

CHS2524 - INDIVIDUAL PROJECT  
UNIVERSITY OF HUDDERSFIELD

---

# SOFTWARE QUALITY METRICS

---

JACK TIMBLIN  
U1051575

May 6, 2014

# Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
1.1	Problem Brief . . . . .	2
1.2	Users & audience . . . . .	3
1.3	Initial time-plan . . . . .	3
<b>2</b>	<b>Research</b>	<b>4</b>
2.1	Introduction . . . . .	4
2.2	What is 'Software Quality' . . . . .	4
2.2.1	McCall Software Quality Model . . . . .	4
2.2.2	Boehm Software Quality Model . . . . .	6
2.2.3	ISO 25010 . . . . .	6
2.2.4	Comparison of Software Quality Models . . . . .	9
2.3	A brief history into programming languages . . . . .	9
2.4	Software Quality Metrics . . . . .	10
2.4.1	Initial Ideas . . . . .	10
2.4.2	Existing Tools . . . . .	12
2.4.3	Metrics Investigation . . . . .	14
2.5	Research into tools and skills required . . . . .	16
<b>3</b>	<b>Requirements Specification</b>	<b>17</b>
3.1	Programming Language . . . . .	17
3.2	Tools . . . . .	18
3.3	Skills/Knowledge . . . . .	18
3.4	User Requirements . . . . .	19
<b>4</b>	<b>Design</b>	<b>20</b>
4.1	Development methodology . . . . .	20
4.1.1	Agile Method . . . . .	20
4.1.2	Waterfall Method . . . . .	21
4.1.3	Method Choice . . . . .	21
4.2	Choice of Metrics . . . . .	22
4.2.1	Complexity Metrics . . . . .	22
4.2.2	Comment Quality Metrics . . . . .	24
4.2.3	Readability Metrics . . . . .	24
4.3	Choice of Supported Programming Languages . . . . .	25
4.3.1	Structure of the 'hello' Language . . . . .	26
4.4	Testing Strategy . . . . .	26
4.5	Design Documents . . . . .	29
4.5.1	Activity Diagram . . . . .	29
4.5.2	Use case Diagram . . . . .	30
4.5.3	Class Diagrams . . . . .	30
4.6	Design Evaluation . . . . .	42
<b>5</b>	<b>Implementation</b>	<b>43</b>
5.1	Determining the Source Language of a File. . . . .	43
5.2	Initialise Generic Parser/Lexer Instances . . . . .	47
5.3	Initialising the Listener Instance. . . . .	48
5.4	Handling errors while Parsing a File or if the File is Unsupported. . . . .	48

5.5	Implementing the metrics . . . . .	49
5.5.1	cyclomatic Complexity Implementation . . . . .	49
5.5.2	Comment Ratio Implementation . . . . .	49
5.6	Generating User-Readable Output . . . . .	49
<b>6</b>	<b>Testing</b>	<b>50</b>
6.1	Introduction . . . . .	50
6.2	Unit Testing . . . . .	50
6.2.1	Test Suite . . . . .	50
6.2.2	Application Tests . . . . .	61
6.2.3	Analyser Tests . . . . .	61
6.2.4	Language Detection Test . . . . .	62
6.2.5	Parse Tree Construction Tests . . . . .	62
6.2.6	Syntax Error Adapter Tests . . . . .	63
6.2.7	Metric Initialisation Tests . . . . .	64
6.2.8	Notifying metrics When a Rule is Matched Walking the Parse Tree . . . .	64
6.2.9	Unit Tests Output . . . . .	67
6.3	System Testing . . . . .	67
6.3.1	Testing with valid input considered to be good quality . . . . .	68
6.3.2	Testing with valid input considered to be poor quality . . . . .	71
6.3.3	Testing with input that contains syntax errors . . . . .	73
6.3.4	Testing with input that is unsupported . . . . .	74
<b>7</b>	<b>Evaluation</b>	<b>74</b>
7.1	MoSCoW Requirements . . . . .	74
7.2	Product Evaluation . . . . .	75
7.3	Project Evaluation . . . . .	76
<b>8</b>	<b>Running the Application</b>	<b>77</b>
	<b>References</b>	<b>79</b>
<b>A</b>	<b>Tables</b>	<b>80</b>
A.1	Borland Metric Table . . . . .	80
<b>B</b>	<b>Version Control Commit Log</b>	<b>81</b>
<b>C</b>	<b>Sample output</b>	<b>88</b>
C.1	Overall Output . . . . .	88
C.2	Output on Single File . . . . .	89
C.3	Unsupported Files Output . . . . .	90
C.4	Reporting Syntax Error Output . . . . .	90
C.5	Overall Output - Multiple Languages . . . . .	90

# 1 Introduction

## 1.1 Problem Brief

The aim of this project is to design and implement a software “quality” analyser which will be used as a generic analyser to work with multiple programming languages.

Hugh Osborne states that:

*“Exactly what makes a piece of software an example of good programming is debatable, but there are certain signs that are clear indicators of bad programming, such as a lack of comments, bad layout, poor variable names, poor programming structure and poor code reuse.”* (Osborne, 2008)

This explains the idea of “quality” code can be quite vague and this becomes even more abstract when you think about multiple programming languages, as syntax and layout can change dramatically.

There are however some common elements to most programming languages that can be evaluated to essentially get a better degree of understanding of how good a piece of programming is, these are:

- Comments
- Layout
- Variable names
- Programming structure
- Code reuse

So these will be the factors that will be focused on when it comes to determine the overall quality of the software using parsing and lexical analysis.

Another problem that will have to be identified will be to determine a “marking” mechanism to display how “well” programming code conforms to software quality metrics.

## **1.2 Users & audience**

As Hugh Osborne is the client in this project, he will be the sole user for the finished project. This will be mainly used for the marking of student’s coursework where the overall “quality” of the programming is part of the marking scheme, such as the AP&D module taken by second year students.

## **1.3 Initial time-plan**

In the 30 weeks that are allocated in which to complete this project, it will be broken down as follows:

### **0-7 weeks**

This is the current stage in the project and have completed my Project Proposal. The preliminary research which was used to write up the project proposal will also be extended.

### **8 – 12 weeks**

During this stage of the project, the research will be completed and the initial project specification will also be written up which includes taking on a development methodology and also

creating all of the UML and design documents in order to start developing the product in the new term.

### **13 – 21 weeks**

During this stage of the development process, development of a base system will be well under way and have completed all of the main functionality completed, which includes parsing the programming code. At this point a testing strategy will also be developed in order to fix any errors that occur during the testing process. If time allows at the end of the project, a basic GUI will be developed to improve usability.

### **22 – 30 weeks**

At this point the final development stages will be completed and the testing will be undertaken. The final report will also be written up in this time period.

## **2 Research**

### **2.1 Introduction**

The research that needed to be carried out on this sort of project was quite vast. This not only includes attempting to determine a mark of quality on the code being analysed, but also the metrics that actually carry out the analysis to determine the rank.

Research into the history into different programming languages will also have to take place in order to get an idea of what are generic traits that are found in all of them in order to determine what metrics can be used in order to make my application as generic as possible with the time and knowledge available.

Research into different tools that are available will also be undertaken in order to generate a parse tree for a given programming language based on the research into generating parse trees and lexical analysis.

### **2.2 What is 'Software Quality'**

There are many different definitions to what is meant by software quality, and some of these will not be in the scope of this project. Obviously some of the main factors where already described above by Hugh Osborne. The models that also define it well are the models defined by McCall and Boehm and also the ISO 25010 standard.

#### **2.2.1 McCall Software Quality Model**

Jim McCalls software quality model was first introduced in 1977 and it was originally developed for the US Air Force. McCalls quality model has, as shown in figure 1 on page 5, three major perspectives for defining and identifying the quality of a software product, these are:

- **Product Revision**

Product revision is the ability to undergo changes which includes maintainability, flexibility and testability.

- **Product Operations**

Product Operations which define its operation characteristics and this covers correctness, reliability, efficiency, integrity and usability.

- **Product Transition**

Product transition is the adaptability of the product to a new environment and this includes portability, re-usability and interoperability.

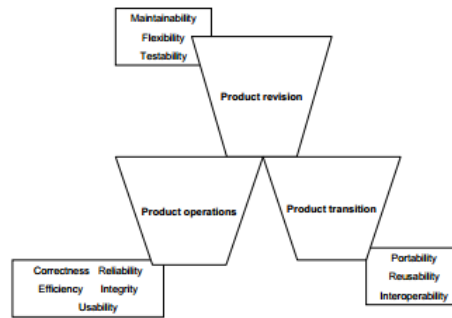


Figure 1: McCall's Quality Model

Figure 2 on page 32 describes the additional criteria that is needed for each of the 11 quality factors as defined by McCall, these are then defined in software quality metrics. Some of these are very relevant to the project and could be used to define the metrics that could be applicable to this project. These are:

- **Conciseness** - Obviously the smaller and more concise a piece of programming is, the easier it is to read and debug.
- **Modularity** - This is the partitioning of a complex software system in order to make it easier to debug and maintain. It also includes how "loosely coupled" the overall system is.
- **Simplicity** - Simplicity is the idea of keeping programming code as simple as possible. A good benchmark for this is to allow a method or class to have one single job.
- **Self-descriptiveness** - This is the most obvious criteria and probably the most important in terms of the actual written programming code. This is how many comments are used. However, this is a delicate balance of not enough or too many comments as being at either end of that spectrum will make the programming code more difficult to read and understand.
- **Efficiency** - Efficiency is important to this project in terms of measuring the order of how complex it is, in other words, trying to measure if the processing time exponentially increases if the scale of the problem trying to be accomplished increases. This will be mainly determined by how deep loop structures are and how many if-else control structures are used in the program.

### 2.2.2 Boehm Software Quality Model

Boehm's quality model is similar in design to that of McCall in that it also presents a hierarchical quality model that starts at high level characteristics and works down and splits up into smaller categories which all each contribute to the overall mark of "quality".

The high level characteristics in Boehm's model address three main question that the client has which are:

- **As-is Utility** - Which means how easily, reliably and efficiently can the application be used.
- **Maintainability** - Which is how easily the application is to understand, modify and re-test.
- **Portability** - Which is how how easily we can use this application in a different environment.

Again this model has some major characteristics, as shown in figure 3 on page 33, that would be applicable to the scope of this project, and these are:

- **Self Descriptiveness** - This is already defined by McCall's model as describes how easily you can read and understand the programming by the amount of commenting.
- **Legibility** - This is how easy the programming itself is to read and understand, by how long the class and methods are and also by the variable names.
- **Conciseness** - Again this is also shown in McCall's model where we want to make the the classes and methods as concise as possible, i.e to be able to complete it's task with as little programming as possible.
- **Structuredness** - This implies that the *"evolution of the program design has proceeded in an orderly and systematic manner, and that standard control structures have been followed in coding the program, etc"*.
- **Efficiency** - Again this is defined in McCalls model to test the order of complexity in order to determine that resources are not wasted.

Though Boehm's and McCall's models might appear similar, the difference is that McCall's model primarily focuses on the characteristics as defined in Boehm's "As-is utility" whereas Boehm's quality model is based on a wider range of characteristics. His model also focuses a lot on the software maintenance cost-effectiveness.

### 2.2.3 ISO 25010

The ISO/IEC 25010 is an international standard for the evaluation of software quality. It replaced the old ISO/IEC 9126 standard in 2011. The standard identifies eight software quality characteristics, each having sub-characteristics. As shown in figure 4 on page 33, the main quality characteristics are defined by the ISO as:

- **Functional Suitability** - *"degree to which a product or system provides functions that meet stated and implied needs when used under specified conditions"*
- **Performance Efficiency** - *"performance relative to the amount of resources used under stated conditions"*

- **Compatibility** - *"degree to which a product, system or component can exchange information with other products, systems or components, and/or perform its required functions, while sharing the same hardware or software environment"*
- **Usability** - *"degree to which a product or system can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use"*
- **Reliability** - *"degree to which a system, product or component performs specified functions under specified conditions for a specified period of time"*
- **Security** - *"degree to which a product or system protects information and data so that persons or other products or systems have the degree of data access appropriate to their types and levels of authorization"*
- **Maintainability** - *"degree of effectiveness and efficiency with which a product or system can be modified by the intended maintainers"*
- **Portability** - *"degree of effectiveness and efficiency with which a system, product or component can be transferred from one hardware, software or other operational or usage environment to another"*

(Reitsma, 2011) The ISO 25010 is the international standard on how to measure software quality. This means I should base my mark of quality on how well it conforms to this standard. Obviously, some of the criteria is not applicable for the scope of this project, for example, it is up to the programmer to correctly test and evaluate their programming code. This product will therefore only determine a mark of quality based on the actual syntax and structure used.

Functional suitability is obviously dependant on the initial requirements agreed upon the programmer and the client it will be delivered to. This is also true for each of the three sub-characteristics of functional suitability as shown in figure 4 on page 33. Obviously in this case the client will provide the marking criteria for what students have to provide. The requirements between Hugh and the student should not affect the functionality of this product at all and so the functional suitability is out of the scope of this project.

Performance efficiency determines how efficient the piece of programming is in accordance to the task it is required to complete and without the exact meaning of the program this analyser is evaluating this would be difficult to gauge. There are many different algorithms that we could use to compute a solution it would be extremely difficult to gauge the most efficient one because we know nothing of the problem domain, especially when the idea of this product is to give an automated mark on quality and not how well it completes the challenge set by the client.

Compatibility is essentially either how well programs interact, exchange information with other programs or the ability to perform its required functions which sharing the hardware or software environment with other programs. Trying to automate a mark of quality based on this criteria for software quality would again be extremely difficult firstly because we know nothing about the problem domain, and secondly the product in this sense would give a mark of the programming itself and not the environment that it was produced for. There could also be discrepancies in compatibility based on the environment the programming was developed and tested on and the environment the code is intended to be executed on depending on the programming language being used and the testing strategy that was carried out.



Usability, as defined in the ISO 25010, refers to the ease in which a piece of software can be used by the end user. Determining this from source code would be extremely challenging and frankly not in the scope of this project. This is because we can't reliably determine what the best designed graphical user interface would be and therefore usability cannot effectively be used to determine a mark of quality in an automated software quality testing sense. The decision on whether a GUI is acceptable is usually made by the client and they can quickly determine the ease of use of a piece of software by just using it.

Reliability of software is probably one of the most important characteristics that a piece of software should possess. After all no-one wants their software to crash or have bad performance issues etc. Again, in terms of automated software quality testing from the source code, there is not guaranteed way to calculate this. The definition of reliability according to ISO 25010 refers to not only the initial reliability of the product, but also the on-going performance and reliability of the software product. This is not only in the scope of the project and also not in the scope of what the client requires in this instance.

Security as defined by ISO 25010 is essentially how well a piece of software protects data entered into system and the degree of data access other systems or products have. This is a very important characteristic that a piece of software must have depending on what sorts of data the system holds, for example the systems in place in banks. This could be modelled based on the procedures and library methods that are put in used, but it would be extremely difficult as the language changed. We could also determine a mark on how well the system handles problems such as synchronised access to data etc, but again this would be change from language to language. In terms of what is needed for the client, this is out of the scope of the project as no personal data is entered or saved in the source code submitted as coursework to Hugh.

Maintainability is very valuable to code quality. If source code is not properly maintained then it would quickly become outdated and unusable and fixing problems would be a lot more difficult. Maintainability has several sub-categories which would be applicable to this project, these are:

1. **Analysability** - This is how easy faults can be found and fixed. This will mainly be down to how easy the source code is to read and understand which is one of the main factors I will be looking for during the automated testing process.
2. **Reusability** - This is basically down to reducing the amount of duplicated code in the source code, which is one of the requirements from the client and will be covered in this project.
3. **Modularity** - Code should be adaptable and parts of it should be able to be changed easily.

The other sub-categories are still good indicators to whether a piece of source code is of good quality or not, but are not really applicable to this project. The main one of these is testability, and as stated above it is really up to the initial developer to make sure that their code is properly tested using appropriate methods.

Portability also has some importance to this project. As stated above, code should be adaptable. This links into the idea of modularity as defined in maintainability. The ease in which software can be executed on different platforms is definitely not part of this project and should be tested by the developer as part of their testing strategy.

#### 2.2.4 Comparison of Software Quality Models

All three of the different models discussed above give a solid idea of what is meant by software quality. They all show that software should be reliable and easily modifiable in the future. All of the models also require that the code is readable, concise and understandable, i.e relevant commenting, appropriate variable, class and method names etc. They also require that the code is as simple and concise and not overly complex, not only so it is easier to understand and read but also because of the performance issues that can occur with overly complex code execution.

### 2.3 A brief history into programming languages

In order to understand some of the generic patterns that occur in different programming languages research in to the history of programming languages had to take place. The first major language appeared in 1957 in the form of FORTRAN (which stands for Formula Translation). The language was designed by IBM to be used for scientific computation. It was a very simple language with only included the statements IF, DO and GOTO. The substantial thing about this language was that it had a compiler, which was used to convert the language syntax into machine code which the computer to use to execute the program. FORTRAN introduced some of the basic data types which are used in most programming languages, these included *"logical variables (TRUE or FALSE), and integer, real, and double-precision numbers"*. (Ferguson, n.d.)

After FORTRAN, one of the next major programming languages to be released was COBOL (which stands for Common Business Oriented Language). The language was developed in 1959 and it was designed to be used for developing business, typically file-orientated, applications and is not designed for writing systems programs. In COBOL there are only three data types used, which are numeric, alphanumeric and alphabetic and it is not a strongly typed language. The main feature that was introduced in COBOL was the idea of "group items" which are the COBOL equivalent of data structures.

In 1958, John McCarthy of MIT developed the LISP (List Processing) programming language. It was one of the first procedural programming languages and was designed to used for artificial intelligence research. Linked lists are one of LISP's major data structures and LISP source code is itself made up of lists, which meant that it could manipulate source code as a data structure. LISP was the main influence behind some of the ideas in computing science including tree data structures and recursion.

The ALGOL (which stands for Algorithmic Language) is a family of computer programming languages which greatly influenced a lot of the programming languages that are in use today. There was three main specifications, which are:

- ALGOL 58
- ALGOL 60
- ALGOL 68

Although ALGOL never reached the level of commercial popularity of FORTRAN and COBOL, it is considered the most important language in terms of its influence in later language development. ALGOL's lexical and syntactic structures became so popular that virtually all languages designed since have been described as being "ALGOL-like". In ALGOL 60 the block structure was introduced, which gave programmers the ability to create blocks of statements and

introduced the idea of variable & global scope as shown in figure 5 on page 34. ALGOL also introduced structured control statements including IF, THEN and ELSE and iteration using for.

Some of the other characteristics that were introduced in ALGOL which are:

- Dynamic Arrays
- Reserved Words - the symbols used for keywords are not allowed to be used as variable or procedure identifiers by the programmer.
- User defined data types - allow the user to design data abstractions.

As stated above, the design of ALGOL was the main influence to a lot of the programming languages that are in use today, from this languages such as Pascal and C were developed and then moving onto even more advanced languages such as Python, C++, Ruby and even Java. Even though some of the syntax is different in each of these different languages, you can see that they all still adopt the "ALGOL-like" style of using blocks of statements and we can use this knowledge in order to conceive some sort of idea of how we can get a idea of quality based on the fact that a lot of today's programming languages adopted the same style.

Obviously, this is really only true to languages that have been influenced by ALGOL, languages influenced by the LISP programming language, for example Prolog, not only have a different syntax, but also a different structure to their source code, and frankly, time cannot be spared to include this functionality into the design so we will be only focusing on the languages that have an "ALGOL-like" structure.

## 2.4 Software Quality Metrics

Even though the software quality models mentioned above give us a good indication of how we determine software quality, we now need a way for this product to generate a "mark" based on these factors. This is where we can use software quality metrics.

The ideas of what metrics to implement in this project where collected from multiple sources. Obviously this choice had to take into consideration that the product should be able to support multiple languages.

### 2.4.1 Initial Ideas

Some of the initial ideas came from the project proposal. Hugh Osborne lists a lot of ideas which suggest factors that could contribute to a piece of source code being of poor quality. As stated above, these indicators included:

- Comments
- Layout
- Variable names
- Programming structure
- Code reuse

There are also other indicators that could affect the quality of a piece of programming, for example method signature length or the complexity of the code.

The comment ratio is a very good indicator of good programming practise. If there is little or no comments in the code, then this makes it extremely difficult to read and understand. The same could be said at the opposite site of the spectrum where if there is too much commenting, then the code takes a lot longer to read and comment duplication could also occur making the source code extremely long. In most programming languages that are "ALGOL-like" the comments will be anything after a `"/"` on a single line, or anything between `"/*"` and `"*/"` so the amount of commenting can easily be determined by simple lexical analysis. I guess we could also take into consideration what the comments should contain as one thing that shouldn't happen is that comments should not repeat what is already indicated in the code.

Hugh Osborne also indicates that source code should have the correct layout. Obviously in some programming languages this will not even be an issue, such as Python which depends on the indentation and layout of the code to compile and execute correctly. This formatting should be done so that the code is easy to read and understand. It should also be the case that the scope of each block section is clear and understandable. This therefore should be tested as part of the metrics used in this project. This would also include such factors as if all of the lines of code can fit on a single line on a single screen, the length of variable names, the number of lines of code, etc. All of these factors reduce readability and understandability.

The idea of good variable names is also given. Hugh Osborne gives the idea that variable names should be the concatenation of English word(s) and that they also should not be overly long. This seems sensible, after all writing good variable names makes it easier to understand what they are used for and what affect they have as simply calling a variable a random set of alphanumeric characters doesn't really give a good description of what that variable is used for.

On the other hand there will be variable names such as `i`, `n`, `x`, etc that will be okay when used for controlling a loop structure such as a `for` or `while` loop. This will definitely be the case when iterating over a data structure where the variable is only used as the index in that said data structure. So obviously checking if the variable has been defined in a loop structure would give us a good indication of what that particular variable is being used for and we can disregard any variable names such as those mentioned above (`i`, `n`, `x`, etc) in our calculations.

The final idea that Hugh gives us in the initial proposal is the importance of code reuse. It is extremely poor and inefficient to have code duplication in source code. Hugh Osborne suggests that that any code that is duplicated after a several number of lines should be placed in its own procedure. There are several suggestions in which we could detect code duplication in programming which are:

1. **Rabin-Karp Algorithm** - which is a string search algorithm that uses hashing to find any one of a set of pattern strings in a text. By using hashing, it can ignore details such as case and punctuation.
2. **Using Abstract Syntax Trees** - in which we can compare sub-trees to determine whether there is code duplication.

As stated above, there are also other qualities in a piece of source code that determine whether it is of good quality, one of which is the idea that long method definitions could reduce readability.

In some programming languages, such as Java and C#, the method signature could be long, not because they are poorly written but because they have the possibility to throw exceptions. There are features of a method signature that are language independent and we can use them to determine whether it is too long. These are the length of the method name and the amount of parameters that are required which can extend the length of the signature.

#### **2.4.2 Existing Tools**

Before carrying out research on the metrics that would be considered for implementation, we should see what other tools are available in the industry to get an idea of what metrics are currently in use and also determine what metrics should be implemented based on the quality factors found in the models above. After carrying out some initial research, It was found that the majority of tools are only really designed to evaluate Java code, but there are tools that can analyse code that is written in a different source language.

The tools that will be analysed in this section will be:

- CCCC
- SourceForge Metrics
- Ndepend
- Borland Together

##### **CCCC**

CCCC or C and C++ Code Counter is a free tool for measurement of source code related metrics developed by Tim Littlefair. It is used as a command-line tool that generates reports on various metrics including LOC (Lines Of Code) and metrics proposed by Chidamber & Kemerer and Henry & Kafuna. A screen-shot of the application running under windows is shown in figure ?? on page ?? . I could not get the command-line version of the tool correctly functioning because it was such an old tool and was no longer correctly supported under Linux or Windows platforms.

##### **Source-forge Metrics**

The source-forge tool is an eclipse metrics plug-in (<http://metrics.sourceforge.net>) which carry out tests on Java source code in order to determine a "mark" for the quality of the code based on the 23 metrics the tool uses. These are shown in the table below.

