ISSN 1744-1986

Technical Report N° 2006/ 22

An Empirical Comparison of Subjective Evaluation and Metrics in the Maintenance of COBOL software

Richard Gorman

30 September, 2006

Department of Computing
Faculty of Mathematics, Computing and Technology
The Open University

Walton Hall, Milton Keynes, MK7 6AA
United Kingdom

http://computing.open.ac.uk

An Empirical Comparison of Subjective Evaluation and Metrics in the Maintenance of COBOL software.

A dissertation submitted in partial fulfilment
of the requirements for the Open University's
Master of Science Degree in Software Development

Richard Gorman
T5913155

6 March 2007

Word Count: **14868**

Preface

I would like to thank my employer LINK Interchange Network Ltd. for their financial support in the form of the sponsorship for this project. Additionally I would like to thank my colleagues for their support, particularly those who responded to my questionnaire.

Finally, I would like to thank my supervisor, Brian Kavanagh, for his support and advice throughout this project.

Table of Contents

List of Figures

List of Tables

Abstract

The cost of software maintenance and in particular the maintenance of legacy software such as COBOL has been widely reported. It is therefore important to be able to measure the maintainability of such software. This study investigates the two primary methods of measuring the maintainability of software; subjective evaluation using software developers, and the more formal and objective approach of software metrics. In addition to these primary methods, a taxonomy of COBOL code smells is developed for potential use in determining software maintainability. The two methods of measurement were applied to a sample of nine COBOL programs. A group of six developers gave a subjective evaluation of the software by answering a questionnaire which assessed their opinion of the software and of the taxonomy of smells, and how these might influence their views. In addition, a total of seven software metrics were used to gain an alternative view of the same software. These metrics were lines of code, McCabe's Cyclomatic Complexity and Halstead's Software Science Indicators.

The results show good correlation between the individual metrics evaluated and an equally strong correlation between the metrics and the results of the subjective evaluation. Only one metric, Halstead's Difficulty showed little relationship to the other metrics or to the results of the subjective evaluation. Interestingly, the subjective evaluation showed greater variation in the results than did the software metrics and the subjective evaluation using the taxonomy of smells showed a good correlation with the initial subjective view, yet it did cause a change in opinion in 42.6% of the evaluations. Additionally, the results showed that whatever the method of measuring maintainability, the age of the software is important, and the older the software, the harder it is likely to be to maintain. The results also showed that complexity and size both contribute to maintainability and also that developer experience leads to a better view of the maintainability of the software.

This study concludes that software maintainability is influenced by a number of factors, including the source code, program age and developer experience. As a result, the subjective element of developer intuition is required to be able to include all of the factors relevant to measuring software maintainability. Metrics alone cannot easily capture all of these factors, and therefore a metric, or combination of metrics should be used to corroborate the opinions of developers, especially where inexperienced developers are involved in the evaluation.

Chapter 1 Introduction

This dissertation investigates the problems posed by software maintenance and how the maintainability of software can be measured, looking specifically at the maintainability of COBOL software. There are a number of different options and opinions as to the best way to measure software maintainability; this study looks to outline these and investigate how they can be used. This chapter outlines the issues around software maintenance and describes the aims of this study.

1.1 Description of the Problem Domain

Historically, software was written in an unstructured manner, leading to the formation of 'spaghetti code' (Boehm, 2006). This code is very difficult to understand and maintain, as the code jumps from place to place and back again. This problem is often attributed to the use of the GO TO statement and led to a movement to eliminate the use of GO TO and the introduction of structured programming (Dijkstra, 1968; 1972). Structured programming introduced modularity and control flow to software, encouraging single entry and exit points in the modules, and became the norm during the 1970s and 1980s. Further development saw Object-Oriented Development become the preferred methodology replacing the procedural and structured programming methods in the 1980s and 1990s. Khan et al. (1995) describe the benefits of object-oriented programming as "better structured and more reliable software for complex systems, greater reusability, more extensibility and easy maintainability", although there are a number of factors which can influence this. However, many organisations still have code which has its origins more than 25 years ago, prior to the growth in object-oriented development, and for many reasons they wish to continue to maintain those systems. These systems may include both structured and unstructured procedural code. Bennett (1995) describes how legacy systems are performing crucial work, utilising years of accumulated experience and knowledge, citing a lack of documentation and user resistance among the factors which have led to the continued use of systems which are hard and costly to maintain, but equally hard and costly to update. It has also been shown that some organisations maintain the use of older technologies as they help to achieve fault tolerance for mission critical systems (Bartlett and Spainhower, 2004). Many of these older technologies rely on COBOL and Koskinen (2004) reports that Gartner found that there were still at least 200 billion lines of COBOL in use in mainframe systems.

The priority for many software development organisations is speed to market and the ability to deliver to deadlines. Organisations are often very reluctant to spend time and effort in ensuring the quality and maintainability of the code which is developed, the main focus being on delivery of projects on time and within budget. The project manager is unlikely to have the quality or future maintainability of the software as his/her top priority, as the emphasis will be on the short-term need to complete the project on schedule and within budget. As Lientz (1983) commented "Management frequently exerts pressure to control costs and modify schedules. This pressure can directly impact the quality and quantity of the maintenance effort".

The problems of software maintenance have been studied in some detail and there is a great deal of evidence to show that maintenance makes up a significant part of the cost of software. Koskinen (2004) reviews the costs found by a number of studies and describes how software maintenance has been shown to make up a high percentage of development cost and that this percentage continues to increase over time, from 67% in 1979 to in excess of 90% in 2000. These costs can be in terms of the time taken to identify and fix problems, as well as the time required to make changes and implement enhancements. Coleman et al. (1995) suggest that although new software development techniques should have led to systems which are easier to maintain, this benefit has been offset by the increased size and complexity of the systems. This increased difficulty and cost of maintaining software over time is known as 'software entropy' or 'software rot' (Hunt and Thomas, 1999; Webopedia, 2001).

The IEEE (1998) defines three types of maintenance carried out on software based on an original proposal by Swanson (1976); adaptive maintenance amends software to work in a changed environment, corrective maintenance resolves problems within the software and perfective maintenance is designed to make software easier to change and maintain, and should make adaptive and corrective maintenance easier. Boehm is quoted as defining software maintenance as "the process of modifying existing operational software" (Stark, 1996) but Mantyla and Lassenius (2006) prefer the term software evolvability to describe "the ease of further developing a software element", as the use of the words maintenance or maintainability suggests keeping the software in its existing state. Mantyla and Lassenius liken software evolvability to perfective maintenance but this neglects corrective and adaptive maintenance which although leaving the function of the software unchanged, does modify the software. For this reason, I prefer to include all three types of maintenance and will use Boehm's definition in relation to my study. Similarly software maintainability has a number of descriptions. For the purpose of my research I have used the definition suggested by Martin and McClure (1983) that maintainability is the "ease with which a software system can be corrected when errors or deficiencies occur and can be expanded or contracted to satisfy new requirements".

A number of factors affect the maintenance of software and must be considered by those responsible for managing software development to create maintainable systems. Vessey and Weber (1983) suggest that the key factors include program complexity, the type of programming used, programmer quality, frequency of maintenance and program age. The managers of the software development need to include these factors along with the demands of their organisations to ensure software development and maintenance can be done quickly, at the minimum cost and to the highest quality. The problem for these managers is how this is best achieved and, equally importantly, how they can tell that their developments are achieving these aims.

One approach to measuring the maintainability of software is through subjective evaluation. The agile community (Beck et al., 2001) suggests that human intuition and the experience of the software developers gives more value than metrics, preferring "individuals and interactions over processes and tools". Fowler and Beck (1999) identified a series of 'bad smells' in object-oriented development which can be used to suggest where code may benefit from refactoring, which is the process of identifying areas which could be changed to improve the design of the code. Essentially the 'bad smells' enable the identification of code where perfective maintenance might be usefully performed. Kataoka et al. (2002) suggest that refactoring produces benefits such as improved design, code that is easier to understand, identification of bugs and making programming faster. Refactoring is generally associated with extreme programming and object-oriented techniques, where an iterative approach is used; however Belyaev et al. (2004) suggest that refactoring although associated with extreme programming methods, is in fact a basic process which can be applied in many scenarios. A classification of the 'bad smells' was proposed by Mantyla et al. (2003) and then used in a subjective evaluation of software (Mantyla et al., 2004). Mantyla (2004) states that "subjective (mostly ad-hoc or 'gut-feeling' based) quality evaluation and improvement is the one most used in practice". My personal experience within a structured software environment would support this view. An investigation into the use of peer reviews in quality evaluation was carried out by Shneiderman (1980), identifying that the evaluations of a number of developers showed some degree of correlation, but also highlighting that there is some diversity of opinion between the developers, which could lead to concern over the use of subjective evaluation.

The alternative to the use of human intuition and subjective evaluation is to make use of metrics which provide a more formal measure of the quality and maintainability of code. The use of metrics to measure maintainability has been investigated widely, both in structured programming by Coleman et al. (1994; 1995), Oman and Hagemeister (1992) and Kafura and Reddy (1987), as well as in the object-oriented environment, with specific focus on refactoring (Simon et al., 2001; Kataoka et al., 2002). Coleman et al. (1994) found that the metrics they studied (HP Maintainability Assessment System and polynomial models) corresponded to the subjective evaluations of the systems experts, while Wake and Henry (1988) concluded that the model they generated “using several quality factors is the direction to follow for predicting software quality”.

The challenge for software development managers is the decision whether to rely on the intuition of the developers, formal metrics or to use a combination of the two.

1.2 Aims and objectives

The aim of this study is to compare the merits of subjective evaluations and maintainability metrics within a COBOL development environment. The intention is to study the subjective evaluation of the quality and maintainability of code using developer opinions of source code. This will be done both before and after the developers have been introduced to the ‘bad smells’, identified from my research, and will lead to a comparison with the results of software maintainability metrics. I will create a taxonomy of smells which adapts and builds on some of the ideas of Mantyla and his colleagues in a series of papers (Mantyla et al., 2003; Mantyla et al., 2004; Mantyla, 2004; Mantyla and Lassenius, 2006). I have used the notion of ‘bad smell’ classifications in a structured COBOL programming environment using definitions of good and bad structured COBOL programming, in addition to some of the smells identified by Fowler and Beck (1999) and classified by Mantyla et al. (2003) where these smells can be applied to structured environments as well as to object-oriented software.

The term subjective evaluation is based on the description used by Mantyla and Lassenius (2006) who liken it to the judging in figure skating or ski-jumping. Such evaluation is based on an individual's opinion which can be swayed by knowledge, expertise and many other factors. The initial views of the developers were collected by use of a survey using a subjective numerical scale. The survey was also used to collect evaluations of the use of the smells. I applied this to COBOL software, using the developers within the team in which I work to investigate their ability to determine and identify potential maintenance problems in code, and areas where improvements could be made to improve maintainability. The results of this study should be applicable to many structured/procedural programming environments, with variations required for the different types of logic within some programming languages.

A comparison of the results of the subjective evaluations and the results of the maintainability metrics when applied to the same source code will help to determine the relative merits of and correlation between the two methods of measuring maintainability. Maintainability metrics are generally based on the size and complexity of the code. Complexity is normally considered from the point of view of source code complexity, but Stark (1996) defines complexity as "the degree to which characteristics that affect maintenance releases are present", including additional characteristics such as age, management process and staff experience which might cast some doubt on the usefulness of source code complexity metrics in measuring maintainability. Common metrics in this area include lines of code, McCabe's Cyclomatic Complexity, Halstead's Effort and Function Point Analysis (Fenton and Pfleeger, 1997); these metrics were among those considered for use.

1.2.1 Research question and project objectives

The research question which this study will try to answer is:

"To what extent can subjective evaluations and maintainability metrics provide a good measure of software maintenance within COBOL software?"

The objectives of the study, which will help to answer this question are:

- Investigate the reports and theory of software maintenance and the use of software metrics and subjective evaluation in maintenance.
- Investigate developer intuition in determining the maintainability of software.
- Determine whether developer intuition can be aided by 'smells' when considering software maintenance.
- Determine whether software metrics can be used to aid or replace the intuition of developers in determining software maintainability in structured systems.
- Compare and contrast the use of subjective evaluation and software metrics in determining the maintainability of software.

1.3 Contribution to Knowledge.

This empirical study will be of interest to those organisations that use structured systems using languages such as COBOL, PASCAL, FORTRAN and PL/1. This study presents information which will allow them to develop and inform their own studies or policies on software evolution and maintenance. The results should indicate the relative merits of evaluation by developers, the use of 'bad smell guidelines' to assist those evaluations, and whether or not those evaluations can be supported or replaced by software metrics. Many organisations retain legacy systems developed on older structured technologies and tend to require very high availability for their software; these organisations will therefore have a strong interest in the maintainability of their software. The comparison between subjective evaluation and metrics will help managers to determine which metrics or techniques, if any, can be introduced to measure maintainability and quality, and whether these can replace or supplement the subjective evaluation and assessments done by the developer and his or her peers.

The study will provide a further empirical study to add to the existing knowledge of subjective evaluation and metrics in software maintenance, utilising some of the ideas in the existing studies to add to the base of knowledge available this area. It will apply some of the recent approaches used in studying object-oriented software maintenance to make further observations on the maintenance and quality approaches to older technologies. In particular the work of Mantyla and Lassenius (2006) provided a number of ideas to be adapted and built on; identifying that there are a limited number of empirical studies in the area of software maintenance, particularly regarding the use of subjective evaluations and how evaluation and metrics contribute to creating maintainable software. The findings of Mantyla and Lassenius are adapted by identifying ‘bad smells’ which may exist within COBOL programs. Stevens et al. (1974) published one of the first papers to identify the elements of structured design to aid maintenance, and there have been a number of studies of good and bad COBOL programming methods (Linderman, 1982; Veerman & Verhoeven, 2006). By combining some of these methods with some of the bad smell classifications of Mantyla et al. (2003), a new empirical study will be created which will provide new information to investigate software maintenance in legacy COBOL systems.

Chapter 2 Literature Review

There have been a number of papers and studies of the issues surrounding software maintenance and the use of human intuition and software metrics in this field. This chapter reviews these existing studies and describes how they have influenced this study.

The studies of Mika Mantyla and his colleagues at the Helsinki University of Technology have provided a number of the ideas for this study. Mantyla and Lassenius (2006) studied subjective evaluation, including how evaluations differ, which factors can have a bearing on the evaluation and how these can be aligned to metrics. They found that evaluators agreed on the existence of certain types of problem and identified that factors which cause such problems include the type of method being studied. The results of the study of metrics suggested that these may be useful in identifying straightforward problems but cast some doubt on the ability to replace subjective evaluations with tools, as some problems were considered difficult or impossible to detect with metrics. To date the studies carried out by Mantyla and his colleagues have involved the use of students rather than developers in commercial environments. This restricts the usefulness of the results and certainly limits their value in commercial real-world environments, as such environments have very different characteristics to the academic environment. The use of developers is one of the recommendations made for future work by Mantyla and Lassenius and is one of the features of this study. The smells used by Mantyla and Lassenius were introduced by Fowler and Beck (1999) in their guide to refactoring. Refactoring focuses on finding code which can be improved in object-oriented software. Fowler and Beck introduced a number of ‘bad smells’ which indicate where object-oriented code could be a candidate for refactoring. Mantyla et al. (2003) built classifications of these ‘bad smells’ and investigated relationships between the classifications. The classifications were then used by Mantyla et al. (2004) to study the use of the ‘bad smells’ in subjective evaluations and how these correlated to the use of source code metrics and measures. This study led me to consider how the subjective evaluation of the good practices of structured COBOL programming would compare to the intuition of developers and also to the more widely used software metrics. These aforementioned papers contain many of the elements which I used in my research, the key difference being the type of system being evaluated. Mantyla and his colleagues studied an object-oriented system and therefore used metrics for such a system, and the evaluations concentrated on refactoring and the ‘bad smells’, which focus on object-oriented systems. I have utilised the ideas in these papers, using metrics and evaluation guidelines which can be applied to a structured and procedural environment. Mantyla et al (2004) and Mantyla and Lassenius (2006) looked specifically at individual smells and how the evaluation of these smells compares to the metrics chosen to measure them. A more appropriate view might be how the maintainability of the software as a whole is impacted by the smell, using metrics to measure maintainability as a whole rather than to measure only certain elements of the software.

