Software Analysis Using Code Metrics

Darren Blanckensee — 1147279

***Abstract*—Due to the rising popularity in the area of Internet of Things (IoT) there has been significant research done on how to implement edge and fog computing in order to improve the speed and efficiency of any communication between edge devices, the fog gateway and the cloud if necessary. Edge and fog computing involve data transmission whether it is between edge devices, between edge devices and fog nodes or between fog gateways and the cloud. All of these forms of transmission can be made more efficient and faster using certain filtering and compression techniques. Using filtering and compression techniques will speed up response time and use less bandwidth which will not only improve user experience but make the edge and fog computing processes more efficient in terms of space, time and energy. This would have direct effects on the efficiency of IoT systems.**

I. Introduction

SOFTWARE is being developed at an ever increasing rate. It is important when writing programs to follow good programming practices and to hold the code one writes to a certain standard so as to ensure a high quality product. Often when software projects are large with many different classes and files with thousands of lines to keep track of it becomes increasingly difficult to maintain a standard as it becomes impossible for developers to read all the lines of code and make sure that the quality requirements are met. One way to address this issue is to make use of software metrics. Software metrics are standards of measurement that provide developers with quantifiable measurements of various characteristics of the code they have developed.

One of the most simple metrics often used as an example in software is the lines of code metric (LOC) which measures how many lines of code exist in a program. This metric alone does not however provide the developer with useful information relating to the quality of the software and whether or not good programming practices are being followed. As So and so once said Puthequoteherepleasedarren. Other more useful metrics exist and the use of these in analysing software projects is the purpose of this report. All code in this project (including the metrics tool used) is written in Python.

The Chosen Metrics section details which metrics have been selected to allow for sound analysis of a code base that provides developers with useful information that could be used to maintain a standard of coding. Along with the selected metrics, explanations of each metric will be provided so as to explain the importance of each of the chosen metrics and how they should be used. The Code Base Analysis section covers the analysis of the authors own code base along with three major releases of the open source software, Freevo Media Library.

II. Chosen Metrics

The metrics being used in this project are the following:

- Cyclomatic Complexity
- Maintainability Index
- Coverage
- Halstead Metrics
- Dependencies

The tools used to calculate these metrics are Radon, Pylint, Graphviz, Webdot and Snakefood. Each of these metrics used are explained in the subsections below.

A. Cyclomatic Complexity

The cyclomatic complexity metric, calculated using the Radon tool, is used to determine the number of distinct paths a program can take during running. It gives the developer an idea of how many decisions are being made by their code. There are various constructs and conditional statements or decisions that have an effect on the cyclomatic complexity of any program. These and their additive effects on the cyclomatic complexity are shown in the table below.

TABLE I
Statements and their effects on cyclomatic complexity

Statement	Effect On Cyclomatic Complexity
If	1
Elif	1
Else	0
For	1
While	1
Except	1
Finally	0
With	1
Assert	1
Comprehension	1
Boolean Operators	1

An example (not the author's code base) and its cyclomatic complexity are shown in figure 1 and the paragraph that follows respectively.

Listing 1. Example code to explain cyclomatic complexity metric.

```
def careForDog(dog, isHungry, isHome):
    if isHungry == True:
        if isHome == True:
            dog.Feed()
        else:
            dog.Locate()
            dog.Feed()
    else:
        if isHome == False:
            dog.Locate()
        else:
            # Do nothing
```

While this code is simple and may not follow the best programming practices it is sufficient to explain how the cyclomatic complexity metric works. There are four ways this code can run. The first being if the dog is hungry and it is home then the dog will be fed. The second being if the dog is hungry but is not home then the dog will be located and fed. The third way is if the dog is not hungry and is not home then the dog will be located. The last way is if the dog is not hungry but is home then nothing is to be done. Because there are four ways this program can be run the cyclomatic complexity of this code is 4????????.

Radon was used to calculate the cyclomatic complexity of the various code bases analysed in this project. Radon's cyclomatic complexity measurement outputs a letter between A and F along with a score larger than zero relating to the number of decisions. A relates to a section of code that has less than five decisions while F relates to more than 41 decisions.

The cyclomatic complexity command on Radon returns these letters and scores for each class, method and function in a section of code. Code with many decisions means many different ways the code can run which leads to high risk that code may not behave as the developer expects or intends. It is ideal to have a cyclomatic complexity score of less than 10 or B [1].

B. Maintainability Index

The maintainability index, calculated using the Radon tool, is used to determine how easily a section of code can be maintained. This metric measures how easy it would be to change this code in future. The maintainability index as a metric was introduced is calculated using the equation below.

$$MI = max[0, 100 * \frac{171 - 5.2 \ln V - 0.23G - 16.2 \ln L + 50 \sin(\sqrt{2.4C})}{171}] \quad (1)$$

Where V is the Halstead volume, G is cyclomatic complexity, L is the number of source lines of code and C is the percent of comment lines in the program converted to radians.

III. Code Base Analysis

IV. Experimental Setup and Computational Model

V. Preliminary Results

VI. Explanation of the Rest of the Work to be Accomplished

VII. Methods for Validations of Expected Results and Exceptions

VIII. Risk Management

IX. Literature Review

X. Schedule and Time-line

XI. Summary of Proposal and Planned Additional Work to Complete

References

[1] Apostolos Papageorgiou, Bin Cheng, Erno Kovacs, Real-Time Data Reduction at the Network Edge of Internet-of-Things Systems. NEC Laboratories Europe Heidelberg, Germany, 2015.