Metric Name	Description
Number of Static Methods (NSM)	Total number of static methods in the selected scope.
Lines of Code (LOC)	Total lines of code in the selected scope.
Afferent Coupling (CA)	The number of classes outside a package that depend on classes inside the package.
Normalized Distance (RMD)	$RMA + RMI - 1$ , this number should be small, close to zero for good packaging design.
Number of Classes (NOC)	Total number of classes in the selected scope.
Specialization Index (SIX)	Average of the specialization index, defined as $NORM * DIT / NOM$ . This is a class level metric.
Instability (RMI)	$CE / (CA + CE)$
Number of Attributes (NOF)	Total number of attributes in the selected scope.
Number of Packages (NOP)	Total number of packages in the selected scope.
Method Lines of Code (MLOC)	Total number of lines of code inside method bodies, excluding blank lines and comments.
Weighted Methods per Class (WMC)	Sum of the McCabe Cyclomatic Complexity for all methods in a class.
Number of Overridden Methods (NORM)	Total number of methods in the selected scope that are overridden from an ancestor class.
Number of Static Attributes (NSF)	Total number of static attributes in the selected scope.
Nested Block Depth (NBD)	The depth of nested blocks of code.
Number of Methods (NOM)	Total number of methods defined in the selected scope.
Lack of Cohesion of Methods (LCOM)	A measure for the Cohesiveness of a class. Calculated with the Henderson-Sellers method. A low value indicates a cohesive class.
McCabe Cyclomatic Complexity (VG)	Counts the number of flows through a piece of code. Each time a branch occurs this metric is incremented by one.
Number of Parameters (PAR)	Total number of parameters in the selected scope.
Abstractness (RMA)	The number of abstract classes (and interfaces) divided by the total number of types in a package.
Number of Interfaces (NOI)	Total number of interfaces in the selected scope.
Efferent Coupling (CE)	The number of classes inside a package that depend on classes outside the package.
Number of Children (NSC)	Total number of direct subclasses of a class.
Depth of Inheritance Tree (DIT)	Distance from class Object in inheritance hierarchy.

This tool gives me a good indicator to what metrics could be used to match the quality characteristics tested in the product. The metrics that seem useful to me are LOC, MLOC, NOM, VG and PAR. The only major problem that can be seen with this tool is that even though it performs a good number of tests, the numerical output is extremely hard to read and understand and gives no real indication to the overall quality of the code. This tool does not even consider any metrics regarding how much commenting has been done.

This is shown in figure 6 on page 34, which is a screen-shot of a java project which is used to sync a database connection. It has a lot of code-duplication (even though this tool does not necessarily perform tests for that.) and the cyclomatic complexity is also high because of the amount of while and if statements used in the application, but because this is the nature of the application there is no way around this.

## **Ndepend**

Ndepend is a commercial code quality analyser for C# and Visual Basic source code. This and Borland Together seem to be the most elaborate and complete. Ndepend however gives a full structured report, in the form of a HTML page, to how well the project performed. It also only shows the metrics that failed with a table of results as shown in figures 7 and 8 on page 35. This gives me ideas on how the output could be presented by the application in such a way that it is easy for the end-user to see how well the code performed.

In total Ndepend has 82 metrics, which are separated into 5 sections, that it uses in order to determine how well source code is written and the documentation can be found at <http://www.ndepend.com/Metrics.aspx>.

## **Borland Together**

The last tool that was examined was the Borland Together tool. Borland Together is an eclipse plug-in and can also be used as a Modelling tool. The metrics provided by Borland are displayed in the table in appendix section A.1 on page 80.

Its not a surprise that Borland surpasses any of its the open-source counterparts in the amount of metrics that it provides, but it does confirm the choices that I have made in the earlier tools and also gives me ideas about additional metrics that could be of interest for this project. The metrics for calculating the comment ratio will also be very useful in this project.

## **Evaluate this product with existing tools.**

Obviously, my product will only be a proof of concept application, but it will have the capability for easy expansion based on the modular design it will have. One of the main advantages of this product over any other product on the market is that it is the only one that will be able to easily add support for new languages, and it also can be used as an external library in form of a jar file. Another plus for this product over any other existing tools is that this one is will be open source which in comparison to something like Borland Together which costs a lot of money in order to use. But on the other hand Borland Together doesn't only handle metric analysis.

Overall, this product won't be as well defined as any of the other tools mainly because of the time constraints and resources that will be going into this project, but it will be able to handle multiple languages with the capability for huge expansion which means, if continued to be developed, could make this tool a real competitor. The expansion of this tool would also be a good research project for a masters.

### **2.4.3 Metrics Investigation**

After looking at the tools that are available and determining what metrics would be applicable to use in this project, research into how to determine the different metrics from source code was the next logical step.

Obviously, some of the metrics defined only really require simple lexical analysis, like checking the comment ratio and also determining the number of lines of code etc. However all of the other metrics require generating a parse tree for a given language in order to determine structure, variable and method calls etc. But again, after generating the parse tree, some metrics don't really require that much calculation in order to determine a good mark of quality. These include checking method signatures, method/class length and also checking that the variable names are

acceptable.

There are several metrics that we can use to attempt to gauge the complexity of a program, the most common of which is McCabe's Cyclomatic Complexity. Complexity is *"inferred by measuring the number of linearly independent paths through the program. The higher the number the more complex the code."* (Naboulsi, 2011) So simply put the more decisions that have to be made, the more complex the code, i.e the amount of if, else, while, switch statements in the code. Obviously the syntax to be checked for would be slightly different in different languages, but since we are focusing on "ALGOL-like" programming languages, this should not be too much of a problem.

As stated in the NIST Special Publication 500-235:

*"There are many good reasons to limit cyclomatic complexity. Overly complex modules are more prone to error, are harder to understand, are harder to test, and are harder to modify. Deliberately limiting complexity at all stages of software development, for example as a departmental standard, helps avoid the pitfalls associated with high complexity software. Many organisations have successfully implemented complexity limits as part of their software programs. The precise number to use as a limit, however, remains somewhat controversial. The original limit of 10 as proposed by McCabe has significant supporting evidence, but limits as high as 15 have been used successfully as well. Limits over 10 should be reserved for projects that have several operational advantages over typical projects."* (Wallace, 1996)

This shows that limiting the amount of complexity in a program does reduce the risk of errors and also makes the source code easier to modify and maintain, and not increasing the limit by 10 unless there is special circumstances. This metric would therefore be useful to implement in this project as reduced complexity does lead to higher quality code overall.

The only criticism that was found with McCabe's cyclomatic complexity metric is that it does not take nesting into consideration. An alternate metric to use would be Prather Metric.

The Prather metric has essentially the same idea as McCabe's because it describes the flow through a program, but it also takes looping structures into account and the effect this has on the complexity of a program. However, applying McCabe's limit of complexity, which is 10, to a metric like this would be ridiculous as any part of code that is inside a loop structure is multiplied by the depth to which the statement is nested instead of incrementing the value when a decision point is discovered. I feel that carrying out both metrics would be more beneficial in order to have a more accurate gauge of complexity, but in terms of this project, calculating McCabe's cyclomatic complexity will give us a good indication of how complex a piece of code is, and this seems to be more widely used metric in the tools evaluated above.

Even though the Prather Metric is impractical to implement in this project, the idea of checking the level of nesting that occurs is however an important quality factor. This is because source code with high levels of nesting becomes a lot harder to read and understand. It also can lead to reduced performance as you get loop structures inside of loop structures etc.



## 2.5 Research into tools and skills required

At this point in the report, the discussion of the tools and skills that required in order to undertake a project such as this will occur.

### Parser/Lexer Generation Tools

The first requirement of this task is to actually generate the parse tree on different languages in order to be able to perform metric analysis on it. Obviously we could implement this ourselves, but that would be a final year project in its own right without even performing any analysis on the result. Because of this it was decided to carry out some research into universal parse tree generation tools, and the one most prominent was Antlr.

Antlr (Anorther Tool for Language Recognition) is a parser generator for reading, processing, executing or translating structured text or binary files. Antlr depends on a grammar on which it can generate a parser and lexer in Java for the target language. So this tool can be used on multiple languages on the condition that we have the specified grammar for Antlr to use.

Using a tool like this gives us the freedom to be able to easily add support to new languages by providing Antlr with a grammar in which it can generate a lexer and parser for. Obviously we would also have to create a testing strategy to ensure that the new language is supported by the metrics. There are loads of open-source grammars which are currently being hosted on Github by the Antlr community, this is located at <https://github.com/antlr/grammars-v4>.

There also has been a remarkable add-on to Antlr called AntlrWorks 2 (<http://tunnelvisionlabs.com/products/demo/antlrworks>) which is a GUI for designing and testing grammars in Antlr, we can also test and run the generated parsers and lexers to test they correctly generate a parse tree for a given grammar.

### Integrated Development Environment (IDE)

Obviously, this already means that Java will be the language that this product will be implemented in as Antlr generates a parser and lexer in Java. This means that research on a suitable IDE to implement my designs needs to take place. The two obvious choices in this case would be either NetBeans or Eclipse as they are pretty much the industry standard.

#### Eclipse

Eclipse is pretty much the standard when it comes to development. The latest release (4.3.1, Kepler) supports a variety of different languages. One of the main advantages to using eclipse over other IDE's is the amount of plug-ins that are available which offer different functionality from adding new language syntax support to adding UML support, and Borland Together which is evaluated above, is a good example of this. Eclipse actually provide a marketplace for plug-ins which can be used at this is located at <https://marketplace.eclipse.org/>.

#### NetBeans

NetBeans is an community based IDE that is sponsored by Oracle. The latest stable release 8.0 supports the development of desktop and web applications with support for PHP, Java and C/C++. Just like Eclipse, NetBeans has plug-in support. Even though the overall feel and usability of NetBeans is a lot easier than that of Eclipse it does lack in some areas. It doesn't have no where near the same level of plug-in support that eclipse

has and nearly all of the plug-ins are open-source.

### Version Control

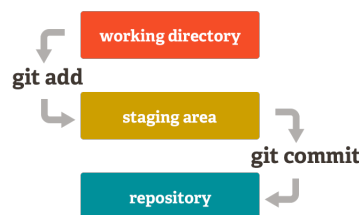
Even though this is not a vital requirement for part of this project, having this product under a form of version control not only acts as a back-up, but it also acts as a constantly evolving log each time a commit is made. The two main Version Control applications available are Subversion and Git. Both of these version control systems allow you to commit work and update the code base and also leave a message to what has been changed, however Git has some major advantages over Subversion.

### Branching & Merging

Branching and Merging in Git is the feature that makes it stand apart from the rest. It allows you to be able to develop a new idea away from the main code base, and be able to commit it for other contributors to work on and if the idea works it can be merged into the main branch, or you can rollback to the existing branch. This is good when developing experimental code onto of a tested code-base and reduces the risk of damaging the whole application's code-base.

### Staging Area

Git is the only source control manager that has what is known as the "staging area". This is the area commits are initially sent to so that all of the commits made can be reviewed before they are permanently committed to the repository.



### Online Repositories

There are also repositories that can be hosted on-line such as GitHub (<https://github.com/>) and BitBucket (<https://bitbucket.org/>). These are invaluable because they allow the community to work on projects from anywhere in the world. These repositories are also good for projects like this as they have the functionality to allow private repositories. Having a repository on-line not only allows me to work from anywhere, it also acts as a on-line back-up and log of the work that I have done.

## 3 Requirements Specification

The explanation and justification of the choices of tools and skills that will be used to complete this project in explained in this section as well as including the user requirements.

### 3.1 Programming Language

Obviously the three main choices are Python, C# or Java as these are the three programming languages that I have the most experience in. The choice that I decided to make was Java, partly

because I have the most experience using this language during my studies in University, from Android applications to distributed systems using JINI and CORBA. It also made it an even easier choice as the Parser generation tool outputs a parser in Java.

Another good reason to use Java as the implementation language is that it is a very easy language to execute regardless of what platform it is running on, so this automatically makes my product platform independent.

### 3.2 Tools

The tools that will be used in this project will obviously consist of a Parser/Lexer generation tool, an IDE and some sort of Version Control. The parser generation tool that will be used is obviously Antlr as it is by far the best generation tool, and makes my application easily expandable by just supplying a grammar to support new languages. Not only this but it can also generate simpler grammars to pick out attributes that are applicable to all "ALGOL-like" languages, for example, to gauge how much nesting that occurs in a language we just need to count the amount of opening/closing braces that occur inside one of another, this is shown in figure 9 on page 35 which has a nesting depth of 3.

The IDE has to be able to model out the designs for this product using the Unified Modelling Language and also be able to have a facility to debug and test my code effectively. As Java will be used as the implementation language, the research into IDE's was based around this. The IDE that will be used is Eclipse, again because it is the one I have the most experience with and also because of the Borland Together modelling tools which means I can easily model my application using UML and create a "template" project based on the class diagram. The JUnit test suite can also be used to carry out Unit Testing on my application as well as have a complete debugging perspective which means we can effectively walk through the code at breakpoints to fix errors.

The version control system that will be used is Git, mainly because we can store this project on-line in a private repository which essentially acts as a back-up of my code that can be retrieved at any location. As stated above, this will also provide a log of all the changes during development and this log will be attached to the appendix after completion.

There are some other miscellaneous tools that will also be used as part of this project and need mentioning. Obviously, part of this project is to make my product usable on multiple platforms and this will need testing. This will be done on a variant of Linux running as a virtual machine, using the open source implementation Oracle Virtualbox which is available at <https://www.virtualbox.org/>. The variant of Linux that will be used is Ubuntu available at <http://www.ubuntu.com/>. Testing will also be carried out on Windows which is available on the computers at University.

### 3.3 Skills/Knowledge

There are also some skills that will be required during the course of this project, these will not only include parsing & lexing the source code. We firstly and most obviously have to have knowledge in a variety of programming languages. This is not only so we can write the application, which as stated above will be in Java, but also to write tester code in other programming languages to determine if my application can actually handle other languages.

Apart from this we will also have to carry out testing using the JUnit testing suite and the Unified Modelling Language, which will be again vital to the success of this project in terms of designing and testing the application.

As stated above in section 3.2, the Git version control system will be used in order to track the status of this application throughout the development and testing processes. This will in turn need some knowledge in how to use Git and more importantly, the Git commands in order to actual stage and commit changes and revert changes to the repository when required. This also includes pushing my commits to a remote private repository on GitHub (<https://www.github.com>).

Finally, The use of bash scripts and windows command prompt scripts will be required in order to use the application on different platforms.

### 3.4 User Requirements

The product that will be produced during the development of this project will be a proof of concept for the idea of generating a 'mark' of quality for input source code in a variety of different languages. The criteria that determines whether a piece of source code is deemed as high quality has been determined in section 2.4 on page 10.

The idea of this product is to design and implement it as such a way that it becomes easily expandable if it happened to be pursued in an commercial environment with more resources, or as a possible masters thesis. The specifics of the functional requirements and User interface requirements are as follows:

#### Must

1. To complete research into metrics and parsing techniques.
2. To select appropriate metrics to implement in this project.
3. To allow the system to be developed to be adaptable to different languages.
4. To allow for further expansion by adding new metrics.

Items 2 & 3 in this section describe the way that this project is going to have a modular and expandable design. This is where each metric will be designed as a module and additional metrics can be 'plugged-in' to the system. This is the same with adding additional language support, because of the way that the system will use Antlr or generate a parser and lexer from a grammar. This means that adding additional grammars to use automatically gives support to a new language.

The main area that needs to be completed is that of generating a parse tree from source code, as actually applying metrics to the source depends on this so this should be tackled first.

Using this strategy will help to cope with the effects of being over ambitious during development. This also means that if development takes longer than expected we can scrap less valuable metrics or drop support for less valuable programming languages, but because of the design on the system they have the potential to be added at a later date. The metrics that will be implemented as part of this project are described in section 4.2 on page 22.

### Should

1. To draw the results from these metrics to give a rating of the program code quality for a given piece of source code.
2. To be able to indicate where the scores were derived from in order to give an indicator of what needs to be improved.
3. To be usable on multiple platforms i.e Windows or Linux.
4. To produce suitable output, which is easy to read and understand to the end user.

These points should definitely be taken into consideration, but they depend on the points above to be completed beforehand.

### Could

1. Automatically improve the layout of code based on flaws found by running the metrics.

### Won't

1. Develop a Graphical User Interface for the product.

The application that will be developed in the course of this product will run from the command line as developing a GUI for this application adds a lot more complexity to the application which I quite frankly don't have the available time to complete. This is mainly due to not only adding extra components to design and test, but also the amount implementation that goes into creating a robust and user-friendly interface. But again, if the product could be implemented in such a way to allow for such functionality in the future this would be welcomed.

## 4 Design

### 4.1 Development methodology

Choosing a suitable development methodology to use in the project is vital to its overall success. There are two suitable methodologies that we could use in the development of this application and these are the agile and waterfall methods.

#### 4.1.1 Agile Method

The agile method proposes alternatives to traditional project management. Agile approaches are typically used in software development in order to respond to unpredictability (*Agile Methodology*, n.d.). The Agile development methodology provides opportunities to assess the direction of the project throughout the development life-cycle. This is done by introducing development iterations in which a potentially shippable product is delivered at the end of each cycle. Using iterations rather than the traditional waterfall approach can be highly advantageous as it means that the development team can adapt to any changes in the requirements easily and implement these changes in the next iteration. An iteration usually has the same development life-cycle as the waterfall method as shown in figure 10 on page 36 in which it takes requirements and then designs, implements and tests the product. The only change is that in an agile method this is

usually done in short sprints but with multiple iterations.

The agile method means that we can also take a more modular approach to product development. As in each iteration we can focus on specific parts of the application. An example in terms of this project, we could focus on the parser generation in one cycle, but then focus on a specific metric in another.

This approach however is only advantageous if the requirements are likely to change throughout the development process. If this is not the case, than using this method may actually be slower than using the traditional waterfall method.

#### 4.1.2 Waterfall Method

The waterfall method is the complete opposite of the agile method, in which you can't go back to any milestone above. i.e you can't go back to the design phase once implementation has begin and you can't go back to implementation when verification is done.

The advantages of using the waterfall method are:

- Design errors are captured before any software is written saving time during the implementation phase.
- Excellent technical documentation is part of the deliverables and it is easier for new programmers to get up to speed during the maintenance phase. The approach is very structured and it is easier to measure progress by reference to clearly defined milestones.
- The total cost of the project can be accurately estimated after the requirements have been defined (via the functional and user interface specifications).
- Testing is easier as it can be done by reference to the scenarios defined in the functional specification.

The disadvantages of using the waterfall method are:

- Clients will often find it difficult to state their requirements at the abstract level of a functional specification and will only fully appreciate what is needed when the application is delivered. It then becomes very difficult (and expensive) to re-engineer the application.
- The model does not cater for the possibility of requirements changing during the development cycle.
- A project can often take substantially longer to deliver than when developed with an iterative methodology such as the agile development method.

(*The Waterfall Development Methodology*, n.d.)

#### 4.1.3 Method Choice

After discussing the differences between the two methodologies, the conclusion that has been established is that in this case the best methodology to use would that of a waterfall methodology. Obviously, the fact that we have no prior knowledge regarding how long each part of the system will take to develop is a big disadvantage of using this method, however, the fact that we have a solid idea of what we are looking to achieve means that we can effectively design this product in

linear fashion because the clients requirements are not likely to change at the end of this project. So using an agile method over the more traditional waterfall method in this instance would in reality increase the amount of time that it would take to develop a complete system, and also reduce the amount of duplicate JUnit testing that we would do in each iterative cycle.

The only advantage that can be seen in using an agile method in this instance is that we have a working application at the end of each iteration which would mean that we could stop iterating at any given point and still have a fully working product that could be deliver to the client, which means we could control how much time we spend on the development.

In this instance, the best development methodology to use for this project would that of a waterfall methodology.

## 4.2 Choice of Metrics

The metrics that have been chosen for implementation as part of this project are listed below. These are 8 metrics that have been considered to be the most valuable in terms of determining code quality based on the traits that we are looking for based on the research carried out and the initial ideas that have been put forward by Hugh Osborne. Also it would be important to reiterate that the application will be implemented in such a way that additional metrics could be added in the future.

- **Complexity Metrics**

- Depth of nesting
- Weighted method count
- Cyclomatic complexity
- Method count

- **Comment Quality**

- Comment ratio

- **Readability**

- Procedure declaration length
- Lines of code
- Variable naming convention.

### 4.2.1 Complexity Metrics

#### **Depth Of Nesting**

Depth of Nesting is the amount of nesting that occurs throughout a class, this is measured in most languages by the amount of braces inside of each other. It can also be defined in other ways. Example of this are the amount of colons (':') followed by indentation on the next line that is used in the Python language, or the use of the keywords 'begin' and 'end' in the ALGOL language.

Obviously, the hardest part of implementing this metric is defining what the source language defines as a block or 'scope' of code, and this will have to be provided by the parser itself, or more specifically, the grammar that defines the language. If Antlr does not provide these facilities

this will involve parsing the grammar file independently or providing a separate options file with the grammar which defines the options for each metric. The second option would be the more logical choice, as this will be a lot easier to parse than a grammar file and some of the parsed grammar file will be redundant for a particular metric, meaning that we would be processing parts of the file needlessly.

This metric will determine the location where the most nesting occurs in a class. NDepend determines that a suitable value to determine good quality code is under 4 as if it is higher than 4 in any one instance of nesting, then the code becomes harder to read and maintain. (*Metrics Definitions*, n.d.)

### **Weighted Method Count**

The Weighted method count (WMC) metric was originally defined in A Metrics Suite for Object Orientated Design by Chidamber & Kemerer.

The WMC metric is defined as a sum of McCabe's Cyclomatic Complexity of all methods and constructors declared in a class. This metric is a good indicator of how much effort will be necessary to maintain and develop a particular class. A lower WMC score usually indicates to a class with better abstraction and polymorphism. While a class with a high complexity value is a good indicator that this class is very application specific and does more than one job, and therefore harder to test, reuse and maintain.

This metric will also require some information about the source programming language that it is evaluating. Firstly, it will need to know how a method and constructor is declared, which is for example 'public void hello()' in Java or 'def hello()' in Python. It will also need a list of all the keywords to search for to gauge the Cyclomatic Complexity of a given method or constructor. Even though McCabe's Cyclomatic Complexity will already implemented as part of this project, it would not be applicable to just use its methods to define this sum for WMC because this will destroy the modularity that metrics must possess as they should be able to be added and removed at any time.

An appropriate threshold for the WMC lower limit is 1, because a class should at least consist of one method. An upper limit for WMC of a class is harder to define, but as shown in section 2.4.3, the McCabe's Cyclomatic Complexity upper threshold is 10 for a single method, so it only seems realistic to use 50 as a good reference point for the sum of complexities for the class as a whole.

### **Cyclomatic Complexity**

Cyclomatic Complexity is used to determine how many paths there is through a program. This is measured as the sum of decision points that are used, for example in Java these include:

- for
- while
- if
- case
- default



- continue
- catch
- ?: (ternary operator)

As stated above, this metric will need information about the programming language that it is evaluating, as it needs to know what keywords to look for.

Methods where cyclomatic complexity is higher than 10 are hard to understand and maintain as stated in section 2.4.3.

### Method Count

This is the number of methods that a class contains. This method needs information on how a language defines a method from the parser.

NDepend documentation states:

*"Types where NbMethods > 20 might be hard to understand and maintain but there might be cases where it is relevant to have a high value for NbMethods. For example, the System.Windows.Forms.DataGridView third-party class has more than 1000 methods."*  
(Metrics Definitions, n.d.)

This states that a good threshold to use when determining if a class has too many methods is 20, but this does entirely depend on what the class is being used for.

## 4.2.2 Comment Quality Metrics

### Comment Ratio

This will be calculating the comment ratio of a class. Most languages take on the same form for defining a comment in a piece of source code. This is anything after a `'//'` and a new line and anything between `'/*'` and the closing `'*/'`. I think that this could be used as a default setting for this metric to determine how much of a file is commenting. Obviously there will be some languages that use different syntax in order to declare a comment and this can be defined in an options file which will override the default setting.

It seems reasonable to set the thresholds on the comment to code ratio in a piece of source code at between 20 and 40 percent of the overall text in the file.

## 4.2.3 Readability Metrics

### Procedure Declaration Length

This metric checks to see if any method declaration in a class is defined to be too long. This will be case that the name of the method is too long or that it requires too many parameters.