2.1 Subjective Evaluation of Software

The subjective evaluation of software has been studied by Basili and Hutchens (1983) who looked at the skills of programmers and how differences in their relative skills can be measured. The study found that a programmer's ability to deal with complexity could be measured and suggests that a team can work more effectively than individuals would. This study is important in highlighting the need to look at programmer ability when carrying out the analysis of the subjective evaluation. Shneiderman (1980) investigated the human element of software development looking at how the programmers have an impact on software and its quality, and how factors such as personality and programming style can influence the development of software.

2.2 Comparisons of subjective evaluation and metrics in software maintenance

There have been a number of studies which have evaluated the relative merits of subjective evaluation and metrics within software maintainability. Coleman et al (1994, 1995) performed a comparison of software metrics for analysis of software maintainability against the human intuition of developers on 11 software systems, and concluded that the metrics corresponded to the intuition. Metrics help to support the views of developers, which on their own are often not considered enough to persuade management of the need for software maintenance and re-engineering. The metrics studied suggested models which could be applied to a system to indicate when it is in need of perfective maintenance. The models used were mostly complex metrics and models, including models built specifically for use at Hewlett Packard, but models based on Halstead's Effort, McCabe's Cyclomatic Complexity and lines of code were also used. It would be expected that the key metric studied, the HP Maintainability Assessment System, which is used specifically by HP, would match the opinion of those experts working in that organisation, so it may be interesting to concentrate on the more widely available metrics to provide more widely applicable results.

Kafura and Reddy (1987) studied software complexity metrics and their relationship to the actual maintenance carried out on a system, looking at three different versions of the same system. They concluded that a relationship exists between the complexity shown by metrics and the actual maintenance, and that the metrics agreed with expert opinion of the system. As in my research, this study looked at the holistic view of maintenance rather than the metrics for individual smells. It also used a number of complexity metrics, such as McCabe's Cyclomatic Complexity, lines of code, and Halstead's Effort, which are easily measured in most systems using readily available tools. The subjective evaluation was carried out by two experts, which meant that the study did not review the impact on the subjective evaluation of code of other factors such as programmer ability or experience. This lack of demographic evaluation is something which is also apparent in many of the recent studies including those by Mantyla et al. and Coleman et al.

These demographic factors are taken into account by Rosenberg (1997) who looked at two areas where quantifying maintainability is important. The use of source code metrics in determining the likelihood of corrective maintenance was reviewed, suggesting little correlation between the well known source code size and complexity metrics and actual defects, and that semantic and not syntactic metrics are necessary. Additionally the difficulty of maintenance activity is considered and again, Rosenberg believes the problems are semantic, not syntactic. Code intelligibility, modularity and decay are the suggested areas which cause maintenance difficulties, incorporating non-syntactic factors such as development environment, programmer skill and domain knowledge. A comparison of the use of human techniques such as code walkthrough and inspection with the use of computer-based tools when attempting to find problems in software was carried out by Myers (1978). The results of the study suggested that these human techniques were at least as effective in terms of cost and effort as the tools.

Gibson and Senn (1989) researched the possible relationship between system structure and maintainability, using a range of programmers in an experiment looking at maintenance of three versions of a COBOL system with structural differences. The study used the commonly researched metrics, including McCabe's and Halstead's metrics, and concluded that structure did impact maintenance performance. The most interesting conclusion drawn was that the differences in structure were not discernable to the programmer, leading to a conclusion that subjective assessment of software complexity was of questionable benefit, in contradiction to the findings of a number of other papers (Myers, 1978; Kafura and Reddy, 1987). This conclusion is limited however to a view of how structure affects maintainability and does not review the effects of the other characteristics of the software. Further support for the need to consider additional elements of the software other than those focussed on in many studies is provided by Mathias et al. (1999), who reviewed the use of software measures and metrics in programmer comprehension, analysing how previous research had used the top-down and bottom-up comprehension models. By looking at a series of studies which had used different metrics to suggest programmer comprehension, they suggested that these studies all neglect to consider the effect that the nature of the software has on the ability of a metric to indicate comprehension.

A number of studies have focussed on building a complex maintainability model for a specific environment. Oman and Hagemeister (1992) built a single maintainability index for a software system by using a combination of maintainability metrics to allow prediction of maintenance needs in a way which can be measured. The study included a large number of simple metrics to measure elements such as age, size, reuse, complexity, nesting and modularity. This study used only metrics to measure these attributes, with the exception of supporting documentation where subjective evaluation was used. This study suggested a large number of attributes need to be considered in determining whether subjective evaluation can be guided by the good and bad attributes, while Vessey and Weber (1983) proposed a number of additional factors which may affect software maintenance and carried out empirical investigations on these factors. Some evidence was found for the impact of program complexity, programming style and frequency of maintenance, but surprisingly the effects of programmer quality and program age were found to have little impact. This is a useful study in considering both how subjective evaluation may be carried out and some of the additional factors beyond the source code which may impact maintainability or a developer's view of software maintainability. These additional factors highlight some of the gaps in the previous studies where only the expert opinion was considered in carrying out a subjective evaluation.

Two more recent studies have looked at object-oriented software and the use of metrics to support developers in refactoring object-oriented code. Kataoka et al (2002) looked specifically at coupling metrics to evaluate the success of refactoring in improving maintainability. The not unreasonable view presented of subjective evaluation was that experienced developers and analysts may be able to carry out evaluations unaided, but metrics and quantitative analysis could assist others to assess maintainability. Simon et al. (2001) view metrics as a method of assisting a developer who "has to be the last authority" and they try to supplement the developer in finding areas for refactoring by using source code analysis to create visualisations of code which identify the 'bad smells' and candidates for refactoring. The view that the developer is required to make the ultimate decision is no doubt influenced by the link between the refactoring and the agile community, which favours developers over tools.

2.3 Using metrics to assess maintainability

Subjective evaluation was historically the principal method of measuring maintainability, but much of the recent focus has been on software metrics and their use in measuring maintainability. Wake and Henry (1988) provided a study of the use of metrics in software maintainability, splitting the metrics into code metrics, structure metrics and hybrid metrics. The metrics were applied to C code and included lines of code, McCabe's Cyclomatic Complexity and Halstead's indicators. The study importantly suggests that single metrics do not best determine software quality and maintainability, but models using several metrics seem to be the best way to make predictions in this area. Some metrics did not contribute to the models required for the environments studied, but it is astutely identified that this may not be the case for all environments; the metrics that are meaningful for one system may be different to those for other systems. Zhuo et al. (1993) constructed seven maintainability models to provide assessments of software maintenance, based on a similar set of metrics. These models provide an idea of how models that are more complex can be built for specific systems and make the important distinction that source code is only one of the factors which needs to be measured to get a view of maintainability. These findings support a view that different types of metric are required to build a complete view of maintainability, increasing the complexity of the required model, which could be argued to support the view that human intuition is easier to use than the array of metrics required to measure the many relevant factors. Further to this, Harrison and Walton (2002) investigated the maintenance of legacy software, and the use of metrics in such environments to measure maintenance. They used maintenance metrics as well as program metrics, such as lines of code and McCabe's Cyclomatic Complexity to identify that program size and metrics were in correlation while also showing that program structure does not necessarily provide programs which require less maintenance. They also showed some limited support for the argument that larger programs require more maintenance. These conclusions provide little evidence that metrics to measure programs will predict maintenance. The authors argue this could indicate a need to look at usage and the full lifecycle of the program, but these conclusions may lend further support to the argument that subjective evaluation is the more effective methodology for measuring maintainability.

A Cyclomatic Complexity measure was developed by McCabe (1976) to analyse program complexity, taking the view that the basic lines of code measures were not enough. McCabe's measure gives a graphical representation of the complexity of a program based on the flows through a program. A measure of complexity can then be generated with the intention of ensuring software does not exceed a certain level of complexity. This measure is one of the best known and most often used and studied measures of software complexity and maintainability in structured software systems. There have been several variants of McCabe's Cyclomatic Complexity some of which are easier to calculate than the original (Myers, 1977; Hansen, 1978). These variants were reviewed by Gill and Kemerer (1991) and many were viewed as less effective than the original. However, Gill and Kemerer believe that a complexity density ratio may be even more effective as a measure of how difficult a system may be to maintain; the complexity density ratio divides the Cyclomatic Complexity by the size of the system (source statements) and was shown to be a significant predictor of maintainability.

Stark (1996) reviewed all aspects of the software maintenance process and introduced many factors not considered in many of the studies of metrics in software maintenance. These factors could cast some doubt on the reliance on source code metrics as the sole measure, but Stark believes that metrics enhance the maintenance management. The study does lend credence to a suggestion that the combination of measures required will differ from system to system, as the study was unable to make two well known models fit the systems being investigated (the models proposed by Ash et al. (1994) and AFOTEC (1991)). The model Stark used introduced several factors beyond the software itself into a measure of complexity, using number of changes to reflect size, rather than the usual measures such as lines of code, and considering factors beyond the software such as environment (office and system) and management processes (maturity and tools) as well as the more usual software characteristics such as size and age.

There are, however, a number of other studies that suggest that metrics are a strong measure of maintainability. Lewis and Henry (1989) studied maintainability in large-scale software and developed a methodology for use in commercial systems. They compared the majority of the most common maintainability metrics, covering code, structure and hybrid metrics as Wake and Henry had, comparing the results with actual data regarding errors, and looking for relationships between the metrics and errors. They found that structure and hybrid metrics had a stronger correlation with actual errors than code metrics. This study provides a great insight to commercial organisations about the use of metrics in maintainability. A further comparison of the results of this type of work with the views of developers in the organisations and how their evaluations relate to these types of metric will lead to a very comprehensive evaluation of the methods of determining maintainability.

Banker et al. (1993) investigated the impact of software complexity on the maintenance of COBOL software, taking a rather different approach to many of the studies of the relationship between complexity and maintenance. They chose to avoid the classical academic measures of complexity, such as McCabe's Cyclomatic Complexity and Halstead's Software Science Indicators, using their own view of complexity based on measures of module size, procedure size and branching complexity. This study also provides an excellent maintenance cost model, factoring in the many other factors which can affect maintenance costs, such as experience, skill and the analysis and design method used, thus creating an interesting statistical model of cost using all of these factors alongside the measures of complexity. It is one of the few studies to have taken into account these factors and used additional complexity measures such as module size, procedure size and function points, and it concluded that a high level of software complexity could account for 25% of the costs of software maintenance. Given that maintenance has been shown to be a major cost of software, this percentage cost caused by complexity, while showing that complexity is important, also suggests significant maintenance costs are generated by other factors.

An alternative approach to researching software maintainability metrics was proposed by Land (2002). The proposal was to investigate many of the well known source code metrics and how they measure the maintainability of a system as it changes. These changes would include minor source code changes as well as significant restructuring of code. This approach allowed measures of maintainability after each change and showed what types of code change cause increases or decreases in each of the maintainability models. Although this research will not help organisations to build a model for their own environment without taking into account the individual software environment, it will add a great deal to the information available about how the metrics react to certain types of code change and provide insight into which metrics may be appropriate to use in a given circumstance.

2.4 Summary and Hypotheses

There is a wide range of literature discussing the factors surrounding software maintainability and how it should be measured. There is clearly some support for the use of software metrics; however, the range of findings and the variety of metrics which have been used in the current literature suggests that the metrics required would need to be designed for the needs of a particular environment, and that common maintainability metrics may need to be amended to take additional factors into account. It is apparent that while there is a degree of support for metrics, they are generally viewed as useful as support for the intuition of the developers, and other factors such as the quality and experience of the developer and age of the software should be factored into any measure of software maintainability. This leads me to the following hypotheses:

- Metrics should be used to supplement and corroborate the opinions of developers as to the maintainability of software.

- As well as the actual source code, other factors such as the age of the software and the experience of the developer influence the maintainability of software.

These hypotheses will be tested as part of the analysis concerning the research question for this project.

Chapter 3 Research Methods and Data Collection

This chapter describes the methods used to carry out the research. It outlines the basic research methods used and the data collection and analysis techniques used to capture the data required for the research. The study utilises three main research methods: a case study using software metrics, survey research using questionnaires and a document study. The aim of the survey is to gather subjective evaluations of the software to be compared against software metrics of the same software. The document study will identify 'bad smells' in COBOL and help generate a taxonomy of these smells to be used by the developers in evaluating the software.

3.1 Selection of Software

The software to be analysed in this study was chosen for several reasons. It was originally hoped to study software written in more than one language, but this idea was discarded as it became clear that the study would be more useful if it concentrated on a single language. It may have been necessary to gain permission to publish the code of the software to ensure this study could be correctly interpreted and be of the most use to a wider audience. Therefore I chose to use only software written within the organisation for which I work. This organisation is a medium-sized financial services company in the United Kingdom, where most of the code is developed in COBOL on the HP Non-Stop mainframe. The code is developed by a team of ten developers with various levels of experience. I chose to use nine different pieces of COBOL source code for the maintainability study, where the source code varied in size, age, author and objective. This approach was intended to ensure that the various factors which might affect maintainability would have a chance to be identified across a wide range of software. The code developed by this team is intended to adhere to a set of coding standards developed by various members of the team over recent years. Table 3-1 lists the nine pieces of source code, along with their age and a brief description of the purpose of the code.

Source Code	Year Written	Description of Purpose
NBSCON	1997	This program controls the number of card and PIN numbers requested and maintains statistics of the orders.
ADVRCNL5	1998	A reconciliation program which compares two files of financial transactions and reports anomalies between the two.
EAEPROG	2000	A comparison of two files of card prefix details which merges the data in the two files and reports any discrepancies between the input files.
RETREPS	2000	This program extracts details of transactions performed at cash machines to provide totals for each machine for that business day.
VOUTPREP	2001	This formats a file of transaction presentments to be sent to VISA based on cash machine transactions performed that day.
OMFATDS	2002	This program maintains an audit of updates, additions and deletions made to a file containing the details of cash machines on the system.
MBREXTS	2005	This program extracts the details of member institutions from a file provided by MasterCard.
ISSREVS	2006	This program identifies financial transactions with a reversal (to undo the original), and removes the matched pair from the file.
TXMATCHS	2006	This program matches transactions authorised on-line with the off-line request from the acquirer of the transaction.

Table 3-1 Summary of the nine pieces of source code chosen for this study

3.2 Software Metrics

The case study of the software will utilise metrics which measure the maintainability of the software. There is wide evidence of such case studies having been carried out before (Coleman et al., 1995; Wake and Henry, 1988). The measurement of software maintainability is generally linked to the size, complexity (Lewis and Henry, 1989), quality (Wake and Henry, 1988), program comprehension (Mathias et al., 1999) or readability (Aggarwal et al., 2002) of the software. There is a large collection of metrics which can be used to measure the complexity and quality of software, but it is important that the metrics chosen for the study can be easily measured and are the most relevant to a comparison with subjective evaluation of the same software. It was necessary to measure the code using the metrics chosen within reasonable timescales for this study, and it was also important to be able to obtain results from the metrics which will have meaning in a comparison with subjective evaluation. Wake and Henry (1988) suggest lines of code, McCabe's Cyclomatic Complexity and Halstead's Software Science Indicators, and Zhuo et al. (1993) adds lines of comments and number of blank lines to the list of indicators of software quality. A number of studies have used these metrics to build maintainability models for industrial systems. The HP Maintainability Assessment System is widely reported (Coleman et al., 1994; 1995) as a good example of where metrics can be used to measure software maintainability. Other complexity measures such as Function Point Analysis and the COCOMO model measure complexity in terms of the functionality of the software. I considered all of these methods before determining to use the following measures:

- Lines of code
- McCabe's Cyclomatic Complexity
- Halstead's Software Science Indicators

These measures were chosen having taken into account the factors already described, and also because they focus on the source code, which is the target of this study. Other measures such as Function Point Analysis and the Object Point of the COCOMO model require more than just the source code to obtain the measure, and focus on estimation of cost rather than software maintainability. The fan-in and fan-out measure of Henry and Kafura's Information Flow (Henry and Kafura, 1981) is commonly used to measure the flow into and out of procedures, but this is for measuring entire systems and is not appropriate to the measure of individual pieces of source code, and was therefore discounted as a measure to be used. Land (2002) and Lewis and Henry (1989) recommended similar measures to those I have chosen, avoiding any that are too time-consuming. More complex measures such as the Maintainability Indices of Welker et al. (1997) or the HP Maintainability Index used by Coleman et al. (1994; 1995) are excluded as they would have been too difficult to measure in the timescales of this project as they were not measured by any readily available tools.

