# Code Quality Analysis

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

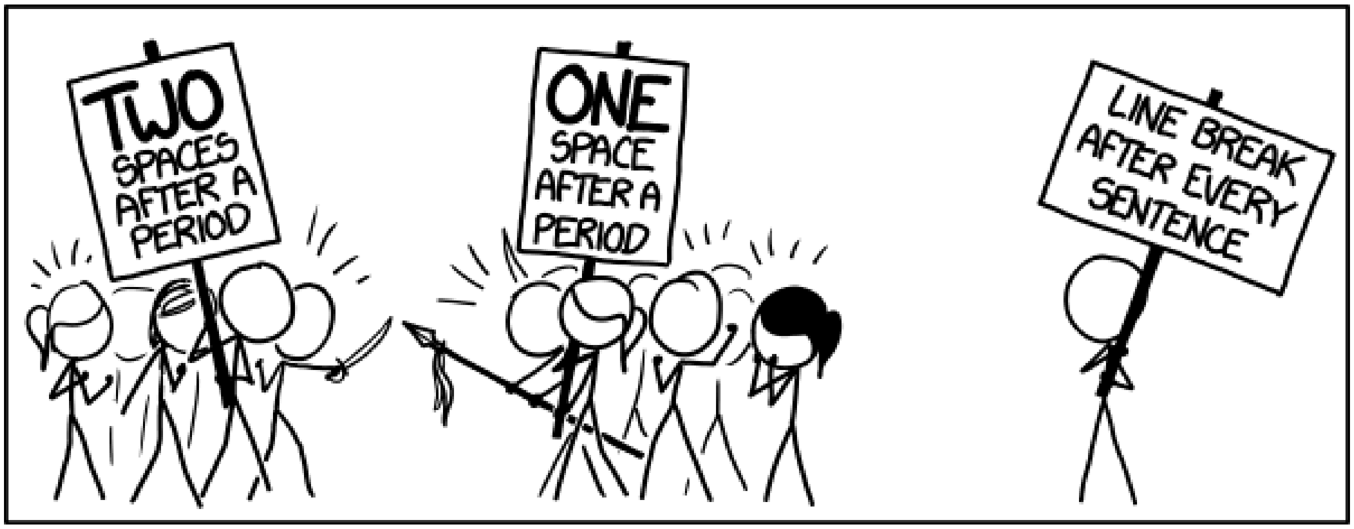


**Figure 1: Third Way, xkcd**

## Abstract

The aim of this report is to outline the use of Code Quality tools, understanding the benefits and drawbacks of using them. To understand where is appropriate to use them and how this can benefit industry and academia. An understanding of the techniques used to measure code quality and how this has changed. Outlining the creation of a bespoke code quality analysis tool that the majority say has made them more interested in analysing code quality.

## 1 Introduction

Measuring code quality can be a subjective task as what is quality code can differ depending on the purpose of the code and the criticality of the produced software. For example the guidance system on an aeroplane will have much higher quality dependencies than a computing students first Java program. What applies to both of these scenarios is that we expect the code that is written to fulfil it's intended purpose. The more critical the piece of software the more we must ensure for code quality, some of the things we are looking for are as such.

- 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.

Code Quality can be accessed by many methods including manual code review but the problem with code review is how long it can take, automatic code analysis tools can speed up and further improve code review.

In this report we will discuss

- The Background of the problem
- The Methodology behind development
- The Development process
- The Created Product
- Appraisal of the system and work
- Conclusions about the project
- Scope for Future work

## 2 Background

Here we will dicuss the background of the problem, where code quality can be used, what are the benefits to using it and how can we create a similar tool.

### 2.1 Applicability to Industry

Where we start to move towards ensuring more quality in our written code is when developing code in Industry, many techniques are utilised which includes the use of code review.

*2.1.1 Code Review* Manual code review is described as peers reviewing the code that has been written to look for problems within the written code, the results of this can be staggering on the outcomes of delivered software. In their 2006 book Jason Cohen describes the outcome of a company after implementing code review for only 3 months. [30]

```
The result: Code review would have saved half the cost of
```

```
fixing the bugs. Plus they would have found 162
additional bugs.
```

This shows the incredible power of having another pair of eyes on code before it is shipped, not only to reduce bugs as previously stated but to transfer knowledge, increase team awareness and create alternative solutions to problems [27]

*2.1.2 Use of tools in Code Review* As we have shown that code review is an important and effective task for creating quality software, we must understand the limitations it has. One of the biggest problems with Code Reviews is how long they can take the use of code quality tools speeds up the code review process by identifying potential problem areas in the code. [34]
The automatic detection and amending of simple errors in code is said to allow for developers to utilise code review for deeper and more subtle issues. [27]

```
Code review is fertile ground to have an impact with
code analysis tools.
```

[27]

## 2.2 Static Analysis

In this project the student will focus on Static Analysis methods which are performed on source code this is as opposed to Dynamic Analysis which is the analysis of the properties of a running program [28]

*2.2.1* ***Cyclomatic Complexity (CC)*** is defined as

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

The piece of code in figure *2 on page 2* has a CC of 1. So does Figure *3 on page 2*. 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 as shown in Figure *4 on page 2*.
CC is computed by evaluating the Control Flow Graph (CFG) of a program.
It can be calculated as

```
CC = E - N + 2p
```

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

**Figure 2: Javascript example of very simple function**

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

**Figure 3: Javascript example of less simple function**

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

**Figure 4: Javascript example of more complex function**

Where E = Edges , N = Nodes and p = connected components (which is always one for an independent function) [46]. Therefore the code in Figure *5 on page 2* is:

```
CC = 4 - 4 + 2
CC = 2
```

Which we can also see from following the graph itself.

```
function test(a,b){
    if (a < 10){
        a++;
    }else{
        a--;
    }
    return a
}
```

**Figure 5: Simple Javascript function with control statements**

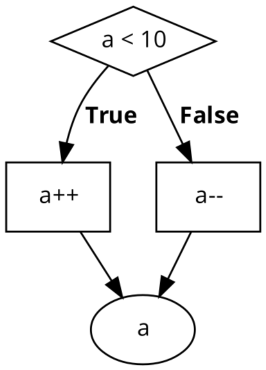


**Figure 6: Control Flow Graph from code sample in *5 on page 2* created using code2flow [35] see Appendix A**

CC can be 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 [46]

```
"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..."
```

Since it's publication there have been various critiques of the metric, the main one being it's correlation with Source Lines Of Code. On a given piece of code SLOC will linearly track CC, proven in a study of over 1.2 million files of source code [40]. While this being the case, the study also found that although LOC has massive predictive power to follow CC in a source code file, there are outliers which caused variance within the study. The relationship between CC and SLOC has been further analysed in a 2014 study, the study posits that this variation becomes more apparent when SLOC increase, control flow statements do no linearly increase with SLOC [44].

```
 "...there is no evidence of a strong
linear correlation between SLOC and CC in this large
corpus. Suggesting that for Java methods CC measures
 a different aspect of source code than SLOC..."
```

[44] Also as CC describes the amount of tests that a piece of software requires it still has it's use in Code Quality analysis.

*2.2.2* ***Halstead Complexity*** In his 1977 book M.H. Halstead described a set of complexity measures. [41]
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 : $B = \frac{V}{3000}$

Example taken from *Eindhoven University of Technology* [54]

```
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 : $\frac{178.4}{3000} = 0.059$

*2.2.3* ***Chidamber and Kemerer Class Design Metrics*** A metric used in languages with OOP features is the Chidamber and Kemerer metrics [29]. These metrics are used to determine the complexity of classes and their methods.

**Weighted Methods per Class (WMC)** This describes the number of methods per class, a high WMC count has been found to lead to more problems in the code. Classes with a high WMC could point to being needed to be split into multiple smaller classes.

**Depth of Inheritance Tree (DIT)** This describes the depth of the inheritance of a class.

```
class animal{
    constructor(){
        this.isAnimal = true;
    }
}

class dog extends animal{
    constructor(){
        super()
        this.isDog = true;
    }
}

class pug extends dog{
    constructor(){
        super()

    }
}
```

**Figure 7: Example JavaScript classes with inheritance**

In Figure *7 on page 3* we have a few example classes, the DIT values for these classes are as such.

```
animal = 0
dog = 1
pug = 2
```

It is recommended to have a DIT of 5 or less. [29]

*2.2.4* ***Other Metrics*** There are other metrics described Generally as Code Smells as described in Clean Code: A Handbook of Agile Software Craftsmanship [45].

**Function Lines** This is a metric that is tied to the number of lines that can fit on a screen, it is suggestive of bad function design if a function cannot fit on the screen, with the typical number of lines used being 20.

**Function Parameters** A function with too many parameters suggests the need for the function to be split into smaller functions or the use of a parameter object to allow for parameter specification in the function call.

```
"The ideal number of arguments for a function is
zero (niladic). Next comes one (monadic), followed
closely by two (dyadic). Three arguments (triadic)
should be avoided where possible. More than three
(polyadic) requires very special justification—and
then shouldn't be used anyway"
```

[45]

**Function / Class Redefinition** This is quite self explanatory, it is usually bad practice to define the same function and/or class twice.

**Self Inheritance** Another self explanatory measure, a class inheriting from itself would be redundant and possibly syntax breaking in many languages.

*2.2.5* ***Problems with metrics*** While considering these metrics it is important to remember the problems associated with them. Not only the critiques (as previously discussed) but also that written software source code is hard to purely quantify. Therefore it is important to ensure that metrics are used as an indication of a possible problem and not as evidence of a problem in it's entirety. One way to understand this is Goodhart's law [39], first described

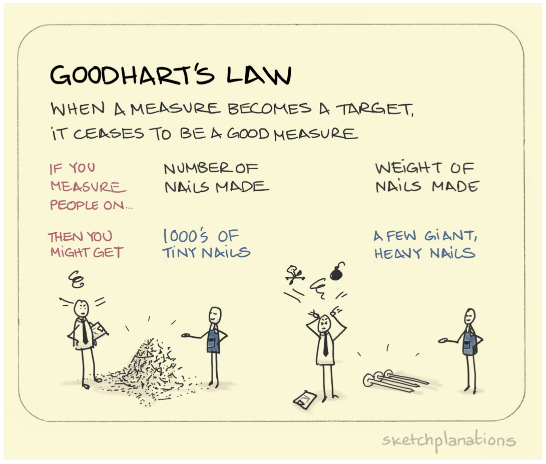


**Figure 8: Goodhart's Law Comic from Sketch Plantations [13]**

to say:

