# Code Quality Analysis

Christy McCarron
cswmccarron@dundee.ac.uk
University of Dundee
Dundee, Scotland, UK

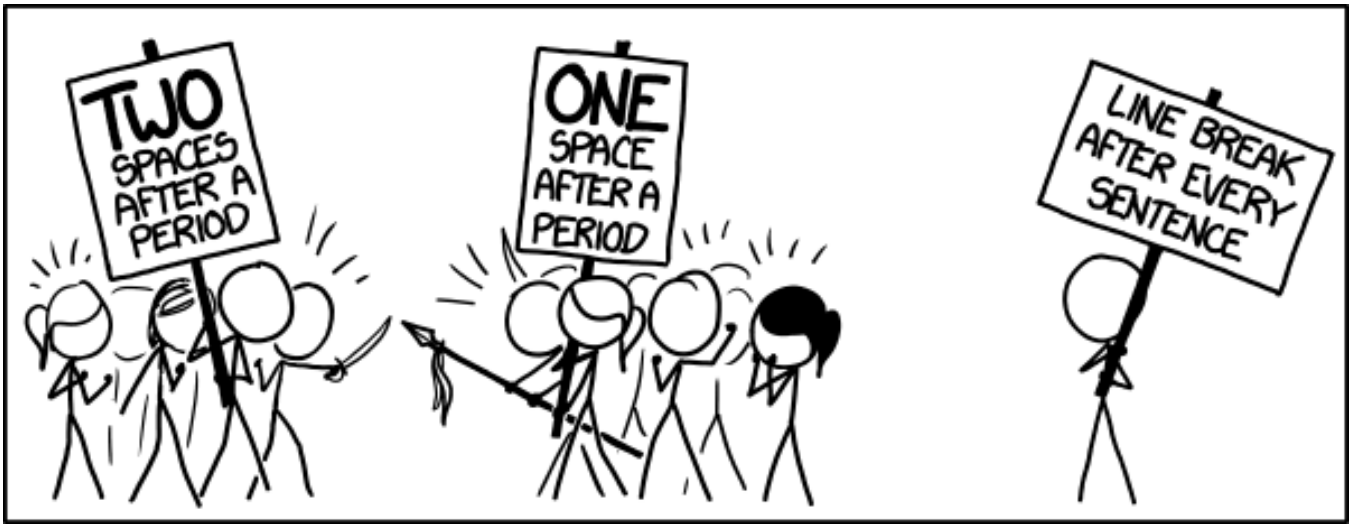


**Figure 1: Third Way, xkcd**

## ABSTRACT

The aim of this project is to develop a static analysis tool that has the ability to be used in teaching, industry and personal settings. Using a combination of syntactical and complexity measures to provide metrics on the quality of code. The node.js application will be able to be run via the command line or via a graphical interface to make determining code quality easier.

## 1 INTRODUCTION

Developing quality code is essential Code quality can be subjective, each person and organisation will have differing needs and wants when it comes to the quality of their code. But there are a many well defined areas that are as universal as can be.

The current progress is mostly research and planning with some small time taken to make prototypes although these have not gotten far.

One stumbling block was the realisation that quantitising the quality of code can be a trap, illustrated by the figure above.

```
Goodhart's Law
"When a measure becomes a target,
it ceases to be a good measure"
```

This has been the main part of my research, ensuring the measures I use are difficult to be gamed rather than completed.

The main stumbling block occured were understanding the depth of which static analysis needs to be, to ensure it is correct, this involves following similar steps to a compiler without producing a program.

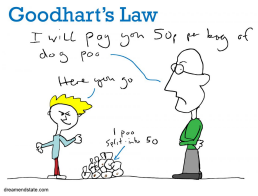


**Figure 2: Goodharts law**

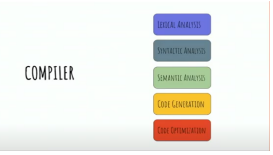


**Figure 3: Compiler**

In the figures above you can see that the early steps of a code analyser and a compiler are very similar. Performing these steps completely means that we can be sure that the source code is being parsed correctly and and proceed with analysis on our tokenised tree.

## 2 BACKGROUND

In this project the student will focus on Static Analysis methods which are performed on source code this is as opposed to Dynamic

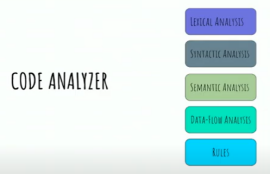


**Figure 4: Code analyser**

Analysis which involves running the program and evaluating the quality while stepping through execution.

There are many ways to measure code quality but what we must establish is:

- The code must do what it is meant to do.
- The code must be able to be tested.
- The code must be well documented.
- The code is readable and understandable.
- The code must be extendable.

## 2.1 Tokenizing

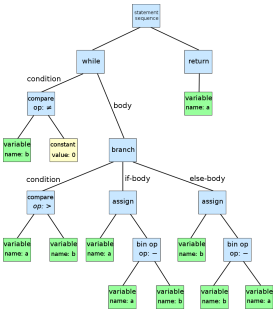


**Figure 5: Abstract Syntax Tree**

In order to create a representation of the code we must tokenize the source code into it's component pieces and store this in an Abstract Syntax Tree. We can then use this representation of the code to preform our analysis. The below code has been transformed into the AST in Figure 5 on page 2 Each possibility in execution is

```
1    while(b != 0){
2        if(a > b){
3            a = a - b
4        }else{
5            b = b -a
6        }
7    }
8    return a
```

**Figure 6: Javascript example of euclid's algorithm**

converted to a tree which follows the path.