The lines of code measure was chosen because of its simplicity; it is easy to collect and there are many tools available to help with this collection if necessary. It is important to be clear what exactly is meant by lines of code, as there are many variations of this measure. The variations of this measure may or may not include blank lines, comment lines, and header or declaration lines in the count. There are also variations based on whether a source statement over two lines should count as one or two lines. The measure described as most common by Fenton and Pfleeger (1997) is that defined by Hewlett Packard, where a line of code is a non-commented source statement. It is this definition that I will use in this study, rather than the other common definition which is to count all lines except blank and comment lines (Conte et al., 1986).

McCabe's Cyclomatic Complexity (McCabe, 1976) measures the structure of the software by determining the number of independent flows through the program. The measure has been extended by Myers (1977) and Hansen (1978) but I will use the original version, as it is more commonly used and a number of tools are available to assist in measuring it. Halstead's Software Science Indicators (Halstead, 1977) are all complexity metrics based on the number of operators and operands in the source code providing measures of effort, difficulty, volume, vocabulary, length and program level. Various studies have used any number of these measures in relation to complexity, with effort, length and volume most widely used. I chose to use effort, volume and length, as they are the most widely used in other studies (Coleman et al., 1994; Zhuo et al., 1993), and also difficulty and vocabulary, which are used to derive some of the former measures. As shown in Figure 3-1, the length measure is a measure of size based on the number of operands in the code. Volume combines this with the vocabulary which measures size based on the operators in the code. The effort measure is perhaps Halstead's best known metric, which estimates the time required to develop the code.

Halstead's measures are based on the following:

n1 – the number of distinct operators

n2 – the number of distinct operands

N1 – the total number of operators

N2 – the total number of operands

Halstead's measures studied in this project are calculated as follows:

Length(L) = N1 + N2

Vocabulary(v) = n1 + n2

Volume(V) = L*LOG2v

Difficulty(D) = (n1/2)*(N2/n2)

Effort = D*V

Figure 3-1 The definition of Halstead's Software Science Indicators

3.2.1 Metric Collection

The metrics chosen for this study were chosen for their relevance to the subject and for their ease of collection. Lines of code measures are easy to collect manually but this can be time-consuming and there are many tools available to collect these automatically. Given that I had chosen to measure lines of code by the number of lines of source code, I chose to use Source Code Counter Pro from GeroneSoft (2006), which was very easy to use to collect these details for COBOL software. McCabe's Cyclomatic Complexity is perhaps the most difficult of the metrics chosen to collect manually; to do so I used LegacyAid by CASEMaker Inc. (2006), which was specifically written for legacy software such as the COBOL on the HP Non-Stop mainframe software used in this study, and which provided a quick and easy means of collecting this measure. Unfortunately, there are few tools available to collect the Halstead Software Science Indicators for COBOL software; the most useful appears to be Surveyor for COBOL by Lexient Corp. (2006) but this was outside the budget of this project and was difficult to obtain. It was therefore necessary to collect the total number and distinct number of operators and operands in the software manually to calculate the Halstead indicators. This was a time-consuming process and could not realistically be used in a commercial environment where a tool would be required to make the process practical. An example of the metrics collected is shown in Figure 3-2, which shows a snippet of code from MBREXTS along with the metrics calculated for that snippet using the outlined methods. The figure shows that McCabe's Cyclomatic Complexity relates to the number of paths through the code, with the two IF ELSE statements creating 4 paths through this code.

```
MAIN-PROGRAM.

     PERFORM 1-INITIALISATION.
     PERFORM Z1-READ-CF0072T1.
     PERFORM 2-PROCESS-CF0072T1 UNTIL END-OF-CF0072T1.
     PERFORM 3-CLOSEDOWN.
     STOP RUN.

  1-INITIALISATION.

     OPEN OUTPUT MBREXTLOG.
     OPEN I-O CF0072T1
              EMBRDATA.

         MOVE ZEROES TO WS-COUNTS
                        END-OF-CF0072T1-FLAG.

     ENTER GETPARAMTEXT OF COBOLLIB
       USING RUN-DATE-TEXT, RUN-DATE-DATE GIVING PARAM-RESULT.
     IF PARAM-RESULT NOT GREATER THAN ZERO
        WRITE MBREXTLOG-REC FROM PRNT-DATE-PARAM-FAIL-MESS
     END-IF.

     WRITE MBREXTLOG-REC FROM PRNT-INIT-COMPLETE-MESS.

  2-PROCESS-CF0072T1.

     IF RUN-DATE GREATER THAN
        EFFECTIVE-TSTAMP
          ADD 1 TO RECS-READ-COUNT
           ON SIZE ERROR PERFORM Z00-SIZE-ERROR
      END-ADD
          PERFORM 21-PROCESS-CF0072T1-REC
          PERFORM Z1-READ-CF0072T1
     ELSE
           MOVE 1 TO END-OF-CF0072T1-FLAG
     END-IF.


  21-PROCESS-CF0072T1-REC.

     PERFORM 211-UPDATE-FILE.
     DELETE CF0072T1 RECORD.

  211-UPDATE-FILE.

     MOVE GCMS-MBR-ID(6:6) TO  EMBR-CF72-ICA.

     MOVE EMBR-CF72-FILE-KEY
           TO EMBR-FILE-KEY.

     READ EMBRDATA KEY IS EMBR-FILE-KEY.

     IF RECORD-NOT-FOUND OR END-OF-FILE
            PERFORM 2112-ADD-RECORD
     ELSE
        PERFORM 2111-UPDATE-RECORD

     END-IF.
```

```
Metrics for this code:

Lines of code:                          35

McCabe's Cyclomatic Complexity          4

Halstead's indicators:
                       Length           95
                       Vocabulary       57
                       Volume           554.12
                       Difficulty       17.97
                       Effort           9956.93
```

Figure 3-2 The various metrics for a snippet of MBREXTS

3.3 A Taxonomy of COBOL smells

For many years there have been studies and expressions of opinion on the good and bad practices seen in programming, from early attempts to define structured programming (Belady & Lehman, 1976; Stevens et al., 1974; Yourdon and Constantine, 1979) to the more recent classifications of 'bad smells' in object-oriented code defined by Fowler & Beck (1999). Mantyla et al. (2003) classified the 'bad smells' and went on to use the classifications in a subjective evaluation of object-oriented software (Mantyla et al., 2004). I wanted to use this idea of a taxonomy of features in code that suggest it will be difficult to maintain. This taxonomy would be provided to the developers carrying out the subjective evaluation to study whether they could agree on the existence of the problems and how their overall perception of the maintainability of the code would be influenced by the features in the taxonomy. Although the smells reviewed by Mantyla et al. were aimed at object-oriented code, as Veerman & Verhoeven (2006) suggest, some of the smells, and hence some of the classifications can be applied to structured programming and COBOL code. I have defined eight smells as a result of reviewing the classifications proposed by Mantyla et al. and reviewing literature and documents relating to good and bad practices in structured programming, with particular reference to COBOL. The taxonomy is not an exhaustive list of good and bad practices, but highlights those which I have identified as most likely to be useful in this study.

3.3.1 Poor sizing

Mantyla et al. introduced the concept of 'bloaters', a category which corresponds to the size of methods and classes, but it is equally true that modules in structured code, and COBOL in particular can become so verbose as to be difficult to maintain. Within COBOL a module can be defined as a section or paragraph of code. Glass & Noiseux (1981) describe the good modules as a "crisp functional breakdown"; size alone is not as important as a focus on building "small functionally oriented pieces". It is also true that a module can be so small that it is better not to have the module but to include the one or two lines of code in place of a call to the module. Rather than retain the 'bloaters' classification I will adapt it in my list of smells to become 'poor sizing', reflecting COBOL modules where the size and function are too small or too large and maintenance is hindered as a result.

3.3.2 Change Preventers

Mantyla et al. define change preventers as two opposite but equally problematic features, one where a single change will affect many classes, and the other where a single class is likely to be affected by a number of types of change. This problem is equally true in COBOL; a change may impact many modules or a module may be affected by many changes. If code is written is such a way, then it can be said to have a change preventer.

3.3.3 Dispensables

This category is described by Mantyla et al. as “something unnecessary which should be removed from the code”. This describes the problems caused by duplicate, dead or redundant code, all of which make maintenance more difficult by leaving in place code which is not needed, unnecessarily adding to the code to be maintained. The reasons for duplicate code or ‘software clones’ is well researched (Ducasse et al., 1999; Johnson, 1994; Baxter et al., 1998) and Baker (1995) found evidence that over 10% of code is ‘cloned’ and could be removed by rewriting. The removal of such code in structured systems has been one of the aims of automated transformation systems (Maher & Sleeman 1983). Redundant code often occurs as a result of cloning and is code that is executed but has no effect on the output of a program, while dead code is the term applied to code that is never executed (Wikimedia Foundation Inc., 2006). Both of these are problematic for maintenance as the maintainer will try to understand the use of code which is in fact not of any use. Dead code may be removed from the execution of a program by the compiler (Knoop et al., 1994) but this does not remove it from the source code to be maintained.

3.3.4 High Coupling

Mantyla et al. classified some of the ‘bad smells’ as ‘couplers’, describing where classes are highly coupled and are essentially too intimately related to each other. High coupling is also a problem in structured programming and is described in many of the studies and papers which introduced structured methods in the 1970s. Stevens et al. (1974) suggest that the “fewer and simpler the connections between modules, the easier it is to understand”, hence a loosely coupled system will be considered easier to maintain. Yourdon and Constantine (1979) describe an easily maintained system as one “in which one can study any one module without having to know very much about any other modules”. A system which exhibits the opposite characteristics is highly coupled and considered more difficult to maintain. Figure 3-3 shows an excerpt of COBOL source code from the ADVRCNL5 software in this study which may be considered to display coupling through its use of the COPY statement to source code of another module.

```
******************************************************************************
*
*     DESCRIPTION        : Get second part of PAN which may vary in length
*
******************************************************************************
 get-second-part SECTION.

   PERFORM WITH TEST BEFORE
   VARYING ws-acc-no-width FROM 8 BY 1
   UNTIL track2 OF atm OF ilf (ws-acc-no-width:1) = "="
      OR track2 OF atm OF ilf (ws-acc-no-width:1) = "D"
      OR ws-acc-no-width > 40

      CONTINUE

   END-PERFORM

   IF WS-ACC-NO-WIDTH > 40
     SET unknown TO TRUE
     SET bad-track2 TO TRUE
     PERFORM write-unknown-record
   ELSE
     SET known TO TRUE
```

```
      SUBTRACT 8
      FROM ws-acc-no-width
      ON SIZE ERROR
         MOVE "Arithmetic overflow on PAN account width count"
           TO log-text OF ws-eventlog-message
         PERFORM report-event
      END-SUBTRACT
    END-IF

    MOVE track2 OF atm OF ilf (8:ws-acc-no-width)
      TO ch-second-part OF rcon-line-lis5

     .
  get-second-part-exit.
    EXIT.

*****************************************************************************
*
*     DESCRIPTION        : Performed when there is an I/O error.
*
*****************************************************************************
  report-event SECTION.

     COPY log-event-2 OF "=BRIT_COPYLIB".

     IF NOT ws-return-after-msg
     THEN
        PERFORM cobol85-completion
     END-IF
     .

  report-event-exit.
     EXIT.

*****************************************************************************
*
*     DESCRIPTION        : Pad ILF sequence num with leading zeros
*
*****************************************************************************
  pad-seq-num SECTION.


     MOVE seq-num OF atm OF ilf
       TO ws-seq-num-chk
     MOVE ZERO TO ws-num-digits
     INSPECT ws-seq-num-chk TALLYING ws-num-digits
         FOR CHARACTERS BEFORE INITIAL " ".
     MOVE ZEROS TO ws-padded-seq-num
     IF ws-num-digits < ws-seq-num-len
        COMPUTE ws-seq-pos =
                ( ws-seq-num-len - ws-num-digits + 1 )
        END-COMPUTE
     ELSE
        MOVE 1 TO ws-seq-pos
     END-IF
     MOVE seq-num OF atm OF ilf
       TO ws-padded-seq-num( ws-seq-pos:ws-num-digits )
     .

  pad-seq-num-exit.
     EXIT.
```

Figure 3-3   Extract from ADVRCNL5 source code to demonstrate coupling

3.3.5 Masking by Comments

One of Fowler & Beck's (1999) 'bad smells' which was not classified by Mantyla et al. was the comments smell, which defines where poor code is being explained by comments. This problem exists if the code could be better written and would not then need the comments. I will describe this as 'Masking by comments'. Figure 3-4 shows a COBOL example from TXMATCHS of the use of comments which may be considered to be masking poor code.

```
21-PROCESS-NEXT-REC.

   MOVE 0 TO MATCH-FOUND-FLAG
             LAST-TRY-FLAG.

**SET KEY
   MOVE ISSR-INST-ID    OF MCE-PRESMNT-REC(6:6)
     TO IKEY-MCE-INST-ID OF LINK-ISS-REC.
   MOVE PAN OF MCE-PRESMNT-REC
      TO IKEY-CARDHOLDER OF LINK-ISS-REC.
   MOVE DATE-TIME OF MCE-PRESMNT-REC(1:6)
      TO IKEY-DATE-012 OF LINK-ISS-REC.
   MOVE ZEROES TO IKEY-ISS-COMPL-AMT OF LINK-ISS-REC.
   MOVE 1240 TO IKEY-MSG-TYP OF LINK-ISS-REC.

   MOVE ZERO TO MATCH-FLAG.

**TERM ID AND RET REF NUMBER NOT AVAILABLE

   IF TERM-ID OF MCE-PRESMNT-REC = "ZZZZZZZZ"
    AND RET-REF-NO OF MCE-PRESMNT-REC = "ZZZZZZZZZZZZ"
      MOVE 1 TO MATCH-FLAG
   END-IF.

**RET REF NO NOT AVAILABLE

   IF MATCH-FLAG-NOT-SET
      IF RET-REF-NO OF MCE-PRESMNT-REC  = "ZZZZZZZZZZZZ"
         MOVE TERM-ID OF MCE-PRESMNT-REC TO IKEY-TERM-ID-041 OF LINK-ISS-REC
         MOVE ZEROES TO IKEY-RET-REF-NO-037 OF LINK-ISS-REC
         MOVE 2 TO MATCH-FLAG
      END-IF
   END-IF.

   IF MATCH-FLAG-NOT-SET

**TERM ID NOT AVAILABLE
      IF TERM-ID OF MCE-PRESMNT-REC = "ZZZZZZZZ"
         MOVE 3 TO MATCH-FLAG
      ELSE
**TERM ID AND RET REF NUMBER BOTH AVAILABLE
```

```
      MOVE TERM-ID OF MCE-PRESMNT-REC TO IKEY-TERM-ID-041 OF LINK-ISS-REC
      MOVE RET-REF-NO OF MCE-PRESMNT-REC(7:6)
         TO IKEY-RET-REF-NO-037 OF LINK-ISS-REC
      MOVE 4 TO MATCH-FLAG
   END-IF
END-IF.

MOVE ZERO TO END-OF-LINK-ISSR-FLAG.

IF MATCH-FLAG = 2 OR 4
   START LINK-ISSR
      KEY NOT LESS THAN LINK-ISS-KEY OF LINK-ISS-REC
         NOT INVALID KEY
            PERFORM 21A-PROCESS-REC
END-IF.

IF NOT MATCH-FOUND
   PERFORM 211-IMPERFECT-MATCHING
END-IF.
```

Figure 3-4 Example from TXMATCHS showing possible masking by comments

3.3.6 Breach of standards

Standards make a program more understandable and therefore easier to maintain, defining naming conventions as well as preferred and allowable language forms. Glass & Noiseux (1981) suggest that although standards are generally dictated by management, the main beneficiaries are the maintainers. They also highlight the fact that standards need to be carefully considered as poor standards can be more damaging than no standards at all. Standards are often unique to an organisation and although their worth is widely debated, considered, pragmatic standards are generally thought to be beneficial (Ledgard & Cave, 1976). Maintainers used to an organisation's standards will find maintenance easier if those standards are followed. This smell will therefore be easier to judge for someone familiar with the standards of the specific organisation.

3.3.7 Aggressive Programming

This smell exists where the code exhibits a lack of defensive programming techniques. Defensive programming helps developers to find problems and prevents errors in production by validation of error conditions and data values to ensure that all eventualities are catered for. The advantage this brings to maintenance activity is in identifying or removing unsafe code, making errors easier to capture, and avoiding unexpected results by the use of safe techniques (Glass & Noiseux, 1981). Linderman (1982) describes some examples of these techniques in COBOL, advocating the avoidance of the CORRESPONDING option with MOVE or ADD, and advising the use of COMPUTE over MULTIPLY or DIVIDE. Other examples of defensive programming in COBOL include the use of success checking following file operations and size error checks following an arithmetic statement such as ADD. Figure 3-5 shows examples of both defensive and aggressive programming in COBOL for both file operations and an arithmetic operation.