```
"Any observed statistical regularity
will tend to collapse once pressure is
placed upon it for control purposes"
```

[39] Which was generalised by Marilyn Strathern to be:

```
"When a measure becomes a target,
it ceases to be a good measure"
```

[57] This is to say that when we use a statistic or a metric as the pure evaluation of a system, the person or organisation trying to fit the metric will seek to improve their score on the metric to the detriment of the overall system as emblematized by Figure *8 on page 4*.
While Goodhart used this to describe the Monetary Policy in the United Kingdom, it has been used in various situations such as Automotive Carbon Emissions, Individual Productivity in Software Engineering and Ratings in the British University system just to name a few [52] [37] [57].

*2.2.6* ***Social and ethical consdierations with research*** A problem with the research into metrics on code quality is the collection of source code to analyse, the data collected has 2 primary sources, Companies providing their software to researchers and use of Open Sourced Repositories. Both of these have problems associated with them. Companies have an incentive to want their software to be seen as superior to their competitors, this means that a company is likely to share cherry picked code that that consider to be of good quality, skewing the data. Open source repositories have a different and more ethical concern, is that participation in Open Source communities is skewed towards the perspectives of Male programmers in the age category of 25-34 [49] [58].

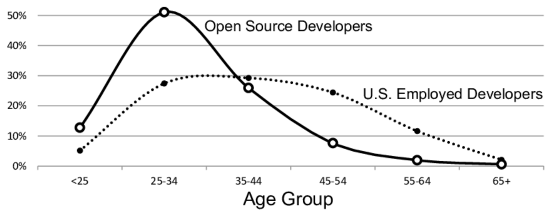


**Figure 9: Graph showing Differing ages between Open Source Developers and Employed Developers [49]**

## 2.3 Existing Code Quality Tools

There are many tools used by the modern developer to increase the quality of their code, as shown these are important to be used in industry, but should also be used in education [56] [42].

*2.3.1* ***AST Explorer*** AST explorer is a web application that allows you to view the JSON representation of the Abstract Syntax Tree (AST) of a language, it supports over 20 languages and up to 20 parsers for a given language. It also has the ability depending on the parser used to suggest readability and code styling fixes [4].

*2.3.2* ***Clang Suite*** Clang is a compiler capable of compiling c++ code, it has various tools associated with it for ensuring code quality. One of the most popular and widely used throughout industry is Clang Tidy [7]. Clang Tidy is a command line tool that can check for finding typical programming errors and readability, it also can use Clang Static Analyzer [6] to preform control flow analysis.

*2.3.3* ***ESLint*** ESLint is a static analyser that can be built into an Integrated development environment [61] or used as part of a Continuous Integration pipeline [60]. The problems that ESLint looks for can be customised and it can be configured to automatically fix issues it encounters [19].

## 2.4 Parsing

In order to perform static code quality analysis we must parse the language, this is similar to a compiler in the beginning stages as shown in Figure *10 on page 5* and Figure *11 on page 5*. A parser takes input data, in our case Source Code, as text and builds a Data Structure, in our case an Abstract Syntax Tree (ABS) that represents the structure of the input.

*2.4.1* ***Automatic Parsing*** A parser can be written by hand or can be generated by a parser generator, this is what we call Automatic Parsing or Semi-Automatic Parsing. An example of a Parser Generator is Nearley.js[17] which implements the Earley Parsing Algorithm [33]. Parser Generators typically take a definition of the

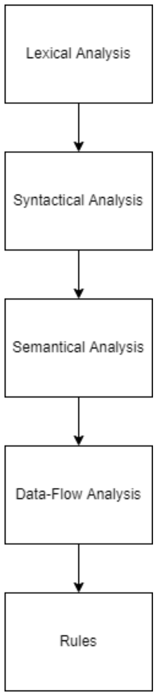


**Figure 10: Flow Diagram of stages of code analyser See Appendix B**

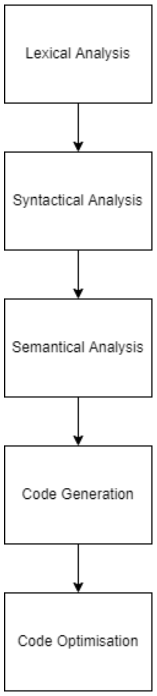


**Figure 11: Flow Diagram of stages of a compiler See Appendix A**

language, the most common format of which is Backus-Naur-Form (BNF). BNF is what we call a meta language this is because it is a language that describes the syntax of another language, in this case a programming language. In the case of the example shown in Figure *12 on page 5* a "CommonToken" can be many different things, of these it can be an "IdentifierName" which can either be an "IdentifierStart" or two tokens together in the form of "IdentifierName" and "IdentifierPart" [48].

*2.4.2* ***Manual Parsing*** If we don't want to use a parser generator we can write the parser ourselves, there are a few distinctions between the different methods of parsing that it is important to be aware of.

*2.4.2.1 Top-down vs Bottom-up.* A Top-Down parser works from the top of the parse tree and works down to the lowest element (or leaf). A Bottom-Up Parser is the opposite of this which starts from the lowest leaf and works it's way up to the top of the parse tree. Bottom-up parsers tends to use right most derivation and Top-down parsers tend to use left most derivation, meaning that when evaluating nodes the right or left child of the current node is chosen. The difference is shown in a top down parsing manner between left and right most derivation in Figure *13 on page 6* and Figure *14 on page 6* [26].

*2.4.2.2 Recursive Descent.* In section *2.4.1 on page 4* we discussed the grammar of a parser, a Recursive Descent Parser mimics this grammar in it's implementation , this will be shown later on in the report in the development section.

```
CommonToken::
    IdentifierName
    Punctuator
    NumericLiteral
    StringLiteral
    Template

IdentifierName::
    IdentifierStart
    IdentifierName IdentifierPart
```

**Figure 12: BNF grammar example in the form of an Excerpt from the ecmascript grammar definition [1]**

*2.4.2.3 Predictive Parsing.* Another feature of parsing is Backtracking, in certain parsers , possibilities for the correct parse are evaluated and abandoned. Although with smart creation of grammar it is possible to use a lookahead to the next token to predict what to parse [59].

*2.4.3 Abstract Syntax Tree.* Whatever parsing methodology we use, we need to create an AST 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 *15 on page 6*

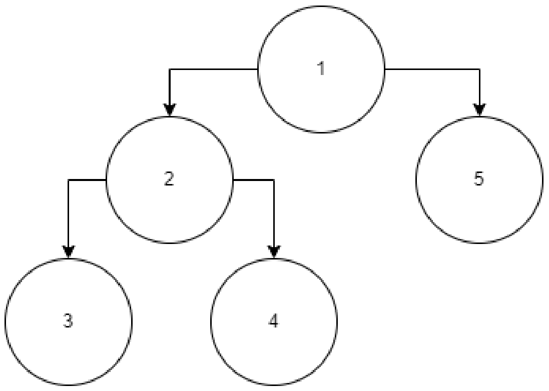


**Figure 13: Top Down Left Most Derivation of Parse Tree Diagram See Appendix L**

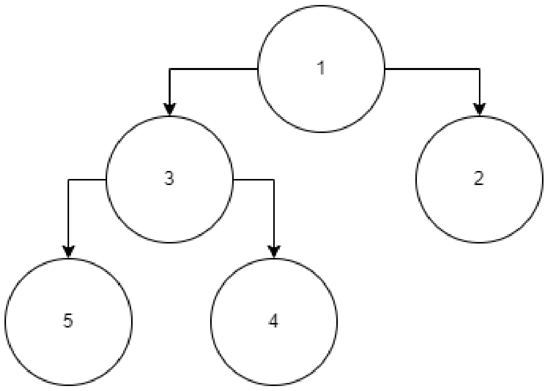


**Figure 14: Top Down Right Most Derivation of Parse Tree Diagram See Appendix M**

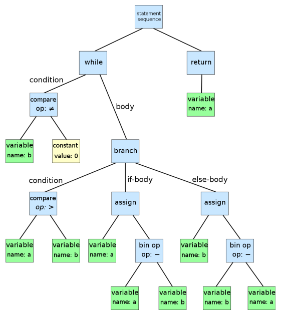


**Figure 15: Abstract Syntax Tree of Euclidean Algorithm [32]**

## 2.5 Legal issues relating to Software Quality

As the world of software development progresses and the measurement of code quality becomes standard practice ensuring high quality software can become a criminal liability. It could be argued that it is negligence to not use such a common practice as Code Quality analysis in the checking for bugs. Under the Tort Theory of Negligence. [38]

```
"...responsibility is limited to only harmful defects
that could have been detected and corrected through
“reasonable” software practices."
```

[38]. Therefore if a defect in the software was to cause harm to someone, and that bug could have been found via the use of Code Quality Analysis ,the person or organisation could be liable.

