

REPURPOSING CODE METRICS FOR USE WITHIN MODERN DAY PROGRAMMING

LUKE RICKARD

A dissertation submitted in partial fulfilment of the requirements of Dublin Institute of Technology for
the degree of M.Sc. in Computing (Advanced Software Development)

July 2016

I certify that this dissertation which I now submit for examination for the award of MSc in Computing (Advance Software Development), is entirely my own work and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged within the text of my work.

This dissertation was prepared according to the regulations for postgraduate study of the Dublin Institute of Technology and has not been submitted in whole or part for an award in any other Institute or University.

The work reported on in this dissertation conforms to the principles and requirements of the Institute's guidelines for ethics in research.

Signed: _____

Date: 4 July 2016

ABSTRACT

Code metrics have existed since the early day of programming. Prior to the development of modern day compilers, metrics were used to identify defects early in the development process. Over time the theories were reworked to keep abreast of the changing programming landscape the most notable of which was moving from procedural programming to the introduction of the widely used object-oriented paradigm. Manual assessment of code has always been an area of contention among programmers and the introduction of ‘best practices’ in automated fashion shifted code metrics into yet another area than its original intention.

This dissertation aims to re-examine code metrics by identifying possible relationships that may exist between various metrics, analysing code to identify key characteristics that impact code metrics and then tie that back into their possible impact on modern day ‘best practices’ including topics like ‘code readability’. Results show that certain code metrics, combined with other coding factors can be combined to highlight code considered to be of a high ‘readability’ quality.

Key Words: *code metrics, software testing, unit testing, code coverage, cyclomatic complexity*

ACKNOWLEDGEMENTS

First and foremost I owe a massive debt of gratitude to my supervisor Damian Gordon. His enthusiasm, support and guidance helped me immensely over the course of this dissertation.

I would also like to add a special thanks to my sister Kelley for all her proof reading over the past few weeks.

Special thanks also to my parents for their constant support and encouragement over the years.

Table of Contents

1. Introduction	1
1.1 Project Background.....	1
1.2 Project Description	2
1.3 Project Aims and Objectives.....	4
1.4 Project Evaluation	4
1.5 Thesis Roadmap	5
2. Exploring Metrics.....	7
2.1 Introduction.....	7
2.2 Metrics within Software Development.....	7
2.3 Definitions.....	8
2.3.1 Object-Oriented Programming.....	9
2.3.2 Representing Real-World Items	9
2.3.3 Instances of a Class.....	9
2.3.4 Public versus Private Methods.....	10
2.3.5 Unit Tests.....	11
2.3.6 Code Coverage	11
2.4 Code Metrics	11
2.4.1 McCabe's Cyclomatic Complexity.....	12
2.4.2 Halstead E's	14
2.4.3 Building Upon McCabe's Cyclomatic Complexity	14
2.4.4 Critiquing Halstead and McCabe	15
2.5 Metrics for Object-Oriented Code.....	19
2.5.1 Chidamber and Kemerer Metric Suite	19
2.5.2 Building Upon Chidamber Object-Oriented Metrics.....	24
2.6 Applying Object-Oriented Best Practices to Code Metrics	30
2.7 The Law of Demeter	32
2.8 Conclusions	35
3. Applied Metrics	37
3.1 Introduction.....	37
3.2 Alternative Approaches to Code Metrics	37
3.2.1 Service Oriented Design.....	37
3.2.2 Metrics using Program Slicing.....	39
3.2.3 Tools for Metrics	43
3.2.4 Code Comments	45

3.2.5 Validation of Code Metrics	45
3.3 Code Metrics & Commercial Applications	46
3.3.1 Tackling Project Costs	46
3.3.2 Evaluation of Code Metrics within Hewlett-Packard	48
3.3.3 Alternative Takes on Applying Code Metrics	49
3.3.4 Classification of Metrics based on Defect Categories	51
3.3.5 Aggregation of Code Metrics	55
3.3.6 Evaluation of Metrics through Java Developers	57
3.3.7 Using Code Metrics to Automate Reviews	57
3.3.7 Applying Cyclomatic Complexity to Y2K	58
3.3.8 Standardisation of Metrics	58
3.4 Clean Code	59
3.4.1 Meaningful Names	60
3.4.2 Functions	61
3.4.3 The Stepdown Rule	61
3.4.4 Commenting Code	62
3.5 Key Findings	62
3.6 Conclusion	63
4. Data Exploration	65
4.1 Introduction	65
4.2 Roslyn Overview	65
4.3. Metrics Tools	67
4.3.1 NDepend	68
4.3.2 Visual Studio Code Analysis	68
4.3.3 ReSharper	69
4.3.4 Tableau	69
4.4 Roslyn Metrics	69
4.5 High Scoring Metrics	71
4.5.1 CSharpCodeAnalysis Project	72
4.5.2 Cyclomatic Complexity and Class Coupling	73
4.5.3 Depth of Inheritance and Class Coupling	74
4.5.4 Lines of Code and Class Coupling	74
4.5.5 Cyclomatic Complexity and Depth of Inheritance	75
4.5.6 Cyclomatic Complexity and Lines of Code	76
4.5.7 Depth of Inheritance and Lines of Code	77
4.5.8 Examination of code	78

4.6 Low Scoring Metrics	79
4.6.1 MicrosoftCodeAnalysisCSharpScripting Project	80
4.6.2 Cyclomatic Complexity and Class Coupling	80
4.6.3 Depth of Inheritance and Class Coupling.....	81
4.6.4 Lines of Code and Class Coupling.....	82
4.6.5 Cyclomatic Complexity and Depth of Inheritance	82
4.6.6 Cyclomatic Complexity and Lines of Code	83
4.6.7 Depth of Inheritance and Lines of Code	84
4.6.8 Examination of code	84
4.7 Average Scoring Project.....	86
4.7.1 RoslynTestPdbUtilities Project	86
4.7.2 Cyclomatic Complexity and Class Coupling	87
4.7.3 Depth of Inheritance and Class Coupling.....	87
4.7.4 Lines of Code and Class Coupling.....	88
4.7.5 Cyclomatic Complexity and Depth of Inheritance	89
4.7.6 Cyclomatic Complexity and Lines of Code	90
4.7.7 Depth of Inheritance and Lines of Code	90
4.7.8 Examination of code	91
4.8 Key Findings	91
4.8.1 Cyclomatic Complexity and Class Coupling	93
4.8.2 Depth of Inheritance and Class Coupling.....	93
4.8.3 Lines of Code and Class Coupling.....	93
4.8.4 Cyclomatic Complexity and Depth of Inheritance	93
4.8.5 Cyclomatic Complexity and Lines of Code	93
4.8.6 Depth of Inheritance and Lines of Code	94
4.8.7 Noteworthy Points	94
4.9 Conclusions	94
5. Impact of Code Readability on Metrics.....	96
5.1 Introduction.....	96
5.2 Developing the Extraction Software.....	96
5.2.1 MethodsPerClass	99
5.3 Data Preparation.....	104
5.3.1 FixingData	106
5.3.2 Formatting Type Name.....	109
5.3.3 Filtering of Partial Classes.....	110
5.3.4 Merging the Data	111

5.4 Exploring the Merged Data Set	111
5.4.1 Merged Data: Class Coupling and Number of Public Methods	111
5.4.2 Merged Data: Class Coupling and Number of Private Methods	112
5.4.3 Merged Data: Cyclomatic Complexity and Number of Public Methods	113
5.4.4 Merged Data: Cyclomatic Complexity and Number of Private Methods	114
5.4.5 Merged Data: Depth of Inheritance and Number of Public Methods	114
5.4.6 Merged Data: Depth of Inheritance and Number of Private Methods	115
5.4.7 Merged Data: Lines of Code and Number of Public Methods	116
5.4.8 Merged Data: Lines of Code and Number of Private Methods	116
5.5 Key Findings	117
5.6 Conclusions	118
6. Evaluation	119
6.1 Introduction	119
6.2 Tools	120
6.2.1 NUnit	120
6.2.2 Code Coverage	120
6.3 Unit Testing Evaluation	121
6.4 Evaluating Merged Data	126
6.4.1 Binder Class	128
6.4.2 CodeGenerator Class	129
6.4.3 CSharpCompilation Class	130
6.5 Summary of Key Quantitative Findings	131
6.6 Qualitative Evaluation of Code Readability	132
6.6.1 Interviewee Profiles	132
6.6.2 Software Terminology	134
6.6.3 Code Readability	135
6.7 Summary of Key Qualitative Findings	137
6.8 Conclusions	138
7. Conclusions and Future Work	140
7.1 Introduction	140
7.2 Conclusions	140
7.2.1 Existing Literature	140
7.2.2 Data Exploration	141
7.2.3 Impact of Code Readability on Metrics	143
7.2.4 Evaluation	145

7.3 Future Work	146
7.3.1 Introducing New Metrics	146
7.3.2 Alternative Open Source Solutions.....	147
7.3.3 Programming Paradigms.....	147
7.3.4 Community Evaluation.....	147
7.3.5 Development Methodology	148
7.3.6 Design Patterns.....	148
7.3.7 Extracting Data Using Platform Libraries	148
7.3.8 Open Source Testing.....	149

Table of Figures

Figure 2.1: Modelling a car in the object-oriented programming paradigm	9
Figure 2.2: Object-Oriented Code Modelling of a Car and Code Creating a new instance of Car called toyota	10
Figure 2.3: Control Graph	13
Figure 2.4: Cyclomatic Complexity of Common Control Structures	14
Figure 2.5: Myer highlighting the need to take Else branches into account	17
Figure 2.6: Mapping of Metrics to OOD Elements	23
Figure 2.7: A Correlation Matrix of Normalized Measures	30
Figure 3.1: Categories of Cohesion	38
Figure 3.2: C Function that computes the sum, average and product numbers from 1 to n	41
Figure 3.3: C function in Figure 3.2 in a program dependence graph	41
Figure 3.4: Data slices for C function as defined in Figure 3.2	42
Figure 3.5: CQMM activities	51
Figure 3.6: The process of extracting static code, churn and network metrics	53
Figure 3.7: Churn Metrics	53
Figure 3.8: Code Metrics	54
Figure 3.9: Network Metrics	55
Figure 3.10: The Squal Model	56
Figure 3.11: Definition of integer d	60
Figure 3.12: Suggested names for integer d	60
Figure 3.13: Example of code using poor naming convention	61
Figure 3.14: Example of code with refactored variable names	61
Figure 4.1: Visual Studio IDE highlights Bus in red as it has detected an error	66
Figure 4.2: Roslyn open source solution on github.com	67
Figure 4.3: The process of generating metric data from Roslyn	69
Figure 4.4: Identifying Projects with High Scoring Metrics	72
Figure 4.5: Identifying Types with High Scoring Metrics	73
Figure 4.6: Cyclomatic Complexity and Class Coupling	73
Figure 4.7: Depth of Inheritance and Class Coupling	74
Figure 4.8: Lines of Code and Class Coupling	75
Figure 4.9: Cyclomatic Complexity and Depth of Inheritance	76
Figure 4.10: Cyclomatic Complexity and Lines of Code	76
Figure 4.11: Depth of Inheritance and Lines of Code	77

Figure 4.12: CodeGenerator Class	78
Figure 4.13: GreenNodeExtensions Class.....	79
Figure 4.14: Identifying Projects with Low Scoring Metrics.....	80
Figure 4.15: Cyclomatic Complexity and Class Coupling.....	81
Figure 4.16: Depth of Inheritance and Class Coupling	81
Figure 4.17: Lines of Code and Class Coupling.....	82
Figure 4.18: Cyclomatic Complexity and Depth of Inheritance	83
Figure 4.19: Cyclomatic Complexity and Lines of Code	83
Figure 4.20: Depth of Inheritance and Lines of Code	84
Figure 4.21: Figure 4.21: CSharpPrimitiveFormatter Class	85
Figure 4.22: CSharpTypeNameFormatter Class	85
Figure 4.23: Cyclomatic Complexity and Class Coupling.....	87
Figure 4.24: Depth of Inheritance and Class Coupling	88
Figure 4.25: Lines of Code and Class Coupling.....	88
Figure 4.26: Cyclomatic Complexity and Depth of Inheritance	89
Figure 4.27: Cyclomatic Complexity and Lines of Code	90
Figure 4.28: Depth of Inheritance and Lines of Code	90
Figure 4.29: AsyncStepInfo Class	91
Figure 4.30: High Scoring Metrics	92
Figure 4.31: Low Scoring Metrics	92
Figure 4.32: Average Scoring Metrics	92
Figure 4.33: Small Multiples showing Cyclomatic Complexity and Lines of Code	94
Figure 5.1: Overview of Roslyn Structure	97
Figure 5.2: MethodsPerClass Class Diagram	98
Figure 5.3: Definition of MethodsPerClass	99
Figure 5.4: Definition of DataSet class used within MethodsPerClass.....	100
Figure 5.5: Definition of SetUp method.....	100
Figure 5.6: Definition of the foreach statement used to iterate through all the Roslyn binaries	100
Figure 5.7: Definitions of three methods used within MethodsPerClass	101
Figure 5.8: Definition of ExtractTypes methods	102
Figure 5.9: Definition of FormatName.....	102
Figure 5.10: Definition of FilterForPartialTypes.....	103
Figure 5.11: Definition of AddTypesToWorkSheet.....	103
Figure 5.12: Save new workbook to local hard disk.....	103
Figure 5.13: Data Preparation using CRISP-DM	104

Figure 5.14: FixingData Class Diagram.....	105
Figure 5.15: Definition of FixingData class	106
Figure 5.16: Definition of SetUp method	106
Figure 5.17: Reading in data from workbook on local hard disk.....	107
Figure 5.18: Definition of GetDataFromWorksheet.....	107
Figure 5.19: Definition of FormatTypeName.....	107
Figure 5.20: Definition of FilterForPartialTypes.....	108
Figure 5.21: Definition of AddTypesToWorksheet	108
Figure 5.22: Definition of SaveWorkbook.....	108
Figure 5.23: Definition of FormatName	109
Figure 5.24: Definition of FormatTypeName.....	109
Figure 5.25: Definition of FilterForPartialTypes.....	110
Figure 5.26: Definition of FilterForPartialTypes.....	111
Figure 5.27: Class Coupling and Number of Public Methods.....	112
Figure 5.28: Class Coupling and Number of Private Methods.....	113
Figure 5.29: Cyclomatic Complexity and Number of Public Methods.....	113
Figure 5.30: Cyclomatic Complexity and Number of Private Methods.....	114
Figure 5.31: Depth of Inheritance and the Number of Public Methods	115
Figure 5.32: Depth of Inheritance and the Number of Private Methods	115
Figure 5.33: Lines of Code and Number of Public Methods	116
Figure 5.34: Lines of Code and Number of Private Methods	117
Figure 5.35: Overview of findings from scatter plots	117
Figure 6.1: Definition of GeneratePrimes Class.....	122
Figure 6.2: Definition of GeneratePrimes Class Unit Tests.....	122
Figure 6.3: Definition of Refactored GeneratePrimeNumbers Method.....	123
Figure 6.4: Definition of Private Methods used by GeneratePrimeNumbers	124
Figure 6.5: Definition of Unit Tests for both GeneratePrimes and PrimeGenerator	125
Figure 6.6: Code Coverage results for both GeneratePrimes and PrimeGenerator	125
Figure 6.7: Code Metrics results for both GeneratePrimes and PrimeGenerator	126
Figure 6.8: Number of Private Methods and Lines of Code (A)	127
Figure 6.9: Number of Private Methods and Cyclomatic Complexity (B).....	127
Figure 6.10: Binder Code Snippet One.....	128
Figure 6.11: Binder Code Snippet Two.....	128
Figure 6.12: Binder Code Snippet Three.....	129
Figure 6.13: CodeGenerator Code Snippet.....	130
Figure 6.14: CSharpCompilation Code Snippet.....	130

Figure 6.15: Interviewee profiles overview	132
Figure 6.16: Number of years interviewees spent working in Information Technology	133
Figure 6.17: Job titles of interviewees	133
Figure 6.18: Daily interaction with code of interviewees	134
Figure 6.19: Testing approaches of companies that interviewees work within	134
Figure 6.20: Common software related terms interviewees were familiar with.....	135
Figure 6.21: Code snippets presented to interviewees to be rate for readability	136
Figure 6.22: Code snippets presented to interviewees to be rate for readability	136
Figure 6.23: Experts view of complexity of BubbleSort.java	137
Figure 7.1: Small Multiples of Cyclomatic Complexity	142
Figure 7.2: Overview of correlations between code metrics and public and private methods.....	144
Figure 7.3: Illustration of how a heat map looks.....	148

1. Introduction

1.1 Project Background

Fagan (1976) was one of the early adopters of using static code analysis to identify errors early in the software development process. He argued that the cost involved in fixing errors increased the later the error was discovered during the development process. In 2002 van Emden and Moonen applied software inspection to detecting code smells within object-oriented languages. He describes code smells as “*patterns that are generally associated with bad design and bad programming practices*” for example, “*code duplication*” and “*long methods*”. This was in contrast to existing code inspection tools that looked at low-level aspects of code to identify problems with “*pointer arithmetic, memory (de)allocation, null references [...] etc.*”. Yüksel and Sozer (2013) highlighted an issue with code inspections impacting developer productivity by having too many uncategorised alerts including false positives. In his research he proposed a technique to classify these types of alerts with a goal to providing real value through code inspections.

Yüksel and Sozer (2013) highlighted the issue of static code analysis providing too many alerts to developers including false positives. This resulted in developers spending too much time investigating each alert to establish how significant it was. The research proposed an approach to classifying the alerts using machine-learning techniques. Steidl (2013) noted that a large part of source code is comments and although crucial to understanding the code they are not taken into account when evaluating the quality of the code. He argues that source code comments should be included in the metrics used when analysing code quality (Steidl *et al.*, 2013). Balachandran (2013) stated that while peer code reviews was a cost-effective way of conducting code reviews it consisted of a significant amount of human capital. He argued that static analysis could be used to reduce manual code reviews. The paper looked at using checks for code violations and “common defect patterns” in an effort to reduce the amount of manual intervention involved.

De Silva, Kodagoda, & Perera (2012) compared three complexity metrics including “*McCabe’s Cyclomatic Complexity, Halstead’s software science and Shao and*

Wang's cognitive functional size". The paper concluded that Shao and Wang's cognitive functional style was most beneficial in real world applications. Sarwar, Shahzad, & Ahmad (2013) looked at addressing the nesting problem with the Cyclomatic Complexity metric, a long established metric used to determine the complexity of code. Tosun, Caglayan, Miranskyy, Bener, & Ruffolo (2011) stated that researchers found that "*code, churn, and network metrics as significant indicators of defects*". However, it also stated that not all may be informative for all defect categories.

Casalnuovo, Devanbu, Oliveira, Filkov, & Ray (2015) used a large dataset provided by GitHub to explore "*connection between asserts and defective occurrence*". This demonstrates how open source projects can provide the large datasets on which to conduct tests in an effort to gain new insights into old issues.

Lei, Cheng, Bing, & Sato (2015) used code coverage as a "*concrete measurement of testability*" and demonstrated the necessity of testing less visible code areas.

Overall it can be concluded that establishing which metrics to apply to a static analysis to in an effort to identify issues early in the software development life cycle (SDLC) has many challenges. While selecting one or two metrics may not be enough, applying all known metrics, can result in too many alerts being generated, as highlighted by Yüksel and Sozer (2013).

1.2 Project Description

The majority of software metrics were developed in the 20th century when compilers simply compiled the code that was presented to them, but modern compilers optimise and re-structure code to achieve a more effective machine level code, which in turn changes the effectiveness of existing software metrics. This research seeks to bring software metrics into the 21st century by using them in a hitherto unanticipated way by attempting to determine their effectiveness as indicators of code quality and readability.

Although there has been a long history of defining code metrics that can be used to detect issues through the use of static code analysis, early in the development process, there is still much discussion as to which of these metrics are most valuable and there is much room for discussion as to application in modern day software development. There are many approaches that can be taken in applying code metrics, the most obvious being their intended use from when first developed but as previously mentioned much of this has been solved by modern day compilers. That said certain parts are still valid, such as determining the complexity of piece of code. As De Silva *et al.* (2012) found, asking programmers how complex a piece of code is can lead to wide range of viewpoints.

van Emden & Moonen (2002) brought a new dimension to the area by introducing the concept of detecting ‘code smells’ as part of static code analysis. This moved code metrics away from being simply a binary search for code that is either good or bad and looks to how the code is styled or if the code adopts best practice.

The main issues of focus for this research regarding code metrics are:

1. Researching the existing area of code metrics in depth in order to gain a complete picture and identify key theories that have been developed.
2. The introduction of coding best practices into the area of code metrics.
3. Evaluating if the code metric theories can be used in commercial development of software
4. Identifying relationships between the various code metrics

Points One and Three above will be researched using existing literature whereas Points Two and Four will be the subject of formulating a new data set and examining the data through the use of visuals in an effort to identify relationships and examine sample code to detect common patterns that make point to causation of the metric scores. It is in attempting to identify this causation that the link between code metrics and modern best practices in programming will be analysed.

In order to test the hypothesis sample code will be required. To ensure the data is of practical value a popular open source project will be selected and analysed. The provides a higher value to the data set as it will be an open source solution that is in

active use and therefore have all complexities and trade-offs that are commonly made when developing code in the real world and not just exhibit textbook style and practice. This will involve sourcing tools to extract the new metric data and then creating data visualisations in an effort to identify any possible relationships that may exist. In addition, code samples will be analysed at times in order to identify possible characteristics in the code that result in the various code metric results. It is these characteristics in the code that may also be able to help new insight into the impact common best practice has on code metrics.

1.3 Project Aims and Objectives

The key research question can be stated as follows: *Can metrics from the past century be repurposed to for use in modern day programming?*

The research question can be broken down into the following objectives:

- Conduct research in the more broad area of code metrics
- Identify key metrics that have provided value over the years
- Introduce new modern day best practices to the area
- Identify how these metrics can assist or impact modern day best practice

The key hypothesis formulation here is as follows:

- **H₀:** Metrics defined for the 20th century coding issues are repurposed for 21st century coding issues.
- **H₁:** Metrics used to identify coding issues in the past can be repurposed to identify good coding practices for modern day programming

1.4 Project Evaluation

The process of evaluation will use a Mixed Method Approach, first there will be a quantitative evaluation, which will involve taking any findings and evaluating using a combination of unit testing and code coverage metrics to assist in determining the

validity of the findings. In addition, it will also involve assessing the code where the finding was made in an effort to identify any possible causation. Secondly, there will be a qualitative assessment done in the form of interviews by people from the information technology industry. Ideally they would have a background in testing or software development. In addition to assessing any finds that may arise as a result of the research the interviews will also attempt to gain insight into the level of understanding that currently exists within the software industry in relation to code metrics.

1.5 Thesis Roadmap

Chapter Two will seek to lay a solid foundation by examining in depth some the most important areas of code metrics. This will provide the building blocks from which the subsequent chapters can build.

Following this Chapter Three will look to explore areas that have grown out of the key concepts from Chapter Two. There will also be special attention paid to the various attempts that were made to use code metrics with established commercial organisations.

Chapter Four will seek to explore new metric data in an attempt to identify any new insights. This involves identifying a popular open source project in order to analyse data that is used in practice within the industry and extract metrics from that data using commercially available tools. This exploration phase will rely on creating data visualisations in order to identify these new insights. In addition, it will also examine code snippets at random in an effort to identify any patterns or characteristics that cause the metrics to be.

Chapter Five will consist of taking one or more key findings from Chapter Four and exploring it in more depth. This may take the form of examining a relationship between two or more metrics or a characteristic found through the random sampling of code.

Taking any findings from Chapter Five, Chapter Six will attempt to evaluate these. There are various tools available that may assist in this process including unit testing of code and code coverage i.e. the percentage of code exercised by a given suite of tests. In addition, this section will also consist of a qualitative element of assessment.

Chapter Seven will present the conclusions of this research, and also suggest future directions that this project might take.

2. Exploring Metrics

2.1 Introduction

Metrics in software engineering are defined by different researchers in a variety of different ways, so much so that it is difficult to establish a consensus as to what exactly metrics are. The term is used in many aspects of the overall Software Development Lifecycle (SDLC) and can be described as having some sort of measurable component. This chapter will begin with a general discussion on software metrics that will assist with framing where code metrics, the main topic of this research, fits into the overall picture. It will then look to define some of the terminology that is contained throughout this paper so as to remove any possible ambiguity. From there the paper will dive deep into the area of code metrics.

Starting with McCabe's Cyclomatic Complexity, the paper will dig into the original thought processes of the area in order to establish a solid understanding of the key areas that code metrics have grown out of. Building on from McCabe and looking at the various critiques offered on his approach and others the chapter will then shift towards the object-oriented paradigm of software development with special attention to be paid to Chidamber & Kemerer's (1991) suite of software metrics. From there it will move to van Emden's (2002) work on bringing modern best practices of what are known as 'code smells' to the area of code metrics and finish by looking at arguably one of the most fundamental principles in object-oriented programming, known as *The Law of Demeter*.

2.2 Metrics within Software Development

Metrics in relation to software is a very broad as it can relate to many different aspects of software or the development of software. For example, the ratio between developers and testers could be a metric or the number of bugs found per 100,000 lines of code could also be a metric. There are many books and papers that discuss metrics in the form of project management that are often referred to as software metrics, for example, the budget for a software project i.e. what percentage has been spent on average each

day versus what percentage of the software is completed. For this reason it is extremely important to define what is being referred to as a metric within this research. It will focus on what are known as code metrics. These are metrics taken directly from code already written by developers. It is in effect software, either in the form of commercially available tools or as a result of in-house development that is used to analyse code as it is written. As will be shown throughout this chapter, it can be used to extract varying types of data, for example, identify areas of the code as having a high-level of complexity and therefore be considered as having a higher risk of being defective. In addition, it is important to realise that these checks are generic i.e. the same checks can be performed on all areas of the code without any specific software being developed for a particular feature or subcomponent.

The earlier decades of programming involved a large amount of procedural code whereas the 1990s saw an explosion of object-oriented languages with C++, Java and Microsoft's C# growing in popularity. Although this research does not focus on any one particular language, Microsoft's C# will provide the basis for data exploration and experimentation in later chapters.

Prior to looking at code metrics, some definitions will be provided to some of the key concepts and themes found throughout the paper.

2.3 Definitions

This section will introduce some definitions of the various concepts found throughout the paper.

2.3.1 Object-Oriented Programming

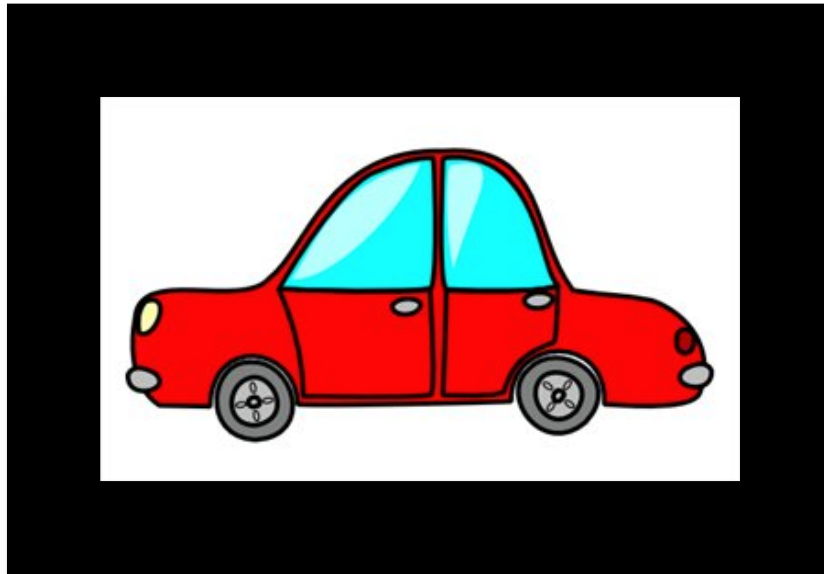


Figure 2.1: Modelling a car in the object-oriented programming paradigm

2.3.2 Representing Real-World Items

The most conceptual way to comprehend object-oriented programming is to consider Figure 2.1 above showing a car. Object-Oriented programming provides programmers with a means to take a real world object like the car above and represent it as within a programme. To do this, the car above would be a class. The class would literally be called `Car` (with an uppercase C). Within the `Car` class, there are mainly two things: properties and methods. The properties are like items of the car such as `FuelType`, `TireSize`, `MaxSpeed`, and `NumberOfGears` etc. whereas methods allow behaviours of the class to be performed. For example, the `Car` class could have methods called `DriveForward` or `ChangeGear` (Meyer, 1988).

2.3.3 Instances of a Class

Once the `Car` class has been defined, this allows another part of the code outside of this particular class create instances of the `Car` class. The `Car` class is no use programming-wise until an instance of the class is created. Most programming languages provide the *new* keyword to do this. In the case of the `Car` class, the new

instance called Toyota could be created. Now the programmer can use the properties to get information about the Toyota car that has been created while also using the methods `DriveForward` and `ChangeGear` in order to get the Toyota car to do useful things. When talking about methods, the word ‘invoke’ will be used. Figure 2.2 shows car as represented in object-oriented programming and the piece of code using the *new* keyword to create an instance of the car called *toyota*. The new instance has access to the methods including `DriveForward` (Jacobson, 1992).



```
class Car
{
    public string FuelType { get; set; }
    public int TireSize { get; set; }
    public int MaxSpeed { get; set; }
    public int NumberOfGears { get; set; }

    public void DriveForward()
    {
        //Code that makes car drive forward
    }

    public void ChangeGear()
    {
        //Code that changes the gear of the car
    }
}

class Program
{
    static void Main(string[] args)
    {
        Car toyota = new Car();
        toyota.DriveForward();
    }
}
```

Figure 2.2: Object-Oriented Code Modelling of a Car and Code Creating a new instance of Car called *toyota*

Behind the scenes programming languages send messages to methods that invoke them but the details of this are beyond the scope of this paper and are not required in understanding the basic concepts being presented here. The `Car` class above, as an example, is very simplified but provides the overall concept of how classes can be used within programming language to model items in the real world, like cars or boats or people etc.

2.3.4 Public versus Private Methods

The `Car` class above had two methods, `DriveForward` and `ChangeGear`, both of which allow other parts of a program to invoke them, once an instance of the `Car` class has been created. A class can have two types of these methods, public and private. The methods in the `Car` class were both public as code outside of the class could invoke them. Private methods on the other hand are not accessible outside of the class. They are restricted in access and are therefore only available to code within the

class. There are many reasons why a programmer would choose to make a method within a class private, but for now it is suffice to say that is something controlled by the programmer and can be used to set access levels to different pieces of class functionality (Meyer, 1988).

2.3.5 Unit Tests

A key aspect of developing software is testing the software. Software testing is an entire area in itself with many different practices from manually testing entire products to automated tests that run each time a new piece of code added to product. One of the many concepts that arise is that of unit testing. Unit testing allows the tester to write tests that exercise every avenue of a piece of code. The test would normally break down a large application and select a segment as small as a class and perform tests on every aspect of that class. A unit test should normally only test one thing. For example, a unit test can test that an if-statement, a piece of code that checks if a value or combination of values returns true or false, performs correctly for all inputs. One if-statement could have multiple tests depending how what needs to be taken into account to evaluate the if-statement (Osherove, 2015).

2.3.6 Code Coverage

As previously mentioned, unit tests evaluate all avenues of a piece of code to ensure the code is executing as expected. A key aspect of writing unit tests is determining when all of these avenues have been covered. Tools are available that allow a tester to calculate what is known as code coverage. Code coverage tools provide the percentage value of the code that has been covered by a given suite of tests (Osherove, 2015).

2.4 Code Metrics

From the earliest days of code development the issue of defects and defective code has been a challenge for software engineers and organisations dependent on large-scale software platforms. Beginning with Fagan (1976) arguing that the cost of fixing such defects increased the later they were discovered in the development process has led to a lot of theorising as to how best tackle the issue. One area of focus was to develop a

generic way to statically analyse in an effort to identify areas with a high risk of being defective. One of the earliest works in this area came from McCabe (1976) and has become the benchmark against which many other theories are either compared against or branches from.

2.4.1 McCabe's Cyclomatic Complexity

Thomas McCabe (1976) posed a question regarding the development of software: *"How to modularize a software system so the resulting modules are both testable and maintainable?"* McCabe argued that given the considerable portion of software development that is devoted to testing and maintenance of a system that a mathematical technique was required in order to quantify which modules of a system would be *"difficult to test or maintain"*. In an attempt to answer this question, McCabe (1976) defined a complexity measure, using graph theory that measures and controls the number of paths through a program. Providing examples in FORTRAN programs McCabe showed that complexity is independent of physical size and *"complexity depends only on the decision structure of a program"* (T. McCabe, 1976).

The first issue that arises with this approach is the problem that *"Any program with a backward branch potentially has an infinite number of paths."* Although using the total number of paths through a program is possible, it has been found to be impractical and therefore McCabe's Cyclomatic Complexity measure is defined in terms of basic paths – *"that when taken in combination will generate every possible path"* (T. McCabe, 1976).

In order to explain the Cyclomatic Complexity measurement, McCabe presents some mathematical preliminaries.

Definition 1: The cyclomatic number $V(G)$ of a graph G with n vertices, e edges, and p connected components is

$$V(G) = e - n + p$$

Theorem 1: In a strongly connected graph G , the cyclomatic number is equal to the maximum number of linearly independent circuits.

Using this theorem, McCabe then associated a given program with a directed graph that has unique entry and exit nodes. *“Each node in the graph corresponds to a block of code in the program where flow is sequential and the arcs correspond to branches taken in the program”* (T. McCabe, 1976).

The graph shown in Figure 2.3 is a control graph where it is assumed that each node can be reached by the entry node and that each node can reach the exit node, with “a” being the entry node and “f” being the exit node (T. McCabe, 1976).

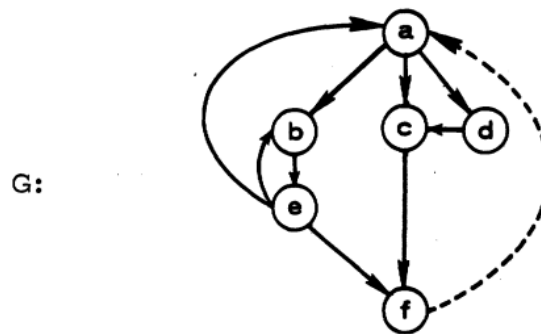


Figure 2.3: Control Graph

As exit node ‘f’ has been branched back to the entry node ‘a’ the graph is now strongly connected i.e. *“there is path joining any pair of distinct vertices”* and therefore fulfils theorem 1.

M McCabe (1976) defined the Cyclomatic Complexity of some common control structures:

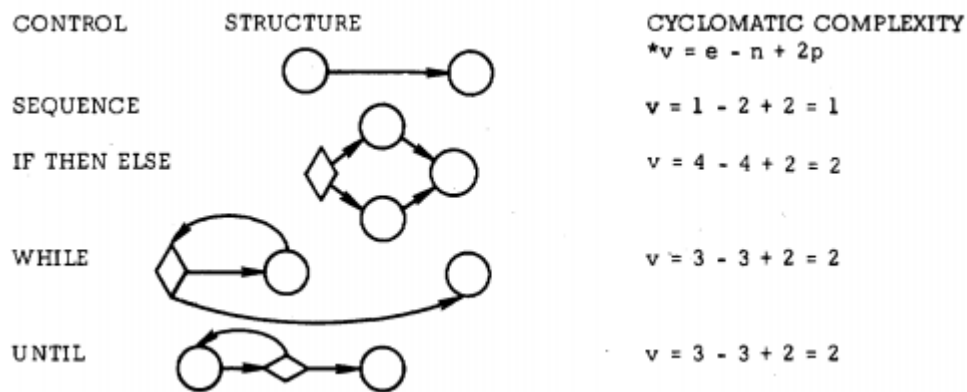


Figure 2.4: Cyclomatic Complexity of Common Control Structures

In addition, McCabe (1976) also identified some properties of the Cyclomatic Complexity

1. $V(G) \geq 1$
2. $V(G)$ is the maximum number of linearly independent paths in G ; it is the size of the basis set.
3. Inserting or deleting functional statement to G does not affect $V(G)$.
4. G has only one path if and only if $V(G) = 1$.
5. Inserting a new edge in G increases $V(G)$ by unity.
6. $V(G)$ depends only on the decision structure of G .

2.4.2 Halstead E's

According to Halstead (1972) the effort required to generate a program “*can be derived from simple counts of distinct operators and operands and the total frequencies of operators and operands*”. Given these, Halstead computes the number of mental comparisons required to generate a program (Curtis, Sheppard, Milliman, Borst, & Love, 1979).

2.4.3 Building Upon McCabe's Cyclomatic Complexity

Curtis *et al.* (1979) used three software complexity measures, namely McCabe's Cyclomatic Complexity, Halstead's E and “*the length as measured by the number of statements*” in an attempt to estimate the future maintenance cost of a software program. He argued that with the cost of maintaining the program to be three times greater than the cost of the initial development, such a measurement could prove invaluable to the software managers (Curtis *et al.*, 1979).

First, they look to Halstead's 1972 theory stating that, *"algorithms have measureable characteristics analogous to physical laws"*. Halstead defined the amount of effort required to generate a program as *"simple counts of distinct operators and operands and the total frequencies of operators and operands"*. Using these four quantities Halstead calculates *"the number of mental comparisons required to generate a program"* (Curtis *et al.*, 1979).

They then take McCabe's definition of complexity of counting the number of *"basic control path segments which, when combined, will generate every possible path through the program"*. While noting that no exact mathematical relationship exist between McCabe's and Halstead's metrics they do point out that as the number of control paths increases in a program there would be an anticipated increase in the number of operators and therefore a significant correlation between the two would not be surprising (Curtis *et al.*, 1979).

They go on to examine the differences between psychological complexity, the characteristics that make a program difficult to understand to human readers and computational complexity, correctness of the program that both Halstead and McCabe analysis are attempting to define. In an effort to understand to what extent both of these metrics *"assess the psychological complexity of understanding and modifying software"*, they found that *"assessing the psychological complexity of software appears to require more than a simple count of operators, operands and basic control paths"* (Curtis *et al.*, 1979).