Defensive Programming

```
           READ FILE-A NEXT.
           IF NOT SUCCESSFUL
                PERFORM Z1-LOG-ERROR.

           ADD 1 TO WS-COUNT
              ON SIZE ERROR PERFORM Z0-SIZE-ERROR.
```

Aggressive Programming

```
           READ FILE-A NEXT.

           ADD 1 TO WS-COUNT.
```

Figure 3-5 Examples of defensive and aggressive programming

Defensive programming could have been considered within the breach of standards smell, but I have chosen to keep it separate, as standards are unique to individual organisations and do not have to include the defensive programming techniques.

3.3.8 COBOL structure mines

Veerman & Verhoeven (2006) describe what they call COBOL 'mines', a name introduced by Field & Ramalingam (1999). These are essentially problems within COBOL software stemming from poor structure and control flow. They focus on the use of GO TO and PERFORM THRU statements. The dangers of the GO TO statement have long been identified (Dijkstra, 1968), and GO TO was effectively removed from COBOL in the ANSI standards of 1985 (ANSI, 1985); however, some historical code containing GO TO will still exist in many organisations and compilers do still allow its use. The problems created by GO TO, PERFORM THRU and other techniques which lead to poor control flow are highlighted by code which has multiple entry or exit points in any section; well-structured code should exhibit single entry and exit points. Programs which avoid these problems are described perfectly by Van Gelder (1977) as programs "in which loops and conditional sentences are properly nested and entered only at their beginning… There must be no GO TOs from one process to another. Further, each PERFORM in one process must refer to another process that appears later in the program". Figure 3-6 shows an example of the use of GO TO in OMFATDS, which was studied in the survey.

```
A100-PROCESS-OMF SECTION.
MOVE SPACES TO SSV-REC.
MOVE ";" TO SSV-SEP1 SSV-SEP2 SSV-SEP3 SSV-SEP4 SSV-SEP5 SSV-SEP6
            SSV-SEP7 SSV-SEP8 SSV-SEP9
ADD 1 TO ws-records-read-total.
IF APPL-CDE OF OMF-LAYOUT = ATDD1-APPL-CDE-C
   OR APPL-CDE OF OMF-LAYOUT = ATDS1-APPL-CDE-C
   OR APPL-CDE OF OMF-LAYOUT = TDF-APPL-CDE-C
   ADD 1 TO ws-records-read-atd
   MOVE LNET OF OMF-LAYOUT TO SSV-LN
   MOVE user-num OF omf-layout TO ws-num-4
   MOVE ws-num-4 TO ws-group
   MOVE user-num OF omf-layout TO ws-num-4
   MOVE ws-num-4 TO ws-user
   MOVE ws-who-changed TO SSV-WHO
   MOVE fm-dat OF omf-layout TO SSV-WHEN
ELSE
   GO TO A110-READ-NEXT.

IF APPL-CDE OF OMF-LAYOUT = ATDD1-APPL-CDE-C AND
   FM-TYP OF OMF-LAYOUT = ADDITION THEN
             PERFORM B100-ADDITION
ELSE IF APPL-CDE OF OMF-LAYOUT = ATDD1-APPL-CDE-C AND
        FM-TYP OF OMF-LAYOUT = UPDATE-BEFORE-IMAGE THEN
               MOVE omf-layout TO WS-OMF-DYNAMIC-DATA-NCR-BEFORE
ELSE IF APPL-CDE OF OMF-LAYOUT = ATDD1-APPL-CDE-C AND
     FM-TYP OF OMF-LAYOUT = UPDATE-AFTER-IMAGE THEN
               PERFORM C100-CHANGE
ELSE IF APPL-CDE OF OMF-LAYOUT = ATDD1-APPL-CDE-C AND
     FM-TYP OF OMF-LAYOUT = DELETION THEN
               PERFORM D100-DELETION
ELSE IF APPL-CDE OF OMF-LAYOUT = ATDS1-APPL-CDE-C AND
     FM-TYP OF OMF-LAYOUT = UPDATE-BEFORE-IMAGE THEN
               MOVE omf-layout TO WS-OMF-STATIC-DATA-NCR-BEFORE
ELSE IF APPL-CDE OF OMF-LAYOUT = ATDS1-APPL-CDE-C AND
     FM-TYP OF OMF-LAYOUT = UPDATE-AFTER-IMAGE THEN
               PERFORM C100-CHANGE
ELSE IF APPL-CDE OF OMF-LAYOUT = TDF-APPL-CDE-C AND
     FM-TYP OF OMF-LAYOUT = ADDITION THEN
               MOVE REC-IMAGE OF OMF-LAYOUT TO WS-OMF-TDF
               PERFORM G100-TDF-ADDITION
ELSE IF APPL-CDE OF OMF-LAYOUT = TDF-APPL-CDE-C AND
     FM-TYP OF OMF-LAYOUT = DELETION THEN
               MOVE REC-IMAGE OF OMF-LAYOUT TO WS-OMF-TDF
               PERFORM H100-TDF-DELETION
ELSE IF APPL-CDE OF OMF-LAYOUT = TDF-APPL-CDE-C AND
     FM-TYP OF OMF-LAYOUT = UPDATE-BEFORE-IMAGE THEN
               MOVE omf-layout TO WS-OMF-LAYOUT-BEFORE
               MOVE REC-IMAGE OF OMF-LAYOUT TO WS-OMF-TDF-BEFORE
ELSE IF APPL-CDE OF OMF-LAYOUT = TDF-APPL-CDE-C AND
     FM-TYP OF OMF-LAYOUT = UPDATE-AFTER-IMAGE THEN
               MOVE REC-IMAGE OF OMF-LAYOUT TO WS-OMF-TDF
                           PERFORM I100-TDF-CHANGE.

A110-READ-NEXT.
READ omf-file NEXT RECORD
    AT END
    MOVE 1 TO ws-file-read.
```

Figure 3-6 Example of a COBOL Structure Mine in OMFATDS

In describing the smells to the developers in the survey, I did not want to be too prescriptive and detailed in the descriptions of the smells to ensure that the subjective element was still the dominant focus of the survey. The descriptions given were simply guidance to the developers giving the general characteristics of each smell for consideration in the evaluation. Appendix A contains the descriptions of the smells given to developers. These descriptions are less formal than those above to help the participants in the survey to understand them. The descriptions were reviewed by a colleague of a similar level of experience and knowledge to the intended participants and revised based on comments made by that reviewer.

3.4 Subjective Evaluation

The final element of the research focuses on the subjective evaluation of the software. In order to facilitate this subjective evaluation, a two-part survey of a group of developers was conducted to ascertain their opinions of the maintainability of the software. There are several existing studies which have employed these methods. Kataoka et al. (2002) and Mantyla and Lassenius (2006) used subjective evaluation and the opinion of developers as part of investigations into software maintenance and Shneiderman (1980) studied the use of peer reviews in software maintenance and described the environment in which the reviews were carried out. His study suggested peer reviews were productive but did identify potential bias in some of the responses. The first part of my survey is designed to gain an understanding of the developer's initial subjective view of the quality and maintainability of the software, while the second part asks for evaluation based on the subjective assessment of the existence in the software of the smells outlined in Section 3.3.

3.4.1 Developing the Survey

The survey collected the participant's name, and asked for their opinion of their experience in working with COBOL and in software development in general. This experience is measured on a Likert scale of 1-5, from not very experienced to very experienced. The notes provided with the survey assured the participants of the confidentiality of the survey and of the ways in which their responses will be used; however, the completion of the name section was optional to ensure that if any of the participants were uncomfortable with their opinions being known or identified, then they could complete the survey anonymously. Kitchenham & Pfleeger (2002) debate whether this type of question should appear at the start or end of a survey, as having them at the start of the survey may discourage the respondents; however, in the case of my survey which used invited and willing participants, I considered such precautions unnecessary.

The first section of the survey introduced the survey and its aims, and then collected the participant's opinion of each of the modules provided for review. The questions asked for their familiarity with the code and their opinion of the maintainability of the piece of software, both on a numerical scale of 1-6. They were also asked whether they had ever made a change to that piece of code to help investigate the possibility of bias or simply a feeling of understanding where a developer has already worked with the software. The final question in the first part of the survey the participants ranked the software in order of difficultly to maintain to help identify whether the views were consistent across the developers relative to their experience and previous working understanding of the software.

The second section of the survey followed up the opinions gathered in the first section by introducing the COBOL ‘bad smells’ defined in Section 3.3 and asking the developers opinion as to the existence of each of the smells in each of the individual pieces of code reviewed. This was also measured on a numerical scale of 1-6. For each piece of code, an additional question asked whether the participant’s view of the overall maintainability of that piece of code has changed. This was measured using a seven-point Likert scale, from ‘much easier’ to maintain to ‘much harder’ to maintain. This scale differs from that used in most of the other questions, where I chose to use a scale with an even number of possible responses, as I was concerned that the questions would receive a large number of neutral answers. The Likert scale is recognised to have the disadvantage of allowing a neutral answer (Rea and Parker, 2005; Fink and Kosecoff, 1998); the use of an even-numbered scale encourages the participant to answer either positively or negatively. The final question in the second part of the survey used a five-point Likert scale to gather the developers’ opinion of whether each of the smells contributes to maintainability. This allows analysis of whether a subject’s own opinion of a smell may influence how much they feel it exists, and whether it makes code more maintainable. This question also helps validate my taxonomy of smells for COBOL.

A trial of the survey was carried out with a manager within my organisation who was not one of the final participants in the actual survey. This trial allowed me to check that the questions were unambiguous and that the survey was practical to carry out and administer. One of the significant risks identified in survey and questionnaire research is non-response, especially in mail surveys (Oppenheim, 1992), but I felt I could avoid this by having an invited group of participants who had agreed to take part. I chose to provide the survey to the participants in electronic form and allow them two weeks to complete the survey in their own time, rather than the alternative of arranging a time and place for all of the participants to gather and complete the survey, having been provided the necessary review materials in advance. A copy of the survey was sent to the respondents by e-mail, along with the necessary guidance notes, smells document, and copies of the source code to be reviewed. Appendix B shows the survey that was sent to the developers. I chose this method, as I felt this would allow for a more relaxed approach and not introduce a time constraint. The downside of this approach, was that it did not ensure completion of the survey within the timescales, and even among a small group of respondents, I needed to chase up some of the responses.

3.4.2 The participants in the subjective evaluation survey

The survey was carried out with six of the team of developers within my organisation. The participants have worked with the selected software code to varying degrees. Most of the developers have worked in this type of environment for some time, so an effort was made to involve the least experienced member of the team to help investigate the affect of experience on the evaluation. The remaining participants were selected for their availability and willingness to take part in the survey.

Chapter 4 Results

This chapter describes the results of the research into software metrics and the surveys of developer opinion on the software chosen for this project. The analysis of these results will enable a validation of the hypotheses from Chapter 2 and will provide an answer to the research question.

4.1 The Software Metrics

The results of the software metrics applied to each of the nine pieces of software being evaluated are shown in Table 4-1. These results show that every one of the metrics identify the same program, MBREXTS, as being the easiest to maintain, and a review of the rankings of the code for each of the metrics shows that in many cases the rankings of the software are similar. There are however, some results which stand out. When ranking the code according to these metrics from 1 (easiest to maintain) to 9 (hardest) for each metric, there are generally only small ranges of up to 3 displayed by the rankings for each piece of code. The rankings for each piece of software are shown in Table 4-2. However, EAEPROG and OMFATDS have large ranges, and the ranking of those programs using Halstead's Difficulty measure are both anomalous, with OMFATDS ranked 3rd compared to the other metrics which ranged from 7th to 9th, and EAEPROG ranked 7th compared to the range of 2nd to 4th for the other metrics. Similarly VOUTPREP showed a ranking of 8th using Halstead's Vocabulary, while the other metrics ranged from 4th to 5th.

Metric / Software	Lines of Code	McCabe's Cyclomatic Complexity	Halstead's Length	Halstead's Vocabulary	Halstead's Volume	Halstead's Difficulty	Halstead's Effort
EAEPROG	521	33	926	210	7143.39	74.54	532499.13
ISSREVS	540	38	910	173	6765.51	42.09	284762.27
MBREXTS	273	8	273	110	1851.31	24.72	45767.23
OMFATDS	1562	109	3689	504	33117.19	38.24	1266527.08
ADVRCNL5	1710	110	2290	412	19892.09	97.01	1929761.01
RETREPS	1515	99	2278	420	19851.05	62.82	1247043.04
TXMATCHS	1360	122	2196	325	18324.07	88.03	1613047.71
VOUTPREP	1212	41	1991	433	17437.62	49.61	865225.54
NBSCON	726	44	1207	207	9286.04	52.94	491576.36

Table 4-1 Software Metric results for the software studied

These discrepancies suggest that although there are many metrics available for use, it can be shown that the choice of metric can significantly influence the relative view of the maintainability of software.

Metric / Software	Lines of code	McCabe's Cyclomatic Complexity	Halsteads Length	Halsteads Vocabulary	Halsteads Volume	Halsteads Difficulty	Halsteads Effort
EAEPROG	2	2	3	4	3	7	4
ISSREVS	3	3	2	2	2	2	2
MBREXTS	1	1	1	1	1	1	1
OMFATDS	8	7	9	9	9	3	7
ADVRCNL5	9	8	8	6	8	9	9
RETREPS	7	6	7	7	7	6	6
TXMATCHS	6	9	6	5	6	8	8
VOUTPREP	5	4	5	8	5	4	5
NBSCON	4	5	4	3	4	5	3

Table 4-2 Ranking the software according to the software metrics

Analysis of the correlation between the different metrics gives support to this finding. I chose to use the Pearson Product-Moment Correlation Coefficient to analyse the correlation between the metrics, as this is considered the optimal measure of correlation between two numerical variables (Fink, 1995). Throughout my analysis of the results, where correlation is described, it is described using the definition given by Fink (1995, p. 36), which is outlined in Table 4-3.

Correlation	Fink's Description
0 – 0.25	No or little relationship
0.25 – 0.5	Fair degree of relationship
0.5 – 0.75	Moderate to good relationship
0.75 – 1	Very good to excellent relationship

Table 4-3 The meanings of correlation coefficient values

This correlation can be used to see just how closely the results of the different metrics relate to each other. Table 4-4 shows the correlations between the metrics and highlights that the lines of code measure demonstrates an excellent relationship with each of the other metrics with the exception of Halstead's Difficulty; Halstead's Difficulty shows a weak relationship with some of the other metrics and only moderate relationships with any other metric. It is interesting to note that the relationship between some of the Halstead indicators are not as strong as might be expected for a series of indicators essentially based on the same variables (total number and distinct number of operators and operands).

	McCabe's Cyclomatic Complexity	Halstead's Length	Halstead's Vocabulary	Halstead's Volume	Halstead's Difficulty	Halstead's Effort
Lines of code	0.904083	0.901312	0.931197	0.89779	0.51638	0.93806
McCabe's		0.838293	0.747289	0.823136	0.599589	0.933624
Hal. Length			0.939355	0.998959	0.246053	0.777534
Hal. Voc.				0.944861	0.293258	0.783974
Hal. Vol.					0.22198	0.76704
Hal. Diff.						0.744298

Table 4-4 Pearson Product-Moment Correlation Coefficient of the seven metrics

The difficulty indicator shows correlation of less than 0.3 with the vocabulary, length and volume measures, essentially having little to no relationship with the other metrics. To a lesser extent the same could be said of the effort indicator. These results may seem surprising given that the indicators are derived from the same variables, but a closer investigation of the formula for difficulty shows this to be an understandable scenario. The variation in correlation between the different metrics shows the difficulty in using metrics alone to measure maintainability. A key question to be asked by an organisation would be which metric they should use. As has been seen, many larger software organisations develop their own metrics specific to their environment, using various factors including those measured by the metrics in this study (Coleman et al., 1995). These results suggest that no single metric could be recommended to suggest maintainability; however a combination of metrics could be useful.