```
        while(b != 0){
            if(a > b){
                a = a - b
            }else{
                a = b -a
            }
        }
        return a
```

**Figure 16: Javascript example of euclid's algorithm**

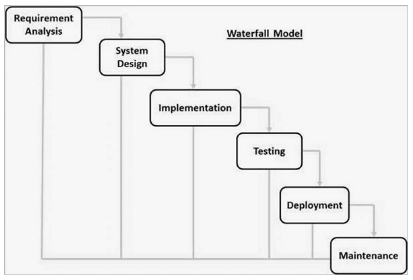


**Figure 17: Flow graph showing steps of waterfall method [2]**

## 2.6 Summary

In summary, we have seen that Code Quality Analysis and the tools used to perform it , is and are valuable software development practices. Not only in the monetary sense but in the learning it can convey and facilitate. We have seen the types of metrics Code Quality can measure and discussed the critiques and rebuttals, not only to the metrics themselves but to following a metric in itself. We have discussed some of the various tools used and the development patterns used to construct them.
It is now time to implement such a system, to understand it from a deeper level and to produce an artifact that would provide value to end users.

## 3 Methodology

### 3.1 Project Management

When choosing methodologies the student had 2 choices, to use a traditional software management technique such as Waterfall or a more modern approach such as Agile. Waterfall is as described by Dr. Winston W. Royce [53] and as shown in Figure *17 on page 6*. To be a management strategy in which the stages are quite rigid and unchanging, best suited to a project that is less likely to have changes during the development process as the subject area being explored was new to the student, it was going to be likely to see change within requirements. The creation of an artifact for end users and the communication with them during the process. Whereas Waterfall is a rigid methodology thriving on predefined

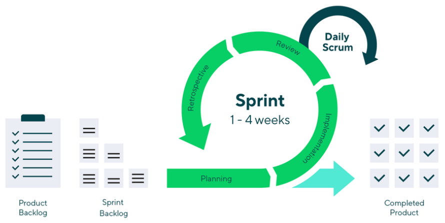
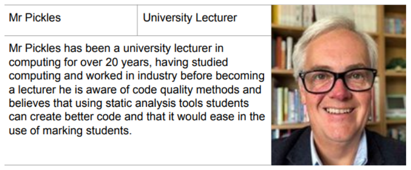


Figure 18: Steps of a sprint [14]



| Mr Pickles | University Lecturer |
|---|---|
| Mr Pickles has been a university lecturer in computing for over 20 years, having studied computing and worked in industry before becoming a lecturer he is aware of code quality methods and believes that using static analysis tools students can create better code and that it would ease in the use of marking students. | |

Figure 19: Lecturer User Persona, see appendix D

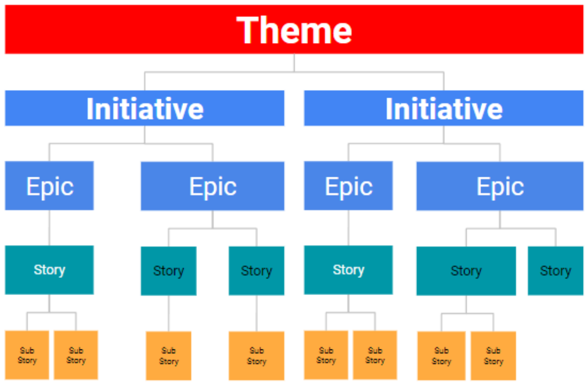


Figure 20: Structure of User Stories, See Appendix E

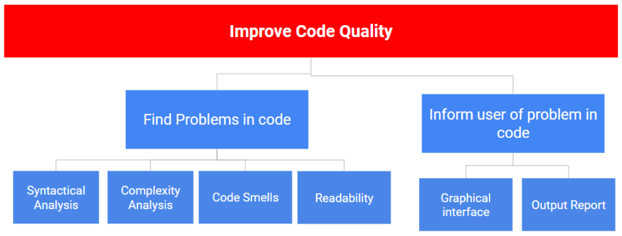
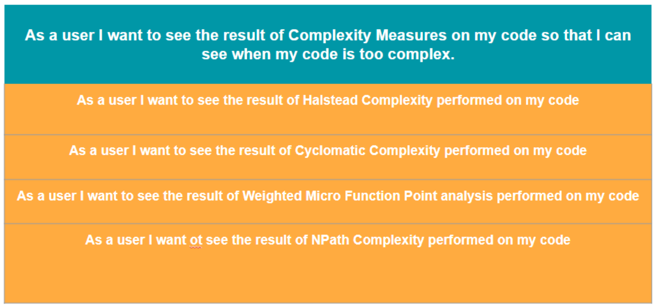


Figure 21: User Stories Theme, Initiatives and Epics, See Appendix E



As a user I want to see the result of Complexity Measures on my code so that I can see when my code is too complex.

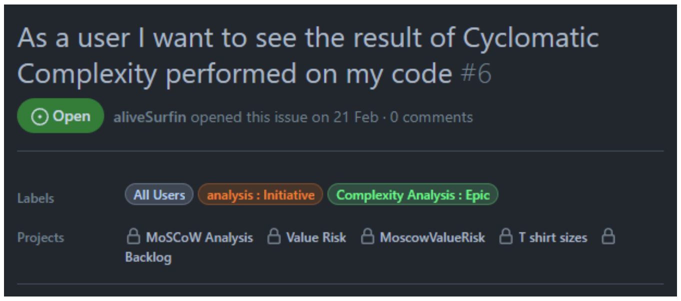
As a user I want to see the result of Halstead Complexity performed on my code

As a user I want to see the result of Cyclomatic Complexity performed on my code

As a user I want to see the result of Weighted Micro Function Point analysis performed on my code

As a user I want ot see the result of NPath Complexity performed on my code

Figure 22: Complexity Analysis User Story, See Appendix E



As a user I want to see the result of Cyclomatic Complexity performed on my code #6

Open aliveSurfin opened this issue on 21 Feb · 0 comments

Labels: All Users, analysis : Initiative, Complexity Analysis : Epic

Projects: MoSCoW Analysis, Value Risk, MoscowValueRisk, T shirt sizes, Backlog

Figure 23: User Story as Github Issue, See Appendix F

requirements Agile, as described in The Agile Manifesto [36] allows for constantly changing requirements, adapting to the needs of the project and the end user and often working better for smaller projects [47]. With this information in mind, the student took the approach of implementing agile with 2 week long sprints *18 on page 7*.

*3.1.1* ***Creation of User Personas*** User personas are an idea of a stereotypical user within a system. The student created 3 user stories to help better understand the end users of the system, these include Lecturer, Developer and Student. See an example in Figure *19 on page 7* See Appendix D.

*3.1.2* ***Creation of User Stories*** User Stories were created with the User Personas in mind. User Stories are a clear and brief description of functionality that valuable to end users [31]. Within the process of creating User Stories, a structure was created as shown in Figure *20 on page 7* and Figure *21 on page 7*. A theme is the Overall idea of what we are trying to create, in a larger organisation or project there could be multiple themes although as the project is limited in scope there is only one theme, the student felt it was prudent to ensure that the overall goal for the project to be instantiated. An Initiative is downstream from the Theme, it is something that we want to achieve and it's Epics will again be lower level ideas of how to achieve this. Within an epic we finally have our user stories which describe a goal from the perspective of a user. Each User Story may have Sub Stories which take a more granular look at the component parts to complete that User Story , an example is Figure *22 on page 7*

*3.1.3* ***User Story Prioritisation and Estimation*** User Stories were then converted to Github Issues for ease of tracking see Figure *23 on page 7*.
Using the Project Boards the issues were Prioritised and Estimated

using a number of different methods. The first of which is MoSCoW analysis which is a prioritisation technique in which the story is categorised into 4 distinct groups.

- **MUST** - Must have this item
- **SHOULD** - Should complete this item if it is possible
- **COULD** - Could have this item if it is possible
- **WON'T** - Won't have this item, but would be nice to have in the future.

[43] User stories were separated into their distinct categories, see Appendix G.
The next method of prioritisation was Value/Risk analysis.

- **Value** - How much value will completing this story item deliver to the created user personas.
- **Risk** - How difficult will it be to implement, will it take a long time to learn the skills associated and implement the feature. Will it have a high risk of failure?

These are then slotted into a matrix of 4 sections

- **High Value - Low Risk** - These are the priority story points as they are the easiest to complete and provide the most value to users.
- **High Value - High Risk** - Less priority than above but still important as they provide high value to users.
- **Low Value - Low Risk** - Nice to have , only completed when downtime in the project.
- **Low Value - High Risk** - Unless there is massive downtime in development or the Risk value changes there is usually not a reason to implement this features

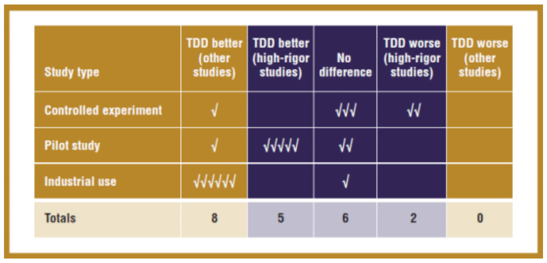
User Stories were separated into a Value/Risk board , see Appendix H.
These two prioritisation methods are combined to create a final priority board. This created these final sections. See appendix I.

(1) **High Low - Must**
(2) **High High - Must**
(3) **High Low - Should**
(4) **High High - Could**
(5) **Low Low - Could**
(6) **High High - Should**
(7) **Low High - Won't**
(8) **Low High - Could**

These priorities would be used to determine the backlog for the project and sprints.
The final technique used was an estimation technique known as T-Shirt Sizing. Stories are estimated to be of the following sizes.

- **XS**
- **S**
- **M**
- **L**
- **XL**

The first story was slotted into M and then subsequent tasks were added to the board in relation to the first task, this allowed the student to easier allocate based on bigger or smaller. See Appendix J.



| Study type | TDD better (other studies) | TDD better (high-rigor studies) | No difference | TDD worse (high-rigor studies) | TDD worse (other studies) |
|---|---|---|---|---|---|
| Controlled experiment | √ | | √√√ | √√ | |
| Pilot study | √ | √√√√√ | √√ | | |
| Industrial use | √√√√√√ | | √ | | |
| Totals | 8 | 5 | 6 | 2 | 0 |

**Figure 24: Effect of tdd on software outcomes [55]**

## 3.2 Test Driven Development

For the backend coding, a Test Driven Development approach was taken, this involves writing the tests for a piece of software before you write the software this is said to increase code quality and also creates a cleaner design, by forcing the developer to envision the end function while writing the tests as shown in Figure *25 on page 8* [55].
The student found this held true as although a test driven development approach was taken, a few parts of the software was written without a test in mind and these always turned to be less well designed and more likely to develop bugs.