The threshold that Hugh gave for this metric would be if the declaration can fit one line on the screen. Obviously this metric will count all of the instances and save the position that this occurs so that some quality feedback can be given to the end user.

### Lines Of Code

Obviously, the amount of lines of code that a class has is also a good quality indicator because if the class is too long it becomes increasingly difficult to find bugs and update code. If a class is too long, it could also possibly mean that the class could be split up into smaller classes.

An open-source implementation of this metric, called CLOC (<http://cloc.sourceforge.net/>), could be used to initially count the lines of code in a class, as this implementation is already language independent. We could then use this figure to determine whether the class is too long.

### Variable Naming Conventions

Variable names in a class should represent what they are being used for. For example, if there is a boolean value that checks for validity, then 'isValid' would be an applicable name for that instance. Obviously being able to determine what the programmer is going to be using each variable for, especially if the variable names don't correspond to this, would be extremely difficult to accomplish.

However, as Hugh pointed out earlier on, we can check for standard naming conventions that should be applied to variable names to sure that they are not complete nonsense. This would include not only checking that the names are of concatenated English words and that they are camel-cased. We also stated that there are some instances where names such as 'i', 'j', 'k' etc would be valid, for example in a loop such as a for loop.

This should be one of the easier metrics to implement as no additional information is needed in order to gauge what the variable names are as this will definitely already be defined in the grammar.

## 4.3 Choice of Supported Programming Languages

There is a vast number of different programming languages that I could choose to support in this project. I have chosen just to support 2 languages as part of this project just to prove that the product can in-fact manage to run metrics on multiple programming languages.

The first language that I have decided to support is Java. This is because it is an already well-defined "ALGOL-like" programming language.

The other language that will be supported will be a simple language that will be defined as an ANTLR grammar which will be called the 'hello' language, mainly down to the typical 'hello world' use for simple applications and because it needed to be a language name that is not already defined in order to get stop getting incorrectly detected by the analyser.

The reasons behind defining a new language instead of choosing to support an existing language was for two reasons, which are:

1. We can demonstrate that the analyser truly supports any given language.
2. We are able to entirely define the parser/lexical rules of the language, which will ensure that the grammar is truly supported by the analyser.

The syntax of the 'hello' grammar will be simple and will be loosely based on any high-level object orientated language.

#### 4.3.1 Structure of the 'hello' Language

As stated above, the syntax and structure of the 'hello' language is loosely based on any typical high-level languages. A typical class can be defined as:

```
/* typical class commenting */
class Test {

    /* typical method commenting */
    void testMethod(tester1, tester2) {

        //typical inline commenting.
        try {
            if(tester1 <= tester2) {
                print(tester1);
            }
        } catch (exception) {
            print(exception);
        }

    }

}
```

As you can see the syntax is pretty similar to any high-level language. There are a few differences however, which are:

- There are no visibility modifiers.
- There are no object types.
- there are no parameter/field assignments.
- only if/else, switch, try/catch and method calls are supported.
- comments are written by using `/* */` or `//`.

Obviously as you can see, this language is extremely limited in functionality and has only really designed as a proof of concept to demonstrate that the analyser can support multiple languages.

#### 4.4 Testing Strategy

I will be taking a common approach to testing when it comes to making sure that my product is as robust as possible and does exactly as I have defined in the product specification above. This will be done by carrying out a series of JUnit tests on the different modules of the application to make sure that they all function as required and handle error appropriately. This will be carried out during development as well as a final stage of tests after completion.

After this I will also be carrying out system testing, in which I use the application as the client would using a series of test source code written in a variety of programming languages

to make sure that it performs as required. Obviously, this is the verification stage as defined in the waterfall methodology. Any changes will then be made after testing is complete, in the maintenance stage of the development process.

An outline of the tests that I will be carrying out are as follows:

### **JUnit Tests**

The majority of Unit Testing will be done during development, not only to test that a part works correctly after I have completed it, but also to check that adding extra code elsewhere doesn't effect existing components. The Unit Tests will be carried out on each module independently, as system testing will test that all of the components are correctly working together after development has been completed.

- **Parser & Lexer Generation** - Unit tests will be carried out on this part of the application to check the following requirements:
  - That a parser & lexer can be generated from the grammar provided, checks will also be carried out that determine that errors are handled correctly when the source code cannot be parsed from the grammar provided, for example when the source code has syntax errors, but as stated above making sure that the source code provided compiles is the responsibility of the programmer and not in the scope of this project.
  - That we can call methods of the parser to walk the generated parse tree and gain information about the source code.
  - That we can use the generated lexer to perform lexical analysis on the source code, this is required to determine things such as comment ratio etc.
  - That we can use the parser & lexer generator tool on source code written in different languages and still end up with a valid parser & lexer to use, or that we can handle error appropriately if this is not the case.
- **Metrics** - each metric will have the same tests performed on them as they all have the same requirements and unit tests will be carried out to check for the following requirements:
  - That a metric can perform its task successfully, for example method nesting count, or McCabe's cyclomatic Complexity checks. We will also test that these modules handle error correctly if it occurs.
  - That a metric can determine a mark of quality based on the analysis it performed on the source code.
- **Miscellaneous** - we also have some miscellaneous unit tests that we have to carry out on this application.
  - That we can output the results from analysis in a suitable format, possibly a HTML file.
  - That the application can determine an overall mark of quality based on all of the results from each of the metric modules.
  - That the application can start correctly and successfully use supplied source code as input, obviously if my application cannot take in source code, it won't ever produce a result.
  - Testing that the main application can find all of the required modules it needs on start-up, for example the parser & lexer tools as well as all of the different metrics.

### **System Testing**

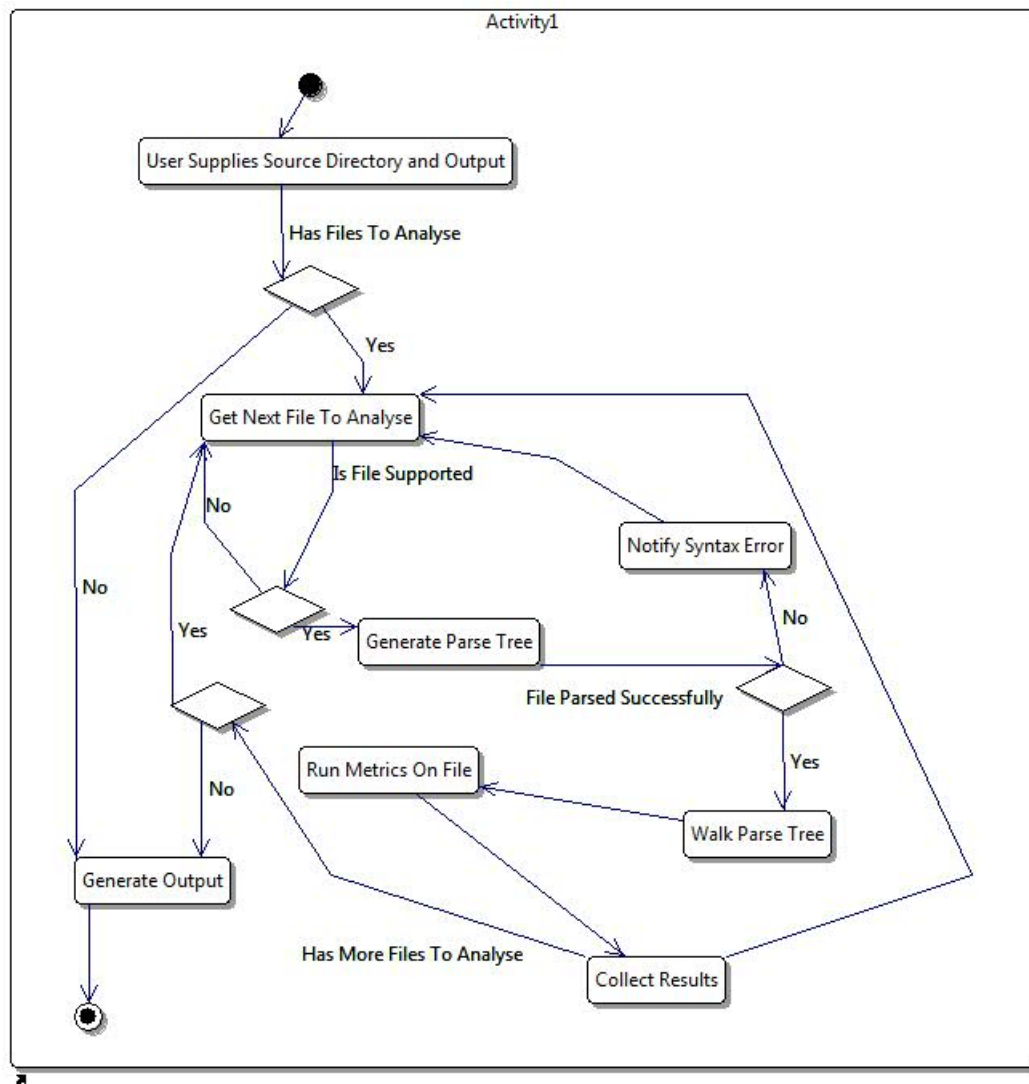
As I stated above, I will also be carrying out system testing on the completed application by using the application the same way as the client would to check that everything works correctly when we put the whole system together. This will include carrying out the following tests using tester source code in a variety of suitable programming languages on multiple platforms.

- **Testing with valid input which is considered to be of good quality** - This will be test to see if my project can input source code which is considered to be of high quality in accordance to the metrics I have put in place in this application. Obviously the expected result is that an output is produced and the result is that my application determined that the source code was of high quality.
- **Testing with valid input which is considered to be of poor quality** - Again this will be testing that output can be produced from this application, but the difference is I will be expecting that my application outputs that the source code is of poor quality.
- **Testing with invalid output** - This test will be to determine that my application can actually handle invalid input correctly.

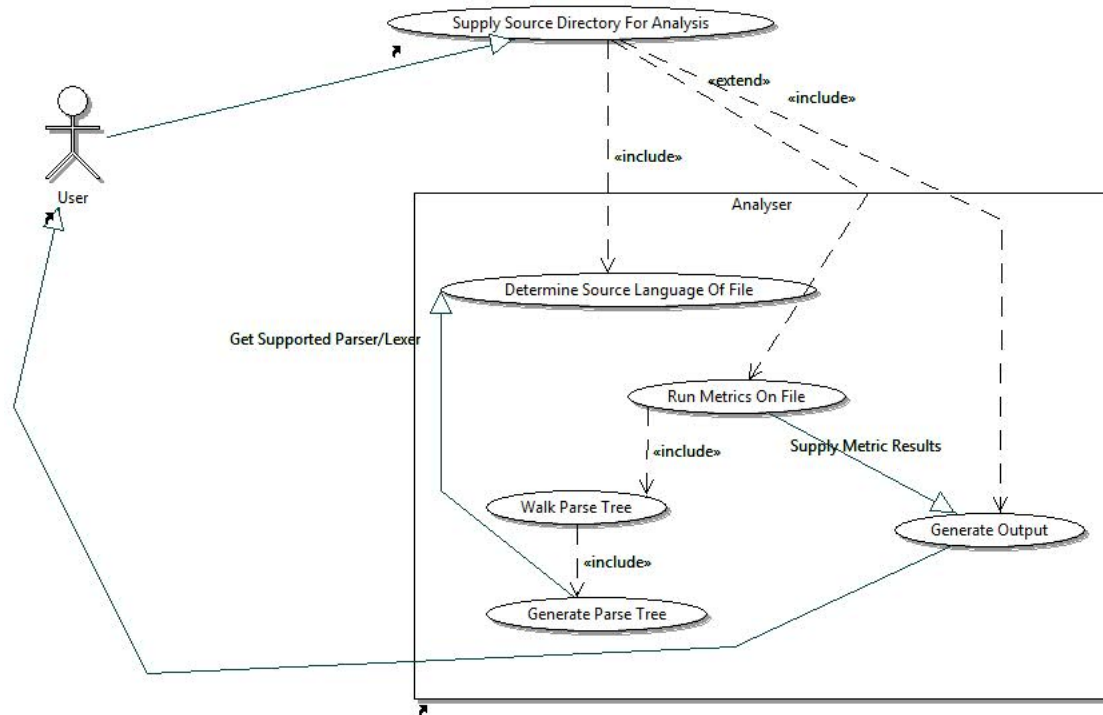
Obviously, as I stated above, if any of the tests that I define during the implementation and testing stages of this project fail, these will then be modified and fixed as defined by the maintenance stage of the waterfall development methodology.

## 4.5 Design Documents

### 4.5.1 Activity Diagram

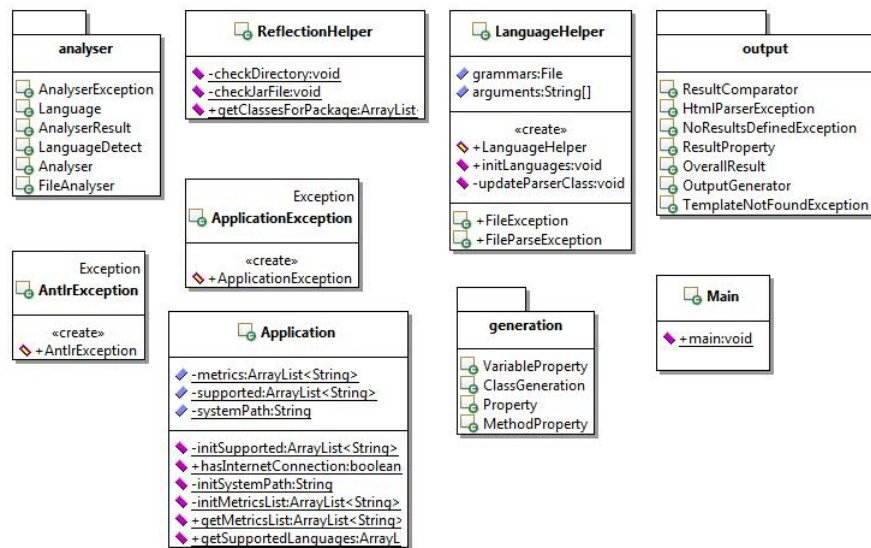


#### 4.5.2 Use case Diagram

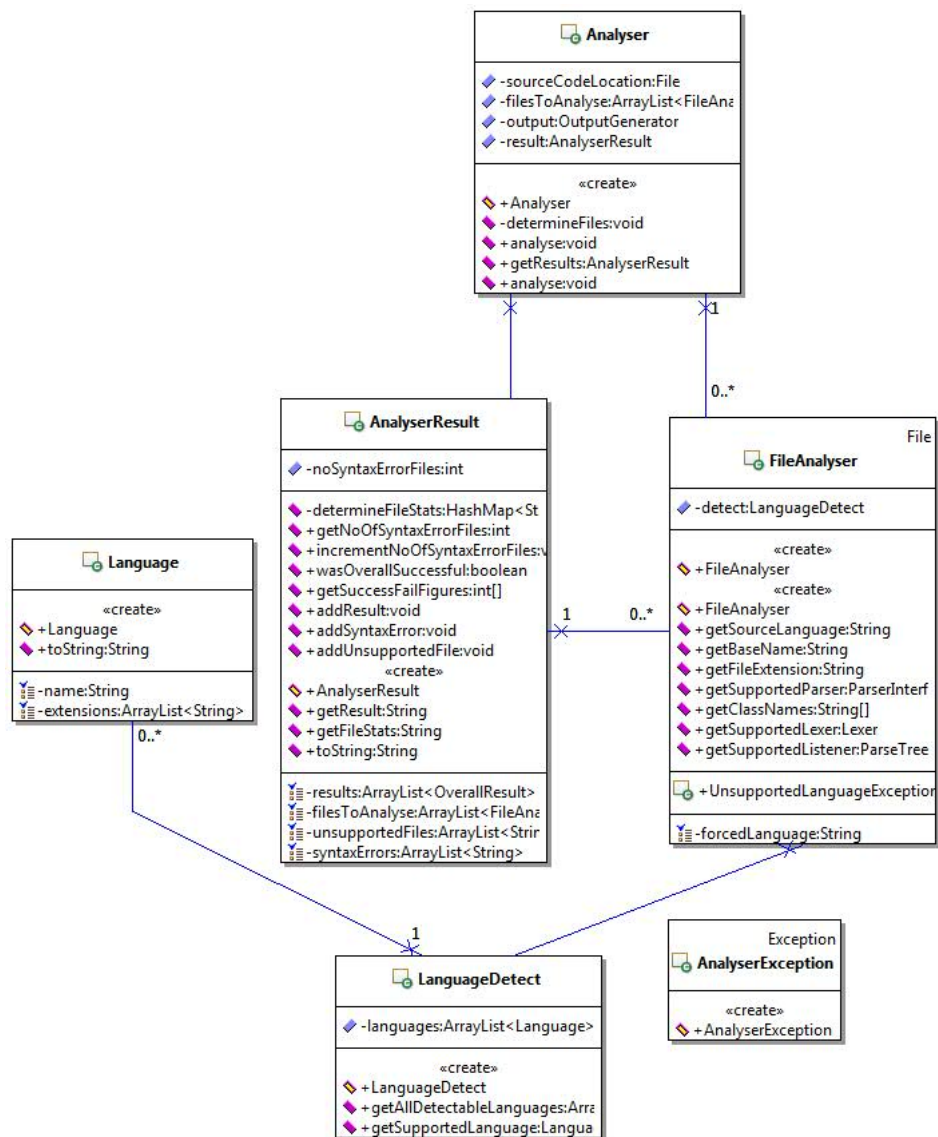


#### 4.5.3 Class Diagrams

Core



## Core - Analyser





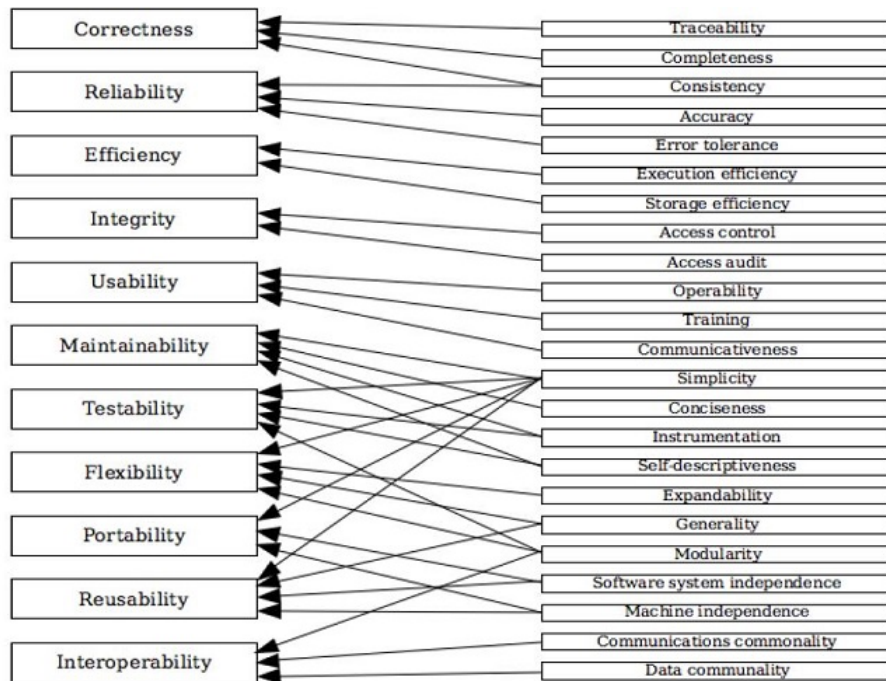


Figure 2: McCall's Quality Model Criteria

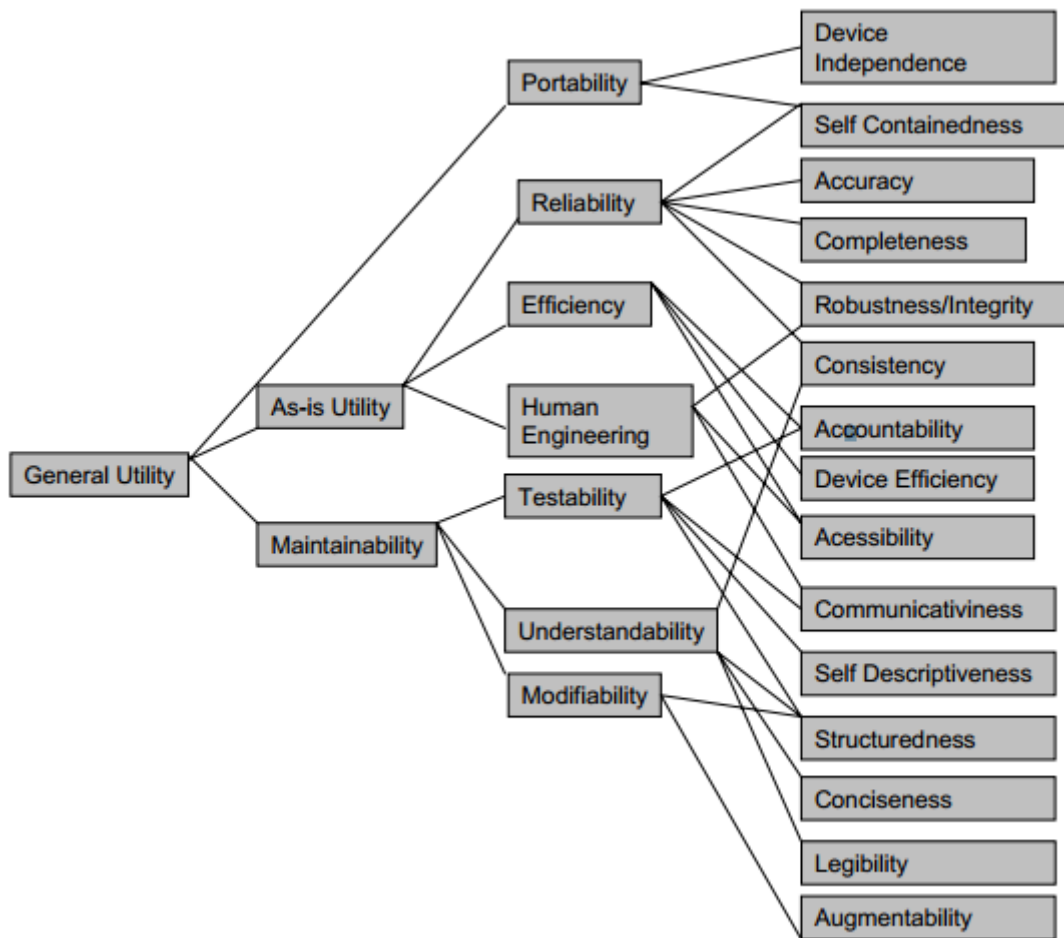


Figure 3: Boehm's Software Quality Model



Figure 4: ISO 25010

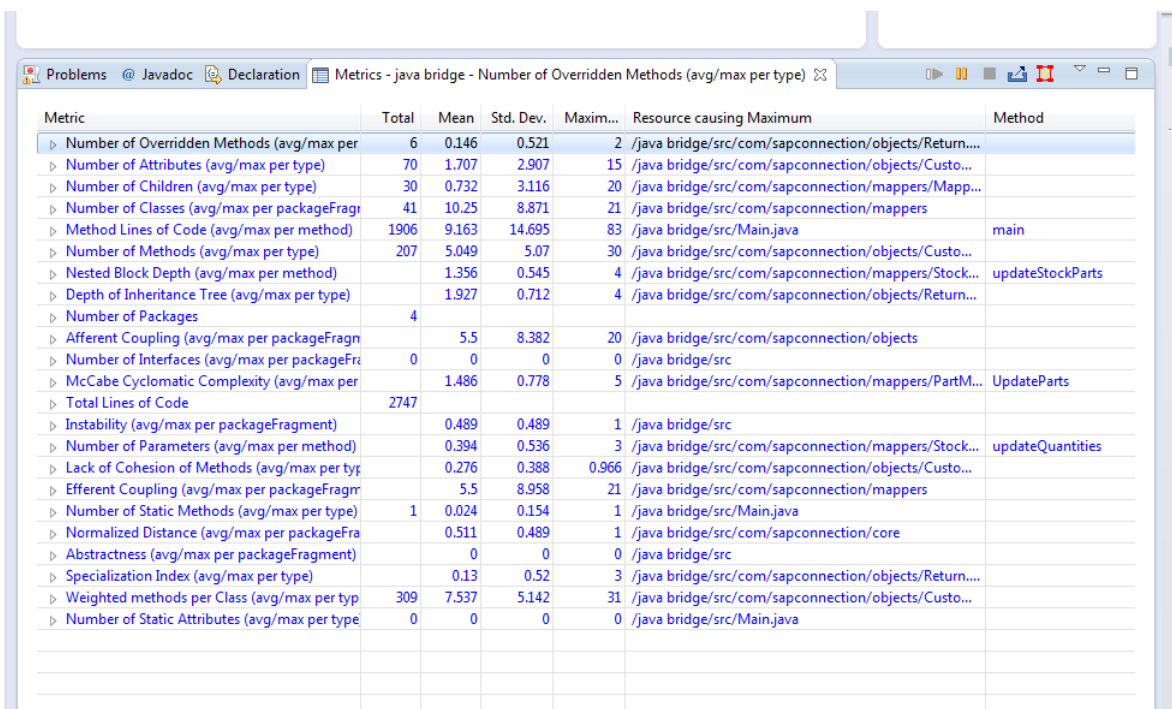
```

begin
    integer x;

    procedure example;
        begin
            integer x;
            ...
        end;
    end;
end;

```


Figure 5: Example ALGOL source code in which the variable x in procedure example refers to a different variable than that in the global scope.