4.2 Subjective Evaluation

The survey carried out with the developers allows analysis of a number of elements of developer intuition, allowing a review of the potential uses of subjective evaluation in assessing maintainability. Appendix C summarises the answers given in the surveys completed by the six developers. The clearest indication of the relative views of maintainability is given by Question 12 of the survey which was designed to collect the developers' opinions of the relative maintainability of the software. This was achieved by asking for a ranking of the software in order of maintainability. Analysis of the responses to this question shows that there was much greater variation in the results for each piece of software when using subjective evaluation than there was for the software metrics. Only two of the pieces of code reviewed (OMFATDS and ADVRCNL5) had a smaller range of values and standard deviation for their rankings among the six developers than among the seven metrics. This finding suggests that the opinions of developers are more variable than the results of the metrics and is further supported when looking at the ratings given by the developers in part b of Questions 3-11; on a scale of 1 to 6, seven of the nine pieces of code received scores ranging over at least 4 of the possible 6 values assessing maintainability. These findings of some diversity of opinion between the developers support the findings of Shneiderman (1980) in his studies of peer reviews.

Question 12 in the survey collected the rankings given to the software in terms of maintainability; by taking the average ranking across all developers, it is possible to rank the software to give an overall view from all of the developers. Similarly, by finding the average maintainability score in part b of Questions 3 to 11, it is possible to get the average score given to each piece of software and rank the software according to this. These rankings can be seen in Table 4-5.

Ranking / Software	Metric Ranking	Ranking based on Q12 of survey	Ranking based score given in Q3-11	Ranking based on view of smells
EAEPROG	3	5	3=	3
ISSREVS	2	1	2	1
MBREXTS	1	2	1	2
OMFATDS	8	6	6=	9
ADVRCNL5	9	9	9	6
RETREPS	6	7	6=	5
TXMATCHS	7	4	3=	4
VOUTPREP	5	8	8	7
NBSCON	4	3	5	8

Table 4-5 Ranking the software according to the different methods of evaluation

A comparison of these two methods of ranking using Spearman's rank correlation coefficient shows a very good correlation of 0.875, which helps to validate the consistency and credibility of the answers given by the developers in the survey. It is necessary to use Spearman's rank correlation coefficient in this case, rather than the Pearson Product-Moment Coefficient as the data being analysed here is ordinal data based on ranking rather than the actual numerical results which are better analysed using the Pearson measure.

4.2.1 Using smells to influence developer evaluation

The second section of the survey was designed to collect the developer's views on the existence of potential smells which I have suggested characterise code that is difficult to maintain. This process allows analysis of how these smells influence the views of developers and whether the existence of smells could be used to assess maintainability. The most important use of these results was to ascertain whether the developer's opinion changed as a result of considering the smells. In only one of the nine pieces of software reviewed did none of the developers' view's change (ISSREVS), while in six of the nine cases at least half of the developers changed their view. This finding suggests that the use of the smells does influence developers and could help to get a better view of maintainability. Figure 4-1 shows that the developers view changed 42.6% of the time however, 33.3% of these changes were slight, the maintenance was viewed as harder or easier only 9.3% of the time, and never much harder or much easier. This finding suggests the impact of using the smells may not be as strong as it first appeared. It is also interesting to note that when the developer's opinion was influenced by the use of the smells, the view was almost twice as likely to be that maintenance was harder rather than easier.

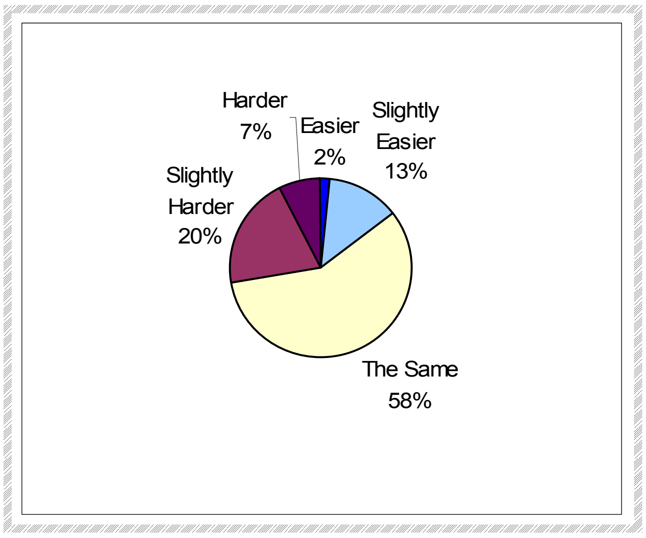

Figure 4-1 The change in the developers opinion of maintainability after considering the code smells

Further analysis can be done to compare the views on smells with the original opinions by obtaining a ranking using the scores given to the existence of smells in each piece of software and ranking them based on the level of smells believed to exist by the developers (see Table 4-5 for the ranking). In doing this, no weighting is given to any particular smell; they are all considered to have the same influence. This ranking can be compared to the two rankings based on the initial subjective evaluation to see how well they correlate. This comparison shows that there is a relatively good, but not excellent Spearman's correlation coefficient of 0.6125 against the ranking based on Question 12, and a coefficient of 0.7667 against the ranking using ratings in Questions 3 to 11. This method of using the average score of a group of developer's concerning their views on the existence of smells could be considered for use as an additional subjective metric within an organisation; however, getting the required number of developers to review every piece of code developed may be an unrealistic proposition.

The final question in the survey was designed to validate the taxonomy of smells and also to discover whether the opinion on the validity of a smell influenced the developer's opinion of its existence. The responses showed that there was generally strong agreement with the use of most of the smells in determining the maintainability of code, with both the mode and median result being 'agree' that the smells do impact maintainability across the whole group of developers. This result also applies to each individual developer with the exception of one who had a mode of 'neither agree or disagree' but the median of 'agree'. As the median is considered the best measure when dealing with a small number of respondents, it can be said that all developers agreed in general with the smells. By reviewing the individual smells, it is clear that Breach of Standards is considered the most influential in affecting maintainability, followed by poor sizing and COBOL structure mines. For each of these smells, the evaluators either agreed or strongly agreed that they had an influence on maintainability. With three of the smells, some evaluators disagreed that the smell influenced maintainability. Both Aggressive Programming and Dispensables had one evaluator who disagreed but still displayed an overall agreement, while Masking by Comments did not have general support as an influence, and had as much disagreement as agreement. This suggests Masking by Comments is not a valid smell when evaluating maintainability.

Having introduced guidelines for use by the developers, it would be interesting to ascertain whether the developers' view's on the use of the smell in determining maintainability affected their ability to identify its existence. Where developers showed a level of agreement that a smell contributed to poor maintainability, they were more likely to identify it, giving an average score of 2.46 to the existence of evidence of the smell, compared to an average of 1.83 where they disagreed and only 1.54 where they were neutral. It is also true that the two smells said to be displayed most in the software (Breach of Standards and Poor Sizing) were also the two with the most support among the developers, whereas the least supported (Masking by Comments) was also said to be the least evident. This could suggest that a developer is more likely to find a smell if they believe it has an influence, or conversely they may agree that the smell has an effect if they see evidence of it in the code being reviewed.

4.2.2 Comparison with the software metrics

One of the key aims of this project was to investigate the relationship between subjective evaluations and software metrics. Having collated the software metrics for the code being reviewed, a comparison with the subjective evaluation survey produces some interesting results. The software metrics can be compared to the outcome of the subjective evaluation by creating rankings based on the results in Table 4-1. Taking the average of these rankings across the seven metrics gives an overall metric ranking, which can be compared to the rankings based on evaluations (See Table 4-5). This comparison shows that the ranking based on Question 12 shows a very good Spearman correlation to the metrics of 0.9125, while the relationship between the metrics and the ranking based on Questions 3-11 and the ranking based on the smells still showed good, but not as strong correlation (0.775 and 0.65 respectively).

Reviewing the subjective evaluation rankings against the rankings created by each individual metric again shows the variation in the results given by the different metrics. The findings are shown in Table 4-6 again using Spearman's rank correlation coefficient. Many of the correlations are between 0.7 and 0.8 showing good correlation, but again the Halstead Difficulty indicator shows the weakest relationships with correlation as low as 0.183.

Evaluation / Metric	Evaluation Ranking (Q12)	Ranking based on maintainability rating (Q3-11)	Ranking based on existence of smells
Lines of Code	0.9375	0.633	0.7625
McCabe's Cyclomatic Comp.	0.746	0.517	0.575
Halstead's Length	0.929	0.717	0.8
Halstead's Vocabulary	0.846	0.717	0.633
Halstead's Volume	0.929	0.717	0.8
Halstead's Difficulty	0.604	0.183	0.492
Halstead's Effort	0.879	0.567	0.717
Average	0.839	0.683	0.579

Table 4-6 Spearman's correlation of the subjective evaluation against the metrics

It is also interesting to note that the ranking based on Question 12 shows significantly better correlation with the rankings based on metrics (with an average correlation of 0.839) compared to the rankings based on the ratings in Questions 3-11 and the rankings based on the existence of smells, which averaged 0.683 and 0.579 respectively, supporting the relationship with the average of the rankings given by the metrics. This finding suggests that the initial ranking of the developers is closer to that given by the software metrics than any ranking based on scores derived from the subjective evaluation carried out by the developers.

4.2.3 The effect of program age on maintainability

One of the factors suggested as a contributor to increased maintenance requirements for software is program age. Vessey and Weber (1983) suggested this was a factor but were unable to show any great support for this in their investigations. An analysis of the results of the subjective evaluation and software metrics do show a general reduction in maintainability as the age of software increases, with the average rankings of the software increasing with age when measured by both metrics and the three methods of subjective rankings described in Section 4.2.2. Figure 4-2 shows the average metric and subjective ranking of the software increasing according to the age of the software.

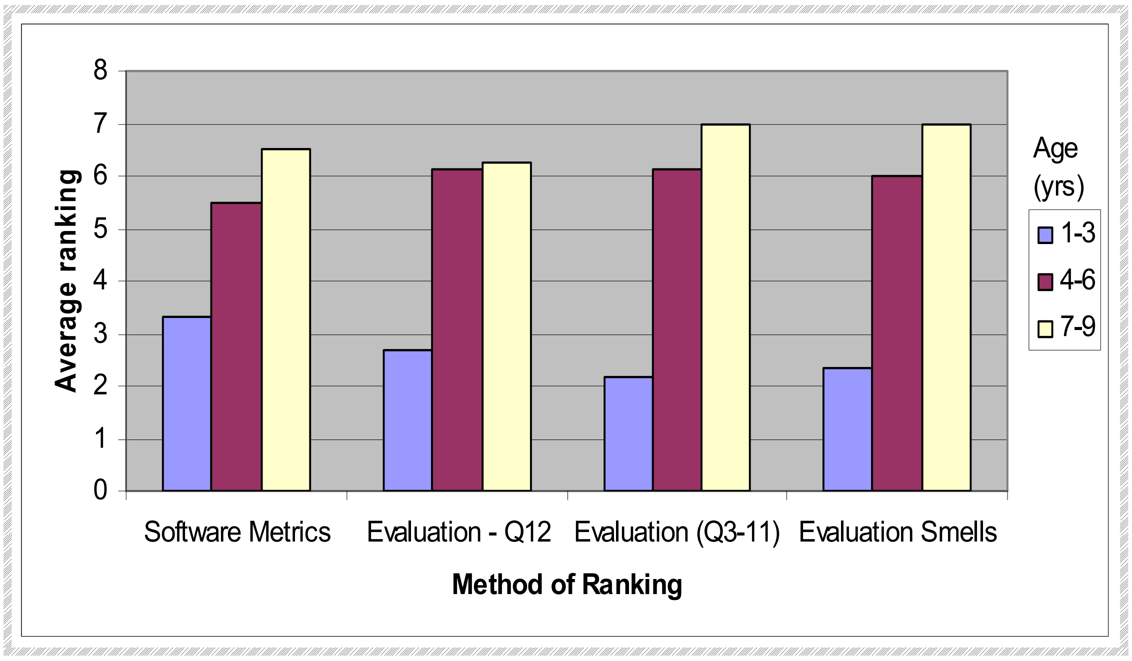

Figure 4-2 Rankings of the software according to program age

4.2.4 The developers' knowledge of the software and maintainability opinion

One of the elements of using subjective evaluation which gave some concern was the possibility of biased opinions based on the developers having worked with the code previously and having therefore contributed to its maintainability in a practical way. Reviewing the results of the survey using part (c) of each of the Questions 3-11 shows that the average difficulty to maintain rating given to software by somebody who had not worked on the software before was 3.091, while the average rating given to the software by those who had worked on a piece of software before was 3.9. This finding would suggest that if anything, developers who have worked on software tend to believe that software is more difficult to maintain than those who have not. This result allays fears of bias towards the software, but suggests either a more critical eye being cast on the software based on the experience of updating that software, or possibly a desire to avoid appearing biased. It is of course possible that this is just a statistical anomaly, given the size of the sample in this study and the possibility that the developers wrote the code themselves, have experience of the style in which the code is written or have knowledge of the author of the code.

4.2.5 Developer experience and the effect on the subjective evaluation

Vessey and Weber also suggested that programmer experience was a possible factor in the maintainability of code. They looked at this issue from the point of view of programmers with more experience writing code that is more maintainable; however, it is also possible that the experience of a programmer would affect the way they evaluate code for maintainability and how the introduction of smells may influence their views. There is a suggestion based on the results of this evaluation that experience does lead to a view of maintenance being easier; however, the support for this suggestion from the survey is fairly weak. Those developers who would fall into the 'most experienced' category, those with more than 7 years' experience in both COBOL and software development gave an average difficultly to maintain rating of 3.16, whereas those in the less experienced category (less than 7 years' experience) gave an average of 3.39, indicating on average they considered the software slightly harder to maintain than their more experienced counterparts.

It is also possible to consider how the experience of a developer may affect the level to which they are influenced by or willing to accept the smells suggested when categorising poor software. The less experienced group amended their view of a piece of software 66.6% of the time, whereas the more experienced developers amended their view only 30.6% of the time, suggesting that less experienced developers may be more influenced by the suggestion of a smell, or may not have considered the smell originally, and find the smells useful in determining the maintainability of the software. This finding indicates that the use of smell guidelines in measuring maintainability may be particularly advantageous as an aid to inexperienced developers involved in maintenance activities.

4.3 Hypotheses and Research Question

The review of literature led to the suggestion of two hypotheses in answer to the research question. The first hypothesis suggested that metrics can be used to supplement and corroborate the opinions of developers as to the maintainability of software. The analysis of the results supports this hypothesis to a large extent. Each of the methods of measuring the maintainability of software showed at least a good relationship with the combined results of the software metrics and in the majority of cases the relationship with each of the individual metrics is a good relationship or better. These findings support the suggestion that metrics can be used to corroborate the opinions of developers. Although there are good relationships between developer opinions and the software metrics, many of the relationships fall short of being very good or excellent. This finding shows that there are some differences between developer opinion and the metrics, demonstrating that metrics can provide a different perspective on maintainability and therefore add another dimension by supplementing developer opinions with a more formal approach.

The second hypothesis was that in addition to source code, other factors such as program age and developer experience will have an influence on the maintainability of software. Program age was suggested as a factor in software maintainability by Vessey and Weber (1983), but they were surprised to be unable to support this suggestion with evidence. The evidence from the results of both metrics and subjective evaluation shows that older code is less maintainable than more recent code, suggesting that age is a factor in making software more difficult to maintain. Additionally I investigated the influence that a developer's exposure to a piece of software and their level of development experience has on their perception of software maintainability. The suggestion from the survey is that exposure to a piece of software will lead to a developer being more critical of the maintainability of the software. On the other hand, I have also been able to show that in general, a more experienced developer will have a more favourable view of the maintainability of a piece of software, although the difference in view from those less experienced is not as clear as might have been expected. This finding does, however, lend a little support to the view that the experience of a developer influences their view on the maintainability of software, adding further support to my second hypothesis, which was formulated from a review of the existing literature.

The main purpose of this project was to answer the research question. The evidence presented has helped to validate the hypotheses proposed following the literature review and to begin to answer the research question. It has been shown that it is generally true to say that software metrics support the subjective evaluation of developers, and given the correct choice of metric or metrics, it is possible to use metrics to validate subjective evaluation. Metrics can provide more than just corroboration of evaluations carried out by developers, by supplementing the evaluations with more formal measures, they can ratify the opinions of the developers given the existence of variations in their opinions. The support for the second hypothesis has shown that there are a number of factors beyond the quality of the source code which play a role in the maintainability of software, such as the age of the source code and developer experience (both in past maintenance and of future maintainers). The influence that additional factors such as these have on maintainability shows that metrics based entirely on the source code cannot alone be used to measure maintainability. The lack of widely available metrics which can measure factors beyond the source code indicates that it will always be necessary to obtain the subjective view of developers, who are more flexible than the metrics and who can take additional factors into account. Formal metrics should be used to corroborate subjective evaluation, especially where a single developer or inexperienced developers are undertaking the subjective evaluation, as the opinions of developers have been shown to have some variance. This variance suggests that maintainability is in itself inherently subjective. Changes to software are carried out by individual developers and it is the current developers' opinion of maintainability which is important to a change, as metrics cannot be used to reflect the experience and preferences of the developer asked to make changes.