```
import Parser from "../Parser"
// table of tests [program,expectedOutput]
const testTable = [
    [``, {
        "type": "Program",
        "body": []
    }]
]

const parser = new Parser()
describe('Testing empty program ', () => {
    describe.each(testTable)('parsing %s', ((program, expected) => {

        test(`returns ${JSON.stringify(expected)}`, () => {
            expect(parser.parse(program)).toEqual(expected)
        })
    }))
})
```

**Figure 25: Empty Program Parser test from "/parsing/parser/tests/empty-program.test.js" See Appendix K**

## 3.3 Summary

In summary the student decided to use an agile management method, utilising stories and a backlog to ensure development. As well as using test driven development to ensure the quality of their software.

## 4 Development

Development was tracked using a diary See Appendix S.
We will discuss the Technologies that were used during development and the reasons they were used. We will then take a walk through each sprint and discuss the development work done during each.

## 4.1 Technologies

### *4.1.1 Front-end*

*4.1.1.1 React.js* [21] Is a front-end application framework, it's benefits include the ability to segment code into components that can be reused throughout the application and it's speed in the browser, due to optimisations in it's bundling features. It was chosen due to these benefits and the fact the student had previous experience using it. There was a possibility of using another framework such as AngularJS [3] but as it is entering end of life in favour of the TypeScript version of Angular and the student had little experience with TS it felt more appropriate to use React.js.

*4.1.1.2 Prism.js* [20] is a javascript highlighting library. It was chosen due to it's various themes and the speed of execution.

*4.1.2 Back-end* A javascript based backend was chosen to ensure compatibility between the front-end and back-end of the system.

*4.1.2.1 Node.js* [50] is a javascript framework, it was chosen for it's massive amount of packages and the students experience in using it.

*4.1.2.2 Express.js* [18] is a node backend framework, it was chosen for it's simplicity of hoisting the static files to the server.

## 4.2 Sprint 1

The first sprint was interrupted due to the student catching covid. Despite this the student created a baseline for development, creating the basis of the web application. A React.js[21] front end was created using the React.js tool of Create-react-app [10]. Simple backend infrastructure was created, a node.js[50] application using express.js[18] to serve the static webpage created from React.js.

## 4.3 Sprint 2

Using a website from my research [4] as inspiration the student created initial designs as shown in Figures 26 27 28 29 30 32. These designs were created with the understanding that the code-editor should be as large as possible to allow the user to edit freely.

The next step was to highlight the code entered, with the research into prism.js [20] it seemed perfect for use in highlighting the code. The only problem is that prism.js is intended for highlighting static code These items are also described in the User Stories Appendix as seen on the prism.js Examples page. An excellent article was found written by Oliver Geer [51] describing the process of having a textarea above a highlighted code element.

As shown in Figure *33 on page 11* the textarea has events attached in the react fashion [15]. Both elements have React Refs [24] attached to allow us to access the elements throughout our React Component. When the onChange listener is fired on the textarea, the text from the text is set to state so we can access in our render function. We add an extra character to the text as a code block will ignore an empty line as shown in Figure *34 on page 11*.
On a key event being fired we check if tab is the key pressed, if it is we cancel the tabbing by calling "event.preventDefault()" [11]. We then get the cursor position and insert a tab between before and after the selection. as shown in Figure *34 on page 11*.

We set the state to the new text, starting the previous process again and then in the callback of setState [22] we update the cursor position.

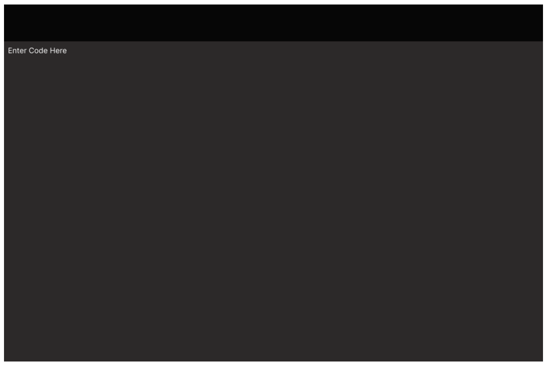
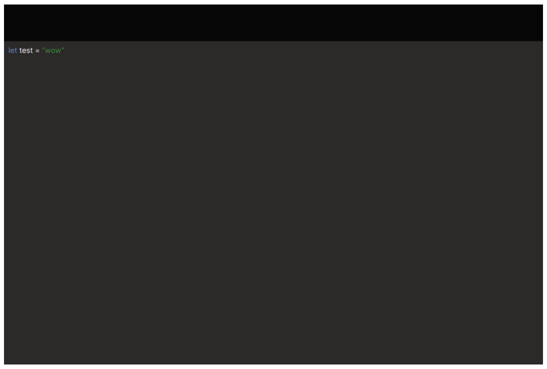
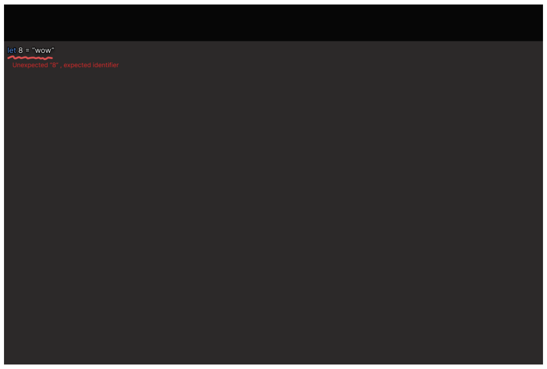


**Figure 26: Code editor designs See Appendix N**



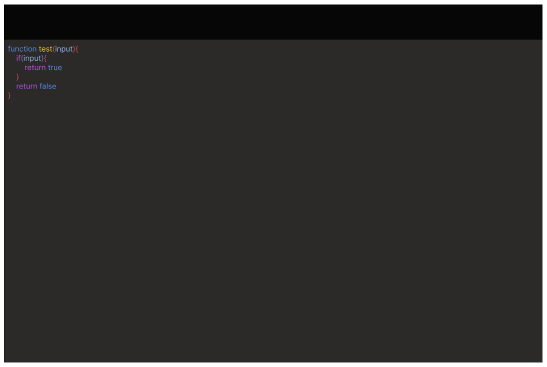
```
let test = "wow"
```

**Figure 27: Code editor designs See Appendix N**



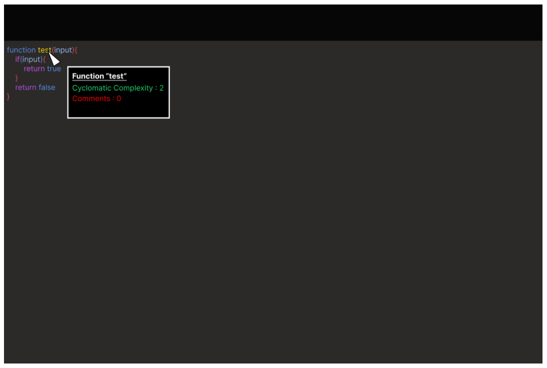
```
let 8 = "wow"
  Unexpected "8" , expected identifier
```

**Figure 28: Code editor designs See Appendix N**



```
function test(input){
  if(input){
    return true
  }
  return false
}
```

**Figure 29: Code editor designs See Appendix N**



```
function test(input){
  if(input){
    return true
  }
  return false
}
```

Function "test"
Cyclomatic Complexity : 2
Comments : 0

**Figure 30: Code editor designs See Appendix N**

This is all tied together by the process described by Figure *31 on page 10*. The code set to state is used in the render function and applied to the code element as shown in Figure *35 on page 11* by specifying "this.state.text". After the render function completes a special react function activates "componentDidUpdate" [23] which is called by react when the component updates e.g. via a state update. So once we set the text we can call highlight on it.

## 4.4 Sprint 3

The focus for Sprint 3 was to get the deployment sorted for user testing, there was a problem here. The css for highlighting using prism was not working when using a development build. The problem was that the bundler was packaging the scss files as modules, meaning that prism styling wasn't being applied. Scss modules are beneficial when using multiple components as you can specify the same class name and have them bundled differently. The fix for this was to update the react version as this was a bug fixed in a newer version. An overlay was added to inform the user of the testing procedure. The code-editor was then deployed to heroku https://cq-code-editor-testing.herokuapp.com/ and a questionnaire was sent out to potential users.

*4.4.1 Parser Course.* During this sprint the student realised they were lacking knowledge on the implementation of a parser so the student enrolled themselves in a course Building a parser from scratch by Dmitry Soshnikov [5] which is aimed at teaching the basics of compilers by creating a parser for a hypothetical language. The rest of the sprint was used following this course.

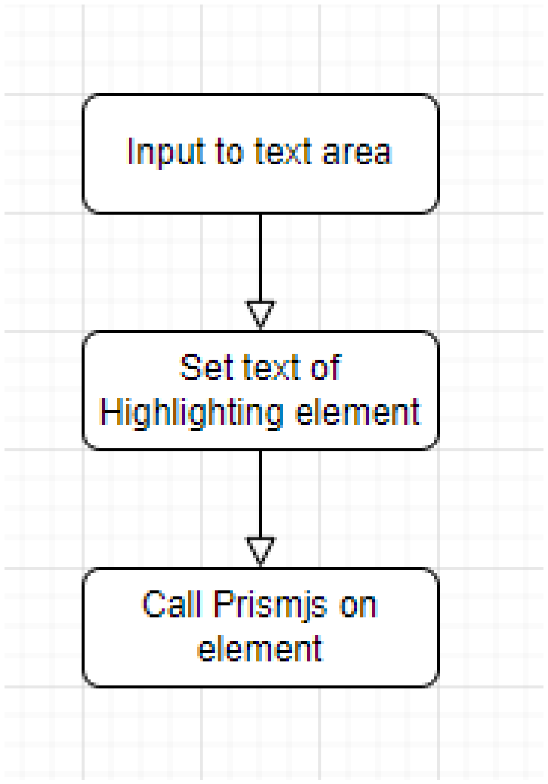


**Figure 31: Code editor activity diagram See Appendix N**

Server Status
Logo

**Figure 32: Code editor wireframe See Appendix N**

## 4.5 Sprint 4

The student completed the course on parsing and started on creating their own version of the parser. As previously described in the background section. This would be a **Top-down Recursive Descent Parser**. Taking the knowledge from the course and expanding on it to create a JavaScript parser.

*4.5.1 Tokenising.* Tokenising is the initial stage of parsing, we need to create tokens for our parser to parse. The student utilised a

```
<textarea
    ref={this.state.editingRef}
    placeholder="Enter Code Here "
    className={styles.editing}
    spellCheck="false"
    onChange={(ev) => {
        this.handleInput(ev);
    }}
    onScroll={(ev) => {
        this.handleScroll(ev);
    }}
    onKeyDown={(ev) => {
        this.handleKeyDown(ev);
    }}
></textarea>
<pre
    ref={this.state.highlightingRef}
    className={styles.highlighting}
    aria-hidden="true"
    >
<code className="language-javascript" id="highlighting-
     content">
    {this.state.text}
</code>
```

**Figure 33: React code for code editor from "/client/src/components/code_editor/CodeEditor.jsx" See Appendix K**

Regular Expression [25] array to create a token hierarchy as seen in Figure *38 on page 12* where regular expressions are matched with their corresponding token See TOKEN_CONST_TYPES.js in Appendix Q. This is then looped through by the tokeniser and if the token is not not found within the hierarchy an error is raised, if it is found it is checked for special cases e.g. the multi line string where the location must be handled because of the column and line tracking. Then the token is returned. See Figure *37 on page 12.*

*4.5.2 Parsing.* In Figure *36 on page 12* we see the parse function that starts everything, our grammar starts from the Program node which has the grammar

```
Program
    StatementList
```

This again shows the similarity between our final code and the Grammar as described in the background section. In Figure *39 on page 12* we parse a StatementList which is added to based on the null check performed in Figure *40 on page 13* because we eat semi-colons and newlines as if they were empty statements and don't add them to the parse tree, otherwise a normal statement is added from the grammar definition. We keep following the grammar tree down until we get to the most basic of elements which is the Literal as shown in *41 on page 13.*
In each of these examples we can see a few things, the use of the lookahead to determine the type, this is the predictive element of the parser and the use of AST_TYPES which is a constant file used to aid development which specifies the string names of all the AST node types See Appendix P.

*4.5.3 Testing.* Testing was performed using jest [16] and used to test every part of the parser. All of the testing suites used a test table which which allowed multiple tests to be run in the same suite with similar descriptions. In *42 on page 14* the test checks that the program parses an empty statement correctly. "testTable" is an array of arrays which contain ["input",expected].

