarded the size factor. If we look at A and G we also find that they are made up of relatively many lines of code, and F is rather complex but is also a low-volume module.

Thus, the outcome of this simple but rather straightforward validation of our results speaks in favor of a detailed counting with the FFP method. As we predicted, the FPA method fails to take into account the complexity factors that are inherent in real-time systems, such as the number of algorithms and the number of sub-processes. Moreover, we need a detailed and comprehensive documentation in order to make accurate use of the FPA and FFP methods. To be able to count all functionality included in a system, we need low-level design documents, and perhaps even source code to find the parameters needed.

Conclusion

To be able to decide if Function Point Analysis or Full Function Points should be used for counting the projects at SBS, both methods were tested on a concluded project (or more correctly: part of a project). The tests were made in two rounds. In the first round only general documents, overviews of the system, were considered, and in the second round more specific documents of the low-level design and the source code were included. These results were then compared to the opinions of some of the system developers that had been involved in the project. The comparison showed that the FFP method applied to detailed documentation and code was most successful in terms of agreement with the opinions of the developers.

Reference

[1] Tadesse, Milkias (2013) , Design Metrics on Prediction of Open Source Software Complexity , master thesis, pp 35

[2] Syed Muhammad Ali Shah, Maurizio Morisio (2013), Complexity Metrics Significance for Defects: An Empirical View, Proceedings of the 2012 International Conference on Information Technology and Software Engineering Lecture Notes in Electrical Engineering Volume 212, pp 29-37

[3] Sandro Morasca (2013), Fundamental Aspects of Software Measurement, Software Engineering , Lecture Notes in Computer Science Volume 7171, pp 1-45

[4] Gagandeep Singh Hardeep Singh (2013), Effect of software evolution on metrics and applicability of Lehman's laws of software evolution, ACM SIGSOFT Software Engineering Notes. Volume 38 Issue 1, pp-1-7

[5] Mark A. Jacobs (2013), Complexity: Toward an empirical measure, Technovation, Volume 33, Issues 4–5, pp- 111–118

[6] Goodman, P. (1993). Practical implementation of software metrics. London: McGraw- Hill.

[7] Grady, R. B. (1992). Practical software metrics for project management and process improvement. New Jersey: Prentice Hall.