The wide range of factors which can influence maintainability have been difficult to capture in a single measure so it is necessary to use subjective evaluation to gain a view of maintainability and to get support for the evaluation by using metrics. Either these metrics could be the commonly available source code metrics to provide corroboration, or a metric or metrics tailored specifically for an individual software environment, taking into account the factors appropriate to that environment. In more mature organisations where metrics are already in use, it will be easier to evolve the metrics to adapt to the individual environment and subjective evaluation may receive less emphasis.

4.3.1 Analysis of methodology

The survey has provided some good results in studying the measurement of software maintainability; however, there are several elements of the survey and my research which could be improved upon in future studies. The method for carrying out the survey proved to be less effective than I had anticipated, even with a group of participants who had agreed to take part; the problem of non-response did materialise and I needed to chase the responses which took longer than anticipated. It would have been preferable to have more than six developers answering the survey, but the organisation being studied has a relatively small development team, most of who were involved in this study. Having more developers would have enabled more confidence in the statistical relevance of the answers and would also have meant that the problem of non-response would not have been so important, as one or two late or failed responses would not have been as important if the number of participants had been greater. Six responses was probably the minimum required to get reasonable results from the survey, so late or non-response could have been a significant issue, whereas the use of a larger group could have removed these problems. The subjective evaluation could also have been improved by including a number of ‘independent’ developers among the participants, which would have added to the number of participants, and would also have provided the opinions of developers who were genuinely independent and whose opinions could not have been influenced by previous experience of the software or knowledge of the developers of the software.

The collection of metrics was another area of my research where there were aspects which could be improved on. I was able to obtain the required data for analysis, but the collection of the Halstead indicators was very time-consuming as no tools were readily available to collect the metrics for the software being used. If the finances had been available, it would have been desirable to collect these metrics automatically using a tool and to extend the number of metrics studied.

Chapter 5 Conclusions

This chapter summarises the conclusions of the research I have carried out, considering the implications for the problem domain and the practical uses of the results..

The aim of this study was to investigate the relative merits of subjective evaluation and software metrics in the maintenance of COBOL software. The study concentrated on some of the COBOL software in a small software development team in a U.K. financial organisation. The subjective evaluation was carried out by surveying six developers from within the team, while the metrics analysis concentrated on seven commonly researched source code metrics. It is clear from earlier research that both subjective evaluation and software metrics are considered suitable for measuring the maintainability of both structured and object-oriented software. The study focussed specifically on the maintenance of COBOL software and how the various methods of measuring maintainability can be used. It has been shown that there is a very good relationship between subjective evaluation and most of the common software metrics when used to measure the maintainability of COBOL software. Similarly there is strong correlation among most of the individual metrics, which actually show lower variation than subjective evaluation, where there is more significant variation in opinion of maintainability. Of the metrics studied only one, Halstead's Difficulty did not show a good relationship with the subjective evaluation or with the other metrics researched, and does not look suitable for use in measuring maintainability.

Most software metrics focus on the semantics of the source code; this study has shown that it is not just the attributes of the source code which influence maintainability, identifying that program age and the experience of the developer both influence the maintainability of software. This finding corroborates the views of several other investigations which advocate the use of many factors in the investigation of software maintainability (Stark, 1996; Vessey and Weber, 1983; Wake and Henry, 1988). The influence of factors outside the source code syntax makes the subjective evaluation key to measuring maintainability, as the common metrics focus on source code; only one or two specialised maintainability metrics have been able to include additional factors and these tend to have been designed for specific environments. Metrics and tools can therefore best be used to help to support the developer's opinion. This view supports the findings of Kataoka et al. (2002) that tools can be used to support the work of developers. The metrics should therefore be to used corroborate and support developer opinion, but cannot be used alone to give a full view of software maintainability. Only in very mature organisations is it likely that metrics will have evolved to a point where they might replace subjective evaluation.

The use of smells has been shown to be of limited value in measuring maintainability. Evaluation of the smells outlined does support the subjective views of the developers, and with the exception of the Masking by Comments smell, all smells were supported by the developers as good indicators of poor maintainability. There does however, appear to be limited value in using the smells to aid the developers as the smells did not cause a significant change in the developers' opinions of the maintainability of the software. This was particularly true of the experienced developers and the use of smells should therefore be limited to guiding inexperienced developers who are involved in measuring maintainability. The benefits of metrics do suggest that developers should be trained in their use to aid their ability to evaluate code and create evidence to be used in support of their views.

Chapter 6 Future research

This study has shown that both subjective evaluation and software metrics should be used in combination to provide the best approach to measuring maintainability in COBOL software. There are various areas in which the methods used in this study could be improved upon, and a number of areas which deserve further investigation. Further investigations could focus on larger development organisations where more developers are available to contribute to the subjective evaluation. Alternatively independent developers from outside the organisation could be used to obtain additional views on the software being evaluated, or to create a larger sample of evaluators to give results with more statistical significance.

Additional research should also focus on extending the range of software metrics being studied, including metrics which take factors outside the source code into account. By studying additional larger and more mature organisations it should be possible to look at metrics which are already in use within organisations, including metrics developed specifically for that organisation which take the environment and other factors into account (e.g. HP Maintainability Assessment System). This further work would enable more conclusions to be drawn as to the best metrics to use in supporting subjective evaluation of software maintainability, and would also allow investigations to focus on the maturity of process within an organisation and whether this influences the role of metrics and subjective evaluation. It would also be interesting to apply this study to different types of structured software environment to investigate whether the results obtained from studying COBOL software could be applied to environments using structured languages such as PL/1 or PASCAL. Similarly, this approach could be extended to object-oriented software environments to build on this study and the approaches of Mantyla and Lassenius (2006).

Appendix A – Smell guidelines given to developers

COBOL ‘Smells’

The following ‘smells’ have been suggested as contributing to making COBOL code more difficult to maintain. Please ensure you have completed SECTION 1 of the questionnaire before reading these any further. These are required to have been read prior to completing SECTION 2. Your answers in this section should be based on your understanding of the smells as they are described here.

Poor Sizing

This occurs where modules of code have become so large as to be difficult to maintain. In COBOL a module can be a section or paragraph of code. However, size alone is not important it is the focus on building “small functionally oriented pieces”. It is also true that a module can be so small that it is better not to have the module, but to include the one or two lines of code in place of a call to the module. All of these scenarios are covered by this category.

Change Preventers

This covers two opposite but equally problematic features, one where a single change may impact many modules or a module may be affected by many changes as a result of the way the code is written. If code is written is such a way, then it can be said to have a change preventer.

Dispensables

A dispensable is something unnecessary which should be removed from the code. This describes the problems caused by duplicate, dead or redundant code. Redundant code often occurs as a result of cloning, and is code that is executed but has no effect on the output of a program, while dead code is the term applied to code that is never executed. Duplicate code is often created where a section of code is copied and reused in several places, when it could be used only once if parts of the code were rewritten. All of these features are problematic for maintenance as the maintainer tries to understand the use of code, which is in fact not of any real use.

High Coupling

Coupling is the level of dependency on other modules or files that software has. Where code is highly coupled or dependent, means that the code has too many or too complex links to other code or files. An easily maintained module will have simple obvious relationship with other code, whereas a more difficult to maintain module will be highly coupled, with complex or obscure relationships to other code and modules.

Masking by Comments

This category defines where poor code is being explained by comments. This problem exists if the code could be better written, and would not then need the comments.

Breach of standards

Standards make a program more understandable and therefore easier to maintain. Standards define naming conventions, as well as preferred and allowable language forms. Maintainers used to an organisations standards will find maintenance easier if those standards are followed. This 'smell' will therefore only be able to be judged by someone familiar with the standards of the specific organisation. Please judge this smell based on your knowledge of your organisations standards, however formal or informal they may be.

Aggressive Programming

This smell exists where the code exhibits a lack of defensive programming techniques. Defensive programming helps developers in finding problems, and also prevents errors in production by validation of error conditions and data values to ensure that all eventualities are catered for. Examples of defensive programming would be the use of a success check following an IO operation, or the use of ON SIZE ERROR following an arithmetic operation.

COBOL structure mines

These are essentially problems within COBOL software stemming from poor structure and control flow. They focus on the use of GO TO and PERFORM THRU type statements which allow the flow of code to jump from place to place in an unstructured manner. This is highlighted by code which has multiple entry or exit points in any section; well structured code should exhibit single entry and exit points. Programs which avoid these problems are described perfectly as – “a program in which loops and conditional sentences are properly nested and entered only at their beginning….There must be no GO TOs from one process to another. Further, each PERFORM in one process must refer to another process that appears later in the program”.

Appendix B – Questionnaire provided to developers

Developer Questionnaire.

Thank You for taking part in this survey. The aim of the survey is to provide data to a study investigating the merits of subjective evaluation in determining the maintainability of COBOL software.

Evaluation material

You have been provided nine different pieces of COBOL source code to review, which you may be familiar with to varying degrees. The code is packaged in folders which include any referenced libraries you may need, as well as the main source code. Your level of previous knowledge of the software is not important to this survey; the aim is to gather information concerning your familiarity and opinion of the maintainability of the different pieces of code. You have also been supplied with a list of COBOL 'smells'. As stated in the covering note, please do not read this document until you are prompted to do so in the questionnaire notes.

Please answer each question by marking the relevant box with a cross, preferably electronically by double clicking the box and marking it as checked.

The following section collects a few personal details about you and your experience of COBOL programming and software development. All answers in this survey will be confidential, and are being used only for the study of software maintainability. However, if you prefer to remain anonymous, then please leave the name blank, but complete the other sections of the survey, the results will still be of use.

Name:

1 a) How experienced do you consider yourself with COBOL software?

Not at all — Very experienced

1☐ 2☐ 3☐ 4☐ 5☐

b) How many years have you worked with COBOL?

Up to 3 years	3-7 years	7 years or more
☐	☐	☐

2 a) How experienced do you consider yourself in Software Development?

Not at all — Very experienced

1☐ 2☐ 3☐ 4☐ 5☐

b) How many years have you worked in Software Development?

Up to 3 years	3-7 years	7 years or more
☐	☐	☐

<u>SECTION 1.</u>

Based on your knowledge of each of the following modules of COBOL code, and where necessary referring to the source code provided for review, please answer each of the questions which will require you to give your opinion on the **ease of understanding and maintenance** of each piece of code. There are no right or wrong answers to these questions; it is your subjective view that is important. Do not feel that you have to understand every line of the code to answer these questions; this is about your perception of the code.

3. EAEPROG

a) Prior to reviewing the source code for this survey, how familiar do you consider you were with this code?

Not at all   Very Familiar
1☐ 2☐ 3☐ 4☐ 5☐ 6☐

b) Having reviewed the code and using any previous experience, what is your opinion of the ease of understanding and maintenance of this code?

Easy to maintain   Difficult to maintain
1☐ 2☐ 3☐ 4☐ 5☐ 6☐

c) Have you ever made changes to this piece of code?

Yes☐ No☐

4. ISSREVS

a) Prior to reviewing the source code for this survey, how familiar do you consider you were with this code?

Not at all Very Familiar

1☐ 2☐ 3☐ 4☐ 5☐ 6☐

b) Having reviewed the code and using any previous experience, what is your opinion of the ease of understanding and maintenance of this code?

Easy to maintain Difficult to maintain

1☐ 2☐ 3☐ 4☐ 5☐ 6☐

c) Have you ever made changes to this piece of code?

Yes☐ No☐

5. MBREXTS

a) Prior to reviewing the source code for this survey, how familiar do you consider you were with this code?

Not at all Very Familiar

1☐ 2☐ 3☐ 4☐ 5☐ 6☐

b) Having reviewed the code and using any previous experience, what is your opinion of the ease of understanding and maintenance of this code?

Easy to maintain Difficult to maintain

1☐ 2☐ 3☐ 4☐ 5☐ 6☐

c) Have you ever made changes to this piece of code?

Yes☐ No☐

6. OMFATDS

a) Prior to reviewing the source code for this survey, how familiar do you consider you were with this code?

Not at all Very Familiar
1☐ 2☐ 3☐ 4☐ 5☐ 6☐

b) Having reviewed the code and using any previous experience, what is your opinion of the ease of understanding and maintenance of this code?

Easy to maintain Difficult to maintain
1☐ 2☐ 3☐ 4☐ 5☐ 6☐

c) Have you ever made changes to this piece of code?

Yes☐ No☐

7. ADVRCNL5

a) Prior to reviewing the source code for this survey, how familiar do you consider you were with this code?

Not at all Very Familiar
1☐ 2☐ 3☐ 4☐ 5☐ 6☐

b) Having reviewed the code and using any previous experience, what is your opinion of the ease of understanding and maintenance of this code?

Easy to maintain Difficult to maintain
1☐ 2☐ 3☐ 4☐ 5☐ 6☐

c) Have you ever made changes to this piece of code?

Yes☐ No☐

8. RETREPS

a) Prior to reviewing the source code for this survey, how familiar do you consider you were with this code?

Not at all Very Familiar
1☐ 2☐ 3☐ 4☐ 5☐ 6☐

b) Having reviewed the code and using any previous experience, what is your opinion of the ease of understanding and maintenance of this code?

Easy to maintain Difficult to maintain
1☐ 2☐ 3☐ 4☐ 5☐ 6☐

c) Have you ever made changes to this piece of code?

Yes☐ No☐

9. TXMATCHS

a) Prior to reviewing the source code for this survey, how familiar do you consider you were with this code?

Not at all Very Familiar
1☐ 2☐ 3☐ 4☐ 5☐ 6☐

b) Having reviewed the code and using any previous experience, what is your opinion of the ease of understanding and maintenance of this code?

Easy to maintain Difficult to maintain
1☐ 2☐ 3☐ 4☐ 5☐ 6☐

c) Have you ever made changes to this piece of code?

Yes☐ No☐

10. VOUTPREP

a) Prior to reviewing the source code for this survey, how familiar do you consider you were with this code?

Not at all Very Familiar

1☐ 2☐ 3☐ 4☐ 5☐ 6☐

b) Having reviewed the code and using any previous experience, what is your opinion of the ease of understanding and maintenance of this code?

Easy to maintain Difficult to maintain

1☐ 2☐ 3☐ 4☐ 5☐ 6☐

c) Have you ever made changes to this piece of code?

Yes☐ No☐

11. NBSCON

a) Prior to reviewing the source code for this survey, how familiar do you consider you were with this code?

Not at all Very Familiar

1☐ 2☐ 3☐ 4☐ 5☐ 6☐

b) Having reviewed the code and using any previous experience, what is your opinion of the ease of understanding and maintenance of this code?

Easy to maintain Difficult to maintain

1☐ 2☐ 3☐ 4☐ 5☐ 6☐

c) Have you ever made changes to this piece of code?

Yes☐ No☐

12. Please rank in order from hardest to maintain to easiest to maintain the modules you have been asked to review (place 1 beside the hardest to maintain, 2 next to the next hardest, continuing through to 9 next to the easiest to maintain.

EAEPROG	__
ISSREVS	__
MBREXTS	__
OMFATDS	__
ADVRCNL5	__
RETREPS	__
TXMATCHS	__
VOUTPREP	__
NBSCON	__

SECTION 2.

This section follows up on the previous section, and again refers to the ease of understanding and maintenance of the eight pieces of COBOL code that you have been asked to review.

You have also been provided with a series of definitions of ‘bad smells’ in COBOL code. Please review these at this point. These ‘smells’ are features of COBOL code, which it is suggested will lead to the code being harder to maintain. Please revisit each of the modules you have previously reviewed, and consider each in turn against the ‘bad smells’ of COBOL which you have been given.

For each piece of code, please answer the following questions having considered the descriptions of the ‘bad smells’. Remember, there are no right or wrong answers to these questions; it is your opinion that counts.

13. EAEPROG

a) In your opinion how much does the of code exhibits signs of each of the smells.

	No Evidence					Significant Evidence
Poor Sizing	1☐	2☐	3☐	4☐	5☐	6☐
Change Preventer	1☐	2☐	3☐	4☐	5☐	6☐
Dispensables	1☐	2☐	3☐	4☐	5☐	6☐
High Coupling	1☐	2☐	3☐	4☐	5☐	6☐
Masking by Comments	1☐	2☐	3☐	4☐	5☐	6☐
Breach of Standards	1☐	2☐	3☐	4☐	5☐	6☐
Aggressive Programming	1☐	2☐	3☐	4☐	5☐	6☐
COBOL Structure Mines	1☐	2☐	3☐	4☐	5☐	6☐

b) Having considered the 'bad smells' has your opinion of this codes maintainability changed compared to your original assessment. In comparison to your original opinion, do you think the code is easier or harder to maintain?