Metric	Total	Mean	Std. Dev.	Maxim...	Resource causing Maximum	Method
▶ Number of Overridden Methods (avg/max per type)	6	0.146	0.521	2	/java bridge/src/com/sapconnection/objects/Return....	
▶ Number of Attributes (avg/max per type)	70	1.707	2.907	15	/java bridge/src/com/sapconnection/objects/Custo...	
▶ Number of Children (avg/max per type)	30	0.732	3.116	20	/java bridge/src/com/sapconnection/mappers/Mapp...	
▶ Number of Classes (avg/max per packageFragm...	41	10.25	8.871	21	/java bridge/src/com/sapconnection/mappers	
▶ Method Lines of Code (avg/max per method)	1906	9.163	14.695	83	/java bridge/src/Main.java	main
▶ Number of Methods (avg/max per type)	207	5.049	5.07	30	/java bridge/src/com/sapconnection/objects/Custo...	
▶ Nested Block Depth (avg/max per method)		1.356	0.545	4	/java bridge/src/com/sapconnection/mappers/Stock...	updateStockParts
▶ Depth of Inheritance Tree (avg/max per type)		1.927	0.712	4	/java bridge/src/com/sapconnection/objects/Return...	
▶ Number of Packages	4					
▶ Afferent Coupling (avg/max per packageFragm...		5.5	8.382	20	/java bridge/src/com/sapconnection/objects	
▶ Number of Interfaces (avg/max per packageFragm...	0	0	0	0	/java bridge/src	
▶ McCabe Cyclomatic Complexity (avg/max per ...)		1.486	0.778	5	/java bridge/src/com/sapconnection/mappers/PartM...	UpdateParts
▶ Total Lines of Code	2747					
▶ Instability (avg/max per packageFragment)		0.489	0.489	1	/java bridge/src	
▶ Number of Parameters (avg/max per method)		0.394	0.536	3	/java bridge/src/com/sapconnection/mappers/Stock...	updateQuantities
▶ Lack of Cohesion of Methods (avg/max per typ...		0.276	0.388	0.966	/java bridge/src/com/sapconnection/objects/Custo...	
▶ Efferent Coupling (avg/max per packageFragm...		5.5	8.958	21	/java bridge/src/com/sapconnection/mappers	
▶ Number of Static Methods (avg/max per type)	1	0.024	0.154	1	/java bridge/src/Main.java	
▶ Normalized Distance (avg/max per packageFra...		0.511	0.489	1	/java bridge/src/com/sapconnection/core	
▶ Abstractness (avg/max per packageFragment)		0	0	0	/java bridge/src	
▶ Specialization Index (avg/max per type)		0.13	0.52	3	/java bridge/src/com/sapconnection/objects/Return....	
▶ Weighted methods per Class (avg/max per typ...	309	7.537	5.142	31	/java bridge/src/com/sapconnection/objects/Custo...	
▶ Number of Static Attributes (avg/max per type)	0	0	0	0	/java bridge/src/Main.java	

Figure 6: The result from the SourceForge Metric Tool on an example Java project.

## Summary of Rules violated


Rules can be checked live at development-time, from within Visual Studio. [Online documentation.](#)


NDepend rules report too many flaws on existing code base? Use the option **Recent Violations Only!**

Display 25 records

Q












Name	# Matches	Elements	Group
 Quick summary of methods to refactor	25	methods	Code Quality
 Methods too big	1	methods	Code Quality
 Methods too complex	1	methods	Code Quality
 Methods potentially poorly commented	8	methods	Code Quality
 Methods with too many local variables	7	methods	Code Quality
 Types with too many methods	3	types	Code Quality
 Class with no descendant should be sealed if possible	41	types	Object Oriented Design
 A stateless class or structure might be turned into a static type	10	types	Object Oriented Design
 Non-static classes should be instantiated or turned to static	22	types	Object Oriented Design
 Methods should be declared static if possible	125	methods	Object Oriented Design
 Classes that are candidate to be turned into structures	15	types	Design

Figure 7: The main report from Ndepend.

methods	# lines of code (LOC)	# IL instructions	Cyclomatic Complexity (CC)	IL Cyclomatic Complexity (ILCC)	IL Nesting Depth	# Parameters	# Variables	# O
Convert(Nullable<Double>,String)	15	169	0	16	6	2	3	1
ConvertSetsViewModelOriginalUnitPropertyToOriginalUnit()	25	92	0	1	0	0	21	1
ConvertSetsViewModelConvertedValuePropertyForFahrenheitInputToCorrectlyConvertedCelsiusTemperature()	20	75	0	1	0	0	17	1
ConvertSetsViewModelConvertedValueProper								

Figure 8: In Depth Details from the report.

```

if(valid) {
    for(int i = 0; i < data.size(); i++) {
        if(alsovalid) {
            ...
        }
    }
}

```

Figure 9: Example Java code which shows a nesting depth of 3.

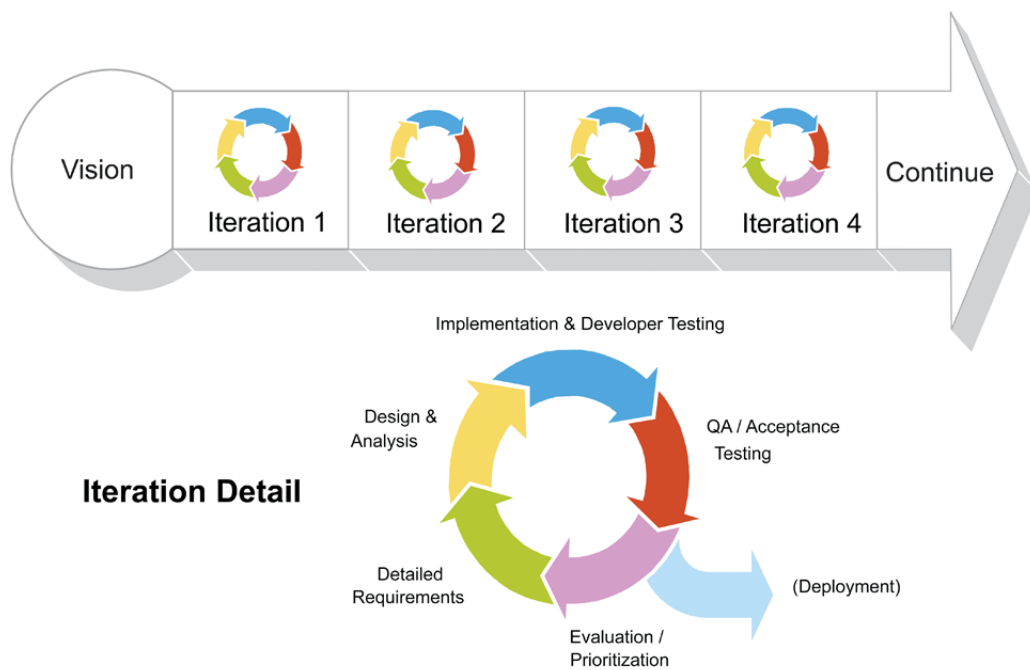
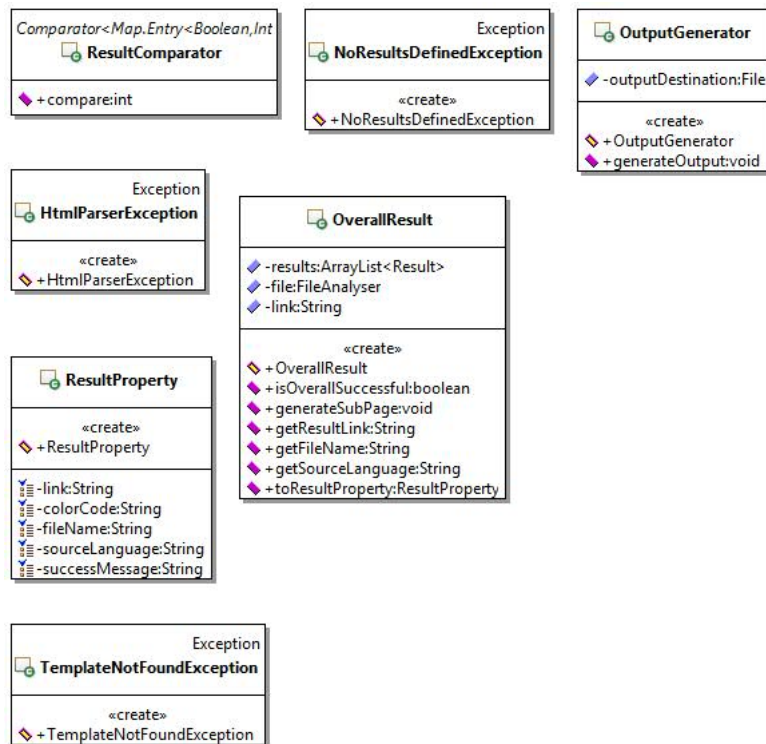
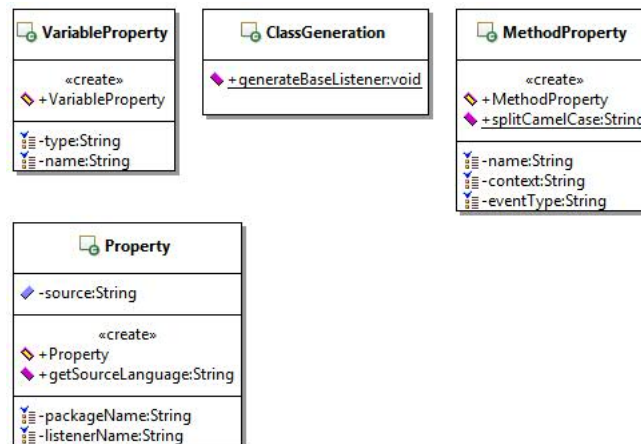


Figure 10: The Agile development life-cycle

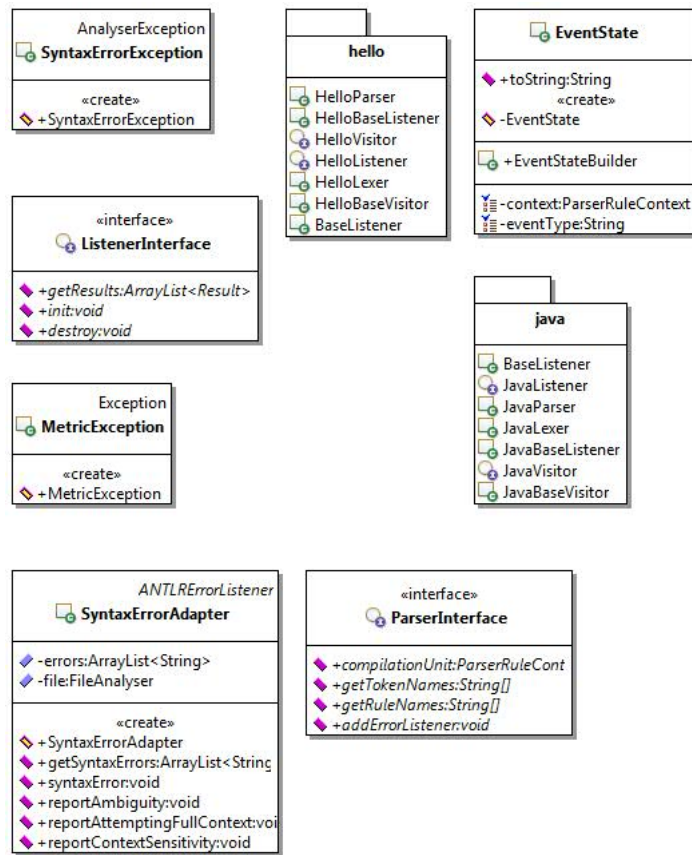
## Core - Output



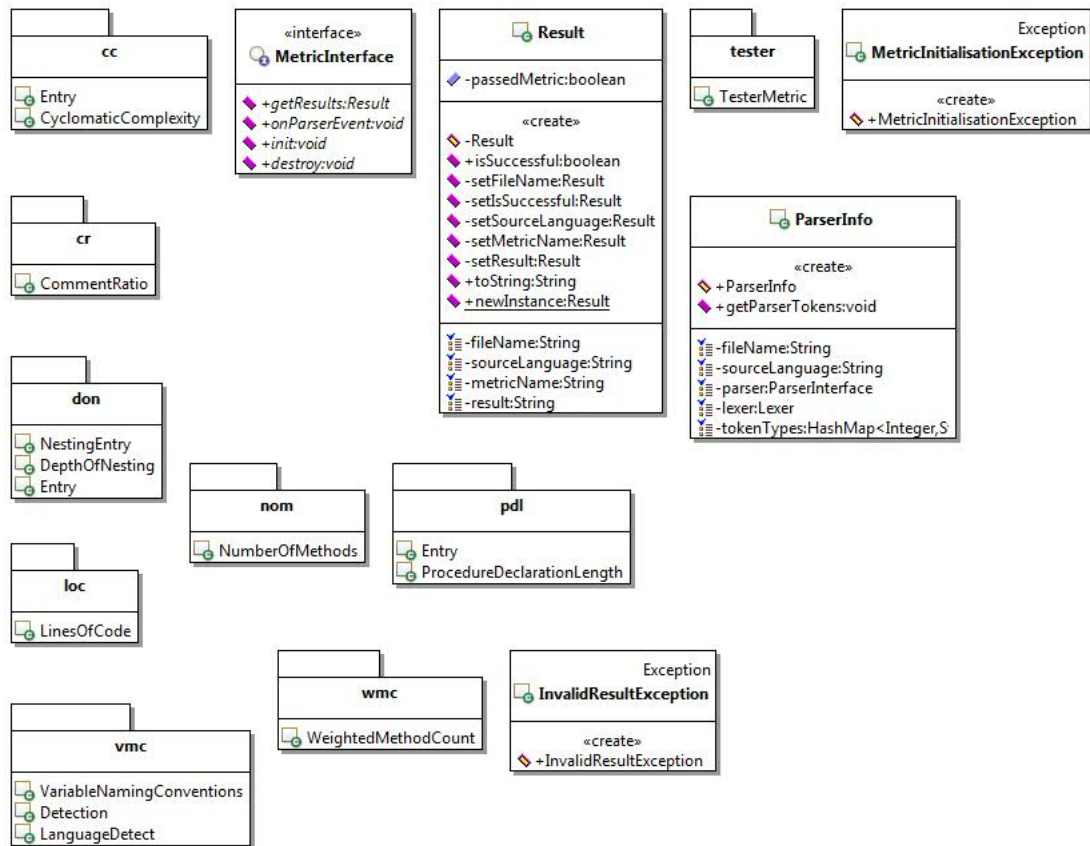
## Core - Generation



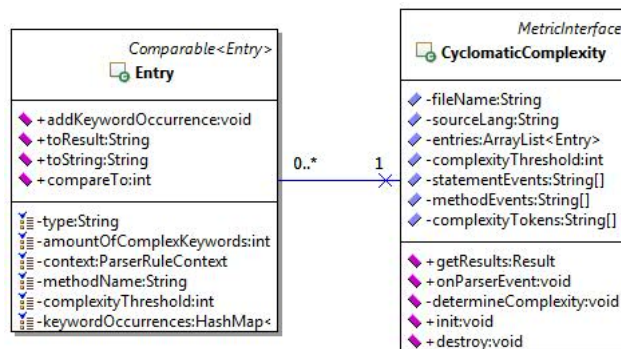
## Language



## Metric

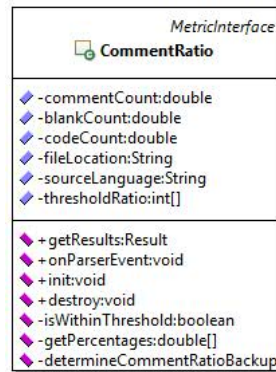


## Metric - Cyclomatic Complexity

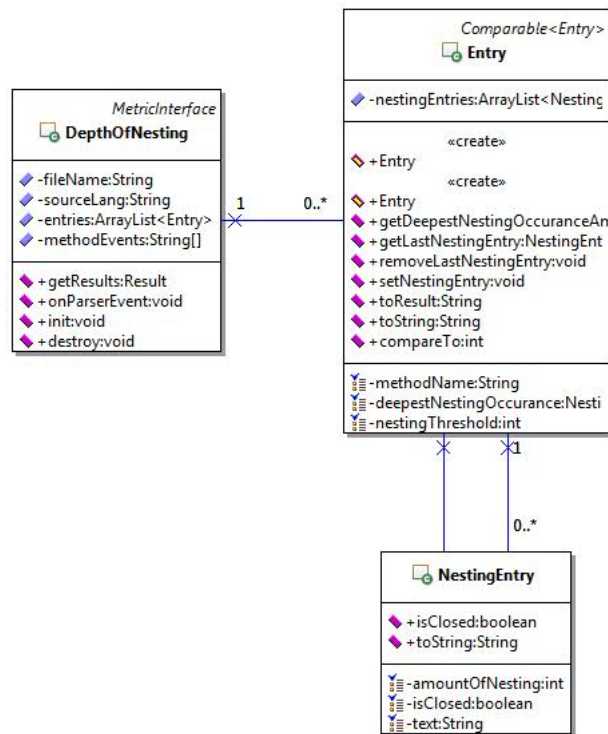




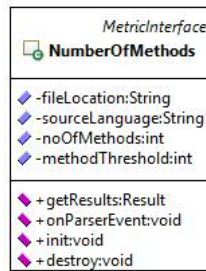
## Metric - Comment Ratio



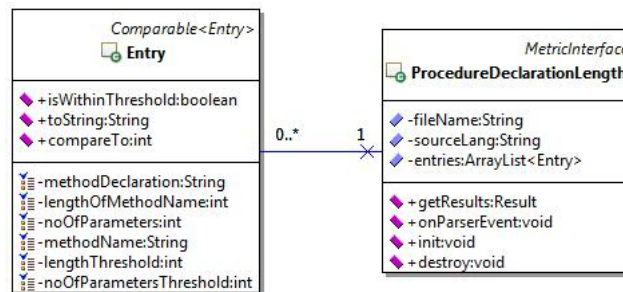
## Metric - Depth of Nesting



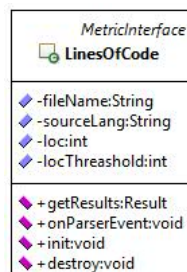
## Metric - Number of Methods



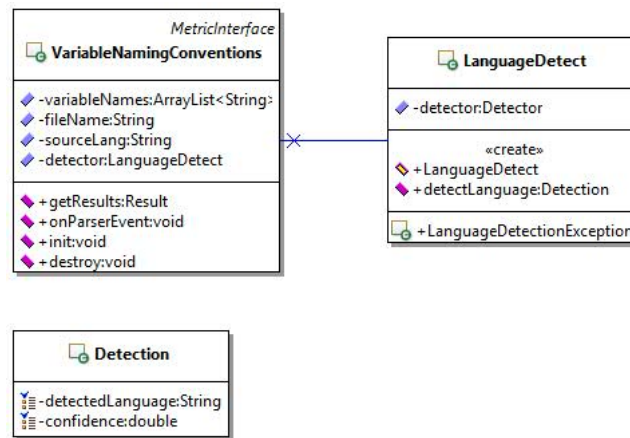
## Metric - Procedure Declaration Length



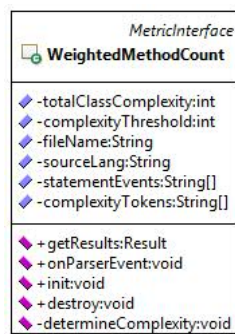
## Metric - Lines of Code



## Metric - Variable Naming Conventions



## Metric - Weighted Method Count



## 4.6 Design Evaluation

This application needed to have a modular design in order to allow for easy expandability. This included added new metrics and new supported languages, which are the generated classes from the ANTR parser generator.

Even though there are two language packages in the language class diagram above, the class diagram for these has been omitted as they have been implemented and tested by the maintainer of the ANTLR parser generator.

Because of the modular design approach there was main functionality that had to be implemented as part of the initial design. This included:

- Having a way of 'autoloading' metric modules into the application by a generic interface (which in this case is defined as *MetricInterface*), so obviously looking for classes in the metric package that implement this interface and determine this a metric to 'load'. Designing metrics this way mean that a new metric can be implemented easily.

- Having a way to determine what languages are currently supported by the application, i.e what grammars have been inputted into ANTLR and a Parser/Lexer combo have been generated.
- Having a way to determine the source language of the file being analysed was supported, i.e a parser/lexer has been provided.
- Having a way to generically walk a parse tree generated from any Parser/Lexer class and notify metric classes when a rule has been matched.

The way that I decided to design this application was to use Java's reflection API in order to determine the languages that were currently 'installed' into the system, as each of the different language packages had a similar package name as shown in the language class diagram above.

After completing some research into the ANTLR parser generator, it was determined that the classes generated were obviously named after the grammar they are defined for, so for example in the Java grammar, the parser/lexer and listener classes generated were called 'JavaParser', 'JavaLexer' and 'JavaBaseListener'. This obviously produced design factors in which the application could initialise and call the correct Parser/Lexer classes to correctly parse the file currently being analysed. This is where the introduction of the 'FileAnalyser' class and the generic Parser-Interface/ListenerInterface interfaces were introduced into the design, in which we can detect the source language of the file and initialise the Parser/Lexer and Listener interface instances using reflection.

The metrics also had to have a generic interface as stated above so that we could initialise all of the metrics in the listener class and call the method 'onParserEvent()' could be called when an event is triggered while walking the parse tree.

The overall design of the application was based on trying to make it as easily expandable as possible and also so that it outputs the results in a user friendly fashion in the form of HTML documents.

## 5 Implementation

The first thing that had to be completed during implementation was the ability to first take in a file or directory and determine what files will be analysed and also determine the source language of the current file use the generated Parser/Lexer instances to actually walk the parse tree. As stated in the design evaluation above, this posed a variety of different problems because the classes were defined by the name of the grammar, which would change based on the determined source language of the file currently being analysed.

### 5.1 Determining the Source Language of a File.

The source language of the file currently being analysed was initially found by using the file extension of the file being analysed, for example all source Java files end with the extension '.java'. But as development continued, a massive flaw was found in this solution because for some languages the file extension did not match the source language exactly or a source language could use multiple file extensions, for example in C we have the file extensions '.c' or '.h'.

The only real way that we could fully determine the source language was to either attempt

```

public class Language {
    private String name;

    private ArrayList<String> extensions;

    /**
     * Initialises a language object.
     * @param name the name of the language
     * @param extensions a list of file extensions this language uses.
     */
    public Language(String name, ArrayList<String> extensions) {
        this.name = name;
        this.extensions = extensions;
    }

    /**
     * returns the name of this language
     * @return the name of this language.
     */
    public String getName() {
        return this.name;
    }

    /**
     * returns the list of extensions that this language uses
     * in its files.
     * @return the list of extensions.
     */
    public ArrayList<String> getExtensions() {

```

Figure 11: A snippet of the Language class, which is just an encapsulation of a single entry in the language.yaml file

to parse the current file with all of the Parsers until a match is obtained, or we could keep a simple configuration file which we could match a file extension to a source language name. Obviously the latter is the most logical choice to allow for expansion because the more languages we support, the more initial 'trial' parses in order to determine the source language of the file and this in turn would take a lot more computing power, especially if we are analysing multiple files.