2.4.4 Critiquing Halstead and McCabe

Shen, Conte and Dunsmore (1983) offered a critique of Halstead's theory of software science and looked at some of the metrics published in the intervening years since Halstead's first publication circa 1977. Noting that as software metrics are growing with ever increasing importance and simple measures such as lines of code are an inadequate measure of complexity, they stated that there was a multitude of factors that affect programmers productivity including the *"type of program being developed, size of the program, implementation language, interface complexity among modules in the*

program, experience of the programmers involved programming techniques employed” as well as the environment itself. Even taking all these factors into account they found it did not lead to a “useful” estimator of programming effort required. They instead argued in favour of a “*model of the programming process based upon a manageable number of major factors that affect programming*” as they believed this would lead to more useful metrics for software managers to work with.

Their analysis had some criticisms of the software science, such that some of the early code samples used to validate software science was quite small and there were many ambiguities around deciding which operators and operands should be included in the count citing the example of the GO TO label as being “*a unique operator for each unique label*”. Overall they did concede that software science E “*is at least as good an effort measure as most others being used*” (Shen, Conte, & Dunsmore, 1983).

McCabe’s Cyclomatic Complexity defines a “*mathematical technique that will provide a quantitative basis for modularisation and allow us to identify software modules that will be difficult to test or maintain*”. On rejecting ‘lines of code’ as there was no obvious relationship between length and module complexity, McCabe defined the measure of complexity by examining the number of control paths through a module (Shepperd, 1988).

McCabe used graph theory to overcome the issue of a having an infinite number of paths through code that contains a backward branch. By representing each executable statement as a node with the edges representing the control flow, any piece of procedural software could be depicted as a directed graph. Given this representation of the code and provided this directed graph is strongly connected i.e. every vertex is reachable from every other vertex, it can be used to determine the number of basic paths contained in the program, which, when combined together, “*can generate all possible paths through the graph or program*” (Shepperd, 1988).

Sheppard (1988) offered a critique of McCabe’s Cyclomatic Complexity as a software metric. He cites Myer’s criticism that the metric “*fails to distinguish between selections with and without ELSE branches*”. This is shown in Figure 2.5 below,

where Myer's metric takes the ELSE into account unlike the McCabe's Cyclomatic Complexity (Shepperd, 1988).

IF X = 0 THEN ... ELSE ...;	$v(G) = 2$ Myers = (2:2)
IF X = 0 AND Y > 1 THEN ... ELSE ...;	$v(G) = 3$ Myers = (2:3)
IF X = 0 THEN IF Y > 1 THEN ... ELSE ... ELSE . . . ;	$v(G) = 3$ Myers = (3:3)

Figure 2.5: Myer highlighting the need to take Else branches into account

Sheppard (1988) also notes the fact that McCabe's thinking in defining the complexity revolved around FORTRAN as opposed to more recent languages of the time, namely Ada. This in turn makes the mapping from code to graph more ambiguous. In addition, Sheppard also highlights the controversy around the metric being "*insensitive to complexity contributed from linear sequences of statements*".

The fact that $v = 1$ will remain true for "*a linear sequence of any length*" was another area of controversy and had other researchers offer alternatives to McCabe's proposal. These included Hansen's 2-tuple of Cyclomatic Complexity although Baker and Zweben also took issue with this approach (Shepperd, 1988).

Another criticism of McCabe's Cyclomatic Complexity was that it increased when applying what are considered good programming practices. As noted by Sheppard (1988) only two out of twenty six of Kernighan and Plauger's rules of good programming style resulted in a decrease in the complexity. All decisions carry the same weight for McCabe's Cyclomatic Complexity regardless of why nesting was applied in a particular fashion. Many researchers would argue that modularity of a program is better viewed through 'coupling' and 'cohesion', something that is not captured by McCabe's metric (Shepperd, 1988).

While noting that the difference between software engineering and other established branches of engineering was the lack of an accepted set of metrics with software

engineering, Gill and Kemerer (1991) argued that the absence of which would lead to software development remaining in a “*stagnant craft-type*” mode that made it difficult to pass knowledge to the next generation of engineers. By having well-established metrics, engineers could quantify projects and evaluate tools and processes more effectively.

They highlighted the maintenance of software systems as one of the key areas in need of a metric and evaluating the complexity of code that needs to be modified. He cites McCabe’s description of the primary purpose of the metric as to “*identify software modules that will be difficult to test or maintain*”. Their paper does not seek to evaluate whether McCabe’s Cyclomatic Complexity fully captures all the complexity of a system but rather answer Sheppard’s question regarding McCabe’s complexity measure; can McCabe’s Cyclomatic Complexity serve as a “*useful engineering approximation*” (Gill & Kemerer, 1991).

They argue that while the assumptions which exist linking code complexity to high maintenance cost have been criticized as relatively weak, studies have shown that a large amount of resources have gone into engineers attempting to understand code when making changes during maintenance. Their paper goes on to look at the relationship between McCabe’s Cyclomatic Complexity and the maintenance of software systems. Included in their findings, Gill and Kemerer (1991), found that metrics proposed by Myers and Hansen as well as McCabe’s Cyclomatic Complexity were all highly correlated. As consistent with Sheppard’s findings, the data suggested there was unlikely to be “*any practically significant different results using*” Myers or Hansen’s metrics over McCabe’s. In addition, it was also found that the length measure was also highly correlated to the complexity measure. Taking this information, Gill (1991), defined a complexity density metric defined “*as the ratio of Cyclomatic Complexity [...] metric to thousand lines of executable code*”. Although citing the use of a small sample, Gill (1991), did note the results of his experiments as “*sufficiently interesting*” to warrant further study and if they continued to hold that the use of the complexity density was a quantitative way to determine software maintainability.

Sarwar *et al.* (2012) argued that due to McCabe's Cyclomatic Complexity being introduced using the linear programming language Fortran and as this language contains no functions or classes and hence these concepts were omitted, this measure of complexity is not suffice for code developed using the Object-Oriented Paradigm or Service-Oriented Architecture. They attempted to calculate the complexity of the Windows Communication Foundation (WCF), a framework that implements Service Oriented Architecture and presented a complex algorithm that he argued could be used in estimating the production and maintenance cost of a project using this framework (Sarwar, Ahmad, & Shahzad, 2012).

2.5 Metrics for Object-Oriented Code

2.5.1 Chidamber and Kemerer Metric Suite

Chidamber and Kemerer (1991) presented a suite of software metrics within the object-oriented paradigm. They were based on the insight and experience of existing software engineers working with object-oriented code and evaluated against widely accepted software metric evaluation criteria. It was argued that in order for the object-oriented paradigm to move from what was a 'craft' to a more conventional engineering, metrics and measures were a requirement. In addition to this, outlined areas in which such metrics could be used to aid management including: cost and schedule estimating, recruitment forecasting and future maintenance requirements.

They began by highlighting some of the criticisms of current software metrics, both procedural language metrics and object-oriented language metrics. While the former of these is more often criticised for "*being without solid theoretical base*" and "*failing to display what might be termed normal predictable behaviour*", the latter's criticisms are more focused on not supporting key object-oriented concepts such as classes, inheritance, encapsulation and message passing. They presents six metrics that are specific to object design as it is considered a "*unique aspect*" of object-oriented programming (Chidamber & Kemerer, 1991).

Chidamber and Kemerer (1991) refers to Booch's (1986) definition of object-oriented design as *"the process of identifying objects and their attributes, identifying operations suffered by and required of each object and establishing interfaces between objects"* and breaks down the design of classes into three steps, namely, defining objects, identifying object attributes and establishing the communication between objects. They cite Wand (1990) in defining two things as being coupled if one of them *"acts upon"* the other while taking Bunge's (1977) definition of the similarity of two objects, *"the intersection of the sets of properties of two objects"* as the basis for defining the cohesiveness of methods as the *"degree of similarity"* between methods. Having a high degree of similarity between methods means they have a higher degree of cohesiveness and therefore a higher degree of encapsulation. He goes on to use Bunge's (1977) definition of complexity, *"numerosity of its composition"* i.e. something complex has a large number of properties, as a base for defining complexity to be *"the cardinality of its set of properties"*.

They go on to look at the scope of properties within a class noting that Wand (1987) defines a class on *"the basis of the notion of scope"*. From this Chidamber (1991) defines two concepts relating to the inheritance hierarchy within the first being depth of inheritance, the height of a class within the inheritance tree, and secondly number of children of a class, the number of descendants of that class. He argues that these concepts are useful in determining the scope of a class such that while the depth of inheritance can establish by what degree a class is influenced by *"the properties of its ancestors"* the number of children *"indicates the potential impact on descendants"* (Chidamber & Kemerer, 1991).

Finally, they look at measures of communication within object-oriented programming noting that objects only form of communication is through message passing and therefore defines a response set for an object as *"the set of all messages that can be invoked in response to a message to the object"*. In addition he highlights that this set may include methods outside of the object as methods from one object may invoke methods from another object in response to an incoming message.

Based on four years of projects developed by software engineers Chidamber and Kemerer (1991) go on to define six metrics that while specific to object-oriented design are not language specific.

Weighted Methods Per Class (WMC)

Weighted Methods Per Class (WMC) is defined as:

$$WMC = \sum_{i=1}^n ci$$

If all static complexities are considered to be unity, $WMC = n$, the number of methods.

WMC refers directly to the complexity of a class and Chidamber (1991) argues that the number of methods and the complexity of the methods of a class are an indicator as to how much time and effort will be involved in maintaining the class. He goes on to state the larger the class, the greater the potential impact on the child classes inheriting from that class. Finally, he makes the point that classes with large numbers of methods do not lend themselves for reuse (Chidamber & Kemerer, 1991).

Depth Of Inheritance Tree (DIT)

Depth of Inheritance of a class is the DIT metric for that class. Chidamber (1991) argues that the deeper a class is within a hierarchy, the greater number of methods it will inherit and therefore the more complex the class becomes (Chidamber & Kemerer, 1991).

Number Of Children (NOC)

Number Of Children (NOC) is defined as:

$NOC =$ number of immediate sub-classes subordinated to a class in the class hierarchy.

This metric looks to determine how many subclasses will inherit the behaviour of the parent class. Chidamber (1991) argues that it is more favourable to have depth over breadth in a class hierarchy as it promotes reuse through inheritance stating it is not

considered “*good practice*” to have “*standard number of subclasses*” and that higher up classes should have more child classes than classes lower in the hierarchy. He goes on to point out that classes with a large number of children may require more testing (Chidamber & Kemerer, 1991).

Coupling between Objects (CBO)

Coupling between Objects (CBO) is defined as:

$CBO = \text{Number of non-inheritance related couples}$

This takes the concept of two objects being considered coupled if they “*act upon*” each other i.e. one invokes methods of the other. Chidamber (1991) argues that this form of coupling hampers the reuse of the class and inter-object coupling should be kept to a minimum as the class will be sensitive to changes making maintenance more difficult (Chidamber & Kemerer, 1991).

Response For a Class (RFC)

Response For a Class (RFC) is defined as:

$RFC = | RS |$ where RS is the response set for the class.

In this metric, the response set is the “*set of methods available to the object*” while the cardinality “*is a measure of the attributes of an object*”. In addition, Chidamber (1991) also notes that as it includes methods from outside the object, “*it is also a measure of communication between the objects*”. He argues that the larger the number of methods that are invoked in response to a message, the more complex that class becomes and therefore the more testing it will require. In addition, he goes on to point out that the larger the number of methods available outside of a class, the greater the knowledge required to test the class (Chidamber & Kemerer, 1991).

Lack of Cohesion in Methods (LCOM)

Lack of Cohesion in Methods (LCOM) is defined as:

Consider a Class C1 with methods M1, M2, ... Mn

Let (I_i) = set of instance variables used by method M_i .

There are n such sets $(I_1) \dots (I_n)$

LCOM = the number of disjoint sets formed by the intersection of the n sets.

This follows the concept of looking for the degree of similarity of methods by examining the common instance variables. If there are no common instance variables then the degree of similarity is zero. It should be noted that this does separate out where “*each of the methods operates on unique sets of instance variables and the case where only one method operates on a unique set of variables*”. The fewer number of disjoint sets implies greater similarity of methods (Chidamber & Kemerer, 1991).

Chidamber (1991) argues that this metric is important as a lack of cohesion between classes indicates that the class is trying to do too much and should therefore be split into multiple classes. Having low cohesion in classes indicates a degree of complexity within that class (Chidamber & Kemerer, 1991).

Metric	Object Definition	Object Attributes	Object Communication
WMC	✓	✓	
DIT	✓		
NOC	✓		
RFC		✓	✓
CBO			✓
LCOM		✓	

Figure 2.6: Mapping of Metrics to OOD Elements

Chidamber (1991) cites Weyuker’s (1988) list of properties used when evaluating software metrics, which he then applies to the previous six metrics, outlined above. The properties are

- Property 1: Non-coarseness
- Property 2: Non-uniqueness (notion of equivalence)
- Property 3: Permutation is significant
- Property 4: Implementation not function is important

- Property 5: Monotonicity
- Property 6: Non-equivalence of interaction
- Property 7: Interaction increases complexity

Chidamber (1991) found that all six failed to meet property three and property seven. In addition to this, the RFC metric failed to satisfy property six and the DIT metric failed to satisfy property five but only in the case of “*combining two objects in different parts of the tree*” (Chidamber & Kemerer, 1991).

Chidamber (1991) provides some reasoning as to why not all of the metrics satisfy all of the properties. In the case of all six failing to property three he suggests that the permutations within an object are not necessarily significant while arguing that in the case of property seven, interaction increases complexity, is not applicable to object-oriented design.

2.5.2 Building Upon Chidamber Object-Oriented Metrics

Li and Henry (1993) argue that software metrics provide a “*quantitative means*” within the software development process citing a quote by DeMarco that “*you cannot control what you cannot measure*”. In addition they argue that these same are dependent on statistical validation. Their paper, mainly concerned with the Object-Oriented programming paradigm, looks at existing software metrics, while also proposing new ones. Finally, they validate these metrics on data collected from existing commercial software systems.

They identify two categories of software metrics, the first being software product metrics, that focuses on source code and design documentation, while the second, software process metrics, focuses on the man hours involved in a project, noting that his paper is only concerned with the former.

Prior to looking at metrics specific to the Object-Oriented programming paradigm, Li and Henry (1993), reviews certain metrics used with procedural programming namely Halstead’s software science metrics and Bail’s size metrics that are lexical measures i.e. they count specific lexical tokens in a program. They also note McCabe’s

Cyclomatic Complexity, based on deriving a directed graph from the programs control flow, and a group of metrics that measure the inter-connectivity of system components.

On comparing procedural paradigm metrics to that of objected-oriented metrics they note the that “*object oriented metrics are not as numerous as those in the procedural paradigm*” They go on to look at the different characteristics exhibited between the procedural paradigm and the object-oriented one, highlighting concepts that only exist within object-oriented programming and not procedural programming, namely inheritance, classes and message passing and highlighting the fact that metrics previously mentioned do not cover such characteristics (Li & Henry, 1993).

They looked at three groups of object-oriented metrics. Group one consisted of the metrics proposed by Chidamber and Kemerer (1991), the second group looked at the metrics proposed within the paper while the last group looked at size metrics in the object-oriented paradigm (Li & Henry, 1993).

Chidamber and Kemerer (1991) who proposed six object-oriented design metrics, namely: Depth of Inheritance Tree (DIT), Number of Children (NOC), Coupling Between Objects (CBO), Response For a Class (RFC), Lack of Cohesion of Methods (LCOM), and Weighted Method per Class (WMC). Only Coupling Between Objects CBO was not used from this list (Li & Henry, 1993).

Depth of Inheritance Tree (DIT)

The DIT measures the position of a class in the inheritance hierarchy. The aim here is to gauge how many properties can be accessed from the class in question with classes lower in the hierarchy of inheritance the more properties the class may inherit from the super-classes above it. As pointed out by Li (1993) this higher the DIT metric, the harder it is to maintain the class.

The root class' DIT is zero:

DIT = inheritance level number; ranging from 0 to N; where N is positive integer (Li & Henry, 1993)

Number of Children (NOC)

The NOC measures the number of direct children a class has. It is the reverse concept of the DIT already mentioned. The more children a class has the more a change to that class can impact the system.

The calculation of the NOC is as follows:

NOC = number of direct sub-classes; ranging from 0 to N; where N is a positive integer (Li & Henry, 1993)

Response For a Class (RFC)

The RFC metric measures the cardinality of the response set of a class. Li (1993) suggests that a class with a higher the RFC metric the harder it is to maintain “*since calling a large a large number of methods in response to a message makes tracing an error more difficult*”. The RFC is calculated as follows:

RFC = number of local methods + number of methods called by local methods; ranging from 0 to N; where N is a positive integer.

Lack of Cohesion of Methods (LCOM)

LCOM measures the lack of cohesion of a class. Stevens *et al.* best define cohesion, as a “*measure of the degree to which the elements of a module belong together*”, therefore, a module considered as being highly cohesive means “*all elements are related to the performance of a single function*” (Stevens, Myers, & Constantine, 1974).

“The calculation of LCOM is the number of disjoint sets of local methods. Disjoint sets are a collection of sets that do not intersect with each other. Any two methods in one disjoint set access at least one common local instance variable:

LCOM = number of disjoint sets of local methods; no two sets intersect; any two methods in the same set share at least one local instance variable; ranging from 0 to N; where N is a positive integer” (Li & Henry, 1993).

Weighted Method per Class (WMC)

WMC measures the static complexity of all the methods. This is calculated by summing together McCabe's Cyclomatic Complexity as applied to each method in the class.

WMC = summation of the McCabe's Cyclomatic Complexity of all local methods; ranging from 0 to N; where N is a positive integer.

Li and Henry (1993) define two objects as coupled if they "*act upon each other*". The research identifies three types of coupling between objects: coupling through inheritance, coupling through message passing and coupling through data abstraction. We will now look at each of these in more detail (Li & Henry, 1993).

Coupling through inheritance

Li and Henry (1993) argue that although inheritance is used to promote software reuse it can also violate encapsulation and information hiding. It adds complexity by exposing attributes, encapsulated in the super class, in a less restricted sub-class (Li & Henry, 1993).

Coupling through message passing

In the object-oriented paradigm, messages are sent from one object to another as a form of communication. Message Passing Coupling (MPC) is a measurement of the complexity of message passing between the classes.

MPC = number of send-statements defined in a class.

This helps to establish how dependent local methods are upon methods in other classes. It does not indicate the number of messages received by the class (Li & Henry, 1993).

Coupling through Abstract Data Types (ADT)

ADT is where a data type is defined by its behaviour from the point view of the user of that data type. Li and Henry (1993) argues "*a variable declared within a class X may have a type of ADT which is another class definition, thereby causing a particular type*

of coupling between the X and the other class, since X can access the properties of the ADT class". This metric is called as Data Abstraction Coupling and is defined as:

$DAC = \text{number of ADTs defined in a class.}$

In other words, the more ADTs a class has, the more complex coupling is taking place within that class (Li & Henry, 1993).

Lastly, Li and Henry (1993) looked at size metrics used within the Object-Oriented paradigm namely Number of Methods (NOM and two additional size metrics.

Number of Methods (NOM)

As local methods define a class interface, Li (1993) argues that the Number of Methods (NOM) is the best interface metric to use. It is simply defined as

$NOM = \text{number of local methods}$

In other words, the more methods exposed by a class, the more complex that class becomes (Li & Henry, 1993).

SIZE1

Li (1993) takes the Lines of Code (LOC) metric, calling it SIZE1 and calculates it simply by counting all the semicolons in a program. It is defined as

$SIZE1 = \text{number of semicolons in a class.}$

SIZE2

The second size metric used by Li (1993), SIZE2, is the number of properties inclusive of attributes and methods that are defined in a class. It is defined as

$SIZE2 = \text{number of attributes} + \text{number of local methods.}$

In total Li (1993) defines ten metrics for use within the object-oriented paradigm. They are abbreviated as follows

DIT = Depth of Inheritance Tree

NOC = Number of Children

MPC = Message Passing Coupling

RFC = Response For Class

LCOM = Lack of Cohesion Of Methods

DAC = Data Abstraction Coupling

WMC = Weighted Method Complexity

NOM = Number of Methods

SIZE1 = Number of Semicolons per Class

SIZE2 = Number of Methods plus number of Attributes

In addition, they define the maintenance effort for the study as

Change = number of lines changed per class in its maintenance history (Li & Henry, 1993).

While noting that prior knowledge of the relationship between software maintainability and software metrics is sparse within the object-oriented paradigm, they set one of the goals of their research as to identify a relationship between the two (Li & Henry, 1993).

Li and Henry (1993) found that there was a “*strong relationship between the metrics and the maintenance effort*” within the object-oriented paradigm. In addition, it was found that the maintenance effort could be predicted from the combinations of the metrics collected from the code (Li & Henry, 1993).

Systa, Ping and Muller (2000) argues that identifying metrics within object-oriented programming requires a different approach to that of imperative programming languages in that key metrics that need to be identified are related to design and overall code quality. These include the coupling, cohesion and complexity between classes as well the complexity of the inheritance hierarchy. They point out that by identifying high or low complexity areas of a system or tightly coupled areas, can assist in decision making when adding new features or performing maintenance work.

They used the suite of object-oriented metrics as defined by Chidamber and Kemerer which were then broken into three categories: *inheritance*, *communication* and *complexity metrics*. These were then applied to FUJABA systems, a system designed to do round trip engineering using the Unified Modeling Language (UML), Story Driven Modeling (SDM), Design Patterns and Java. The results are shown in Figure 2.7 below where they are defined as a coefficient of greater than 0.4 as being correlated.

	CC	WMC	LCOM	RFC	CBO	NOC	DIT
CC	1,00	0,98	0,38	0,69	0,53	0,15	0,04
WCM	0,98	1,00	0,37	0,69	0,54	0,14	0,06
LCOM	0,38	0,37	1,00	0,41	0,18	-0,01	0,31
RFC	0,69	0,69	0,41	1,00	0,72	-0,02	0,45
CBO	0,53	0,54	0,18	0,72	1,00	-0,13	0,12
NOC	0,15	0,14	-0,01	-0,02	-0,13	1,00	-0,10
DIT	0,04	0,06	0,31	0,45	0,12	-0,10	1,00

Figure 2.7: A Correlation Matrix of Normalized Measures

2.6 Applying Object-Oriented Best Practices to Code Metrics

van Emden and Moonen (2002) proposed taking the technique of software inspections, used to improve software quality, and using it to detect bad programming design patterns within the code, known as code smells. Software inspection is the process of carefully examining code in some fashion i.e. traversing the code to generically, in an effort to identify aspects of it that may highlight positive or negative issues early in the software development lifecycle. One of the major arguments in favour of code inspections is that the cost of identifying and fixing issues in code decreases the earlier in the cycle it is discovered.

Originally, code inspections focused on low-level bug chasing, attempting to identify null references or out of bounds expectations, whereas van Emden and Moonen (2002) sought to instead use this process to detect ‘code smells’ a metaphor introduced by Martin Fowler in his book *Refactoring: Improving the Design of Existing Code*. Code smells provide a rule of thumb or a guiding principle when refactoring code and can be

used to decide *when* and *what* to refactor. Examples of code smells are duplicated code, long methods or classes, too much functionality in a class or violating encapsulation.

By introducing this process when developing code, it can ensure that coding standards are being applied to all areas when developing on a large-scale software system. Coding standards are a way for a company to ensure that all code being generated is of the same standard regardless of who and where it is written. van Emden and Moonen (2002) argues experience has shown that publishing a set of coding standards alone is not enough as there are various reasons why some engineers will simply ignore them or feel somehow restricted by them and therefore not apply them to code being developed.

The identification of code smells is not an exact science and requires a mix of factors to be taken into account when defining them for a particular project including past experience of issues that have arisen, the domain being developed and the subjective opinions of engineers working on the code. Therefore an important aspect of applying the detection of code smells in a project is that the configurable (van Emden & Moonen, 2002).

van Emden and Moonen (2002) distinguishes the two different types of code smells into “*primitive smell aspects and derived smell aspects*” for example a method containing a switch statement would be considered a primitive aspect whereas a class not using any of the methods provided by its superclass would be a derived aspect. This allowed her to apply a four step approach to analysing the code; Find all entities of interest, inspect them for primitive smell aspects, store information about entities and primitive smell aspects in a repository and finally, derive smell aspects from the repository.

van Emden and Moonen (2002) applied a defined set of code smells to the CharToon system that consisted of 46,000 lines of code (less comments and blank lines) and 147 classes. They were able to successfully highlight the code smells the majority of which came from the use of typecasts. They highlight the fact that most automatic code inspection tools, namely C analyser LINT and the Java version of which is

JLINT, focus on identifying defects in the code in an effort to build higher quality code whereas the approach here is to ensure that good design practices are adhered to. He points to an interesting area of future work that involves taking what Beck and Fowler describe as “*maintenance smells*”. These are smells that arise while code is being maintained and include concepts such as divergent change, different parts of a class are changed for different scenarios, shotgun surgery, changing many classes for one related change and parallel inheritance hierarchies, being forced to subclasses in one place in order to add a subclass elsewhere. Detecting smells such as these cannot be done by simply looking the current source code alone but would require analysing the change sets as code patches are pushed for maintenance or new features added (van Emden & Moonen, 2002).

2.7 The Law of Demeter

Perhaps one of the most significant metrics or measure that can be applied within an object-oriented program was introduced by Lieberherr, Holland and Riel (1988) and is a simple language independent rule that made applying the concept of modularity and encapsulation within an object-oriented programming more intuitive for programmers. It is known as the *Law of Demeter*. It argues that the benefit of applying the Law of Demeter included minimising code duplication, number of parameters in methods and number of methods per class. In addition, it would reduce coupling between methods, improve information hiding and narrower interfaces would lead to more maintainable code.

Based on and named after the Demeter system, which provided high-level class based object oriented systems that allowed for a larger number of utilities such as parsers, printers and type checkers to other class-based object-oriented systems, Lieberherr, Holland and Riel (1988) argued that the Law of Demeter promoted “*maintainability and comprehensibility*”. It was designed with a view to growing software over time as opposed to ‘big bang’ updates and in order to achieve this the code would have to be written in a well-formed manner.

They define the Law of Demeter, as: *“For all classes C, and for all methods M attached to C, all objects to which M sends a message must be instances of classes associated with the following classes:*

- 1. The argument classes of M (including C).*
- 2. The instance variable classes of C.*

Objects created by M, or by functions or methods which M calls, and objects in global variables are considered as arguments of M” (K. Lieberherr, Holland, & Riel, 1988).

The Law of Demeter has two primary purposes; making modifications to program simpler and reducing the complexity of the code by restricting the *“number of types”* the programmer is aware of when writing within a method.

The motivation behind the Law of Demeter is to make the software *“as modular as possible”* and by abiding by it, methods will only be aware of the *“immediate structure”* of the class to which it belongs. This in turn means that any changes to the class will only require examining the methods within that class as to what impact the change will have. It reduces *“nested message sending’s”* and prohibits the nesting of *“generic accessor function calls”* i.e. calls that return objects that existed before the function is called (K. Lieberherr *et al.*, 1988).

Lieberherr’s (1988) aim was to condense many well-known principles into a *“single statement”*, one that could be checked at compile time. They include coupling, information hiding, information restriction, localising information, narrow interfaces and structural induction.

Coupling

Generally it is considered good practice, within object-oriented software design, to have minimal coupling between classes. By abiding by the Law of Demeter, which limits the methods that can be called from within a given method and therefore, reduces the amount of coupling that can occur between various classes within a program (K. Lieberherr *et al.*, 1988).

Information hiding

The Law of Demeter promotes information hiding by preventing methods from retrieving a “*subpart of an object*” and enforcing them to traverse intermediate methods in small steps to perform the same result. Even though programmers of object-oriented languages can prevent the use of certain methods by making them private, a feature that complements the law, Lieberherr *et al.* (1988) goes further than and argues that these methods should still be used in a restricted way regardless of them being public (K. Lieberherr *et al.*, 1988).

Information restriction

Based on Parnas (1986) work on the modular structure of complex systems, Lieberherr *et al.* (1988) restricts the use generic method calls and argues that information restriction complements information hiding. As was the case with information hiding, methods can be public, but their use is restricted.

Localising information

The Law of Demeter ensures that on examining a method, a programmer is only required to be aware of the types that are closely related to the class within which the method exists. This allows them to be independent of the rest of the system, leading to less complexity within the system (K. Lieberherr *et al.*, 1988).

Narrow interfaces

The Law of Demeter promotes the use of narrow interfaces between interacting entities. Lieberherr *et al.* (1988) argues that a method should only have access to as much information as it requires in order for it do its job.

Structural induction

The Law of Demeter is based upon the fundamental thesis of Denotational Semantics i.e. “*The meaning of a phrase is a function of the meanings of its immediate constituents*”. Lieberherr *et al.* (1988) highlight a trade-off to applying the Law of Demeter that although it decreases the complexity of the methods, it increases the number of methods. This may lead to an issue where there are “*too many operations in a type*” i.e. too many methods in a class. In order to solve this issue, he argues that

functional abstractions should not be provided via a method but instead a module, therefore hiding all of the underlying low-level methods.

They outline an approach when creating instances of classes where a factory class is employed that contains the code that creates the new class and in turn is used to provide instances of the class when required. This in effect prevents multiple places in the code where a class is created and therefore needs to be updated when a change is required (K. Lieberherr *et al.*, 1988).

They outline two variations of the Law of Demeter, namely weak and strong, the former defining instance variables as only those within the given class while the later defines them as anything within the given class and any inherited from parent classes (K. Lieberherr *et al.*, 1988).

2.8 Conclusions

This chapter began defining some of the concepts used throughout this research. It looked at how real world items are represented within the object-oriented programming paradigm through the use of classes.

From there, it studied in detail McCabe's Cyclomatic Complexity, a technique that allows a piece of to be measured with relation to its perceived complexity. Building upon this, the chapter touched on other metrics such as Halstead's E and then examined various critiques of these theories.

Among the criticisms of McCabe's Cyclomatic Complexity was Sheppard who cited Myer's finding that it failed to take account for ELSE branches. Sheppard also took exception to the fact that it revolved around FORTRAN and not more modern languages of the time.

Regardless of these criticisms, many of which are valid, McCabe's complexity measure has become a key cornerstone that many other theories have been built around.

The second half of the chapter began by looking at Chidamber and Kemerer metric suite for the object-oriented paradigm and continued by examining in detail how this suite was then built upon over time. The work of Chidamber and Kemerer was ultimately found to be both timely and accurate. The suite of metrics as defined here became a foundation on which many other theories have been built over time.

Following this, the research looked van Emden and Moonen who offered a new slant on code metrics by introducing the concept of analyzing code for what are known as ‘code smells’ i.e. code that is considered to be bad practice, in an attempt to identify this in automated fashion as new code is committed to a project.

Arguably one of the key offshoots from the traditional thinking in the area of code metrics was van Emden and Moonen’s work that merged both code metrics and the concept of ‘code smells’. This brought a new lens that code metrics could be viewed through. No longer were metrics solely for the purpose of detecting coding errors but were now being used to identify bad practice i.e. not something a modern day compiler might reject but something that had, up until then, only be analyzed by the human eye.

The chapter concludes by looking at one of the most important pieces of research with regard to applying the concept of modularity and encapsulation with object-oriented programming that is known as The Law of Demeter. Purposely designed to be simple in nature and language independent, it assists programmers in applying a key principal when developing software. Building upon this, the next chapter will look at various metrics that have grown out of these theories and importantly the various attempts of applying the metrics within established commercial organisations. In addition, it also seek to build upon van Emden and Moonen’s work by introducing the concept of ‘clean code’ put forward by Martin (2008) with a view to analyzing its impact on code metrics.

3. Applied Metrics

3.1 Introduction

Following on from the previous chapter that looked at some of the fundamental theories within code metrics, this chapter will be skewed towards how these and other metrics fared when applied within established commercial organisations. In addition, it will also look at some new, albeit less well-known metrics that have been put forward over the years, some of which could be considered off-shoots from what was in the previous chapter.

The chapter will close by looking to build upon van Emden (2002), where the concept of introducing ‘code smells’ analysis into the area code metrics, and introduce some key aspects of work by Martin (2008) from his ‘clean code’ concepts. This includes challenging some of the established thinking around comments in code and identifies practices to follow in naming and constructing functions. In addition to this, it will also look at what Martin (2008) calls the *Stepdown Rule*.

3.2 Alternative Approaches to Code Metrics

This section will look at a range of alternative approaches to code metrics.

3.2.1 Service Oriented Design

Pereplechikov, Ryan and Frampton (2007) presented a set of design-level metrics that examined the various types of cohesion within service-oriented design. Taking the definition of cohesion from Stevens *et al.* as a “*measure of the degree to which the elements of a module belong together*” therefore a module to be considered as being highly cohesive means “*all elements are related to the performance of a single function*”. They expand on this citing the semantic categories of cohesion, namely: Coincidental, Logical, Temporal, Procedural, Communicational, Sequential and Functional. This ordinal scale ranges from the weakest to the strongest form, in this case Coincidental to Functional. In the case of object-oriented design Eder *et al.* has redefined cohesion as the “*degree to which the methods and attributes of a class*

belong together” where the semantic ordinal scale for object-oriented classes is: Separable, Multi-faced, Non-delegated, Concealed and Model.

While noting that the previous work did not map cohesive metrics to their semantic category, they do just that and map quantitative metrics to the qualitative categories of cohesion. They observe some differences between traditional software systems that could be as a collection of interconnected objects and Service Oriented Architecture that break the application into stateless services that are autonomous to others within the system. This concept adds a new layer of abstraction, a service on top of classes that serve to aggregate groups of methods.

They redefine the cohesion categories while proposing two additional ones, external and implementation thus giving a completed list of: Coincidental, Logical, Communicational, External, Sequential, Implementation and Conceptual. Each of these is defined as follows:

Categories	Description
Coincidental	A service encapsulates unrelated functionality i.e. its interface provides unrelated functionality that has no meaningful relationship.
Logical	A service provides common utility functionality e.g. data update and/or retrieval.
Communicational	Service operations use the same data abstractions.
External	Service consumers use all service operations.
Sequential	Service operations are sequentially connected.
Implementation	All service interface operations are implemented by the same implementation operations.
Conceptual	There are meaningful semantic relationships between operations of a service in terms of some identifiable domain level concept.

Figure 3.1: Categories of Cohesion

Pereplechikov *et al.* (2007) identifies the following metrics: Service Interface Data Cohesion (SIDC), Service Interface Usage Cohesion (SIUC), Service Sequential

Usage Cohesion (SSUC), Strict Service Implementation Cohesion (SSIC), Loose Service Implementation Cohesion (LSIC) and Total Interface Cohesion of a Service (TICS).

3.2.2 Metrics using Program Slicing

Al Dallal (2009) looked at a similarity-based functional cohesion metric and argued that cohesion is an important factor to be considered when evaluating software design. From a software engineering point of view, cohesion is desirable as it provides reusability and maintainability modules. In her paper, Al Dallal (2009) introduces a metric based on the strongest level of cohesion, functional cohesion that refers to how closely module parts are related based on outputs. The metric, similarity-based functional cohesion (SBFC), measures the functional cohesion of a module for both procedural and object-oriented languages, by looking at the “*degree of similarity between the data slices*” of a module.

Al Dallal (2009) argues that cohesion is an indication of a high quality software design. A highly cohesive module is one that cannot be easily split into separate modules and he looks to measure basic cohesiveness of modules. Citing Yourdon and Constantine’s proposed seven levels of cohesiveness each of which indicate how much a module contributes towards performing a task. In ascending order, in accordance with their desirability, they are: coincidental, logical, temporal, procedural, communicational, sequential and functional. On the other hand, Emerson, using a control flow graph as the basis for representing a module, proposed three levels: data cohesion, control cohesion and superficial cohesion. When examining levels of cohesion in a module, the module is seen as a set of process components and a module that has a single processing or highly related one is considered to be highly cohesive. He contends that functional cohesion is the most desirable level as it provides for reusability and maintainability.

Al Dallal (2009) looked at Weiser’s program slicing concept, where “*the value of a variable at some point in a program is called a program slice*” and sought to take Longworth’s suggestion of using sliced-based metrics to indicate cohesiveness. Ott

and Thuss first introduced the idea of slice-based metrics where slices were the output of each module as the output variables indicated the tasks the module.

Bieman and Ott introduced the concept of applying data slices to the measuring of functional cohesion by measuring output variables of modules. In addition, Al Dallal (2009) also noted metrics used to measure various types of cohesion including between methods in a class, lack of cohesion between methods, normalised Hammering distance (NHD), scaled Normalised Hammering distance, class cohesion (CC) and a sensitive class cohesion metric (SCOM).

Al Dallal (2009) introduces a new metric called similarity-based functional cohesion (SBFC) used to measure the function cohesiveness within procedural programs as well as methods within object-oriented ones. The measurement technique employed by SBFC is based upon Bieman and Ott's data slicing concept of measuring cohesion. Taking three metrics from Bieman and Ott, metrics that at times contradict each other when measuring cohesiveness, the SBFC metric provides a single measure that eliminates the ambiguity caused by having three individual metrics. He conducted an empirical study that demonstrated high correlations between similarity-based functional cohesion SBFC and the Bieman and Ott metrics. In addition, SBFC satisfied all the properties as outlined by Briand *et al.* for module cohesiveness and is also useful when refactoring weakly cohesive modules. In effect, the similarity-based functional cohesion (SFBC) "*measures similarity between pairs of data slices*".