```
handleInput(event) {
    event.persist(); // stops react from recycling the
        SyntheticEvent on re-render
    let text = this.state.editingRef.current.value;
    if (text[text.length - 1] == "\n") {
        text += " ";
    }
    this.setState({ text });
}
handleKeyDown(event) {
    if (event.key === "Tab") {
      event.preventDefault();
      this.handleTab();

      return;
}

handleTab() {
    let text = this.state.text;

    let before = text.slice(0, this.state.editingRef.
        current.selectionStart);
    let after = text.slice(
      this.state.editingRef.current.selectionEnd,
      text.length
    );

    let cursor = this.state.editingRef.current.
        selectionStart + 1;
    console.log(cursor);

    let newText = before + "\t" + after;


    this.state.editingRef.current.value = newText;
    this.setState({ text: newText }, () =>this.state.
        editingRef.current.setSelectionRange(cursor,
        cursor));
  }
```

**Figure 34: Event handlers for code editor from "/client/src/-components/code_editor/CodeEditor.jsx" See Appendix K**

```
<code className="language-javascript" id="highlighting-
     content">
    {this.state.text}
</code>

componentDidUpdate() {
    setTimeout(() => Prism.highlightAll(), 0);
  }
```

**Figure 35: Final steps of creating highlighted text from "/client/src/components/code_editor/CodeEditor.jsx" See Appendix K**

## 4.6 Sprint 5

Sprint 5 continued with progress on the parser. More javascript support was added, including the ability for statements to be ended

```
/**
* Initial Parse Function
*
*
* Starts tokenizer and updates lookahead
*
*
* Starts parsing from Program
*
* Attaches comments if available
* @param {string} input - Input string of source code
* @returns {object} - Program AST
*/
parse(input) {
    this.source = input;
    this.tokenizer.update(this.source)
    this.lookahead = this.tokenizer.next();
    const ast = this.Program()
    if (this.tokenizer.comments.length) {
        ast.comments = this.tokenizer.comments
    }
    return ast
}

/**
* Program:
*
*       -> StatementList
* @returns {object} Program AST
*/
Program() {
    return {
        type: AST_TYPES.Program,
        body: this.StatementList()
    }
}
```

**Figure 36: Initial parser steps See Appendix O**

```
        if (type === TOKEN_TYPES.MULTI_LINE_STRING) {
            this.handleMultiLine(tokenValue)
            type = TOKEN_TYPES.STRING; // we've handled
                 the multilines so just treat as a string
        }
        return {
            type: type,
            value: tokenValue,
            loc: { start: pos, end: this.position() },
        }
    }
throw new ParseSyntaxError(`Unexpected token: "${cur[0]}"
     at ${pos.line}:${pos.column}`,
        { type: "Unknown", value: cur[0], loc: { start:
             pos, end: pos } })
```

**Figure 37: Tokeniser.js See Appendix Q**

by newlines, unary and not operators, functions declarations, iteration statements, javascript class definitions, function calls and finally memmber expressions.

There was also AutoComplete and autofill functionality added due to user testing, this will be described in user evaluation.

Finally the rest of focus was used on the creation of the evaluation metrics, this was made quite simple by the use of the created

```
const TOKEN_SPEC = [
    [/^\n/, TOKEN_TYPES.NEWLINE], // caught by below,
         must come before
    [/^\s/, TOKEN_TYPES.WHITESPACE],
    [/^\/\/.*/, TOKEN_TYPES.SINGLE_LINE_COMMENT],
    [/^\/\*[\s\S]*?\*\//, TOKEN_TYPES.MULTI_LINE_COMMENT
         ],
```

**Figure 38: tokenHierarchy.js See Appendix Q**

```
        /**
        * StatementList
        *
        *      StatementList Statement -> Statement ...
        * @param {string}[d=null] stopLookingPast
        * @returns {Object[]} Array of Statements
        */
       StatementList(stopLookingPast = null) {
           let list = this.addStatementIfNotNull([])
           while (this.lookahead.type !== TOKEN_TYPES.EOF
                && this.lookahead.type !==
               stopLookingPast) {
               list = this.addStatementIfNotNull(list)
           }
           return list
       }

       /**
        * Attempt to parse statement and add to list if
             not null
        * @param {Array} list StatementList
        * @returns {Array} StatementList
        */
       addStatementIfNotNull(list) {
           const statement = this.Statement()
           if (statement != null) {
               list.push(statement)
           }
           return list
       }
```

**Figure 39: StatementList parsing See Appendix O**

AST. Cyclomatic Complexity was analysed by using the visitor pattern to crawl through the AST node and increment the complexity when finding a jumping statement, see appendix T /complexity/cyclomatic.js. Halstead Complexity was analysed by crawling through the entire AST and finding the operands and operators, see Appendix T /complexity/halstead.js. Both of these are tested in Appendix T /complexity/complexity.test.js
These are both called by evaluate.js which also checks class complexity measures, SLOC errors and function param errors. The tests for this can be found in Appentix T "evaluate.test.js"

```
        /**
        * Statement
        *
        *       : ExpressionStatement
        *       | ForStatement
        *       | DoWhileStatement
        *       | WhileStatement
        *       | ReturnStatement
        *       | ClassDeclaration
        *       | FunctionDeclaration
        *       | VariableStatement
        *       | BlockStatement
        *       | IfStatement
        *       | null : EOF | SEMI_COLON | NEWLINE
        *
        * @returns {Object} Statement
        */
       Statement() {
           if (this.lookahead.type === TOKEN_TYPES.EOF) {
               return null
           }
           switch (this.lookahead.type) {
               case TOKEN_TYPES.SEMI_COLON:
                   this.eat(TOKEN_TYPES.SEMI_COLON);
                   return null;
               case TOKEN_TYPES.NEWLINE:
                   this.eat(TOKEN_TYPES.NEWLINE);
                   return null
```

**Figure 40: Statement parsing See Appendix O**

## 5 Description of the final product

The final project is hosted at https://code-quality-honours.herokuapp.com/ See Appendix U.

**Overall Product**

In Figure *43 on page 14* we can see a screenshot of the final product, the code editor on the left and the analysis output on the right.

**Code Editor**

In Figure *44 on page 14* we can see a screenshot of the autocomplete feature, this allows the user to autocomplete code using a list of all the javascript reserved words and any words already typed in the editor by pressing tab
The user will also have brackets and strings autofilled while typing.
In Figure *45 on page 14* we can see the code the user has entered that has been highlighted.

**Analysis**

In Figure *46 on page 15* we can see the output in the analysis panel, this displays the output of the analysis on the code alongside the source code. Also in Figure *47 on page 15* we can see that clicking on a element in the output panel highlights that code within the code editor. Finally in Figure  The download button shown in *49 on page 15* allows the user to download a copy of the report.
In Figure *48 on page 15* we can see the ast panel which allows users to view the AST of their source code.

In figure *50 on page 15* we can see that the user has entered incorrect syntax. By then opening the syntax panel and clicking on the error , the user can find the syntax error in the code, as shown in Figure *51 on page 15*

```
        /**
        * Literal propretor
        *
        * Literal
        *
        *       :NumericLiteral
        *       :StringLiteral
        *       :BooleanLiteral
        *       :NullLiteral
        *
        * @throws {ParseSyntaxError} Throws error on
             unexpected literal production
        * @returns {Object} Literal
        */
       Literal() {
           switch (this.lookahead.type) {
               case TOKEN_TYPES.NUMBER:
                   return this.NumericLiteral();
               case TOKEN_TYPES.STRING:
                   return this.StringLiteral();
               case TOKEN_TYPES.TRUE:
                   return this.BooleanLiteral(this.
                        lookahead.type);
               case TOKEN_TYPES.FALSE:
                   return this.BooleanLiteral(this.
                        lookahead.type);
               case TOKEN_TYPES.NULL:
                   return this.NullLiteral();
           }
           /* istanbul ignore next */
           const loc = this.lookahead == null ? "" : `at
                ${this.lookahead.loc.start.line}:${this.
                lookahead.loc.start.col}`;
           /* istanbul ignore next */
           const type = this.lookahead == null ? "unknown
                " : `"${this.lookahead.type}"`;
           /* istanbul ignore next */
           throw new ParseSyntaxError(`Unexpected literal
                 production of type: ${this.lookahead.
                type} : ${this.lookahead.value} at ${this
                .lookahead.loc.start.line}:${this.
                lookahead.loc.start.column}`, this.
                lookahead)
       }
```

**Figure 41: Literal parsing See Appendix O**

## 6 Appraisal

### 6.1 System Appraisal

Multiple methods were used to ascertain the completeness of the application these will be discussed in relevance to the ends of the system.

*6.1.1 Back-end* The back-end of the system was appraised partly through user testing as will be discussed in the front-end section next. But primarily through the use of unit and integration tests using jest [16]. Testing was brought to 99%+ coverage in all source files as shown by the overall report in Figure *52 on page 16* and shown in Appendix V.

*6.1.2 Front-end* The front end of the system was tested via user evaluation.
The first of these was a questionnaire on the code editor [8] which