The LanguageDetect class reads and generates an ArrayList of Language objects, which are essentially encapsulations of a single entry, as shown in figure 11 on page 44. We can see in figure 12 on page 45, that the LanguageDetect class reads the configuration file and then we can just query the ArrayList of Language objects to determine the source language and from there we can determine if the language is supported by the analyser.

After determining the source language of the file, the next task was to determine whether the language was supported by the analyser. Early on in the design process, it was decided that the application would need static system properties that could be accessed anywhere in the application and this included what languages were supported and what metrics were currently 'installed'. This was implemented as the Application class, and we could call the methods getSupportedLanguages and getMetricList anywhere in the application, and this is shown in figure 14 on page 46.

At this point, if the file is supported, we would attempt to parse the source language into a parse tree, as shown in figure 15 on page 46, and then walk the tree triggering events in the listener instance when a rule is matched, essentially notifying the metrics that matched a specific rule in the grammar.

```

languages = new ArrayList<Language>();
Yaml yaml = new Yaml();
try {

    //load the YAML file.
    InputStream input = new FileInputStream(new File(Application.getSystemPath()+"config/languages

    Map o = (Map) yaml.load(input);

    //iterate through the entries and parse each entry into a Language object.
    for(Map.Entry<String, Map> entry : (Set<Map.Entry<String, Map>>)o.entrySet()) {

        Map v = entry.getValue();
        ArrayList<String> e = new ArrayList<String>();
        if(v.containsKey("primary_extension")) {
            e.add(((String)v.get("primary_extension")).substring(1));
        }
        if(v.containsKey("extensions")) {
            List<String> ex = (List<String>) v.get("extensions");
            for(String ext : ex) {
                e.add(ext.substring(1));
            }
        }

        if(!e.isEmpty()) {
            languages.add(new Language(entry.getKey(), e));
        }
    }
}

```

Figure 12: A snippet of the LanguageDetect class, which reads the configuration file.

```

Java:
  type: programming
  ace_mode: java
  color: "#b07219"
  primary_extension: .java

```

Figure 13: The Java entry in the languages configuration file

```

/**
 * iterates through the metrics packages to find which classes are declared
 * to be 'metrics'. A metric is defined as any class that implements
 * MetricInterface.
 *
 * @version 1.1 - uses reflection instead of depending on a certain file
 * location.
 * @return an ArrayList<String> the class names of the found metrics.
 * @throws ClassNotFoundException when a class could not be found.
 */
private static ArrayList<String> initMetricsList() throws ClassNotFoundException {
    ArrayList<String> classes = new ArrayList<String>();
    ArrayList<Class<?>> cs = ReflectionHelper.getClassesForPackage("org.codeanalyser.metric");
    for (Class c : cs) {
        Class<?>[] in = c.getInterfaces();
        if (in.length != 0) {
            if (in[0].getSimpleName().equals("MetricInterface") && !c.getSimpleName().equals("TesterMetric")) {
                classes.add(c.getName());
            }
        }
    }
}

/**
 * iterates over the packages in 'languages' to determine which languages
 * are currently supported by the system.
 *
 * @version 1.1 - used reflection to determine the package names instead of
 * depending on a specific file location.
 * @return an ArrayList<String> of supported language names.
 */
private static ArrayList<String> initSupported() throws ClassNotFoundException {
    ArrayList<String> support = new ArrayList<String>();
    String pc = "org.codeanalyser.language";
    ArrayList<Class<?>> classes = ReflectionHelper.getClassesForPackage(pc);
    for (Class c : classes) {
        try {
            String n = c.getPackage().getName();
            String ln = n.substring(pc.length() + 1, n.length());
            if (!ln.isEmpty() && !support.contains(ln)) {
                support.add(ln);
            }
        } catch (StringIndexOutOfBoundsException e) {}
    }

    return support;
}

```

Figure 14: A snippet of the Application class which is used to grab System properties of the application, like getting the supported languages or metrics by using reflection.

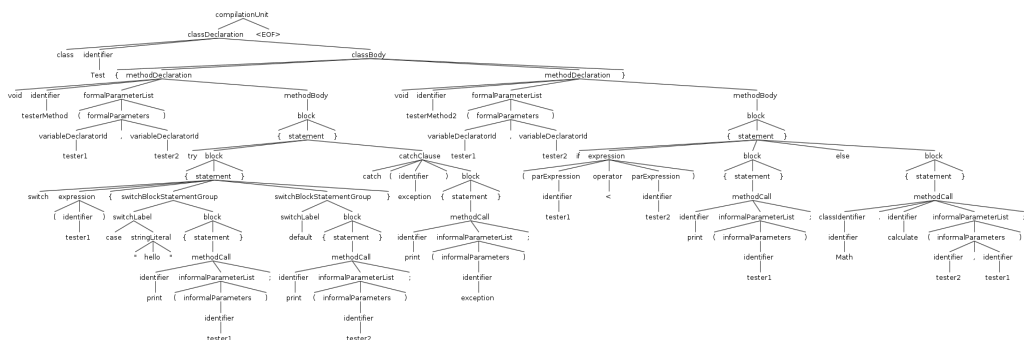


Figure 15: A parse tree produced for the 'hello' language using the generated parser and lexer instances.

```

public interface ParserInterface {

    /**
     * calls the root rule in the parser and returns the parse tree.
     * @return the parse tree for the source file.
     */
    public ParserRuleContext compilationUnit();

    /**
     * gets the token names used in the parser.
     * @return the token names.
     */
    public String[] getTokenNames();

    /**
     * get the rule names that were defined in the grammar.
     * @return the rule names.
     */
    public String[] getRuleNames();

    /**
     * add a ANTLRErrorListener to the parser to listen
     * for errors that occur.
     * @param listener the error listener to attach to the parser
     */
    public void addErrorListener(ANTLRErrorListener listener);

}

```

Figure 16: The ParserInterface interface which all of the generated Parser classes implement. It depends on the root rule being declared as 'compilationUnit'.

## 5.2 Initialise Generic Parser/Lexer Instances

As stated above, ANTLR generates a Parser/Lexer instance using the grammars name, so this would mean we would need to generate a different object for each different language we wanted to analyse. This would be impossible to implement as we wouldn't know what classes to instantiate until the analyser was provided source directory at run-time, and the object variables have to be declared at compile-time. The solution that was found was to implement an interface, called `ParserInterface`, that each of the generated Parsers would implement meaning that we could use reflection to initialise the correct parser instance and then just cast it to a `ParserInterface`. The only limitation with this approach is that ANTLR generates a method which is defined by the root rule in the grammar to obtain the parse tree, and this means that we need the root rule to be the same in all of the grammars so that the parser class can in-fact implement `ParserInterface` and this is shown in figure 16 on page 47.

Initialising the Lexer instance was a lot easier as all of the required methods were defined in ANTLR's core `Lexer` class, with the exception of the tokens that were defined in the grammar. So it was plausible to use the same technique that was applied to the Parser but just cast it to its super-class. However, the tokens that were defined in the grammar are obviously not defined in the core `Lexer` class as they are unique to the grammar. These should be provided to metrics in-case lexical analysis needs to be performed. The tokens can be gained by again using reflection and obtaining the final static int variables defined in the lexer class. this is shown in figure 17 on page 48.



```

/**
 * gets the tokens that have been defined by the grammar.
 * @version 1.1 updated to use reflection and read the final
 * static int fields that are defined in the {SOURCE_LANGUAGE}Parser
 * classes when antlr generates them. This means that it no longer depends
 * on a certain file location.
 */
public void getParserTokens() {
    Class<?> c = parser.getClass();
    HashMap<Integer, String> tokens = new HashMap<Integer, String>();
    for(Field f : c.getFields()) {
        if(f.getType().getName().equals("int")) {
            if(!f.getName().startsWith("RULE_")) {
                try {
                    tokens.put(f.getInt(null), f.getName());
                } catch (IllegalAccessException e) {}
            }
        }
    }
    this.tokenTypes = tokens;
}

```

Figure 17: The method in the ParserInfo class which grabs the tokens from a generated lexer class.

### 5.3 Initialising the Listener Instance.

As described above, all of the listener classes generated by ANTLR are not only defined by the grammar, but they also don't initialise or notify any of the metrics classes so they can carry out analysis on the current file. Obviously this meant that we need to generate our own listener for each newly generated grammar which accomplishes out this task. Figure 27 on page 60 shows the basic template for this new listener class which implements the `ListenerInterface` interface which means we can initialise this Listener generically in the same way as the Parser and Lexer classes. Figure 27 on page 60 also shows the main method in this class which is the `init` method. This uses the `Application.getMetricList` method and reflection to initialise all of the metrics to the generic `MetricInterface` and stores them all in an `ArrayList` so that we can just iterate through them when a enter/exit rule method is triggered while walking the tree.

### 5.4 Handling errors while Parsing a File or if the File is Unsupported.

There may be instances in analysing a file where the file is either unsupported by the analyser or the file contains syntax errors, which would mean generating a parse tree would be unsuccessful. Obviously determining whether a language is unsupported is easily done by using the `LanguageDetect` class.

The way that we can handle syntax errors is to attach a `ANTLRErrorListener` to the Parser and Lexer instances so that if an error is found this listener is notified. I implemented a custom `ANTLRErrorListener` that only listens for syntax errors. Figure 25 on page 58 shows that the listener not only saves all of the syntax errors that occur, but it also keeps a count of all the files that contained syntax errors so that we can deliver this information in the generated output.

All of the functionality to get the supported Parser/Lexer and Listener instances are encapsulated into the `FileAnalyser` class. This is so we can iterate through the files we will be analysing and then initialise an instance of a `FileAnalyser` and then we can get the supported Parser/Lexer and Listener instances by the methods provided as shown in figure 24 on page 57. The `FileAnalyser` class is used in the `analyse` method of the `Analyser` which as shown in figure 26 on page 59.

## 5.5 Implementing the metrics

As stated above, the application uses reflection to search for any class that implements the `MetricInterface` interface in the `org.codeanalyser.metric` package. The `MetricInterface` has four methods that have to be implemented by any metric used in this application which are shown in figure 23 on page 56. The method `init` is called when the metrics are initialised in the listener and it provides the metric with a `ParserInfo` object which is all the information about the current file being analysed, like the location, source language and the Lexer/Parser instance that were used to generate the Parse Tree. The method `onParserEvent` is the method that is called each time a rule is matched while walking the parse tree and gives the metric access to the `EventState` object which contains information about the event, such as the rule that was triggered and the `ParserRuleContext` which is the segment of the parse tree that matched the current rule.

In total, there are 8 metrics that have been defined as part of this project, as declared in the requirements specification, including `cyclomaticComplexity` and `CommentRatio`.

### 5.5.1 cyclomatic Complexity Implementation

cyclomatic Complexity is the amount of complex keywords that are used in any of the method or constructors defined in the source class. As you can see in figure 22 on page 55 the complexity keywords can be defined as either a statement, a catch clause or a switch block statement group rule in the grammar. So we can just listen out until any of these rules is matched and check to see if the token used was a complexity keyword and increment the count for that particular method/constructor. Then for the result all we have to determine is if the worst defined method/constructor in the class had a complexity value that was over the threshold of 10 as defined.

### 5.5.2 Comment Ratio Implementation

Determining the comment ratio depends solely on using lexical analysis to determine when a comment token is matched. So as you can see in figure 21 on page 54 this metric performs all of its calculation in the `init` method because we don't need to be notified for any rules that are matched while walking the parse tree. In this metric we need to determine the amount of lines that are considered to be commenting i.e the amount of `COMMENT` or `LINE_COMMENT` tokens that are found, and also the amount of any other tokens that are found, which are considered to be code. We also need to determine the amount of blank lines that occur so we can correctly gauge the comment to code ratio. The results in this metric are if the comment ratio falls between 2 specific thresholds.

## 5.6 Generating User-Readable Output

Analysing files using metrics is completely useless without providing the user with some useful feedback based on how well their source code performed. This is where the `OutputGenerator`

class comes in. This class utilises `StringTemplate` (found at <http://www.stringtemplate.org>) to populate a template file with variables which in this case is the `AnalyserResult` object (shown in figure 20 on page 53). The `AnalyserResult` class encapsulates an entire result from the overall analysis which includes the locations of the unsupported files, the syntax errors that occurred, the file statistics, i.e the percentages of each supported source language was found and obviously an `ArrayList` of `Result` objects provided by the metrics. As shown in figure 19 on page 52, the `generateOutput` method in the `OutputGenerator` class is responsible for pushing variables to the template and then saving the output to the destination provided by the user in the form of HTML web pages.

Because we could potentially analyse an unlimited amount of files, the output had to be easy to understand and read. This is why it was decided to generate the main bulk of the output for each individual file separately and then link to these in the index page, so the user can get an overall summary of the results on the index page and then open a new page for any individual file to get detailed information as shown in the example output provided in the appendix in section C on page 88.

## 6 Testing

### 6.1 Introduction

As stated in the testing strategy above, I have taken a typical approach to testing that the application works correctly and this included carrying out JUnit testing during development and also carrying out system testing after the implementation was complete and then fix any bugs that occurs during any of these two phases.

### 6.2 Unit Testing

#### 6.2.1 Test Suite

```
/**
 * Runs All of the defined UnitTests for the application.
 * @author Jack Timblin - U1051573
 */
@RunWith(Suite.class)
@Suite.SuiteClasses({ApplicationTest.class, AnalyserTest.class, FileAnalyserTest.class,
LanguageDetectTest.class, HelloParseTreeConstruction.class, JavaParseTreeConstruction.class,
SyntaxErrorAdapterTest.class, MetricInitialisationTest.class})
public class MainTestSuite {

    @Before
    public void setUp() throws Exception {
        Application.systemPath =
            "/home/jack/Documents/University/CodeAnalyser/";
    }
}
```

```

group OutputTemplate;

main(unsupportedFiles, hasUnsupported, results, hasSyntax,
syntaxErrors, overallResult, fileStats) := <>
\<!DOCTYPE HTML>
\<html>
  \<head>
    \<title>Code Analyser Results\</title>
  \</head>
  \<body>
    \<div style='width:50%;border-bottom:1px solid;'>
      \<h1>Code Quality Analyser Results.\</h1>
      <overallResult>
      <fileStats>
    \</div>
    \<div>
      \<table>
        \<thead>
          \<tr>
            \<td>Metric Results\</td>
          \</tr>
        \</thead>
        \<tbody>
          <results:renderResult()>
        \</tbody>
      \</table>
    \</div>
    \<div style='width:50%;border-top:1px solid;'>
      \<table>
        \<thead>
          \<tr>
            \<td>Unsupported Files\</td>
          \</tr>
        \</thead>
        \<tbody>
          <if(hasUnsupported)>
            <unsupportedFiles:renderUnsupported()>
          <else>
            \<tr>
              \<td>No Files Were Unsupported.\</td>
            \</tr>
          <endif>
        \</tbody>
      \</table>
      \<table>
        \<thead>
          \<tr>
            \<td> Syntax Errors\</td>
          \</tr>
        \</thead>
        \<tbody>
          <if(hasSyntax)>
            <syntaxErrors:renderUnsupported()>
          <else>
            \<tr>
              \<td>No Files Contained Syntax Errors.\</td>
            \</tr>
          <endif>
        \</tbody>
      \</table>
    \</div>
  \</body>
  \<style>
    table thead td {
      font-weight:bold;
    }
    table table thead td {
      font-weight:normal;
    }
  \</style>
\</html>
<>

```

Figure 18: A snippet of the StringTemplate template used to render the output

```

*/
public void generateOutput(AnalyserResult result) throws NoResultsDefinedException,
    TemplateNotFoundException,
    HtmlParserException {

    //check that the output directory exists.
    if (!this.outputDestination.exists() || !this.outputDestination.isDirectory()) {
        throw new TemplateNotFoundException("Output Location does not exist or is not a directory");
    }

    //check that there are some results to display.
    if (result.getResults().isEmpty()) {
        throw new NoResultsDefinedException("No Results Defined From Running The Metrics.");
    }

    //foreach of the overallResult entries
    //make a single ResultProperty entry to pass to ST.
    ArrayList<ResultProperty> res = new ArrayList<ResultProperty>();
    for (OverallResult re : result.getResults()) {
        res.add(re.toResultProperty(outputDestination));
    }

    //generate the HTML file using the template.
    STGroupFile group = new STGroupFile(Application.getSystemPath()+"antlr/templates/OutputTemplate.stg");
    ST main = group.getInstanceOf("main");

    main.add("unsupportedFiles", result.getUnsupportedFiles());
    main.add("hasUnsupported", (result.getUnsupportedFiles().size() > 0));
    main.add("results", res);
    main.add("hasSyntax", (!result.getSyntaxErrors().isEmpty()));
    main.add("syntaxErrors", result.getSyntaxErrors());
    main.add("overallResult", result.getResult());
    main.add("fileStats", result.getFileStats());

    //save the rendered template to a file.
    File output = new File(this.outputDestination.getAbsolutePath() + "/output.html");

    if (output.exists()) {
        output.delete();
    }

    try {
        FileWriter writer = new FileWriter(output);
        writer.append(main.render());
        writer.flush();
        writer.close();
    } catch (IOException e) {
        throw new TemplateNotFoundException("Could not open, read or write output destination.");
    }
}

```

Figure 19: The generateOutput method in the OutputGenerator class, populates the template shown in figure 18 on page 51.

```

/**
 * determines the percentages of each file type that was passed to the
 * analyser.
 * @return the HashMap with the key as the source language and value as
 * the percentage.
 */
private HashMap<String, Double> determineFileStats() {
    HashMap<String, Double> map = new HashMap<String, Double>();
    for(FileAnalyser f : this.filesToAnalyse) {
        String source = f.getSourceLanguage();
        source = (source == null) ? "Unknown" : source.substring(0, 1).toUpperCase()+source.substring(1);
        Double i = map.get(source);
        map.put(source, (i != null) ? i + 1 : 1);
    }

    HashMap<String, Double> percentages = new HashMap<String, Double>();
    for(Map.Entry<String, Double> entry : map.entrySet()) {
        double d = Math.round((entry.getValue()/this.filesToAnalyse.size())*100);
        percentages.put(entry.getKey(), d);
    }

    return percentages;
}

/**
 * gets the amount of files that successfully passed and failed.
 * @return the amount of files that successfully passed and failed.
 */
public int[] getSuccessFailFigures() {
    HashMap<Boolean, Integer> map = new HashMap<Boolean, Integer>();
    for (OverallResult r : this.results) {
        Integer i = map.get(r.isOverallSuccessful());
        map.put(r.isOverallSuccessful(), (i != null) ? (int) i + 1 : 1);
    }

    Integer t = map.get(true); Integer f = map.get(false);
    f = (f == null) ? 0 : f;

    //include failures from syntax errors or if any files were unsupported.
    if((this.getNoOfSyntaxErrorFiles() != 0 || !this.unsupportedFiles.isEmpty()) {
        f = f + (this.getNoOfSyntaxErrorFiles() + this.unsupportedFiles.size());
    }
    return new int[] {(t == null) ? 0 : t, f};
}

/**
 * gets the number of files that have has one of more
 * syntax errors occur.
 * @return the number of files as more than one syntax error
 * can occur per file.
 */
public int getNoOfSyntaxErrorFiles() {
    return this.noSyntaxErrorFiles;
}

/**
 * increment the number of files by 1.
 */
public void incrementNoOfSyntaxErrorFiles() {
    this.noSyntaxErrorFiles++;
}

/**
 * determines if the amount of successful files was more than that
 * of what failed, this includes checking if the metric results were
 * overall successful.
 * @return true if this analysis was overall successful, or false if not.
 */
public boolean wasOverallSuccessful() {

```

Figure 20: The AnalyserResult class which is an overall encapsulation of the result of running analysis on a source location.

```

//get the lexer.
Lexer l = info.getLexer();
int cmc = 0, bc = 0, to = 0;
ArrayList<Integer> lines = new ArrayList<Integer>();
ArrayList<Integer> clines = new ArrayList<Integer>();

//get the tokens that were found by performing lexical analysis
//on the file.
List<? extends Token> tokens = l.getAllTokens();
for (Token t : tokens) {

    //get the type type.
    String tt = tokenTypes.get(t.getType());
    if(!tt.equals("LINE_COMMENT") && !tt.equals("COMMENT")) {
        //this is not a token, just record the line.
        if(!lines.contains(t.getLine())) {
            lines.add(t.getLine());
        }
        if(!clines.isEmpty()) {
            //a previous multiline comment exists.
            if(t.getLine() <= clines.get(0)) {
                cmc--;
            }
            //if this was the last line the multicomment does up to.
            if(t.getLine() == clines.get(0)) {
                //remove this entry.
                clines.clear();
            }
        }
    } else {
        //if there is no code on the same line.
        if(tt.endsWith("COMMENT")) {
            //get the amount of lines this comment rolls over.
            int noLines = t.getText().split("\n").length;
            clines.add(t.getLine()+noLines);
            //check this doesnt have code on as well.
            cmc = cmc+noLines;
            if(lines.contains(t.getLine())) {
                cmc = cmc-1;
            }
        }
    }
}

//determine the number of blank lines, i.e a line that is empty.
try {
    BufferedReader reader = new BufferedReader(new FileReader(new File(info.getFileName())));
    String line;
    while((line = reader.readLine()) != null) {
        if(line.isEmpty()) {
            bc++;
        }
        to++;
    }
} catch (Exception e) {
    throw new MetricInitialisationException(e.getMessage());
}

//set the calculated values to the values used in the result.
this.blankCount = bc;
this.commentCount = cmc;
this.codeCount = to-(bc+cmc);
}

```

Figure 21: A snippet of the Comment Ratio class which performs lexical analysis using the provided Lexer to determine the comment to code ratio.

```

@Override
public Result getResults() throws InvalidResultException {

    //determine which method had the most complexity and whether it went over the
    //threshold.
    Entry e = new Entry();

    //if there are entries to use.
    if(!entries.isEmpty()) {
        //if there is more than one method/constructor.
        if(entries.size() > 1) {
            //sort the entries in natural order based on the amount of
            //complex keywords.
            Collections.sort(entries);
            e = entries.get(entries.size()-1); //get the last object.
        } else {
            //get the only entry and see how well it done.
            e = entries.get(0);
        }
    }

    return Result.newInstance(fileName, sourceLang, this.getClass().getSimpleName(), e.toResult(),
        (e.getAmountOfComplexKeywords() <= this.complexityThreshold
         && e.getAmountOfComplexKeywords() != -1));
}

@Override
public void onParserEvent(EventState state) {
    if(Arrays.asList(methodEvents).contains(state.getEventType())) {
        //if we're in a constructor or a method, count the amount of complexity keywords in that
        //method or constructor, store these in a ArrayList.
        Entry currentEntry = new Entry();
        currentEntry.setContext(state.getContext());
        currentEntry.setMethodName(state.getContext().getChild(1).getText());
        currentEntry.setType(state.getEventType());
        currentEntry.setAmountOfComplexKeywords(0);
        currentEntry.setComplexityThreshold(complexityThreshold);
        this.entries.add(currentEntry);

    } else if (Arrays.asList(statementEvents).contains(state.getEventType())) {
        //statements only appear inside constructors or methods
        //so we can safely assume the last entry in our entries is the
        //current method/constructor.
        if(!this.entries.isEmpty()) {
            Entry e = this.entries.get(this.entries.size()-1);
            this.determineComplexity(state.getContext(), e);
        }
    }
}

/**
 * determines the complexity of the class as a whole.
 * @param context the statement context.
 */
private void determineComplexity(ParserRuleContext context, Entry entry) {
    for(String s : this.complexityTokens) {
        if(s.equalsIgnoreCase(context.getStart().getText().toUpperCase())) {
            int c = entry.getAmountOfComplexKeywords();
            entry.setAmountOfComplexKeywords(c+1);
            entry.addKeywordOccurrence(context.getStart().getText());
            break;
        }
    }
}
}