"Program slicing is the task of finding all the statements in a program that directly or indirectly influence the value of a variable occurrence". This can be either static or dynamic, where static involves finding all statements that affect the value of a variable and dynamic finds the slice based on a set of inputs. Al Dallal's (2009) paper is mainly concerned with intra-procedural slicing, the process of computing the slices of a given procedure as opposed to inter-procedural slicing, computing the slices of a multi-procedural program. From an intra-procedural point, there are three main algorithms, data flow equations, information flow relations and program dependence graphs (PDG) and Al Dallal argues that of these PDG's is the more efficient. PDG represents simple statements, that include assignment statement, read and write statements, and control predicates as nodes. Conditional and compound statements are represented by

more than one node. In addition, there are two types of edges within a PDG, a data dependence edge and a control dependence edge where a data dependence edge indicates that the connected nodes have a computation in one node that depends directly on a value computed in the other node and a control dependence edge “*implies that the result of the predicate expression*” of one node is a factor for deciding whether to execute code at the other node (Al Dallal, 2009).

```

1 void NumberAttributes(int n, int &sum, double &avg, int &product) {
2     int i=1;
3     sum=0;
4     product=1;
5     while (i<=n) {
6         sum=sum+i;
7         product=product*i;
8         i=i+1;
9     }
10    avg=static_cast<double>(sum)/n;
11 }

```

Figure 3.2: C Function that computes the sum, average and product numbers from 1 to n

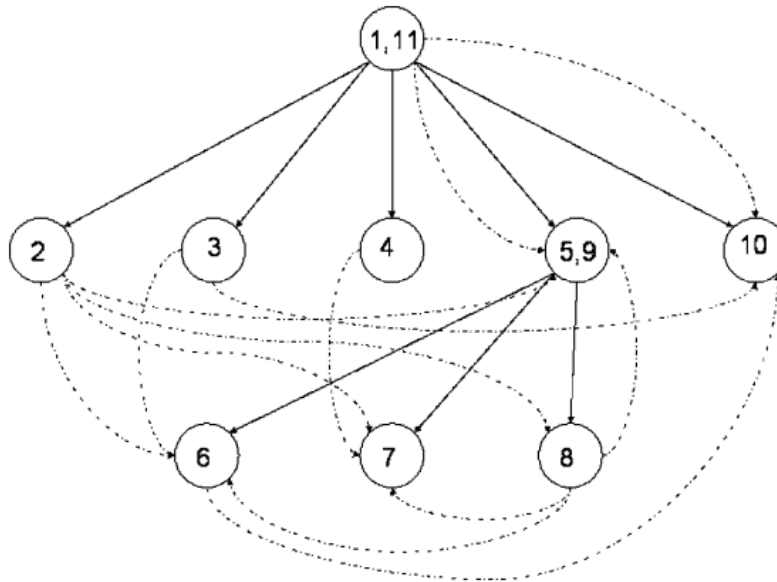


Figure 3.3: C function in Figure 3.2 in a program dependence graph

Figure 3.2 above shows an example C function that computes the sum, average and product of numbers from 1 to n, where n is an integer value ≥ 1 while Figure 3.3 is the same C function represented as a program dependence graph (PDG) where solid lines represent control dependence edge and dotted lines represent data dependence

edges. Citing Briand and Ott four properties for cohesion metrics, namely, non-negativity and normalisation, null value and maximum value, monotonicity and cohesive modules, Al Dallal (2009) argues that a cohesion metric must satisfy all these properties to be considered a cohesive indicator.

Al Dallal (2009) further cites Bieman and Ott as having introduced the concept of data slicing and “*applied it as an abstraction to measure the functional cohesion of the module*”. Using the C function example in Figure 3.2, the data slices found for C function are shown in Figure 3.4 below

Slice attributes		Slice contents (line numbers)
variable	Line no.	
sum	6	1, 2, 3, 5, 6, 8
avg	10	1, 2, 3, 5, 6, 8, 10
product	7	1, 2, 4, 5, 7, 8

Figure 3.4: Data slices for C function as defined in Figure 3.2

Al Dallal (2009) cites Bieman and Ott categorisation of data slices into two: glue token, a data token that exists in more than one data slice and super-glue token, a data token that exists in all slices of data and glue stickiness while stickiness of a glue token was based on how many data slices it bound. In their work, Bieman and Ott used three cohesion metrics, namely strong functional cohesion (SFC), ratio of super-glue tokens to total number of data tokens in a module, weak functional cohesion (WFC), ratio of glue tokens to total number of data tokens in a module and adhesiveness (A) of the module, the ratio of total adhesiveness of glue tokens to the total possible adhesiveness or each data token is used by each data slice.

Noting that each of the three metrics have a value of between zero and one, Al Dallal (2009) highlights the difficulty with having three metrics when only attempting measure one thing, namely that three figures do not provide immediate clarity and the second being the difficulty when trying to compare two or more modules. For example, the metrics may be SFC=0, WFC=1 and A=0.67 which does not make it immediately obvious as to whether the module has low cohesion or high cohesion and

secondly how does that compare to a module with cohesion measurements of $SFC=0.25$, $WFC=0.75$ and $A=0.58$.

Al Dallal (2009) cites Bonjia and Kidanmariam metric, class cohesion that uses the degree of similarity between methods as the basis for the measurement, that is the ratio of the number of shared attributes versus the number of distinct attributes referenced by both methods. From this the cohesion is defined as “*ratio of the summation of the similarities between all pairs of methods to the total number of possible pairs of methods*”.

Al Dallal (2009) notes that most cohesion metrics for object-oriented programs examine the interactions between methods and instance variables. The similarity-based functional cohesion (SBFC) relies on Bieman and Ott’s work where data slices of the modules are used as abstractions for measuring functional cohesion. By taking each pair of data slices individually as opposed to all at once, the SBFC metric is more precise and sensitive to changes.

Al Dallal (2009) concludes that the simplicity of the similarity-based functional cohesion (SBFC) metric will lead to a greater adoption among software engineers therefore improving the quality and modularity of products in the long run. An experimental study that compared SBFC to Bieman and Ott’s metrics supported the hypothesis that SFBC was a cohesion indicator.

3.2.3 Tools for Metrics

Novak, Krajnc and Zontar (2010) argued that static code analysis tools were becoming more crucial in the software development lifecycle (SDLC) and created taxonomy of commonly used tools. He classified the tools in categories: technology, availability of rules and extensibility. The tools use techniques including syntactic pattern matching, data flow analysis, model checking and verification theorems. They looked the emergence of code reviews as a “*successful fight against maintainability problems*” as they can increase reliability and security while noting the process of manually reviewing code by senior engineers takes time. In contrast the use of automated tools

are fast, can be run more often while containing the same level of knowledge as a human reviewer (Novak, Krajnc, & Zontar, 2010).

They define static code analysis as the analysis of “*software that is executed without actually running programs*”. Static code analysers generally build state models of the code and then determine how the program reacts in each state. This compares to dynamic analysis, where the program is executed, normally with test inputs. In addition, they also state that both methods are prone to false positives.

They highlight some of the issues that static code analysers can detect including common mistakes that compilers do not check for such as “*memory overruns, cross site scripting attacks, injections and various other boundary cases* “. Static code analysers operate in different ways, some on source code and intermediate code others examine libraries created. The types of issues that static code analysers can identify are:

- Syntactic problems
- Unreachable source code
- Undeclared variables
- Uninitialized variables
- Unused functions and methods
- Variables used prior to initialisation
- Unused values from functions
- Incorrect use of pointers

While there are many benefits of using static code analysers, they come up short in certain aspects, including identification of poor code design or malicious code.

Novak *et al.* (2010) concludes that while there are many benefits to using static code analysis tools including the early detection of issues within the software development lifecycle, they should be used in conjunction of manual code analysis and other code reviewing tools.

3.2.4 Code Comments

Steidl *et al.* (2013) argued that although many software developers consider code comments to be crucial in understanding the development most quality analysis systems ignore system commenting on evaluating the system. The paper presented a detailed approach to categorizing comments using machine learning techniques and by providing metrics tailored to suit each category showed how the quality aspects of the model could be assessed.

Steidl defined the problem as quality analysis systems either ignoring comments completely or restricting the comment ratio as a metric i.e. not giving enough value to it and provided a comment based classification that resulted in a semi-automated approach for both quantitative and qualitative evaluation of the quality of the comments.

It found that comment classification provided better quantitative insights into system documentation over the existing simple comment ratio metric.

3.2.5 Validation of Code Metrics

Schneidewind (1992) argued that software metrics themselves should be subject to the same rigor as all other areas of engineering and therefore need to be validated in the same way to ensure they measure what they say they measure prior to use. His paper proposes a “*validation methodology*” that is not specific to any particular metric and is mapped to six criteria: association, consistency, discriminative power, tracking, predictability and repeatability that allow the user to gain an understanding of how metric can be applied (Schneidewind, 1992).

Meneely, Smith and Williams (2013) argued that the burden of proof in validating new software metrics resulted in a debate on what constituted a “*valid*” software metric. His paper conducted a systematic literature review that extracted forty-seven unique validation criteria and performed comparative analysis on them. It was found that there was wide “*diversity of motivations and philosophies*” that indicated that the process was a complex one.

Anderson (2004) cited the ISO 9126 Production Evaluation Standard to identify all the attributes of high quality software: Functionality, Reliability, Usability, Efficiency, Maintainability and Portability. He goes on to identify a range of metrics that can be used to determine a risk factor as to the likelihood of defects occurring or the difficulty level of maintenance required for the system. The metrics identified are: McCabe's Cyclomatic Complexity metrics which measures linearly independent paths through a program module, Halstead Complexity measures that analyse operators and operands, Henry and Kafura metrics that examine coupling between modules and Troy and Zweben metrics which look at the complexity of structure and calls coming in and out (Anderson, 2004).

3.3 Code Metrics & Commercial Applications

In addition to the theorising of code metric many practitioners of software development introduced code metrics to large-scale industrial projects.

3.3.1 Tackling Project Costs

Coleman, Ash, Lowther, and Oman (1994) cite several examples of noteworthy people estimating the cost of maintaining software over its initial development cost ranging from Fred Brooks estimate that maintenance was *"40% or more of the cost of developing it"* while Dean Morton, executive vice president and chief operating officer at Hewlett-Packard estimated between 60 to 80% of research and development personnel were involved in *"maintenance activities"* and went to say that 40 to 60 per cent *"of the cost of production"* was maintenance related.

Taking these statistics as to the cost of maintenance in software related projects, Coleman *et al.* (1994) went on to demonstrate how analysing software to generate data on maintainability could be used to help when making business related decisions for these projects, including buy versus build decisions and subcomponent quality analysis. They looked at five previously defined methods for quantifying software maintainability arguing that all five *"compute reasonably accurate maintainability scores"* based on existing metrics

- Hierarchical multidimensional assessment models
- Polynomial regression models
- An aggregate complexity measure
- Principal components analysis
- Factor analysis

The Hewlett-Packard (HP) management team selected hierarchical multidimensional assessment models and Polynomial regression models, as they required quick and easy indices for use by engineers. These were then applied to industrial system within HP.

Hierarchical multidimensional assessment models

Using Oman and Hagemester's hierarchy model, Coleman *et al.* (1994), divided up maintainability into "*three underlying dimensions or attributes*", namely control structure, information structure and typography, naming and commenting. After identifying each metric, an "*index of maintainability for each dimension can be defined as a function of those metrics*". The three dimensional scores can then be summed up to give an overall maintainability index. In this case they "*used existing metrics to calculate a deviation from acceptable ranges and then use the inverse of that deviation as an index of quality*".

They go on to explain that by using a method called *weight and trigger-point-analysis* can be used to "*quantify maintainability by calculating a "degree of fit" from a table of acceptable metric ranges*". When the value falls outside of this range, it is an indication that the maintainability is lower for that component. For example, if the "*acceptable range*" for the *average lines of code* is between 5 and 75, then values falling outside of this range would be considered as an indication that the code is of lesser quality.

Polynomial regression models

Coleman *et al.* (1994) describes polynomial regression models as a "*statistical method for predicting values of one or more response (dependent) variables from a college of predictor (independent) variables*". He explains that these models were intended for

use by maintenance practitioners and as such were “*calibrated to HP engineers’ subjective evaluation*”. One noteworthy finding was the fact that of fifty regression models constructed, while trying to identify simple models that were generic enough to use on wide range of systems, “*all tests clearly indicated that Halstead’s volume and effort metrics were the best predictors of maintainability*”. The most applicable regression model was a “*four-metric polynomial based on Halstead’s effort metric and on metrics measuring extended Cyclomatic Complexity, lines of code and number of comments*”. It should be noted that regarding number of comments metric that small modules with large blocks of comments skewed the results and therefore required the model to be tweaked to measure comments as a percentage with an upper ceiling limit imposed.

Coleman *et al.* (1994) applied the metrics outlined above to industrial systems within Hewlett-Packard and the US Department of Defence and presented the results unaltered. Overall, they concluded that “*automated maintainability analysis*” is possible at various system levels and the metrics were applied to eleven software systems and assisted in making decisions regarding including buy versus build decisions and subcomponent quality analysis.

3.3.2 Evaluation of Code Metrics within Hewlett-Packard

Grady (1994) argued that the major uses of software metrics were project estimation and progress monitoring, evaluation of work products, process improvement through failure analysis and experimental validation of best practices. Basing his paper on practical experience working on projects with Hewlett-Packard, he broke down each area in detail but this paper is only concerned with the section; evaluation of work products (Grady, 1994).

Cyclomatic Complexity

Grady (1994) outlines how Hewlett-Packard (HP) successfully used Cyclomatic Complexity, a metric based on a programs decision count i.e. all the programs conditional statements to build graphs that help locate problem areas within the code. During one such study involving over 800,000 lines of code the engineers plotted a relationship between program decision counts and code updates checked into the

source control which found that seventy five per cent of the changes fell where the decision count was highest. *“The number of updates was proportional to the number of decision statements”*. By drawing a trend line and adding in the cost and schedule effects of changing modules more than three times *“they concluded that fourteen was the maximum decision count to allow in a program”*. He duly notes that McCabe originally suggested ten based on testing difficulty. An additional interesting observation by Grady (1994) is when he points out that the Cyclomatic Complexity is a *“measure of control complexity”* and therefore is more valuable in control-oriented applications as was the case here as opposed to data-oriented applications.

Design complexity

Grady (1994) looks at a metric used in data-oriented complexity called fanout squared. He explains that the fanout of a module is number of calls from that module and that fanout squared was used in three studies and had shown to correlate well to *“the probability of defects”*. What is most interesting with this particular metric is that it can be calculated prior to writing the code i.e. at the design phase. Grady (1994) cited an example where a defect-prone module was identified as the source to fifty per cent of the defects although it only contained eight per cent of the code. When the fanout squared of this module was examined against all the thirteen modules in the system, it had the largest number of them all. Overall, there was a strong correlation between the fanout squared metric and the post-release defect count.

3.3.3 Alternative Takes on Applying Code Metrics

Constantine (1996) classifies metrics specific to software usability into three categories: preference metrics, that are that assess aspects such user interface design for ease of use, performance metrics, that quantifiable metrics which attempt to determine error rates or execution time and predictive or design metrics that look to evaluate the properties of the design such as screen layouts. His paper looked at user interface design metric called Visual Coherence, based on the software engineering concept of cohesion, the degree to which component parts are considered conceptually or semantically interrelated. Visual Coherence is used to *“measure how closely an arrangement of visual components matches the semantic relationships among the concepts represented by that component”*. The use of Visual Coherence as a predictive

metric was found to assist in predicting “*user preferences, ease of use ratings of interpretability, attractiveness, and quality of layout*” (Constantine, 1996).

Kontogiannis (1997) argues that the cloning of code fragments in large-scale systems results in “*redundant code, higher maintenance costs, and less modular systems*” and proposes to use five metrics, including McCabe’s and Kafura complexity metrics, to identify this code duplication including scenarios where instance variable names differ and statements have been added. By using these standard software engineering metrics, Kontogiannis (1997) attempted to identify the duplicated code by “*examining structural and data flow characteristics*” and found that it could successfully retrieve sixty per cent of duplicated code (Kontogiannis, 1997).

Plosch *et al.* (2010) argues that software quality is a key factor for any software product and continuous monitoring is an “*indispensable*” task in the software development lifecycle. By analysing the outputs of different static code analysers, they developed a method systematically assessing and improving the quality of software development projects. Their paper focused on code quality as opposed to overall software quality and defines it as the “*capability of source code to satisfy the state and implied needs for the current software project*”. While noting that continuous quality monitoring is important as fixing issues sooner in the development cycle is less expensive, They also cite Balzert who states the “*early fixing of source code related problems prevents propagation of these errors into subsequent phases*” (Plosch *et al.*, 2010).

While pointing to previously documented “*classical*” software metrics like McCabe’s Cyclomatic Complexity or Chidamber and Kemerer’s object-oriented metric suite, Plosch *et al.*’s paper (2010) argues that being solely reliant on metric is not sufficient and static code analysers are indispensable to ensure code is adhering to best practice guidelines.

Although there is a range of tools available to perform static code analysis, none supported “*continuous code quality management*”, hence Plosch *et al.* (2010) developed a tool called ConQAT that integrates the results of various other static code analysers into a dashboard overview. They looked at applying the Code Quality

Monitoring Method (CQMM) that systematically improves code quality of a software project and based on the Evaluation Method for Internal Software Quality (EMISQ). While EMISQ worked on the basis of a “one-time assessment”, CQMM extends this to continuously measuring by automation many of the steps involved.

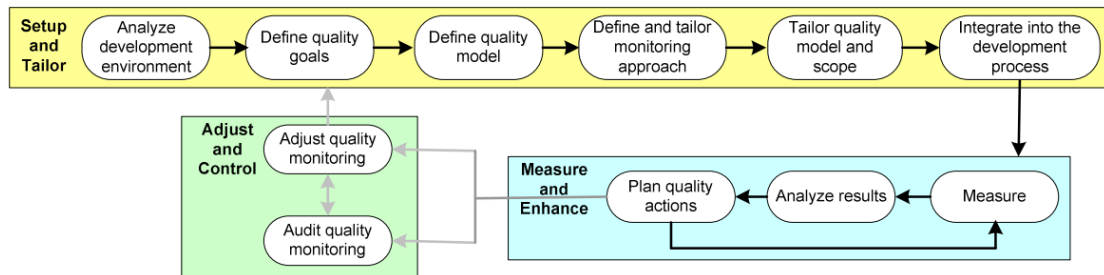


Figure 3.5: CQMM activities

As shown in Figure 3.5, CQMM consists of eleven activities that are divided into three major groups: Setup and Tailor, Measure and Enhance and Adjust and Control.

Plosch *et al.* (2010) conducted feasibility studies with selected projects with two software projects written in Java with 30,000 and 10,000 lines of code respectively. This concluded that although the application of the method worked the experiments were too limited to gain any further general conclusions. The pilot projects did indicate that CQMM could be integrated into development projects with different goals. At time of publication, CQMM was due to be rolled out in 25 projects within the Indian division of Siemens AG.

3.3.4 Classification of Metrics based on Defect Categories

Tosun *et al.* (2011) conducted a case study on software metrics for different defect categories. He argued that although past research had shown “code, shurn and network metrcis” as indicators of defects, that not all metric sets are indicators within all defect categories and only one of the metric types may be responsible for the majority of a defect category. Previous work by Tosun *et al.* indicated that “defect category sensitive prediction models” preform better than general models as “each category has different characteristics in terms of metrics”.

Building on previous work, Tosun *et al.* (2011) extended the model taking into account churn, code and network metrics and found that churn metrics were best for predicting all defects while code and network metrics correlation varied depending on category. For example, network metrics had a higher correlation than code metrics for defects reported during functional testing where the reverse was found when defects were reported during system testing.

Tosun (2011) set out to investigate “*the most representative metric set for predicting different defect categories*” and outlined three research objectives:

- Analyse the relationship between metric sets and defect categories
- Predict defects using the most representative metric set
- Build specialised prediction models for three defect categories

They analysed the history of a large-scale enterprise product in order to extract “*static code and churn metrics at software method/function level*“. The product spanned a history of over 20 years of updates of which they selected a part of this product consisting of 500,000 lines of code.

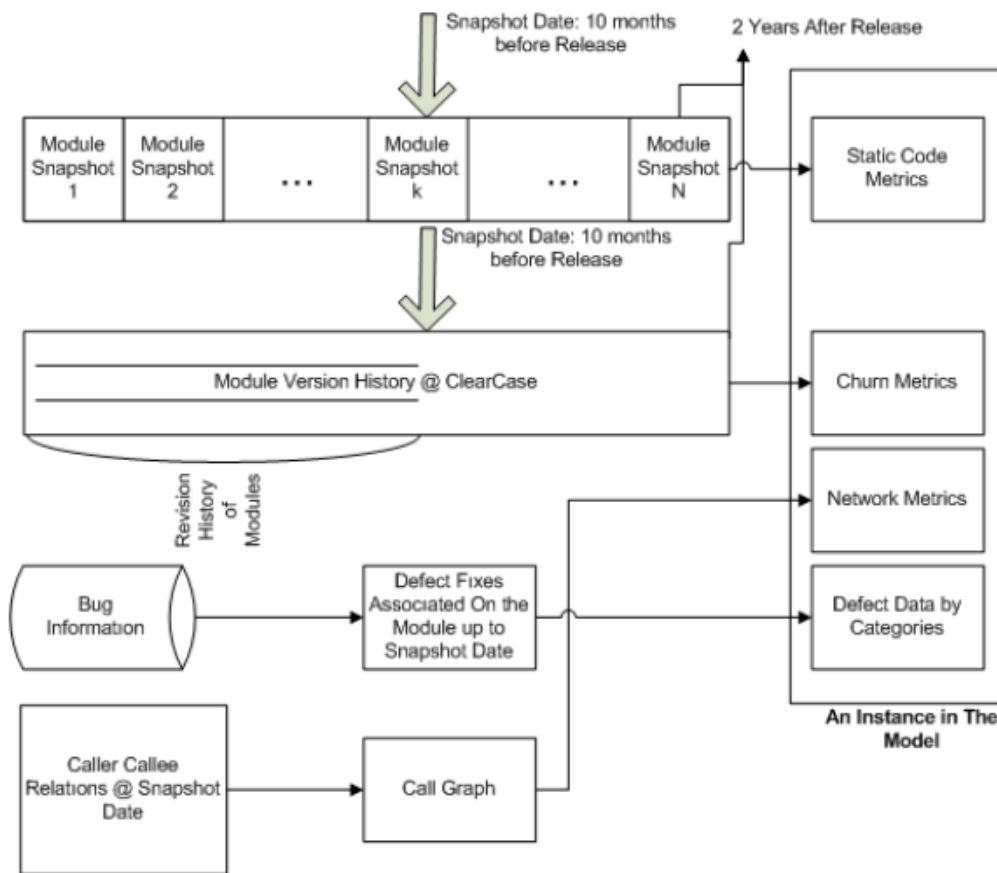


Figure 3.6: The process of extracting static code, churn and network metrics

Tosun (2011) extracted static code, churn and network metrics six months prior to release date, the process of which is shown in Figure 3.6 above, and this provided a base point. To extract network metrics, they built a call graph of the network by extracting caller-callee relations.

Churn metrics			
<i>Number of edits</i>	number of edits done on a method	<i>Lines Added</i>	total number of lines added
<i>Number of unique committers</i>	number of unique committers edited a method	<i>Lines Re-moved</i>	total number of lines removed
<i>Total churn</i>	total number of lines edited		

Figure 3.7: Churn Metrics

Attribute	Description	Attribute	Description
Code metrics			
<i>Cyclomatic density, $vd(G)$</i>	the ratio of module's cyclomatic complexity to its length	<i>Essential complexity, $ev(G)$</i>	the degree to which a module contains unstructured constructs
<i>Design density, $dd(G)$</i>	condition/ decision	<i>Cyclomatic complexity, $v(G)$</i>	# linearly independent paths
<i>Essential density, $ed(G)$</i>	$(ev(G)-1)/(v(G)-1)$	<i>Maintenance severity</i>	$ev(G)/v(G)$
<i>Decision count</i>	# of decision points	<i>Condition count</i>	# of conditionals
<i>Branch count</i>	# of branches	<i>Call pairs</i>	# calls to other functions
<i>Multiple condition count</i>	# multiple conditions	<i>Normalized cyclomatic complexity, $norm v(G)$</i>	$v(G) / nl.$
<i>Unique operands</i>	$n1$	<i>Total operators</i>	$N1$
<i>Total operands</i>	$N2$	<i>Unique operators</i>	$n2$
<i>Difficulty (D)</i>	$1/L$	<i>Length (N)</i>	$N1 + N2$
<i>Level (L)</i>	$(2/n1)*(n2/N2)$	<i>Programming effort (E)</i>	$D*V$
<i>Volume (V)</i>	$N*\log(n)$	<i>Programming time (T)</i>	$E/18$
<i>Vocabulary</i>	$n1+n2$	<i>Error estimate</i>	estimated # errors
<i>Executable LOC</i>	Source lines of code containing only code and white space	<i>Lines of Comment</i>	Source lines of comments
<i>Blank LOC</i>	Blank lines of code	<i>Total LOC</i>	Total source lines of code
<i>Code and comment LOC</i>	Source lines of code containing only code and comments		

Figure 3.8: Code Metrics

Network Metrics			
<i>Average Shortest Path</i>	fraction of shortest paths that node is a part	<i>Degree</i>	direct call count of a module
<i>Betweenness</i>	# shortest paths that contains the software module X over all shortest paths between any node i,j	<i>In Degree Centrality</i>	# modules called directly by the software module X over all the software modules
<i>Closeness Centrality</i>	Mean shortest path from the software module X to other reachable software modules	<i>Out Degree Centrality</i>	# modules that calls directly the software module X over all the software modules
<i>Page rank</i>	relative importance of nodes		

Figure 3.9: Network Metrics

Once the snapshot was taken any defect recorded against a module, labelled that module as defective. Defects were also labelled by their categories: field defects, system testing defects and functional testing defects (Tosun *et al.*, 2011).

Tosun *et al.* (2011) began by conducting a statistical analysis to “understand the relationship between metrics and defect categories”. From this it was evident that “churn metrics have strong correlations with all defect categories” and that code metrics are more significant than system testing defects over network metrics. In addition, network metrics have higher correlations with functional testing defects and field defects.

3.3.5 Aggregation of Code Metrics

Mordal-Manet *et al.* (2011) built an empirical model for continuous and weighted metric aggregation. Arguing that software metrics alone are not enough to determine the quality of software and hence there is a trend towards aggregating various metrics in an effort to make better determinations when analysing software for quality purposes. Citing the example of combining Cyclomatic Complexity with test coverage highlights the importance of covering complex methods over accessors, they present the issues they encountered on designing a quality model called Squale, a model

validated over a four year period with two large multinational companies: Air France-KLM and PSA Peugeot-Citroen (Mordal-Manet *et al.*, 2011).

While noting the fact that software metrics are becoming more of an objective measurement of software quality, they argue that these metrics computed individually and therefore do provide an overall quality assessment at a higher level. Although aggregation models such as the ISO 9126 have been created, Mordal-Manet *et al.* (2011) notes several issues with it, including the fact it is difficult to compute, models based on it provide overall assesment with simple averages weighing which is seeing as just smoothing out results and often results are translated into discrete scales i.e. good, average or bad. They explain how the Squal Model uses formulas to aggregate metrics in an effort to provide a quality indication for the overall project. This model was designed in 2006 and put into production in Qualixo, Air France-KLM and PSA Peugeot-Citroen.

The Squal Model is composed of four levels, divided into two groups.

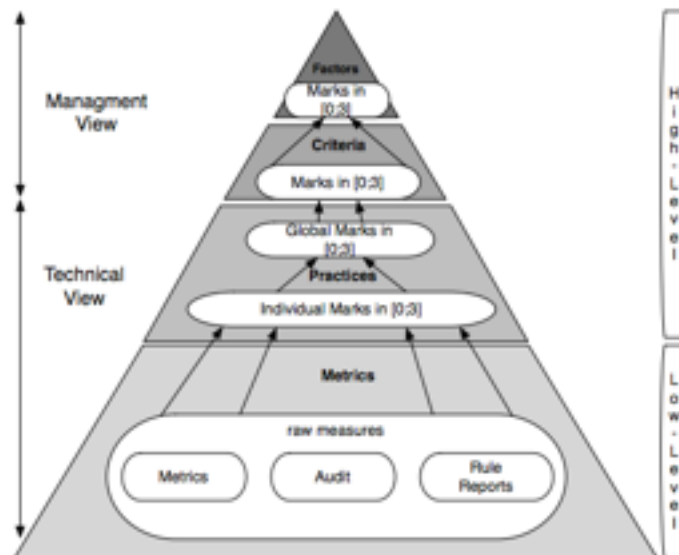


Figure 3.10: The Squal Model

The model is composed of metrics, high-level criteria and factors. As Mordal-Manet *et al.* (2011) explains, “Each computed metric gives a mark in its own range while criteria and factors give a mark between 0 and 3”. Then “transforming raw marks into global marks in a given interval occurs in a new level between criteria and metrics

called practices”. Practices are an important part of the model as it transforms low level metrics into high-level marks reflecting software quality (Mordal-Manet *et al.*, 2011).

By including the practices section, the Squal Model overcomes the issues of the ISO 9126 model of providing an overall quality assesment without retaining the low-level metrics on which the data was based. Its bottom-up approach ensures that the high-level quality results are continuously based on concrete, repeatable measures (Mordal-Manet *et al.*, 2011).

3.3.6 Evaluation of Metrics through Java Developers

De Silva *et al.* (2012) undertook an empirical study of three code complexity metrics; McCabe’s Cyclomatic Complexity, Halstead’s software science and Shao and Wang’s cognitive functional style, in order to determine which was most suitable in the real world. On the matter of metric evaluation, they cite Weyuker’s nine properties and Briand *et al.* five properties as the most commonly used. However, he argues that determining a complexity metric using a theoretical properties is not reliable and thus conducted an empirical study using thirty developers on ten open source java programs.

De Silva *et al.* (2012) analysed the ten java programs and manually calculated the metrics for each followed by having thirty programmers rank the programs based on their own judgement as to the complexity of them. Overall, they determined that Shao and Wang’s cognitive functional style as the most suitable to be used in practice. In addition, they concluded that effective lines of code and experts ranking had a high correlation while Halstead’s two formulae, actual length and estimated length also had a high correlation.

3.3.7 Using Code Metrics to Automate Reviews

Balachandran (2013) argued that using a Review Bot, a tool that integrated automatic static analysis into the code review process. It was found that developers agreed to fix 93% of all automated comments generated by the review bot tool. This in-turn reduced the amount of manual time required to review code. In addition, the Review Bot also

made recommendations in the assignment of the reviews based on file change history of source code.

3.3.7 Applying Cyclomatic Complexity to Y2K

McCabe (1996) proposed an approach to the Year 2000 date issue, where the year field was truncated to two fields and hence when moving from 1999 to 2000 would in fact set the year to 00, using an extended version of the original Cyclomatic Complexity measure to deal specifically with data (McCabe, 1996).

By specifying a set of data elements such as *“a single element”* or *“all elements of a particular data type, or all global elements”*, this data-complexity metric is *“calculated by first removing all control constructs that do not interact with the referenced data elements in the specified set and then computing the Cyclomatic Complexity”*. By specifying all the global data elements, such as date, gives *“an external coupling measure that determines encapsulation”*. This could in turn be used to quantify that the Year 2000 upgrade effort required (McCabe, 1996).

3.3.8 Standardisation of Metrics

Ordonez and Haddad (2008) argues that although metrics are widely recognised they are yet to be standardised within the software industry. His paper looked at some of the existing software metrics in addition to documenting experiences from companies in the industry including Hewlett-Packard (HP), Motorola, NASA and Boeing. They cite an article by William T. Ward who described Hewlett-Packard's (HP's) *“10 x software quality improvement”* initiative. Taking data from a *“software metrics database and an industry profit-loss model to develop a method to compute the actual cost of a software defects”* (Ordonez & Haddad, 2008).

The Software Quality Engineering Group estimates the turnaround time on fixing defects to be approximately 20 hours. Taking this as a starting point and by applying it to a product that had approximately 110 defects found and fixed during the testing process leads to a Figure of around 2200 hours of engineering time or \$165,000 (taking a rate of \$75 per engineering hour) giving a cost of approximately \$1500 per defect. In addition, Ordonez and Haddad (2008) note that these costs are purely

calculating profit and loss based on engineering time they do not take into account the loss of sales to a company for a product being late to market or any contractual costs that may be incurred as a result of project delays due to defective software.

Quoting Tom DeMarco “*You can’t control what you can’t measure*”, Ordonez and Haddad (2008) point out that all other engineering disciplines use quantitative measurements to gain better control over projects and quality within those projects. Although they outline a variety of metrics used within the development of software including cost and effort estimation, productivity measures, data collection, quality assessment, reliability models, process metrics, project metrics and product metrics the focus for this paper will be on project metrics. They argue that these project metrics that include, lines of code, Cyclomatic Complexity and code coverage during test execution, can lead to high quality products.

They also examined the use of metrics in detail at Hewlett-Packard (HP), Motorola, NASA and Boeing. Of notable interest is NASA’s application of design and code reliability metrics where NASA’s Software Assurance Technology Centre (SATC) source code analyser to identify error prone modules “*based on source code complexity, size, and modularity*”. From the various incarnations of complexity, SATC used Cyclomatic Complexity, the number of independent paths, and “*found that by combining size and complexity makes the most effective evaluation*”. They noted, “*large modules with high complexity tend to have the lowest reliability*”. In addition, they listed out the metrics used by the NASA for object-oriented quality analysis as: Weighted Methods per Class (WMC), Response For a Class (RFC), Coupling Between Objects (CBO), Depth In Tree (DIT) and Number Of Children (NOC). They overall found that metrics used early in the development of software did prevent defects later in the project and this in turn decreased overall development costs.

3.4 Clean Code

van Emden and Moonen (2002) introduced the concept of code smells to the area of code metrics in an attempt to automate the process of identifying bad practices developed by Martin Fowler in his book *Refactoring: Improving the Design of*

Existing Code. Building upon this, this paper will introduce a selection of modern thinking in the context of software development as outlined in his book, *Clean Code*, by Robert C. Martin (2008). It introduces several key concepts in an attempt to bring together some of the most important aspects to be considered when developing software. The phases most often used are “craft” and “clean” and are considered as key concepts by Martin. The concepts presented here will include topics such as meaningful names, writing functions and the commenting code.

3.4.1 Meaningful Names

Martin (2008) argues, “*choosing good names takes times but saves more than it takes*“. If a developer is writing the name of the variable and leaves a comment beside it, then that name does not reveal the true intent of that variable. An example is shown in the code snippet below.

```
int d; // elapsed time in days
```

Figure 3.11: Definition of integer d

Defining an int with the name d tells the reader of this code nothing. It provides no context for which the variable exists. If, as the comment suggests, stores the number of elapsed time in days then there are many other names that would provide actual meaning the variable (Martin, 2008).

```
int elapsedTimeInDays;  
int daysSinceCreation;  
int daysSinceModification;  
int fileAgeInDays;
```

Figure 3.12: Suggested names for integer d

Martin (2008) argues that it is not the simplicity of the code that comes into play, it is the implicitly i.e. the degree to which the context is not explicit in the code itself.

To reinforce this point, Martin provides an example.

```

public List<int[]> getThem() {
    List<int[]> list1 = new ArrayList<int[]>();
    for (int[] x : theList)
        if (x[0] == 4)
            list1.add(x);
    return list1;
}

```

Figure 3.13: Example of code using poor naming convention

The code provided above is actually a snippet from the board game, minesweeper. Taking this into context and by simply renaming of variables, the functionality of the above piece of code becomes much more obvious.

```

public List<int[]> getFlaggedCells() {
    List<int[]> flaggedCells = new ArrayList<int[]>();
    for (int[] cell : gameBoard)
        if (cell[STATUS_VALUE] == FLAGGED)
            flaggedCells.add(cell);
    return flaggedCells;
}

```

Figure 3.14: Example of code with refactored variable names

With newly renamed variables, it quickly becomes clear what the code is doing and how it fits into an overall context, all with changing how the code is written.

3.4.2 Functions

Martin argues that functions should be small with no more than two to four lines of code, do only one thing and only have one level of abstraction. This makes the function easy to read for anyone looking to understand the code.

3.4.3 The Stepdown Rule

By combining these short functions with the Stepdown Rule, Martin argues that code should be written using top-down approach, which allows for reading the code from top to bottom, “*descending one level of abstraction at a time*”.

3.4.4 Commenting Code

Martin argues for a slightly more nuanced version of writing comments in code than one would normally associate with the approach of the more comments in code the better. *“Nothing can be quite so helpful as a well-placed comment. Nothing can clutter up a module more than frivolous dogmatic comments. Nothing can be quite so damaging as an old crufty comment that propagates lies and misinformation.”* (Martin, 2008)

Martin’s main points against writing comments in code is twofold; first, comments that are not kept up to date when code is updated are misleading and lead the reader astray and secondly they are normally used to compensate for a developer not being able to write his/her code clearly enough.

3.5 Key Findings

Although it was defined in an era when coding was predominately of the procedural paradigm, McCabe’s Cyclomatic Complexity has become a cornerstone on which many other code metric theories are based.

Chidamber and Kemerer’s metric suite laid the foundation of metrics at a time when the object-oriented paradigm was surging in popularity among programmers.

By introducing ‘code smells’ to code metrics van Emden and Moonen (2002) brought a new viewpoint to how code metrics could be used in modern programming. No longer was it simply a way of measuring for complexity or areas at high risk of being defective but also brought the concept of enforcing coding principals early in the development of software.

The Law of Demeter, a language independent rule, that allows for developers to ensure that basic concepts of modularity and encapsulation are applied in the development of object-oriented code.

The section on clean code introduced the idea of viewing code metrics through a new lens. It challenges well-established practices around code comments and argues for strict naming conventions in order to allow second readers of code readily follow the logic of the code.

3.6 Conclusion

This chapter began by looking at some less well-known code metrics including service-oriented design that attempted to define design-level metrics and program slicing, a similarity-based functional cohesion metric. Although these metrics introduced some new areas there was little or no evidence that additional research grew from them nor any attempts made to apply them within industry.

Other notable points touched on were the growing number of tools within the area, sometimes referred to as static code analysers, while Steidl *et al.* (2013) argued that comments in code should be weighted i.e. code with high number of comments should be considered of more value. The first section concluded with a look at how code metrics can be validated and ultimately highlighted how diverse opinions are on this matter.