```
import Parser from "../Parser"
// table of tests [program,expectedOutput]
const testTable = [
    [
        `;
`, {
            "type": "Program",
            "body": []
        }
    ]

]




const parser = new Parser()
describe('Testing empty statement', () => {
    describe.each(testTable)('parsing %s', ((
        program, expected) => {

        test(`returns ${JSON.stringify(expected)}
            `, () => {
            expect(parser.parse(program)).toEqual
                (expected)
        })
    }))

})
```

**Figure 42: Example test See Appendix R**

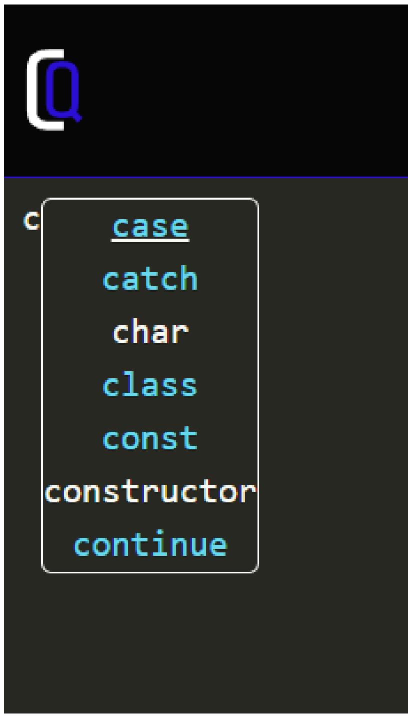


**Figure 44: AutoComplete See Appendix U**

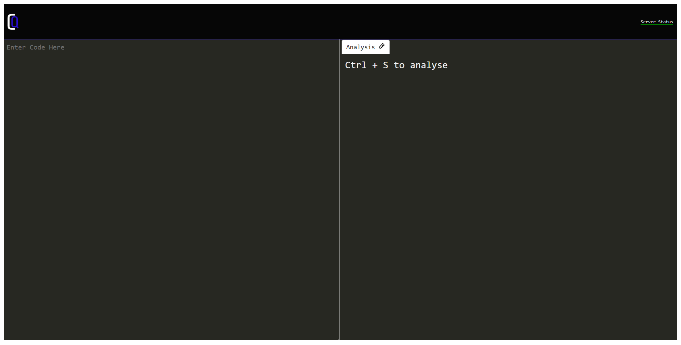
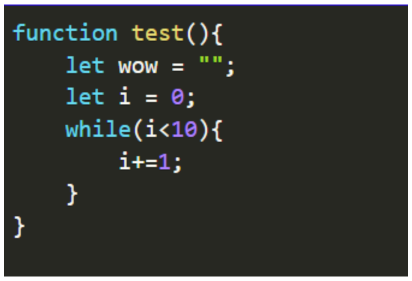


**Figure 43: Final Product See Appendix U**

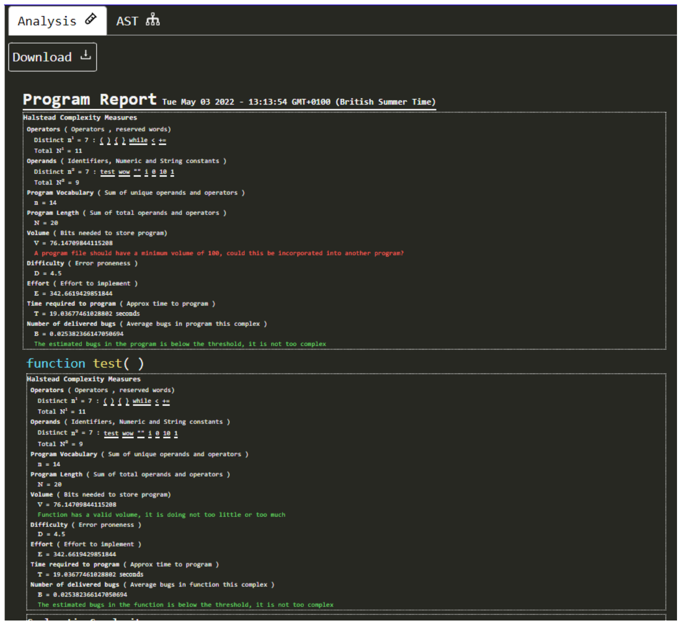


```
function test(){
    let wow = "";
    let i = 0;
    while(i<10){
        i+=1;
    }
}
```

**Figure 45: Highlighted Source Code See Appendix U**

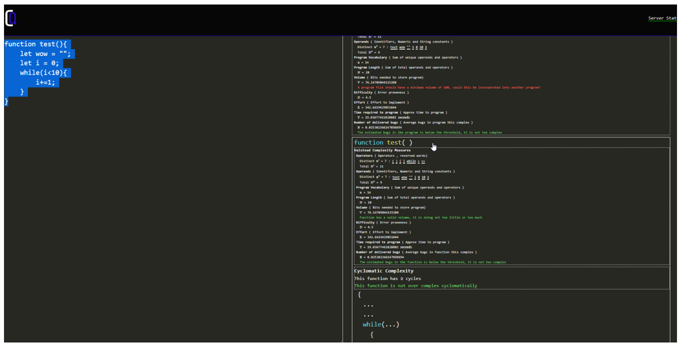
was accompanied with the participant information sheet and informed consent information. The questionnaire asked the users to voice their opinion qualitatively on the artifact. The responses were mixed, most liked the dark design of the application, although there were comments that the code editor was too simple and needed more features, this is what prompted the implementation of the autocomplete and autofill features.
Another deployment was created with the updated features [9]. The previous questionnaire prompted users to email the student if they wanted to participate in more testing, there was a response and an interview conducted. The user was asked to further use the system as they normally would and to use the new autocomplete and autofill features. The user commented that they liked the new features and especially like that it autofilled words they had previously entered. This session also unearthed a bug where in large files autocomplete would stop working which was fixed. They also mentioned that they wanted to be able to click to autocomplete, this feature did not end up being implemented. See Appendix W for anonymised interview notes.

Finally the system was evaluated by the use of a Questionnaire. The final system was deployed [12] and users were prompted to evaluate the system quantitatively. As shown in Figure *54 on page 16* users were happy with the analysis output panels with 100% of responses being Agree or Strongly Agree. Sentiment towards the code editor was mixed but remained neutral at worst. 67% of responses strongly agreed that this application had made them more interested in code quality analysis, this is



```
Program Report Tue May 03 2022 - 13:13:54 GMT+0100 (British Summer Time)
Halstead Complexity Measures
 Operators ( Operators , reserved words)
   Distinct n^1 = 7 : ( ) { } while < +=
   Total N^1 = 11
 Operands ( Identifiers, Numeric and String constants )
   Distinct n^2 = 7 : test wow "" i 0 10 1
   Total N^2 = 9
 Program Vocabulary ( Sum of unique operands and operators )
   n = 14
 Program Length ( Sum of total operands and operators )
   N = 20
 Volume ( Bits needed to store program)
   V = 76.14709844115208
   A program file should have a minimum volume of 100, could this be incorporated into another program?
 Difficulty ( Error proneness )
   D = 4.5
 Effort ( Effort to implement )
   E = 342.6619429851844
 Time required to program ( Approx time to program )
   T = 19.03677461028802 seconds
 Number of delivered bugs ( Average bugs in program this complex )
   B = 0.025382366147050694
   The estimated bugs in the program is below the threshold, it is not too complex
function test( )
Halstead Complexity Measures
 Operators ( Operators , reserved words)
   Distinct n^1 = 7 : ( ) { } while < +=
   Total N^1 = 11
 Operands ( Identifiers, Numeric and String constants )
   Distinct n^2 = 7 : test wow "" i 0 10 1
   Total N^2 = 9
 Program Vocabulary ( Sum of unique operands and operators )
   n = 14
 Program Length ( Sum of total operands and operators )
   N = 20
 Volume ( Bits needed to store program)
   V = 76.14709844115208
   Function has a valid volume, it is doing not too little or too much
 Difficulty ( Error proneness )
   D = 4.5
 Effort ( Effort to implement )
   E = 342.6619429851844
 Time required to program ( Approx time to program )
   T = 19.03677461028802 seconds
 Number of delivered bugs ( Average bugs in function this complex )
   B = 0.025382366147050694
   The estimated bugs in the function is below the threshold, it is not too complex
```

Figure 46: Analysis Output See Appendix U



```
function test(){
    let wow = "";
    let i = 0;
    while(i<10){
        i+=1;
    }
}
```