```

Figure 22: The implementation of the cyclomatic Complexity metric



```

/**
 * this is the interface that all metrics should implement.
 * @author Jack Timblin - U1051575
 */
public interface MetricInterface {

    /**
     * returns the results from this metric, this will
     * be the results when this metric is run on a single class.
     * @return the results from running the metric.
     * @throws InvalidResultException if the returned result is invalid.
     */
    public Result getResults() throws InvalidResultException;

    /**
     * Called when an event is triggered while walking the parse tree.
     * @param state the state of the event.
     */
    public void onParserEvent(EventState state);

    /**
     * give the metric initial information regarding the file
     * it is evaluating.
     * @param info all the initial information a metric needs.
     * @throws MetricInitialisationException when a fatal error occurs initialising the metric.
     */
    public void init(ParserInfo info) throws MetricInitialisationException;

    /**
     * called after the analysis is completed on a single file, can be used
     * to reset variables etc.
     */
    public void destroy();
}

```

Figure 23: The MetricInterface interface that all 'main' metric classes should implement to be 'autoloaded' into the application.

```

/**
 * returns the Parser instance that is supported by the language of this file.
 * This is done by checking the file extension type of the file and matching it against
 * supported the supported languages.
 * @param stream The stream of tokens provided after lexical analysis has been done.
 * @return a Parser instance in the generic form of a ParserInterface.
 * @throws org.codeanalyser.core.analyser.FileAnalyser.UnsupportedLanguageException if the language was not
 * supported, the Parser could not be instantiated properly or did not implement ParserInterface.
 */
public ParserInterface getSupportedParser(TokenStream stream) throws UnsupportedLanguageException {

    String[] names = this.getClassNames();

    try {
        Constructor c = Class.forName("org.codeanalyser.language."+names[0]+"."+names[1]+"Parser").getConstructor(TokenStream.class);
        return (ParserInterface) c.newInstance(stream);
    } catch (Exception e) {
        throw new UnsupportedLanguageException(e.getMessage());
    }

}

/**
 * returns the Lexer instance that is supported by the language of this file.
 * This is done by checking the file extension type of the file and matching it against
 * supported the supported languages.
 * @return An instance of a Lexer supported by this language cast to its superclass which can be
 * used generically.
 * @throws org.codeanalyser.core.analyser.FileAnalyser.UnsupportedLanguageException if the language was not
 * supported or the Lexer could not be instantiated properly.
 */
public Lexer getSupportedLexer() throws UnsupportedLanguageException {

    String[] names = this.getClassNames();

    try {
        Constructor c = Class.forName("org.codeanalyser.language."+names[0]+"."+names[1]+"Lexer").getConstructor(CharStream.class);
        return (Lexer) c.newInstance(new ANTLRFileStream(this.getAbsolutePath()));
    } catch (Exception e) {
        throw new UnsupportedLanguageException(e.getMessage());
    }

}

/**
 * returns the parseTreeListener instance that supports this file. This is calculated by
 * if the file extension matches a package.
 * @return a ParseTreeListener that supports this language
 * @throws org.codeanalyser.core.analyser.FileAnalyser.UnsupportedLanguageException if the language was not
 * supported or the Listener could not be instantiated properly.
 */
public ParseTreeListener getSupportedListener() throws UnsupportedLanguageException
{
    String[] names = this.getClassNames();

    try {
        ParseTreeListener listener = (ParseTreeListener) Class.forName("org.codeanalyser.language."+names[0]+"."+names[1]+"BaseListener").newInstance();
        ((ListenerInterface)listener).init(this);
        return listener;
    } catch (Exception e) {
        throw new UnsupportedLanguageException(e.getMessage());
    }

}

```

Figure 24: The methods that attempt to retrieve the supported Parser/Lexer instances based on the current files detected language.

```

    /**
     * An CustomANTLRErrorListener which count the amount of syntax errors
     * that occur in the parser or lexer. We can see if the errors array is empty to
     * determine if any syntax errors occurred.
     * @author Jack Timblin - U1051575
     */
    public class SyntaxErrorAdapter implements ANTLRErrorListener {

        private final ArrayList<String> errors;
        private final FileAnalyser file;

        /**
         * initialises a new SyntaxErrorAdapter
         * @param file the file currently being analysed.
         */
        public SyntaxErrorAdapter(FileAnalyser file) {
            this.errors = new ArrayList<String>();
            this.file = file;
        }

        /**
         * gets the syntax errors that occurred while parsing
         * the file currently being analysed.
         * @return the syntax error message strings.
         */
        public ArrayList<String> getSyntaxErrors() {
            return this.errors;
        }

        /**
         * triggered when a syntax error occurs.
         * @param rcgnzr the recogniser used, you can access the context.
         * @param offendingSymbol The offending token in the input token stream.
         * @param line The line number in the input where the error occured.
         * @param charPositionInLine The character position within the line where
         * the error occurred.
         * @param msg the error message.
         * @param e The exception generated by the parser.
         */
        @Override
        public void syntaxError(Recognizer<?, ?> rcgnzr, Object offendingSymbol,
                               int line, int charPositionInLine, String msg, RecognitionException e) {
            this.errors.add("File: " + file.getAbsolutePath() + ", Error: " + msg);
        }
    }

```

Figure 25: The SyntaxErrorAdapter that is used to keep track of syntax errors that occur while attempting to parse a file.

```

//src
// Analyses the source location provided in the constructor.
//
public void analyse() {
    for (FileAnalyser file : this.filesToAnalyse) {
        try {
            System.out.println("Started Analysing File: " + file.getAbsolutePath());
            //generate parse tree from source file.
            SyntaxErrorAdapter ea = new SyntaxErrorAdapter(file);

            Lexer lexer = file.getSupportedLexer();
            lexer.addErrorListener(ea);
            CommonTokenStream tokens = new CommonTokenStream(lexer);
            ParserInterface parser = file.getSupportedParser(tokens);
            parser.addErrorListener(ea);
            ParserRuleContext tree = parser.compilationUnit();

            //check to see if any syntactical errors occurred.
            if (!ea.getSyntaxErrors().isEmpty()) {
                this.result.incrementNoOfSyntaxErrorFiles();
                for (String er : ea.getSyntaxErrors()) {
                    this.result.addSyntaxError(er);
                }
                throw new SyntaxErrorException("A Syntax Error Occured Parsing File: " + file.getAbsolutePath());
            }

            ParseTreeListener listener = file.getSupportedListener();
            ParseTreeWalker walker = new ParseTreeWalker();

            //walk the parse tree calling methods in the metrics.
            walker.walk(listener, tree);

            ArrayList<Result> re = ((ListenerInterface) listener).getResults();
            //gather the results from the analysis.
            OverallResult res = new OverallResult(
                re,
                file.getAbsolutePath()
            );

            //add this to result array.
            this.result.addResult(res);

            //call destroy on metrics.
            ((ListenerInterface) listener).destroy();

        } catch (FileAnalyser.UnsupportedLanguageException e) {
            this.result.addUnsupportedFile(file.getAbsolutePath());
            System.out.println(e.getMessage());
        } catch (NoResultsDefinedException e) {
            System.out.println(e.getMessage());
        } catch (SyntaxErrorException e) {
            System.out.println(e.getMessage());
        } catch (InvalidResultException e) {
            System.out.println(e.getMessage());
        }
    }

    try {
        System.out.println("Generating Output");
        //render the output for this analysis.
        this.output.generateOutput(this.result);
    } catch (Exception e) {
        System.out.println(e.getMessage());
    }
}

```

Figure 26: The analyse method in the Analyser class, which takes in a source location and attempts to analyse the file by first initialising the supported parser/lexer and listener and then generating a parse tree. It then walks the parse tree and then collects the results from the metrics.

```

public class BaseListener extends JavaBaseListener implements ListenerInterface {

    private ArrayList<MetricInterface> metrics;
    private FileAnalyser file;

    @Override
    public void init(FileAnalyser file) throws MetricException {
        this.file = file;
        metrics = new ArrayList<MetricInterface>();
        //initialise the metrics.
        try {
            ParserInfo info = new ParserInfo(file);
            for (String metric : Application.getMetricsList()) {
                MetricInterface m = (MetricInterface) Class.forName(metric).newInstance();
                try {
                    m.init(info);
                } catch (MetricInitialisationException e) {
                    System.out.println("Failed to Initialise Metric: \"" + metric + "\", "
                        + "Error: " + e.getMessage());
                    continue;
                }
                metrics.add(m);
            }
        } catch (Exception e) {
            throw new MetricException(e.getMessage());
        }
    }

    /**
     * gathers the results from the metrics in this Listener.
     * @return an ArrayList<Result> of string results from each of the metrics.
     */
    @Override
    public ArrayList<Result> getResults() {
        ArrayList<Result> results = new ArrayList<Result>();
        for (MetricInterface mi : metrics) {
            try {
                Result r = mi.getResults();
                if (r != null) {
                    results.add(r);
                }
            } catch (InvalidResultException e) {
                System.out.println(e.getMessage());
            }
        }
        return results;
    }

    /**
     * generated method to call enterBlock while walking the parse tree.
     * @param context <ps>The context/area of the parse tree that this rule applies to.</ps>
     */
    @Override
    public void enterBlock(JavaParser.BlockContext context) {
        //build state object.
        EventState.EventStateBuilder builder = new EventState.EventStateBuilder();
        EventState state = builder.setContext(context).setEventType("ENTER_BLOCK").build();
        for (MetricInterface metric : metrics) {
            //start the metrics evaluation at this event.
            metric.onParserEvent(state);
        }
    }

    @Override
    public void destroy() {
        //do any more work before destruction.
        for (MetricInterface metric : metrics) {
            metric.destroy();
        }
    }
}

```

Figure 27: A snippet of the BaseListener used to initialise metrics and then call the onParserEvent method when a rule is found by ANTR when walking the parse tree. You can see a the enterBlock method which is triggered when the walker enters a rule matching 'block' as defined in the grammar. This is why each grammar has to have the rule names, so the triggered events are the same.

## 6.2.2 Application Tests

These tests ensure that the System properties class `Application` provides the correct information.

```
/**
 * Tests the system property Application class.
 * @author Jack Timblin - U1051575
 */
public class ApplicationTest {

    /**
     * Test of getMetricsList method, of class Application.
     */
    @Test
    public void testGetMetricsList() {
        System.out.println("getMetricsList");
        ArrayList<String> result = Application.getMetricsList();
        if(result.isEmpty()) {
            fail("metricsList should not be empty");
        }
        System.out.println(result);
    }

    /**
     * Test of getSupportedLanguages method, of class Application.
     */
    @Test
    public void testGetSupportedLanguages() {
        System.out.println("getSupportedLanguages");
        ArrayList<String> result = Application.getSupportedLanguages();
        if(result.isEmpty()) {
            fail("supportedLanguages should not be empty");
        }
        System.out.println(result);
    }

    /**
     * tests hasInternetConnection of class Application.
     */
    @Test
    public void testHasInternetConnection() {
        System.out.println("hasInternetConnection");
        boolean hasInternet = Application.hasInternetConnection();
        assertTrue(hasInternet);
    }
}
```

## 6.2.3 Analyser Tests

These tests ensure that the analyser functions correctly with some tester data.

```
/**
 * Test of analyse method, of class Analyser.
 */
@Test
public void testAnalyseForced() {
    System.out.println("analyseForced");
    analyser.analyse(FORCED);
    File f = new File(OUTPUT+"output.html");
    assertTrue(f.exists());

    //I know there is 2 java files and one 'hello' file
    //in the SOURCE location, so because this test has been forced
    //to FORCED, there will be some syntax errors.
    AnalyserResult result = analyser.getResults();
    //System.out.println("NoOfErrorFiles: " + result.getNoOfSyntaxErrorFiles());
    assertTrue(result.getNoOfSyntaxErrorFiles() == 4);
}

/**
 * Test of analyse method, of class Analyser.
 */
@Test
public void testAnalyse() {
    System.out.println("analyse");
    analyser.analyse();

    //test output was successfully generated.
    File f = new File(OUTPUT+"output.html");
    assertTrue(f.exists());

    //I know all files are supported
    //and it contains one file with a syntax error so the success/fail
    //should be 2/1.
    int[] expected = {6, 1};
    int[] result = analyser.getResults().getSuccessFailFigures();
    System.out.println(Arrays.toString(result));
    //assertArrayEquals(expected, result);
}
```

### 6.2.4 Language Detection Test

These tests ensure that we can detect a language based on the files extension.

```
public class LanguageDetectTest {

    private LanguageDetect detect;
    private final String SOURCE = "hello";
    private final String EXTENSION = "hello";
    private final int SUPPORTED_COUNT = 247;
    private final String UNSUPPORTED_EXTENSION = "unsupported";

    @Before
    public void setUp() {
        detect = new LanguageDetect();
    }

    @After
    public void tearDown() {
        detect = null;
    }

    /**
     * Test of getAllDetectableLanguages method, of class LanguageDetect.
     */
    @Test
    public void testGetAllDetectableLanguages() {
        System.out.println("getAllDetectableLanguages");
        ArrayList<Language> result = detect.getAllDetectableLanguages();
        assertEquals(SUPPORTED_COUNT, result.size());
    }

    /**
     * Test of getSupportedLanguage method, of class LanguageDetect.
     */
    @Test
    public void testGetSupportedLanguage() {
        System.out.println("getSupportedLanguage");
        Language result = detect.getSupportedLanguage(EXTENSION);
        assertNotNull(result);
        assertEquals(SOURCE, result.getName());
        Language falseResult = detect.getSupportedLanguage(UNSUPPORTED_EXTENSION);
        assertNull(falseResult);
    }

}
```

### 6.2.5 Parse Tree Construction Tests

These tests ensure that we can still generate a valid parse tree for all of the supported languages in the application.

```
@Test
public void testHelloParseTreeConstruction() {
    System.out.println("testHelloParseTreeConstruction");

    try {

        FileAnalyser helloFile = new FileAnalyser("testData/Test.hello");
        Lexer l = helloFile.getSupportedLexer();
        CommonTokenStream s = new CommonTokenStream(l);
        ParserInterface p = helloFile.getSupportedParser(s);

        ParserRuleContext tree = p.compilationUnit();
        System.out.println(tree.toStringTree((Parser)p));

        //assert that there are 'nodes' on the tree and that
        //the string representation is not an empty string.
        assertTrue(!tree.toStringTree((Parser)p).isEmpty() && tree.getChildCount() != 0);

    } catch (FileAnalyser.UnsupportedLanguageException e) {
        fail(e.getMessage());
    }

}
```

```

@Test
public void testJavaParseTreeConstruction() {
    System.out.println("testJavaParseTreeConstruction");

    try {
        FileAnalyser helloFile = new FileAnalyser("testData/Test2.java");
        Lexer l = helloFile.getSupportedLexer();
        CommonTokenStream s = new CommonTokenStream(l);
        ParserInterface p = helloFile.getSupportedParser(s);

        ParserRuleContext tree = p.compilationUnit();
        System.out.println(tree.toStringTree((Parser)p));

        //assert that there are 'nodes' on the tree and that
        //the string representation is not an empty string.
        assertTrue(!tree.toStringTree((Parser)p).isEmpty() && tree.getChildCount() != 0);
    } catch (FileAnalyser.UnsupportedLanguageException e) {
        fail(e.getMessage());
    }
}

```

## 6.2.6 Syntax Error Adapter Tests

These tests ensure that the application correctly records any syntax errors that occur.

```

public class SyntaxErrorAdapterTest {

    private FileAnalyser fileSyntaxError;

    @Before
    public void setUp() {
        this.fileSyntaxError = new FileAnalyser("testData/TestSyntaxError.java");
    }

    @After
    public void tearDown() {
    }

    /**
     * Test of getSyntaxErrors method, of class SyntaxErrorAdapter.
     */
    @Test
    public void testHasSyntaxErrors() {
        System.out.println("hasSyntaxErrors");
        try {
            //set up a test parse with a file that has syntax errors.
            SyntaxErrorAdapter instance = new SyntaxErrorAdapter(fileSyntaxError);
            Lexer lexer = fileSyntaxError.getSupportedLexer();
            CommonTokenStream s = new CommonTokenStream(lexer);
            ParserInterface parser = fileSyntaxError.getSupportedParser(s);
            lexer.addErrorListener(instance);
            parser.addErrorListener(instance);

            //generate parse tree.
            ParserRuleContext context = parser.compilationUnit();
            //check that a syntax error was recorded.
            assertTrue(instance.getSyntaxErrors().size() > 0);
        } catch (FileAnalyser.UnsupportedLanguageException e) {
            fail("Could not initialise parser or lexer.");
        }
    }
}

```



### 6.2.7 Metric Initialisation Tests

These tests ensure that the metrics are initialised correctly.

```
public class MetricInitialisationTest {

    private FileAnalyser file;
    private ListenerInterface listener;

    @Before
    public void setUp() {
        try {
            this.file = new FileAnalyser("testData/Test2.java");
            listener = (ListenerInterface)file.getSupportedListener();
        } catch (FileAnalyser.UnsupportedLanguageException e) {
            fail(e.getMessage());
        } catch (ClassCastException e) {
            fail(e.getMessage());
        }
    }

    @After
    public void tearDown() {
        if(listener != null) {
            listener.destroy();
        }
        this.file = null; this.listener = null;
    }

    @Test
    public void testMetricInitialisation() {
        System.out.println("metricInitialisation");
        try {
            //attempt to cast this to ListenerInterface to call init,
            //which initialises all of the installed metrics.
            listener.init(file);
        } catch (MetricException e) {
            fail(e.getMessage());
        }
    }
}
```

### 6.2.8 Notifying metrics When a Rule is Matched Walking the Parse Tree

These tests ensure that we can call `onParserEvent` in the metric classes when a rule is matched while walking the generated parse tree. These tests use a tester metric which just outputs to the console when an event when an event is triggered. Sample output for these tests is shown in figure 28 on page 66.

```

@Before
public void setUp() {
    javaFile = new FileAnalyser("testData/Test2.java");
    helloFile = new FileAnalyser("testData/Test.hello");
}

@After
public void tearDown() {
    javaFile = null; helloFile = null;
}

@Test
public void testWalkJavaParseTree() {
    try {
        System.out.println("walkJavaParseTree");
        //construct the parser, lexer, listener and walker.
        Lexer l = javaFile.getSupportedLexer();
        CommonTokenStream s = new CommonTokenStream(l);
        ParserInterface p = javaFile.getSupportedParser(s);
        TesterJavaListener jl = new TesterJavaListener();
        ParseTreeWalker walker = new ParseTreeWalker();

        //call init in the listener to initialise the metrics.
        jl.init(javaFile);

        //generate the parse tree.
        ParserRuleContext tree = p.compilationUnit();

        //walk the tree.
        walker.walk(jl, tree);

        //call destroy on the metrics.
        jl.destroy();
    } catch (Exception e) {
        fail(e.getMessage());
    }
}

@Test
public void testWalkHelloParseTree() {
    try {
        System.out.println("walkHelloParseTree");
        //construct the parser, lexer, listener and walker.
        Lexer l = helloFile.getSupportedLexer();
        CommonTokenStream s = new CommonTokenStream(l);
        ParserInterface p = helloFile.getSupportedParser(s);
        TesterHelloListener jl = new TesterHelloListener();
        ParseTreeWalker walker = new ParseTreeWalker();

        //call init in the listener to initialise the metrics.
        jl.init(helloFile);

        //generate the parse tree.
        ParserRuleContext tree = p.compilationUnit();

        //walk the tree.
        walker.walk(jl, tree);

        //call destroy on the metrics.
        jl.destroy();
    } catch (Exception e) {
        fail(e.getMessage());
    }
}

```

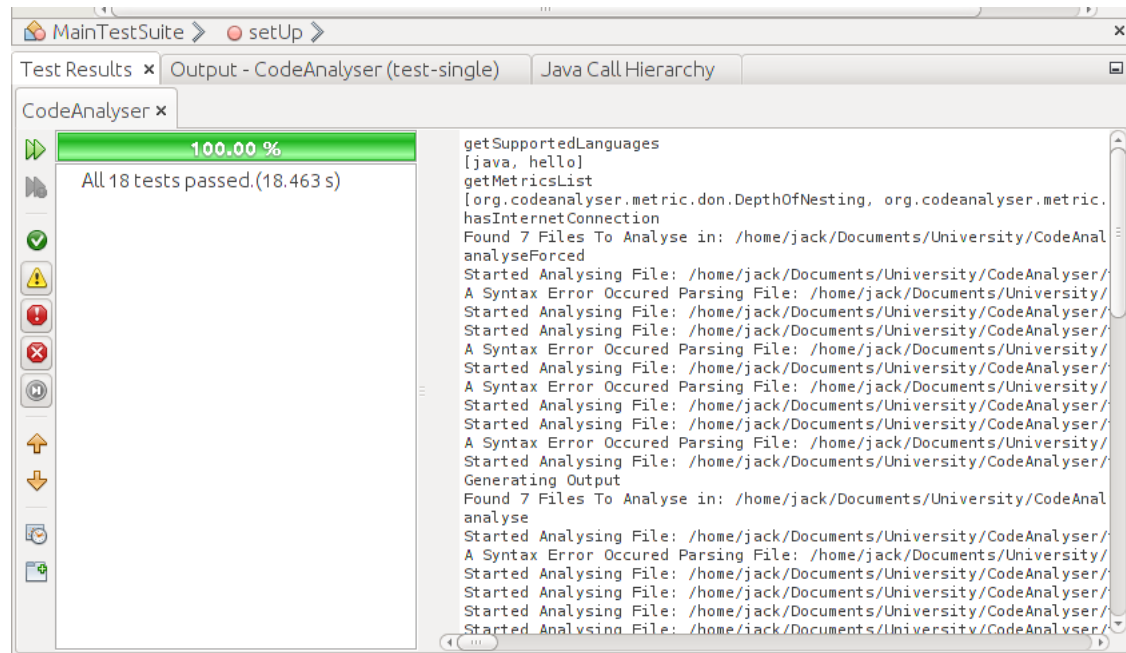
```

walkJavaParseTree
init() called: SOURCE: Java
onParserEvent() called: EVENT: ENTER_CLASS_DECLARATION
onParserEvent() called: EVENT: ENTER_METHOD_DECLARATION
onParserEvent() called: EVENT: ENTER_VARIABLE_DECLARATOR_ID
onParserEvent() called: EVENT: ENTER_VARIABLE_DECLARATOR_ID
onParserEvent() called: EVENT: ENTER_METHOD_BODY
onParserEvent() called: EVENT: ENTER_STATEMENT
onParserEvent() called: EVENT: ENTER_STATEMENT
onParserEvent() called: EVENT: ENTER_STATEMENT
onParserEvent() called: EVENT: ENTER_STATEMENT
onParserEvent() called: EVENT: ENTER_STATEMENT
onParserEvent() called: EVENT: ENTER_STATEMENT
onParserEvent() called: EVENT: ENTER_METHOD_DECLARATION
onParserEvent() called: EVENT: ENTER_VARIABLE_DECLARATOR_ID
onParserEvent() called: EVENT: ENTER_VARIABLE_DECLARATOR_ID
onParserEvent() called: EVENT: ENTER_METHOD_BODY
onParserEvent() called: EVENT: ENTER_STATEMENT
onParserEvent() called: EVENT: ENTER_STATEMENT
onParserEvent() called: EVENT: ENTER_STATEMENT
onParserEvent() called: EVENT: ENTER_STATEMENT
onParserEvent() called: EVENT: ENTER_STATEMENT
onParserEvent() called: EVENT: ENTER_STATEMENT
onParserEvent() called: EVENT: ENTER_SWITCH_BLOCK_STATEMENT_GROUP
onParserEvent() called: EVENT: ENTER_STATEMENT
onParserEvent() called: EVENT: ENTER_STATEMENT
onParserEvent() called: EVENT: ENTER_SWITCH_BLOCK_STATEMENT_GROUP
onParserEvent() called: EVENT: ENTER_STATEMENT
onParserEvent() called: EVENT: ENTER_CONSTRUCTOR_DECLARATION
destroy() called

```

Figure 28: Output from the MetricParseTreeWalkTests, which shows all of the events that the tester metric was notified for while we walked the parse tree. It also shows the calls for init and destroy.

### 6.2.9 Unit Tests Output



## 6.3 System Testing

The system testing was carried out after the initial implementation of the product was completed to check that it could firstly complete the task that it was initially designed for, and to also handle error robustly and effectively.

As described in the testing strategy above, this will involve:

- Testing the analyser with valid input that is considered to be of good quality in 2 supported languages
- Testing the analyser with valid input that is considered to be of poor quality in 2 supported languages
- Testing the analyser with input that contains syntax errors.
- Testing the analyser with input that is unsupported.