Much Easier	Easier	Slightly Easier	The Same	Slightly Harder	Harder	Much Harder
☐	☐	☐	☐	☐	☐	☐

14. ISSREVS

a) In your opinion how much does the of code exhibits signs of each of the smells.

	No Evidence					Significant Evidence
Poor Sizing	1☐	2☐	3☐	4☐	5☐	6☐
Change Preventer	1☐	2☐	3☐	4☐	5☐	6☐
Dispensables	1☐	2☐	3☐	4☐	5☐	6☐
High Coupling	1☐	2☐	3☐	4☐	5☐	6☐
Masking by Comments	1☐	2☐	3☐	4☐	5☐	6☐
Breach of Standards	1☐	2☐	3☐	4☐	5☐	6☐
Aggressive Programming	1☐	2☐	3☐	4☐	5☐	6☐
COBOL Structure Mines	1☐	2☐	3☐	4☐	5☐	6☐

b) Having considered the 'bad smells' has your opinion of this codes maintainability changed compared to your original assessment. In comparison to your original opinion, do you think the code is easier or harder to maintain?

Much Easier	Easier	Slightly Easier	The Same	Slightly Harder	Harder	Much Harder
☐	☐	☐	☐	☐	☐	☐

15. MBREXTS

a) In your opinion how much does the of code exhibits signs of each of the smells.

	No Evidence					Significant Evidence
Poor Sizing	1☐	2☐	3☐	4☐	5☐	6☐
Change Preventer	1☐	2☐	3☐	4☐	5☐	6☐
Dispensables	1☐	2☐	3☐	4☐	5☐	6☐
High Coupling	1☐	2☐	3☐	4☐	5☐	6☐
Masking by Comments	1☐	2☐	3☐	4☐	5☐	6☐
Breach of Standards	1☐	2☐	3☐	4☐	5☐	6☐
Aggressive Programming	1☐	2☐	3☐	4☐	5☐	6☐
COBOL Structure Mines	1☐	2☐	3☐	4☐	5☐	6☐

b) Having considered the ‘bad smells’ has your opinion of this codes maintainability changed compared to your original assessment. In comparison to your original opinion, do you think the code is easier or harder to maintain?

Much Easier	Easier	Slightly Easier	The Same	Slightly Harder	Harder	Much Harder
☐	☐	☐	☐	☐	☐	☐

16. OMFATDS

a) In your opinion how much does the of code exhibits signs of each of the smells.

	No Evidence					Significant Evidence
Poor Sizing	1☐	2☐	3☐	4☐	5☐	6☐
Change Preventer	1☐	2☐	3☐	4☐	5☐	6☐
Dispensables	1☐	2☐	3☐	4☐	5☐	6☐
High Coupling	1☐	2☐	3☐	4☐	5☐	6☐
Masking by Comments	1☐	2☐	3☐	4☐	5☐	6☐
Breach of Standards	1☐	2☐	3☐	4☐	5☐	6☐
Aggressive Programming	1☐	2☐	3☐	4☐	5☐	6☐
COBOL Structure Mines	1☐	2☐	3☐	4☐	5☐	6☐

b) Having considered the 'bad smells' has your opinion of this codes maintainability changed compared to your original assessment. In comparison to your original opinion, do you think the code is easier or harder to maintain?

Much Easier	Easier	Slightly Easier	The Same	Slightly Harder	Harder	Much Harder
☐	☐	☐	☐	☐	☐	☐

17. ADVRCNL5

a) In your opinion how much does the of code exhibits signs of each of the smells.

	No Evidence					Significant Evidence
Poor Sizing	1☐	2☐	3☐	4☐	5☐	6☐
Change Preventer	1☐	2☐	3☐	4☐	5☐	6☐
Dispensables	1☐	2☐	3☐	4☐	5☐	6☐
High Coupling	1☐	2☐	3☐	4☐	5☐	6☐
Masking by Comments	1☐	2☐	3☐	4☐	5☐	6☐
Breach of Standards	1☐	2☐	3☐	4☐	5☐	6☐
Aggressive Programming	1☐	2☐	3☐	4☐	5☐	6☐
COBOL Structure Mines	1☐	2☐	3☐	4☐	5☐	6☐

b) Having considered the ‘bad smells’ has your opinion of this codes maintainability changed compared to your original assessment. In comparison to your original opinion, do you think the code is easier or harder to maintain?

Much Easier	Easier	Slightly Easier	The Same	Slightly Harder	Harder	Much Harder
☐	☐	☐	☐	☐	☐	☐

18. RETREPS

a) In your opinion how much does the of code exhibits signs of each of the smells.

	No Evidence					Significant Evidence
Poor Sizing	1☐	2☐	3☐	4☐	5☐	6☐
Change Preventer	1☐	2☐	3☐	4☐	5☐	6☐
Dispensables	1☐	2☐	3☐	4☐	5☐	6☐
High Coupling	1☐	2☐	3☐	4☐	5☐	6☐
Masking by Comments	1☐	2☐	3☐	4☐	5☐	6☐
Breach of Standards	1☐	2☐	3☐	4☐	5☐	6☐
Aggressive Programming	1☐	2☐	3☐	4☐	5☐	6☐
COBOL Structure Mines	1☐	2☐	3☐	4☐	5☐	6☐

b) Having considered the ‘bad smells’ has your opinion of this codes maintainability changed compared to your original assessment. In comparison to your original opinion, do you think the code is easier or harder to maintain?

Much Easier	Easier	Slightly Easier	The Same	Slightly Harder	Harder	Much Harder
☐	☐	☐	☐	☐	☐	☐

19. TXMATCHS

a) In your opinion how much does the of code exhibits signs of each of the smells.

	No Evidence					Significant Evidence
Poor Sizing	1☐	2☐	3☐	4☐	5☐	6☐
Change Preventer	1☐	2☐	3☐	4☐	5☐	6☐
Dispensables	1☐	2☐	3☐	4☐	5☐	6☐
High Coupling	1☐	2☐	3☐	4☐	5☐	6☐
Masking by Comments	1☐	2☐	3☐	4☐	5☐	6☐
Breach of Standards	1☐	2☐	3☐	4☐	5☐	6☐
Aggressive Programming	1☐	2☐	3☐	4☐	5☐	6☐
COBOL Structure Mines	1☐	2☐	3☐	4☐	5☐	6☐

b) Having considered the 'bad smells' has your opinion of this codes maintainability changed compared to your original assessment. In comparison to your original opinion, do you think the code is easier or harder to maintain?

Much Easier	Easier	Slightly Easier	The Same	Slightly Harder	Harder	Much Harder
☐	☐	☐	☐	☐	☐	☐

20. VOUTPREP

a) In your opinion how much does the of code exhibits signs of each of the smells.

	No Evidence					Significant Evidence
Poor Sizing	1☐	2☐	3☐	4☐	5☐	6☐
Change Preventer	1☐	2☐	3☐	4☐	5☐	6☐
Dispensables	1☐	2☐	3☐	4☐	5☐	6☐
High Coupling	1☐	2☐	3☐	4☐	5☐	6☐
Masking by Comments	1☐	2☐	3☐	4☐	5☐	6☐
Breach of Standards	1☐	2☐	3☐	4☐	5☐	6☐
Aggressive Programming	1☐	2☐	3☐	4☐	5☐	6☐
COBOL Structure Mines	1☐	2☐	3☐	4☐	5☐	6☐

b) Having considered the 'bad smells' has your opinion of this codes maintainability changed compared to your original assessment. In comparison to your original opinion, do you think the code is easier or harder to maintain?

Much Easier	Easier	Slightly Easier	The Same	Slightly Harder	Harder	Much Harder
☐	☐	☐	☐	☐	☐	☐

21. NBSCON

a) In your opinion how much does the of code exhibits signs of each of the smells.

	No Evidence					Significant Evidence
Poor Sizing	1☐	2☐	3☐	4☐	5☐	6☐
Change Preventer	1☐	2☐	3☐	4☐	5☐	6☐
Dispensables	1☐	2☐	3☐	4☐	5☐	6☐
High Coupling	1☐	2☐	3☐	4☐	5☐	6☐
Masking by Comments	1☐	2☐	3☐	4☐	5☐	6☐
Breach of Standards	1☐	2☐	3☐	4☐	5☐	6☐
Aggressive Programming	1☐	2☐	3☐	4☐	5☐	6☐
COBOL Structure Mines	1☐	2☐	3☐	4☐	5☐	6☐

b) Having considered the ‘bad smells’ has your opinion of this codes maintainability changed compared to your original assessment. In comparison to your original opinion, do you think the code is easier or harder to maintain?

Much Easier	Easier	Slightly Easier	The Same	Slightly Harder	Harder	Much Harder
☐	☐	☐	☐	☐	☐	☐

22. To what extent do you agree that the smell described will make code more difficult to maintain if it is found to exist in COBOL code?

	Strongly Disagree	Disagree	Neither Agree or disagree	Agree	Strongly Agree
Poor Sizing	☐	☐	☐	☐	☐
Change Preventer	☐	☐	☐	☐	☐
Dispensables	☐	☐	☐	☐	☐
High Coupling	☐	☐	☐	☐	☐
Masking by Comments	☐	☐	☐	☐	☐
Breach of Standards	☐	☐	☐	☐	☐
Aggressive Programming	☐	☐	☐	☐	☐
COBOL Structure Mines	☐	☐	☐	☐	☐

Thank you for taking the time to complete this survey – your input is very much appreciated. Please return your survey to Richard Gorman, either by e-mail, or print the completed survey and place it in an envelope if you prefer to remain anonymous.

Appendix C – A summary of the completed surveys

The table below shows a summary of the answers to Questions 1 and 2 of the questionnaire.

Question	**No of respondents for each answer**				
	1	2	3	4	5
How experienced do you consider yourself with Cobol software?	0	2	3	1	0
	Up to 3 years	3 – 7 years	7 years or more		
How many years have you worked with Cobol?	1	1	4		
	1	2	3	4	5
How experienced do you consider yourself in Software Development?	0	1	1	0	4
	Up to 3 years	3 – 7 years	7 years or more		
How many years have you worked in Software Development?	0	2	4		

The table below shows the answers to Questions 3 to 11

Question and Code	Number of respondents for each answer					
EAEPROG	1	2	3	4	5	6
Prior to reviewing the source code for this survey, how familiar do you consider you were with this code?	5	0	0	0	0	1
Having reviewed the code and using any previous experience, what is your opinion of the ease of understanding and maintenance of this code?	0	3	1	0	2	0
	Yes	No				
Have you ever made changes to this piece of code?	0	6				
ISSREVS	1	2	3	4	5	6
Prior to reviewing the source code for this survey, how familiar do you consider you were with this code?	3	2	0	0	0	1
Having reviewed the code and using any previous experience, what is your opinion of the ease of understanding and maintenance of this code?	1	5	0	0	0	0
	Yes	No				
Have you ever made changes to this piece of code?	1	5				
MBREXTS	1	2	3	4	5	6
Prior to reviewing the source code for this survey, how familiar do you consider you were with this code?	4	2	0	0	0	0
Having reviewed the code and using any previous experience, what is your opinion of the ease of understanding and maintenance of this code?	1	3	1	1	0	0
	Yes	No				
Have you ever made changes to this piece of code?	0	6				

OMFATDS	1	2	3	4	5	6
Prior to reviewing the source code for this survey, how familiar do you consider you were with this code?	3	1	0	0	1	1
Having reviewed the code and using any previous experience, what is your opinion of the ease of understanding and maintenance of this code?	0	1	2	1	2	0
	Yes	No				
Have you ever made changes to this piece of code?	2	4				
ADVRCNL5	1	2	3	4	5	6
Prior to reviewing the source code for this survey, how familiar do you consider you were with this code?	5	0	0	0	1	0
Having reviewed the code and using any previous experience, what is your opinion of the ease of understanding and maintenance of this code?	0	1	1	1	2	1
	Yes	No				
Have you ever made changes to this piece of code?	1	5				
RETREPS	1	2	3	4	5	6
Prior to reviewing the source code for this survey, how familiar do you consider you were with this code?	5	0	0	0	1	0
Having reviewed the code and using any previous experience, what is your opinion of the ease of understanding and maintenance of this code?	0	1	1	3	1	0
	Yes	No				
Have you ever made changes to this piece of code?	1	5				

TXMATCHS	1	2	3	4	5	6
Prior to reviewing the source code for this survey, how familiar do you consider you were with this code?	3	1	0	1	1	0
Having reviewed the code and using any previous experience, what is your opinion of the ease of understanding and maintenance of this code?	0	1	3	2	0	0
	Yes	No				
Have you ever made changes to this piece of code?	2	4				
VOUTPREP	1	2	3	4	5	6
Prior to reviewing the source code for this survey, how familiar do you consider you were with this code?	3	1	1	0	1	0
Having reviewed the code and using any previous experience, what is your opinion of the ease of understanding and maintenance of this code?	0	1	2	1	1	1
	Yes	No				
Have you ever made changes to this piece of code?	2	4				
NBSCON	1	2	3	4	5	6
Prior to reviewing the source code for this survey, how familiar do you consider you were with this code?	5	0	1	0	0	0
Having reviewed the code and using any previous experience, what is your opinion of the ease of understanding and maintenance of this code?	1	1	1	1	2	0
	Yes	No				
Have you ever made changes to this piece of code?	1	5				

The following table shows the total number of times each code scored a ranking in Question 12 of the questionnaire.

	Rankings								
	1	2	3	4	5	6	7	8	9
EAEPROG		1		1			3		1
ISSREVS	1				1			1	3
MBREXTS						1	1	3	1
OMFATDS	1		1	1	1	2			
ADVRCNL5	2	2	2						
RETREPS	1	2	1	1		1			
TXMATCHS			1	2	2			1	
VOUTPREP	1		1	1	1	1		1	
NBSCON		1			1	1	2		1

The results for each program in part a of Questions 13 – 21 of the questionnaire are shown in the table below. The number of times a response was given for each smell for each program is shown.

EAEPROG	1	2	3	4	5	6
Poor Sizing	2	3	1	0	0	0
Change Preventer	2	3	1	0	0	0
Dispensables	2	4	0	0	0	0
High Coupling	4	2	0	0	0	0
Masking by Comments	4	2	0	0	0	0
Breach of Standards	0	0	3	2	1	0
Aggressive Programming	0	0	4	1	1	0
Cobol Structure Mines	4	0	2	0	0	0
ISSREVS	1	2	3	4	5	6
Poor Sizing	4	2	0	0	0	0
Change Preventer	2	1	3	0	0	0
Dispensables	5	1	0	0	0	0
High Coupling	1	4	1	0	0	0
Masking by Comments	6	0	0	0	0	0
Breach of Standards	4	1	1	0	0	0
Aggressive Programming	4	2	0	0	0	0
Cobol Structure Mines	6	0	0	0	0	0
MBREXTS	1	2	3	4	5	6
Poor Sizing	2	3	1	0	0	0
Change Preventer	2	3	0	1	0	0
Dispensables	5	0	1	0	0	0
High Coupling	2	3	0	0	1	0
Masking by Comments	5	1	0	0	0	0
Breach of Standards	1	1	2	1	1	0
Aggressive Programming	2	2	1	0	1	0
Cobol Structure Mines	5	1	0	0	0	0

OMFATDS	1	2	3	4	5	6
Poor Sizing	0	1	0	3	2	0
Change Preventer	0	1	3	1	1	0
Dispensables	0	3	2	1	0	0
High Coupling	0	5	1	0	0	0
Masking by Comments	6	0	0	0	0	0
Breach of Standards	1	1	1	1	2	0
Aggressive Programming	0	0	1	3	2	0
Cobol Structure Mines	2	0	0	3	1	0
ADVRCNL5	1	2	3	4	5	6
Poor Sizing	1	0	1	3	1	0
Change Preventer	1	2	2	0	1	0
Dispensables	2	3	0	1	0	0
High Coupling	1	2	2	1	0	0
Masking by Comments	4	1	0	1	0	0
Breach of Standards	1	1	3	0	1	0
Aggressive Programming	3	3	0	0	0	0
Cobol Structure Mines	2	0	0	3	0	1
RETREPS	1	2	3	4	5	6
Poor Sizing	0	3	3	0	0	0
Change Preventer	2	0	1	3	0	0
Dispensables	1	4	1	0	0	0
High Coupling	1	0	1	3	0	0
Masking by Comments	3	3	0	0	0	0
Breach of Standards	1	3	1	0	1	0
Aggressive Programming	5	0	0	1	0	0
Cobol Structure Mines	2	0	0	3	1	0

TXMATCHS	1	2	3	4	5	6
Poor Sizing	0	2	2	1	1	0
Change Preventer	2	0	3	0	0	1
Dispensables	2	2	2	0	0	0
High Coupling	1	2	3	0	0	0
Masking by Comments	3	0	2	1	0	0
Breach of Standards	2	2	1	0	1	0
Aggressive Programming	5	1	0	0	0	0
Cobol Structure Mines	4	2	0	0	0	0

VOUTPREP	1	2	3	4	5	6
Poor Sizing	1	2	2	0	1	0
Change Preventer	1	1	2	1	1	0
Dispensables	3	0	1	1	1	0
High Coupling	1	0	3	2	0	0
Masking by Comments	3	2	0	1	0	0
Breach of Standards	1	1	1	2	1	0
Aggressive Programming	0	2	3	0	1	0
Cobol Structure Mines	4	2	0	0	0	0

NBSCON	1	2	3	4	5	6
Poor Sizing	1	2	2	1	0	0
Change Preventer	1	3	1	1	0	0
Dispensables	3	3	0	0	0	0
High Coupling	1	2	2	1	0	0
Masking by Comments	4	2	0	0	0	0
Breach of Standards	0	0	1	3	2	0
Aggressive Programming	0	2	2	2	0	0
Cobol Structure Mines	2	0	0	3	0	1

The results for each program in part b of Questions 13 – 21 of the questionnaire are shown in the table below. The number of times a response was given for each program is shown.