```
 Operands ( Identifiers, Numeric and String constants )
   Distinct n^2 = 7 : test wow "" i 0 10 1
   Total N^2 = 9
 Program Vocabulary ( Sum of unique operands and operators )
   n = 14
 Program Length ( Sum of total operands and operators )
   N = 20
 Volume ( Bits needed to store program)
   V = 76.14709844115208
   A program file should have a minimum volume of 100, could this be incorporated into another program?
 Difficulty ( Error proneness )
   D = 4.5
 Effort ( Effort to implement )
   E = 342.6619429851844
 Time required to program ( Approx time to program )
   T = 19.03677461028802 seconds
 Number of delivered bugs ( Average bugs in program this complex )
   B = 0.025382366147050694
   The estimated bugs in the program is below the threshold, it is not too complex
function test( )
Halstead Complexity Measures
 Operators ( Operators , reserved words)
   Distinct n^1 = 7 : ( ) { } while < +=
   Total N^1 = 11
 Operands ( Identifiers, Numeric and String constants )
   Distinct n^2 = 7 : test wow "" i 0 10 1
   Total N^2 = 9
 Program Vocabulary ( Sum of unique operands and operators )
   n = 14
 Program Length ( Sum of total operands and operators )
   N = 20
 Volume ( Bits needed to store program)
   V = 76.14709844115208
   Function has a valid volume, it is doing not too little or too much
 Difficulty ( Error proneness )
   D = 4.5
 Effort ( Effort to implement )
   E = 342.6619429851844
 Time required to program ( Approx time to program )
   T = 19.03677461028802 seconds
 Number of delivered bugs ( Average bugs in function this complex )
   B = 0.025382366147050694
   The estimated bugs in the function is below the threshold, it is not too complex
Cyclomatic Complexity
This function has 2 cycles
This function is not over complex cyclomatically
 {
  ...
  ...
  while(...)
    {
```

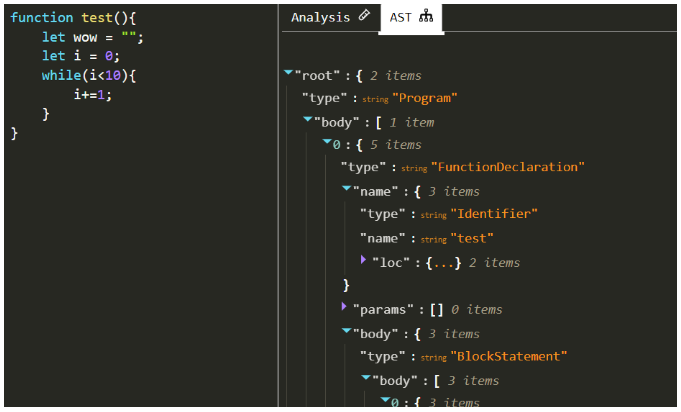
Figure 47: Clicking on function in analysis See Appendix U

## 6.2 Appraisal of Work

I believe that work was generally good on the project. If I was to do this project again I would likely use a pre-made parser, this would have made the project so much easier and allowed for more to be accomplished during the project, understanding this it was a learning experience that has given the student deeper understanding of how parsing works.
I would have also spent less time on research and more time on development, again this gave me a deep understanding of the subject matter but more time for development would have been beneficial.

## 7 Conclusion

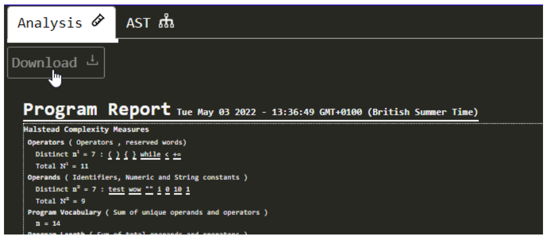
The goal of this project was to understand the code quality analysis process and learn how parsing is done, the student believes this has been achieved. The artifact produced meets the requirements of 18/30 of the user stories created, with the majority of the stories left uncompleted being smaller stories that are made easier by the work that has been completed.
Coming into this project with no understanding of how static analysis worked and how parsers were made it has been a great learning experience that the student believes will benefit them in the long term. how do we take what we have learned forward
The final artifact created needs work to be a widely used application but as the results from the user testing show, it has created an interest in the participants in code quality analysis. This is justification enough to the student that it has been a success.



```
function test(){
    let wow = "";
    let i = 0;
    while(i<10){
        i+=1;
    }
}
```

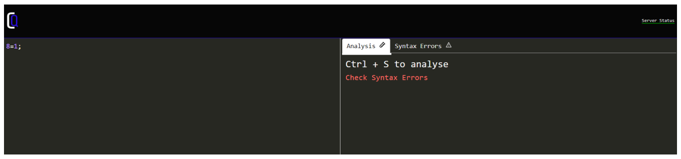
```
▾"root" : { 2 items
  "type" : string "Program"
  ▾"body" : [ 1 item
    ▾0 : { 5 items
      "type" : string "FunctionDeclaration"
      ▾"name" : { 3 items
        "type" : string "Identifier"
        "name" : string "test"
        ▸"loc" : {...} 2 items
      }
      ▸"params" : [] 0 items
      ▾"body" : { 3 items
        "type" : string "BlockStatement"
        ▾"body" : [ 3 items
          ▾0 : { 3 items
```

Figure 48: AST panel See Appendix U



```
Program Report Tue May 03 2022 - 13:36:49 GMT+0100 (British Summer Time)
Halstead Complexity Measures
 Operators ( Operators , reserved words)
   Distinct n^1 = 7 : ( ) { } while < +=
   Total N^1 = 11
 Operands ( Identifiers, Numeric and String constants )
   Distinct n^2 = 7 : test wow "" i 0 10 1
   Total N^2 = 9
 Program Vocabulary ( Sum of unique operands and operators )
   n = 14
```

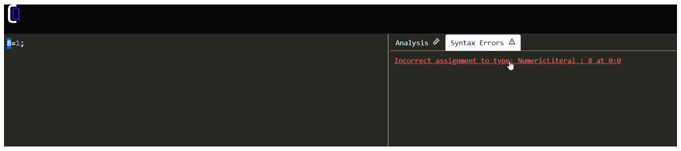
Figure 49: Download Button See Appendix U



```
8=1;
```

```
Ctrl + S to analyse
Check Syntax Errors
```

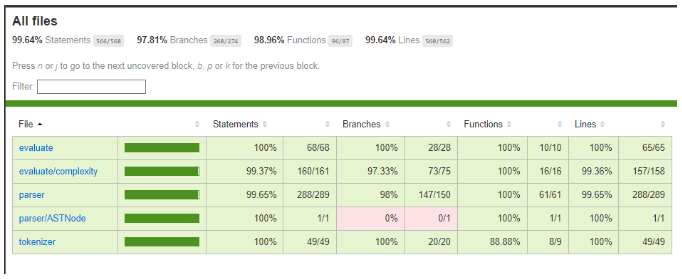
Figure 50: Syntax Error See Appendix U