Obviously the two supported languages that will be tested are Java and the 'hello' language.

### 6.3.1 Testing with valid input considered to be good quality

```
jack@jack-Inspiron-1525:~$ java -jar Documents/University/CodeAnalyser/dist/CodeAnalyser.jar analyser --source Documents/University/CodeAnalyser/testData/systemTesting/good --output good/
Found 2 Files To Analyse in: /home/jack/Documents/University/CodeAnalyser/testData/systemTesting/good
Started Analysing File: /home/jack/Documents/University/CodeAnalyser/testData/systemTesting/good/GoodQuality.hello
Started Analysing File: /home/jack/Documents/University/CodeAnalyser/testData/systemTesting/good/GoodQuality.java
Generating Output
jack@jack-Inspiron-1525:~$
```

## Code Quality Analyser Results.

### Overall Result:

Amount of Files Analysed: 2

Was Overall Successful: **SUCCESS**

Successful/Failed: 2/0

### File Statistics:

Hello 50.0%

Java 50.0%

---

### Metric Results

**Filename:** /home/jack/Documents/University/CodeAnalyser/testData/systemTesting/good/GoodQuality.hello

**Source Language:** hello

**Overall Result:** **Passed**

[Metric Results](#)

**Filename:** /home/jack/Documents/University/CodeAnalyser/testData/systemTesting/good/GoodQuality.java

**Source Language:** java

**Overall Result:** **Passed**

[Metric Results](#)

---

### Unsupported Files

No Files Were Unsupported.

### Syntax Errors

No Files Contained Syntax Errors.

## Code Quality Analyser Results - GoodQuality.hello

### Metric Results

<b>Metric Name:</b> CyclomaticComplexity	<b>Was Successful:</b> Success
--	--------------------------------

#### Comments:

CyclomaticComplexity Results:  
Method With Most Complexity: setAnotherVariable  
Amount of Complexity Keywords Found: 3  
Complexity Threshold: 10

#### Keyword Occurances:

default:	1
case:	2

<b>Metric Name:</b> CommentRatio	<b>Was Successful:</b> Success
----------------------------------	--------------------------------

#### Comments:

CommentRatio Results:  
Code Percentage: 65.0%  
Comment Percentage: 31.0%  
Blank Lines Percentage: 4.0%

#### Thresholds:

Min: 20%  
Max: 40%

<b>Metric Name:</b> DepthOfNesting	<b>Was Successful:</b> Success
------------------------------------	--------------------------------

#### Comments:

DepthOfNesting Results:  
Method With Most Nesting: setAnotherVariable  
Biggest Nesting That Occured: 2  
Nesting Threshold: 4

#### Source Code Text:

```
switch(one){case"1">{print(one);}case"2"{print(two);}default{print(one,two);}}
```

<b>Metric Name:</b> LinesOfCode	<b>Was Successful:</b> Success
---------------------------------	--------------------------------

#### Comments:

LinesOfCode Result:  
Lines Of Code: 25  
Threshold: 100

<b>Metric Name:</b> NumberOfMethods	<b>Was Successful:</b> Success
-------------------------------------	--------------------------------

#### Comments:

Number of Methods In Class: 1  
Method Threshold: 10

<b>Metric Name:</b> ProcedureDeclarationLength	<b>Was Successful:</b> Success
--	--------------------------------

#### Comments:

ProcedureDeclarationLength Results:  
Amount of Methods Within Threshold: 1  
Worst defined procedure declaration: setAnotherVariable

#### Thresholds:

Method Length: 20  
No Of Variables: 5

<b>Metric Name:</b> VariableNamingConventions	<b>Was Successful:</b> Success
---	--------------------------------

#### Comments:

VariableNamingConventions Results:  
Determined Overall Language of Variable Names: en  
Total Variables Found: 1

<b>Metric Name:</b> WeightedMethodCount	<b>Was Successful:</b> Success
---	--------------------------------

#### Comments:

WeightedMethodCount Result:  
Total Amount of Complexity Keywords Found: 3  
Complexity Threshold: 50

## Code Quality Analyser Results - GoodQuality.java

### Metric Results

<b>Metric Name:</b> CyclomaticComplexity	<b>Was Successful:</b> <b>Success</b>
<b>Comments:</b> CyclomaticComplexity Results: Method With Most Complexity: setAnotherVariable Amount of Complexity Keywords Found: 3 Complexity Threshold: 10	
<b>Keyword Occurances:</b>	
default:	1
case:	2
<b>Metric Name:</b> CommentRatio	<b>Was Successful:</b> <b>Success</b>
<b>Comments:</b> CommentRatio Results: Code Percentage: 69.0% Comment Percentage: 31.0% Blank Lines Percentage: 0.0%	
<b>Thresholds:</b> Min: 20% Max: 40%	
<b>Metric Name:</b> DepthOfNesting	<b>Was Successful:</b> <b>Success</b>
<b>Comments:</b> DepthOfNesting Results: Method With Most Nesting: setAnotherVariable Biggest Nesting That Occured: 1 Nesting Threshold: 4	
<b>Source Code Text:</b> { switch(one){ case1.two=1;break;case2.two=2;break;default.two=one;break; } }	
<b>Metric Name:</b> LinesOfCode	<b>Was Successful:</b> <b>Success</b>
<b>Comments:</b> LinesOfCode Result: Lines Of Code: 39 Threshold: 100	
<b>Metric Name:</b> NumberOfMethods	<b>Was Successful:</b> <b>Success</b>
<b>Comments:</b> Number of Methods In Class: 1 Method Threshold: 10	
<b>Metric Name:</b> ProcedureDeclarationLength	<b>Was Successful:</b> <b>Success</b>
<b>Comments:</b> ProcedureDeclarationLength Results: Amount of Methods Within Threshold: 1 Worst defined procedure declaration: setAnotherVariable	
<b>Thresholds:</b> Method Length: 20 No Of Variables: 5	
<b>Metric Name:</b> VariableNamingConventions	<b>Was Successful:</b> <b>Success</b>
<b>Comments:</b> VariableNamingConventions Results: Determined Overall Language of Variable Names: en Total Variables Found: 5	
<b>Metric Name:</b> WeightedMethodCount	<b>Was Successful:</b> <b>Success</b>
<b>Comments:</b> WeightedMethodCount Result: Total Amount of Complexity Keywords Found: 3 Complexity Threshold: 50	

### 6.3.2 Testing with valid input considered to be poor quality

```
jack@jack-Inspiron-1525:~$ java -jar Documents/University/CodeAnalyser/dist/CodeAnalyser.jar analyser --source Documents/University/CodeAnalyser/testData/systemTesting/bad --output good/
Found 2 Files To Analyse in: /home/jack/Documents/University/CodeAnalyser/testData/systemTesting/bad
Started Analysing File: /home/jack/Documents/University/CodeAnalyser/testData/systemTesting/bad/BadQuality.java
Started Analysing File: /home/jack/Documents/University/CodeAnalyser/testData/systemTesting/bad/BadQuality.hello
Generating Output
jack@jack-Inspiron-1525:~$
```

## Code Quality Analyser Results.

### Overall Result:

Amount of Files Analysed: 2

Was Overall Successful: **FAILED**

Successful/Failed: 0/2

### File Statistics:

Java 50.0%

Hello 50.0%

---

### Metric Results

**Filename:** /home/jack/Documents/University/CodeAnalyser/testData/systemTesting/bad/BadQuality.java

**Source Language:** java

**Overall Result:** Failed

[Metric Results](#)

**Filename:** /home/jack/Documents/University/CodeAnalyser/testData/systemTesting/bad/BadQuality.hello

**Source Language:** hello

**Overall Result:** Failed

[Metric Results](#)

---

### Unsupported Files

No Files Were Unsupported.

### Syntax Errors

No Files Contained Syntax Errors.



## Code Quality Analyser Results - BadQuality.hello

### Metric Results

<b>Metric Name:</b> CyclomaticComplexity	<b>Was Successful:</b> <b>Failure</b>
<b>Comments:</b> CyclomaticComplexity Results: Method With Most Complexity: setAnotherVariable Amount of Complexity Keywords Found: 12 Complexity Threshold: 10 <b>Keyword Occurances:</b> default: 1 if: 8 case: 3	
<b>Metric Name:</b> CommentRatio	<b>Was Successful:</b> <b>Failure</b>
<b>Comments:</b> CommentRatio Results: Code Percentage: 88.0% Comment Percentage: 12.0% Blank Lines Percentage: 0.0% <b>Thresholds:</b> Min: 20% Max: 40%	
<b>Metric Name:</b> DepthOfNesting	<b>Was Successful:</b> <b>Failure</b>
<b>Comments:</b> DepthOfNesting Results: Method With Most Nesting: setAnotherVariable Biggest Nesting That Occured: 5 Nesting Threshold: 4 <b>Source Code Text:</b> if(fjdfn){if(fjdfn>ffgfgfgh){if(fjdfn	
<b>Metric Name:</b> LinesOfCode	<b>Was Successful:</b> <b>Success</b>
<b>Comments:</b> LinesOfCode Result: Lines Of Code: 49 Threshold: 100	
<b>Metric Name:</b> NumberOfMethods	<b>Was Successful:</b> <b>Success</b>
<b>Comments:</b> Number of Methods In Class: 1 Method Threshold: 10	
<b>Metric Name:</b> ProcedureDeclarationLength	<b>Was Successful:</b> <b>Success</b>
<b>Comments:</b> ProcedureDeclarationLength Results: Amount of Methods Within Threshold: 1 Worst defined procedure declaration: setAnotherVariable <b>Thresholds:</b> Method Length: 20 No Of Variables: 5	
<b>Metric Name:</b> VariableNamingConventions	<b>Was Successful:</b> <b>Failure</b>
<b>Comments:</b> VariableNamingConventions Results: Determined Overall Language of Variable Names: en Reason For Failure: Confidence in Language Detection was too low. Total Variables Found: 5	
<b>Metric Name:</b> WeightedMethodCount	<b>Was Successful:</b> <b>Success</b>
<b>Comments:</b> WeightedMethodCount Result: Total Amount of Complexity Keywords Found: 12 Complexity Threshold: 50	

## Code Quality Analyser Results - BadQuality.java

### Metric Results

**Metric Name:** CyclomaticComplexity **Was Successful:** **Failure**

#### Comments:

CyclomaticComplexity Results:  
Method With Most Complexity: setAnotherVariable  
Amount of Complexity Keywords Found: 15  
Complexity Threshold: 10

#### Keyword Occurrences:

default:	1
for:	2
if:	8
continue:	1
case:	2
while:	1

**Metric Name:** CommentRatio **Was Successful:** **Failure**

#### Comments:

CommentRatio Results:  
Code Percentage: 91.0%  
Comment Percentage: 9.0%  
Blank Lines Percentage: 0.0%

#### Thresholds:

Min: 20%  
Max: 40%

**Metric Name:** DepthOfNesting **Was Successful:** **Failure**

#### Comments:

DepthOfNesting Results:  
Method With Most Nesting: setAnotherVariable  
Biggest Nesting That Occured: 5  
Nesting Threshold: 4

#### Source Code Text:

```
(System.out.println("Printing While True");if(vfgfg==10){this.fjdfn=1;}this.fjdfn-=1;)
```

**Metric Name:** LinesOfCode **Was Successful:** **Success**

#### Comments:

LinesOfCode Result:  
Lines Of Code: 64  
Threshold: 100

**Metric Name:** NumberOfMethods **Was Successful:** **Success**

#### Comments:

Number of Methods In Class: 1  
Method Threshold: 10

**Metric Name:** ProcedureDeclarationLength **Was Successful:** **Failure**

#### Comments:

ProcedureDeclarationLength Results:  
Amount of Methods Within Threshold: 0  
Worst defined procedure declaration: setAnotherVariable

#### Thresholds:

Method Length: 20  
No Of Variables: 5

**Metric Name:** VariableNamingConventions **Was Successful:** **Failure**

#### Comments:

VariableNamingConventions Results:  
Determined Overall Language of Variable Names: en  
Reason For Failure: Confidence in Language Detection was too low.  
Total Variables Found: 14

**Metric Name:** WeightedMethodCount **Was Successful:** **Success**

#### Comments:

WeightedMethodCount Result:  
Total Amount of Complexity Keywords Found: 15  
Complexity Threshold: 50

### 6.3.3 Testing with input that contains syntax errors

```
jack@jack-Inspiron-1525:~$ java -jar Documents/University/CodeAnalyser/dist/CodeAnalyser.jar analyser --source Documents/University/CodeAnalyser/testData/systemTesting/TestSyntaxError.java --output good/
Found 1 Files To Analyse in: /home/jack/Documents/University/CodeAnalyser/testData/systemTesting/TestSyntaxError.java
Started Analysing File: /home/jack/Documents/University/CodeAnalyser/testData/systemTesting/TestSyntaxError.java
line 8:22 missing ')' at ';'
A Syntax Error Occured Parsing File: /home/jack/Documents/University/CodeAnalyser/testData/systemTesting/TestSyntaxError.java
Generating Output
No Results Defined From Running The Metrics.
jack@jack-Inspiron-1525:~$
```

## Code Quality Analyser Results.

**Overall Result:**  
Amount of Files Analysed: 1  
Was Overall Successful: **FAILED**  
Successful/Failed: 0/1

**File Statistics:**  
Java 100.0%

---

**Metric Results**  
No Results were defined.

---

**Unsupported Files**  
No Files Were Unsupported.

**Syntax Errors**  
File: /home/jack/Documents/University/CodeAnalyser/testData/systemTesting/TestSyntaxError.java,  
Error: missing ')' at ','

### 6.3.4 Testing with input that is unsupported

```
jack@jack-Inspiron-1525:~$ java -jar Documents/University/CodeAnalyser/dist/CodeAnalyser.jar analyser --source Documents/University/CodeAnalyser/testData/systemTesting/Test.php --output good/
Found 1 Files To Analyse in: /home/jack/Documents/University/CodeAnalyser/testData/systemTesting/Test.php
Started Analysing File: /home/jack/Documents/University/CodeAnalyser/testData/systemTesting/Test.php
Unsupported Language
Generating Output
No Results Defined From Running The Metrics.
jack@jack-Inspiron-1525:~$
```

## Code Quality Analyser Results.

**Overall Result:**  
Amount of Files Analysed: 1  
Was Overall Successful: **FAILED**  
Successful/Failed: 0/1

**File Statistics:**  
PHP 100.0%

---

**Metric Results**  
No Results were defined.

---

**Unsupported Files**  
/home/jack/Documents/University/CodeAnalyser/testData/systemTesting/Test.php

**Syntax Errors**  
No Files Contained Syntax Errors.

## 7 Evaluation

This section is intended to reflect upon the development process, and the achievements of this project.

### 7.1 MoSCoW Requirements

An obvious way to determine the success of the project is to see if it met the original requirements as defined in the requirements specification.

All of the 'must have' requirements were all completed successfully, but these were essentially core requirements that needed to be completed before any of the other functionality could be completed.

The 'should' criteria have also fully been completed. All of the metrics do return results in which we can use to determine the quality of the source code being analysed, but obviously some of the metrics could not determine where the score came from, like for example the `LinesOfCode` metric determines the amount of lines of code a source code has, so the only indicator on improvement could be how many lines the source language was over the pre-defined threshold.

As stated in the requirement specification above, because the source language used in the application was Java, it made it extremely easy to run the application on different platforms.

The 'could' criteria as defined in the requirements specification was not implemented as part of this application mainly because of the time restraints on the application and it would require to provide the metrics with more information in order to be able to automatically generate the code needed to replace portions of code that were deemed as being of 'poor' quality.

The 'won't' criteria was definitely not implemented as part of the application mainly due to the amount of work that would do into developing a GUI for the application that would quite frankly not add a lot of functionality. The amount of time and work that would be involved to implement a GUI and make it as robust as possible could not be justified as the application still would of outputted HTML webpages and still would have completed the analysis in exactly the same way. This would have been more of a cosmetic feature.

## 7.2 Product Evaluation

The product did not change much from the original design. This was mainly due down to the amount of research that was carried out before starting the design and implementation processes. The only changes that were made to the original design was because of the way that ANTLR generated the Parser/Lexer classes. Because they adopted the grammar name in the class names the design had to be changed slightly to allow for this.

It was also intended to give this application the ability to automatically use the ANTLR Parser generator in order to generate Parsers/Lexers for any new grammar files in the allocated grammar directory. This functionality was actually implemented as the `LanguageHelper` class in the core package. However, after completing this functionality it became apparant that it was extremely difficult to dynamicly generate new java classes and allow them to be used in the current run. This is because any new Java classes that were generated had to be compiled in to .class files and be part of the class-path. After attempting to complete this functionality it was becoming clear that too much time was being devoted to this part of the application so it was disbanded. However, the current implementation can be used to generate the `BaseListener` instance for any Parser/Lexer classes that has manually been added to the application by using the `ClassGeneration.generateBaseListener()` method.

The part of the application that went a lot better than expected was the implementation of class that generated the output files. During implementation there was a lot of changes to the overall design of the output, but because the output was generated into HTML classes we had

a lot of techniques that can be used to make the output easier to read and understand for the end-user as you can see in section C of the appendix.

Overall, the design and implementation of the product went extremely smoothly, but some additional research into ANTLR was needed when it came to actually implementing the metrics to learn how to effectively manipulate the portion of the parse tree that was returned when an event is triggered while walking the parse tree, but this obviously helped the development of other aspects of the application where manipulation of the parse tree was required.

### 7.3 Project Evaluation

Overall, the project was a lot more difficult than first expected. However the finished project was a lot more advanced than first anticipated when the designs were being made. The biggest problem about undertaking a project of this stature is managing your time effectively.

The first term of this project was concerned with research into not only the area of metrics and the criteria that made good code, but also research into the history of programming languages. This term was therefore relatively easy to manage in terms of time, but because of the amount of research that had to be undertaken for this project, it felt like we had to rush some areas of the research, for example research into the ANTLR parser generator, this in turn backfired when it came to implement the application in which we had to backtrack and complete research into this area before we could proceed with the development.

The second term was characterised by the design and development of the project's application. This term therefore was harder to manage in terms of time because even though we could make a good initial estimate into how long the implementation process was going to take, it was never written in black and white. When unforeseen errors occurred during the development process this put the project behind schedule and this meant that some functionality had to get omitted because of this. An example of this is when the implementation of the `LanguageHelper` class was undertaken, because of the amount of time that was spent in the initial stages attempting to get this functionality working, it didn't leave a lot of time at the end of the implementation stage to attempt to complete some of the 'could' or 'won't' features that were defined in the requirements specification.

One main factor that helped the progress of the implementation stage was the use of the git version control system. This not only allowed us to keep an on-line repository of the product which could be easily accessed at other locations, but it also meant that a log was kept of the entire implementation process as shown in section B of the appendix. Keeping a log was good for two reasons, which were:

- We now have an entire history of the product's implementation stage to attach to this report.
- When an error occurred in the implementation process we could easily see where a change was implemented and effectively 'roll-back' to a specific version of the application instead of attempting to work our way back manually which gave us a lot more control over the application and saved a lot of time debugging the source code.

Overall, the project had a few problems along the way, but we managed to complete the 'must' and 'should' requirements of the requirements specification and have a decent end product.

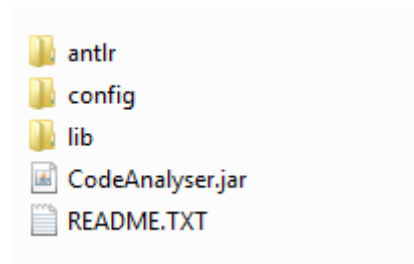


Figure 29: The directory structure of the working application

This product would definitely make a good project to take further and maybe use as a masters research project as there is some much potential for growth in the application because of its modular design.

## 8 Running the Application

The finished artifact is in the form of a runnable jar file, which is located in the dist directory in the source code CD attached to this report. The application depends on some outside directories, for example the template files and the language configuration file, and the jar file should be placed in the same directory the 'antlr', 'config' and 'lib' directories as also provided in the dist directory as shown in figure 29 on page 77.

This application needs to be run on the command line and has several input arguments that are required in order for it to function correctly. As shown in figure 30 on page 78 you can see what arguments are available by using the '-h' or '-help' argument.

The minimum required arguments to run the analyer are '-s/-source' and '-o/-output'. As shown in figure 30 on page 78 the '-s/-source' argument is the source location of the files to be analysed and the '-o/-output' argument is the location in which to save the generated output files, baring in mind that if the source location is quite large, the number of generated output files will be quite large also.

Typical use of the application can be seen in figure 31 on page 78.

```

u1051575@CW201-09 /k/Year3/CodeAnalyser/dist (master)
$ java -jar CodeAnalyser.jar analyser --help
usage: analyser
  -fl,--force-language <arg>    force language, no auto-language detection
                                  will take place.
  -h,--help                      Displays this help message
  -o,--output <arg>             output file location
  -s,--source                    source file location to analyse
  -ug,--update-grammars         updates the supported grammars.

u1051575@CW201-09 /k/Year3/CodeAnalyser/dist (master)
$

```

Figure 30: Using the -h or -help option when running the analyser displays the help text.

```

u1051575@CW201-09 /k/Year3/CodeAnalyser/dist (master)
$ java -jar CodeAnalyser.jar analyser --source ../testData/systemTesting/good -
-output ../output
Found 2 Files To Analyse in: k:\Year3\CodeAnalyser\dist\..\testData\systemTestin
g\good
Started Analysing File: k:\Year3\CodeAnalyser\dist\..\testData\systemTesting\goo
d\GoodQuality.hello
Started Analysing File: k:\Year3\CodeAnalyser\dist\..\testData\systemTesting\goo
d\GoodQuality.java
Generating Output

u1051575@CW201-09 /k/Year3/CodeAnalyser/dist (master)
$

```

Figure 31: Typical use of the application providing a source location and an output location.