The next section examined the application of code metrics within industry and looked at various attempts to use them within established organisations. Companies from Hewlett-Packard (HP) to Siemens and Air France-KLM were among those that implemented metrics of sort in an attempt to identify area at high risk of being defective. While there were indications of success it could be argued that it was patchy and often the code metric analysers added layers of complexity.

The chapter concluded by introducing a subset Martin's (2008) 'clean code' concepts. This looked at principles that should adhered to when writing code and included theories on making variable and method names meaningful, writing small and concise functions, using the Stepdown Rule approach to laying out classes and arguing against having large numbers of comments within the code. This last point regarding comments was in direct contrast to what had been encountered earlier in the chapter

where Steidl *et al.* (2013) argued that the larger the number of comments in the code, the better, whereas Martin (2008) argues that comments can be misleading as many times they are not updated when the code is changed and are normally used to compensate for poorly written code.

The next chapter look to gain new insights into code metrics by creating a new dataset and exploring it in detail. In order to accomplish this, a popular open source project will need to be identified along with tools that can be used to extract the data. Once the data is extracted, then data visualisation will be created in order to explore the data in greater depth.

4. Data Exploration

4.1 Introduction

This chapter will discuss the selection of an open source project from which code metrics can be generated and explored in detail. By opting for a popular open source project that is used within industry and extracting raw data from the research will be provided with metrics that are true reflection of the constant trade-offs that are made when programming in the real world. The newly generated data set will include various different metrics that will allow this research to explore for possible relationships between them while at times taking code snippets for additional analysis in search of common code patterns that may occur.

It begins with an overview of the open source project, followed by a detailed description of the various tools required to extract the code metrics data and in turn presents data visualisations. These visuals can in turn be used to identify possible relationships between the various metrics. It will achieve this by first looking at metrics for the solution at a high-level and then select projects with high, low and average metrics to see if any relationships hold true in each category.

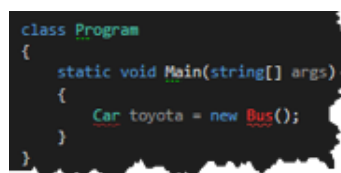
Each of these will in-turn follow the same process of analysis, by creating scatter-plots that compare each of the metrics side-by-side in an effort to identify where relationships exist and hold true regardless of the metrics for the given project. At various points, code examples will be included in an effort to identify causation of what the scatter plot correlations are indicating.

The chapter will conclude by looking at any findings that may provide further insight into the relationships that exist between the various metrics.

4.2 Roslyn Overview

With modern computers, the compiler used to compile human readable code into low-level code that can be interpreted by a machine, is done at an incredibly fast rate and is not given a lot of thought by developers. The details of how compilers work is beyond the scope of this research but it suffices to say that human readable code, i.e. code that programmers write, is not consumable by a machine and therefore specialist code is written that converts this human readable code into machine readable code. This process is known as compiling the code (Aho Alfred, Ravi, & Ullman Jeffrey, 1986). Over time tools for writing code, known as integrated development environments (IDE) have grown in popularity. They allow programmers to write code with built-in features that warn them early in the process if the code is incorrect. For example, the IDE shown in Figure 4.1 is indicating that there is an issue with the code. In this example, the IDE cannot find an existing class `Bus` and therefore has highlighted it as an issue before the programmer has even attempted to compile the code.

For these features to be built into IDE's the IDE must be able to access the underlying solution that is compiling the code. One of these solutions, used within the .NET ecosystem, is called Roslyn.



```
class Program
{
    static void Main(string[] args)
    {
        Car toyota = new Bus();
    }
}
```

Figure 4.1: Visual Studio IDE highlights `Bus` in red as it has detected an error

Roslyn is one of many projects provided by the .NET Foundation, an independent organisation that aims to open development around the .NET ecosystem. The open-sourced code with over 13,000 commits and 154 contributors consists of over 100 projects and thousands of classes, making it the perfect candidate on which to analyse code metrics.

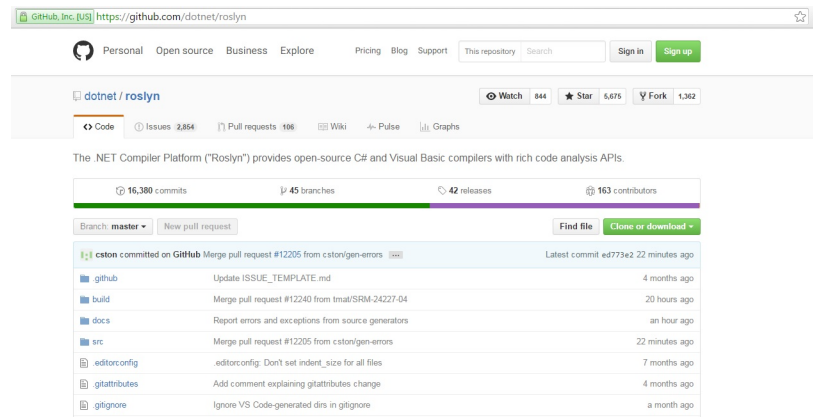


Figure 4.2: Roslyn open source solution on github.com

The Roslyn project opens up all the details of how the .NET compiler is implemented and provides API's that developers can harness in order to gain access to the information contained with the compiler. This in turn allows IDE's such as visual studio to call these API's as the programmer is writing the code and provide instance feedback on various aspects of what is being written. As shown in Figure 4.1 above, the visual studio IDE has highlighted the word `BUS` in red. It has detected an error before the programmer has even compiled the code.

4.3. Metrics Tools

This section will look at the various tools required to access, build and generate code metrics for the Roslyn solution as well as those used to generate data visualisations in an effort to do further exploratory work on the data.

In order to generate a new data set of metrics from the Roslyn open source solution, various tools were evaluated in order to determine their suitability. By combining different commercially available tools to extract the required data from the solution, it reduced the need for new tools to be developed. In addition to the tools providing the required capabilities to perform their function, it was also important that licenses were available at no cost for a period of months.

This section will provide an overview of these tools and evaluate their suitability.

4.3.1 NDepend

NDepend is a static analysis tool for .NET code, which enables the user to generate data relating to code metrics for a project. Developed by Patrick Smacchia circa 2004, it covers many well-known code metrics including Cyclomatic Complexity, class coupling and lines of code. Overall NDepend was not chosen although it could generate code metrics, it in effect generated too much data with no obvious way for it to be broken down, for example, exported to an Excel document for further analysis. This is possibly related to the software being proprietary with a focus on it being an all-encompassing analysis tool, which in turn makes it less favourable to be used as part of this research. NDepend does not appear to have any student licenses available and hence is only available for free for an initial 30-day evaluation period.

4.3.2 Visual Studio Code Analysis

Visual Studio is the integrated development environment for the .NET platform. Developed by Microsoft circa 1997 with new releases on an average of every two years, Visual Studio has several versions available, most of which require a commercial license with the exception of the community edition, which is free. Visual Studio as a product has a wide range of functionality, including the ability to write code in various languages, including C#, F# and visual basic, provides compilers to build code with enterprise editions providing functionality around load testing and other advanced features. For the purpose of this research Visual Studio will be mainly used to build existing open source projects with the aim of extracting code metrics for further analysis.

In addition, Visual Studio comes with the ability to generate code metrics including Cyclomatic Complexity, class coupling, lines of code and depth of inheritance that enables the code metrics of the project to be analysed. In contrast to NDepend, Visual Studio also provides functionality that allows all of the metric data generated to be exported out into an Excel document and therefore allows for further analysis into the data beyond what the initial tools provided for. This makes the Visual Studio code metric data more suited for research, as the goal is to take the data generated by existing tools and further develop this with additional analysis and insights.

4.3.3 ReSharper

Developed for commercial purposes by JetBrains, ReSharper is a plugin that adds additional functionality for developers using Visual Studio. First launched circa 2000, it was the basis for JetBrains to go on and develop many fully-fledged IDE for the most popular programming languages including PHP, Java and Python. Its features include the ability to find types and code snippets more easily, renaming of classes and other types. The main functionality used for this research is in its ability to allow code in a Visual Studio project to be refactored more easily with functionality such as extracting new methods from existing code, examples of which will be detailed later. It should be noted that JetBrains provide a yearlong student license on all of their products.

4.3.4 Tableau

Tableau is commercial software that specialises in data visualisations. Developed by the Tableau software company based in Seattle, it allows Excel formatted data to be imported providing basis for data exploration by enabling the end-user to create various data visuals to further explore the data. By exporting metrics from Visual Studio and viewing it from various angles in Tableau allows for a deeper analysis of what was initially produced by Visual Studio. By combining existing functionality of readily available tools and further exploring that same data with Tableau allows for more in-depth data exploration of the data. It should be noted that Tableau also provides student licenses.

4.4 Roslyn Metrics

This section will explore the data generated from the Roslyn project using the Visual Studio metrics analysis tools.



Figure 4.3: The process of generating metric data from Roslyn

The following steps were required in generating the data:

1. Building the Roslyn project

Open source projects normally provide build scripts that allow projects to be built i.e. compiled into a lower-level machine code as not all large projects can be built from within Visual Studio given the size of the project. This is generally due to required dependencies being built in a particular order prior to the main project itself being built. In the case of Roslyn, once the project was initially built from the command line, it was then possible to rebuild a subset of the components from within Visual Studio. The binaries generated by the project were then outputted to a binaries folder. This made Roslyn quite friendly to work with.

2. Extracting the raw metrics data

Visual Studio provides an analysis option for both the solution and on a per project basis. For this particular solution the analysis failed to generate on several attempts and therefore the data was collected on a per project basis. It should be noted that although this took slightly longer to complete, it had no impact on the resulting data and therefore time was not invested in identifying the issue(s) that caused the problem with generating the data a solution level. Once the metrics data was generated, it could be easily exported into an Excel document. It should be noted that Visual Studio did not provide any functionality to alter the data in any way or allow for a subset of the data to be exported. It only allowed for all generated data to be exported. In addition, it was not possible to export the data without using Excel (a product developed by Microsoft as part of there office package) i.e. the Open Office equivalent would not suffice.

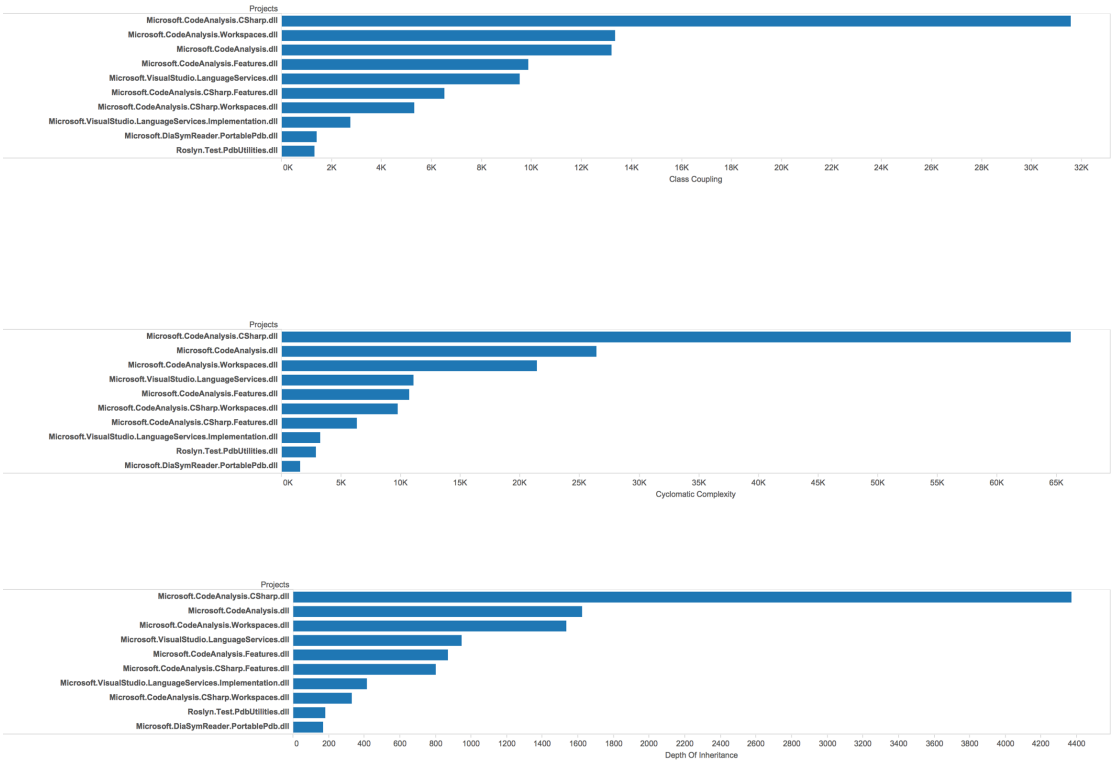
3. Data exploration using data visuals

Once the data was saved to an Excel file, it could then be easily imported into Tableau. From here visuals were created in an effort to explore the data in depth starting with some high-level flipped bar charts and then scatter plots in attempt to identify possible correlations between metrics.

As a starting point to analyse the code, the metrics were taken at a project level, each of which contained anywhere from a few hundred to a few thousand classes. This section will take a look at various areas of the code and analyse each for four metrics, namely: class coupling, Cyclomatic Complexity, depth of inheritance, and lines of code. Each of these metrics were first examined at project level to give overall comparisons and then selected projects with various criteria, for example showing extremely high or low metrics, were selected for further analysis at the class level.

4.5 High Scoring Metrics

The top ten projects for each of the four metrics, class coupling, Cyclomatic Complexity, depth of inheritance and lines of code, are shown as flipped bar charts below. A preliminary analysis of the metrics indicated the CSharpCodeAnalysis project as having the highest for all four metrics.



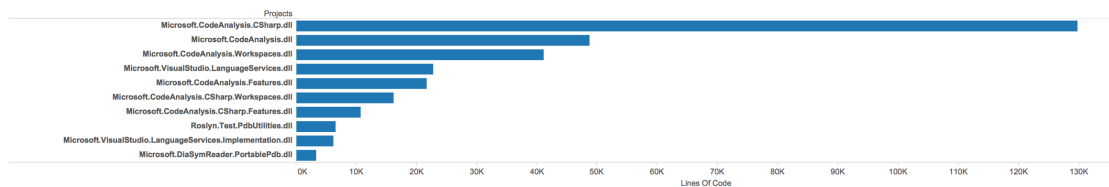
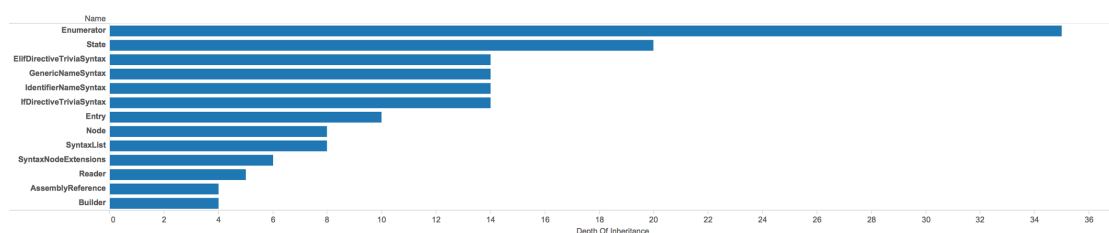
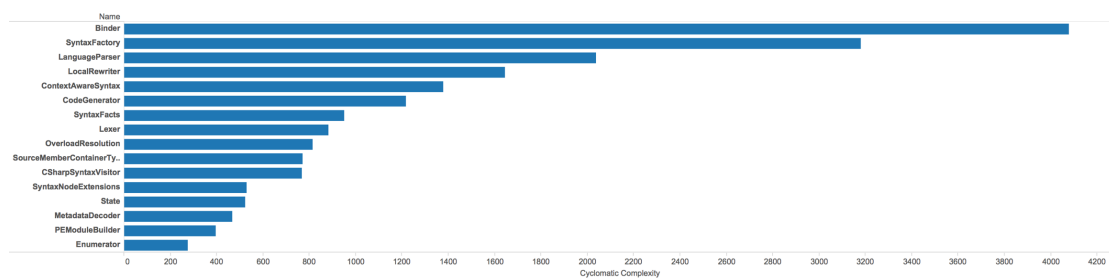
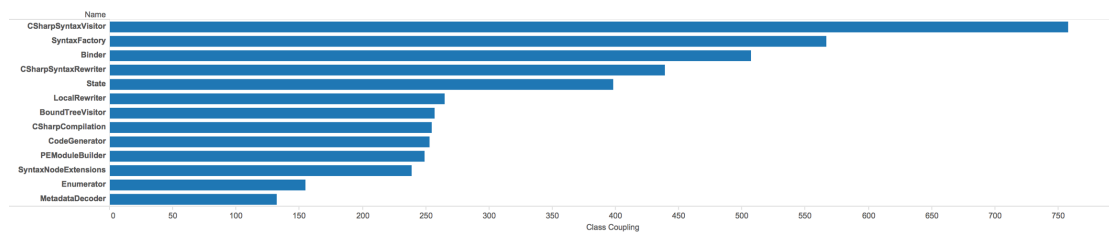


Figure 4.4: Identifying Projects with High Scoring Metrics

4.5.1 CSharpCodeAnalysis Project

Taking a closer look at this project reveals an array of the classes that contribute to this particular project having the highest metrics in the overall code base. Shown below is metrics calculated on a per type basis within the CSharpCodeAnalysis project. Unlike the previous flipped bar charts shown, the metrics for projects all indicate that this project had the highest metric count for all four measured, this collection of visuals show the different types providing the highest metric counts and although the Binder class is top in two of the four metrics, SyntaxFactory is second in two of them.



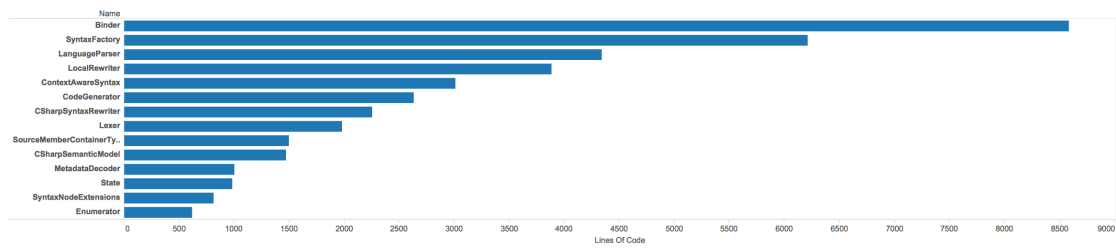


Figure 4.5: Identifying Types with High Scoring Metrics

Taking this as a starting point, the metrics were then used to generate scatter plot visuals pitting the four metrics against each other. Similar to the creation of the flipped-bar charts Tableau allows for the selection of columns versus rows to generate a scatter plot. In addition, a filter was applied to colour code the types that were forming the trend line in the graph. As there were four metrics in play, six scatter plots, complete with trend lines were created that attempt to provide further insight into the data by analysing the various correlations between the metrics. Each of these visuals will now be examined in turn.

4.5.2 Cyclomatic Complexity and Class Coupling

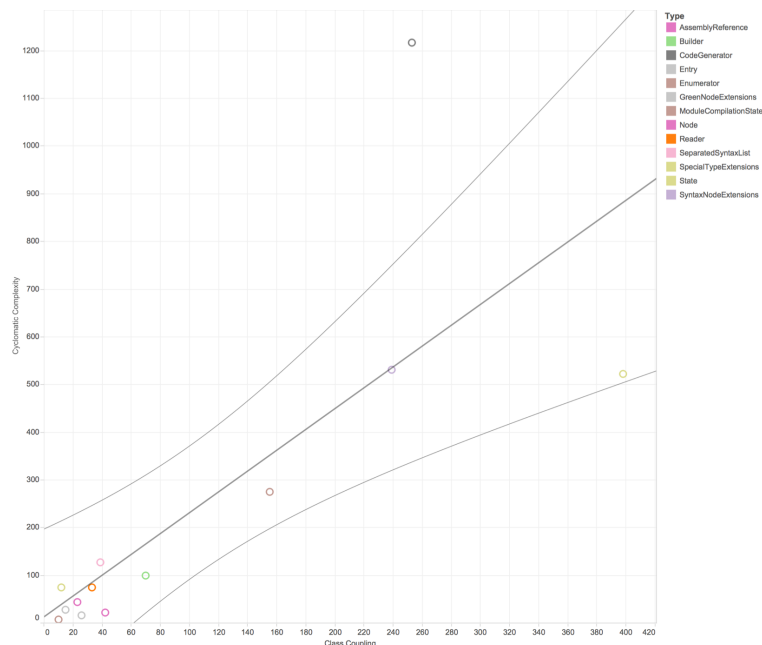


Figure 4.6: Cyclomatic Complexity and Class Coupling

The first of the scatter plots created, show a relatively close relationship between Cyclomatic Complexity and class coupling although it could be argued that the relationship is somewhat stronger within the lower counts of the metrics. Logically this would make sense as the more classes coupled together would likely lead to an increase the amount of logic contained within that class and therefore an increase in Cyclomatic Complexity.

4.5.3 Depth of Inheritance and Class Coupling

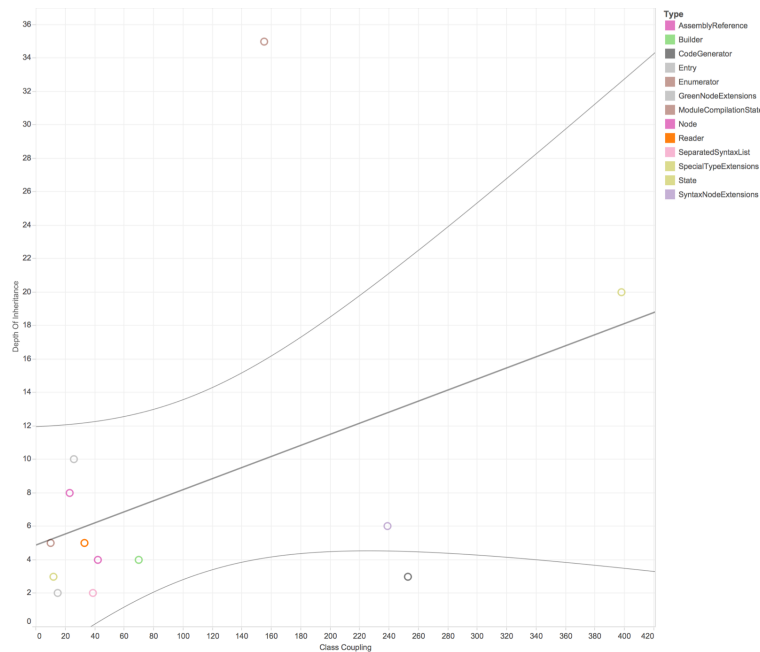


Figure 4.7: Depth of Inheritance and Class Coupling

Not much of a relationship was found during comparisons of depth of inheritance with class coupling. This is not surprising as both class coupling and inheritance have a similar goal in adding functionality to a class either by referencing another class in the case of class coupling or by inheriting behaviour from the class above with inheritance and therefore a class is likely to do only one these and not both.

4.5.4 Lines of Code and Class Coupling

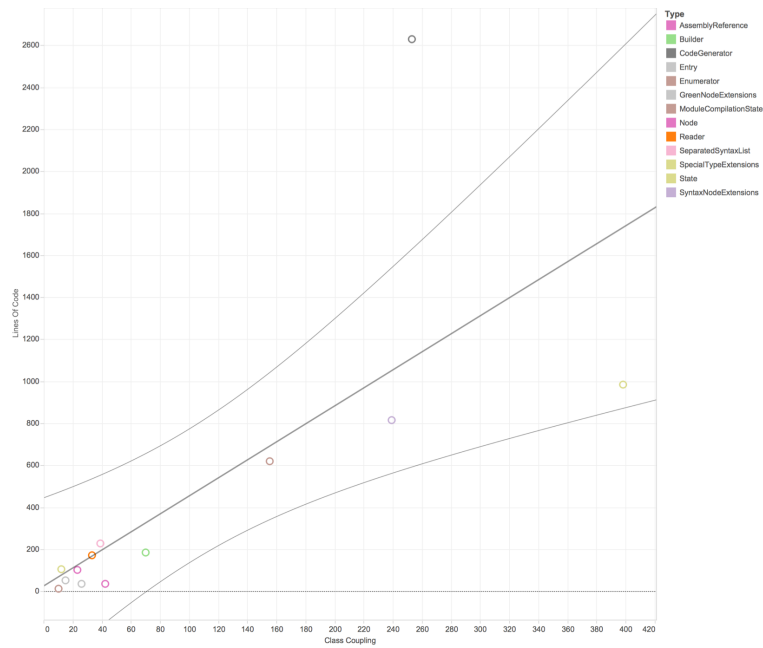


Figure 4.8: Lines of Code and Class Coupling

While lines of code and class coupling show some correlation it was strongest when both metric scores are low. Although this is interesting from a data exploration point of view it is unlikely that there is much causation associated here as logically a class containing hundreds of lines of code would not necessarily be coupled to many other classes and vice versa i.e. a class coupled to many other classes would not necessarily have to have many lines of code.

4.5.5 Cyclomatic Complexity and Depth of Inheritance

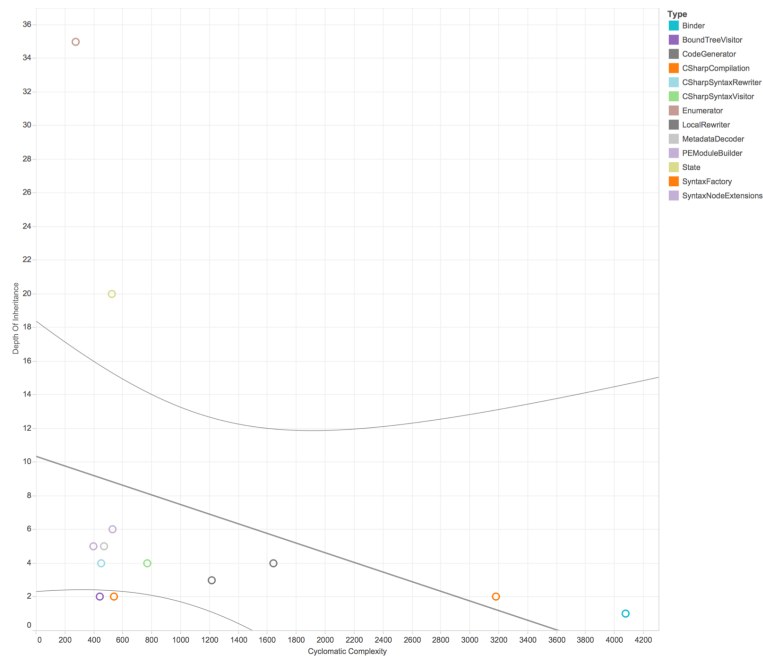


Figure 4.9: Cyclomatic Complexity and Depth of Inheritance

Interestingly, Cyclomatic Complexity and depth of inheritance lack any sort of correlation. It would appear that as more classes are built into an inheritance hierarchy, the Cyclomatic Complexity dissipates.

4.5.6 Cyclomatic Complexity and Lines of Code

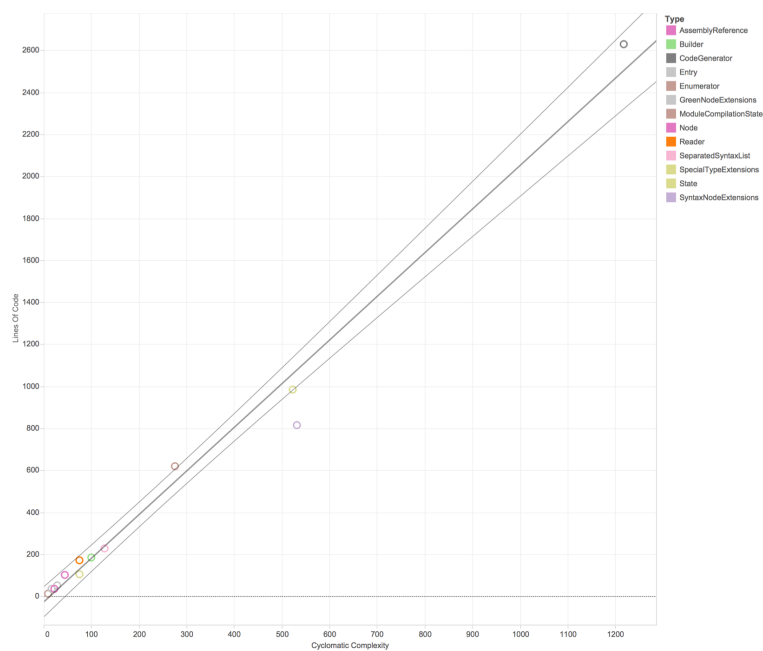


Figure 4.10: Cyclomatic Complexity and Lines of Code

The correlation between Cyclomatic Complexity and lines of code was by far the closest of all the compared metrics. The correlation holds on both the low end of the metrics scores right up to a class that has over two thousand lines of code, having a Cyclomatic Complexity score of above twelve hundred.

4.5.7 Depth of Inheritance and Lines of Code

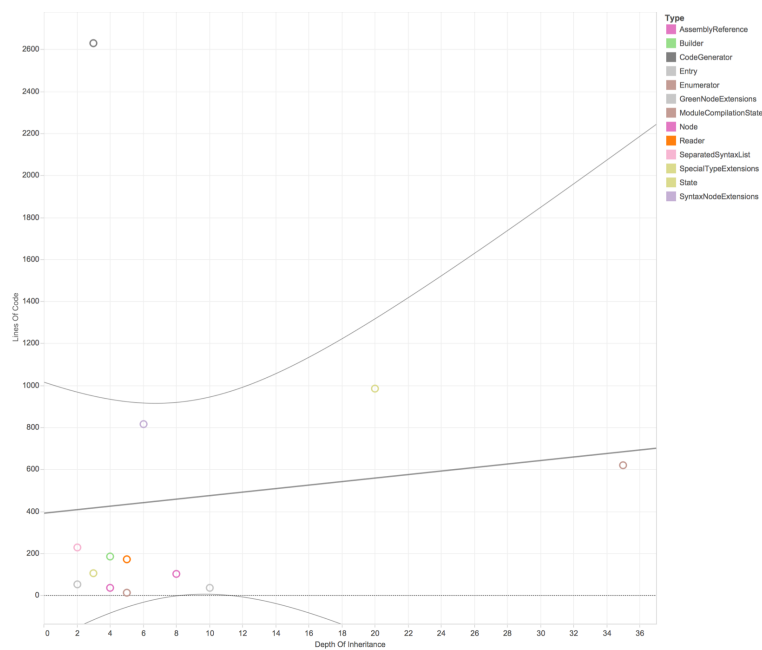


Figure 4.11: Depth of Inheritance and Lines of Code

Lastly lines of code and depth of inheritance showed little to no correlation.

Overall this section provides an initial overview of the various metric data gathered from the Roslyn project. While some metrics have shown extremely strong correlation, for example Cyclomatic Complexity and lines of code, others, namely depth of inheritance and lines of code have shown little to no correlation.

4.5.8 Examination of code

In an effort to dig a little further into the data in an attempt to identify any possible causation for the above correlations, code samples from the project were examined for the various classes that appeared in the visuals above.

CodeGenerator class

```
// Returns (Synthesized), 118 days ago (0 authors, 1 change)
private LocalDefinition LazyReturnTemp
{
    get
    {
        var result = _returnTemp;
        if (result == null)
        {
            Debug.Assert(!_method.ReturnsVoid, "returning something from void method?");

            var bodySyntax = _methodBody.SyntaxOpt;
            if (_ifmitStyle == IfmitStyle.Debug && bodySyntax != null)
            {
                int syntaxOffset = _method.CalculateLocalSyntaxOffset(bodySyntax.SpanStart, bodySyntax.SyntaxTree);
                var localSymbol = new SynthesizedLocal(_method, _method.ReturnType, SynthesizedLocalKind.FunctionReturnValue, bodySyntax);
                result = _builder.LocalSlotManager.DeclareLocal(
                    type: _module.Translate(localSymbol.Type, bodySyntax, _diagnostics),
                    symbol: localSymbol,
                    name: null,
                    kind: localSymbol.SynthesizedKind,
                    id: new LocalDebugId(syntaxOffset, ordinal: 0),
                    pdbAttributes: localSymbol.SynthesizedKind.PdbAttributes(),
                    constraints: localSlotConstraints.None,
                    isDynamic: false,
                    dynamicTransformFlags: ImmutableArray<TypedConstant>.Empty,
                    isSlotReusable: false);
            }
            else
            {
                result = AllocateTemp(_method.ReturnType, _boundBody.Syntax);
            }
            _returnTemp = result;
        }
        return result;
    }
}
```

Figure 4.12: CodeGenerator Class

This first code example shows one of the private methods within the CodeGenerator class. It makes for an interesting example of code that has the potential to have high Cyclomatic Complexity as it has a structure of nested if-statements with an else on the closing if. This is exactly the type of structure that can cause a high Cyclomatic Complexity, as there are a lot of paths the code can take during execution.

GreenNodeExtensions class

```

namespace Microsoft.CodeAnalysis.CSharp.Syntax
{
    0 references | RoslynTeam, 508 days ago | 1 author, 1 change
    internal static class GreenNodeExtensions
    {
        48 references | RoslynTeam, 508 days ago | 1 author, 1 change
        internal static Syntax.InternalSyntax.SyntaxList<T> ToGreenList<T>(this SyntaxNode node) where T : Syntax.InternalSyntax.CSharpSyntaxNode
        {
            return node != null ?
                ToGreenList<T>(node.Green) :
                default(Syntax.InternalSyntax.SyntaxList<T>);
        }

        21 references | RoslynTeam, 508 days ago | 1 author, 1 change
        internal static Syntax.InternalSyntax.SeparatedSyntaxList<T> ToGreenSeparatedList<T>(this SyntaxNode node) where T : Syntax.InternalSyntax.CSharpSyntaxNode
        {
            return node != null ?
                new Syntax.InternalSyntax.SeparatedSyntaxList<T>(ToGreenList<T>(node.Green)) :
                default(Syntax.InternalSyntax.SeparatedSyntaxList<T>);
        }

        28 references | RoslynTeam, 508 days ago | 1 author, 1 change
        internal static Syntax.InternalSyntax.SyntaxList<T> ToGreenList<T>(this GreenNode node) where T : Syntax.InternalSyntax.CSharpSyntaxNode
        {
            return new Syntax.InternalSyntax.SyntaxList<T>((Syntax.InternalSyntax.CSharpSyntaxNode)node);
        }
    }
}

```

Figure 4.13: GreenNodeExtensions Class

The `GreenNodeExtensions` class is an interesting example of a class that contains relatively low number of lines of code but quite a high Cyclomatic Complexity. This would indicate that the large number methods relative to class size is an impacting factor on the calculating of the Cyclomatic Complexity. In addition, the inclusion of the ternary operator, a short handed if-statement, in the method `ToGreenList` above will also increase the complexity of the class and may not be immediately obvious on first inspection.

While this section looked in detail at the project that generated the highest code metrics of the Roslyn solution, the following section will look at the project that generated the lowest metrics. Again, each of the metrics was compared using scatter plots in an attempt to identify which of the correlations above held true.

4.6 Low Scoring Metrics

As projects scoring lower metrics tend to have lower numbers of classes and in-turn less code, a project with a relatively low score was selected for analysis (not the actual lowest as it would be redundant exercise to examine a project containing very little code). The flipped bar charts below were created to give an overview of all of the lowest scoring projects in the Roslyn solution.

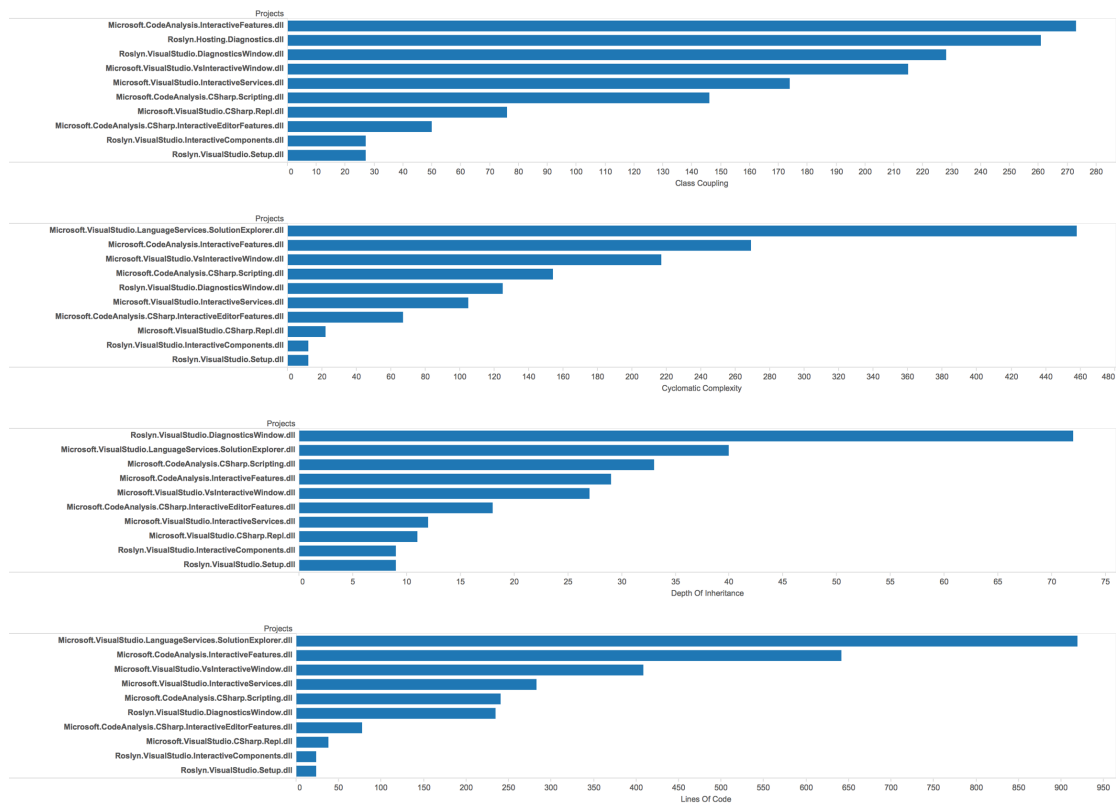


Figure 4.14: Identifying Projects with Low Scoring Metrics

4.6.1 MicrosoftCodeAnalysisCSharpScripting Project

From the flipped bar charts above, the MicrosoftCodeAnalysisCSharpScripting project was selected for a detailed analysis using the same format as CSharpCodeAnalysis seen in the last section. By generating the same visuals, and comparing the metrics against each other using scatter plots, it allows for direct comparisons to be made between projects with high scoring metrics and projects with lower scoring metrics.