```
8=1;
```

```
Incorrect assignment to type: NumericLiteral : 8 at 0:0
```

Figure 51: Syntax Error Clicking See Appendix U



**All files**

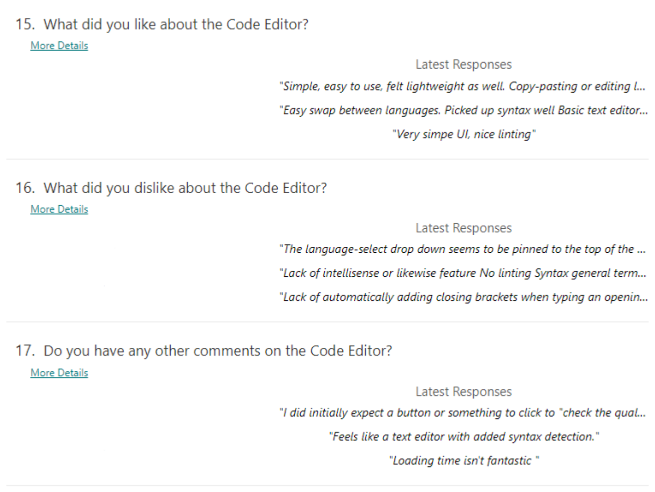
99.64% Statements 566/568 97.81% Branches 268/274 98.96% Functions 96/97 99.64% Lines 560/562

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

Filter:

| File | | Statements | | Branches | | Functions | | Lines | |
|---|---|---|---|---|---|---|---|---|---|
| evaluate | | 100% | 68/68 | 100% | 28/28 | 100% | 10/10 | 100% | 65/65 |
| evaluate/complexity | | 99.37% | 160/161 | 97.33% | 73/75 | 100% | 16/16 | 99.36% | 157/158 |
| parser | | 99.65% | 288/289 | 98% | 147/150 | 100% | 61/61 | 99.65% | 288/289 |
| parser/ASTNode | | 100% | 1/1 | 0% | 0/1 | 100% | 1/1 | 100% | 1/1 |
| tokenizer | | 100% | 49/49 | 100% | 20/20 | 88.88% | 8/9 | 100% | 49/49 |

**Figure 52: Testing Coverage See Appendix V**



15. What did you like about the Code Editor?
More Details

Latest Responses

*"Simple, easy to use, felt lightweight as well. Copy-pasting or editing l...*

*"Easy swap between languages. Picked up syntax well Basic text editor...*

*"Very simpe UI, nice linting"*

16. What did you dislike about the Code Editor?
More Details

Latest Responses

*"The language-select drop down seems to be pinned to the top of the ...*

*"Lack of intellisense or likewise feature No linting Syntax general term...*

*"Lack of automatically adding closing brackets when typing an openin...*

17. Do you have any other comments on the Code Editor?
More Details

Latest Responses

*"I did initially expect a button or something to click to "check the qual...*

*"Feels like a text editor with added syntax detection."*

*"Loading time isn't fantastic "*

**Figure 53: Responses to the code editor testing**

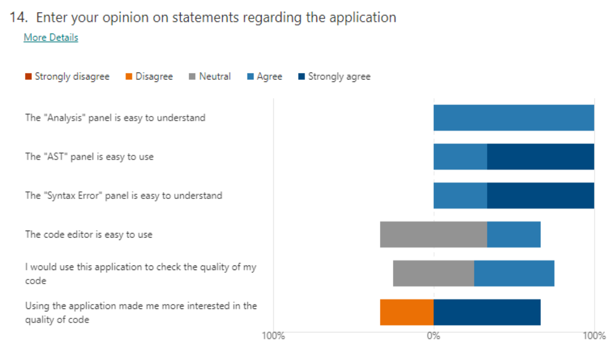


**Figure 54: Responses to the final testing**

## 8 Future Work

There is scope within the project for further work to be done, I will discuss these now.

### 8.1 Expanding of Syntax Support

The parser supports a lot of the JavaScript language although it does not support modern features, bringing the parser up to date to support the full JavaSript grammar [1].

### 8.2 Allow Settings

It would be great to allow the user to select what analysis they want to perform, this would be beneficial for teaching as the educator could customise the output depending on the level of study.

## Acknowledgments

The author wishes to thank their advisor Dr. Craig Ramsay for their help through the entire process, who was always prompt with communication and willing to set up meetings right until the very end. Furthermore was willing to bounce ideas off constantly.

The author would like to thank Dr. Michael Crabb the Teaching Lead for computing at University of Dundee, for the creation of the LaTeX template which was used to write this report.

The author would like to bring attention to the writings of Aaron Swartz on The Guerilla Open Access Manifesto , without the principles therein it would have been much more difficult to complete the project.

## References

[1] [n. d.]. https://262.ecma-international.org/8.0/
[2] [n. d.]. https://www.tutorialspoint.com/sdlc/sdlc_waterfall_model.htm
[3] [n. d.]. Angular: Superheroic JavaScript MVW framework. https://angularjs.org/
[4] [n. d.]. AST Explorer. https://astexplorer.net/
[5] [n. d.]. Building a parser from scratch. http://dmitrysoshnikov.com/courses/parser-from-scratch/
[6] [n. d.]. Clang Static Analyzer. https://clang-analyzer.llvm.org/
[7] [n. d.]. Clang Tidy. https://clang.llvm.org/extra/clang-tidy/
[8] [n. d.]. Code editor testing. https://cq-code-editor-testing.herokuapp.com/
[9] [n. d.]. Code editor testing updated. https://code-editor-updated.herokuapp.com/
[10] [n. d.]. Create a new react app. https://reactjs.org/docs/create-a-new-react-app.html
[11] [n. d.]. Event.preventDefault() - web apis: MDN. https://developer.mozilla.org/en-US/docs/Web/API/Event/preventDefault
[12] [n. d.]. Final testing deploy. https://code-quality-final-testing.herokuapp.com/
[13] [n. d.]. Goodhart's law: When a measure becomes a target, it cease to be a good measure. https://sketchplanations.com/goodharts-law
[14] [n. d.]. Guide to scrum sprints: Wrike Scrum Guide. https://www.wrike.com/scrum-guide/scrum-sprints/
[15] [n. d.]. Handling events. https://reactjs.org/docs/handling-events.html
[16] [n. d.]. Jest · delightful JavaScript testing. https://jestjs.io/
[17] [n. d.]. Nearley.js2.20.1. https://nearley.js.org/
[18] [n. d.]. Node.js web application framework. https://expressjs.com/
[19] [n. d.]. Pluggable javascript linter. https://eslint.org/
[20] [n. d.]. Prism.js. https://prismjs.com/
[21] [n. d.]. React a JavaScript library for building user interfaces. https://reactjs.org/
[22] [n. d.]. React.component. https://reactjs.org/docs/react-component.html#setstate
[23] [n. d.]. React.component. https://reactjs.org/docs/react-component.html#componentdidupdate
[24] [n. d.]. Refs and the dom. https://reactjs.org/docs/refs-and-the-dom.html
[25] [n. d.]. Regular expressions - javascript: MDN. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions
[26] 2019. Difference between top down parsing and bottom up parsing. https://www.geeksforgeeks.org/difference-between-top-down-parsing-and-bottom-up-parsing/
[27] Alberto Bacchelli and Christian Bird. 2013. Expectations, outcomes, and challenges of modern code review. In *2013 35th International Conference on Software Engineering (ICSE)*. 712–721. https://doi.org/10.1109/ICSE.2013.6606617
[28] Thoms Ball. 1999. The Concept of Dynamic Analysis. In *Proceedings of the 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Toulouse, France) *(ESEC/FSE-7)*. Springer-Verlag, Berlin, Heidelberg, 216–234.
[29] S.R. Chidamber and C.F. Kemerer. 1994. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering* 20, 6 (1994), 476–493. https://doi.org/10.1109/32.295895
[30] J. Cohen, E. Brown, S. Teleki, and B. DuRette. 2006. *Best Kept Secrets of Peer Code Review.* Printing Systems. https://books.google.co.uk/books?id=

b9ywHanWN5kC

[31] Mike Cohn. 2004. *User stories applied: For agile software development.* Addison-Wesley Professional.

[32] Dcoetzee. 2011. *An abstract syntax tree for the following code for the Euclidean algorithm.* https://en.wikipedia.org/wiki/Abstract_syntax_tree#/media/File:Abstract_syntax_tree_for_Euclidean_algorithm.svg

[33] Jay Earley. 1970. An Efficient Context-Free Parsing Algorithm. *Commun. ACM* 13, 2 (feb 1970), 94–102. https://doi.org/10.1145/362007.362035

[34] Felipe Ebert, Fernando Castor, Nicole Novielli, and Alexander Serebrenik. 2019. Confusion in Code Reviews: Reasons, Impacts, and Coping Strategies. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER).* 49–60. https://doi.org/10.1109/SANER.2019.8668024

[35] Code 2 Flow. 2022. *code2flow online interactive code to flowchart converter.* https://code2flow.com/

[36] Martin Fowler, Jim Highsmith, et al. 2001. The agile manifesto. *Software development* 9, 8 (2001), 28–35.

[37] T. Fritz. 2016. Measuring individual productivity. In *Perspectives on Data Science for Software Engineering*, Tim Menzies, Laurie Williams, and Thomas Zimmermann (Eds.). Morgan Kaufmann, Boston, 67–71. https://doi.org/10.1016/B978-0-12-804206-9.00013-1

[38] Karen Goertzel. 2016. Legal liability for bad software. *CrossTalk* 29 (01 2016), 23–28.

[39] C. A. E. Goodhart. 1984. *Problems of Monetary Management: The UK Experience.* Macmillan Education UK, London, 91–121. https://doi.org/10.1007/978-1-349-17295-5_4

[40] Jay Graylin, Joanne E Hale, Randy K Smith, Hale David, Nicholas A Kraft, WARD Charles, et al. 2009. Cyclomatic complexity and lines of code: empirical evidence of a stable linear relationship. *Journal of Software Engineering and Applications* 2, 03 (2009), 137.

[41] Maurice H Halstead. 1977. *Elements of Software Science.* Eleviser North-Holland.

[42] Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. 2017. Code Quality Issues in Student Programs. 110–115. https://doi.org/10.1145/3059009.3059061

[43] Janet Kuhn. 2009. Decrypting the MoSCoW analysis. *The workable, practical guide to Do IT Yourself* 5 (2009).

[44] Davy Landman, Alexander Serebrenik, and Jurgen Vinju. 2014. Empirical Analysis of the Relationship between CC and SLOC in a Large Corpus of Java Methods. In *2014 IEEE International Conference on Software Maintenance and Evolution.* 221–230. https://doi.org/10.1109/ICSME.2014.44

[45] Robert C. Martin. 2008. *Clean Code: A Handbook of Agile Software Craftsmanship* (1 ed.). Prentice Hall PTR, USA.

[46] Thomas J McCabe. 1976. *A Complexity Measure.* Technical Report. IEEE.

[47] Mike McCormick. 2012. Waterfall vs. Agile methodology. *MPCS, N/A* (2012).

[48] Daniel D. McCracken and Edwin D. Reilly. 2003. *Backus-Naur Form (BNF).* John Wiley and Sons Ltd., GBR, 129–131.

[49] Patrick Morrison, Rahul Pandita, Emerson Murphy-Hill, and Anne McLaughlin. 2016. Veteran developers' contributions and motivations: An open source perspective. 171–179. https://doi.org/10.1109/VLHCC.2016.7739681

[50] Node.js. [n. d.]. https://nodejs.org/en/

[51] Oliver Geer on Apr 16 and Oliver Geer. 2021. Creating an editable Textarea that supports syntax-highlighted code: CSS-tricks. https://css-tricks.com/creating-an-editable-textarea-that-supports-syntax-highlighted-code/

[52] Mathias Reynaert and James M Sallee. 2016. *Corrective Policy and Goodhart's Law: The Case of Carbon Emissions from Automobiles.* Working Paper 22911. National Bureau of Economic Research. https://doi.org/10.3386/w22911

[53] Winston W Royce. 1987. Managing the development of large software systems: concepts and techniques. In *Proceedings of the 9th international conference on Software Engineering.* 328–338.

[54] Alexander Serebrenik. 2016. 2IS55 Software Evolution.

[55] Forrest Shull, Grigori Melnik, Burak Turhan, Lucas Layman, Madeline Diep, and Hakan Erdogmus. 2010. What Do We Know about Test-Driven Development? *IEEE Software* 27, 6 (2010), 16–19. https://doi.org/10.1109/MS.2010.152

[56] Danilo Silva, Ingrid Nunes, and Ricardo Terra. 2017. Investigating code quality tools in the context of software engineering education: CODE TOOLS IN SOFTWARE ENGINEERING EDUCATION. *Computer Applications in Engineering Education* 25 (02 2017). https://doi.org/10.1002/cae.21793

[57] Marilyn Strathern. 1997. 'Improving ratings': audit in the British University system. *European review* 5, 3 (1997), 305–321.

[58] Bianca Trinkenreich. 2021. Please Don't Go — A Comprehensive Approach to Increase Women's Participation in Open Source Software. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion).* 293–298. https://doi.org/10.1109/ICSE-Companion52605.2021.00131

[59] Des Watson. 2017. *A Practical Approach to Compiler Construction.* Springer. libgen.li/file.php?md5=08f72a9fa85f7793297178164a55580dc

[60] Wikipedia. 2022. Continuous integration — Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/w/index.php?title=Continuous%20integration&oldid=1081041616.

[61] Wikipedia. 2022. Integrated development environment — Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/w/index.php?title=Integrated%20development%20environment&oldid=1084718432.

# Appendices

Each appendix can be found either as a LINK or in the sub folder with it's label, e.g. Appendix A is found in the path "/A/"

## A Control Flow Graph

## B Code Analyser Diagram

## C Compiler Diagram

## D User Personas

## E User Stories

## F Github Issues

https://github.com/aliveSurfin/code_quality/issues

## G MoSCoW Analysis

https://github.com/aliveSurfin/code_quality/projects/1

## H Value Risk

https://github.com/aliveSurfin/code_quality/projects/2

## I Combined MoscowValueRisk Priorities

https://github.com/aliveSurfin/code_quality/projects/3

## J T-Shirt Sizing

https://github.com/aliveSurfin/code_quality/projects/4

## K Source Code

https://github.com/aliveSurfin/code_quality/blob/main/development/code

## L Left Most Derivation Diagram

## M Right Most Derivation Diagram

## N Code Editor Initial Designs

## O Parser

https://github.com/aliveSurfin/code_quality/blob/main/development/code/parsing/parser/Parser.js

## P AST Types

https://github.com/aliveSurfin/code_quality/blob/main/development/code/parsing/parser/AST_CONST_TYPES.js

## Q Tokeniser Folder

https://github.com/aliveSurfin/code_quality/tree/main/development/code/parsing/tokenizer

## R Parser Tests

https://github.com/aliveSurfin/code_quality/tree/main/development/code/parsing/parser/tests

## S Development Diary

https://github.com/aliveSurfin/code_quality/blob/main/development/diary/diary.md

## T Evaluation Features

https://github.com/aliveSurfin/code_quality/tree/main/development/code/parsing/evaluate

## U Final Product

https://code-quality-honours.herokuapp.com/

## V Code Coverage

## W Interview Notes

## X User Testing Results