## References

- Agile methodology*. (n.d.). (<http://agilemethodology.org/>)
- The ALGOL programming language*. (1996). (<http://groups.engin.umd.umich.edu/CIS/course.des/cis400/algol/algol.html>)
- Branching and merging*. (n.d.). (<http://git-scm.com/about/branching-and-merging>)
- CSE 505 lecture notes: ALGOL*. (1999). (<http://courses.cs.washington.edu/courses/cse505/99au/imperative/algol.html>)
- Eclipse standard 4.3.1*. (n.d.). (<https://www.eclipse.org/downloads/packages/eclipse-standard-431/keplersr1>)
- Ferguson, A. (n.d.). *A history of computer programming languages*. ([http://cs.brown.edu/~adf/programming\\_languages.html](http://cs.brown.edu/~adf/programming_languages.html))
- Laurie Williams, S. H., Dright Ho. (2003). *Metrics in eclipse tutorial*. (<http://realsearchgroup.org/SEMaterials/tutorials/metrics/>)
- Li, J. (n.d.). *COBOL*. (<http://www.csee.umbc.edu/courses/graduate/631/Fall2002/COBOL.pdf>)
- Littlefair, T. (n.d.). *CCCC - C and C++ code counter*. (<http://cccc.sourceforge.net/>)
- Metrics 1.3.6 - getting started*. (n.d.). (<http://metrics.sourceforge.net/>)
- Metrics definitions*. (n.d.). (<http://www.ndepend.com/metrics.aspx>)
- Naboulsi, Z. (2011). *Code metrics – cyclomatic complexity*. (<http://blogs.msdn.com/b/zainnab/archive/2011/05/17/code-metrics-cyclomatic-complexity.aspx>)
- Osborne, H. (2008). *Software "quality" control*. (<http://helios.hud.ac.uk/scomhro/Projects/SoftwareQuality/>)
- Parr, T. (2014). *Getting started with ANTLR v4*. (<https://theantlrguy.atlassian.net/wiki/display/ANTLR4/Getting+Started+with+ANTLR+v4>)
- Reitsma, R. (2011). *Software has a new quality standard: Iso 25010*. (<http://ryreitsma.blogspot.co.uk/2011/07/software-has-new-quality-model-iso.html>)
- Rüdiger Lincke, J. L., & Löwe, W. (n.d.). *Comparing software metrics tools*. (<http://www.arisa.se/files/LLL-08.pdf>)
- Staging area*. (n.d.). (<http://git-scm.com/about/staging-area>)
- Wallace, D. R. (1996). *Nist special publication 500-235: Structured testing: A testing methodology using the cyclomatic complexity metric*. (<http://www.mccabe.com/pdf/mccabe-nist235r.pdf>)
- The waterfall development methodology*. (n.d.). ([http://www.learnaccessvba.com/application\\_development/waterfall\\_method.htm](http://www.learnaccessvba.com/application_development/waterfall_method.htm))



## A Tables

### A.1 Borland Metric Table

Metric Name
Number of Attributes (NOA)
Number of Added Methods (NOAM)
Number of Classes (NOC)
Number of Child Classes (NOCC)
Number of Constructors (NOCON)
Number of Client Packages (NOCP)
Number of External Dependencies (NOED)
Number of Import Statements (NOIS)
Number of Local Variables (NOLV)
Number of Members (NOM)
Number of Operations (NOO)
Number of Overridden Methods (NOOM)
Number of Parameters (NOP)
Number of Public Attributes (NOPA)
Number of Operands (NOprnd)
Number of Operators (NOprtr)
Number of Remote Methods (NORM)
Number of Unique Operands (NUOprnd)
Number of Unique Operators (NUOprtr)
Package Cohesion (PC)
Polymorphism Factor (PF)
Package Interface Size (PIS)
Percentage of Package Members (PPkgM)
Percentage of Private Members (PPrivM)
Percentage of Protected Members (PProtM)
Percentage of Public Members (PPubM)
Package Size (PS)
Package Usage Ratio (PUR)
Response For Class (RFC)
Tight Class Cohesion (TCC)
True Comment Ratio (TCR)
Total Reuse of Ancestor percentage (TRAp)
Total Reuse in Descendants percentage (TRDp)
Total Reuse in Descendants unitary (TRDu)
Violations of Demeters Law (VOD)
Weighted Changing Methods (WCM)
Weighted Methods per Class 1 (WMPC1)
Weighted Methods per Class 2 (WMPC2)
Weight of a Class (WOC)
Attribute Complexity (AC)

Attribute Hiding Factor (AHF)
Access of Import Data (AID)
Attribute Inheritance Factor (AIF)
Average Inheritance Usage Ratio (AIUR)
Access of Local Data (ALD)
Access Of Foreign Data (AOFD)
Average Use of Interface (AUF)
Coupling Between Objects (CBO)
Cyclomatic Complexity (CC)
Coupling Factor (CF)
Changing Classes (ChC)
Class Interface Width (CIW)
Class Locality (CL)
Changing Methods (CM)
Clients of Class (COC)
Comment Ratio (CR)
Data Abstraction Coupling (DAC)
Dependency Dispersion (DD)
Depth Of Inheritance Hierarchy (DOIH)
Fan Out (FO)
Halstead Difficulty (HDiff)
Halstead Effort (HEff)
Halstead Program Length (HPLen)
Halstead Program Vocabulary (HPVoc)
Halstead Program Volume (HPVol)
JUnit Test Coverage (JUC)
Lack of Cohesion of Methods 1 (LCOM1)
Lack of Cohesion of Methods 2 (LCOM2)
Lack of Cohesion of Methods 3 (LCOM3)
Lines Of Code (LOC)
Method Hiding Factor (MHF)
Method Invocation Coupling (MIC)
Method Inheritance Factor (MIF)
Maximum Number Of Branches (MNOB)
Maximum Number Of Levels (MNOL)
Maximum Number Of Parameters (MNOP)
Message Passing Coupling (MPC)
Maximum Size Of Operation (MSOO)
Number of Accessor Methods (NAM)
Number of Client Classes (NCC)
Number of Import Classes (NIC)

## B Version Control Commit Log

e07358e Fixed an error when a block was found but it was not inside a method/constructor in DON, an example of this is an interface.  
eda52e9 Added initial design documents to repository as a backup.  
3d5036c Fixed another bug in getSystemPath  
426364a Added better handling to error in getSystemPath  
e599b4c Removed redundant comparator in CC/DON metrics just made the Entry classes implement Comparable<Entry>  
71d394e Changed some fields to final  
cb910e2 Updated nbproject classpath  
fdfd450 Added PDL metric, which determines how well the procedure declarations are defined, i.e if they have too many parameters or if the name is too long  
467ee69 removed redundant import from ParserInfo  
b29386c Updated a method name in Test2.java  
033bc2d adding generated output files  
086f26f Updated nbproject classpath  
98a71ea Added common-lang library  
a1088b4 Added new metric, that measures the level of nesting depth that occurs in each method-/constructor of the class being analysed.  
b718a0c Updated styling in the result of CR metric  
8076a3a Updated CC metric to output the amount of occurrences of each keyword found in the most complex method found.  
8519836 Added event method to notify metrics of the EXIT\_BLOCK context  
3e0d03c Added private constructor to Result so we force the use of Result.newInstance()  
b405a3a Added some new acceptable HTML types to the Whitelist  
8649eac removed some styling in the output template  
7756d6d changed hello grammar rule innerBlock to block  
062cc3b Added new generated output files  
fd4e905 Updated commenting in ApplicationException  
13653bf Refined the code in CC/WMC metrics and also looked out for catch clause events  
fffc070 Changed variable name in metric interface  
4885efb Updated Java BaseListener to event metrics for catch clauses  
bd78ddf Changed the Main2 tester class to run the analyser  
6d5c563 Added a try/catch to test class  
d5b5e97 Updated the grammar to the hello language to have try/catch statements  
5dfad01 Updated autogenerated output file.  
841865a Updated the accuracy of the natural language process when detecting the validity of variable names in a class  
fe5b9a4 Updated LanguageDetect to use the new Detection API  
16f52a0 Renamed Variable to Detection as it no longer encapsulates a single variable it is a detection based on all the variables in the class  
0a4d141 Renamed Variable to Detection as it no longer encapsulates a single variable it is a detection based on all the variables in the class  
b53a940 fixed error in splitCamelCase in which if no replacements were made it wouldnt cut the last character of the end of the string  
749a250 Added helper method to determine if the current user has an internet connection  
49dfa69 Added systemTesting '.hello' files  
ab0d765 Updated the classpath

9789b6c Added new libraries that the languagedetect classes depended upon to call a webservice  
 44c19c3 Added new LanguageDetect classes which utilise a web service  
 26a58cc Added new test source files.  
 fe13445 Updated LanguageDetect in VMC to use an updated file location  
 5ac985a Updated commentRatio to the filelocation of cloc  
 7937ebf Updated ParserInfo to use reflection on the Parser class to get the static final int Token fields  
 50bfcdc Updated File Locations in OutputGeneration to where the templates are located  
 b7c9bf5 Updated File Locations in Analyser/LanguageDetect  
 b099d82 Added ReflectionHelper which has the ability to get all class objects by a package name.  
 ef59e58 Added system file check in Main  
 5cc3e90 Updated file locations in languageHelper  
 942547f Added ApplicationException when a runtime exception occurs on startup  
 d92a807 Updated Application to use Reflection to determine supported languages and metrics instead of depending on a certain file location, also added getSystemPath() which returns the path the jar is currently in  
 5660751 Updating autogenerated files.  
 de655e1 Added new JUnit tests to test initialising metrics and calling metric methods while walking a parse tree.  
 d7dbc86 Updated the TesterMetric to print out information when being used while a parse tree is being walked.  
 3edc939 Removed redundant parameter to init in ListenerInterface, also updated template and any references in Analyser/FileAnalyser  
 5008cc3 Added several JUnit test cases.  
 9ff8ed5 Moved tester data to testData  
 23172d2 Moved tester data to testData  
 b66b4c6 Updated commentRatio to have a back up ratio determiner if CLOC cannot be used  
 c92662a Modified the ParserInfo object so it gives the supported parser and lexer instance to the metrics.  
 fl147b1 Moved the tester files to testData from test  
 44faba2 Made some of the methods private in AnalyserResult  
 d209860 Added getResult() to Analyser so we can get the calculated AnalyserResult object, this is useful in unit testing.  
 75f3254 Updated the constructor in the listener template  
 bf373a2 Updating nb project properties  
 7bc1672 Updating autogenerated output files  
 d97b1aa Updated the two supported grammars, parsers and lexers to look out for commenting.  
 f74e5e0 Added new 'getNoOfSyntaxErrorFiles()' which returns the amount of files with a syntax error, this is so the failed file count is correct if there is more than one syntax error on a file. Also added commenting to toString()  
 f7a5486 Fixed a bug where the count of failed files is invalid if there is more than one single syntax error on a file, also removed some redundant code.  
 0dcb17f Modified Main2 to show a parse tree for the Hello grammar using tester input  
 5ce6ddc updating autogenerated output files  
 de39528 Updated the Analyser to use the AnalyserResult object to track the result and this is then passed to the output generator.  
 c9bbd50 Created AnalyserResult which handles maintaining lists of results, syntax errors and unsupported files and then generating overall output from the analysis on all files and also determining the percentages of different file types that were attempted to be analysed.

bd84c4b Updated the Test.hello file to include some more complexity keywords.

be2d9a9 Updated OutputGenerator to use new AnalyserResult object and also pass file stats and overall result to the template.

508d01e Updated the output template to include overall results from evaluation and also file stats

527721a Updated the LanguageHelper to run a more efficient command when running antlr on linux

2739e38 Added autogenerated output files

fa06497 Added proper commenting for the Overridden methods in ANTLRErrorListener

a5c8eb3 Updated all commenting throughout application

e412507 CHanged Main2 to run the new Hello Parser/Lexer in the test file and display the parse tree.

d51b403 Added a cmd process to run javac on newly generated parser/lexer files so we can use then instantly

bd45e6d Addeed support to language 'hello' which even though it is a simple language, it has the same rules as the java grammar, so is automatically compatible.

77c1e95 Added linux bash script to run 'grun'

9d0fc1c Changed template to use onParserEvent instead of start

a153f96 Modified .hello tester file to follow grammar constraints.

132e381 Added check for when a statement is found outside a method, i.e in a static block in java

2493644 Updating autogenerated output files.

1e40052 Updated the OutputGenerator to pass SyntaxError messages to the template

0a1ec1d Added code to check for syntax errors that occur either in the parser or lexer and these files are not analysed.

7dd1f20 Added Syntax Errors to the OutputTemplate

78ddc4e Updated Autogenerated output file

bd7fea6 Added error check to see if CLOC was able to generate the output YAML file.

76861ec Updated AutoGenerated BaseListener

cae52db MOfified template to show metric error

27b9fce Changed the method in MetricInterface from 'start()' to 'onParserEvent()', seemed more applicable.

f49a07a Updated Tester Main class

19dfce9 Fixed bug in ClassGeneration where the case of the grammar name would sometimes make it so the package to save the generated class could not be found

0fd2206 Changed file name of Hello grammar file

3dd2e8d Changed reference to ParserInformation to ParserInfo in template

555a9e2 Added Tester Test.hello file.

9773c23 Updated Hello language parser to start with 'H' instead of 'h'

1d9306e Updated autogenerated output files

e6ad7cf Added secondary tester Main class 'Main2' which can be used to run tester code.

6b2ed11 Added getSourceLanguage() in FileAnalyser so we pass the correct source language to metrics

7a23155 method generator now adds 'enter' infront of rule names

3cfb4af Added the hello language to the auto-detect YAML document.

893817c Added hello.g4 grammar file

485e959 Added new language 'hello' a very simple grammar

5e46756 Removed EventType and its references in metrics and BaseListeners, used string events instead, easier to dynamically generate, also renamed ParseInformation to ParseInfo

9990180 Added commenting to Exceptions  
 28c7680 Updated ClassGeneration to use all of the rulenames for the BaseListener so that all events get called.  
 a58266f Updated ListenerInterface:init call to use the pass the parser through.  
 609df26 Added commenting to Analyser  
 7fa177e Added commenting to Main  
 4417459 Updated LanguageHelper to initialise the new parser during generation to pass the rule names for the BaseListener  
 689827b Added commenting to Application  
 c491cd3 Used the ResultComparator class instead of an anonymous function  
 ae0e2b1 Moved the comparator used to evaluate if the file 'overall' succeeded to ResultComparator  
 c44c62a Added commenting to ResultProperty  
 1a8a29d Added commenting to TemplateNotFoundException  
 c3ddd84 Added new event types to EventType  
 522d029 Pushed the Parser Instance to the listener so it can be passed to the metrics  
 a07d022 Added commenting to MetricException  
 732ed1a Added getRuleNames() to ParserInterface  
 1e134fe Added new events to base listener  
 c9f57f7 Added commenting to Metric Exceptions  
 541d5ec Updated MetricInterface to use ParserInformation class.  
 d41be5d Added ParserInformation class which is just an encapsulation of information that is passed from the parser to a metric when it is initialised.  
 e7e135e Added commenting to Result class  
 370af66 Updated CyclomaticComplexity metric to use the new ParserInformation class in init and actually completed this metric  
 205238b Updated CommentRatio metric to use the new ParserInformation class in init  
 3485700 Updated LinesOfCode metric to use the new ParserInformation class in init  
 a20085f Updated NumberOfMethods metric to use the new ParserInformation class in init  
 bfd333c Updated Tester metric to use the new ParserInformation class in init  
 0208383 Updated VariableNamingConventions to use the new ParserInformation class in init  
 1238a15 Completed WeightedMethodCount metric, determines the total number of complexity keywords used in a class.  
 bb7cdac Modified Test2.java file to have some complexity keywords  
 770803b Committing autogenerated output files.  
 8b40b47 Updated file permissions to external libraries  
 de58548 Added language profiles.  
 58ece2c Added Object Variable so we can correctly determine the number of acceptable variables even if one or more variables appear with the same name.  
 2d676f6 Changed LanguageDetect to return a single detected language instead of probabilities, this saves on computing time for irrelevant information  
 124e718 Added new error checking mechanism to WeightedMethodCount to 'getResults()'  
 4dfe419 Completed VariableNamingConventions metric, it checks all of the metrics and tries to detect the language as english, also added 'destroy()' to clear the DetectorFactory of its profiles for the next run.  
 5d1dbd5 Updated auto generated BaseListener to include new Template amendments  
 1e2c20d Added 'destroy()' to listenerInterface so we can handle clean up on listeners and metrics  
 24f560e Updated Analyser to call destroy on metrics so they can handle cleanup.  
 a92bb1e Updated listener template with new error checking in results.

ffedc5a Updated autogenerated output

379c4e9 Updated metrics to include 'throws InvalidResultException on getResults' this is so a invalid result will not stop the process, instead it will miss that result out.

3899ba2 Added 4 new metrics and added better error handling when errors occur in metrics

e06ad2d Added language detect library

20343ba updated auto-generated output files

a9b2fa9 Added lang profiles

81d7d0c Completed work to force a source language

a7ce4ea Updated FileAnalyser to be more organized

4a6a26d Added regexs to make a unique file name for the output for subfiles.

97fd9dc Added starter work to forcing a source language in Analyser

997639b Updated autogenerated output files.

ceb720b Renamed Tester File to test if the file is the same

20e118d Added autoupdated netbeans properties

cfea580 Added LanguageDetect which is used to detect a files source code based on its file extension.

2b2058f Added Language, which is just an encapsulation of a single detected Language

7c08dda Updated FileAnalyser to use LanguageDetect to determine the source language

7b16969 Made AnalyserException public so we can use it in Main

eca1278 Added System.out.println messages to Analyser to print progress

6153c5c Updated main so we can pass arguments from the command line.

b510087 Added languages file

7c06aa3 Added open-source YML parser to parse the language file

2215cdb Added Apache Commons CLI library so we can easily add command line arguments

7410c33 Added autogenerated output files

fe391ee Added commenting to ResultProperty class

724fdf8 Moved OverallResult to separate class and completed sub page generation

8c6ff05 Changed the result output from Number of Methods

37812bb Completed OutputGenerator to correctly generate HTML files from Result objects

d33dfb7 Updated FileAnalyser to get the basename for a file

bb65027 Updated Analyser to use OverallResult instead of ArrayList<Result>

1c3cf89 updating autogenerated properties

396644e updated autogenerated output files.

b31915b Updated Output template for sub pages

dc7bfc8 Added Jsoup library for HTML parsing

089cdd3 started work on output html generation and completed some new templates, also language generation also works on linux.

3b08646 Updated Main to test directory usage in Analyser

b35c762 Added tokens to init in ListenerInterface

7ae67f1 Updated auto-generated BaseListener

61498e7 Added tokens to init in MetricsInterface

2b0281d Added ENTER\_CLASS\_DECLARATION to correspond to the new event listener generated in BaseListener classes

04855a9 Removed supplying source language and filename in an event, as these are not passed when the metrics are first initialised.

64bc227 Stopped printing token Array in NumberOfMetrics

a43426c Stopped TesterMetric printing context.

6a4371c Added more tester data classes

d97d1c6 Added getTokenNames() to parserInterface to get the token names from the parser in-

stance

5375d93 Updated ClassGeneration to have event for ConstructorDeclarations and also handles passing the tokens in the current language to all the metrics  
d300b75 Added AnalyserException, which is thrown when Analyser errors occur.  
8755d67 Updated Analyser to be able to handle a directory structure of source code files, rather than just a single file.  
81bba34 Updated FileAnalyser to call init in Listener and Parser generation  
e8d1859 Updated the Listener template to include init and removed redunant EventStateBuilder method calls  
1780292 Added Number of methods metric, which counts the number of methods used in a class.  
46643cd Updated commenting in MetricInterface  
5cefeaa Updated auto-generated BaseListener  
56af76f Updated the tester metric to use the new MetricInterface method init and to use the Result Object  
3c25bd9 Added Result object which is an encapsulation of a Result from a metric for a single file  
c54308b Added init to MetricInterface so that initial values can be pushed to metrics, also used Result Object  
e20e20e Updated auto-generated BaseListener  
8881d43 Added init to ListenerInterface to push initial information to a listener when it is instantiated  
a96f06c Added file name to EventState  
9672463 Added file variable declaration to ClassGeneration  
3a969f0 Updated Analyser/FileAnalyser to use Result object.  
f7d5f6d Updated template file to include init implementation and use result object in getResults()  
db46130 Updated commenting and removed imports in LanguageHelper  
cc3b497 Added commenting to Application.ApplicationListener  
b6b102f Removed unwanted imports from FileAnalyser  
830a7e1 Added commenting to FileAnalyser, also optimised Exception code.  
5ae1b1f Added tester java file to analyse.  
8677c79 Updated Main to test Analyser  
63046af Added Interfaces to have generic parsers and listeners so we can call methods for any parser in any language.  
a6c99a6 Updated EventState to use EventType  
885963f Added EventType which is an type of event that occured while walking the parse tree.  
cbb3845 Added Analyser which is sed to analyse a file or directory  
c947efc Added FileAnalyser which determines if the file is supported and can return the parser/lexer to use on a certain file.  
9937816 Added a tester result String to TesterMetric  
3d8e228 Updated Auto-Generated Files  
55235a1 Updated Generation files to add EventType  
538d14d Updated nbproperties  
34c576a Modified .gitignore  
a34aa70 Updated Parser File on language generation.  
da87b6f Updated BaseListener template  
d9d7453 Added Apache Commons IO library  
f03164c Merge branch 'master' into development  
3107301 Added newly generated Java Parser Classes

09e8705 Updated Main to test Application.getMetricsList();  
0e1a5ff LanguageHelper now compiles correctly and stores in projects classes dir  
0938c0f Application now gets the fully-qualified name of metric classes  
fcb0853 Restored Original nbproperties  
64c0189 Restored Original build.xml  
ae23b3d changed the file structure of the application  
53d1255 Moved AntlrException as it applies outside its original scope, added commenting to Property classes and completed the Application class.  
8f4972c Modified README  
20eff8f Updated Main to use LanguageHelper.  
1642f16 Added BaseListener generation to language generation process  
0a41c11 Changed MetricAbstract to MetricInterface  
bfcf716 Added Java parser/lexer files.  
52204d2 Completed BaseListener generation method  
7bf69f5 Added Class Generation Property Classes  
5ccca35 Added BaseListener template  
7db64d4 Completed Java Parser Generation and started Listener generation  
715df21 Completed Grammar generation.  
f7c9c92 Added Java Grammar, and Added LanguageHelper file to auto generate parser/lexer files for new grammars.  
04c7d5d Update README.md  
a57d69e Initial commit



## C Sample output

### C.1 Overall Output

#### Code Quality Analyser Results.

**Overall Result:**

Amount of Files Analysed: 68

Was Overall Successful: **SUCCESS**

Successful/Failed: 54/14

**File Statistics:**

Unknown 6.0%

Java 94.0%

---

**Metric Results**

**Filename:** e:\virtual host\c\CodeAnalyser\src\org\codeanalyser\core\analyser\Analyser.java

**Source Language:** java

**Overall Result:** **Passed**

[Metric Results](#)

**Filename:** e:\virtual host\c\CodeAnalyser\src\org\codeanalyser\core\analyser\AnalyserException.java

**Source Language:** java

**Overall Result:** **Passed**

[Metric Results](#)

**Filename:** e:\virtual host\c\CodeAnalyser\src\org\codeanalyser\core\analyser\AnalyserResult.java

**Source Language:** java

**Overall Result:** **Passed**

[Metric Results](#)

**Filename:** e:\virtual host\c\CodeAnalyser\src\org\codeanalyser\core\analyser\FileAnalyser.java

**Source Language:** java

**Overall Result:** **Passed**

[Metric Results](#)

**Filename:** e:\virtual host\c\CodeAnalyser\src\org\codeanalyser\core\analyser\Language.java

**Source Language:** java

**Overall Result:** **Passed**

[Metric Results](#)

**Filename:** e:\virtual host\c\CodeAnalyser\src\org\codeanalyser\core\analyser\LanguageDetect.java

**Source Language:** java

**Overall Result:** **Passed**

[Metric Results](#)

## C.2 Output on Single File

### Code Quality Analyser Results - FileAnalyser.java

#### Metric Results

<b>Metric Name:</b> CyclomaticComplexity	<b>Was Successful:</b> <b>Success</b>
<b>Comments:</b> CyclomaticComplexity Results: Method With Most Complexity: getClassNames Amount of Complexity Keywords Found: 3 Complexity Threshold: 10 <b>Keyword Occurances:</b> if: 3	
<b>Metric Name:</b> CommentRatio	<b>Was Successful:</b> <b>Success</b>
<b>Comments:</b> CommentRatio Results: Code Percentage: 70.0% Comment Percentage: 29.0% Blank Lines Percentage: 2.0% <b>Thresholds:</b> Min: 20% Max: 40%	
<b>Metric Name:</b> DepthOfNesting	<b>Was Successful:</b> <b>Success</b>
<b>Comments:</b> DepthOfNesting Results: Method With Most Nesting: getClassNames Biggest Nesting That Occured: 3 Nesting Threshold: 4 <b>Source Code Text:</b> if(supported==null){thrownewUnsupportedLanguageException("Unknown Language");}	
<b>Metric Name:</b> LinesOfCode	<b>Was Successful:</b> <b>Failure</b>
<b>Comments:</b> LinesOfCode Result: Lines Of Code: 234 Threshold: 100	
<b>Metric Name:</b> NumberOfMethods	<b>Was Successful:</b> <b>Success</b>
<b>Comments:</b> Number of Methods In Class: 8 Method Threshold: 10	
<b>Metric Name:</b> ProcedureDeclarationLength	<b>Was Successful:</b> <b>Success</b>

## C.3 Unsupported Files Output

---

### Unsupported Files

```
e:\virtual host\c\CodeAnalyser\src\org\codeanalyser\language\hello\Hello.tokens  
e:\virtual host\c\CodeAnalyser\src\org\codeanalyser\language\hello\HelloLexer.tokens  
e:\virtual host\c\CodeAnalyser\src\org\codeanalyser\language\java\Java.tokens  
e:\virtual host\c\CodeAnalyser\src\org\codeanalyser\language\java\JavaLexer.tokens
```

## C.4 Reporting Syntax Error Output

### Syntax Errors

File: e:\virtual host\c\CodeAnalyser\testData\TestSyntaxError.java, Error: missing ')' at ';'

## C.5 Overall Output - Multiple Languages

### Overall Result:

Amount of Files Analysed: 7

Was Overall Successful: **SUCCESS**

Successful/Failed: 6/1

### File Statistics:

Hello 43.0%

Java 57.0%

---

### Metric Results