4.6.2 Cyclomatic Complexity and Class Coupling

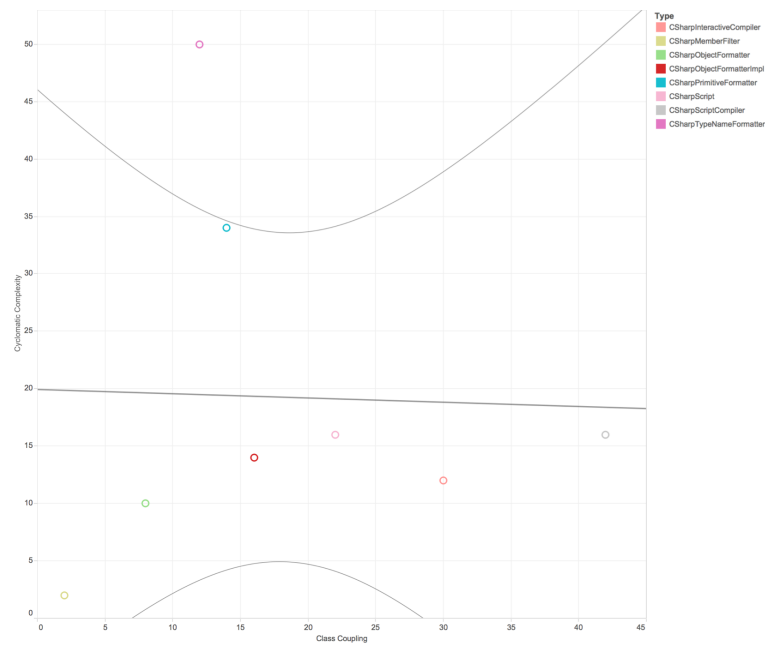


Figure 4.15: Cyclomatic Complexity and Class Coupling

While the previous comparison between Cyclomatic Complexity and class coupling showed a relatively close correlation, it now appears to be non-existent when the overall metric scores are considerably lower.

4.6.3 Depth of Inheritance and Class Coupling

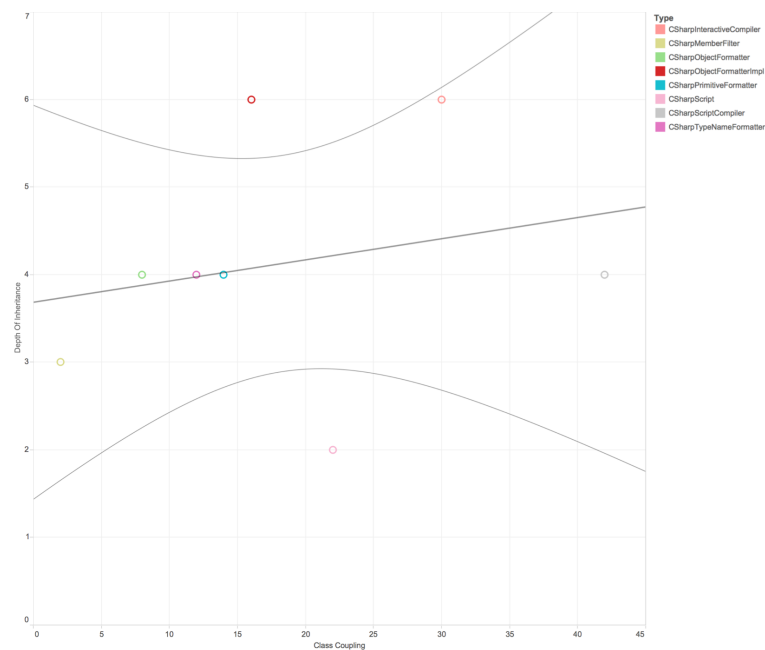


Figure 4.16: Depth of Inheritance and Class Coupling

Not unsurprisingly, for depth of inheritance and class coupling the trend of no significant correlation appears to have continued as no real correlation has appeared when the metrics are considerably lower.

4.6.4 Lines of Code and Class Coupling

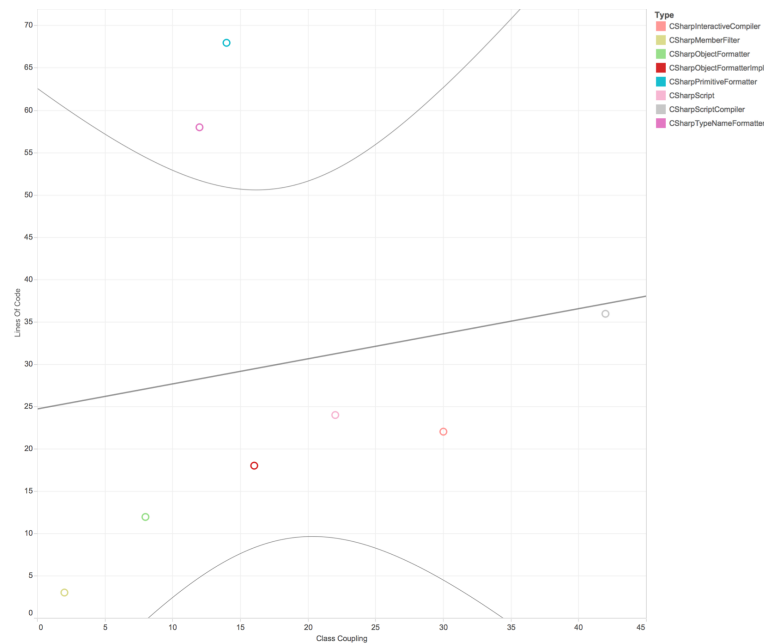


Figure 4.17: Lines of Code and Class Coupling

In contrast to previous analysis where there appeared to be a slight correlation when metrics were lower, this appears to have dissipated and therefore was unlikely to have been of any significance.

4.6.5 Cyclomatic Complexity and Depth of Inheritance

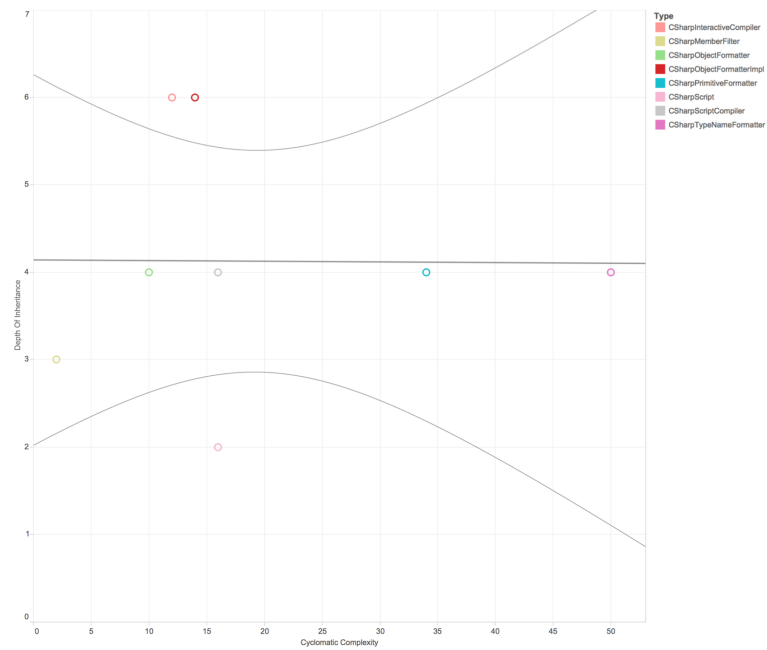


Figure 4.18: Cyclomatic Complexity and Depth of Inheritance

The non-existence of any relationship between Cyclomatic Complexity and depth of inheritance appears to continue regardless of how low or high the metrics are.

4.6.6 Cyclomatic Complexity and Lines of Code

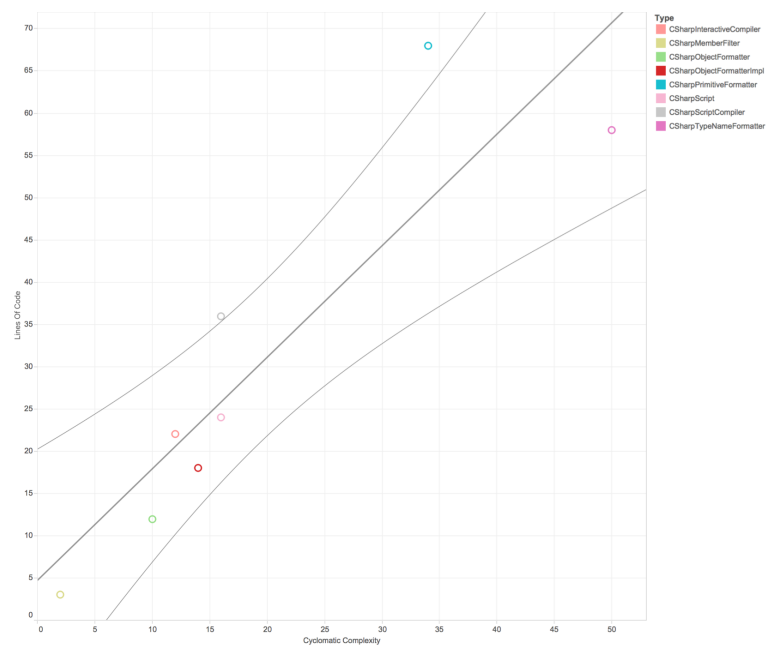


Figure 4.19: Cyclomatic Complexity and Lines of Code

Again the relationship between Cyclomatic Complexity and lines of code appears to be the strongest regardless of how high the metric data indicates.

4.6.7 Depth of Inheritance and Lines of Code

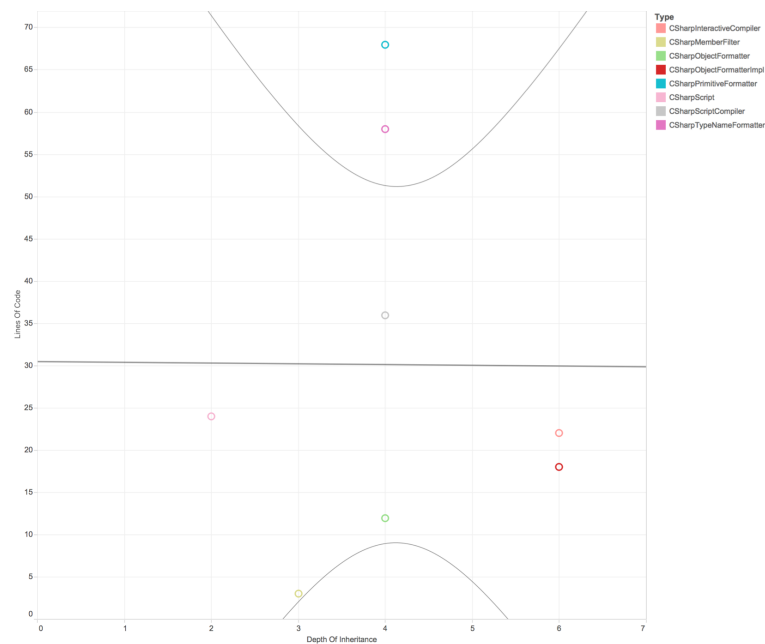


Figure 4.20: Depth of Inheritance and Lines of Code

Continuing the trend from the previous comparison when metrics were a lot higher, there is little to no relationship between depth of inheritance and lines of code.

4.6.8 Examination of code

As with the previous section, code was analysed in an effort to shed further light on the cause for the correlations above.

CSHarpPrimitiveFormatter class

```

1 reference | Andrew Casey, 130 days ago | 1 author, 6 changes
internal class CSharpPrimitiveFormatter : CommonPrimitiveFormatter
{
    3 references | Andrew Casey, 184 days ago | 1 author, 1 change
    protected override string NullLiteral => ObjectDisplay.NullLiteral;

    3 references | Andrew Casey, 184 days ago | 1 author, 1 change
    protected override string FormatLiteral(bool value)
    {
        return ObjectDisplay.FormatLiteral(value);
    }

    4 references | Andrew Casey, 137 days ago | 1 author, 1 change
    protected override string FormatLiteral(string value, bool useQuotes, bool escapeNonPrintable, int numberRadix = NumberRadixDecimal)
    {
        var options = GetObjectDisplayOptions(useQuotes: useQuotes, escapeNonPrintable: escapeNonPrintable, numberRadix: numberRadix);
        return ObjectDisplay.FormatLiteral(value, options);
    }

    4 references | Andrew Casey, 137 days ago | 1 author, 1 change
    protected override string FormatLiteral(char c, bool useQuotes, bool escapeNonPrintable, bool includeCodePoints = false, int numberRadix = NumberRadixDecimal)
    {
        var options = GetObjectDisplayOptions(useQuotes: useQuotes, escapeNonPrintable: escapeNonPrintable, includeCodePoints: includeCodePoints, numberRadix: numberRadix);
        return ObjectDisplay.FormatLiteral(c, options);
    }

    3 references | Andrew Casey, 131 days ago | 1 author, 1 change
    protected override string FormatLiteral(sbyte value, int numberRadix = NumberRadixDecimal, CultureInfo cultureInfo = null)
    {
        return ObjectDisplay.FormatLiteral(value, GetObjectDisplayOptions(numberRadix: numberRadix, cultureInfo: cultureInfo));
    }
}

```

Figure 4.21: Figure 4.21: CSharpPrimitiveFormatter Class

Similar to the second code example shown in the last section, the CSharpPrimitiveFormatter class is a small class with a high Cyclomatic Complexity relative to class size.

CSharpTypeNameFormatter class

```

3 references | Andrew Casey, 130 days ago | 1 author, 4 changes
internal class CSharpTypeNameFormatter : CommonTypeNameFormatter
{
    6 references | Andrew Casey, 184 days ago | 1 author, 1 change
    protected override CommonPrimitiveFormatter PrimitiveFormatter { get; }

    1 reference | Andrew Casey, 184 days ago | 1 author, 1 change
    public CSharpTypeNameFormatter(CommonPrimitiveFormatter primitiveFormatter)
    {
        PrimitiveFormatter = primitiveFormatter;
    }

    4 references | Andrew Casey, 184 days ago | 1 author, 1 change
    protected override string GenericParameterOpening => "<";
    4 references | Andrew Casey, 184 days ago | 1 author, 1 change
    protected override string GenericParameterClosing => ">";
    3 references | Andrew Casey, 184 days ago | 1 author, 1 change
    protected override string ArrayOpening => "[";
    3 references | Andrew Casey, 184 days ago | 1 author, 1 change
    protected override string ArrayClosing => "]";

    3 references | Andrew Casey, 184 days ago | 1 author, 1 change
    protected override string GetPrimitiveTypeName(SpecialType type)
    {
        switch (type)
        {
            case SpecialType.System_Boolean: return "bool";
            case SpecialType.System_Byte: return "byte";
            case SpecialType.System_Char: return "char";
            case SpecialType.System_Decimal: return "decimal";
            case SpecialType.System_Double: return "double";
            case SpecialType.System_Int16: return "short";
            case SpecialType.System_Int32: return "int";
            case SpecialType.System_Int64: return "long";
            case SpecialType.System_SByte: return "sbyte";
            case SpecialType.System_Single: return "float";
            case SpecialType.System_String: return "string";
            case SpecialType.System_UInt16: return "ushort";
            case SpecialType.System_UInt32: return "uint";
            case SpecialType.System_UInt64: return "ulong";
            case SpecialType.System_Object: return "object";

            default:
                return null;
        }
    }
}

```

Figure 4.22: CSharpTypeNameFormatter Class

The method, GetPrimitiveTypeName within the CSharpTypeNameFormatter class contains a switch statement. This type of logic within a class is another example of how the Cyclomatic Complexity of class can

dramatically increase with relatively few lines of code. It increases the number of paths the class has to execute for all possible input values without the class appearing to grow very much.

4.7 Average Scoring Project

The two previous sections looked in depth at the various metrics when compared head to head against each other. This resulted in certain metrics namely Cyclomatic Complexity and lines of code showing strong correlations regardless of whether the metrics are the highest or among the lowest in the overall Roslyn solution. Continuing this data exploration this last section will take an average, mid-level project from the Roslyn solution and again look for correlations through the use of scatter plot data visuals.

Referring back to the initial flipped bar charts showing the projects with the highest metrics, another project was chosen from the lower end of this section named RoslynTestPdbUtilities.

4.7.1 RoslynTestPdbUtilities Project

This section follows the same format as the previous two in an effort to either further establish metrics that have a strong correlation when compared against each other or to discredit previously identified strong correlations.

4.7.2 Cyclomatic Complexity and Class Coupling

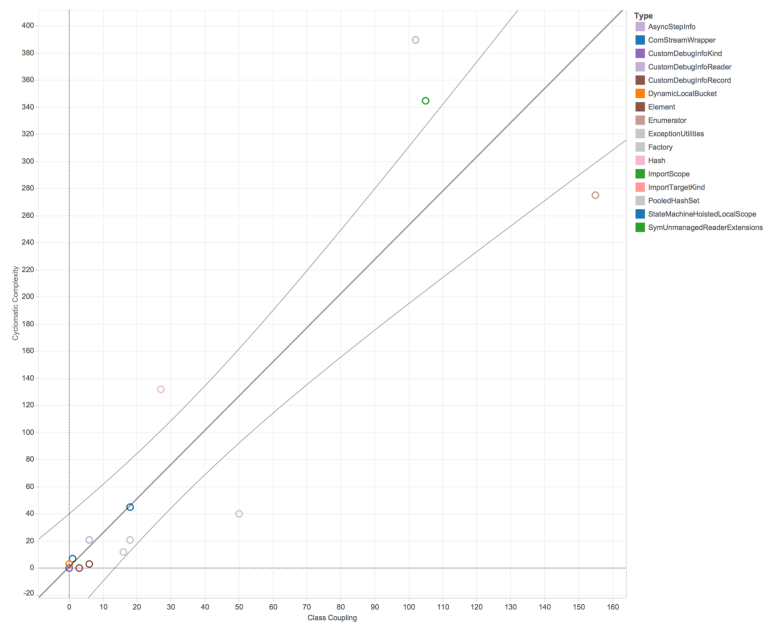


Figure 4.23: Cyclomatic Complexity and Class Coupling

Interestingly when Cyclomatic Complexity and class coupling were compared with high scoring metrics it showed a relatively close correlation whereas that correlation appeared to have fallen away when the metrics were on the opposite end of the scale with extremely low metrics. On examination of the generated scatter plot for a project considered to be in the mid-range of the overall metrics score for the overall Roslyn solution, it appears that this correlation has resurfaced.

4.7.3 Depth of Inheritance and Class Coupling

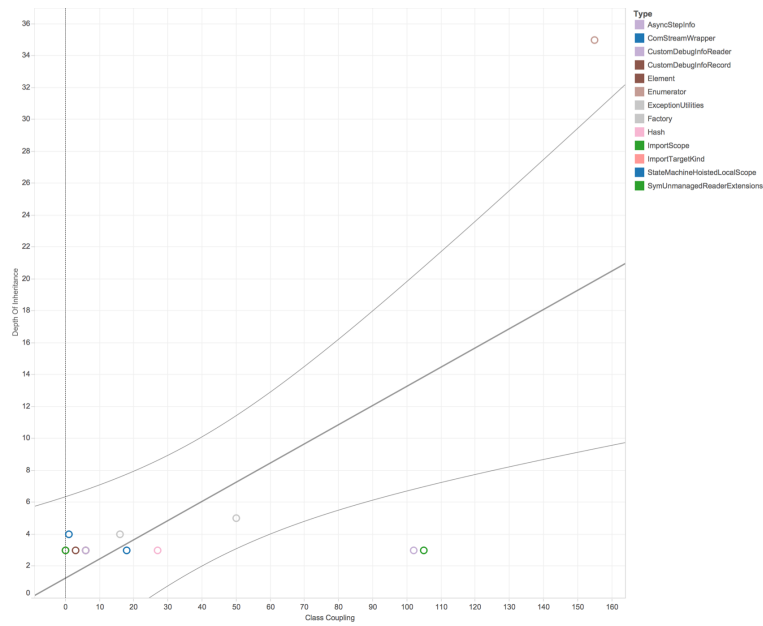


Figure 4.24: Depth of Inheritance and Class Coupling

Previously there was little or no relationship between depth of inheritance and class coupling with this project showing the closest of all three to any sort of relationship between the two metrics.

4.7.4 Lines of Code and Class Coupling

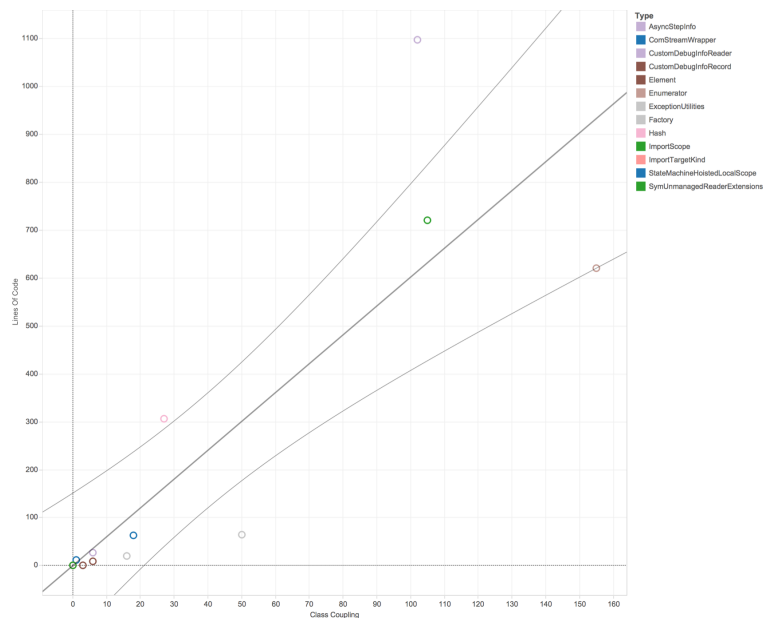


Figure 4.25: Lines of Code and Class Coupling

Having first noted a slight correlation in the high scoring metrics, one that appeared to have fallen away when the metrics scores were significantly lower, seems to have resurfaced when the metrics score moved back up to an average level.

4.7.5 Cyclomatic Complexity and Depth of Inheritance

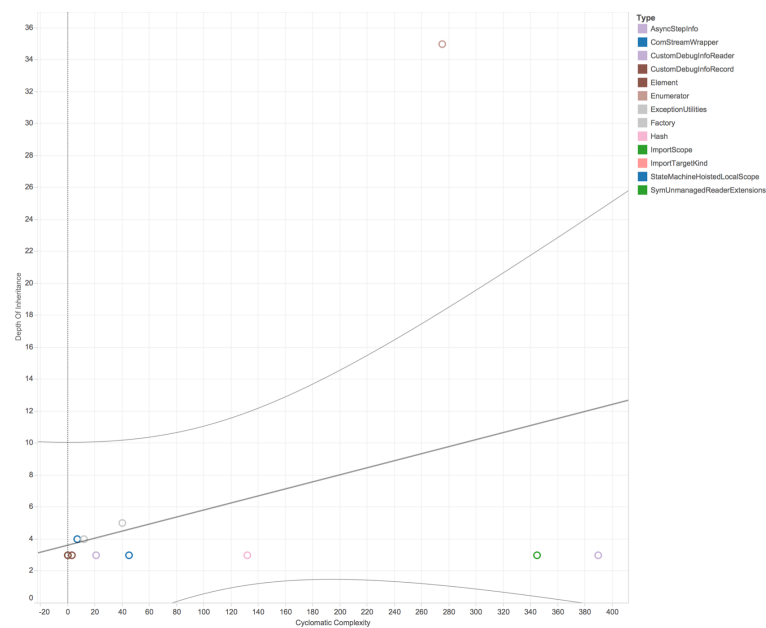


Figure 4.26: Cyclomatic Complexity and Depth of Inheritance

As consistent with the two previous projects it is starting to become well established that no relationship exists between Cyclomatic Complexity and depth of inheritance regardless of the metric score of the overall project.

4.7.6 Cyclomatic Complexity and Lines of Code

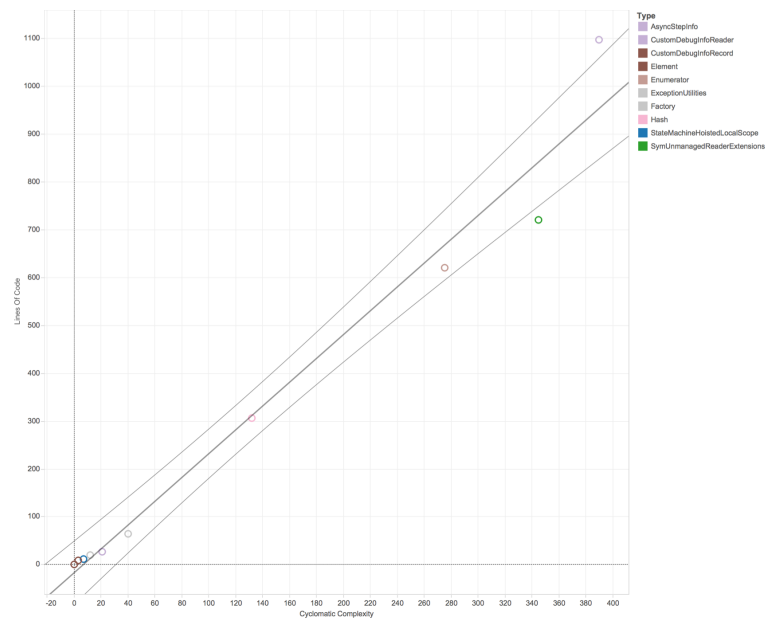


Figure 4.27: Cyclomatic Complexity and Lines of Code

As consistent with the previous two projects, Cyclomatic Complexity and lines of code have the strongest correlation regardless of what the overall metrics scores are for the given project.

4.7.7 Depth of Inheritance and Lines of Code

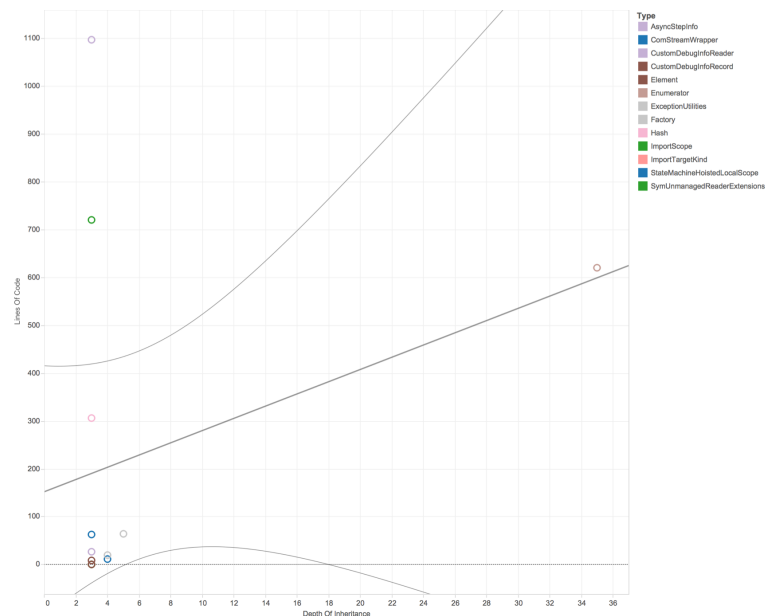


Figure 4.28: Depth of Inheritance and Lines of Code

Consistent with the previous projects depth of inheritance and lines of code have shown no sign of a relationship.

4.7.8 Examination of code

As consistent with the previous sections, code examples were examined to identify any possible causation for the above correlations.

AsyncStepInfo class

```
-- references | Tomas Matousek, 477 days ago | 1 author, 1 change
internal struct AsyncStepInfo : IEquatable<AsyncStepInfo>
{
    public readonly int YieldOffset;
    public readonly int ResumeOffset;
    public readonly int ResumeMethod;

    -- references | Tomas Matousek, 477 days ago | 1 author, 1 change
    public AsyncStepInfo(int yieldOffset, int resumeOffset, int resumeMethod)
    {
        this.YieldOffset = yieldOffset;
        this.ResumeOffset = resumeOffset;
        this.ResumeMethod = resumeMethod;
    }

    -- references | Tomas Matousek, 477 days ago | 1 author, 1 change
    public override bool Equals(object obj)
    {
        return obj is AsyncStepInfo && Equals((AsyncStepInfo)obj);
    }

    -- references | Tomas Matousek, 477 days ago | 1 author, 1 change
    public bool Equals(AsyncStepInfo other)
    {
        return YieldOffset == other.YieldOffset
            && ResumeMethod == other.ResumeMethod
            && ResumeOffset == other.ResumeOffset;
    }

    -- references | Tomas Matousek, 477 days ago | 1 author, 1 change
    public override int GetHashCode()
    {
        return Hash.Combine(YieldOffset, Hash.Combine(ResumeMethod, ResumeOffset));
    }
}
```

Figure 4.29: AsyncStepInfo Class

As consistent with the previous code examples, small methods, irrespective of access modifier, appear to increase the Cyclomatic Complexity of the class overall.

4.8 Key Findings

The data exploration conducted above indicates that while some metric pairings showed no correlation regardless of the overall metric score of that project other metric pairings show quite strong correlations. In addition, some pairings were inconsistent i.e. show correlations in some cases but this correlation then fell away depending on the overall metric score for the project.

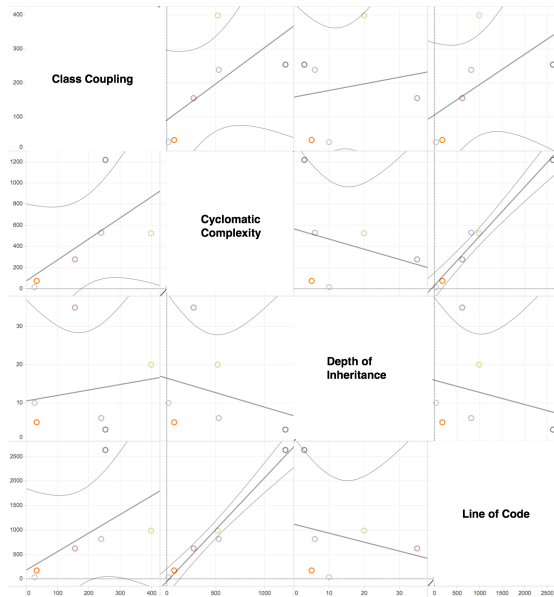


Figure 4.30: High Scoring Metrics

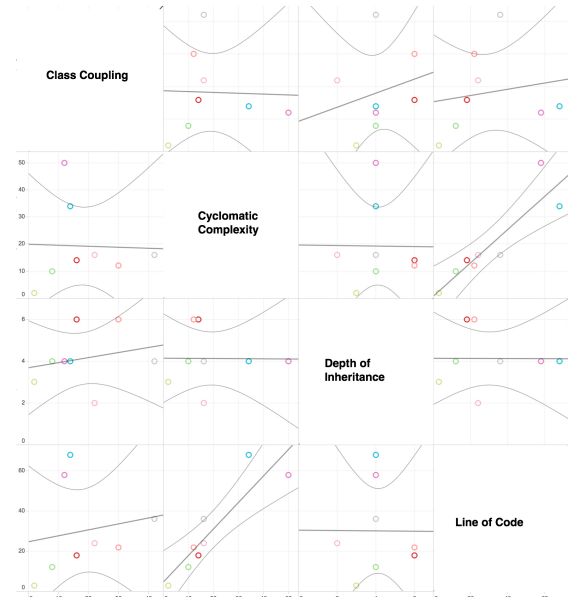


Figure 4.31: Low Scoring Metrics

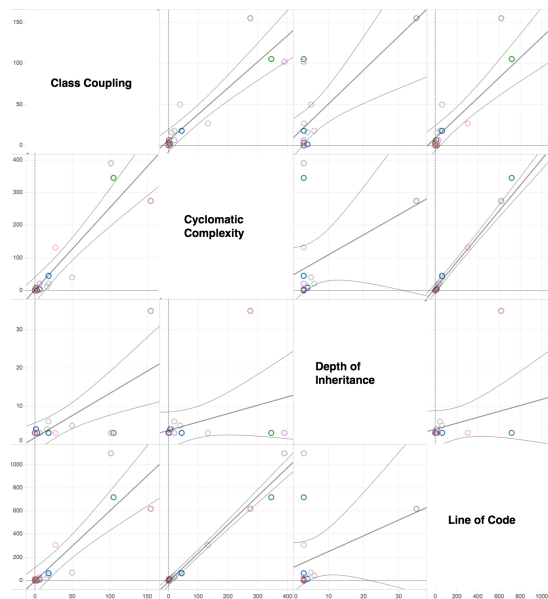


Figure 4.32: Average Scoring Metrics

Figures 4.28 through to 4.30 are scatterplot matrices generated from the visuals seen in the preceding sections of this chapter. These allow for an overview of the data in an effort to identify trends. One immediate and standout trend noticeable by scanning along the third row of each visual is the lack of correlation between Depth of Inheritance and any other metric. In addition, the strong correlation between Line of Code

Code and Cyclomatic Complexity is also notable. The next section will look at these and other findings in detail.

4.8.1 Cyclomatic Complexity and Class Coupling

Having initially shown a relatively close correlation when project metrics scores were high, the same relationship appeared to fall away when the metrics score was at the extreme lower end the scale only for it to regain slightly when project metrics recovered to an average level.

4.8.2 Depth of Inheritance and Class Coupling

Overall depth of inheritance and class coupling showed little to no relationship. A very slight correlation was visible in the average metric scoring project but not something of any real significance.

4.8.3 Lines of Code and Class Coupling

Similar to Cyclomatic Complexity and class coupling, lines of code and class coupling showed a slight correlation when the overall metrics score was high that completely fell away when the metric score was at the extreme low end. This correlation recovered slightly for the average scoring project possibly indicating that it occurs for reasons unknown where metrics scores are higher end of the scale.

4.8.4 Cyclomatic Complexity and Depth of Inheritance

Cyclomatic Complexity and depth of inheritance were two of the most consistent non-existing relationships. All three projects regardless of overall metrics score showed no relationship between Cyclomatic Complexity and depth of inheritance.

4.8.5 Cyclomatic Complexity and Lines of Code

Showing the most consistency for an existing relationship was Cyclomatic Complexity and lines of code. As can be seen in the small multiples visual below, there was a consistent correlation between Cyclomatic Complexity and lines of code.

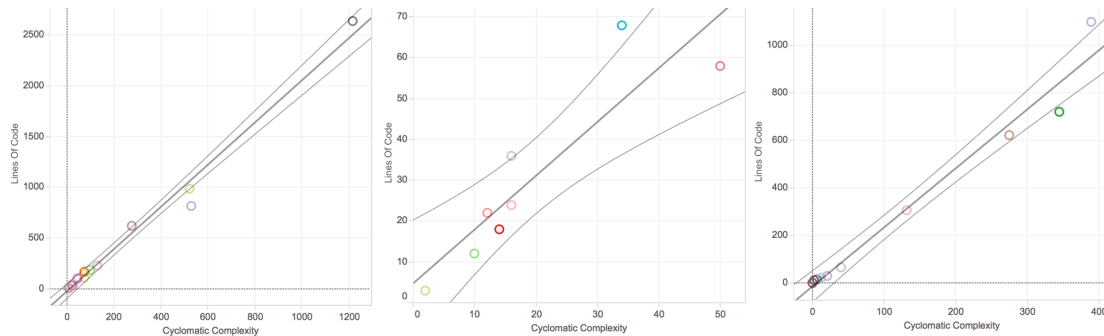


Figure 4.33: Small Multiples showing Cyclomatic Complexity and Lines of Code

4.8.6 Depth of Inheritance and Lines of Code

Similar to depth of inheritance and Cyclomatic Complexity there is consistently no sign of a relationship between depth of inheritance and lines of code. This was shown in all three comparisons regardless of the overall metrics score for the project.

4.8.7 Noteworthy Points

It should be noted that the metric that appear to form little or no relationship with any other metric it was compared against, was depth of inheritance. It consistently failed to establish any correlation regardless of overall metric score on the project or metric it was compared against.

In addition, the various code examples indicated a trend towards an increase in Cyclomatic Complexity when the number of methods in the class increased, regardless of access modifier i.e. public or private methods. Although one example contained a ternary operator, a short-handed if-statement, this did not appear to be required in order for the Cyclomatic Complexity to increase. The occurrence of a large number of methods within a class appeared to lead to an increase in the overall Cyclomatic Complexity of the class.

4.9 Conclusions

After identifying the Roslyn open source solution as having all the attributes required, including size, complexity in terms of its overall functional requirements and actively

worked on as a project, to become the basis of the data set this chapter looked at the various tools involved to extract the data and create the visuals for exploration.

Flipped bar charts were used to identify which projects within the solution had high, average and low metrics. The same format was then used to analyse each one. This enabled the research to identify any relationships that may have carried from one project to another. From time to time code snippets were also selected at random to help shed light on what coding characteristics were causing the various metrics.

While the exploration phase did highlight some findings in terms of the relationships between various metrics, the main two are the consistent relationship between Cyclomatic Complexity and lines of code and the complete lack of relationship between depth of inheritance and any other metric it was paired against. In addition, it was also noted that through random examination of the code, there appeared to be a relationship between the Cyclomatic Complexity and the number of private methods within a class.

It is this last point that will become the focus of the next chapter. In order to explore this relationship more closely i.e. the relationship between Cyclomatic Complexity and the number of private methods, the area will be expanded out to include both public and private methods and all the metrics that were examined in this chapter. For this to be achieved, a way of extracting a new data set containing information relating to the number of both public and private methods needs to be defined. Following that, the new data set will need to be merged with the data that was examined in this chapter.