	Much Easier	Easier	Slightly Easier	The Same	Slightly Harder	Harder	Much Harder
EAEPROG		1	1	2	2		
ISSREVS				6			
MBREXTS			1	5			
OMFATDS				4		2	
ADVRCNL5			2	2	1	1	
RETREPS				3	3		
TXMATCHS			2	3	1		
VOUTPREP			1	3	2		
NBSCON				3	2	1	

The final table summarizes the view of each of the smells as scored in Question 22 of the questionnaire, showing the number of times a response for each of the opinions was received for each smell.

	Strongly Disagree	Disagree	Neither Agree or Disagree	Agree	Strongly Agree
Poor Sizing	0	0	0	5	1
Change Preventer	0	0	1	4	1
Dispensables	0	1	1	3	1
High Coupling	0	0	1	5	0
Masking by Comments	0	2	2	2	0
Breach of Standards	0	0	0	1	5
Aggressive Programming	0	1	1	4	0
Cobol Structure Mines	0	0	0	5	1

References

AFOTEC (1991) ‘Software Maintainability Evaluation Guide’, *AFOTEC Pamphlet 800-2, vol 3,* Kirtland Air Force Base, New Mexico, Air Force Operational and Test Command.

Aggarwal, K.K., Singh, Y. and Chhabra, J.K. (2002) 'An Integrated Measure of Software Maintainability' in *Proceedings of the Annual Reliability and Maintainability Symposium,* IEEE Computer Society Press, pp.235-41.

ANSI (1985) 'Programming Language - COBOL', *ANSI X3.23*.

Ash, D., Alderete, J., Yao, L., Oman, P.W., Lowther, B. (1994) ‘Using software maintainability models to track code health’ in *Proceedings of the International Conference on Software Maintenance,* IEEE Computer Society Press, pp.154-60.

Baker, B.S. (1995) 'On finding duplication and near-duplication in large software systems' in *Proceedings of the Second Working Conference on Reverse Engineering,* IEEE Computer Society Press, pp.86-95.

Banker, R.D., Datar, S.M., Kemerer, C.F. and Zweig, D. (1993) 'Software complexity and maintenance costs', *Communications of the ACM,* vol. 36, no. 11, pp. 81-94.

Bartlett, W. and Spainhower, L. (2004) 'Commercial Fault Tolerance: A Tale of Two Systems.', *IEEE Transactions on Dependable and Secure Computing,* vol. 1, no. 1, pp. 87-96.

Basili, V.R. and Hutchens, D.H. (1983) 'An Empirical Study of a Syntactic Complexity Family.', *IEEE Transactions on Software Engineering,* vol. 9, no. 6, pp. 664-72.

Baxter, I.D., Yahin, A., Moura, L., Sant'Anna, M. and Bier, L. (1998) 'Clone Detection Using Abstract Syntax Trees' in *Proceedings of the International Conference on Software Maintenance,* IEEE Computer Society Press, pp.368-77.

Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R., Mellor, S., Schwaber, K., Sutherland, J. and Thomas, D. Manifesto for Agile Software Development [online], http://agilemanifesto.org/ [Accessed 25 March 2006].

Belady, L.A. and Lehman, M.M. (1976) 'A Model of Large Program Development.', *IBM Systems Journal,* vol. 15, no. 3, pp. 225-52.

Belyaev, E., Shafirov, M. and Oreshnikova, A. (2004) 'Radical Refactoring', *Dr. Dobb's Journal,* vol. 29, no. 1, pp. 26-31.

Bennett, K. (1995) 'Legacy Systems: Coping with Success', *IEEE Software,* vol. 12, no. 1, pp. 19-23.

Boehm, B. (2006) 'A view of 20th and 21st century software engineering' in *Proceedings of the 28th International Conference on Software Engineering,* ACM Press, pp.12-29.

CASEMaker Inc. (2006) *LegacyAid Overview* [online], http://www.casemaker.com/legacyaid/ [Accessed 5 August 2006].

Coleman, D., Brown, D., Lowther, B. and Oman, P. (1994) 'Using Metrics to Evaluate Software System Maintainability', *IEEE Computer,* vol. 27, no. 8, pp. 44-9.

Coleman, D., Lowther, B. and Oman, P. (1995) 'The Application of Software Maintainability Models in Industrial Software Systems', *Journal of Systems Software,* vol. 29, no. 1, pp. 3-16.

Conte, S.D., Dunsmore, H. and Shen, V. (1986) *Software Engineering Metrics and Models,* Menlo Park, California, Benjamin-Cummings.

Dijkstra, E.W. (1968) 'Go To statement considered harmful', *Communications of the ACM,* vol. 11, no. 3, pp. 147-8.

Dijkstra, E.W. (1972) 'Notes on Structured Programming' in O.J. Dahl, E.W. Dijkstra and C.A.R. Hoare (eds) *Structured Programming,* London, Academic Press.

Ducasse, S., Rieger, M. and Demeyer, S. (1999) 'A Language Independent Approach for Detecting Duplicated Code' in *Proceedings of the International Conference on Software Maintenance,* IEEE Computer Society Press, pp.109-18.

Fenton, N.E. and Pfleeger, S.L. (1997) *Software Metrics: A Rigorous and Practical Approach,* Boston, PWS Publishing.

Field, J. and Ramalingam, G. (1999) 'Identifying procedural structure in Cobol programs' in *Proceedings of the 1999 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering,* ACM Press, pp.1-10.

Fink, A. (1995) *How to Analyze Survey Data,* Thousand Oaks, California, Sage.

Fink, A. and Kosecoff, J. (1998) *How to conduct surveys: a step-by-step guide,* Thousand Oaks, California, Sage.

Fowler, M. and Beck, K. (1999) 'Bad Smells in Code' in M. Fowler (ed) *Refactoring - Improving the Design of Existing Code,* Reading, Massachusetts, Addison-Wesley.

GeroneSoft (2006) *GeroneSoft Source Code Counter Pro* [online], http://www.geronesoft.com/ [Accessed 7 August 2006].

Gibson, V.R. and Senn, J.A. (1989) 'System structure and software maintenance performance', *Communications of the ACM,* vol. 32, no. 3, pp. 347-58.

Gill, G.K. and Kemerer, C.F. (1991) 'Cyclomatic Complexity Density and Software Maintenance Productivity', *IEEE Transactions on Software Engineering,* vol. 17, no. 12, pp. 1284-8.

Glass, R.L. and Noiseux, R.A. (1981) *Software Maintenance Guidebook,* Englewood Cliffs, Prentice-Hall.

Halstead, M.H. (1977) *Elements of Software Science,* New York, Elsevier.

Hansen, W.J. (1978) 'Measurement of program complexity by the pair: (Cyclomatic Number, Operator Count)', ACM *SIGPLAN Notices,* vol. 13, no. 3, pp. 29-33.

Harrison, M.S. and Walton, G.H. (2002) 'Identifying high maintenance legacy software', *Journal of Software Maintenance,* vol. 14, no. 6, pp. 429-46.

Henry, S. and Kafura, K. (1981) 'Software Structure Metrics based on Information Flow', *IEEE Transactions on Software Engineering,* vol. 7, no. 5, pp. 510-8.

Hunt, A. and Thomas, D. (1999) *The Pragmatic Programmer: From Journeyman to Master,* Reading, Addison-Wesley.

IEEE (1998) '1219-1998 IEEE Standards on Software Maintenance', *IEEE Software Engineering Standards Collection.*

Johnson, J.H. (1994) 'Substring Matching for Clone Detection and Change Tracking' in *Proceedings of the International Conference on Software Maintenance,* IEEE Computer Society Press, pp.120-6.

Kafura, D. and Reddy, G. (1987) 'The Use of Software Quality Metrics in Software Maintenance', *IEEE Transactions on Software Engineering,* vol. 13, no. 3, pp. 335-43.

Kataoka, Y., Imai, T., Andou, H. and Fukaya, T. (2002) 'A Quantitative Evaluation of Maintainability Enhancement by Refactoring' in *Proceedings of the International Conference on Software Maintenance,* IEEE Computer Society Press, pp.576-85.

Khan, E., Al-A'ali, M. and Girgis, M. (1995) 'Object Oriented Programming for Structural Procedural Programmers', *IEEE Computer,* vol. 28, no. 10, pp. 48-57.

Kitchenham, B.A. and Pfleeger, S.L. (2002) 'Principles of survey research: part 3: constructing a survey instrument', ACM *SIGSOFT Software Engineering Notes,* vol. 27, no. 2, pp. 20-4.

Knoop, J., Ruthing, O. and Steffen, B. (1994) 'Partial dead code elimination' in *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation,* ACM Press, pp.147-58.

Koskinen, J. [Last Updated Sept. 28, 2004], *Software Maintenance Costs* [online], http://www.cs.jyu.fi/~koskinen/smcosts.htm [Accessed March 9 2006].

Land, Rikard. (2002) 'Measurements of Software Maintainability', Unpublished.

Ledgard, H.F. and Cave, W.C. (1976) 'Cobol Under Control.', *Communications of the ACM,* vol. 19, no. 11, pp. 601-8.

Lewis, J. and Henry, S. (1989) 'A Methodology for Integrating Maintainability Using Software Metrics' in *Proceedings of the Conference on Software Maintenance,* IEEE Computer Society Press, pp.32-9.

Lexient Corp. (2006), *Code Analyser by Lexient* [online], http://www.lexientcorp.com/codeanalyzer/products.htm [Accessed 10 August 2006].

Lientz, B.P. (1983) 'Issues in Software Maintenance', *ACM Computer Surveys,* vol. 15, no. 3, pp. 271-8.

Linderman, J.L. (1982) 'Defensive COBOL strategies' in *Proceedings of the thirteenth SIGCSE technical symposium on Computer science education,* ACM Press, pp.205-10.

Maher, B. and Sleeman, D.H. (1983) 'Automatic Program Improvement: Variable Usage Transformations', *ACM Transactions on Programming Language and Systems,* vol. 5, no. 2, pp. 236-64.

Mantyla, M., Vanhanen, J. and Lassenius, C. (2003) 'A taxonomy and an initial empirical study of bad smells in code' in *Proceedings of the International Conference on Software Maintenance,* IEEE Computer Society Press, pp.381-4.

Mantyla, M., Vanhanen, J. and Lassenius, C. (2004) 'Bad Smells - Humans as Code Critics' in *Proceedings of the International Conference on Software Maintenance,* IEEE Computer Society Press, pp.399-408.

Mantyla, M. (2004) ‘Developing New Approaches in Software Design Quality Improvement based on Subjective Evaluations’ in *Proceedings of the 26th International Conference on Software Engineering,* IEEE Computer Society Press, pp.48-50.

Mantyla, M. and Lassenius, C. (2006) ‘Subjective evaluation of software evolvability using code smells: An empirical study’, *Empirical Software Engineering,* vol. 11 no. 3, pp. 395-431.

Martin, J. and McClure, C. (1983) *Software Maintenance: The Problem and Its Solutions,* Englewood Cliffs, New Jersey, Prentice-Hall.

Mathias, K., Cross II, J., Hendrix T. Dean and Barowski, L.A. (1999) 'The role of software measures and metrics in studies of program comprehension' in *Proceedings of the 37th annual Southeast regional conference,* ACM Press, pp.13-9.

McCabe, T.J. (1976) 'A Complexity Measure.', *IEEE Transactions on Software Engineering,* vol. 2, no. 4, pp. 308-20.

Myers, G.J. (1977) 'An extension to the cyclomatic measure of program complexity', ACM *SIGPLAN Notices,* vol. 12, no. 10, pp. 61-4.

Myers, G.J. (1978) 'A controlled experiment in program testing and code walkthroughs/inspections', *Communications of the ACM,* vol. 21, no. 9, pp. 760-8.

Oman, P. and Hagemeister, J. (1992) 'Metrics for Assessing a Software System's Maintainability' in *Proceedings of the Conference on Software Maintenance,* IEEE Computer Society Press, pp.337-44.

Oppenheim, A.N. (1992) *Questionnaire Design, Interviewing and Attitude Measurement,* London, Pinter.

Rea, L.M. and Parker, R.A. (2005) *Designing and Conducting Survey Research - A Comprehensive Guide,* San Francisco, Jossey-Bass.

Rosenberg, J. (1997) 'Problems and Prospects in Quantifying Software Maintainability', *Empirical Software Engineering,* vol. 2, no. 2, pp. 173-7.

Shneiderman, B. (1980) *Software Psychology: Human Factors in Computer and Information Systems,* Cambridge, Massachusetts, Winthrop Publishers.

Simon, F., Steinbruckner, F. and Lewerentz, C. (2001) 'Metrics Based Refactoring' in *Proceeding s of the 5th European Conference on Software Maintenance and Reengineering,* IEEE Computer Society Press, pp.30-8.

Stark, G.E. (1996) 'Measurements for managing software maintenance' in *Proceedings of the 1996 International Conference on Software Maintenance,* IEEE Computer Society Press, pp.152-61.

Stevens, W.P., Myers, G.J. and Constantine, L.L. (1974) 'Structured Design.', *IBM Systems Journal,* vol. 13, no. 2, pp. 115-39.

Swanson, E.B. (1976) 'The dimensions of maintenance' in *Proceedings of the 2nd International Conference on Software Engineering,* IEEE Computer Society Press, pp.492-7.

Van Gelder, A. (1977) 'Structured programming in Cobol: an approach for application programmers', *Communications of the ACM,* vol. 20, no. 1, pp. 2-12.

Veerman, N. and Verhoeven, E. (2006) 'Cobol minefield detection', *Software: Practice and Experience,* vol. 36, no. 14, pp 1605-42.

Vessey, I. and Weber, R. (1983) 'Some factors affecting program repair maintenance: an empirical study', *Communications of the ACM,* vol. 26, no. 2, pp. 128-34.

Wake, S. and Henry, S. (1988) 'A Model Based on Software Quality Factors which Predicts Maintainability' in *Proceedings of the Conference on Software Maintenance,* IEEE Computer Society Press, pp.382-7.

Webopedia [Last Updated August 27, 2001], *Software Entropy* [online], http://www.webopedia.com/TERM/S/software_entropy.html [Accessed March 12 2006].

Welker, K.D., Oman, P.W. and Atkinson, G.G. (1997) 'Development and application of an automated source code maintainability index', *Journal of Software Maintenance,* vol. 9, no. 3, pp. 127-59.

Wikimedia Foundation Inc. [Last Updated 5 June 2006], *Redundant code* [online], http://en.wikipedia.org/wiki/Redundant_code [Accessed 28 June 2006].

Yourdon, E. and Constantine, L.L. (1979) *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design,* Englewood Cliffs, New Jersey, Prentice-Hall, Inc.

Zhuo, F., Lowther, B., Oman, P. and Hagemeister, J. (1993) 'Constructing and Testing Software Maintainability Assessment Models' in *Proceedings of the First International Software Metrics Symposium,* IEEE Computer Society Press, pp.61-70.

Index