## 2.2 Complexity Measures

To measure code complexity there are a number of algorithms which measure different aspects of complexity of the code. These measure are agnostic to the language used so once the source code has been converted to an Abstract Syntax Tree any language may be evauluated by them.

*2.2.1 Halstead Complexity Measures.* In his 1977 book M.H. Halstead described a set of complexity measures. [1]
These are described as such for any software program.

- $n^1$ : the number of unique operators
- $n^2$ : the number of unique operands
- $N^1$ : the total number of operators
- $N^2$ : the total number of operands

Then several measures can be calculated from these.

- Program Vocabulary : $n = n^1 + n^2$
- Program length : $N = N^1 + N^2$
- Calculated estimated program length : $\hat{N} = n^1 \log_2 n^1 + n^2 \log_2 n^2$
- Volume : $V = N * \log_2 n$
- Difficulty : $D = \frac{n^1}{2} * \frac{N^1}{n^1}$
- Effort : D * V
- Time required to program : $T = \frac{E}{18}$
- Number of delivered bugs : $B = \frac{V}{3000}$

```
main()
{
    int a, b, c, avg;
    scanf("%d %d %d", &a, &b, &c);
    avg = (a+b+c)/3;
    printf("avg = %d", avg);
}
```

In this example c program

- $n^1$ : 12
- $n^2$ : 7
- n : 19
- $N^1$ : 27
- $N^2$ : 15
- N : 42
- $\hat{N}$ : $12 \log_2 12 + 7 \log_2 7 = 62.67$
- V : $42 * \log_2 19 = 178.4$
- D : $\frac{12}{2} * \frac{15}{7} = 12.85$
- E : $12.85 * 178.4 = 2292.44$
- T : $\frac{2292.44}{18} = 127.357$ seconds
- B : $\frac{178.4}{3000} = 0.059$

*2.2.2 Cyclomatic Complexity.* Cyclomatic Complexity(CC) is defined as

`The number of linearly independant paths within a piece of code`

For example
This piece of code has a CC of 1

```
function test(a){
    return a
}
```

So does this

```
function test(a){
    let b = a
    b = a*b
    b*=42
```

```
    return b
}
```

But when we add a control statement with 2 paths our control flow graph now contains 2 possible flows which gives us a CC of 2

```
function test(a){
    if(a>2){
        return a
    }
    return b
}
```

This is an excellent measure to follow as not only does it make us segment our code for readability and extendability it ensures there are not too many test cases for a function.
McCabe suggested this in his 1976 paper [2]

> "Programmers have been required to calculate complexity as they create software modules. When the complexity exceeded 10 they had to either recognize and modularize subfunctions or redo the software. The intention was to keep the "size" of the modules manageable and allow for testing all the independent paths..."

What is code quality Why measure code quality Social issues Goodhart

Static vs dynamic analysis dynamic analysis static analysis

Code quality tools AST explorer Clang ESLint Google Closure Compiler code quality methods

Aims / objectives

Create tool to increase code quality deeper understand code quality

## 3 METHODOLOGY

Agile vs Waterfall

Waterfall Agile

Initial backlog / user stories / user personas

## 4 DEVELOPMENT

The main features of the project are as follows

- Web Interface with pseudo code editor
- Panel to control what analysis to perform
- Report Output
- Complexity analysis
- Syntactical analysis
- Code Smell Checking
- Readability checking

These items are also described in the User Stories Appendix

The student decided on using a node stack for the application, with a front end written in react fetching results from a node api. This was chosen as it would allow for seemless use of commandline and web interface use of the program.

Sprints explainer

Sprint x

items planned for sprint

day by day development

changes due to user testing

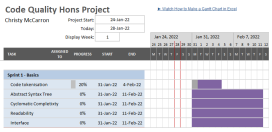


**Figure 7: Gantt Sprint 1**

## 5 DESCRIPTION OF THE FINAL PRODUCT

Front end features

## 6 APPRAISAL

## 7 CONCLUSION

## 8 FUTURE WORK

Reflecting on the progress so far I believe that I have done a good amount of research which I will now apply to coding, I wish I had started prototying earlier as that would have made the planning much easier to estimate time.
As I am following agile I have made simple plans for sprint 1 where my sprints will be 2 weeks, after the end of sprint I will have a retrospective and review the progress completed in sprint 1.
I believe over the course of the project I should be able to complete every user story following this model.
The biggest pitfall that should be encountered is ensuring the abstract syntax tree is correct is this is the bed that everything else lies on, to ensure this it will be developed with a massive amount of unit and integration testing.

javscript full syntax

allow settings

cyclomatic complexity extended to logical statements

## REFERENCES

[1] Maurice H Halstead. 1977. *Elements of Software Science.* Eleviser North-Holland.
[2] Thomas J McCabe. 1976. *A Complexity Measure.* Technical Report. IEEE.

## A FIRST APPENDIX SECTION

Appendix

### A.1 User Stories