5. Impact of Code Readability on Metrics

5.1 Introduction

While the previous chapter explored the metrics data generated from the Roslyn open source solution, looking for possible correlations between the metrics, this chapter will take one its key findings and further expand upon it in an effort to gain new insight.

The focus in this chapter will be on a possible relationship between an increase in the Cyclomatic Complexity of a class and the number of private methods within that class. In order to expand upon this further, the scope will be widened. The chapter will look to examine all metrics seen in the previous chapter against the number of both public and private methods within a class. In order to do this, additional data relating to the number of methods per type will be required from the Roslyn solution. As this data is not readily available with existing tools, additional software was developed to extract this data and merge it with the existing data from the previous chapter. This then enabled more visuals to be created in the form of scatter plots to further examine for possible correlations.

This chapter will begin by walking through all software that was developed in order to extract the new data while highlighting issues encountered in the process. It will then look at the process of merging this data with the existing data generated in the previous chapter. Due to differences in data format, an iterative process was employed until a satisfactory data set was generated.

5.2 Developing the Extraction Software

As no tools were readily available to extract data related to methods contained with the various types of the Roslyn solution, additional software was developed. This section will explain, step-by-step, the details of how the software works.

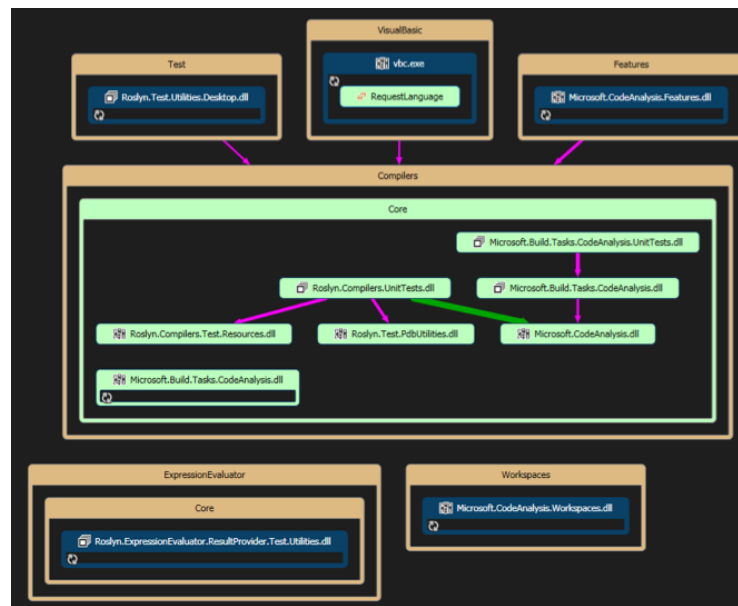


Figure 5.1: Overview of Roslyn Structure

Figure 5.1 shows an overview of how the Roslyn solution is structured and the direction of dependencies at a high level. By harnessing the `Assembly` class provided by the .NET platform the code loads each binary i.e. the compiled dynamic link library (DLL) file and using a concept known as reflection can extract large amounts of data about the binary. The focus of the code, as explained in detail below is to filter out only the data required for this research i.e. the public and private methods of each, format the data and save it out to an Excel spreadsheet.

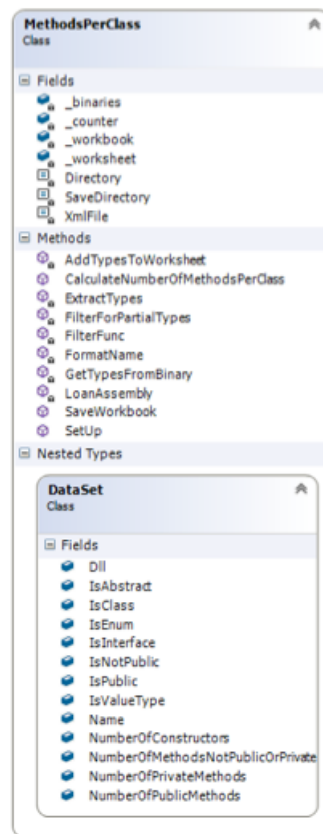


Figure 5.2: MethodsPerClass Class Diagram

The methods developed include:

- `MethodsPerClass` method whose job is to process each binary i.e. compiled DLL from the Roslyn solution, and extract out the data about the number of both public and private methods contained in each type.
- `SetUp` method whose job is to create the initial Excel worksheet that the data will be saved into. This includes defining the column headings that the new data will be saved under.
- `GetTypesFromBinary` method whose job is to find the binary on the local hard disk and load it into memory.
- `FilterFunc` method whose job is to extract out only the types that are required from the binary. By design Microsoft's .NET Assembly class returns large amounts of data and therefore filtering is often required.
- `ExtractTypes` method whose job is to map the required data back to the `DataSet` class. This is the nested class shown in the class diagram of Figure 5.2 above.

- `FormatName` method whose job is to remove unwanted characters from the type names as they are returned from the `Assembly` class.
- `AddTypesToWorksheet` method whose job is to insert a new row into the worksheet. Each insert consists of a type and the number of public and private methods for that type. The full list of extracted data corresponds to the nest `DataSet` class shown in Figure 5.2.

5.2.1 MethodsPerClass

The class written to perform the data extraction was called `MethodsPerClass`. This first figure shows the properties of the class.

```
0 references | 0 changes | 0 authors, 0 changes
public class MethodsPerClass
{
    private const string Directory = "C:\\Code\\roslyn\\Binaries\\Debug\\";
    private const string SaveDirectory = "C:\\Code\\";
    private const string XmlFile = "TypeData_Master.xlsx";
    private readonly XLWorkbook _workbook = new XLWorkbook();
    private IXLWorksheet _worksheet;
    private int _counter;
    private readonly List<string> _binaries = new List<string>
    {
        "Microsoft.CodeAnalysis.dll", "Roslyn.Test.PdbUtilities.dll", "Microsoft.CodeAnalysis.CSharp.dll",
        "Microsoft.DiaSymReader.PortablePdb.dll", "Microsoft.CodeAnalysis.CSharp.EditorFeatures.dll",
        "Microsoft.CodeAnalysis.ExpressionEvaluator.ExpressionCompiler.dll", "Microsoft.CodeAnalysis.CSharp.Features.dll",
        "Microsoft.CodeAnalysis.Features.dll",
        "Microsoft.CodeAnalysis.CSharp.InteractiveEditorFeatures.dll", "Microsoft.CodeAnalysis.InteractiveEditorFeatures.dll",
        "Microsoft.CodeAnalysis.InteractiveFeatures.dll",
        "Microsoft.VisualStudio.InteractiveWindow.dll", "Microsoft.VisualStudio.CSharp.Repl.dll",
        "Roslyn.VisualStudio.InteractiveComponents.dll", "Microsoft.VisualStudio.InteractiveServices.dll",
        "Microsoft.VisualStudio.VsInteractiveWindow.dll", "Microsoft.CodeAnalysis.CSharp.Scripting.dll",
        "Microsoft.VisualStudio.LanguageServices.CSharp.dll",
        "Roslyn.Hosting.Diagnostics.dll", "Microsoft.VisualStudio.LanguageServices.dll", "Microsoft.VisualStudio.LanguageServices.Implementation.dll",
        "Microsoft.VisualStudio.LanguageServices.SolutionExplorer.dll", "Roslyn.VisualStudio.DiagnosticsWindow.dll", "Roslyn.VisualStudio.Setup.dll",
        "Microsoft.CodeAnalysis.CSharp.Workspaces.dll", "Microsoft.CodeAnalysis.Workspaces.dll"
    };
}
```

Figure 5.3: Definition of `MethodsPerClass`

The properties of the class contain the relevant details as to what the class requires in terms of location of the binaries of the Roslyn solution, the name of the file the data will be saved to while the list of binaries the code was to execute on was loaded into a list named `_binaries`. One other notable inclusion here is a type called `XLWorkbook`, a type imported from a package called `ClosedXML`. `ClosedXML` is an open source library available from the nuget package store that allows operations to be completed on Excel files, including the creation of the files, opening existing files and updating files.

```

7 references | 0 changes | 0 authors, 0 changes
public class DataSet
{
    public string Name;
    public string Dll;
    public int NumberOfPublicMethods;
    public int NumberOfPrivateMethods;
    public int NumberOfMethodsNotPublicOrPrivate;
    public int NumberOfConstructors;
    public bool IsPublic;
    public bool IsNotPublic;
    public bool IsInterface;
    public bool IsClass;
    public bool IsAbstract;
    public bool IsEnum;
    public bool IsValueType;
}

```

Figure 5.4: Definition of DataSet class used within MethodsPerClass

The DataSet class provided a type for each piece of information collected from the binaries listed above. The job of this nested type within the program will be explained in more detail shortly but in essence this defines that for particular type various pieces of information were collected including the Name, NumberOfPublicMethods, NumberOfPrivateMethods etc. was collected.

```

0 references | 0 changes | 0 authors, 0 changes
public void Setup()
{
    _counter = 2;

    _worksheet = _workbook.Worksheets.Add("MethodCount");
    _worksheet.Cell("A1").Value = "Name";
    _worksheet.Cell("B1").Value = "Dll";
    _worksheet.Cell("C1").Value = "NumberOfPublicMethods";
    _worksheet.Cell("D1").Value = "NumberOfPrivateMethods";
    _worksheet.Cell("E1").Value = "NumberOfMethodsNotPublicOrPrivate";
    _worksheet.Cell("F1").Value = "NumberOfConstructors";
    _worksheet.Cell("G1").Value = "IsPublic";
    _worksheet.Cell("H1").Value = "IsNotPublic";
    _worksheet.Cell("I1").Value = "IsInterface";
    _worksheet.Cell("J1").Value = "IsClass";
    _worksheet.Cell("K1").Value = "IsAbstract";
    _worksheet.Cell("L1").Value = "IsEnum";
    _worksheet.Cell("M1").Value = "IsValueType";
}

```

Figure 5.5: Definition of Setup method

The Setup method defined here sets up the initial Excel worksheet, where the collected data will be added, and also defines the names seen in the DataSet to each of the columns within that Excel spread sheet.

```

foreach (var binary in _binaries)
{
    IEnumerable<Type> types = GetTypesFromBinary(binary);
    IEnumerable<DataSet> dataSet = ExtractTypes(types, binary);
    dataSet = FilterForPartialTypes(dataSet);
    AddTypesToWorksheet(dataSet);
}

```

Figure 5.6: Definition of the foreach statement used to iterate through all the Roslyn binaries

The main execution of the code is to apply four steps to each of the binaries, as follows: getting the types from the binaries, extracting the required information into the `DataSet` type defined earlier i.e. more information than required is returned and therefore only the required information is extracted out at this point. This will be covered in more detail shortly, filtering out duplication caused by partial types and finally applying the list of `DataSets` to the worksheet. The program completes by saving the worksheet to a local directory. Each of these steps will now be examined in more detail.

```
1 reference | 0 changes | 0 authors, 0 changes
private static List<Type> GetTypesFromBinary(string binary)
{
    return LoadAssembly(binary).GetTypes().Where(FilterFunc()).ToList();
}

1 reference | 0 changes | 0 authors, 0 changes
private static Func<Type, bool> FilterFunc()
{
    return type => !type.Name.StartsWith("<") && !type.Name.StartsWith("_");
}

1 reference | 0 changes | 0 authors, 0 changes
private static Assembly LoadAssembly(string binary)
{
    return Assembly.LoadFrom(Directory + binary);
}
```

Figure 5.7: Definitions of three methods used within `MethodsPerClass`

The code above shows three private methods, the bottom two of which are called by the first method, `GetTypesFromBinary`. This method takes the string name of the binary and calls the third of the three methods shown above, `LoadAssembly`. This method in-turn uses the `Assembly` class provided by the .NET framework to load the assembly from the directory previously defined in the properties shown earlier. Once the assembly is loaded, control returns to the calling method, `GetTypesFromBinary`.

Using the chaining of methods the next action preformed is `GetTypes`. This method, provided by the `Assembly` type, is used to load the binary and returns all the types found within that binary. It then continues to convert the binaries into a list to be returned to the calling method, but before doing so applies a `FilterFunc`, the second method defined above. The `FilterFunc` returns true or false to the `Where` clause, depending on whether or not the name of the type begins with an angle bracket or underscore.

The reasoning behind applying the `FilterFunc` will be explained shortly. It should be noted that the `Where` clause to a method provided `GetTypes` returning an `IEnumerable` an interface definition provided by the .NET framework but the details of which are beyond the scope of this research.

```
1 reference | 0 changes | 0 authors, 0 changes
private IEnumerable<DataSet> ExtractTypes(IEnumerable<Type> types, string binary)
{
    return types.Select(type => new DataSet
    {
        Name = FormatName(type.Name),
        Dll = binary,
        NumberOfPublicMethods = type.GetRuntimeMethods().Count(x => x.IsPublic && !x.IsConstructor),
        NumberOfPrivateMethods = type.GetRuntimeMethods().Count(x => x.IsPrivate && !x.IsConstructor),
        NumberOfMethodsNotPublicOrPrivate = type.GetRuntimeMethods().Count(x => !x.IsPrivate && !x.IsPublic && !x.IsConstructor),
        NumberOfConstructors = type.GetRuntimeMethods().Count(x => x.IsConstructor),
        IsPublic = type.IsPublic,
        IsNotPublic = type.IsNotPublic,
        IsInterface = type.IsInterface,
        IsClass = type.IsClass,
        IsAbstract = type.IsAbstract,
        IsEnum = type.IsEnum,
        IsValueType = type.IsValueType
    }).ToList();
}
```

Figure 5.8: Definition of ExtractTypes methods

The next step in the process is to extract only the information that is required for the data being assembled i.e. the details of public and private methods contained in a class. This is required as the `GetTypes` call shown earlier returns a large amount of data by default when called on a binary.

For the purpose of this research only a small subset of this data is required. The `ExtractTypes` method defined above employs the use of lambda expression, the details of which are beyond the scope of this explanation, but it is sufficient to say that the method is taking each required piece of information from each of the types in the list sent to it by the calling method, creating a new instance of `DataSet` for each, extracting the data it requires and returning the resulting list back to the calling method.

```
1 reference | 0 changes | 0 authors, 0 changes
private static string FormatName(string name)
{
    return name.Contains("`") ? name.Substring(0, name.IndexOf("`", StringComparison.Ordinal)) : name;
}
```

Figure 5.9: Definition of FormatName

In addition, this method also calls an additional method, `FormatName`. This removes any back ticks that appear in the names. The details of this will be explained fully in the next section.

```

1 reference | 0 changes | 0 authors, 0 changes
private IEnumerable<DataSet> FilterForPartialTypes(IEnumerable<DataSet> dataSet)
{
    var filteredSet = new List<DataSet>();
    foreach (var entry in dataSet)
    {
        if (!filteredSet.Exists(x => x.Name == entry.Name))
        {
            filteredSet.Add(entry);
        }
        else
        {
            filteredSet[filteredSet.FindIndex(x => x.Name == entry.Name)].NumberOfPrivateMethods += entry.NumberOfPrivateMethods;
            filteredSet[filteredSet.FindIndex(x => x.Name == entry.Name)].NumberOfPublicMethods += entry.NumberOfPublicMethods;
            filteredSet[filteredSet.FindIndex(x => x.Name == entry.Name)].NumberOfConstructors += entry.NumberOfConstructors;
            filteredSet[filteredSet.FindIndex(x => x.Name == entry.Name)].NumberOfMethodsNotPublicOrPrivate += entry.NumberOfMethodsNotPublicOrPrivate;
        }
    }
    return filteredSet;
}

```

Figure 5.10: Definition of FilterForPartialTypes

At this point the code has loaded the binary and extracted the information from each type and placed it in a list of DataSet types. The next step of the process is to filter out partial types. As this particular piece went through several iterations the details will be explained in the next section.

```

1 reference | 0 changes | 0 authors, 0 changes
private void AddTypesToWorksheet(IEnumerable<DataSet> dataSet)
{
    foreach (var entry in dataSet)
    {
        _worksheet.Cell("A" + _counter).Value = entry.Name;
        _worksheet.Cell("B" + _counter).Value = entry.Dll;
        _worksheet.Cell("C" + _counter).Value = entry.NumberOfPublicMethods;
        _worksheet.Cell("D" + _counter).Value = entry.NumberOfPrivateMethods;
        _worksheet.Cell("E" + _counter).Value = entry.NumberOfMethodsNotPublicOrPrivate;
        _worksheet.Cell("F" + _counter).Value = entry.NumberOfConstructors;
        _worksheet.Cell("G" + _counter).Value = entry.IsPublic;
        _worksheet.Cell("H" + _counter).Value = entry.IsNotPublic;
        _worksheet.Cell("I" + _counter).Value = entry.IsInterface;
        _worksheet.Cell("J" + _counter).Value = entry.IsClass;
        _worksheet.Cell("K" + _counter).Value = entry.IsAbstract;
        _worksheet.Cell("L" + _counter).Value = entry.IsEnum;
        _worksheet.Cell("M" + _counter).Value = entry.IsValueType;
        _counter++;
    }
}

```

Figure 5.11: Definition of AddTypesToWorksheet

When all the data is collected and filtered the code then appends the data into a manageable format of an Excel sheet. The AddTypesToWorksheet method defined above simply loops through the list of DataSet types inserting each into a cell on the Excel worksheet.

```

_workbook.SaveAs(SaveDirectory + XmlFile);

```

Figure 5.12: Save new workbook to local hard disk

The last part of program execution is to save the workbook to the local hard disk.

5.3 Data Preparation

The previous section provided a brief overview of the code used to collect the method information from the Roslyn solution. At three points in the overview, it was noted that certain aspects of code would require further explanation in the context of how the data was prepared. The areas were the application of a FilterFunc, filtering out partial types and formatting of the type name, each of which will be explained in due course.

These areas are in effect dealing with anomalies that arose when attempting to merge the data generated by the code above with the data previously generated using the visual studio tools for calculation of metrics that were examined in the previous chapter. These anomalies lead to both the changes highlighted in the preceding code along with the creation additional code to clean the data provided by code metrics analyser in visual studio. This section will now discuss this iterative process in detail.

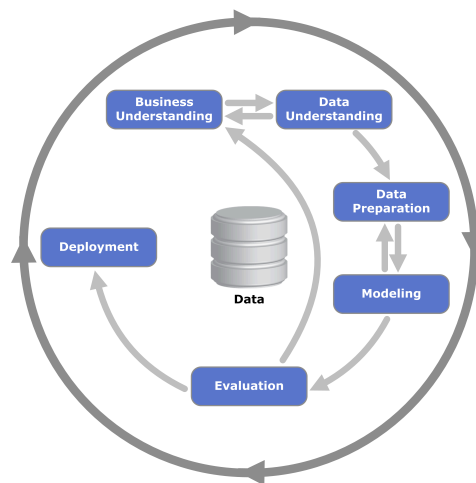


Figure 5.13: Data Preparation using CRISP-DM

As no readily available software had the ability to generate a data set containing the number of methods in each type, new code was written to extract the data from the Roslyn solution. An overview of this code was provided in the previous section. On comparing the generated data with the original metric data it was clear that anomalies were present that meant the data could not be easily merged. This included the naming of types and whether partial types should be considered separately or as part of their complete class etc. A process of cleaning the data to allow both data sets to be merged together was required.

The process undertaken to clean both sets of data followed the Cross Industry Standard Process for Data Mining or CRISP-DM. This is shown in the diagram above. It provides a standard process for data to be generated, data preparation, modelling and evaluation. The two areas that required this iterative process were the formatting of the name and handling of partial types, both of which will be explained in detail shortly. Prior to looking at these issues in detail, a walk-through of the code written to clean the code metric data, generated by visual studio will be undertaken.

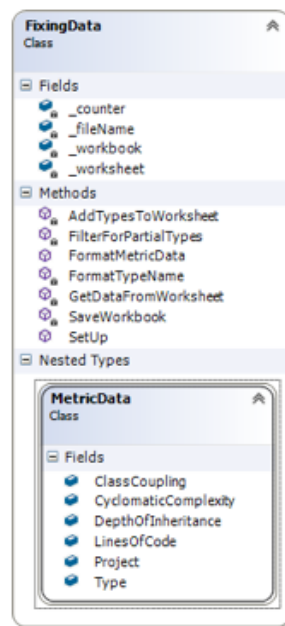


Figure 5.14: FixingData Class Diagram

The methods developed include:

- `FixingData` method whose job is to read the existing metric data set containing all the information regarding Cyclomatic Complexity etc. and reformat it in order for it to merge with the data set on public and private methods within the types.
- `GetDataFromWorksheet` method whose job is to loop through all the metric data contained in an Excel worksheet on the local hard disk and read each line, and create a new type called `MetricData` for each. The `MetricData` class is shown in Figure 5.14 above.
- `FormatTypeName` method whose job is to ensure the name is formatted correctly. This is very important in ensuring the data merges successfully as the merge will be done on the name of the type.

- `FilterForPartialTypes` method whose job is to ensure that all partial classes are recorded as single class.

5.3.1 FixingData

The code used to clean the code metrics data was written in a class called `FixingData`. This section will take brief overview of the code that was written.

```

0 references | 0 changes | 0 authors, 0 changes
class FixingData
{
    private XLWorkbook _workbook;
    private IXLWorksheet _worksheet;
    private int _counter;
    private readonly string _fileName = "C:\\\\Code\\\\Roslyn.xlsx";

    8 references | 0 changes | 0 authors, 0 changes
    public class MetricData
    {
        public string Project;
        public string Type;
        public int CyclomaticComplexity;
        public int DepthOfInheritance;
        public int ClassCoupling;
        public int LinesOfCode;
    }
}

```

Figure 5.15: Definition of `FixingData` class

The first code snippet shown is the properties the `FixingData` class defines to be used during the execution of the code. It defines the `XLWorkbook` that will be used to load the data from the Excel file and also the location of the file on the local hard disk. In addition, it defines a type called `MetricData` that will be used to hold the data read in from the Excel file.

```

0 references | 0 changes | 0 authors, 0 changes
public void Setup()
{
    _counter = 2;
    _workbook = new XLWorkbook();

    _worksheet = _workbook.Worksheets.Add("MetricData");
    _worksheet.Cell("A1").Value = "Project";
    _worksheet.Cell("B1").Value = "Type";
    _worksheet.Cell("C1").Value = "CyclomaticComplexity";
    _worksheet.Cell("D1").Value = "DepthOfInheritance";
    _worksheet.Cell("E1").Value = "ClassCoupling";
    _worksheet.Cell("F1").Value = "LinesOfCode";
}

```

Figure 5.16: Definition of `Setup` method

On the initial setup, just prior to code execution, a new instance of the workbook is created and a new worksheet is added with the column definitions matching the previously defined `MetricData`.

```
var metricData = GetDataFromWorksheet(new XLWorkbook(_fileName).Worksheet(1));
metricData = FilterForPartialTypes(metricData);
AddTypesToWorksheet(metricData);
SaveWorkbook();
```

Figure 5.17: Reading in data from workbook on local hard disk

The main execution of the code takes the format above. It first retrieves the data from the existing workbook. Note that this is the data generated using visual studio's code analysis tool that was examined in detail in the previous chapter. Each of these steps will now be examined in detail.

```
1 reference | 0 changes | 0 authors, 0 changes
private List<MetricData> GetDataFromWorksheet(IXLWorksheet originalWorksheet)
{
    List<MetricData> resultingData = new List<MetricData>();

    int rowCount = originalWorksheet.Rows().AdjustToContents().Count();

    for (var i = 2; i < rowCount; i++)
    {
        var name = FormatTypeName(originalWorksheet.Row(i).Cell(2).GetString());

        if (!name.StartsWith("_"))
        {
            resultingData.Add(new MetricData
            {
                Project = originalWorksheet.Row(i).Cell(1).GetString(),
                Type = name,
                CyclomaticComplexity = Int32.Parse(originalWorksheet.Row(i).Cell(3).GetString()),
                DepthOfInheritance = Int32.Parse(originalWorksheet.Row(i).Cell(4).GetString()),
                ClassCoupling = Int32.Parse(originalWorksheet.Row(i).Cell(5).GetString()),
                LinesOfCode = Int32.Parse(originalWorksheet.Row(i).Cell(6).GetString()),
            });
        }
    }

    return resultingData;
}
```

Figure 5.18: Definition of GetDataFromWorksheet

The method `GetDataFromWorksheet` takes the original worksheet that has been loaded from the disk and extracts the data into a list of type `MetricData`. Using `rowCount` to keep track, it keeps looping through the data adding an entry to the `MetricData` list each time.

```
1 reference | 0 changes | 0 authors, 0 changes
private string FormatTypeName(string name)
{
    if (name.Contains(".") && name.Contains("<"))
    {
        return name.Length - 1 > name.LastIndexOf("<", StringComparison.Ordinal) ?
            name.Substring(name.LastIndexOf("<", StringComparison.Ordinal)+1) :
            name.Substring(name.LastIndexOf("<", StringComparison.Ordinal)+1,
                (name.LastIndexOf("<", StringComparison.Ordinal)-(name.LastIndexOf(".", StringComparison.Ordinal)+1)));
    }
    else if (name.Contains("<"))
    {
        return name.Substring(0, name.LastIndexOf("<", StringComparison.Ordinal));
    }
    else if (name.Contains("."))
    {
        return name.Substring(name.LastIndexOf(".", StringComparison.Ordinal)+1);
    }
    return name;
}
```

Figure 5.19: Definition of FormatTypeName

On retrieving the data from the original worksheet, the above method also calls out to the method defined above, `FormatTypeName`. This is one of the two key functions to this code and will be explained in more detail shortly.

```
1 reference | 0 changes | 0 authors, 0 changes
private List<MetricData> FilterForPartialTypes(List<MetricData> dataSet)
{
    var filteredSet = new List<MetricData>();
    foreach (var entry in dataSet)
    {
        if (filteredSet.Exists(x => x.Type == entry.Type && x.Project == entry.Project))
        {
            filteredSet[filteredSet.FindIndex(x => x.Type == entry.Type)].CyclomaticComplexity += entry.CyclomaticComplexity;
            filteredSet[filteredSet.FindIndex(x => x.Type == entry.Type)].DepthOfInheritance += entry.DepthOfInheritance;
            filteredSet[filteredSet.FindIndex(x => x.Type == entry.Type)].ClassCoupling += entry.ClassCoupling;
            filteredSet[filteredSet.FindIndex(x => x.Type == entry.Type)].LinesOfCode += entry.LinesOfCode;
        }
        else
        {
            filteredSet.Add(entry);
        }
    }
    return filteredSet;
}
```

Figure 5.20: Definition of `FilterForPartialTypes`

Applying this specific filter to partial types is one of the two main functions of this code snippet. For now it is sufficient to say that it is executed at this point in the code and will be examined in more detail later.

```
1 reference | 0 changes | 0 authors, 0 changes
private void AddTypesToWorksheet(IEnumerable<MetricData> dataSet)
{
    foreach (var entry in dataSet)
    {
        _worksheet.Cell("A" + _counter).Value = entry.Project;
        _worksheet.Cell("B" + _counter).Value = entry.Type;
        _worksheet.Cell("C" + _counter).Value = entry.CyclomaticComplexity;
        _worksheet.Cell("D" + _counter).Value = entry.DepthOfInheritance;
        _worksheet.Cell("E" + _counter).Value = entry.ClassCoupling;
        _worksheet.Cell("F" + _counter).Value = entry.LinesOfCode;
        _counter++;
    }
}
```

Figure 5.21: Definition of `AddTypesToWorksheet`

```
1 reference | 0 changes | 0 authors, 0 changes
private void SaveWorkbook()
{
    _workbook.SaveAs("C:\\Code\\Roslyn_Master.xlsx");
}
```

Figure 5.22: Definition of `SaveWorkbook`

Once the data has been formatted in this way, it is populated back into an Excel sheet and saved back on to the local hard disk.

5.3.2 Formatting Type Name

As the name was the identifier on which the two sets of data were merged it was imperative that both sets of type names matched exactly. That led to changes being required on both sets of data. In order to best understand how this was achieved consider the name format applied to each piece of code above.

```
1 reference | 0 changes | 0 authors, 0 changes
private static string FormatName(string name)
{
    return name.Contains("`") ? name.Substring(0, name.IndexOf("`", StringComparison.Ordinal)) : name;
}
```

Figure 5.23: Definition of FormatName

This FormatName method shown above was required eliminate the occurrence of back ticks in type names within the method details extraction code.

```
1 reference | 0 changes | 0 authors, 0 changes
private string FormatTypeName(string name)
{
    if (name.Contains(".") && name.Contains("<"))
    {
        return name.Length-1 > name.LastIndexOf("<", StringComparison.Ordinal) ?
            name.Substring(name.LastIndexOf(".", StringComparison.Ordinal)+1) :
            name.Substring(name.LastIndexOf(".", StringComparison.Ordinal)+1,
                (name.LastIndexOf("<", StringComparison.Ordinal)-(name.LastIndexOf(".", StringComparison.Ordinal)+1)));
    }
    else if (name.Contains("<"))
    {
        return name.Substring(0, name.LastIndexOf("<", StringComparison.Ordinal));
    }
    else if (name.Contains("."))
    {
        return name.Substring(name.LastIndexOf(".", StringComparison.Ordinal)+1);
    }
    return name;
}
```

Figure 5.24: Definition of FormatTypeName

Once the back tick issue was eliminated from methods data side, an issue arose with the structure of the naming convention within the metric data side. This meant that type names with particular characteristics were outputted in the format of these name types.

- MetadataDecoder<ModuleSymbol, TypeSymbol, MethodSymbol, FieldSymbol, Symbol>
- AbstractLookupSymbolsInfo<TSymbol>.UniqueSymbolOrArities
- CompilerDiagnosticAnalyzer.CompilationAnalyzer.CompilerDiagnostic

As the method data format the name types without the type name in angle brackets or preceding it nested class types, it was required that each of these names be reformatted to

- MetadataDecoder
- UniqueSymbolOrArities
- CompilerDiagnostic

This is essentially what the `FormatTypeName` method above is doing.

5.3.3 Filtering of Partial Classes

Before looking at the issues around partial class, the concept of a partial class itself needs to be examined. Normally a class is presented on a single document i.e. a single page. For various reasons, beyond the scope of this research, some languages provide the ability for a class to be split over two or more documents with using the keyword `partial`. As far as the compiler is concerned this is a single class and all parts to a class must be contained in a single project within the visual studio solution. For this reason, this research will also consider a partial class as multiple parts of the same class and therefore treat it as one single class.

The issue that arises here is that merging these partial classes into one class must be done as part of cleaning the data to allow for the two sets to be successfully merged together. Both code snippets below provide the functionality, applied differently to each, that allows for a consistent format of data to be generated. It is not enough to simply merge the names, on doing so the data, whether metric data or methods data, also needs to be accumulated together to provide an overall total for each column.

```
1 reference, 0 changes, 0 authors, 0 changes
private IEnumerable<DataSet> FilterForPartialTypes(IEnumerable<DataSet> dataSet)
{
    var filteredSet = new List<DataSet>();

    foreach (var entry in dataSet)
    {
        if (!filteredSet.Exists(x => x.Name == entry.Name))
        {
            filteredSet.Add(entry);
        }
        else
        {
            filteredSet[filteredSet.FindIndex(x => x.Name == entry.Name)].NumberOfPrivateMethods += entry.NumberOfPrivateMethods;
            filteredSet[filteredSet.FindIndex(x => x.Name == entry.Name)].NumberOfPublicMethods += entry.NumberOfPublicMethods;
            filteredSet[filteredSet.FindIndex(x => x.Name == entry.Name)].NumberOfConstructors += entry.NumberOfConstructors;
            filteredSet[filteredSet.FindIndex(x => x.Name == entry.Name)].NumberOfMethodsNotPublicOrPrivate += entry.NumberOfMethodsNotPublicOrPrivate;
        }
    }

    return filteredSet;
}
```

Figure 5.25: Definition of `FilterForPartialTypes`

```

1 reference | 0 changes | 0 authors, 0 changes
private List<MetricData> FilterForPartialTypes(List<MetricData> dataSet)
{
    var filteredSet = new List<MetricData>();

    foreach (var entry in dataSet)
    {
        if (filteredSet.Exists(x => x.Type == entry.Type && x.Project == entry.Project))
        {
            filteredSet[filteredSet.FindIndex(x => x.Type == entry.Type)].CyclomaticComplexity += entry.CyclomaticComplexity;
            filteredSet[filteredSet.FindIndex(x => x.Type == entry.Type)].DepthOfInheritance += entry.DepthOfInheritance;
            filteredSet[filteredSet.FindIndex(x => x.Type == entry.Type)].ClassCoupling += entry.ClassCoupling;
            filteredSet[filteredSet.FindIndex(x => x.Type == entry.Type)].LinesOfCode += entry.LinesOfCode;
        }
        else
        {
            filteredSet.Add(entry);
        }
    }

    return filteredSet;
}

```

Figure 5.26: Definition of FilterForPartialTypes

5.3.4 Merging the Data

Once both sets of data were generated, they were manually loaded into a single Excel workbook on two separate tabs. From there the Excel sheet was loaded into Tableau where further data exploration could be conducted. It is this data exploration that the remainder of this chapter is concerned with.

5.4 Exploring the Merged Data Set

Having extracted out the method details of each type and merged this data with the code metrics, it allows for additional data exploration where the metrics presented in Chapter Three can be compared against the number of private and public methods within that class. For example, is there any relationship between the number of the class coupling metrics of a group of classes and the number of public and private methods contained within those classes? In order to explore this fully, scatter plots were generated for the CSharpCodeAnalysis. Each metric, class coupling, Cyclomatic Complexity, depth of inheritance and lines of code, were compared against the number of public and private methods in various types.

5.4.1 Merged Data: Class Coupling and Number of Public Methods

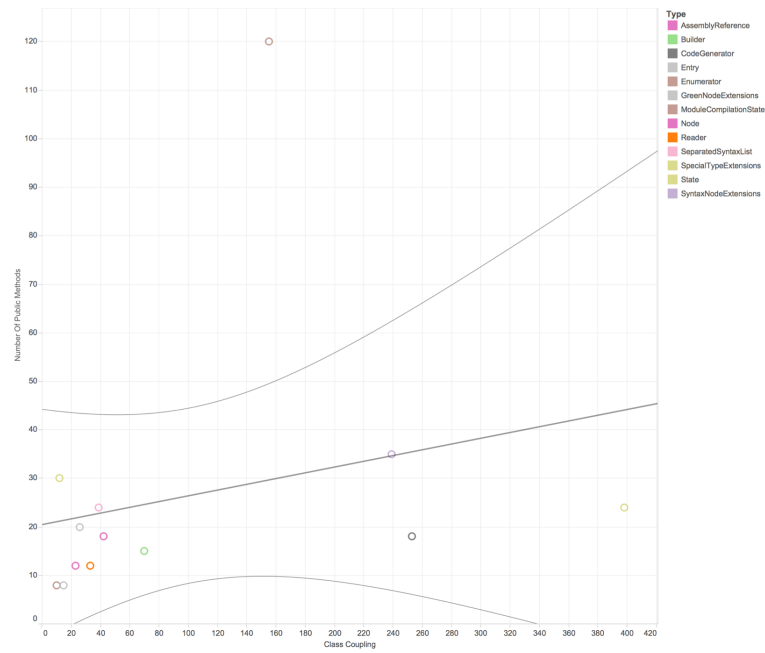


Figure 5.27: Class Coupling and Number of Public Methods

On comparing class coupling to a number of public methods, the scatter plot indicates that not much of a relationship appears to exist between the two. Logically this is not unsurprising as class can be coupled together without necessarily leading to an increase in the number of public methods of that class.

5.4.2 Merged Data: Class Coupling and Number of Private Methods

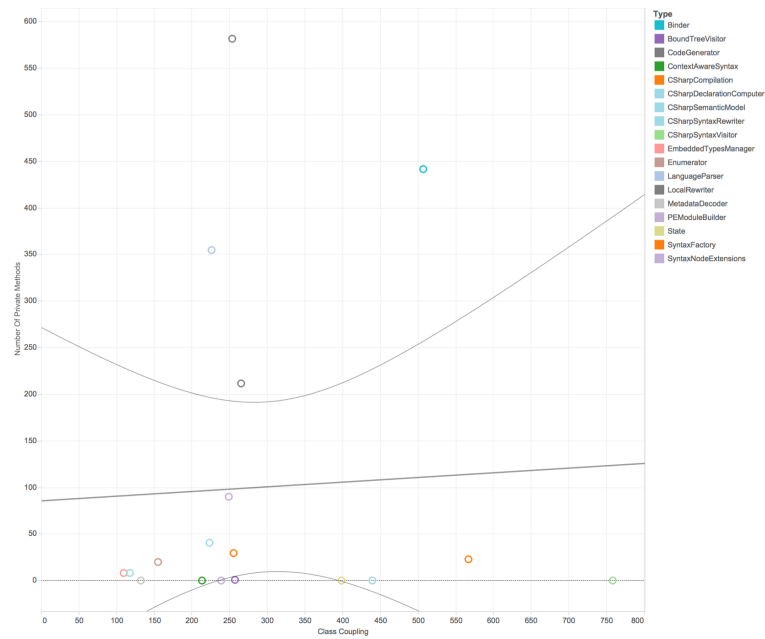


Figure 5.28: Class Coupling and Number of Private Methods

Similar to the comparison of public methods, there appears to be even less of a relationship between class coupling and the number of private methods.

5.4.3 Merged Data: Cyclomatic Complexity and Number of Public Methods

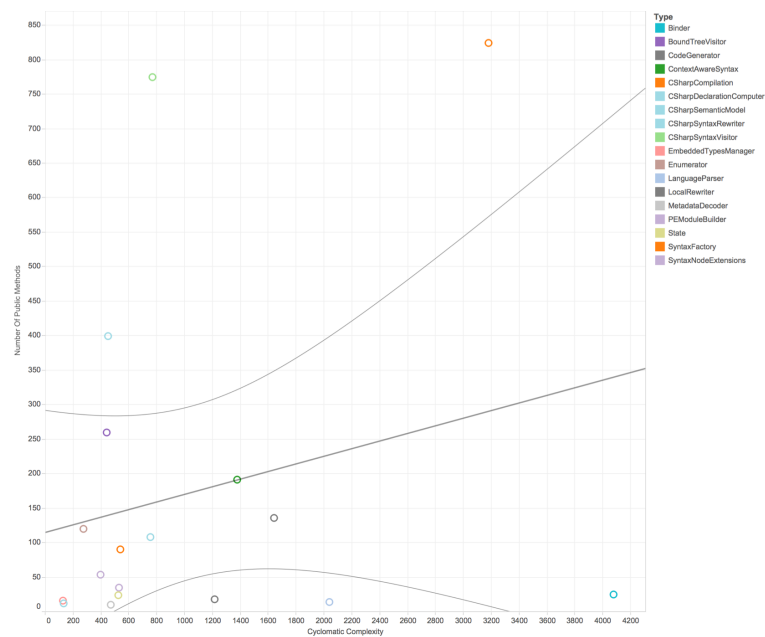


Figure 5.29: Cyclomatic Complexity and Number of Public Methods

The Cyclomatic Complexity metric does not appear to have too much of a relationship with the number of public methods. This is not unsurprising as a lot of complex code logic could be placed in one public method that may increase the overall Cyclomatic Complexity of the class without increasing the public method count of that class.

5.4.4 Merged Data: Cyclomatic Complexity and Number of Private Methods

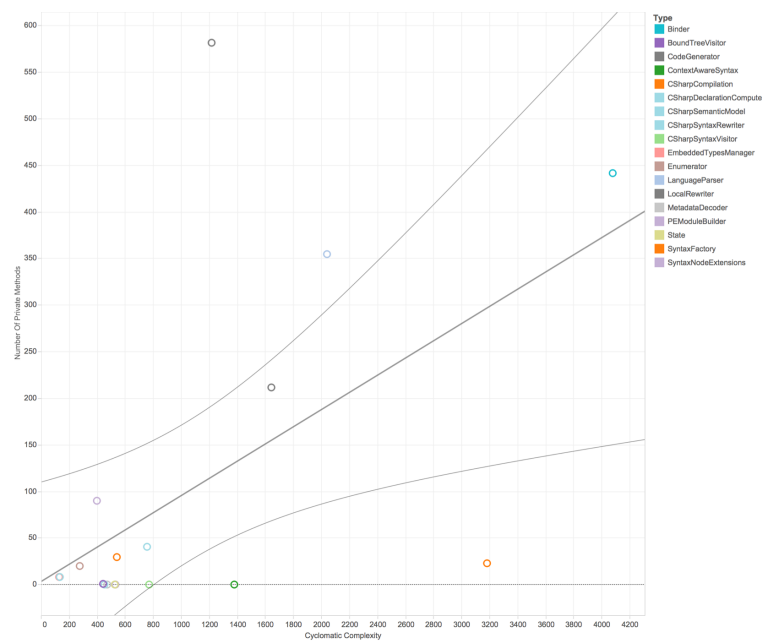


Figure 5.30: Cyclomatic Complexity and Number of Private Methods

Perhaps one of the most interesting findings is this relationship between Cyclomatic Complexity and the number of private methods in a class. This makes for an interesting finding, as it would seem to suggest that as the number of private methods increases in a class, the Cyclomatic Complexity of that class also increases.

5.4.5 Merged Data: Depth of Inheritance and Number of Public Methods

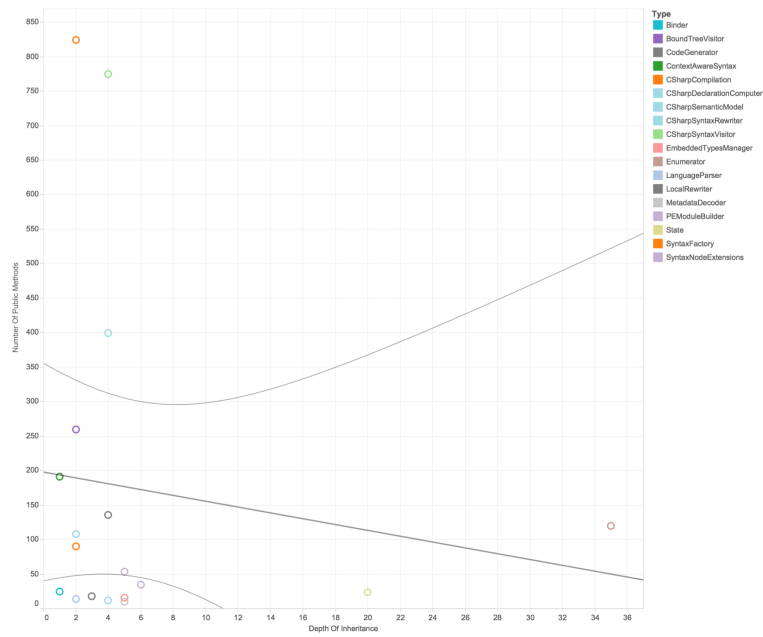


Figure 5.31: Depth of Inheritance and the Number of Public Methods

There appears to be zero relationship between depth of inheritance and the number of public methods.

5.4.6 Merged Data: Depth of Inheritance and Number of Private Methods

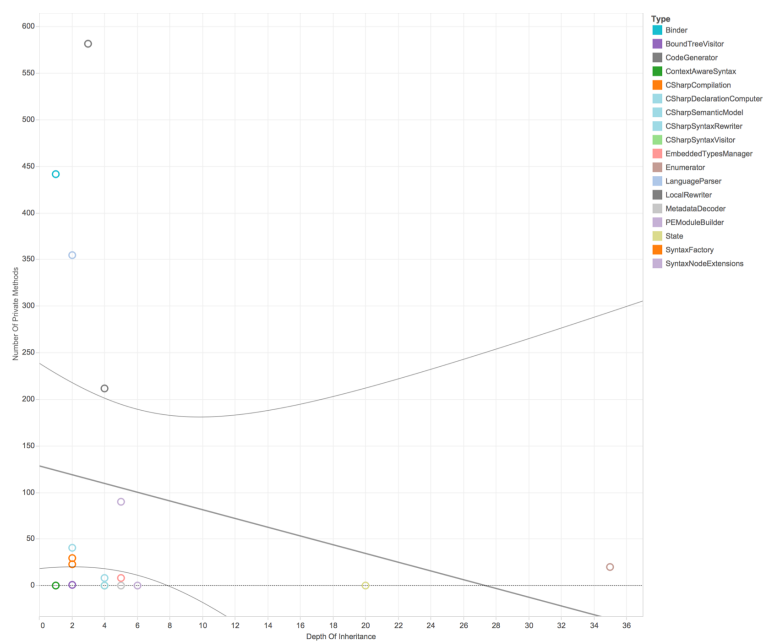


Figure 5.32: Depth of Inheritance and the Number of Private Methods

Similar to public methods, depth of inheritance appears to have no relationship with the number of private methods in the class. This is logically not surprising, as the growing of inheritance tree with classes does not necessarily indicate an increase the number of methods, regardless of access modifier, in that class.

5.4.7 Merged Data: Lines of Code and Number of Public Methods

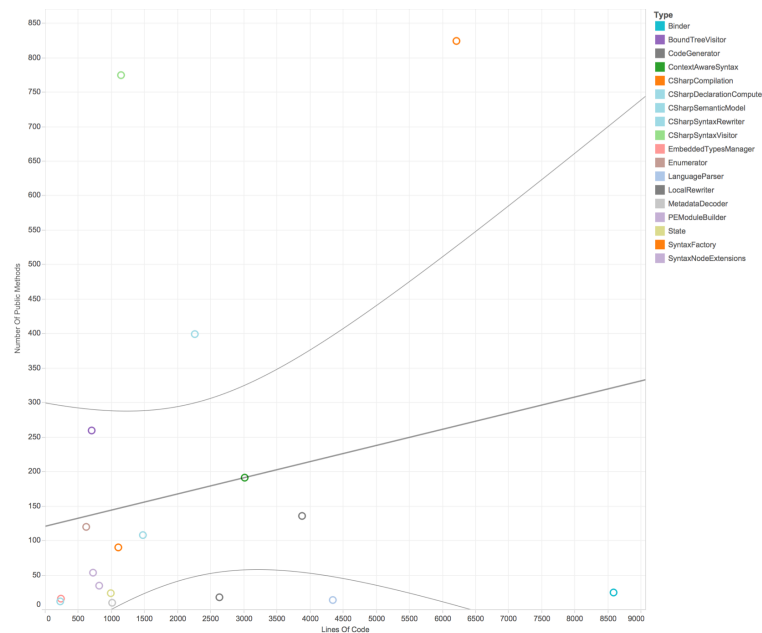


Figure 5.33: Lines of Code and Number of Public Methods

Not unsurprisingly, the number of lines of code does not appear to have much of relationship with the number of public methods in the class. Similar to Cyclomatic Complexity this could be explained by the fact that a lot of complex code and hence an increase in the number of lines of code can increase the metrics but not the public method count as it can be all contained within that one method.

5.4.8 Merged Data: Lines of Code and Number of Private Methods

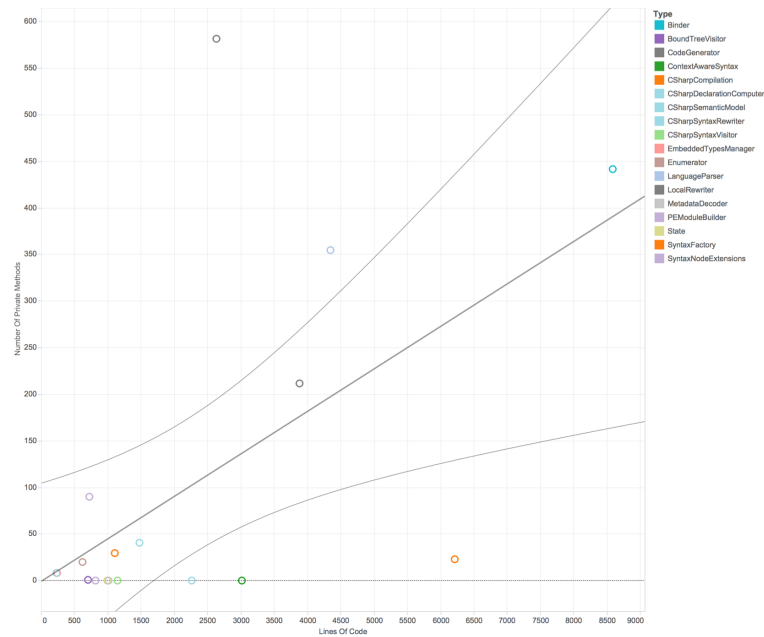


Figure 5.34: Lines of Code and Number of Private Methods

There appears be more of a relationship between lines of code and the number of private methods. This is similar to the same comparison above with Cyclomatic Complexity where a relationship appeared between the number of private methods and the Cyclomatic Complexity of that class.

5.5 Key Findings

	Class Coupling	Cyclomatic Complexity	Depth of Inheritance	Line of Codes
Public Methods	N	N	N	N
Private Methods	N	Y	N	Y

Figure 5.35: Overview of findings from scatter plots

The table above shows an overview summary of the findings with the newly generated merged data set. It indicates a possible relationship between both Cyclomatic Complexity and lines of code with the number of private methods. Unsurprisingly,

depth of inheritance and class coupling appear to offer no firm relationship while the number of public methods does not appear to correlate to any of the metrics above.

5.6 Conclusions

This chapter took a key finding from the Chapter Four and after expanding its scope slightly, looked for possible relationships between the metrics of Chapter Four and possible relationships with the number of both public and private method of a given class.

In order to achieve this, new code was developed that extracted the data from the solution. Then an iterative process was undertaken that allowed the new data to be merged with the data examined in the previous chapter.

Out of the four metrics compared against public and private method counts within the same class only two showed any significant relationship, that of Cyclomatic Complexity and lines of code when paired against private methods. This is consistent with the initial findings of the previous chapter.

In addition, this also impacts on the newly introduced concepts from Martin (2008) and principles of ‘clean code’. By applying the code readability principle within the Stepdown rule it now appears that this will in fact increase the Cyclomatic Complexity of that same class. Given that the original concept behind the Cyclomatic Complexity was to identify code that was overly complex this finding goes against this assumption would suggest that code with a Cyclomatic Complexity can in fact be more human readable. The caveat being that it is in the case of high Cyclomatic Complexity and a high number of private methods.

The next chapter will take this key finding and investigate it further. It will take both a quantitative and qualitative approach and will seek to determine if code that is arguably more readable is in fact more complex as is the original basis of the Cyclomatic Complexity calculation.

6. Evaluation

6.1 Introduction

Chapter Five added a new perspective on code metrics by generating new data that regarding the number of public and private methods contained in each class. It found that an increase in the number of private methods leads to an increase in the Cyclomatic Complexity of the class. This resulted in the conclusion at the end of the last chapter that applying the Stepdown rule as defined by Martin (2008) leads to an increase in the Cyclomatic Complexity of the class. This puts both principles in stark contrast to each other as the Stepdown rule is designed to make code more readable at a human level. The original intention of the Cyclomatic Complexity is to identify code that is complex and has a high risk of being defective.

This chapter will look to evaluate these findings and in order to do so, it will be required to take a two-pronged approach, using a so-called Mixed Methods Approach.

The first section will be a quantitative assessment while the second will be in the form a qualitative assessment. First, unit tests will be used on code samples to give as close to 100% coverage as possible. Using the Stepdown rule discussed in Chapter Two, the same code will then be refactored, and the code coverage metrics will be regenerated to ensure that same set of unit tests provide the same code coverage regardless of number of private methods contained in the class. This same code will also be evaluated to determine if there has been an increase or decrease in any of the code metrics of the class. This will provide an initial basis to determine if testing code, refactored into the step-down style, causes that code to become more difficult to test.

The second part of this quantitative section will be to re-examine a selection of the scatter plots seen in the previous chapter. In addition, classes will be selected from differing areas of the scatter plots to see if those along the trend lines exhibit the type of characteristics that are expected from a class that consists of both high Cyclomatic Complexity and a high number of private methods.

In order to perform a qualitative assessment, interviews will be conducted with people with a background in information technology. This will allow some insights into what common knowledge exists with regard to code metrics and also attempt to determine if applying the Stepdown rule does in fact make code more readable.

6.2 Tools

In order to explain the evaluation via unit testing, first the concept of unit testing and framework used to write unit tests will be discussed.

6.2.1 NUnit

NUnit is an open source project that provides the ability to unit test C# code, the process of taking small segments of code and testing that each line of the code performs as expected. For example, the segment may be a class and within that class is an IF-statement. The purpose of unit testing is to ensure that the IF-statement performs correctly for all scenarios and therefore while one test would ensure the correct behaviour when the IF-statement returns true, a second test would be employed to ensure the correct behaviour when the IF-statement returns false. Essentially unit-testing is a form of white-box testing that tests the code at a very low level i.e. ensuring the correct functionality of an individual IF-statement over more general or black-box testing that may test something more general such as the logging in functionality of a website.

6.2.2 Code Coverage

As unit testing has become more popular the concept of being able to verify that all code written within a project has become more important. For example, a project may have ten classes and out of those ten classes, there are five methods in each. Given that each class and each method can have different levels of complexity there is no rule that can determine exactly how many unit tests are required to fully test those classes. It is possible to write over one hundred tests for the ten classes but that gives no guarantee that each and every scenario is covered.

For this reason, there are tools available called *Code Coverage tools*. When unit tests are run, these tools analyse all of the executed code to check what parts of the code ran and then calculate it as a percentage of the overall code under test. For example, the code coverage tool was analysing an IF-statement followed by an else statement and found that only the IF-statement was entered during the execution of the unit test, it would return that, 50% of the code was covered by that set of unit tests. This research will harness the code coverage functionality as built into Microsoft Visual Studio.

It will allow for verification that a set of unit tests have in fact covered a given piece of code. This in-turn provides a way to verify that a refactored code snippet has not increased or decreased in terms of testing complexity i.e. the same set of unit tests can be used to test the code post-refactor.

6.3 Unit Testing Evaluation

The code snippet below defines a class called `GeneratePrimes` that contains one method, `GeneratePrimeNumbers`. The snippet is long but is an example of code where all the logic is placed within one method with only some comments to help with understanding how the method is achieving its goal. By examining the unit tests it is clear that the `GeneratePrimeNumbers` method does meet the required functionality of the method. Although the code snippet below is quite long it is a great example of a method that while meeting its goal does so in fashion that is unfriendly to a reader that is not the original author. It is difficult to follow and with the exception of a few comments, no attempt was made to add readability.

```

4 references | 0 changes | 0 authors, 0 changes
public class GeneratePrimes
{
    4 references | 0 changes | 0 authors, 0 changes
    public static int[] GeneratePrimeNumbers(int maxValue)
    {
        if (maxValue >= 2) // the only valid case
        {
            // declarations
            int s = maxValue + 1; // size of array

            bool[] f = new bool[s];
            int i;
            // initialize array to true.
            for (i = 0; i < s; i++)
                f[i] = true;
            // get rid of known non-primes
            f[0] = f[1] = false;
            // sieve
            int j;
            for (i = 2; i < Math.Sqrt(s) + 1; i++)
            {
                if (f[i]) // if i is uncrossed, cross its multiples
                {
                    for (j = 2 * i; j < s; j += i)
                        f[j] = false; // multiple is not prime
                }
            }
            // how many primes are there?
            int count = 0;
            for (i = 0; i < s; i++)
            {
                if (f[i])
                    count++; // bump count.
            }
            int[] primes = new int[count];
            // move the primes into the result
            for (i = 0, j = 0; i < s; i++)
            {
                if (f[i]) // if prime
                    primes[j++] = i;
            }
            return primes; // return the primes
        }
        else // maxValue < 2
            return new int[0]; // return null array if bad input.
    }
}

```

Figure 6.1: Definition of GeneratePrimes Class

The unit tests shown in the code snippet below provide the adequate code coverage for the GeneratePrimes class. A good practice for unit tests is only to place one test per class and although that is not followed in this example it still suffices, as the tests are relatively simple, calling GeneratePrimeNumbers a total of four times and asserting on the result a total of eight times.

```

[Test]
public void TestGeneratePrimes()
{
    int[] nullArray = GeneratePrimes.GeneratePrimeNumbers(0);
    Assert.AreEqual(nullArray.Length, 0);
    int[] minArray = GeneratePrimes.GeneratePrimeNumbers(2);
    Assert.AreEqual(minArray.Length, 1);
    Assert.AreEqual(minArray[0], 2);
    int[] threeArray = GeneratePrimes.GeneratePrimeNumbers(3);
    Assert.AreEqual(threeArray.Length, 2);
    Assert.AreEqual(threeArray[0], 2);
    Assert.AreEqual(threeArray[1], 3);
    int[] centArray = GeneratePrimes.GeneratePrimeNumbers(100);
    Assert.AreEqual(centArray.Length, 25);
    Assert.AreEqual(centArray[24], 97);
}

```

Figure 6.2: Definition of GeneratePrimes Class Unit Tests

By applying the stepdown rule, the code in `GeneratePrimes` was refactored into a new class called `PrimeGenerator`. Changing the name just allows for a distinction to be made during this walk-through but suffice that both classes provide the method `GeneratePrimeNumbers`.

At first glance the most notable feature of the `PrimeGenerator` class, the refactored version of `GeneratePrimes`, is how small the one public method has become. Without reading the rest of the class that has been extracted into multiple private methods, the steps taken to achieve the results are much more clear. The method works by checking that the value of `maxValue` is below two. If that proves false, then it executes the other three methods, `UncrossIntegersUpTo`, `CrossOutMultiples` and `PutUncrossedIntegersIntoResult`.

```
private static bool[] crossedOut;
private static int[] result;

4 references | 0 changes | 0 authors, 0 changes
public static int[] GeneratePrimeNumbers(int maxValue)
{
    if (maxValue < 2)
        return new int[0];
    else
    {
        UncrossIntegersUpTo(maxValue);
        CrossOutMultiples();
        PutUncrossedIntegersIntoResult();
        return result;
    }
}
```

Figure 6.3: Definition of Refactored `GeneratePrimeNumbers` Method

It should be noted that the functionality of these code snippets, while interesting are not the main focus. The readability of the code is the main focus point. Can someone, having seen the class for the first time, quickly determine what the code is doing?

Taking a quick overview of the new private methods in the class, it can be seen that each method only performs one operation, for example it contains one for-loop or one math function. As shown in the example below `PutUncrossedIntegersIntoResult` was required to calculate the number of uncrossed integers, the functionality of which was further extracted into a new method directly below it called `NumberOfUncrossedIntegers`.

The remaining parts of `PrimeGenerator` are shown in the code snippets below.

```

1 reference | 0 changes | 0 authors, 0 changes
private static void UncrossIntegersUpTo(int maxValue)
{
    crossedOut = new bool[maxValue + 1];
    for (int i = 2; i < crossedOut.Length; i++)
        crossedOut[i] = false;
}

1 reference | 0 changes | 0 authors, 0 changes
private static void PutUncrossedIntegersIntoResult()
{
    result = new int[NumberOfUncrossedIntegers()];
    for (int j = 0, i = 2; i < crossedOut.Length; i++)
    {
        if (NotCrossed(i))
            result[j++] = i;
    }
}

1 reference | 0 changes | 0 authors, 0 changes
private static int NumberOfUncrossedIntegers()
{
    int count = 0;
    for (int i = 2; i < crossedOut.Length; i++)
    {
        if (NotCrossed(i))
            count++; // bump count.
    }
    return count;
}

1 reference | 0 changes | 0 authors, 0 changes
private static void CrossOutMultiples()
{
    int limit = DetermineIterationLimit();
    for (int i = 2; i <= limit; i++)
    {
        if (NotCrossed(i))
            CrossOutputMultiplesOf(i);
    }
}

1 reference | 0 changes | 0 authors, 0 changes
private static int DetermineIterationLimit()
{
    // Every multiple in the array has a prime factor that
    // is less than or equal to the root of the array size
    // so we don't have to cross off multiples of numbers
    // larger than that root.
    double iterationLimit = Math.Sqrt(crossedOut.Length);
    return (int)iterationLimit;
}

1 reference | 0 changes | 0 authors, 0 changes
private static void CrossOutputMultiplesOf(int i)
{
    for (int multiple = 2 * i;
         multiple < crossedOut.Length;
         multiple += i)
        crossedOut[multiple] = true;
}

3 references | 0 changes | 0 authors, 0 changes
private static bool NotCrossed(int i)
{
    return crossedOut[i] == false;
}

```

Figure 6.4: Definition of Private Methods used by GeneratePrimeNumbers

Now there are two versions of the GeneratePrimeNumbers method, the original GeneratePrimes that put all functionality into one public method, and the PrimeGenerator class, that broke the main public method into simple steps, with each part contained in a private method.

At this point, it needs to be determined if this refactoring has impacted the testability of the class in any way. To achieve this, the same set of unit tests seen earlier were applied to both the original implementation and newly refactored class. The code snippet below shows two unit tests, the first of which was shown earlier and the second one that is an exact copy of the first but tests the newly refactored class PrimeGenerator.

```

[Test]
0 references | 0 changes | 0 authors, 0 changes
public void TestGeneratePrimes()
{
    int[] nullArray = GeneratePrimes.GeneratePrimeNumbers(0);
    Assert.AreEqual(nullArray.Length, 0);
    int[] minArray = GeneratePrimes.GeneratePrimeNumbers(2);
    Assert.AreEqual(minArray.Length, 1);
    Assert.AreEqual(minArray[0], 2);
    int[] threeArray = GeneratePrimes.GeneratePrimeNumbers(3);
    Assert.AreEqual(threeArray.Length, 2);
    Assert.AreEqual(threeArray[0], 2);
    Assert.AreEqual(threeArray[1], 3);
    int[] centArray = GeneratePrimes.GeneratePrimeNumbers(100);
    Assert.AreEqual(centArray.Length, 25);
    Assert.AreEqual(centArray[24], 97);
}

[Test]
0 references | 0 changes | 0 authors, 0 changes
public void TestPrimeGenerator()
{
    int[] nullArray = PrimeGenerator.GeneratePrimeNumbers(0);
    Assert.AreEqual(nullArray.Length, 0);
    int[] minArray = PrimeGenerator.GeneratePrimeNumbers(2);
    Assert.AreEqual(minArray.Length, 1);
    Assert.AreEqual(minArray[0], 2);
    int[] threeArray = PrimeGenerator.GeneratePrimeNumbers(3);
    Assert.AreEqual(threeArray.Length, 2);
    Assert.AreEqual(threeArray[0], 2);
    Assert.AreEqual(threeArray[1], 3);
    int[] centArray = PrimeGenerator.GeneratePrimeNumbers(100);
    Assert.AreEqual(centArray.Length, 25);
    Assert.AreEqual(centArray[24], 97);
}

```

Figure 6.5: Definition of Unit Tests for both GeneratePrimes and PrimeGenerator

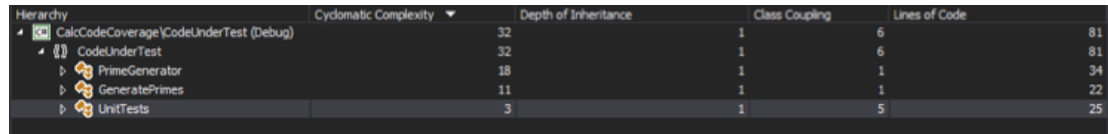
After running each set of tests against each class, the code coverage tool was used to determine if both the classes were in fact covered to the same extent regardless of the fact that the newly refactored class had multiple private methods added to it in an effort to apply the stepdown rule, making the class more readable.

Type to search		Coverage (%)	Uncovered/Total Stmts.
Symbol			
▾ Total		100%	0/131
▾ CalcCodeCoverage		100%	0/131
▾ CodeUnderTest		100%	0/131
▾ CodeUnderTest		100%	0/131
▾ UnitTests		100%	0/28
TestGeneratePrimes()		100%	0/14
TestPrimeGenerator()		100%	0/14
▾ GeneratePrimes		100%	0/42
GeneratePrimeNumbers(int)		100%	0/42
▾ PrimeGenerator		100%	0/61
NotCrossed(int)		100%	0/3
DetermineIterationLimit()		100%	0/4
CrossOutputMultiplesOf(int)		100%	0/6
UncrossIntegersUpTo(int)		100%	0/7
GeneratePrimeNumbers(int)		100%	0/9
CrossOutMultiples()		100%	0/10
PutUncrossedIntegersIntoResult()		100%	0/11
NumberOfUncrossedIntegers()		100%	0/11

Figure 6.6: Code Coverage results for both GeneratePrimes and PrimeGenerator

As shown above the code coverage for each class was still 100%. This indicates that the same set of tests provided the identical amount of testing coverage regardless of the number of private methods added in order to improve the overall readability of the class.

In addition to this, the code metrics were also generated for the code in order to determine if the refactoring of the code did indeed have an impact on the overall metrics.



Hierarchy	Cyclomatic Complexity	Depth of Inheritance	Class Coupling	Lines of Code
CalcCodeCoverage\CodeUnderTest (Debug)	32	1	6	81
CodeUnderTest	32	1	6	81
PrimeGenerator	18	1	1	34
GeneratePrimes	11	1	1	22
UnitTests	3	1	5	25

Figure 6.7: Code Metrics results for both GeneratePrimes and PrimeGenerator

As shown in the figure above both depth of inheritance and class coupling remained the same, as did lines of code. The notable difference though is the increase in the Cyclomatic Complexity for the newly refactored and arguably more readable PrimeGenerator class. The Cyclomatic Complexity of the PrimeGenerator class is eighteen where it was eleven on the less readable GeneratePrimes class.

6.4 Evaluating Merged Data

The previous chapter presented scatter plots generated from the merged data sets of the original metrics from Chapter Four merged with the newly generated data of Chapter Five. This in turn allowed for new scatter plots to be created where the number of both public and private methods was pitted against the metric data of Chapter Four. This found that there was a correlation between the number of private methods and both Cyclomatic Complexity and lines of code. This section will dig deeper into this finding and look at code snippets in an attempt to evaluate this finding.

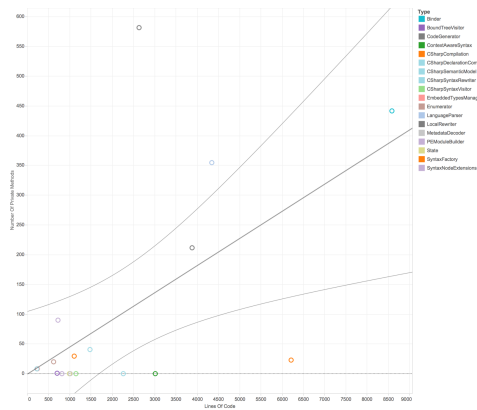


Figure 6.8: Number of Private Methods and Lines of Code (A)

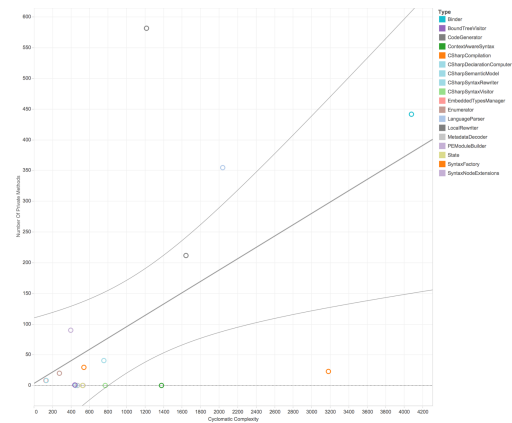


Figure 6.9: Number of Private Methods and Cyclomatic Complexity (B)

As shown in the two scatter plots above there appears to be a correlation between number of private methods and both Cyclomatic Complexity and lines of code.

In order to evaluate this further three specific classes were taken from the scatter plots above and examined in detail. These are the `Binder`, `CSharpCompilation` and `CodeGenerator` classes. It is very important to note that these classes were chosen on the basis of where they lie on the scatter plot. Both scatter plots indicate that the `Binder` class falls right on trend line whereas `CSharpCompilation` has higher lines of code and Cyclomatic Complexity hence lands on the bottom right of the scatter plot. Lastly and in direct contrast to `CSharpCompilation`, `CodeGenerator` lies on the top left of both scatter plots. This indicates that while it has a high number of private methods, it does not have high metrics regarding lines of code and Cyclomatic Complexity.

This provides three classes that lie on different points of the spectrum. It is important to note that of the three classes, the findings of this research to date indicate that `Binder` should be the most readable of the three classes. This is due to the `Binder` class lying right on the trend line of private methods to both lines of code and Cyclomatic Complexity whereas the other classes lie to opposing extremes outside of the trend lines.

6.4.1 Binder Class

The figures are code snippets taken from the `Binder` class within the Roslyn solution.

```
internal partial class Binder
{
    internal CSharpCompilation Compilation { get; }
    private readonly Binder _next;

    internal readonly BinderFlags Flags;

    /// <summary>
    /// Used to create a root binder.
    /// </summary>
    internal Binder(CSharpCompilation compilation)
    {
        Debug.Assert(compilation != null);
        this.Flags = compilation.Options.TopLevelBinderFlags;
        this.Compilation = compilation;
    }

    internal Binder(Binder next)
    {
        Debug.Assert(next != null);
        _next = next;
        this.Flags = next.Flags;
        this.Compilation = next.Compilation;
    }

    protected Binder(Binder next, BinderFlags flags)
    {
        Debug.Assert(next != null);
        // Mutually exclusive.
        Debug.Assert(!flags.Includes(BinderFlags.UncheckedRegion | BinderFlags.CheckedRegion));
        // Implied.
        Debug.Assert(!flags.Includes(BinderFlags.InWettedFinallyBlock) || flags.Includes(BinderFlags.InFinallyBlock | BinderFlags.InCatchBlock));
        _next = next;
        this.Flags = flags;
        this.Compilation = next.Compilation;
    }
}
```

Figure 6.10: Binder Code Snippet One

```
internal bool IsSemanticModelBinder
{
    get
    {
        return this.Flags.Includes(BinderFlags.SemanticModel);
    }
}

// IsEarlyAttributeBinder is called relatively frequently so we want fast code here.
internal bool IsEarlyAttributeBinder
{
    get
    {
        return this.Flags.Includes(BinderFlags.EarlyAttributeBinding);
    }
}

// Return the nearest enclosing node being bound as a nameof(...) argument, if any, or null if none.
protected virtual SyntaxNode EnclosingNameofArgument => null;

/// <summary>
/// Get the next binder in which to look up a name, if not found by this binder.
/// </summary>
internal protected Binder Next
{
    get
    {
        return _next;
    }
}
```

Figure 6.11: Binder Code Snippet Two

```

internal bool CheckOverflowAtCompileTime
{
    get
    {
        return CheckOverflow != OverflowChecks.Disabled;
    }
}

/// <summary>
/// Some nodes have special binder's for their contents (like Block's)
/// </summary>
internal virtual Binder GetBinder(CSharpSyntaxNode node)
{
    return this.Next.GetBinder(node);
}

/// <summary>
/// Get locals declared immediately in scope represented by the node.
/// </summary>
internal virtual ImmutableArray<LocalSymbol> GetDeclaredLocalsForScope(CSharpSyntaxNode node)
{
    return this.Next.GetDeclaredLocalsForScope(node);
}

/// <summary>
/// The member containing the binding context. Note that for the purposes of the compiler,
/// a lambda expression is considered a "member" of its enclosing method, field, or lambda.
/// </summary>
internal virtual Symbol ContainingMemberOrLambda
{
    get
    {
        return Next.ContainingMemberOrLambda;
    }
}

```

Figure 6.12: Binder Code Snippet Three

As expected the `Binder` class comprises of many small private methods that on average consist of a one-line methods. As previously argued this makes the class more readable.

6.4.2 CodeGenerator Class

The code snippet below is taken from the code generator class. As the code generator class is at the top left of the scatter plot it indicates that it should consist of many private methods but is not balanced back out with the appropriate number of lines of code and Cyclomatic Complexity that make it readable.

This assumption is confirmed by examining the code snippet below that defines a method called `LazyReturnTemp`. It shows that unlike the small methods seen in the `Binder` class previously this method is much longer and therefore arguably less readable.

```

private LocalDefinition LazyReturnTemp
{
    get
    {
        var result = _returnTemp;
        if (result == null)
        {
            Debug.Assert(!_method.ReturnsVoid, "returning something from void method?");

            var bodySyntax = _methodBodySyntaxOpt;
            if (_ifEmitStyle == IfEmitStyle.Debug && bodySyntax != null)
            {
                int syntaxOffset = _method.CalculateLocalSyntaxOffset(bodySyntax.SpanStart, bodySyntax.SyntaxTree);
                var localSymbol = new SynthesizedLocal(_method, _method.ReturnType, SynthesizedLocalKind.FunctionReturnValue, bodySyntax)
                {
                    result = _builder.LocalSlotManager.DeclareLocal(
                        type: _module.Translate(localSymbol.Type, bodySyntax, _diagnostics),
                        symbol: localSymbol,
                        name: null,
                        kind: localSymbol.SynthesizedKind,
                        id: new LocalDebugId(syntaxOffset, ordinal: 0),
                        pdbAttributes: localSymbol.SynthesizedKind.PdbAttributes(),
                        constraints: localSlotConstraints.None,
                        isDynamic: false,
                        dynamicTransformFlags: ImmutableArray<TypedConstant>.Empty,
                        isSlotReusable: false);
                }
            }
            else
            {
                result = AllocateTemp(_method.ReturnType, _boundBody.Syntax);
            }
            _returnTemp = result;
        }
        return result;
    }
}

```

Figure 6.13: CodeGenerator Code Snippet

6.4.3 CSharpCompilation Class

As the CSharpCompilation class lies on the bottom right of the scatter plot it indicates that the class has a high number of lines of code and Cyclomatic Complexity. However, it does not consist of a high number of private methods. This leads to the conclusion that the class may be arguably less readable.

On examining the class in the code snippet below it is evident that it is composed of large portions of complex code.

```

private CSharpCompilation(
    string assemblyName,
    CSharpCompilationOptions options,
    ImmutableArray<MetadataReference> references,
    CSharpCompilation previousSubmission,
    Type submissionReturnType,
    Type hostObjectType,
    bool isSubmission,
    ReferenceManager referenceManager,
    bool reuseReferenceManager,
    SyntaxAndDeclarationManager syntaxAndDeclarations,
    AsyncQueue<CompilationEvent> eventQueue = null)
    : base(assemblyName, references, SyntaxTreeCommonFeatures(syntaxAndDeclarations.ExternalSyntaxTrees), isSubmission, eventQueue)
{
    _wellKnownMemberSignatureComparer = new WellKnownMembersSignatureComparer(this);
    _options = options;

    this.builtInOperators = new BuiltInOperators(this);
    _scriptClass = new Lazy<ImplicitNamedTypeSymbol>(BindScriptClass);
    _globalImports = new Lazy<Imports>(BindGlobalImports);
    _previousSubmissionImports = new Lazy<Imports>(ExpandPreviousSubmissionImports);
    _globalNamespaceAlias = new Lazy<AliasSymbol>(CreateGlobalNamespaceAlias);
    _anonymousTypeManager = new AnonymousTypeManager(this);
    This.LanguageVersion = CommonLanguageVersion(syntaxAndDeclarations.ExternalSyntaxTrees);

    if (isSubmission)
    {
        Debug.Assert(previousSubmission == null || previousSubmission.HostObjectType == hostObjectType);
        this.ScriptCompilationInfo = new CSharpScriptCompilationInfo(previousSubmission, submissionReturnType, hostObjectType);
    }
    else
    {
        Debug.Assert(previousSubmission == null && submissionReturnType == null && hostObjectType == null);
    }

    if (reuseReferenceManager)
    {
        referenceManager.AssertCanReuseForCompilation(this);
        _referenceManager = referenceManager;
    }
    else
    {
        _referenceManager = new ReferenceManager(
            MakeSourceAssemblySimpleName(),
            this.Options.AssemblyIdentityComparer,
            observedMetadata: referenceManager.ObservedMetadata);
    }

    _syntaxAndDeclarations = syntaxAndDeclarations;

    Debug.Assert((object)_lazyAssemblySymbol == null);
    if (eventQueue != null) EventQueue.TryEnqueue(new CompilationStartedEvent(this));
}

```

Figure 6.14: CSharpCompilation Code Snippet

Overall it can be concluded from this section there is in fact evidence that when the number of private methods within a class is cross referenced with either lines of code and/either Cyclomatic Complexity that it can provide an indication as to the readability of that class.

6.5 Summary of Key Quantitative Findings

Section 6.3 above harnessed unit testing, code coverage tools and metrics calculations in an effort to evaluate the impact of refactoring code to a more readable format. There were two very notable findings. The first was the fact that the unit tests used with 100% code coverage provided the same level of coverage post the refactoring of the code. That means, code can be refactored into a more readable format without altering the unit tests that cover that code snippet. This also indicates that the code has not become more difficult to test, as the unit tests are identical. Secondly, although the unit tests indicated there was no increase in the level of testing difficulty, the Cyclomatic Complexity of the code increased. Therefore, while Cyclomatic Complexity is showing an increase due to a readability refactor, the code coverage and unit tests are suggesting there has been no increase in complexity.

Section 6.4 dug a little deeper into the results of the metrics versus public and private methods of Chapter Five. It found that classes that fell along the trend line of the scatter plot comparing either lines of code or Cyclomatic Complexity to number of private methods in a type, appeared more likely to exhibit characteristics that are consistent with more readable code. That is, code that consists of small private methods that focus on doing one thing as opposed to methods that run to five-plus lines of code with large amounts of complexity. This was demonstrated by selecting three classes, one from the top left of the scatter plot, one from the bottom right of the scatter plot and that fell right into the trend line of the scatter plot. Only the class along the trend line consisted of the characteristics consistent with readable code.

6.6 Qualitative Evaluation of Code Readability

In an effort to determine the qualitative element of code readability, surveys were conducted on a one-to-one basis with five participants with a background in the technology sector. This section will outline the profiles of the interviewees and the results garnered from the five participants.

6.6.1 Interviewee Profiles

Interviewee #1	Interviewee #1 is male who has 5+ years' experience in the IT sector and works as a team leader. Day to day duties do not involve coding but is involved with people that develop software
Interviewee #2	Interviewee #2 is male who has 1+ years' experience in the IT sector and works as a QA Engineer. He familiar with Unit Testing and Manual Testing
Interviewee #3	Interviewee #3 is female who has 2+ years' experience in the IT sector and works as a manual QA Engineer. She is familiar with Manual Testing
Interviewee #4	Interviewee #4 is female who has 4+ years' experience in the IT sector and works as a software engineer. She is familiar with Unit Testing and Manual Testing
Interviewee #5	Interviewee #5 is male who has 2+ years' experience in the IT sector and works as a software engineer. He is familiar with Unit Testing and Manual Testing

Figure 6.15: Interviewee profiles overview

Each interview began with questions around the person's background. Figure 6.14 below shows that while three out of the five people had worked in the broad category of information technology sector for a period of less than three years, only two had more than three years.

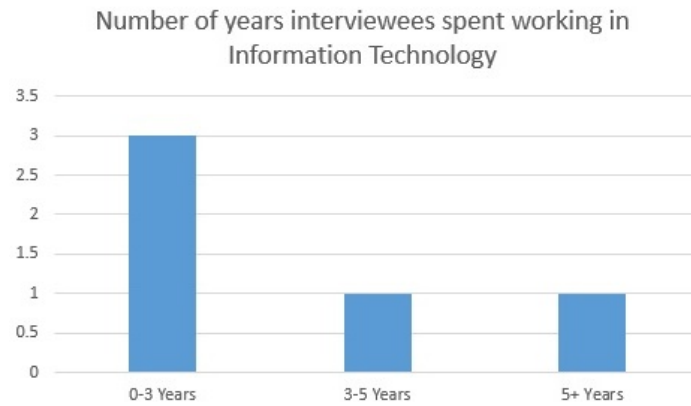


Figure 6.16: Number of years interviewees spent working in Information Technology

Next, each interviewee was asked specifically which discipline area they fell into. This information is represented in the table below. Given a list of five job categories each was asked which matched them best. Four of the five people worked as either a software engineer or a quality assurance engineer.

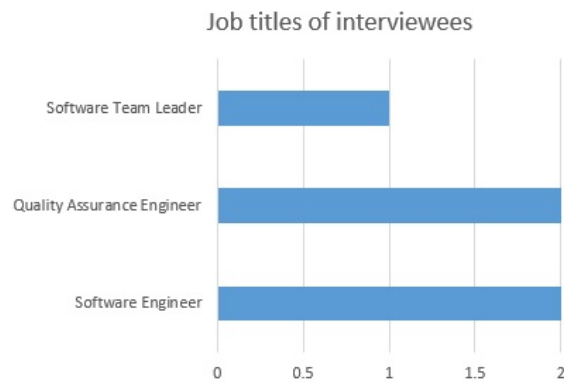


Figure 6.17: Job titles of interviewees

Following that, each interviewee was asked to rate their daily interaction with code from never, meaning that they never see or interact with code as part of their work to always as in a full time software developer. The results of this are represented in the table below.

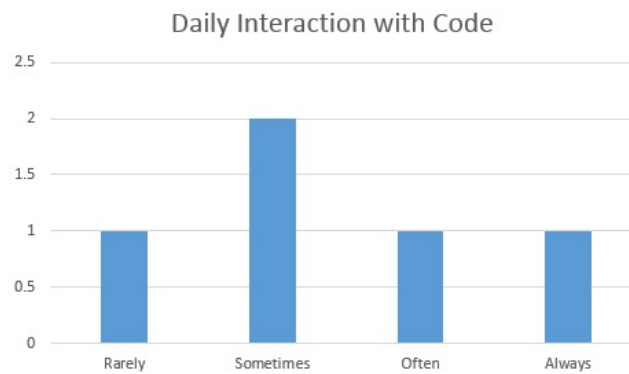


Figure 6.18: Daily interaction with code of interviewees

Finally, each interviewee was asked about how the code within their workplace was tested. For this question, interviewees could select more than one option and were also asked to indicate if it was not applicable for those who work in companies that do not test software with a formal process. The results are presented in the table below.

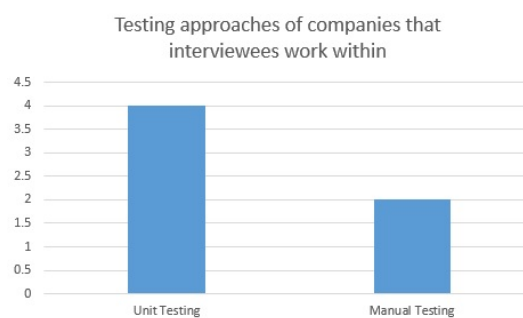


Figure 6.19: Testing approaches of companies that interviewees work within

The results here are interesting for two reasons, the first being that unit testing is clearly quite prevalent over all other forms of testing including manual testing and secondly that no one interviewed performed automated testing that was not unit testing.

This provides enough information to determine that each of the five have enough knowledge of software development to provide relevant data to this research.

6.6.2 Software Terminology

The next part of the interview focused on determining what terminology each person was familiar with. This provides this research with insight into what people working in

the industry are familiar with as opposed to the terminology used by computer scientists conducting research in the area.

The first question each interviewee was asked was if they ever used code metrics in the development of software. The result was an emphatic “No”. This would indicate that regardless of how commonly code metrics are used in within industry, they are not necessarily referred to as code metrics.

In an effort to dig deeper, each interviewee was asked if they were familiar with a list of software engineering terms. The interviewees were asked to input this information into a survey. It should be noted that not all of the terms were code metric specific. The results are presented below.

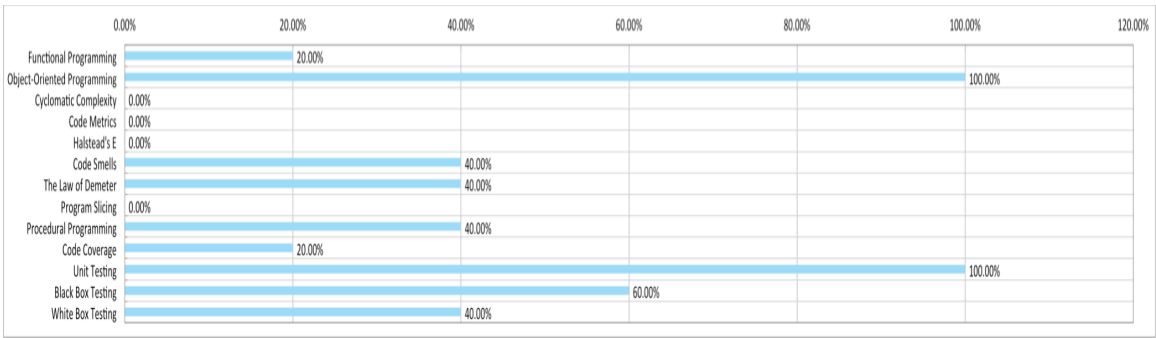


Figure 6.20: Common software related terms interviewees were familiar with

The two most notable results, when all the interviewee data is merged together, is how terms such as object-oriented programming and unit-testing are familiar to all five whereas Cyclomatic Complexity and code metrics were not familiar to anyone.

6.6.3 Code Readability

The last part of the interview centred on code readability and was conducted by showing each interviewee four code snippets. The code snippets were two code samples written in two different ways. The first paid no attention to the ‘readability’ of the code while the second applied the Stepdown rule as defined by Martin (2008). It is important to note that the interviewee was only asked to rate the code on a scale from poor to Excellent for ‘readability’ and no explanation was provided as to why the code was written differently. The code snippets were also presented in random order so the

interviewee was not necessarily asked to rate the less readable code before the more readable code so as to prevent any bias. The code snippets used are taken from Martin (2008).

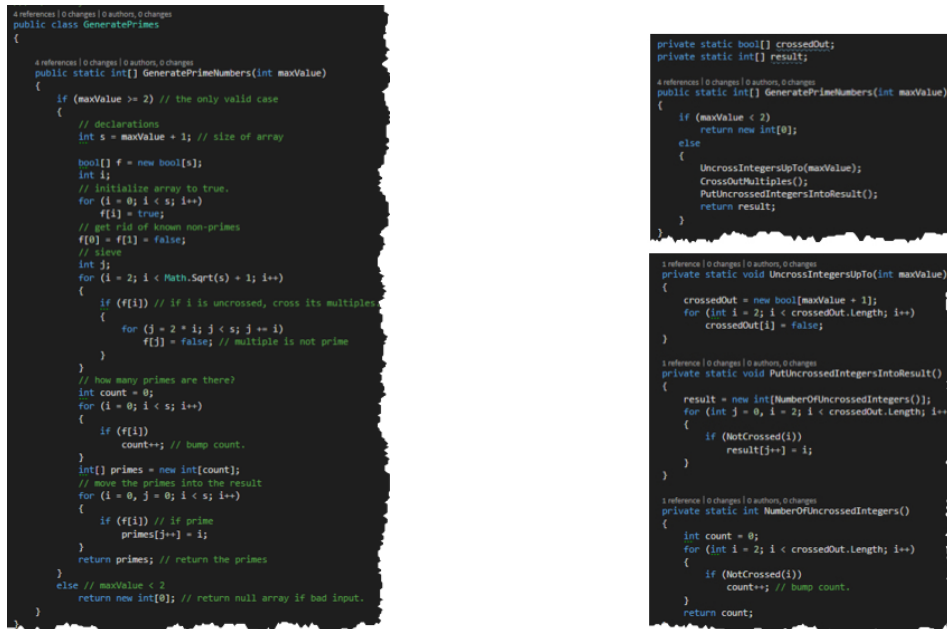


Figure 6.21: Code snippets presented to interviewees to be rate for readability

From the code snippets in Figure 6.20, rated by the five interviewees from poor to Excellent, snippet one was rated poor by one interviewee while the remaining four rated it either below average or average. By comparison snippet two was not rated poor by any interviewee and scored either average or above average by the interviewees.

```

public List<Cell> getFlaggedCells() {
    List<Cell> flaggedCells = new ArrayList<Cell>();
    for (Cell cell : gameBoard)
        if (cell.isFlagged())
            flaggedCells.add(cell);
    return flaggedCells;
}

public List<int[]> getThem() {
    List<int[]> list1 = new ArrayList<int[]>();
    for (int[] x : theList)
        if (x[0] == 4)
            list1.add(x);
    return list1;
}

```

Figure 6.22: Code snippets presented to interviewees to be rate for readability

The code snippets in Figure 6.21 were rated in the same manner as the previous. While snippet two rate below average by three of the four interviewees, two interviewees rated it only average. In comparison, snippet one was rated below average by one interviewee, average by three of the interviewees and above average by one interviewee.

Although the results above do not indicate that a code snippet written with code readability in mind automatically lead to being rated from poor to Excellent there is an underlying suggestion that this code is in fact more readable. This is keeping with De Silva *et al.* (2012) where opinions on code was taken from thirty programmers and compared against metrics, including McCabe’s Cyclomatic Complexity, which found a wide range of opinions. For example, the collected “experts’ rankings” for BubbleSort.java is shown below.

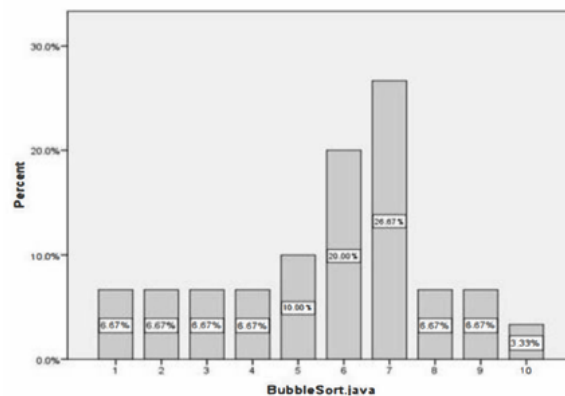


Figure 6.23: Experts view of complexity of BubbleSort.java

Although over 50% of the developers found BubbleSort.java to have a complexity measure in the range of 6 to 7, it is notable that over 12% found it to be in the range of 1 to 2 while a further 9% places it over 9 (De Silva *et al.*, 2012). This highlights how diverse opinions can be when reading code and hence how difficult it is to evaluate a piece of code as readable.

6.7 Summary of Key Qualitative Findings

The qualitative assessment took the approach of interviewing people within the information technology industry. Prior to assessing code readability using code snippets the interviews looked to gain some insight into what terminology people within the industry were familiar with, with specifically around the area of code metrics.

It found that no interviewee had previously used code metrics within industry or at least if they had done, they were not referring to them as code metrics. Terms such as ‘code metrics’ or ‘Cyclomatic Complexity’ did not feature at all. In addition, it was found that the most common way of testing code within the companies the participants were associated with was unit testing and manual testing.

Finally, the participants were asked to examine four code snippets. The code snippets were either written with a readability factor applied or not. This found that while there was no surge in overall rating of the code, there was a very slight improvement on the readability rating of code that was crafted with readability in mind. That concurred with De Silva *et al.* (2012) where assessments on the complexity of java programs had consensus of around 50%, there were still wide swings in terms of what is perceived by one person as complex versus another’s opinion.

6.8 Conclusions

This chapter looked to evaluate some of the findings from Chapter Five. In doing so it looked at the findings from both a quantitative and qualitative point of view.

The unit testing section of the quantitative aspect of the evaluation had two key findings. First that code refactored using the Stepdown rule could be tested using the same set of unit tests as the non-refactored code and hence means the refactored class is not more complex to test. Secondly, it found that the Cyclomatic Complexity of the class did in fact increase for the refactored class.

The second part of the quantitative assessment reviewed the previously seen scatter plots. On close examining of code snippets taken from classes lying in different area of

the scatter plot it was determined that a class found along the trend line i.e. had a balance of Cyclomatic Complexity or lines of code versus the number of private methods, had characteristics consistent with a readable structure. Classes that lay out to either extreme did not appear to these same characteristics.

On the qualitative side of the assessment the main findings were that none of the interviewees were familiar with code metrics or at least referred to them as code metrics. While there was a slight increase in the rating of code readability that followed the Stepdown rule, it was no overwhelming. This goes back to De Silva *et al.* (2012) who also had large swings in opinions when asking developers to rate code for complexity.

7. Conclusions and Future Work

7.1 Introduction

This chapter will take a look back over some of the key findings from of this research. It will begin by looking at the Chapters Two and Three to identify notable points found during the literature review. It will then proceed to Chapters Four and look to review how the data exploration threw up new insights when examining the relationships between various metrics and looking at sample code for causation effects. It will then continue to Chapter Five where a key finding of Chapter Four was examined in great depth. It will then take another look at the key findings when evaluating Chapter Six. The chapter will conclude by looking at some new areas that this research can be taken by outlining some possible future directions.

7.2 Conclusions

This section presents the key conclusions derived from each of the main chapters of this dissertation.

7.2.1 Existing Literature

McCabe's Cyclomatic Complexity became a cornerstone on which many subsequent theories were built, either from a critical standpoint or using it as a basis to expand upon. Although it was first published 40 years ago and was based on the FORTRAN programming language, it still remains relevant post the paradigm shift to object-oriented programming.

Chidamber & Kemerer's suite of metrics for object-oriented programming is arguably the most important paper written in this area. Covering all of the important aspects of that need to be considered when developing code in the object-oriented paradigm. This suite formed the basis for many applications built within industry as well as being a foundation for further work in the area.

For many decades code metrics revolved around the area of identifying code that is at high risk of being defective. By introducing the concept of detecting ‘code smells’ within object-oriented programming, van Emden (2002) brought a new lens through which code metrics can be viewed. It opened up code metrics to being adopted for use in an area that was previously reserved for manual code reviews where programmers looked to weed out bad practice with predefined principles. This was now something that could be automated, and used to highlight areas of that code that although not at high risk of being defective, but considered to have some potential flaws due to poor practices.

Purposely defined to be as a simple language independent rule to make the applying of the concepts of modularity and encapsulation more intuitive for developers, the Law of Demeter has become an industry standard. It ensures that developers can reduce the coupling of classes without even having to consider the area of metrics let alone develop systems that detect or identify it.

Building on van Emden (2002) this paper introduced Marin’s (2008) ‘clean code’ concept to the area of code metrics in an effort to assess what impact would result from applying these concepts to the area of code metrics.

Chapter Three had a key focus on how metrics have been applied within industry and found that companies including Hewlett-Packard (HP) and Air France-KLM had varying degrees of success when using code metrics within house, in an effort to identify code that was at high risk of being defective. While it cannot be claimed that there were no improvements when implementing code metrics, there did appear to be a considerable amount of complexity involved to implement something that was only returning minor gains.

7.2.2 Data Exploration

The data exploration phase of this research took the form of generating a new dataset of metrics from a well-established open source project named Roslyn. This allowed for fresh insights to be sought through examination of the data in the form of visualisations. Flipped bar charts were used initially to identify areas of the project that

could be considered to have high, average and low metric scores. These areas were then explored in greater depth to see if any relationships between the metrics could be identified and then determine if these relationships held through from project to project regardless of overall metric score.

Overall Cyclomatic Complexity and Class Coupling appear to show some correlation but were very much dependent on the overall score of the project. On experiments when the project overall scored highly, the correlation here was stronger whereas at the other extreme low end of the scale the correlation fell away.

Arguably the most consistent relationship that appeared in each phase of the data exploration was that between Cyclomatic Complexity and lines of code. As shown in the small multiples below, the relationship held strong regardless of the overall metrics score of that project.

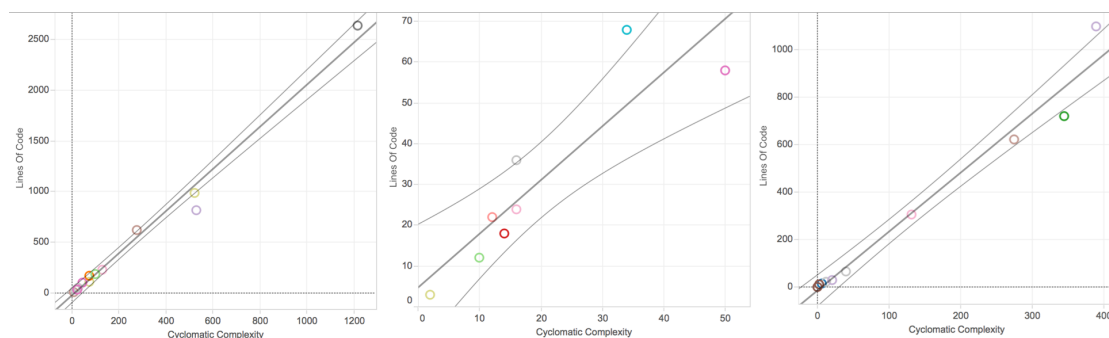


Figure 7.1: Small Multiples of Cyclomatic Complexity

Perhaps slightly surprisingly was the depth of inheritance metric that always failed to find any correlation with any other metric. For each of the comparisons during the data exploration phases the depth of inheritance metric failed, time after time, to show any significant relationship with any of the other metrics.

A few of the code samples taken during the data exploration phases indicated a relation between increases in Cyclomatic Complexity and the number of private methods in a class. By definition, Cyclomatic Complexity is concerned with the complexity of a given piece of code and therefore the impact on increases in the number of private

methods is an interesting discovery. It was this finding that became the focus of the subsequent chapter.

7.2.3 Impact of Code Readability on Metrics

One of the findings of the previous chapter was related to private methods appearing to have an impact on Cyclomatic Complexity, a metric that also tends to correlate well with the lines of code metric. It was this finding that became the focus of Chapter Five. It first sought to extract new data from the Roslyn project and merged it with the data from Chapter Four. It then proceeded to analyse possible relationships between all of the various metrics paired against the number of public and private methods.

Unlike the metrics from the data exploration chapter, where tools were provided to extract the data from the Roslyn project, there are no readily available tools to extract method data from Visual Studio projects. Hence some code was written to compile this data with a view to merging it into the already gathered metric data. Using the .NET library class called Assembly the binaries of the Roslyn project were parsed and the data related to public and private methods within each type was extracted. This data was then formatted and saved to an Excel workbook.

Although the code successfully extracted the data, a process of preparing it to be merged into the existing data set was required in order to ensure that the merged data set could produce the required result.

Two of the most significant parts to this were to define how partial classes were to be handled and also to ensure the name of the types matched correctly. A partial class is a class that, although spilt into multiple files for more manageable human interaction, is a single class when compiled. This was an issue that the code was required to contend with. Instead of defining these in the data as multiple classes the code merged the files plus the relevant information about each into single classes and hence single lines with the Excel sheet. This in turn led to an issue regarding the naming of the types. As the code had combined all of the classes into a single class and named it, this did not allow for the names to merge with the already existing data set of metrics. This required a

second piece of code to be written that reformatted each type name to match. For example, names such as these

- MetadataDecoder<ModuleSymbol, TypeSymbol, MethodSymbol, FieldSymbol, Symbol>
- AbstractLookupSymbolsInfo<TSymbol>.UniqueSymbolOrArities
- CompilerDiagnosticAnalyzer.CompilationAnalyzer.CompilerDiagnostic

were reformatted to

- MetadataDecoder
- UniqueSymbolOrArities
- CompilerDiagnostic

This subsequently allowed for the type names to be merged successfully.

From this merged data set, more scatter plots were generated to identify possible relationships between class methods, whether public or private and the various metrics: class coupling, depth of inheritance, lines of code and Cyclomatic Complexity. This allowed for these metrics to be viewed from a different angle i.e. what relationship, if any, exists between these various metrics and the number of public and private methods of these same classes.

For the majority of the metrics there was little sign of a relationship between them and public and private methods with the notable exception of private methods and both Cyclomatic Complexity and lines of code. This was consistent with the findings of the previous chapter. This is shown in the table below:

	Class Coupling	Cyclomatic Complexity	Depth of Inheritance	Line of Codes
Public Methods	N	N	N	N
Private Methods	N	Y	N	Y

Figure 7.2: Overview of correlations between code metrics and public and private methods

In addition, it was also noted that this finding impacted on aspect of the previously introduced work of Martin (2008). By applying the Stepdown rule, as defined by Martin (2008), it would seem that making the class more ‘readable’ would in fact increase the number of private methods within the class and therefore cause an increase in the Cyclomatic Complexity of that class.

It was this key finding that would become the focus of the subsequent chapter. By employing both a quantitative and qualitative element to the evaluation to determine if code that is refactored to be more ‘readable’ actually impacts the Cyclomatic Complexity of that class.

7.2.4 Evaluation

Evaluating whether or not the application of the Stepdown rule, as defined by Martin (2008), increased the Cyclomatic Complexity of the same class, was the main focus of this chapter. In order to fully assess this, both a quantitative and qualitative element was required (a so-called Mixed Methods Approach). The qualitative element took the form of harnessing unit testing and code coverage tools in order to assess any increase in testing complexity by refactoring the class. It also re-examined scatter plots and code samples from the previous chapter. The qualitative element consisted of interviews with people working within information technology sector.

The first part of the quantitative assessment using unit testing and code coverage tools had two findings. The first was the fact that the unit tests written for a piece of code prior to it being refactored to be more ‘readable’, provided the same level of coverage to the refactored code. This would indicate that code refactored using the Stepdown rule is in fact no more complex to test. In addition, it also found that the refactored code did in fact increase in the overall Cyclomatic Complexity metric.

The second part of this quantitative assessment looked to examine classes taken from the various areas of the scatter plots created in Chapter Five. It took three classes and determined that a class that lay on the trend line i.e. had a correlation between number of private methods and Cyclomatic Complexity did in fact exhibit the characteristics of

a class that would be considered more ‘readable’ as per the Stepdown rule. That is, a class that lay along the trend line consisted of small private methods that each provided a single task as opposed to long and overly complex methods.

On the qualitative side of the assessment it was found that not many of the interviewees were familiar with terms associated with code metrics. On assessing whether code written using the Stepdown rule was in fact more ‘readable’ a slight increase was noted but nothing that could be considered evidentiary. This was in keeping with De Silva *et al.* (2012) who had wide ranging results when attempting to have experienced programmers evaluate code for complexity.

7.3 Future Work

This section will identify areas new directions that this research can be built upon. These include introducing new metrics that can be paired against the number of public and private methods in a class, quantitative evaluation using a different open source project or the introduction of a new programming paradigm such as functional programming.

7.3.1 Introducing New Metrics

As seen in Chapter Two and Three there are many more metrics that could be incorporated into this research. Not all metrics have readily available tools that extract data for close examination but as was discovered in Chapter Five many development platforms including .NET come with libraries that allow for this data to be extracted with a small amount of coding.

By harnessing these libraries to explore some of the less well-known metrics, new insights could be gained into overall area of code metrics. Potential new metrics could include:

- Coupling and Cohesion
- DSQI (design structure quality index)

- Instruction path length
- Maintainability index
- Weighted Micro Function Points
- CISQ automated quality characteristics measures

7.3.2 Alternative Open Source Solutions

With the advent of Github.com there is no shortage of open source projects available online. Many of the studies discussed in Chapter Three involved theories being applied within commercial companies. No longer is it required that commercial companies be involved. Open source solutions enable a huge amount of data to be extracted and examined within this area. Therefore by taking one small example as was done in Chapter Five and applying it to multiple solutions could lead to real evidence as to whether code metrics can provide value to a project.

7.3.3 Programming Paradigms

Object-oriented programming has been in favour since the 1990's. This has led to a lot of work around metrics also favouring object-oriented programming. With the high availability of new tools, including integrated development environments etc., enables more research to be conducted in the areas of functional programming, or aspect-oriented programming, or logic programming.

7.3.4 Community Evaluation

As well as code, GitHub also provides a community of interested participants who could be used to evaluate the readability of code. They could be used to discuss in more detail the characteristics of good code, and how they use metrics and which ones they prefer. By focusing on large-scale data gathering of this kind, new insights could be gained into what is being used in practice as opposed to within individual companies. It is also worth noting that most practices within commercial companies are never published publicly. This data could provide a basis from which new research could focus on metrics used in practice that could lead to new avenues or directions for code metrics.

7.3.5 Development Methodology

It could be worth exploring if the way in which the code was developed impacts the code readability and/or the metrics. If code is developed using Paired Programming that will demonstrate a measurable difference in terms of the metrics than if the code is developed using Scrum. Companies that employ commercial tools for the Scrum and Agile process contain a lot of data that could provide new insight into the overall impact of these practices. This could then be extended to see if these same differences occur when using a traditional waterfall model.

7.3.6 Design Patterns

The increasing use of Design Patterns means that more standardised and well-known solutions are being applied to programming problems, which may be improving the overall quality of code. It might be worth investigating a software project that uses a lot of design patterns to see if this impacts the metrics. This could be done in an automated way that creates heat maps of where the patterns appear in the code and then analyse metrics closely in that area. In addition, this data could then be compared to a project using no patterns in an effort to gain further insight into the impact of patterns on code metrics.



Figure 7.3: Illustration of how a heat map looks

7.3.7 Extracting Data Using Platform Libraries

In Chapter Five a .NET library was used to extract data relating the number of public and private methods of a class. It is worth noting that this library contained vast amounts of data relating to the binaries being examined. This data is not specifically

targeted at code metrics but there is no doubt that more data exists within these libraries that may shed new insights in the overall area of code metrics. Research could be undertaken that specially analysed all of the data that these libraries produce in an effort to go beyond what is already known about code metrics.

7.3.8 Open Source Testing

While sites like Github.com contain a vast amounts of open source software, Travis-CI.org is a self-described home of testing. Programmers using Github.com can harness Travis to automatically take their new code changes and run all the tests associated with that solution. Many of these solutions and test results are open source and therefore an opportunity exists to analyse common patterns of defects occurring and cross-referencing this with the code metrics of that same area of code. For example, if a particular class is identified as having consistently failing unit tests this could be compared against the Cyclomatic Complexity of that class. This could shed some new insights into which metrics provide the most value in identifying code that is at high risk of being defective.

BIBLIOGRAPHY

- Aho Alfred, V., Ravi, S., & Ullman Jeffrey, D. (1986). Compilers: principles, techniques, and tools. *Reading: Addison Wesley Publishing Company*.
- Al Dallal, J. (2009). Software similarity-based functional cohesion metric. *Software, IET*, 3(1), 46-57. doi:10.1049/iet-sen:20080054
- Anderson, J. L. (2004, 20-23 Sept. 2004). *Using software tools and metrics to produce better quality test software*. Paper presented at the AUTOTESTCON 2004. Proceedings.
- Balachandran, V. (2013, 18-26 May 2013). *Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation*. Paper presented at the Software Engineering (ICSE), 2013 35th International Conference on.
- Casalnuovo, C., Devanbu, P., Oliveira, A., Filkov, V., & Ray, B. (2015, 16-24 May 2015). *Assert Use in GitHub Projects*. Paper presented at the Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on.
- Chidamber, S. R., & Kemerer, C. F. (1991). Towards a metrics suite for object oriented design. *SIGPLAN Not.*, 26(11), 197-211. doi:10.1145/118014.117970
- Coleman, D., Ash, D., Lowther, B., & Oman, P. (1994). Using metrics to evaluate software system maintainability. *Computer*, 27(8), 44-49. doi:10.1109/2.303623
- Constantine, L. L. (1996, 24-27 Nov 1996). *Visual coherence and usability: a cohesion metric for assessing the quality of dialogue and screen designs*. Paper presented at the Computer-Human Interaction, 1996. Proceedings., Sixth Australian Conference on.
- Curtis, B., Sheppard, S. B., Milliman, P., Borst, M. A., & Love, T. (1979). Measuring the Psychological Complexity of Software Maintenance Tasks with the Halstead and McCabe Metrics. *Software Engineering, IEEE Transactions on, SE-5*(2), 96-104. doi:10.1109/TSE.1979.234165
- De Silva, D. I., Kodagoda, N., & Perera, H. (2012, 12-15 Dec. 2012). *Applicability of three complexity metrics*. Paper presented at the Advances in ICT for Emerging Regions (ICTer), 2012 International Conference on.
- Fagan, M. E. (1976). Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3), 182-211. doi:10.1147/sj.153.0182

- Gill, G. K., & Kemerer, C. F. (1991). Cyclomatic Complexity density and software maintenance productivity. *Software Engineering, IEEE Transactions on*, 17(12), 1284-1288. doi:10.1109/32.106988
- Grady, R. B. (1994). Successfully applying software metrics. *Computer*, 27(9), 18-25. doi:10.1109/2.312034
- Halstead, M. H. (1972). Natural laws controlling algorithm structure? *SIGPLAN Not.*, 7(2), 19-26. doi:10.1145/953363.953366
- Jacobson, I. (1992). *Object-oriented software engineering*: ACM.
- Kontogiannis, K. (1997, 6-8 Oct 1997). *Evaluation experiments on the detection of programming patterns using software metrics*. Paper presented at the Reverse Engineering, 1997. Proceedings of the Fourth Working Conference on.
- Lei, M., Cheng, Z., Bing, Y., & Sato, H. (2015, 23-24 May 2015). *An Empirical Study on Effects of Code Visibility on Code Coverage of Software Testing*. Paper presented at the Automation of Software Test (AST), 2015 IEEE/ACM 10th International Workshop on.
- Li, W., & Henry, S. (1993). Object-oriented metrics that predict maintainability. *Journal of systems and software*, 23(2), 111-122.
- Lieberherr, K., Holland, I., & Riel, A. (1988). Object-oriented programming: an objective sense of style. *SIGPLAN Not.*, 23(11), 323-334. doi:10.1145/62084.62113
- Lieberherr, K. J., & Riel, A. J. (1988, 11-15 Apr 1988). *Demeter: a case study of software growth through parameterized classes*. Paper presented at the Software Engineering, 1988., Proceedings of the 10th International Conference on.
- Martin, R. C. (2008). *Clean Code: A Handbook of Agile Software Craftsmanship*: Prentice Hall PTR.
- McCabe, T. (1976). A Complexity Measure. *Software Engineering, IEEE Transactions on*, SE-2(4), 308-320. doi:10.1109/TSE.1976.233837
- McCabe, T. (1996). Cyclomatic Complexity and the year 2000. *Software, IEEE*, 13(3), 115-117. doi:10.1109/52.493032
- Meneely, A., Smith, B., & Williams, L. (2013). Validating software metrics: A spectrum of philosophies. *ACM Trans. Softw. Eng. Methodol.*, 21(4), 1-28. doi:10.1145/2377656.2377661

- Meyer, B. (1988). *Object-oriented software construction* (Vol. 2): Prentice hall New York.
- Mordal-Manet, K., Laval, J., Ducasse, S., Anquetil, N., Balmas, F., Bellingard, F., . . . McCabe, T. J. (2011, 1-4 March 2011). *An Empirical Model for Continuous and Weighted Metric Aggregation*. Paper presented at the Software Maintenance and Reengineering (CSMR), 2011 15th European Conference on.
- Novak, J., Krajnc, A., & Zontar, R. (2010, 24-28 May 2010). *Taxonomy of static code analysis tools*. Paper presented at the MIPRO, 2010 Proceedings of the 33rd International Convention.
- Ordonez, M. J., & Haddad, H. M. (2008, 7-9 April 2008). *The State of Metrics in Software Industry*. Paper presented at the Information Technology: New Generations, 2008. ITNG 2008. Fifth International Conference on.
- Osherove, R. (2015). *The art of unit testing*: MITP-Verlags GmbH & Co. KG.
- Perepletchikov, M., Ryan, C., & Frampton, K. (2007, 11-12 Oct. 2007). *Cohesion Metrics for Predicting Maintainability of Service-Oriented Software*. Paper presented at the Quality Software, 2007. QSIC '07. Seventh International Conference on.
- Plosch, R., Gruber, H., Ko, x, rner, C., & Saft, M. (2010, Sept. 29 2010-Oct. 2 2010). *A Method for Continuous Code Quality Management Using Static Analysis*. Paper presented at the Quality of Information and Communications Technology (QUATIC), 2010 Seventh International Conference on the.
- Sarwar, M. M. S., Ahmad, I., & Shahzad, S. (2012, 17-19 Dec. 2012). *Cyclomatic Complexity for WCF: A Service Oriented Architecture*. Paper presented at the Frontiers of Information Technology (FIT), 2012 10th International Conference on.
- Schneidewind, N. F. (1992). Methodology for validating software metrics. *Software Engineering, IEEE Transactions on*, 18(5), 410-422. doi:10.1109/32.135774
- Shen, V. Y., Conte, S. D., & Dunsmore, H. E. (1983). Software Science Revisited: A Critical Analysis of the Theory and Its Empirical Support. *Software Engineering, IEEE Transactions on*, SE-9(2), 155-165. doi:10.1109/TSE.1983.236460
- Shepperd, M. (1988). A critique of Cyclomatic Complexity as a software metric. *Software Engineering Journal*, 3(2), 30-36.

- Steidl, D., Hummel, B., & Juergens, E. (2013, 20-21 May 2013). *Quality analysis of source code comments*. Paper presented at the Program Comprehension (ICPC), 2013 IEEE 21st International Conference on.
- Stevens, W. P., Myers, G. J., & Constantine, L. L. (1974). Structured design. *IBM Syst. J.*, 13(2), 115-139. doi:10.1147/sj.132.0115
- Suleman Sarwar, M. M., Shahzad, S., & Ahmad, I. (2013, 10-12 Sept. 2013). *Cyclomatic Complexity: The nesting problem*. Paper presented at the Digital Information Management (ICDIM), 2013 Eighth International Conference on.
- Systa, T., Ping, Y., & Muller, H. (2000, Feb 2000). *Analyzing Java software by combining metrics and program visualization*. Paper presented at the Software Maintenance and Reengineering, 2000. Proceedings of the Fourth European.
- Tosun, A., Caglayan, B., Miranskyy, A. V., Bener, A., & Ruffolo, N. (2011). *Different Strokes for Different Folks: A Case Study on Software Metrics for Different Defect Categories*. Paper presented at the Proceedings of the 2nd International Workshop on Emerging Trends in Software Metrics, Waikiki, Honolulu, HI, USA.
- van Emden, E., & Moonen, L. (2002, 2002). *Java quality assurance by detecting code smells*. Paper presented at the Reverse Engineering, 2002. Proceedings. Ninth Working Conference on.
- Yuksel, U., & Sozer, H. (2013, 22-28 Sept. 2013). *Automated Classification of Static Code Analysis Alerts: A Case Study*. Paper presented at the Software Maintenance (ICSM), 2013 29th IEEE International Conference on.